
Université de Liège
Faculté des Sciences
Appliquées

# Computational and Theoretical Synergies between Linear Optimization and Supervised Machine Learning

Alejandro Marcos Alvarez

Doctoral thesis

October 5, 2016

# Remerciements

L'achèvement d'une thèse de doctorat est souvent synonyme de soulagement, de fierté, d'excitation et de tristesse. Toutes ces émotions ressenties simultanément. En tout cas, c'est ce que moi je ressens. Ce court texte est destiné à remercier tous ceux qui m'ont aidé, épaulé et gratifié de leur amitié, d'une manière ou d'une autre, au cours de mes années en tant que doctorant.

Tout d'abord, je tiens à sincèrement remercier le Professeur Louis Wehenkel pour m'avoir donné l'inestimable opportunité de réaliser une thèse de doctorat. Son ouverture d'esprit, son enseignement et ses conseils ont largement contribué au succès de ce travail qui aurait été bien moins intéressant sans lui. Je lui suis reconnaissant d'avoir partagé avec moi son savoir incroyable et de m'avoir poussé en dehors de ma zone de confort en m'encourageant à penser toujours mieux et toujours plus loin. Merci aussi pour m'avoir régulièrement aidé, au travers de nos discussions autour de l'Institut, à atteindre les 10000 pas par jour recommandés par les médecins.

Ce travail aurait également été très différent et ne serait certainement jamais arrivé à son terme sans l'aide du Professeur Quentin Louveaux. C'est sous son aile que j'ai découvert le monde mystérieux et fascinant de l'optimisation dans lequel une amélioration d'un $\varepsilon$ déclenche l'hystérie des foules. Je tiens à le remercier sincèrement pour toutes les discussions à propos de tout et de rien que nous avons eues autour d'innombrables cappuccinos et americanos. La communication bidirectionnelle qui a toujours caractérisé nos interactions m'a considérablement enrichi et a joué un rôle majeur dans le développement de mon raisonnement scientifique.

I then want to thank all the members of my jury for accepting to review my work. En particulier, je tiens à remercier les Professeurs Bertrand Cornélusse, Yves Crama, Damien Ernst et Pierre Geurts d'avoir accepté de faire partie de mon jury et pour leurs précieux conseils. I also would like to express my gratitude to Prof. Jesús De Loera for being part of my jury, for the fascinating discussions about polytopes and for welcoming me at UCDavis. Finally, I want to thank Prof. Marco Lübbecke for agreeing to be part of my jury.

I must then thank Makoto Yamada for his friendship, for taking me under his wing, for teaching me so many things about the research world, and for giving me one of the most wonderful opportunities of my life. He should be given significant credit for the researcher I am today as he passed on to me his passion about research as well as his methodology in all its aspects. Thank you very much for the research discussions, for introducing me to the Japanese culture and food, and for helping me master the art of chopstick holding.

Je tiens également à remercier mes collègues Julien Becker, Thibaut Cuvelier, Raphaël Fonteneau, Damien Gerard, Vân Anh Huynh-Thu, Arnaud Joly, Akisato Kimura, Philippe Latour, Francis Maes, Raphaël Marée, Matthieu Philippe, François Schnitzler, Marie Schrynemackers, Koh Takeuchi, François Van Lishout et Diane Zander pour avoir rendu mes années de doctorat si amusantes et si intéressantes. Merci aussi à Samuel d'avoir partagé son bureau, sa machine à café et ses innombrables astuces. Merci à Antoine pour les balades à vélo, le jeu de danse et le projet du cours de vision. Merci à Isabelle pour son esprit cartésien et pour avoir apporté une touche de maths dans notre institut. Merci à Kevin pour son organisation sans faille, ses procès-verbaux rédigés sur des supports improbables et sa bonne humeur

permanente. Et merci à Sébastien pour son amour des sorties, sa passion pour les bières, ses bons plans et son enthousiasme inégalé.

Je tiens enfin à remercier ma famille et mes amis d'avoir été là depuis le début, dans les bons comme dans les mauvais moments, et de me supporter dans tout ce que j'accomplis et dans tout ce que je suis. Merci à mes parents pour m'avoir donné l'opportunité et la chance d'avoir la vie que j'ai et pour m'avoir soutenu dans tous les choix que j'ai faits. A mis abuelos y a mi madrina por su amor, su confianza y su soporte. A Elena pour m'avoir démontré par l'absurde à quel point la science était belle. A mes amis Jean-Michel, Jérémie et Sylvain pour les discussions politiques, les mauvais films, les fous rires et tous les moments passés ensemble.

Finalement, je remercie Morgan pour tout.

# Contents

# Chapter 1

# Introduction

Mathematical optimization is a field of mathematics that is sadly often forgotten when it comes to advertising science to the public. Broadly speaking, optimization consists in minimizing or maximizing a function by assigning appropriate values to its input variables. The search for the optimum of the function, i.e., the minimum or the maximum, is generally further complicated by the fact that the set of allowed input values is restricted by a certain number of constraints. Optimization may not be the most appealing scientific field and can even seem appalling to the uninitiated, but its importance cannot be lessened. Indeed, although it is most of the time hidden in the shadows, optimization is behind almost every important scientific challenge that keeps the researchers busy nowadays. From protein structure prediction (Liwo et al., 1999), through to scheduling (Bertsimas and Tsitsiklis, 1997; Graham et al., 1979), VLSI circuit layout (Cong et al., 1996), and operation of electrical networks (Josz et al., 2015; Gonzalez et al., 2014), optimization is everywhere and plays an increasing role in our society. These applications merely illustrate a few optimization problems in order to emphasize the importance of optimization techniques in present-day science, but it is crucial to understand that the role of optimization is central to most hard scientific problems.

On the other hand, machine learning, which is a subfield of computer science and artificial intelligence, is a research domain that is, unlike optimization, very well publicized. Machine learning techniques indeed play a leading part in the most advertised (and sometimes criticized) advances in computer science. More specifically, machine learning is a discipline that studies algorithms that can learn from data and subsequently use the acquired knowledge in one way or another. Machine learning is nowadays very widely used in applications where a lot of data is available, and where decisions using the available data have to be made either repeatedly, quickly, or very accurately. Examples of such applications include spam filtering (Blanzieri and Bryl, 2008), speech and handwriting recognition (Yu and Deng, 2012; Plamondon and Srihari, 2000), cancer prediction and prognosis (Cruz and Wishart, 2006), and recommendation systems (Ricci et al., 2011). Machine learning techniques are very powerful, but are too often considered as an end in itself, and too rarely seen as a means to solve larger problems. Additionally, and despite their evident successes in many applications and their ever increasing theoretical foundations, machine learning techniques are sometimes

disregarded by some scientific disciplines that are prone to judge the field too immature and too inaccurate to be of any practical utility.

This thesis is at the crossroads between mathematical optimization and machine learning and intends to bridge the gap that still separates them. Indeed, although these fields are usually studied separately, there exist clear relationships between them that easily appear to the keen observer. The clearest relationship is unidirectional: optimization is very often used as a tool to solve machine learning problems. As such, optimization is seen as a component of machine learning approaches. On the contrary, the other relationship, the one in the opposite direction, is too rarely acknowledged and leveraged. In this work, we explicitly study the latter link by using machine learning techniques as a component of optimization algorithms. With this work, we try to strengthen the connections that exist between both fields by showing that what machine learning can bring to optimization is at least as important as what optimization brings to machine learning. In a word, we intend to further close the optimization-machine learning loop and to show that both fields are intrinsically intertwined both practically and theoretically.

In order to illustrate why machine learning can be used to improve optimization algorithms, let us remind that most optimization algorithms have, at some point, to take decisions during the course of the solution procedure. These decisions may be theoretically justified, but may be arbitrary as well. It turns out that the arbitrary decisions taken by an optimization algorithm are very important because they often critically condition the efficiency of the algorithm. However, there exist many optimization problems for which no theoretically efficient algorithm (and, hence, no theoretically good decisions) exists. For those problems for which the optimization algorithm takes arbitrary decisions, taking the good decision at the right time may be a determining factor to finding good solutions in a short amount of time. Unfortunately, finding good decisions is typically hard especially since good decisions vary from problem to problem. With that in mind, we propose to use machine learning techniques to extract, from optimization data, useful information that can be used to identify good decisions in order to provide optimization algorithms with the good decisions at the right moments. The motivation behind this idea is based on the fact that

1. if, at a given moment, a decision is deemed good for a given problem, a similar decision is probably good for a similar problem;

2. machine learning can be used to (i) quantify the goodness of a decision from data, (ii) identify similar situations, and (iii) provide appropriate decisions according to the current situation.

The goal of this thesis is to show how useful data can be generated and leveraged through machine learning techniques in order for the optimization algorithms to take better decisions based on optimization data previously observed in similar situations.

The research direction presented in this work is rather unusual. Indeed, using machine learning techniques within optimization algorithms in order to improve the performance of the solvers is only rarely investigated. Studying all kinds of optimization problems and all optimization algorithms is certainly an unrealistic assignment. We therefore concentrate our attention on linear and mixed-integer linear optimization, and, more particularly, on the simplex and branch-and-bound algorithms.

Note that combining optimization with learning is not very common, but it is not a new idea either. Since the beginning of the 1980s, many techniques leveraging this approach have been developed. Mentioning every relevant piece of work would be hardly feasible here. Instead, let us merely cite applications such as satisfiability problems (Hutter et al., 2006), general mixed-integer optimization (Moll et al., 1998; Boyan and Moore, 2000), planning (Veloso et al., 1995; Xu et al., 2007), and job scheduling (Zhang and Dietterich, 1995) that have greatly benefited from the application of learning methods to solve optimization problems. For additional examples, we refer the reader to the book of Battiti et al. (2008) that is dedicated to the use of learning techniques in optimization algorithms.

This thesis is the result of several years of research and is organized as a collection of chapters that are more or less self-contained. Each chapter can, to some extent, be read independently from the others. The remainder of the document is organized as follows. First, Chapters 2 and 3 review the basic concepts of both mathematical optimization and machine learning that are required to understand the contents of this work. Chapter 4 then presents a first way to use machine learning as a component of the branch-and-bound algorithm by applying machine learning techniques to variable branching. We then present, in Chapter 5, another approach that makes use of machine learning algorithms to efficiently parallelize the branch-and-bound algorithm. Chapter 6 next explores how the theory behind machine learning can be applied to theoretical aspects of a famous linear optimization algorithm, namely the simplex method. Finally, Chapter 7 concludes this manuscript with some closing remarks.

## Contributions of the thesis

Chapter 4 describes the methodology that we set up in order to speed up the branching strategy used within the branch-and-bound algorithm. Machine learning is used here to create a function that can be used as a cheap proxy instead of an existing expensive branching strategy. The chapter describes the developed features, the data generation procedure, the training of the proxy, and the experimental results. Additionally, we propose the method both in a batch setting and in an online setting.

Chapter 5 presents how we leverage machine learning to evaluate the complexity of a MIP problem. The complexity function trained with learning techniques is then applied, as an illustrative example, to a naive parallel branch-and-bound implementation in order to balance the workload between several processors. The chapter details the features developed for this application, the data generation procedure, the training part, as well as the experimental results. We also propose a simple theoretical analysis to illustrate how machine learning can provide additional information about the subproblems to assess upfront the quality of a proposed workload distribution.

Chapter 6 finally explores how the theory behind machine learning, and more particularly the theory of reinforcement learning, can be applied to investigate theoretical questions about the simplex method, an optimization algorithm used to solve linear programming problems. More specifically, we study the question of the polynomiality of the simplex algorithm in a general case. We propose a new methodology that uses reinforcement learning techniques to create and study pivoting rules. The advantage of our approach relies in the fact that the

tools that we use are quite well understood from a theoretical point of view. The theory behind the methods that we apply can thus be used to theoretically analyze the simplex and, more specifically, its complexity. It is to be noted that the study that we propose is merely a first draft and that conclusive final results are still to be found. The contribution of the chapter relies in the way the complexity of the simplex algorithm is studied.

## Other research carried out during the thesis

This manuscript describes the research that I carried out on the topic 'machine learning and mathematical programming'. During my thesis, I also worked on other projects that are outside the scope of the thesis. These are listed hereunder.

1. A. Marcos Alvarez, F. Maes, and L. Wehenkel (2012). Supervised learning to tune simulated annealing for in silico protein structure prediction. In *ESANN 2012 proceedings, 20th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*, pages 49–54 (Marcos Alvarez et al., 2012).

2. A. Kimura, K. Ishiguro, M. Yamada, A. Marcos Alvarez, K. Kataoka, and K. Murasaki (2013). Image context discovery from socially curated contents. In *Proceedings of the 21st ACM international conference on Multimedia*, pages 565–568 (Kimura et al., 2013).

3. A. Marcos Alvarez, M. Yamada, A. Kimura, and T. Iwata (2013). Clustering-based anomaly detection in multi-view data. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 1545–1548 (Marcos Alvarez et al., 2013b).

4. A. Marcos Alvarez, M. Yamada, and A. Kimura (2013). Exploiting socially-generated side information in dimensionality reduction. In *Proceedings of the 2nd international workshop on Socially-Aware Multimedia*, pages 9–12 (Marcos Alvarez et al., 2013a).

5. S. Piérard, A. Marcos Alvarez, A. Lejeune, and M. Van Droogenbroeck (2014). On-the-fly domain adaptation of binary classifiers. In *23rd Belgian-Dutch Conference on Machine Learning (BENELEARN)*, pages 20–28 (Piérard et al., 2014).

# Chapter 2

# Understanding mathematical optimization

As discussed in the introductory chapter, mathematical optimization is a field of mathematics that studies how to assign values to a certain number of variables so as to minimize (or maximize) a function of those variables. Optimization attracts increasing attention of the research community due to its growing importance. This chapter is intended as an introduction to the basic concepts and mechanisms of mathematical optimization used in this thesis.

We first introduce the general concept of an optimization problem. We then specialize the general problem to define linear and mixed-integer linear optimization problems. Additionally, we describe, for each problem type, the most common algorithm used to solve such problems.

## 2.1 Mathematical optimization in general

Generally speaking, a mathematical optimization problem is a problem of the form

$$\begin{aligned} \min \quad & f(\boldsymbol{x}) \\ \text{s.t.} \quad & h_i(\boldsymbol{x}) \leq 0,\ i = 1, \ldots, m. \end{aligned} \tag{2.1}$$

In this formulation, the vector $\boldsymbol{x} = (x_1, \ldots, x_n)^\top \in \mathbb{R}^n$ represents the variables of the problem, the function $f(\cdot)$ represents the objective function that we seek to minimize[1], and the $m$ functions $h_i(\cdot)$ represent $m$ constraints restricting the set of the acceptable values for the variables $\boldsymbol{x}$. This set of acceptable values is typically called the feasible region, or feasible set, disregarding the nature of the constraints. For a given solution $\boldsymbol{x}'$, we say that a constraint $i$ is active (or tight) if the constraint $i$ is satisfied with equality at $\boldsymbol{x}'$, i.e., $h_i(\boldsymbol{x}') = 0$. This formulation is very general and encompasses most optimization problems that one can find in practice. There exist however other problems that cannot be formulated according to Equation (2.1). Such problems are not discussed in this work.

[1]From now on, we focus on minimization problems, but the same analysis could be carried out with maximization problems.

Depending on the form of the functions $f(\cdot)$ and $h_i(\cdot)$, the general formulation can be simplified yielding specialized forms of the problem, which can be grouped in families or classes. Thanks to the restrictions on the forms of the functions in each class, some properties that are common to all problems in the class naturally appear. These properties usually render the problems easier to analyze since supplementary information can be used. Additionally, these properties facilitate the development of algorithms that are tailored to a specific problem class and that exploit the properties of the class. As an example, imagine a class where the functions $f(\cdot)$ and $h_i(\cdot)$ are all linear functions (more precisely, affine functions) of the variables $\boldsymbol{x}$. The problems in this class are then called linear optimization problems. These problems are of practical interest because they happen to be relatively easy to solve due to the numerous properties arising from the linearity of the functions appearing in the problems.

In general, optimization problems are hard to solve, and developing an approach that can cope with all possible optimization problems at once seems, for the moment, out of reach. This is the reason why developing specialized algorithms, tailored to specific problem classes, is a much easier and much more sensible research direction. As a consequence, the focus of mathematical optimization is on studying different classes of optimization problems in order to expose interesting properties, and on developing class-specific algorithms that exploit the properties of the considered class.

Note that a mathematical optimization problem is also often called a mathematical programming problem. The terms optimization and programming are thus equivalent in most situations. In the following of this document, we will use both designations interchangeably.

## 2.2 Linear optimization

Linear optimization is concerned with so-called linear programming (LP) problems, obtained when the objective and the constraint functions are linear functions of the variables. These problems are probably the simplest optimization problems that can be thought of, but, despite their simplicity, they are central to many applications involving optimization. This section introduces their formal definition, as well as one of the main algorithms used to solve them.

### 2.2.1 Problem definition

Linear programming (LP) problems are problems of the form

$$\begin{aligned} \min \quad & \boldsymbol{c}^\top \boldsymbol{x} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} \\ & \boldsymbol{x} \in \mathbb{R}^n, \end{aligned} \tag{2.2}$$

where $\boldsymbol{c} \in \mathbb{R}^n$, $\boldsymbol{A} \in \mathbb{R}^{m \times n}$, and $\boldsymbol{b} \in \mathbb{R}^m$ denote the cost coefficients, the coefficient matrix, and the right-hand side, respectively. The feasible region of LP problems is a polyhedron, referred to as a polytope if it is bounded, obtained through the intersection of $m$ half spaces. The hyperplanes defining the half spaces that bound the polyhedron correspond to the inequality constraints defined by the rows of the coefficient matrix $\boldsymbol{A}$. The intersection of $n$ constraints is called a vertex of the polyhedron.

In general, if the polyhedron is bounded, there exists at least one optimal solution. Several optimal solutions may simultaneously exist if the problem is degenerate. Leaving aside degeneracy and unboundedness, which are not discussed here, the linearity of the constraints and of the objective function of LP problems implies the existence of a unique minimum and implies that any discovered (local) minimum is also the global minimum. That minimum is achieved at the border of the feasible region, and, more specifically, at a vertex of the polyhedron defining the feasible set. The fact that any minimum is a global minimum and its location are important features that contribute to the (relative) simplicity of LP problems. We later explain how these features are leveraged to solve such problems.

Figure 2.1 illustrates a simple 2-variable optimization problem. The red line and the grey area represent the objective function that we seek to minimize and the feasible region of the problem, respectively. The feasible region $P$ is, in this case, a polytope bounded by a certain number of linear constraints. Since there is no degeneracy in the problem, the optimal solution $x^*$ minimizing the objective function corresponds to a vertex of the polytope.

Degeneracy and unboundedness (Vanderbei, 2014) are important aspects of LP problems that need to be addressed in practice. However, for the sake of simplicity, we intentionally omit the discussion of those important issues here, since the discussion of linear programming does not suffer any loss in generality when leaving aside degeneracy and unboundedness (De Loera, 2011). We therefore consider, from now on, that the problems that we study are bounded and non-degenerate.

### 2.2.2 The simplex algorithm

An optimization algorithm tries to find a point in the feasible region of the problem that minimizes the value of the objective function. This definition of optimization algorithms is valid in general. In the specific case of linear programming, we have seen that the optimum of the problem, if it exists, corresponds to a vertex of the polyhedron defining the feasible region. In this section, we present the most popular optimization algorithm for linear programming, namely the simplex method, that exploits the knowledge of the possible locations of the optimum to efficiently solve the problem.

The description of the simplex method given in this section is deliberately general and does not go into particulars. The detailed description of the mechanisms of the simplex algorithm is beyond the scope of this work, and does not contribute significantly to the understanding of our contribution. For details about the simplex method, we refer the interested reader to Bertsimas and Tsitsiklis (1997).

#### Main mechanisms

The simplex method (Dantzig, 1987), originally proposed by Dantzig in 1947, is an optimization algorithm that travels through the vertices of the polyhedron to reach the one vertex that corresponds to the optimal solution. Once the optimal vertex is reached, the optimization terminates since the global minimum has been found. Starting from an arbitrary vertex, the simplex method iteratively selects a neighbor of the current vertex and moves to the selected vertex. This procedure is repeated until the optimal vertex, corresponding to the optimal

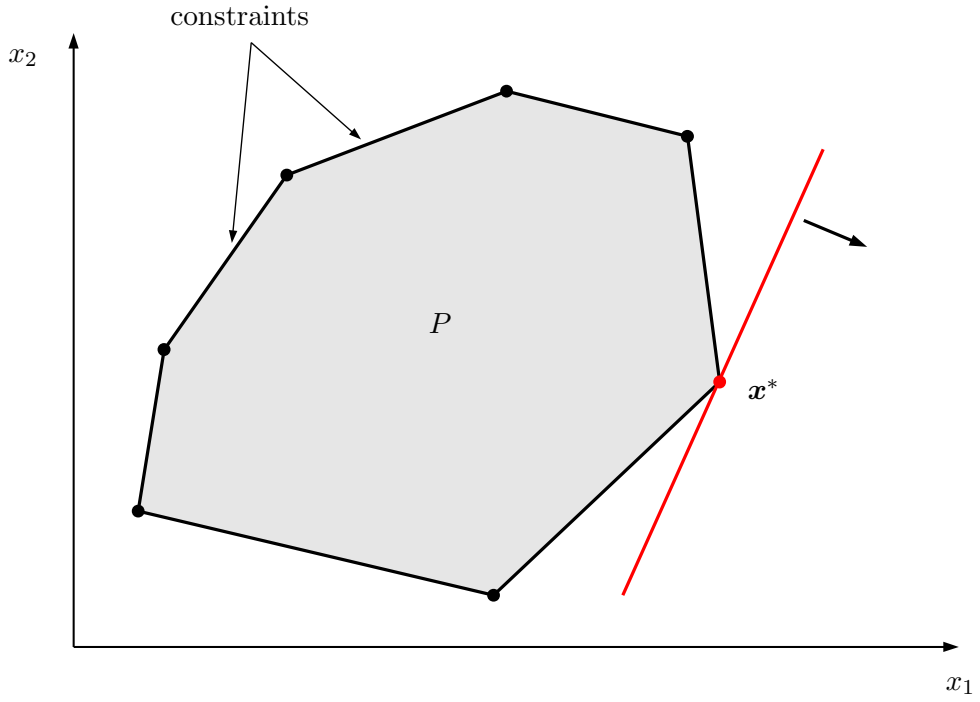



Figure 2.1: Feasible region and objective function of a linear programming problem.

solution, is found. Figure 2.2 illustrates the path followed by the simplex method on a simple tridimensional problem from an arbitrary initial vertex.

The previous description of the simplex method corresponds to the geometric interpretation of the algorithm, which is simpler to understand. The algorithm actually stands on strong mathematical foundations, directly related to the mathematical formulation of LP problems. However, the simplex method does not solve directly problems in the form (2.2). Instead, the simplex solves problems in the so-called *standard form*, which is given by

$$\begin{aligned} \min \quad & \boldsymbol{c}^\top \boldsymbol{x} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \\ & \boldsymbol{x} \in \mathbb{R}^n_+, \end{aligned} \tag{2.3}$$

where $\boldsymbol{c} \in \mathbb{R}^n$, $\boldsymbol{A} \in \mathbb{R}^{m \times n}$, and $\boldsymbol{b} \in \mathbb{R}^m$ denote the cost coefficients, the coefficient matrix, and the right-hand side, respectively. A problem in the form (2.2) can be easily converted to an equivalent formulation in the standard form (2.3), and vice versa. Note, however, that equivalent problems may have different numbers of variables and constraints, and that the parameters $\boldsymbol{c}$, $\boldsymbol{A}$, and $\boldsymbol{b}$ for both formulations will generally be different. From now on, we exclusively work with LP problems given in the standard form (2.3).

The algebraic concept corresponding to vertices of a polyhedron is called a basic feasible solution. A basic feasible solution (BFS) is a solution, i.e., a set of values given to the variables, that (1) satisfies all constraints of the problem, (2) activates all equality constraints, and (3)

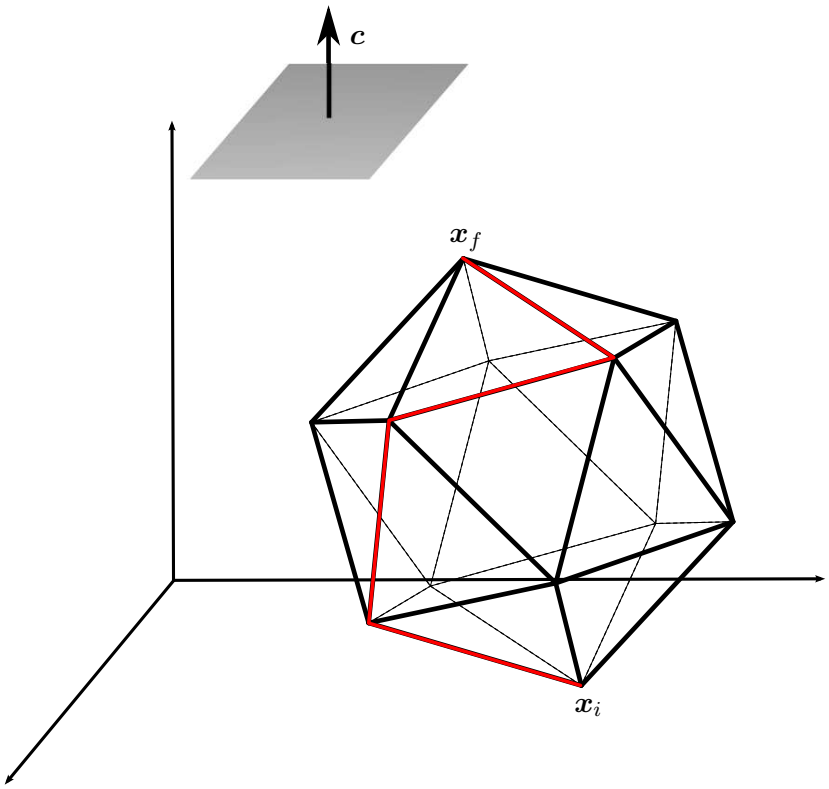



Figure 2.2: Illustration of the simplex method on a simple tridimensional polytope. The starting point of the method is the vertex $\boldsymbol{x}_i$. The method then iteratively moves from one vertex to one of its neighbors until the optimal vertex, in this case $\boldsymbol{x}_f$, is reached. The algorithm then terminates since the optimal solution has been found. The vector $\boldsymbol{c}$ represents the objective function that is to be minimized.

activates other (non-equality) constraints such that exactly $n$ of the active constraints are linearly independent. Because the problems are assumed to be in the standard form, a basic feasible solution implies that $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ is always satisfied, and that $n - m$ variables are equal to 0 (condition (3) of the definition of a BFS). In other words, a solution that can be qualified as a BFS corresponds to a vertex of the polyhedron of the problem. Two BFS are said to be adjacent if they share $n - 1$ linearly independent active constraints. Naturally, two adjacent BFS correspond to two adjacent vertices, in such a way that the simplex algorithm sums up to traveling through adjacent BFS.

Given that there are $n-m$ variables that are 0 in a BFS, the remaining (non-zero) variables are fixed by solving the equation $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$. The $n$ variables in the problem are thus separated in two sets: $m$ basic variables that have a non-zero value, and $n - m$ non-basic variables that are 0. By definition, the set $B$, called the 'basis', contains the indices of the basic variables, and the matrix $\boldsymbol{B}$, called a basis matrix, corresponds to the $m$ columns of $\boldsymbol{A}$ for the basic variables. Since the non-basic variables are 0, the values of the basic variables are obtained by

$$\boldsymbol{x}_B = \boldsymbol{B}^{-1}\boldsymbol{b}, \tag{2.4}$$

where $\boldsymbol{x}_B$ is the solution vector for the basic variables.

The normal operation of the simplex algorithm involves jumping from one BFS to an adjacent one. This can be easily achieved by changing only one active constraint in the current BFS. In order to do that, one must first choose a so-called entering variable, i.e., a non-basic variable that enters the basis, but also a leaving variable, i.e., a basic variable that becomes non-basic. A pivoting rule is a rule according to which the entering and leaving variables are chosen (in general, the effect of a pivoting rule is to move to a better BFS, i.e., a BFS with lower objective value).

During a pivot, when the chosen variable enters the basis, the corresponding active constraint is deactivated. In order for the new solution to remain a BFS, a new constraint must become active and a basic variable $x_i$ with $i \in B$ must thus leave the basis, i.e., $x_i$ becomes 0 and activates the corresponding constraint $x_i \geq 0$. Which variable $x_i$ leaves the basis is decided based on the following formulae

$$\begin{aligned} \boldsymbol{d}_B &= -\boldsymbol{B}^{-1}\boldsymbol{A}_{:j}\,, \\ i &= \underset{i \in B | d_{B(i)} < 0}{\operatorname{argmin}} -\frac{x_{B(i)}}{d_{B(i)}}\,, \end{aligned}$$

where $j$ and $\boldsymbol{A}_{:j}$ are the index of the entering variable and the corresponding column in matrix $\boldsymbol{A}$, respectively. The vector $\boldsymbol{d}_B$ represents the direction that one follows in order to make variable $j$ enter the basis. The minimum is taken to ensure that the new solution obtained when following direction $\boldsymbol{d}_B$ remains feasible and allows to identify which variable is the first to leave the basis when direction $\boldsymbol{d}_B$ is followed.

When the entering and leaving variables are identified, the basis $B$ is updated accordingly and the procedure is repeated. Starting from an initial basic feasible solution, the simplex algorithm uses the above formulae to travel through adjacent BFS until an optimal solution is found. Knowing whether a solution is optimal or not is easy with the concept of reduced costs. The reduced cost $\overline{c}_i$ for variable $x_i$ is an updated version of the initial cost $c_i$ that is computed as follows

$$\overline{c}_i = c_i - \boldsymbol{c}_B^\top \boldsymbol{B}^{-1}\boldsymbol{A}_{:i}\,,$$

where $\boldsymbol{c}_B$ corresponds to the initial cost of the current basic variables. An optimal solution is found when all the reduced costs are non-negative. Intuitively, the reduced cost $\overline{c}_i$ represents the increase of the value of the objective function per unit increase of the variable $x_i$. Clearly, since we are solving a minimization problem, only the negative reduced costs are promising, since a positive $\overline{c}_i$ implies increasing the objective function. Consequently, the non-negativity of the reduced costs associated with a given basic feasible solution implies the optimality of the considered solution. This criterion is used to terminate the simplex algorithm.

### The simplex tableau

The previous equations define how the simplex works from a purely algebraic point of view. However, for a better understanding, it is interesting to consider the so-called tableau form of the simplex algorithm that translates the previous equations into a much more tangible form.

Let us assume that $\boldsymbol{x} = [\boldsymbol{x}_B; \boldsymbol{x}_N]$ is a basic feasible solution and let $\boldsymbol{x}_B$ and $\boldsymbol{x}_N$ denote the basic and non-basic variables, respectively. The equality constraints $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ can be rewritten as

$$\boldsymbol{B}\boldsymbol{x}_B + \boldsymbol{N}\boldsymbol{x}_N = \boldsymbol{b},$$

where $\boldsymbol{A} = [\boldsymbol{B}; \boldsymbol{N}]$ (without loss of generality, it is assumed that the basic variables have the indices $\{1, \ldots, m\}$, while the non-basic variables correspond to the indices $\{m+1, \ldots, n\}$). Left multiplying both sides by $\boldsymbol{B}^{-1}$, the equation becomes

$$\mathbb{I}_m \boldsymbol{x}_B + \boldsymbol{B}^{-1}\boldsymbol{N}\boldsymbol{x}_N = \boldsymbol{B}^{-1}\boldsymbol{b}, \tag{2.5}$$

where $\mathbb{I}_m$ is the identity matrix of size $m$. Since the non-basic variables are assumed to be null, i.e., $\boldsymbol{x}_N = \boldsymbol{0}$, this equation is equivalent to Equation (2.4) previously introduced. Interestingly, Equation (2.5) can be rewritten as

$$\overline{\boldsymbol{A}}\boldsymbol{x} = \overline{\boldsymbol{b}},$$

with $\overline{\boldsymbol{A}} = \left[\mathbb{I}_m; \boldsymbol{B}^{-1}\boldsymbol{N}\right]$ and $\overline{\boldsymbol{b}} = \boldsymbol{B}^{-1}\boldsymbol{b}$, which indicates that, for a basic feasible solution, the initial system of equations can be written as an equivalent system where the sub-matrix of $\overline{\boldsymbol{A}}$ corresponding to the basic variables is equal to the identity matrix. In particular, the identity matrix (together with the fact that $\boldsymbol{x}_N = \boldsymbol{0}$) allows to easily find the values of the basic variables (the solution for the $i$th basic variable is equal to the $i$th component of $\overline{\boldsymbol{b}}$).

The converse is true as well. Indeed, if a system of equations $\overline{\boldsymbol{A}}\boldsymbol{x} = \overline{\boldsymbol{b}}$ is such that $m$ columns of the matrix $\overline{\boldsymbol{A}}$ represent an identity matrix of size $m$, then the variables corresponding to the $m$ columns forming the identity matrix are basic variables (the others being non-basic). In particular, this implies that in order to find a BFS, it suffices to transform the initial system of equations into an equivalent system where $m$ columns represent an identity matrix (one must also ensure that all the components of $\overline{\boldsymbol{b}}$ are non-negative). Note that the transformation must be done in such a way that the solutions to both systems are the same, i.e., the transformation is done with operations on the rows.

We can now define the so-called simplex tableau that is obtained by concatenating the matrices $\overline{\boldsymbol{A}}$, $\overline{\boldsymbol{b}}$, and $\overline{\boldsymbol{c}}$ in a single matrix, called the tableau,

$$\boldsymbol{T} = \begin{bmatrix} \overline{\boldsymbol{A}} & \overline{\boldsymbol{b}} \\ \overline{\boldsymbol{c}}^\top & \times \end{bmatrix},$$

where $\overline{\boldsymbol{c}}$ denotes the reduced costs and $\times$ indicates the absence of a value ($\overline{\boldsymbol{c}}$ has only $n$ values while the concatenation of $\overline{\boldsymbol{A}}$ and $\overline{\boldsymbol{b}}$ has $n+1$ columns). During the course of the algorithm, the tableau evolves, through row operations applied to $\boldsymbol{T}$, to explore successive BFS until the optimal BFS is found.

Indeed, the normal operations of the simplex, which involve jumping from one BFS to an adjacent one (i.e., pivoting), can be easily translated in terms of the system of equations. Actually, since pivoting merely consists in exchanging the roles of a basic and a non-basic variable, pivoting the system $\overline{\boldsymbol{A}}\boldsymbol{x} = \overline{\boldsymbol{b}}$ simply implies transforming the current system into a new system where the column corresponding to the entering variable becomes one column of the identity matrix. The position of the 1 in the column of the entering variable is determined by the leaving variable to ensure that $\overline{\boldsymbol{A}}$ still contains an identity matrix as sub-matrix after

the pivot. In mathematical terms, once the entering variable is chosen, say $i$, the leaving variable $j$ is such that

$$\begin{aligned}\overline{A}_{lj} &= 1,\\ \overline{A}_{kj} &= 0,\ \forall k \neq l,\\ l &= \underset{k=1,\dots,m|\theta_k>0}{\operatorname{argmin}} \theta_k,\end{aligned}$$

with

$$\theta = \left[\frac{\overline{b}_1}{\overline{A}_{1i}};\ \frac{\overline{b}_2}{\overline{A}_{2i}};\ \dots;\frac{\overline{b}_m}{\overline{A}_{mi}}\right].$$

When the entering and leaving variables are identified, $i$ and $j$ (and $l$ the row corresponding to variable $j$ in the identity matrix), respectively, the system of equations can be transformed into an equivalent system representing a new BFS. This update can be done easily with the following operations

$$\begin{cases} T'_{1:} & = T_{1:} - \frac{T_{1i}}{T_{li}} T_{l:} \\ \vdots & \\ T'_{l:} & = \frac{1}{T_{li}} T_{l:} \\ \vdots & \\ T'_{m:} & = T_{m:} - \frac{T_{mi}}{T_{li}} T_{l:} \\ T'_{(m+1):} & = T_{(m+1):} - \frac{T_{(m+1)i}}{T_{li}} T_{l:} \end{cases}$$

where $T$ and $T'$ denote the simplex tableau before and after pivoting, respectively.

Note that, when the tableau is updated to reflect a new BFS, the reduced costs are updated as well. Indeed, checking whether a BFS is optimal is done by checking that all reduced costs are non-negative and updating them is thus important in order to determine whether the algorithm must continue pivoting or not. In vector form, the reduced costs are given by

$$\overline{\boldsymbol{c}}^\top = \boldsymbol{c}^\top - \boldsymbol{c}_B^\top \boldsymbol{B}^{-1}\boldsymbol{A}$$

and are updated at the same time as the rest of the tableau, i.e., during pivoting. Note that, for a given BFS, the reduced costs corresponding to the basic variables are equal to 0. Therefore, when the BFS changes, i.e., after a pivot, the reduced costs must be updated as well to ensure that the reduced costs of the new basic variables are equal to 0. This update is done in the same way as for the rest of the tableau, i.e., with row operations.

### An example

We now illustrate the theoretical description of the simplex algorithm hereabove given. The considered problem is as follows:

$$\begin{array}{ll} \min & -4x_1 - 2x_2 - 3x_3 \\ \text{s.t.} & \left\{ \begin{array}{rcrcrcrcrcrcl} 2x_1 & + & x_2 & + & x_3 & + & x_4 & & & & & = & 2 \\ 2x_1 & + & 2x_2 & + & 3x_3 & & & + & x_5 & & & = & 5 \\ x_1 & + & 2x_2 & + & x_3 & & & & & + & x_6 & = & 6 \end{array} \right. \\ & x_i \geq 0. \end{array}$$

First, the simplex tableau is initialized with the initial matrices $\boldsymbol{A}$, $\boldsymbol{b}$, and $\boldsymbol{c}$.

| | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $\overline{\boldsymbol{b}}$ |
|---|---|---|---|---|---|---|---|
| | 2 | 1 | 1 | 1 | 0 | 0 | 2 |
| | 2 | 2 | 3 | 0 | 1 | 0 | 5 |
| | 1 | 2 | 1 | 0 | 0 | 1 | 6 |
| $\overline{\boldsymbol{c}}$ | −4 | −2 | −3 | 0 | 0 | 0 | |

Then, a variable that enters the basis must be chosen (according to a pivoting rule). Here, the variable $x_1$ is chosen as entering variable because it has the smallest reduced cost. We now compute the ratio vector to identify the leaving variable. The ratios are $\theta = [1;\ 2.5;\ 6]$ and the smallest ratio corresponds to the first component of the vector, which, in turn, indicates that variable $x_4$ must leave the basis when variable $x_1$ enters. As explained above, in order to make a variable enter the basis, one can modify the problem with row operations and make zeros appear in the column corresponding to the entering variable with a unique 1 appearing at the position corresponding to the leaving variable. Since the leaving variable is $x_4$ and since the 1 in the $x_4$ column was located in the first row, we modify the problem to make zeros appear everywhere in the column of $x_1$ except for the first element of the column that will contain a 1. Applying the following row operations,

$$\left\{ \begin{array}{ll} r_1' & = \frac{1}{2} r_1 \\ r_2' & = r_2 - \frac{2}{2} r_1 \\ r_3' & = r_3 - \frac{1}{2} r_1 \\ \overline{c}' & = \overline{c} - \frac{-4}{2} r_1 \end{array} \right.$$

the tableau becomes

| | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $\overline{\boldsymbol{b}}$ |
|---|---|---|---|---|---|---|---|
| | 1 | 1/2 | 1/2 | 1/2 | 0 | 0 | 1 |
| | 0 | 1 | 2 | −1 | 1 | 0 | 3 |
| | 0 | 3/2 | 1/2 | −1/2 | 0 | 1 | 5 |
| $\overline{\boldsymbol{c}}$ | 0 | 0 | −1 | 2 | 0 | 0 | |

Following the same pivoting rule (i.e., the variable with the smallest reduced cost enters the basis), variable $x_3$ is chosen as entering variable. Computing the ratios, we obtain $\theta = [2;\ 1.5;\ 10]$. The smallest ratio indicates that variable $x_5$ must leave the basis. After the

adequate row operations, the tableau becomes:

| | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $\overline{\boldsymbol{b}}$ |
|---|---|---|---|---|---|---|---|
| | 1 | 1/4 | 0 | 3/4 | −1/4 | 0 | 1/4 |
| | 0 | 1/2 | 1 | −1/2 | 1/2 | 0 | 3/2 |
| | 0 | 5/4 | 0 | −1/4 | −1/4 | 1 | 17/4 |
| $\overline{\boldsymbol{c}}$ | 0 | 1/2 | 0 | 3/2 | 1/2 | 0 | |

which is optimal since all reduced costs are non-negative (implying that making a variable enter the basis will increase the objective value).

In this last tableau, variables $x_2$, $x_4$, and $x_5$ are non-basic (and thus equal to 0 in the solution). The remaining variables ($x_1$, $x_3$, and $x_6$) are basic and their value in the current solution is given by the current right hand side. In the end, the optimal solution is

$$\boldsymbol{x}^* = [1/4;\ 0;\ 3/2;\ 0;\ 0;\ 17/4]\,.$$

### A few pitfalls of the method

The description that we give of the simplex algorithm merely explains its main mechanisms. There exist many stumbling blocks that can prevent the algorithm from running correctly. In the following, we present two problems that may affect the proper operation of the method and briefly outline how these issues can be addressed. More specifically, we describe the issue of finding an initial basic feasible solution and the problem of identifying, at each iteration, the entering variable.

**Initialization of the simplex method** In general, jumping from one basic feasible solution (vertex) to another is easy, both geometrically and algebraically, but finding an initial BFS can be difficult in some situations. If a BFS does not emerge naturally from the problem formulation, it is necessary to use some tricks in order to find the starting BFS of the simplex. One of the most common approaches to solve that problem is called the big-$M$ method. This method consists in introducing $m$ artificial variables, and in using the simplex method to solve the initial problem augmented with these variables. The new problem is given by

$$\begin{aligned} \min \quad & \boldsymbol{c}^\top \boldsymbol{x} + M\mathbf{1}^\top \boldsymbol{y} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} + \boldsymbol{y} = \boldsymbol{b} \\ & \boldsymbol{x} \in \mathbb{R}^n_+ \\ & \boldsymbol{y} \in \mathbb{R}^m_+, \end{aligned}$$

where $\boldsymbol{y}$ and $\mathbf{1}$ represent the artificial variables and a vector of ones, respectively. The parameter $M$ is a scalar that is chosen to be 'big' enough, so that the solution to the new problem leads to $\boldsymbol{y} = 0$. Since each $y_j$ appears in only one constraint, the artificial variables constitute a basis of the modified problem. The simplex is then applied to the new problem with the set of artificial variables as an initial basis. During the iterations of the algorithm, the artificial variables are removed from the initial basis due to the large value of the parameter $M$. At the end of the optimization, if the initial problem admits a basic feasible solution, the optimal basis of the modified problem does not contain any artificial variables anymore, and

that basis is also a basis of the initial problem. By solving a modified problem, the big-$M$ method finds a basic feasible solution to the initial problem, if such a solution exists. The discovered basic feasible solution is then used to initialize the simplex method for the initial problem. Besides the big-$M$ method, a similar technique, called the two-phase method, can be used to find an initial BFS. We omit the description of that procedure here, and refer the reader to Bertsimas and Tsitsiklis (1997) for a thorough explanation of both approaches.

**Pivoting rule** Using the simplex in practice rises one important question: how must the entering basic variable be chosen? Answering that question is not a trivial matter since the answer conditions the efficiency of the algorithm. Leaving aside the study of the complexity of the simplex (which is addressed later), we discuss here the more prosaic cycling problem. Indeed, in some circumstances, if no care is taken when choosing the entering basic variable, the resulting new BFS may be one that has been previously visited. This phenomenon is known as cycling since the algorithm returns, after a while, to a vertex that it already processed. Due to the waste of computational resources it implies, this behavior is naturally undesirable. The literature lists many different 'pivoting rules' that are used to choose the entering basic variable. Among them, the simplest non-cycling rule, known as the Bland's rule, chooses the entering variable $x_i$ as one with negative reduced cost and, if ties occur when deciding which variable leaves the basis, chooses the one with the smallest index $i$. This rule ensures that the objective function decreases with each new visited BFS and is proved to never cycle (Bertsimas and Tsitsiklis, 1997). In general, when a pivoting rule chooses an entering variable with negative reduced cost and if the polytope is non-degenerate, cycling never happens. However, since degeneracy is encountered quite often in practice, it is possible that, in some rare cases, cycling occurs. Fortunately, there are several ways to deal with cycling issues in practice, including anticycling pivoting rules and random perturbations, but describing these elements is beyond the scope of this work.

## 2.3 Mixed-integer linear optimization

Mixed-integer linear programming (MILP or, more simply, MIP) problems are obtained when some variables of an LP problem are constrained to integer values only. These additional constraints may seem harmless, but actually render the problems much more difficult (Vanderbei, 2014). Nonetheless, these problems are of practical interest because the integer variables model behaviors that are out of reach of continuous variables. The integer variables are indeed used when it is not possible to use continuous variables. For instance, if a variable represents the number of units of a given product that are manufactured, sold, or purchased, using continuous variables makes no sense since the units of product are usually indivisible (it is not possible to build or buy 0.5 plane or 0.5 car). Additionally, binary integer variables (that can take the values 0 or 1 only) are often used to model choices or decisions. For instance, a binary variable may model whether a power plant should be turned on or off depending on the electricity market price (it is not possible to turn on half of the plant only).

In this section, we formally define MIP problems and describe the main mechanisms of the branch-and-bound, which is the most popular algorithm used to solve these problems.

### 2.3.1 Problem definition

Mixed-integer programming (MIP) problems are problems of the form

$$\begin{aligned} \min \quad & \boldsymbol{c}^\top \boldsymbol{x} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} \\ & x_j \in \mathbb{Z}\,,\ \forall j \in I \\ & x_j \in \mathbb{R}\,,\ \forall j \in C\,, \end{aligned} \tag{2.6}$$

where $\boldsymbol{c} \in \mathbb{R}^n$, $\boldsymbol{A} \in \mathbb{R}^{m \times n}$, and $\boldsymbol{b} \in \mathbb{R}^m$ denote the cost coefficients, the coefficient matrix, and the right-hand side, respectively. $I$ and $C$ are two sets containing the indices of the integer and continuous variables, respectively.

Figure 2.3 illustrates the same problem as in Figure 2.1, with the main difference that both variables can now only take integer values. The feasible region becomes a collection of feasible points, which turn out to be all feasible solutions. The feasible points are represented as grey points in the figure. The optimal solution may not correspond to a vertex of the polyhedron (defined by the constraints $\boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}$) anymore, and the simplex algorithm cannot be applied as is. One possible approach to solve such problems could be to list all possible solutions satisfying all constraints, and returning the one with the smallest objective function value as the optimum of the problem.

#### In this work

For the sake of generality and easiness, we introduce the MIP problems and the branch-and-bound algorithm in the general case, i.e., for the MIP problems formulated in the form of Equation (2.6). In this work, we focus, however, more specifically on binary MIP problems, where the integer variables can only take values 0 or 1. These problems are of the form

$$\begin{aligned} \min \quad & \boldsymbol{c}^\top \boldsymbol{x} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} \\ & x_j \in \{0,1\}\ ,\ \forall j \in I \\ & x_j \in \mathbb{R}\,,\ \forall j \in C\,. \end{aligned} \tag{2.7}$$

Throughout this work, we focus solely on binary MIP problems. There are essentially three reasons for this. First, despite the proximity between general MIPs and binary MIPs, the analysis is easier to carry out on binary problems. For instance, the optimization tree has a much clearer interpretation in the binary case than in the general case (going down one level in the tree means setting one variable to 0 or 1 in the binary case, while this is not true for general MIPs). Second, most MIP problems usually involve only binary integer and continuous variables. Indeed, in 80% of the problems of the MIPLIB2010 (Koch et al., 2011), which is a library of benchmark MIP problems obtained from the industry, all integer variables are binary variables. The remaining 20% may contain general integer, binary integer and continuous variables. The ratio of general MIPs is thus quite low. Finally, it is often possible to formulate general MIP problems in the binary form. These elements justify the focus on binary MIP problems throughout this research.

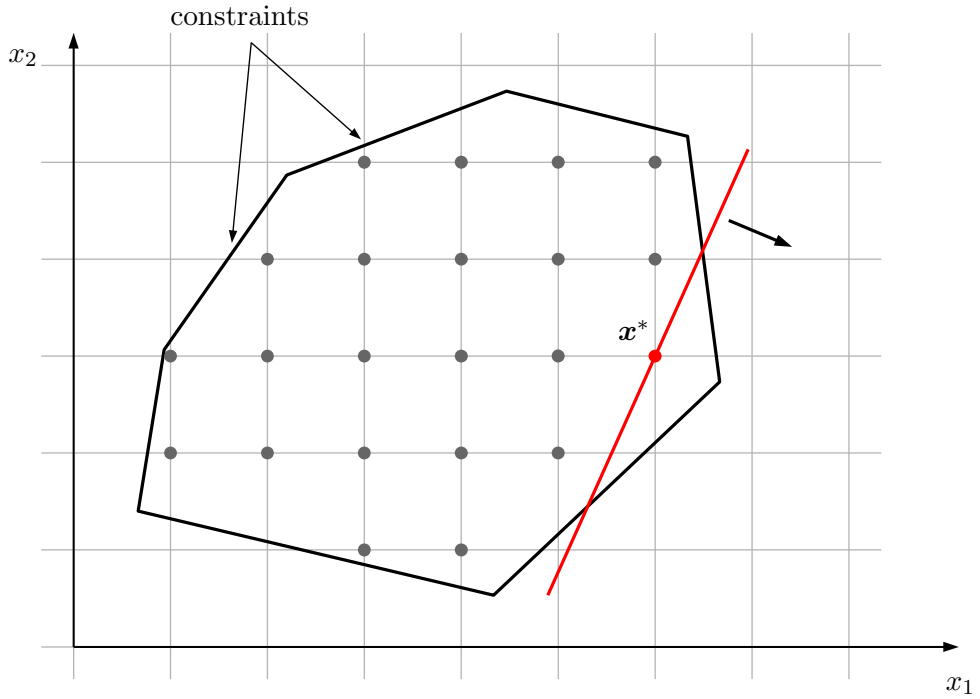



Figure 2.3: Feasible region and objective function of a mixed-integer linear programming problem. In this case, the feasible region is not a polyhedron anymore. The feasible region is composed of the points that lie inside the polyhedron defined by the constraints and for which variables $x_1$ and $x_2$ have an integer value.

### 2.3.2 The branch-and-bound algorithm

The branch-and-bound (B&B) algorithm (Land and Doig, 1960) is probably the most widely used algorithm to solve MIP problems in practice. B&B is an exact method in the sense that, when the method terminates, the solution is guaranteed to be optimal. Of course, since the MIP problems are very difficult, B&B may take a very long time before terminating. In that case, it is still possible to stop the optimization earlier and to obtain a feasible (integer) solution, but the optimality of the solution is not guaranteed anymore. In the following, we concisely explain the main mechanisms of B&B in the case of general MIP problems. The description that we give is valid for a minimization problem, but a similar reasoning applies when the problem is a maximization problem.

Solving problems in the form (2.6) is rather difficult. It is interesting to note that, on the other hand, the very similar LP-problems of the form (2.2) can be solved much more efficiently with simple methods. The branch-and-bound algorithm leverages this fact by turning the solution of a MIP problem into the solution of successive (simpler) LP problems. More specifically, B&B builds an optimization tree in which each node $s$ represents a version of the initial optimization problem where

1. the integrality constraints of the variables in $I$ are relaxed, i.e., $x_i \in \mathbb{R}$ for $i \in I$, in place of $x_i \in \mathbb{Z}$;
2. additional constraints of the form $x_i \leq v$ (or $x_i \geq v$) are added, where $v$ is some scalar (see later what $v$ represents).

Because the integrality constraints are relaxed, the problem contained in each node is called a linear programming relaxation (LP-relaxation) and can be solved efficiently with a traditional linear programming method, e.g., the simplex algorithm. Another advantage of the linear relaxation is that it gives a lower bound on the optimal objective value (this is discussed later in greater detail). We denote the solution of the LP-relaxation at a given node $s$ of the B&B by $\boldsymbol{x}'(s)$ (written $\boldsymbol{x}'$ for short if it is clear from the context which node it corresponds to), and we call, with a little abuse of terminology, the variable $x_i$, with $i \in I$, a fractional variable if it has a fractional value in the current solution $\boldsymbol{x}'$. The set of fractional variables of $\boldsymbol{x}'(s)$ is denoted $F(s)$, i.e.,

$$F(s) = \left\{ i \,\middle|\, i \in I \wedge x'_i(s) \notin \mathbb{Z} \right\}.$$

The value $\boldsymbol{c}^\top \boldsymbol{x}'(s)$ is called the objective value of the solution $\boldsymbol{x}'(s)$.

We now turn to the description of B&B, whose pseudocode is given in Algorithm 1.

As mentioned earlier, B&B creates an optimization tree. That tree is created iteratively and, at every moment, the tree represents the current state of the ongoing optimization. The tree is composed of open nodes, i.e., the nodes that are yet to be explored, which are stored in a queue, and of closed nodes, which are the nodes already processed by the algorithm. An iteration of the B&B consists in three steps: (1) choose an open node $s$ from the tree, (2) solve the LP-relaxation contained at the chosen node, and (3), based on the solution of the LP-relaxation, decide what to do next. Step (1) conditions how the optimization tree is explored, e.g., breadth-first or depth-first. Step (2) is trivial since it merely consists in applying an LP algorithm to the LP-relaxation corresponding to the chosen node. Finally, step (3) is the heart of B&B and its main mechanisms deserve a more detailed description. Step (3) can actually be divided in three different cases.

1. The first possible case is encountered either when the LP-relaxation at the chosen node is infeasible or when the objective value of the LP-relaxation at the chosen node is greater than the objective value of the best integral solution found thus far. In that case, B&B does not continue the exploration of that branch of the tree since that branch cannot yield a better integral solution. The operation that allows to discard the nodes by comparing the objective values of the LP-relaxations with the objective value of the best available integral solution is called 'bounding'.
2. The second possible case encountered during step (3) corresponds to a node whose LP solution respects all initial integrality constraints, i.e., the set $F$ is empty at that node. In that case, the algorithm has found an integral solution (not necessarily optimal) of the problem and the exploration of that branch of the tree is stopped (the node is marked as closed and no child nodes are created).
3. Third, if the solution of the LP-relaxation at node $s$, i.e., the solution $\boldsymbol{x}'$ of the LP-relaxation obtained through step (2), violates at least one of the initial integrality constraints, i.e., if the set $F(s)$ is non empty, the algorithm creates two child nodes, corresponding to two new subproblems. In order to create the children, B&B chooses a

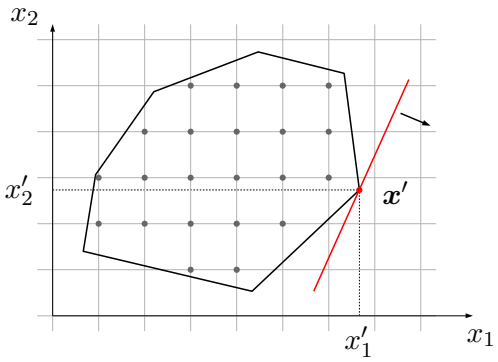
**Algorithm 1** Branch-and-bound algorithm.

**Inputs:** $p$ a MIP problem of the form (2.6)

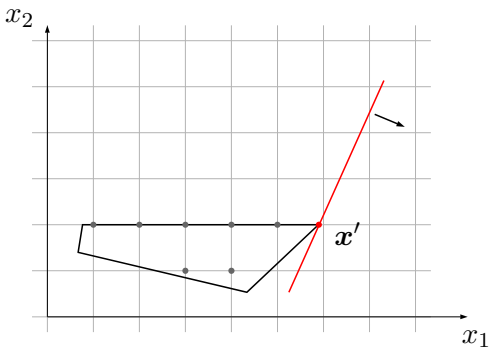
1: **procedure** BRANCH-AND-BOUND($p$)
2: $\boldsymbol{x}_{\text{best}} = \textbf{null}$ ; $o_{\text{best}} = +\infty$ ; $o_{\text{dual}} = -\infty$
3: $s = p$ with all integrality constraints relaxed ▷ initial LP relaxation of $p$
4: $q = \{s\}$ ▷ initialize the queue
5: **while** $q \neq \emptyset$ **do**
6: $s$ = retrieve next node to process and remove from the queue $q$
7: $\boldsymbol{x}'$ = solve the LP relaxation corresponding to node $s$
8: $o_{\text{dual}}$ = update dual bound with $\boldsymbol{c}^\top \boldsymbol{x}'$ if required ▷ dual bound update
9: **if** ($s$ is infeasible) **or** ($\boldsymbol{c}^\top \boldsymbol{x}' \geq o_{\text{best}}$) **then**
10: **continue**
11: **end if**
12: $F(s) = \{i \,|\, i \in I \wedge x'_i \notin \mathbb{Z}\}$ ▷ compute the set of current fractional variables
13: **if** $F(s) = \emptyset$ **then** ▷ integral solution found
14: **if** $\boldsymbol{c}^\top \boldsymbol{x}' \leq o_{\text{best}}$ **then** ▷ primal bound update
15: $\boldsymbol{x}_{\text{best}} = \boldsymbol{x}'$ ; $o_{\text{best}} = \boldsymbol{c}^\top \boldsymbol{x}'$
16: **end if**
17: **else** ▷ solution is fractional
18: $i^*$ = find a branching variable in $F(s)$ ▷ find a branching variable
19: $c_{\text{left}} = s + (x_{i^*} \leq \lfloor x'_{i^*} \rfloor)$ ▷ create left child by adding a constraint to $s$
20: $c_{\text{right}} = s + (x_{i^*} \geq \lceil x'_{i^*} \rceil)$ ▷ create right child by adding a constraint to $s$
21: $q = q \cup \{c_{\text{left}}; c_{\text{right}}\}$
22: **end if**
23: **end while**
24: **return** ($\boldsymbol{x}_{\text{best}}$ ; $o_{\text{best}}$ ; $o_{\text{dual}}$)
25: **end procedure**

variable $x_i$, with $i \in F(s)$, and adds, to the current subproblem, additional constraints that are meant to cut, from the current set of feasible solutions, the current value of variable $x_i$, i.e., $x'_i \notin \mathbb{Z}$. One child subproblem is created by adding to the current subproblem one constraint of the form $x_i \leq \lfloor x'_i \rfloor$, and the other child subproblem is created by the addition of $x_i \geq \lceil x'_i \rceil$, such that variable $x_i$ is forbidden to take, in the descendants of the current node, a value in the open interval $(\lfloor x'_i \rfloor, \lceil x'_i \rceil)$. In particular, these two constraints ensure that, in the subtree rooted at the current node, the variable $x_i$ will never take the value $x'_i$ again. This operation is called branching on variable $x_i$ and it results in the expansion of the tree by two new open nodes. The exploration of the parent is then over and the corresponding node is labeled as closed. Figure 2.4 illustrates a branching operation on a toy problem and also shows the resulting child problems.

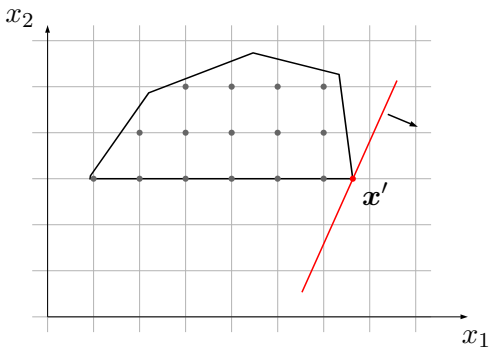
In the two first cases, the optimization tree is not developed, i.e., the nodes are marked as closed and no child nodes are created, which reduces the total size of the tree, and, hence, the optimization time.




(a) Parent node. The solution of the LP-relaxation at the current node is not integral.




(b) Left child. The constraint $x_2 \leq \lfloor x_2' \rfloor$ is added to the parent problem.




(c) Right child. The constraint $x_2 \geq \lceil x_2' \rceil$ is added to the parent problem.

Figure 2.4: Illustration of the branching operation performed by B&B. Subfigures (a), (b), and (c) represent the parent node, and the resulting left and right child nodes, respectively. In this example, variable $x_2$ is chosen as branching variable, but variable $x_1$ could have been chosen too. The solution of the LP-relaxation at the current node is not integral. In order to create two child problems, branching is thus performed by adding, to the set of constraints of the parent problem, one additional constraint for each child. The polyhedrons defining the LP-relaxations of the child problems correspond to the intersection between the polyhedron of the parent and the additional constraint. For each subproblem, a number of integral solutions are removed by the addition of the constraint.

The three steps composing B&B are applied iteratively to the optimization tree, whose root node corresponds to a version of the original problem where all integrality constraints are relaxed. Once the tree has been entirely explored, i.e., when the list of open nodes is empty, the integral solution with the smallest objective value is returned as the optimal solution of the initial problem. Note that if the optimization is to be stopped before all the nodes are processed (e.g., because of time constraints), the algorithm can still provide a feasible solution (if it has found one), but that solution is not guaranteed to be optimal. The optimality can only be proved through the exploration of the entire optimization tree.

**Primal and dual bounds** One of the advantages of B&B is that, at any point in time during the optimization, there exist two bounds on the objective value of the problem being solved. These bounds are very useful to assess how far from termination the algorithm is and how good the available solutions are.

The first bound, known as 'primal bound', corresponds to the smallest objective value of all integral solutions found thus far. Actually, a real primal bound becomes available as soon as the algorithm finds a first feasible solution. While no solution is available, the primal bound is usually set to $+\infty$. Fortunately, primal bounds become available very early in the optimization. Indeed, state-of-the-art solvers traditionally use 'heuristics' to find primal bounds for the current problem. Heuristics are functions that can turn non-integral solutions into (non-optimal) integral solutions. These heuristics can be applied at any node of the tree and are often applied, in particular, at the root node. This implies that primal bounds are usually available from the very beginning of the optimization. Note, however, that heuristics are, in general, not guaranteed to succeed and may be applied in vain, thus leaving the primal bound to $+\infty$. The second bound, also called 'dual bound', corresponds to the smallest objective value of the solutions of the LP-relaxations contained in the open nodes of the tree.

The primal bound is an upper bound on the optimal objective value (corresponding to the optimal integral solution) of the problem and is used to prune the nodes in the B&B tree. The optimal objective of the current problem is thus, at worst, equal to the primal bound. The dual bound, on the other hand, is a lower bound on the optimal objective value. At any moment, the optimal objective is, at best, equal to the dual bound. During the algorithm, the primal bound decreases and the dual bound increases and the algorithm terminates when the two bounds become equal. Consequently, the bounds difference can be used to evaluate whether the algorithm is close to termination or not. At the beginning of B&B, the primal bound is initialized to infinity (unless integer solutions are available) and the dual bound is set to the objective value of the LP-relaxation at the root node.

**Remark 1** In this work, we sometimes refer to the dual bound increase of a given branching. This should be understood as 'the difference between the objective value of the LP-solution of the child nodes and the objective value of the LP-solution of the parent node'. Since the algorithm terminates when the primal bound becomes equal to the dual bound, a 'good' branching is, in general, one that will yield a large dual bound increase. Note that the dual bound is global, i.e., it has the same meaning throughout the entire optimization tree, while the dual bound increase is a local concept (local to a parent and its two children).

**Remark 2** The above description of B&B is universal, in the sense that it is the version of B&B applied to general MIP problems. When the studied problem is binary, the procedure can be simplified a bit. In the binary case, the initial relaxation for each variable $x_i \in \{0, 1\}$ is $x_i \in [0, 1]$. Consequently, the left child node is created by the addition of the constraint $x_i \leq 0$, while the right child is created by the addition of $x_i \geq 1$. These constraints force the variable to 0 and 1 in the left and right child, respectively. The resulting optimization tree becomes simpler to understand and analyze.

### Additional features of the B&B

The above description of the B&B focuses only on the main mechanisms of the algorithm and is enough to implement a pure version of the method. Several tricks and additional features can be used to improve the efficiency of the implementation in order to solve larger and harder problems. Without going into the details, we describe here some important techniques that can dramatically improve the performance of the B&B. More specifically, we discuss the preprocessing, the branching strategy, the node selection strategy, and the cutting planes.

**Preprocessing** In all MIP solvers, it is common to apply so-called preprocessing techniques to the problem to be solved before calling B&B. The basic idea behind preprocessing is to render the problem easier so that it can be solved more quickly by B&B. More specifically, preprocessing analyzes the problem and extracts interesting facts that can be used to remove redundant or useless variables and/or constraints from the formulation. Preprocessing can additionally tighten the bounds on certain variables based on the properties of the problem. Preprocessing techniques can also sometimes determine that the problem is infeasible without actually applying B&B. It is to be noted however that applying those techniques is somewhat time consuming and that a tradeoff must be found to ensure that the optimization time gain due to a smaller problem is not lost in preprocessing time. There exist many different preprocessing techniques that can decrease the time needed to solve a problem, but the thorough description of those methods is beyond the scope of this work.

**Branching strategy** The branching strategy, i.e., the function that chooses an index $i$ in the set $F(s)$ containing the indices of the fractional variables, is probably the component of B&B that most influences the efficiency of the optimization. Indeed, the quality of the branching strategy conditions the number of nodes that the algorithm explores before terminating. This number of nodes has, of course, to be as small as possible so as to minimize the time required to solve the problem.

Good branching decisions typically reduce the number of nodes processed by the algorithm. Indeed, if the subproblems created by a branching have objective values that are greater than the current primal bound, then these child nodes can be pruned from the tree, thus reducing the remaining amount of work. On the other hand, a bad branching decision, one that does not increase much the objective value, will end up with two new nodes to process in the queue. Taking a good branching decision is usually not trivial and is quite time consuming. There is usually a tradeoff between, on the one hand, taking a good branching decision, and,

on the other hand, quickly taking a decision. We omit here the details about the different branching strategies, since a much more complete description is given in Chapter 4.

**Node selection strategy** Another important aspect of B&B is the order in which the nodes are explored by the algorithm, which is called the node selection strategy. There exist several simple possibilities derived from the standard existing tree search methods. Depth-first and breadth-first are obvious examples of such simple strategies. Another frequently used approach, called best-bound, consists in selecting as next node to process the node with the lowest LP-relaxation objective value. In general, the best-bound strategy is applied when one wants to increase the dual bound on the current optimization and is typically regarded as the one that yields the smallest number of nodes. However, best-bound is also known to be memory consuming because the nodes accumulate in the queue. On the other hand, the memory requirements of the depth-first strategy are very limited and depth-first usually finds integral solutions quite quickly. Indeed, since depth-first travels the optimization tree from the top to the very bottom, depth-first creates more and more constrained versions of the original problem and therefore leads to the creation of nodes that have a higher probability of being 'closed' either due to infeasibility or due to integrality of their solution (the latter case may additionally possibly decrease the primal bound, and hence lead to further pruning of the tree). Consequently, depth-first is usually applied in order to decrease the current primal bound and in restricted-memory settings.

All node selection strategies have advantages and shortcomings that must be taken into account during the optimization. For instance, if a solver wants to contain the amount of memory it uses, depth-first is probably the best choice. However, the approaches to node selection implemented in commercial solvers do not use a single strategy but commonly involve several strategies between which the solver switches during the optimization to leverage the strengths of the available strategies.

**Cutting planes** Cutting planes, or cuts for short, are additional constraints that are added to the problem in order to 'tighten' the formulation. These cuts are important since they reduce the feasible set of the problem and, hence, tend to speed up the optimization. The goal of the cuts is to shrink the initial polyhedron, i.e., to tighten the formulation, with the hope that the vertices of the shrunk polyhedron become closer to integer solutions. If that happens, i.e., if the vertices of the shrunk polyhedron correspond to integer solutions, the solutions of the LP-relaxations become integral and the problem becomes much easier since no branching is required as the optimal solution of the LP-relaxation is already integral. Note that the cuts must not rule out integer solutions, since they are meant to discard non-integral solutions only. Figure 2.5 illustrates how cuts tighten and simplify the initial problem. There exist many different types of cutting planes, each of which with different properties, but their thorough description is beyond the scope of this work.

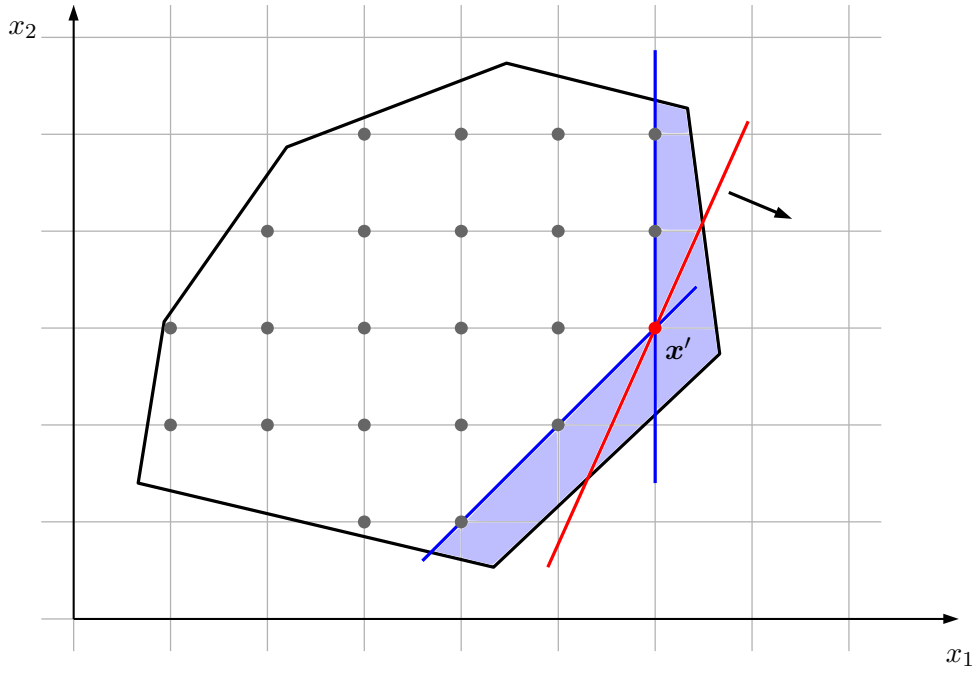



Figure 2.5: Example with two cutting planes tightening the formulation of an initial MIP problem. In that case, the optimal solution of the relaxation of the new formulation (i.e., the optimal vertex of the new polyhedron) corresponds to an integer solution, and the problem is thus solved directly with an LP technique, rather than with B&B. The cutting planes are displayed in blue in the figure and the area removed by the cuts corresponds to the shaded area.

# Chapter 3

# Understanding machine learning

Machine learning (ML) is a relatively new science and the community seems not to have agreed on a common definition yet. ML can thus be described in different terms depending on the point of view and the sensitivity of the scientist. Some define machine learning as a subfield of computer science that is concerned with the development, study, and implementation of algorithms that are able to learn from and make predictions on data (see, e.g., Hastie et al., 2009). Some others, including the author of this manuscript, allow the contour of the definition to be more flexible. This alternate definition considers machine learning as a subfield of artificial intelligence focused on the development, study, and implementation of methods allowing machines to evolve through a learning procedure (see, e.g., Samuel, 1959; Gutierrez, 2015). This second definition encompasses the first one but also comprises more applications and methods. However, and despite the apparent disagreements about the definition, the community does not worry much over such inconsequential problems.

Machine learning is a very wide field of research and introducing every aspect of it, seen from different perspectives, is, at best, a very laborious work, at worst, a titanic task, and, in the end, definitely not the goal of this short introduction. Therefore, and leaving aside our own wide definition of machine learning, this section solely introduces the different facets of ML that are mandatory for the proper understanding of this work, disregarding some major aspects of utmost intellectual interest.

In this work, we only consider the so-called 'supervised machine learning' framework. The goal of this short introduction is merely to give enough information to the unfamiliar reader so that the remainder of this thesis becomes clear to everybody. The main intuition as well as the main mechanisms are provided in this document, but we refer the reader to appropriate scientific material for a deeper understanding of these concepts. The machine learning specialists may safely continue their way through the manuscript and jump directly to more advanced chapters.

This chapter first introduces the supervised machine learning framework that is central to this research. Then, the following sections describe in greater detail the supervised ML algorithms that we use in this work.

## 3.1 A gentle introduction to supervised machine learning

### 3.1.1 Definition

Let $\mathcal{T}$ be some task that maps input states $s \in \mathcal{S}$ to an output space $\mathcal{Y}$, i.e.,

$$\mathcal{T} : \mathcal{S} \mapsto \mathcal{Y},$$

where $\mathcal{S}$ is the set of possible input states of the task. The task $\mathcal{T}$ is not necessarily well characterized. For instance, it can be a black box, like a physical system or a human behavior. Examples of tasks $\mathcal{T}$ include protein folding and spam filtering. In the first case, the state $s$ corresponds to a sequence of amino acids, which univocally describes a protein, the output $y$ corresponds to the tridimensional structure of the protein defined by the input sequence, and the task $\mathcal{T}$ corresponds to the folding process guided by the laws of physics and chemistry. In the second example, the input state $s$ is an email, the output $y$ is a label (spam or not) and the task consists for the receiver of the email in deciding whether the email is spam or not.

In general, it may be desirable to mathematically characterize or to simulate the task in order to study and better understand the process. The previous examples perfectly illustrate this goal. In the case of the protein folding problem, being able to predict the tridimensional structure of a protein without long and costly experiments would be a major obstacle out of the way of drug designers, possibly leading to groundbreaking discoveries in medicine. Similarly, being able to predict whether an email is spam or not without human intervention could save users a substantial amount of time. Supervised machine learning has the potential to achieve those goals. Generally speaking, supervised learning techniques use observations of the task $\mathcal{T}$ to approximate its behavior. The outcome of the learning procedure can then be used as a proxy for $\mathcal{T}$, i.e., as an approximation of the real task, in numerous situations.

For the sake of simplicity, we introduce machine learning in the easy case where the output is a scalar, i.e., $\mathcal{Y} \subset \mathbb{R}$. Nonetheless, there exist applications where this is not true and where the output belongs to more complicated spaces, e.g., $\mathcal{Y} \subset \mathbb{R}^p$, where $p$ is the dimension of the output space. Protein folding is an example of such an application where the output is multidimensional. Additionally, other settings (e.g., structured output spaces) have also been studied, but are not discussed here.

Basically, supervised ML focuses on the construction, from available data, of a function $f^*$ that mimics the task $\mathcal{T}$, i.e.,

$$\mathcal{T}(\cdot) \approx f^*(\cdot) \in \mathcal{F}^* : \Phi \mapsto \mathcal{Y},$$

where the function $f^*$ maps inputs from a space $\Phi \subset \mathbb{R}^d$ to an output space $\mathcal{Y} \subset \mathbb{R}$ and $\mathcal{F}^*$ represents the set of possible mappings from $\Phi$ to $\mathcal{Y}$. Ideally, the input of $f^*$ should be $s \in \mathcal{S}$, the input state of $\mathcal{T}$ itself. However, representing the complete state is often a difficult problem, e.g., because its dimensionality is too high or because it contains a lot of irrelevant information. For this reason, in the machine learning community, the inputs $\boldsymbol{\phi} \in \mathbb{R}^d$ of the functions $f \in \mathcal{F}^*$ are usually characteristics, called 'features', that represent a simplified version of the input state $s$. Designing meaningful features is a key factor to success in a learning problem, since the features control how close to the real task $\mathcal{T}$ the approximated function $f^*$ can get. The features are discussed in greater detail in Section 3.1.5.

Formally, a supervised machine learning algorithm $\mathcal{A}$ is a procedure of the form

$$\mathcal{A} : (\Phi \times \mathcal{Y})^* \mapsto \mathcal{F},$$

where the wildcard '$*$' stands for $\forall N \in \mathbb{N}$ and $\mathcal{F}$ denotes the set of candidate functions, which is referred to as the 'hypothesis space' and depends on the particular learning algorithm that is used. Note that the set $\mathcal{F}$ is usually different from the set $\mathcal{F}^*$ and is such that $\mathcal{F} \subset \mathcal{F}^*$.

More practically, a supervised machine learning algorithm takes as input a training set $D_{\text{train}} = \{(\boldsymbol{\phi}_i, y_i)\}_{i=1}^N \in (\Phi \times \mathcal{Y})^N$ (of any size $N$) obtained from the task $\mathcal{T}$ and outputs a function $\hat{f} \in \mathcal{F}$ that minimizes some loss function $\mathcal{L}_{\text{train}}$ on the given dataset. Stated in mathematical terms, the learned function, or model, $\hat{f}$ resulting from the application of algorithm $\mathcal{A}$ to the training set $D_{\text{train}}$ is often specified by

$$\hat{f}(\cdot) = \mathcal{A}(D_{\text{train}}) = \underset{f \in \mathcal{F}}{\operatorname{argmin}} \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}_{\text{train}}(y_i, f(\boldsymbol{\phi}_i)). \tag{3.1}$$

Naturally, the hope is that the machine learning algorithm produces a function $\hat{f}$ that approximates closely enough the sought function $f^*$, i.e.,

$$\hat{f}(\boldsymbol{\phi}) \approx f^*(\boldsymbol{\phi}), \ \forall \boldsymbol{\phi} \in \Phi.$$

Equation (3.1) that defines $\hat{f}$ implies that fitting a model to a given dataset, i.e., learning the model, amounts to solving an optimization problem. For most learning algorithms, the learning procedure can indeed (sometimes with some effort) be regarded as an optimization problem. Of course, not all optimization formulations are easy to solve and choosing between several learning algorithms is often based on the difficulty of the underlying optimization problem (computation time, uniqueness of the solution, etc.). Note that the loss function $\mathcal{L}_{\text{train}}$ used in Equation (3.1) is, in general, such that

$$\mathcal{L}_{\text{train}} : \mathcal{Y}^2 \mapsto \mathbb{R}$$

and is used to measure, for a candidate model $f'$, how close a certain output value $y' = f'(\boldsymbol{\phi}_i)$ is to some target output $y = y_i$ given in the training sample.

It is interesting to mention that some learning algorithms do not solve exactly Equation (3.1), but a modified version of it, where additional elements are added within the minimization for several purposes (e.g., to control the complexity of the algorithm, to favor sparsity of the model, etc.). Consequently, some learning algorithms, instead of Equation (3.1), solve the more general equation

$$\hat{f}(\cdot) = \mathcal{A}(D_{\text{train}}) = \underset{f \in \mathcal{F}}{\operatorname{argmin}} \left\{ \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}_{\text{train}}(y_i, f(\boldsymbol{\phi}_i)) + \mathcal{R}(f) \right\}, \tag{3.2}$$

where $\mathcal{R} : \mathcal{F} \mapsto \mathbb{R}$ is an additional component minimized by the learning algorithm that can serve several purposes. In some situations, $\mathcal{R}(\cdot)$ may be referred to as a regularization term. Equation (3.2) encompasses most learning methods, but it is still possible that a few learning algorithms require a more complex mathematical formulation. We omit this discussion here.

Let us finally mention that the strength of supervised machine learning relies on its ability to generalize behaviors observed on data with very few assumptions needed. This makes it a powerful tool when one wants to imitate unknown functions for which no, or very little, information is available. The main requirement is that the machine learning procedure is given a dataset containing input-output pairs $(\boldsymbol{\phi}_i, y_i)$ obtained from the task $\mathcal{T}$ that the ML algorithm is trying to imitate. Those input-output pairs can be obtained by simulation, there is no need to actually know the functional representation of the real underlying function that is to be learned.

### 3.1.2 Batch and online learning

As mentioned above, the main requirement to apply supervised machine learning techniques is to have at one's disposal a training set $D_{\text{train}} = \{(\boldsymbol{\phi}_i, y_i)\}_{i=1}^{N} \in (\Phi \times \mathcal{Y})^N$ containing input-output pairs obtained from the task $\mathcal{T}$. Within supervised learning, there exist two main frameworks that differ by the initial availability of the data.

First, the most common framework, known as batch or offline learning, assumes that the data is available to the learning algorithm from the very beginning. In that case, Equation (3.1) can be solved directly, during a training or learning phase and then subsequently used to make predictions. However, it is not always realistic to assume that the entire dataset is available from the beginning. In this situation, another learning framework can be used. The second main framework, known as online learning, deals with the case where the data arrives progressively. There are several possibilities regarding the arrival rate of the data. For instance, the observations may arrive one at a time or may arrive packed, i.e., several observations become simultaneously available. An online learning algorithm is typically an iterative procedure that receives, at each iteration, so-called mini-batches, i.e., small training sets of the form $D_j = \{(\boldsymbol{\phi}_i, y_i)\}_{i=1}^{\tilde{N}}$ with $1 \leq \tilde{N} \ll N$ and $j$ the iteration number. At a given iteration $j$, online algorithms maintain a learned model $\hat{f}_j$ that is the result of successive learning steps that used the mini-batches as training data. When a new mini-batch arrives, i.e., $D_j$ becomes available, the algorithm uses the newly available data to update the current model and generate a new approximation $\hat{f}_{j+1}$ of the target function. Hopefully, the approximation becomes better as new mini-batches become available, i.e., $\hat{f}_{j+1}$ is better than $\hat{f}_j$. Unfortunately, this is not always the case and special care must be taken to ensure that the approximation of the task does not deteriorate as the data arrives.

The border between batch and online learning is quite fuzzy and the two frameworks do not differ much. Indeed, all algorithms can be applied in one framework or in the other. It is especially true for online learning algorithms that can be adapted very easily to the batch setting. It suffices to consider that the data is available from the very beginning and that, subsequently, no more data arrives. The converse is true as well, but applying batch algorithms in the online case is usually less easy and much less efficient from the computational point of view. Indeed, most batch algorithms require the entire dataset to build the model and incorporating new data cannot be done easily. A possibility to learn in an online setting with a batch algorithm would be to retrain the model, when new data arrives, with the dataset augmented with the newly available observations. It clearly appears that such a procedure is not very efficient.

From the computational perspective, online learning may seem better. It is true in general, but this is usually achieved at the expense of the learning accuracy. Indeed, online learning algorithms tend to be slightly weaker predictors than batch algorithms. It turns out that both settings have advantages and shortcomings and that no one is better that the other in all aspects. Deciding which algorithm to use depends, in the end, on the considered application. In this work, we use both settings, since we tackle problems whose specific requirements favor either one setting or the other.

### 3.1.3 Training, test, and generalization errors

The performance of a machine learning algorithm or of a learned model is traditionally evaluated through the computation of error measures. The computed errors can then be used to determine which algorithm or model is the best. In machine learning, we distinguish three types of errors: the training error, the test error, and the generalization error.

In general, the end goal of a learning algorithm is to discover the relationship that exists between the entire input space $\Phi$ and the entire output space $\mathcal{Y}$. The generalization error measures this aspect by quantifying how well a model approximates the sought input-output relationship over the entire space. Assuming that there exists a joint distribution over the input-output space $\Phi \times \mathcal{Y}$ according to which the observations are drawn, i.e., $(\boldsymbol{\phi}, y) \in \Phi \times \mathcal{Y}$ follow a distribution $\mathcal{D}$, the generalization error for a given learned model $\hat{f}$ is defined as

$$e_{\text{gen}}\left(\hat{f}\right) = \mathbb{E}_{(\boldsymbol{\phi},y)\sim\mathcal{D}}\left[\mathcal{L}_{\text{test}}\left(y_i, \hat{f}\left(\boldsymbol{\phi}_i\right)\right)\right],$$

where the expectation is taken over the observations $(\boldsymbol{\phi}, y)$ drawn from distribution $\mathcal{D}$ and $\mathcal{L}_{\text{test}} : \mathcal{Y}^2 \mapsto \mathbb{R}$ is a loss, or error, function.

Minimizing the generalization error is the goal that is pursued by all learning algorithms. However, the generalization error is typically very hard to compute because the input-output space $\Phi \times \mathcal{Y}$ is usually very large (or even infinite and uncountable, for instance, if one feature or the output is a scalar) and because it requires the knowledge of the distribution $\mathcal{D}$. When it is not conceivable to compute exactly the generalization error, the test error is used instead to assess the quality of a model. The test error is computed from a finite collection of observations $D_{\text{test}}$, whose elements are assumed to be independent and identically distributed according to $\mathcal{D}$. More specifically, the test error is given by

$$e_{\text{test}}\left(\hat{f}\right) = \frac{1}{N_{\text{test}}} \sum_{(\boldsymbol{\phi}_i, y_i)\in D_{\text{test}}} \mathcal{L}_{\text{test}}\left(y_i, \hat{f}\left(\boldsymbol{\phi}_i\right)\right),$$

where the test set $D_{\text{test}}$ is such that $D_{\text{test}} = \{(\boldsymbol{\phi}_i, y_i) : (\boldsymbol{\phi}_i, y_i) \sim \mathcal{D},\ i = 1, \ldots, N_{\text{test}}\}$ and $\mathcal{D}$ is the same distribution as for the generalization error. The test error favorably estimates the generalization error and is somewhat easier to compute (because it does not require the knowledge of the distribution $\mathcal{D}$). The test error is therefore very often used, in place of the generalization error, to compare the quality of several learned models to identify the best one.

While the generalization and test errors are used to assess a learned model, the training error is, on the other hand, used by the learning algorithm to train a model, i.e., to choose a function $\hat{f} \in \mathcal{F}$, so that the chosen model fits the available data. For a given model $\hat{f}$,

the training error $e_{\text{train}}$ corresponds to the error computed from the training set $D_{\text{train}} = \{(\boldsymbol{\phi}_i, y_i)\}_{i=1}^{N}$ that is used for learning[1], i.e.,

$$e_{\text{train}}\left(\hat{f}\right) = \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}_{\text{train}}\left(y_i, \hat{f}\left(\boldsymbol{\phi}_i\right)\right).$$

A learning algorithm typically chooses a model that minimizes the training error $e_{\text{train}}$, although additional terms can be taken into account during the minimization (see, for instance, Equation (3.2) and its discussion). The training error is simple to compute and is used as a proxy for the generalization error during the learning. The training error can indeed be regarded as a proxy for the generalization error under certain circumstances, but the correlation between both errors is not guaranteed. In particular, note that the training error only models the performance of the proposed model for the training set $D_{\text{train}}$, which represents only a portion of the input-output space on which the generalization error focuses.

It is important to understand that the training error, on the one hand, and the test and generalization errors, on the other hand, are intrinsically different. While the former is used to train the model from the available data, the latter are used to assess the learned function. Besides, the data used to compute the test and generalization errors is different from the training data and it is very important to keep it that way. Indeed, since the training error is meant to be a proxy to the generalization error, it is tempting to conclude that a model that performs well in terms of the training error will perform well in terms of the generalization error. Such a shortcut is dangerous because it omits to take into account the fact that the training data is available in limited amount. What happens is that the learning algorithm naturally places too much emphasis on the training data and totally ignores the rest of the input-output space. This is a normal behavior, but it implies that care should be taken to make sure that the learning does not become too specific. The generalization and test errors are meant to evaluate independently from learning the quality of a model and are therefore computed from data that is as independent as possible from the training data. In the end, their goal is to tell whether the model generalizes well, i.e., is able to predict a correct output for unseen data, or not. Generalization is the true goal of machine learning and making sure that the trained models are able to generalize what they observe in the training data is of primary importance.

Also note that the losses $\mathcal{L}_{\text{test}}$ and $\mathcal{L}_{\text{train}}$, which are used by the test and train errors, respectively, may be the same, but may also be different. One of the reasons for this is that $\mathcal{L}_{\text{test}}$, which corresponds to the real error that one wants to minimize for a given application, may be very poorly suited to optimization. In that case, it often happens that another loss, $\mathcal{L}_{\text{train}}$, which can be handled much more easily by an optimization algorithm, is used as a proxy to $\mathcal{L}_{\text{test}}$ during the training of the model. Obviously, this further justifies the use of the test error to assess the quality of the model in the real setting of the considered application since another loss is used during the training.

[1]Note that, ideally, the training set $D_{\text{train}}$ is, like the test set, composed of independent and identically distributed observations drawn from the joint distribution $\mathcal{D}$. Unfortunately, this is not necessarily the case and a poorly chosen training set may hinder the ability of the learning algorithm to correctly approximate the input-output relationship over the entire space.

### 3.1.4 Model complexity

The set $\mathcal{F}$ of possible output functions (see Equations (3.1) and (3.2)) depends on the particular machine learning algorithm that is used and possibly on the values of some parameters. The richness of the set $\mathcal{F}$ is referred to as the expressive power or 'descriptive complexity' of the set $\mathcal{F}$. The richer the set $\mathcal{F}$, the finer the dynamics that can be captured. From this, it seems that adding to $\mathcal{F}$ complex functions can only yield better results, but this is not true and controlling (both downwards and upwards) the complexity of learning is important. Indeed, restricting the size of $\mathcal{F}$, i.e., allowing only simple models, may result in missing some important aspects of the sought input-output relationship. For instance, if a straight line is used to approximate data arranged according to a square function, the model will inevitably fail to correctly approximate some portions of the space. On the other hand, excessively increasing the richness of $\mathcal{F}$, i.e., allowing very complex models, may encourage the learning algorithm to focus on complicated dynamics that seemingly matter, but actually do not. In the extreme case, the model can even learn noise, which should obviously be avoided as much as possible. It therefore appears that tuning the complexity of the hypothesis space is important in order to get the machine learning algorithm to work as expected on the considered problem. The machine learning community studied this complexity aspect quite thoroughly, but the technicalities that are hidden behind the complexity are beyond the scope of this work. We refer the reader to appropriate literature for further details (see, e.g., Vapnik, 2013).

When one wants to improve the performance of a learned model, re-learning a more complex model on the same data is usually a good idea. Unfortunately, as detailed above, increasing the complexity does not always yield the expected results. In order to understand why a greater complexity can be a problem, it is important to keep in mind that the training and generalization errors are different by nature. While the former measures the accuracy of a model on the training data, the latter is meant to assess the model over the entire input-output space. Consequently, since both errors are computed on fundamentally different sets, a model that yields a small training error is not guaranteed to yield a small generalization error, and can very well produce a very large one. This phenomenon is known as overfitting. Overfitting occurs when the model starts learning dynamics that are specific to the training set, but that are meaningless for the rest of the input-output space. Overfitting implies that the learned model focuses too much on the provided dataset and does not 'generalize' enough. Overfitting is illustrated in Figure 3.1, which shows that increasing the complexity of the hypothesis space beyond a certain point may be useful to decrease the training error, but becomes counterproductive when it comes to the minimization of the generalization error.

### 3.1.5 The features

As briefly discussed in Section 3.1.1, the input of the candidate models $f \in \mathcal{F}$ is most often not $s \in \mathcal{S}$, the input of task $\mathcal{T}$, itself. Indeed, the raw input $s$ can represent a wide variety of things from numbers and text, to complex objects like images, large molecules, or time series. Instead of using directly $s$ and in order to make the input more easily understandable by machines, the arguments of the functions $f \in \mathcal{F}$ are typically a simplified version of $s$, which take the form of vectors of features or characteristics. These vectors of features can

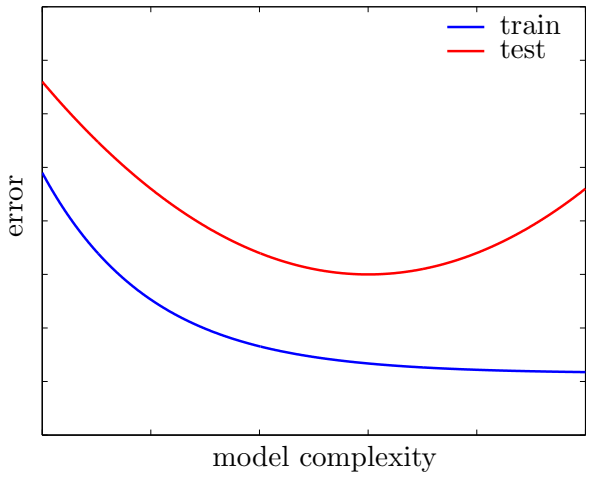



Figure 3.1: Training and test error vs. model complexity. Example showing overfitting when the trained model overfits the training set at the expense of the generalization (test) error.

usually be manipulated much more easily by an algorithm than the raw input $s$. Moreover, the features are designed so that they represent only part of the current state $s$, ideally the part that most influences the output.

Formally, a vector of features, or features for short, is a vector of characteristics that are extracted from the input state $s \in \mathcal{S}$, i.e.,

$$\mathcal{C}(s) = \boldsymbol{\phi} \in \Phi \subset \mathbb{R}^d,$$

where $\mathcal{C}$ is a function that computes the desired characteristics from the input $s \in \mathcal{S}$. In general, features are numeric, either continuous or categorical, but other features such as strings, histograms, or graphs can be used as well. The concept of 'feature' exists in other fields and, for instance, is known as 'explanatory variable' in the statistics community.

The importance of features is twofold. First, representing an arbitrary input object $s$ by a vector of features constitutes, in a sense, a standard in the field of machine learning that allows a learning algorithm to be developed independently of applications. Second, the features often critically determine the efficiency of learning methods. Indeed, as they represent only part of the current state of the task, it is important that the parts described by the features are actually correlated with the desired output. For this reason, the features need to be carefully designed and tailored to the problem of interest.

To illustrate features with some example, let us use the image classification problem, which consists in assigning a class (e.g., house, food, human, car) to an image. The problem thus amounts to analyzing the content of an input image and to deciding to which class the image must be assigned to. There exist many features that can be used for such an application, e.g., histograms of colors, histograms of gradients, and presence/absence of certain shapes. Designing features is a complicated task and new features are often described in dedicated papers. In the computer vision domain, feature descriptors include the well-known SIFT (Lowe, 1999), GIST (Oliva and Torralba, 2001), and HoG (Dalal and Triggs, 2005) features.

## 3.2 The regression problem

There exist different machine learning frameworks that depend on several aspects of the task under study. For instance, the output $y$ of interest can be either continuous or categorical (i.e., take only a finite number of values) or can have a special structure, depending on the considered problem. The nature of the output is one way to characterize different learning frameworks. In this manuscript, we concentrate on the regression problem.

The regression problem focuses on the basic case where the output $y$ is a continuous real number. In this setting, the loss functions $\mathcal{L}$ take simple forms like, for example,

$$\mathcal{L}\left(y_i, f\left(\boldsymbol{\phi}_i\right)\right) = \begin{cases} \left|y_i - f\left(\boldsymbol{\phi}_i\right)\right|, & \text{known as the absolute error,} \\ \left(y_i - f\left(\boldsymbol{\phi}_i\right)\right)^2, & \text{known as the squared error.} \end{cases}$$

Regression finds many applications in practice, e.g., in the financial industry (credit scoring, algorithmic trading), in biology (gene expression, protein characterization), and in power systems (load forecasting, security assessment). This section endeavors to describe two traditional regression methods: linear regression and regression trees.

### 3.2.1 Linear regression

Linear regression is probably the simplest learning technique for regression problems. Linear regression makes the assumption that the output $y \in \mathbb{R}$ is linearly dependent on the inputs $\boldsymbol{\phi} \in \mathbb{R}^d$. In mathematical terms, linear regression finds a function $\hat{f}$ such that

$$\hat{f}\left(\cdot\right) \in \mathcal{F} = \left\{ f\left(\cdot\right) : f(\boldsymbol{\phi}) = \sum_{i=1}^{d} w_i \boldsymbol{\phi}(i) = \boldsymbol{w}^\top \boldsymbol{\phi} \right\}, \tag{3.3}$$

where $\boldsymbol{w} \in \mathbb{R}^d$ and $\boldsymbol{\phi} \in \mathbb{R}^d$ are two column vectors representing the parameters of the model and the inputs, respectively, and $\boldsymbol{\phi}(i)$ represents the $i$th component of the vector $\boldsymbol{\phi}$. Linear regression is illustrated in Figure 3.2.

Training a linear regression model in the form of Equation (3.3) is generally easy. Indeed, the search for a function is replaced by the search for optimal weights $\hat{\boldsymbol{w}}$ that completely define the sought function $\hat{f}$. Assuming a training set $D_{\text{train}} = \{(\boldsymbol{\phi}_i, y_i)\}_{i=1}^{N}$, we can build a feature matrix $\boldsymbol{K}$, whose rows are composed of the features $\boldsymbol{\phi}_i$, and an output matrix (column vector) $\boldsymbol{Y}$, whose unique column is composed of the training outputs $y_i$. In matrix form, $\boldsymbol{K}$ and $\boldsymbol{Y}$ are given by

$$\boldsymbol{K} = \begin{bmatrix} \boldsymbol{\phi}_1^\top \\ \boldsymbol{\phi}_2^\top \\ \vdots \\ \boldsymbol{\phi}_N^\top \end{bmatrix}, \text{ and } \boldsymbol{Y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix},$$

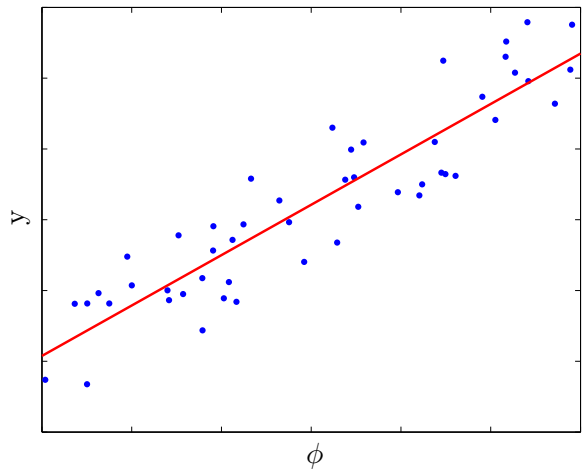



Figure 3.2: Linear regression example with a unidimensional input ($d = 1$) and noisy data. The blue points correspond to the available data that is used to train the model represented by the red line.

respectively. In order to fit the model to the data, i.e., in order to find the function $\hat{f}(\cdot)$, it then suffices to solve the following optimization problem

$$\begin{aligned}\hat{\boldsymbol{w}} &= \underset{\boldsymbol{w}\in\mathbb{R}^d}{\operatorname{argmin}} \sum_{i=1}^{N} \left(\boldsymbol{w}^\top \boldsymbol{\phi}_i - y_i\right)^2 \\ &= \underset{\boldsymbol{w}\in\mathbb{R}^d}{\operatorname{argmin}} \left(\boldsymbol{K}\boldsymbol{w} - \boldsymbol{Y}\right)^\top \left(\boldsymbol{K}\boldsymbol{w} - \boldsymbol{Y}\right) \\ &= \underset{\boldsymbol{w}\in\mathbb{R}^d}{\operatorname{argmin}} \left\|\boldsymbol{K}\boldsymbol{w} - \boldsymbol{Y}\right\|_2^2, \end{aligned} \tag{3.4}$$

where the squared error is used as loss function and $\hat{f}(\boldsymbol{\phi}) = \hat{\boldsymbol{w}}^\top \boldsymbol{\phi}$. Note that other loss functions could be used, but the matrix form formulation may not be as straightforward as in the case of the squared loss. Additionally, a regularization term $\mathcal{R}(\boldsymbol{w})$ (see Equation (3.2)) is often added within the minimization in Equation (3.4) to control the complexity of the model. Such modifications of the linear regression method are not covered here.

Fitting the model to the data, i.e., finding the model parameters $\hat{\boldsymbol{w}}$ with Equation (3.4), is formulated as a minimization problem that can be solved quite easily. Indeed, under mild conditions (among others that the matrix $\boldsymbol{K}$ has rank $d$), the minimized function has a positive-definite Hessian matrix, which implies that the function is convex, and, hence, possesses a unique minimizer. Setting the first derivative of the loss function to 0 yields

$$\hat{\boldsymbol{w}} = \left(\boldsymbol{K}^\top \boldsymbol{K}\right)^{-1} \boldsymbol{K}^\top \boldsymbol{Y},$$

which expresses the solution of linear regression as a simple product of matrices (see, e.g., Hastie et al., 2009).

In statistics, this approach to estimate the parameters $\hat{\boldsymbol{w}}$ (augmented with a few assumptions on the data) is known as the ordinary least squares method. Note that Equation (3.4)

is written when it is assumed that the entire dataset is available at learning time. This corresponds to batch linear regression. It is also possible to easily adapt linear regression to the online learning setting by, for instance, applying stochastic gradient descent to Equation (3.4). In that case, each term of the sum is minimized separately, as the data arrives. We omit the particulars here.

Among other advantages, linear regression is a simple method in many aspects. It is easy to understand, to train, and to use, and both training and prediction are computationally efficient. On the downside, the assumptions (linear dependency between the inputs and the output) are very strong and much likely to be incorrect in many practical cases. Obviously, if the assumed model strongly differs from the real function that generates the data, learning such a model does not make much sense. In this respect, the features are of utmost importance since linear regression can only model linear dependencies between the inputs and the output. The features must thus be chosen such that the output is linearly dependent on them. Actually, the features are probably the most critical factor that conditions the performance of the method. Additionally, other aspects, like the scale of the features, can hinder the performance of linear regression. These issues illustrate that the simplicity of the method comes at a cost: modeling complex input-output relationships is possible only if the features are designed by keeping in mind the limitations of the method. In the end, choosing or developing features that work well with linear regression may overweigh the benefits of using such a simple method. Besides the features, numerical aspects are another potential issue of linear regression. Indeed, the stability and the relevance of the results significantly depend on the data and, more specifically, on how well the matrix $\boldsymbol{K}$ is conditioned. If $\boldsymbol{K}$ is ill-conditioned, the solution is usually very sensitive to perturbations in the data, e.g., noise, and the results may consequently be unexploitable because a small change in the data (either in $\boldsymbol{K}$ or in $\boldsymbol{Y}$) can yield a totally different solution. When such situations arise, using a regularized version of linear regression (see Equation (3.2)) often improves the conditioning (and, hence, the stability) of the method. Ridge regression is an example of regularized linear regression. In the case of ridge regression, Equation (3.4) becomes

$$\hat{\boldsymbol{w}} = \underset{\boldsymbol{w}\in\mathbb{R}^d}{\operatorname{argmin}} \|\boldsymbol{K}\boldsymbol{w} - \boldsymbol{Y}\|_2^2 + \|\boldsymbol{\Gamma}\boldsymbol{w}\|_2^2 ,$$

where $\boldsymbol{\Gamma}$ is called the Tikhonov matrix and is often a multiple of the identity matrix, i.e., $\boldsymbol{\Gamma} = \lambda\mathbb{I}$. Like traditional linear regression, ridge regression possesses an analytical solution, given by

$$\hat{\boldsymbol{w}} = \left(\boldsymbol{K}^\top\boldsymbol{K} + \boldsymbol{\Gamma}^\top\boldsymbol{\Gamma}\right)^{-1}\boldsymbol{K}^\top\boldsymbol{Y},$$

with the difference that the conditioning is improved due to the presence of $\boldsymbol{\Gamma}$ (Hastie et al., 2009). Intuitively, the effect of regularization in ridge regression is to favor small weights $\hat{\boldsymbol{w}}$ since one component of the minimization is the $L_2$-norm of a multiple of the weight vector. Besides ridge regression, there exist other types of regularization that improve the conditioning of matrix $\boldsymbol{K}$ and that have different effects on the solution (e.g., favor sparse solutions), but these other methods are not discussed here.

Despite a few shortcomings, linear regression is a good learning algorithm (because of its simplicity) that may not produce the best prediction results but that is often used as a first tool to tackle new problems. Indeed, in the context of machine learning, it is typically appropriate to try simple methods first and to move on to more complicated approaches when

required, in order to avoid as much as possible overfitting issues. In general, machine learners like to apply the 'Occam's razor' principle, which states that the simplest explanation is often the best one. The interpretation of this principle in terms of machine learning implies that simple models should be preferred when possible.

### 3.2.2 Regression trees

When simple methods like linear regression are not enough, one can resort to more complicated learning techniques. This section focuses on the description of regression trees that, despite their conceptual simplicity, are, in many aspects, a very powerful approach. We first present the single regression tree approach, before briefly describing, in Section 3.2.3, more advanced approaches that involve ensembles of trees.

A regression tree is a powerful and versatile learning technique that uses (in general) hyperplanes to partition the input space, i.e., the feature space, into several regions. The idea is to partition the input space into regions within which the behavior of the studied system is simple enough. A region is typically defined by a collection of inequalities of the form $\boldsymbol{\phi}(j) \leq t$ or $\boldsymbol{\phi}(j) > t$, where $\boldsymbol{\phi}(j)$ and $t$ are referred to as the splitting variable and the split point, respectively (Hastie et al., 2009). Each region thus represents a part of the feature space where the dynamics of the initial system are easier to approximate than when the entire input space is considered as a whole. In each region, a simple model (e.g., a constant) is learned in order to approximate the system in that particular part of the input space.

The rationale behind the concept of the regression tree is conceptually simple but remains extremely powerful. Furthermore, regression trees can cope with both numerical and categorical inputs, and can be used for both classification and regression. An example of a regression tree is depicted in Figure 3.3.

We now focus on the training of regression trees in the simplest case where the model used in each region is a constant. Assuming that the feature space is partitioned into $m$ regions denoted $R_1, R_2, \ldots, R_m$, the prediction $f(\boldsymbol{\phi})$ of an arbitrary regression tree corresponding to an input vector $\boldsymbol{\phi}$ is given by

$$f(\boldsymbol{\phi}) = \sum_{k=1}^{m} c_k I(\boldsymbol{\phi} \in R_k), \tag{3.5}$$

where $c_k$ is a constant associated with region $R_k$, and $I(\boldsymbol{\phi} \in R_k)$ is an indicator function whose value is 1 if $\boldsymbol{\phi}$ belongs to region $R_k$ and 0 otherwise. Similarly to the linear regression case, our aim is to build the function $\hat{f}(\boldsymbol{\phi})$, of the form of Equation (3.5), such that a loss, say the squared loss, is minimized over our training set $D = \{(\boldsymbol{\phi}_i, y_i)\}_{i=1}^{N}$, i.e.,

$$\hat{f}(\cdot) = \underset{f \in \mathcal{F}}{\operatorname{argmin}} \sum_{i=1}^{N} \left(f(\boldsymbol{\phi}_i) - y_i\right)^2, \tag{3.6}$$

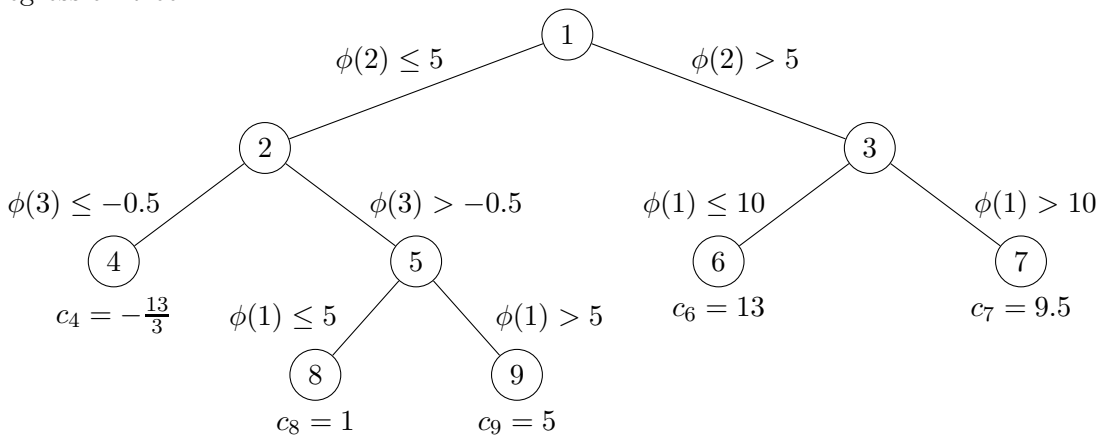
where $\mathcal{F}$ is the set of functions that can be written as in Equation (3.5).

In order to build the tree, the algorithm has to automatically find the splitting variables and split points in order to define the regions $R_k$. The constants $c_k$ corresponding to each

Training set:

$$\boldsymbol{K} = \begin{bmatrix} \boldsymbol{\phi}_1^\top \\ \boldsymbol{\phi}_2^\top \\ \boldsymbol{\phi}_3^\top \\ \boldsymbol{\phi}_4^\top \\ \boldsymbol{\phi}_5^\top \\ \boldsymbol{\phi}_6^\top \\ \boldsymbol{\phi}_7^\top \\ \boldsymbol{\phi}_8^\top \\ \boldsymbol{\phi}_9^\top \\ \boldsymbol{\phi}_{10}^\top \\ \boldsymbol{\phi}_{11}^\top \\ \boldsymbol{\phi}_{12}^\top \end{bmatrix} = \begin{bmatrix} 12 & 15 & 3 \\ 13 & 10 & -2 \\ 3 & 15 & 8 \\ 6 & 6 & 4 \\ 8 & 20 & -2 \\ 2 & 5 & -1 \\ 7 & -2 & -2 \\ 12 & 3 & -2 \\ 2 & 4 & 1 \\ 1 & 4 & 1 \\ 4 & -1 & 5 \\ 8 & 2 & 6 \end{bmatrix} \qquad \boldsymbol{Y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \\ y_9 \\ y_{10} \\ y_{11} \\ y_{12} \end{bmatrix} = \begin{bmatrix} 9 \\ 10 \\ 13 \\ 12 \\ 14 \\ -4 \\ -5 \\ -4 \\ 1 \\ 0 \\ 2 \\ 5 \end{bmatrix}$$

A regression tree:




Training examples - leaf association:

| leaf # | data points belonging to the leaf | leaf # | data points belonging to the leaf |
|---|---|---|---|
| 4 | $\left(\boldsymbol{\phi}_6^\top, y_6\right) = \left([2,5,-1], -4\right)$<br>$\left(\boldsymbol{\phi}_7^\top, y_7\right) = \left([7,-2,-2], -5\right)$<br>$\left(\boldsymbol{\phi}_8^\top, y_8\right) = \left([12,3,-2], -4\right)$ | 6 | $\left(\boldsymbol{\phi}_3^\top, y_3\right) = \left([3,15,8], 13\right)$<br>$\left(\boldsymbol{\phi}_4^\top, y_4\right) = \left([6,6,4], 12\right)$<br>$\left(\boldsymbol{\phi}_5^\top, y_5\right) = \left([8,20,-2], 14\right)$ |
| 8 | $\left(\boldsymbol{\phi}_9^\top, y_9\right) = \left([2,4,1], 1\right)$<br>$\left(\boldsymbol{\phi}_{10}^\top, y_{10}\right) = \left([1,4,1], 0\right)$<br>$\left(\boldsymbol{\phi}_{11}^\top, y_{11}\right) = \left([4,-1,5], 2\right)$ | 7 | $\left(\boldsymbol{\phi}_1^\top, y_1\right) = \left([12,15,3], 9\right)$<br>$\left(\boldsymbol{\phi}_2^\top, y_2\right) = \left([13,10,-2], 10\right)$ |
| 9 | $\left(\boldsymbol{\phi}_{12}^\top, y_{12}\right) = \left([8,2,6], 5\right)$ | | |

Figure 3.3: Regression tree example. This example provides a training set, a regression tree learned from the given data, and a table showing which training example goes into which leaf of the tree. When a new output value $y$ has to be predicted for a given input vector $\boldsymbol{\phi}$, this input vector is fed at the root of the tree and then travels down the tree until it reaches a leaf. The predicted value is then the $c$ value corresponding to the reached leaf.

region are easy to find. Indeed, differentiating the loss in each region and setting the derivative to 0 leads to constants $c_k$ that are given by (in the case of the squared loss)

$$c_k = \frac{\sum_{i=1}^{N} y_i I\left(\boldsymbol{\phi}_i \in R_k\right)}{\sum_{i=1}^{N} I\left(\boldsymbol{\phi}_i \in R_k\right)},$$

which corresponds to the average of the output values $y_i$ of the observations $\boldsymbol{\phi}_i$ belonging to region $R_k$. The constants $c_k$ can thus be easily computed once the structure of the tree is fixed.

Unfortunately, finding the tree structure that minimizes the total loss given in Equation (3.6) is a very hard problem (Hastie et al., 2009). Instead of searching for the global optimum, the tree learning algorithm proceeds in a greedy manner to find a simple local optimum. Starting from the root node with the entire dataset, the algorithm needs to find a split, i.e., a splitting variable $j^*$ and a split point $t^*$, that partitions the initial input space into two regions

$$R_l(j^*, t^*) = \{\boldsymbol{\phi} \,|\, \boldsymbol{\phi}(j^*) \leq t^*\} \text{ and } R_r(j^*, t^*) = \{\boldsymbol{\phi} \,|\, \boldsymbol{\phi}(j^*) > t^*\},$$

such that the total local loss is minimized. In mathematical terms, this corresponds to finding $j^*$ and $t^*$ such that

$$(j^*, t^*) = \underset{j,t}{\operatorname{argmin}} \left[ \sum_{i:\boldsymbol{\phi}_i \in R_l(j,t)} \left(y_i - c_l(j,t)\right)^2 + \sum_{i:\boldsymbol{\phi}_i \in R_r(j,t)} \left(y_i - c_r(j,t)\right)^2 \right] \tag{3.7}$$

with

$$c_l(j,t) = \frac{\sum_{i=1}^{N} y_i I\left(\boldsymbol{\phi}_i \in R_l(j,t)\right)}{\sum_{i=1}^{N} I\left(\boldsymbol{\phi}_i \in R_l(j,t)\right)} \text{ and } c_r(j,t) = \frac{\sum_{i=1}^{N} y_i I\left(\boldsymbol{\phi}_i \in R_r(j,t)\right)}{\sum_{i=1}^{N} I\left(\boldsymbol{\phi}_i \in R_r(j,t)\right)}.$$

where $c_l(j,t)$ and $c_r(j,t)$ correspond to the averages of the output values $y$ of the training observations belonging to $R_l(j,t)$ and $R_r(j,t)$, respectively. For a given splitting variable $j$, the optimal split point $\hat{t}$ can be found somewhat efficiently provided that the data points $(\boldsymbol{\phi}_i, y_i)$ are sorted according to $\boldsymbol{\phi}(j)$. Since finding the optimal $\hat{t}$ for a given $j$ is easy, the optimal pair $(j^*, t^*)$ can thus be determined by evaluating the function to minimize for each possible splitting variable. Once the optimal split is known, the initial dataset is partitioned into two new datasets according to the optimal splitting variable and optimal split point. The algorithm then applies the same procedure, i.e., finding the optimal split and partitioning the dataset, for the datasets resulting from the previous split. In that way, a regression tree is grown where each node corresponds to a region defined by a group of splits. The regions appearing in the prediction function defined by Equation (3.5) correspond to the regions defined by the leaves of the tree, i.e., the tree has $m$ leaves that partition the input space in $R_1$, $R_2$, ..., $R_m$. Note that the inner nodes are not part of the prediction procedure itself, but are used to efficiently find the leaf corresponding to an input vector $\boldsymbol{\phi}$.

The previous paragraph describes how the tree is grown starting from its root. The natural question that arises now concerns the stopping criterion. When should the previous algorithm stop, or, in other words, how large should the tree be? A tree that is too small may miss important dynamics appearing in the data, while an oversized tree may overfit the training data. The size of the tree, i.e., the number of nodes, is traditionally regarded as an image of

the complexity of the model. Finding a good balance between the fitting of the dataset and the tree complexity is thus an important aspect of the method. A simple approach to limit the size of the tree consists in forbidding splits of a node if the optimal split induces a decrease in the considered loss that is not significant enough. Another approach consists in preventing splits of nodes whose number of elements is below a certain threshold $n_{\min}$. A more advanced approach to control the complexity of the method consists in growing large trees capturing the dynamics of the data in a very detailed manner and then pruning the resulting trees, i.e., removing nodes, in order to reduce the complexity of the model. The details of the pruning procedure are left out of this manuscript. We refer the reader to, e.g., Hastie et al. (2009) for more information on this issue.

Regression trees have the advantage that they are simple to understand, to train and to use. Additionally, they can be very accurate, i.e., achieve small losses, provided that a large enough dataset is available. On the downside, regression trees are said to have a high variance. Indeed, the learning procedure is heavily dependent on the training set and small changes in the data can produce radically different trees. Additionally, a potential issue may arise regarding the prediction time: regression trees are not the most stable learning models when it comes to prediction time. Indeed, predicting an output value for a given input is achieved by traversing the learned tree. Because the trees may be unbalanced and because the predictions can take different paths down the tree, the prediction time can differ significantly between two different inputs. By comparison, the linear regression method is much more stable: prediction consists in computing an inner product between the weights and the input, which will be equivalent for all predictions whatever the input.

### 3.2.3 Ensemble of regression trees

As mentioned in the previous section, one of the major drawbacks of regression trees is their high variance: a small change in the training data can possibly yield completely different regression trees. The wording 'high variance' is thus used because of the sensitivity of the method to the training set. Fortunately, there are solutions to decrease the variance of tree-based methods and, therefore, to increase their robustness. We concisely describe here two methods, namely the 'random forests' and the 'extremely randomized trees', that favorably replace single regression trees in many practical applications. Before detailing the differences between the traditional regression trees and these two more advanced methods, we first define the so-called ensemble learning paradigm and the bagging approach.

#### Ensemble methods and bagging

Ensemble learning is a machine learning paradigm that leverages the so-called 'wisdom of crowds' in the context of machine learning. More specifically, ensemble learning consists in combining a set of learning methods in order to achieve higher predictive power. There exist several implementations of this principle, but we focus here solely on bagging.

Bagging (Breiman, 1996), also known as bootstrap aggregation, is an ensemble approach whose primary goal is to increase the stability of learning algorithms (Hastie et al., 2009). More specifically, bagging averages the output of many relatively unbiased models in order

to reduce the variance of the averaged model. The bagging procedure is composed of two steps. First, $M$ random datasets denoted $D_i$ are generated by sampling (uniformly with replacement) from the initial dataset $D_{\text{train}}$. These datasets are often called bootstrap samples or bootstrapped datasets. Second, a single learning algorithm is applied separately on each bootstrap sample in order to yield $M$ different models $f_i$, with $i = 1, \ldots, M$. The models are then aggregated and the prediction of the aggregated model for a given input $\boldsymbol{\phi}$ is given by

$$f_{\text{avg}}(\boldsymbol{\phi}) = \frac{1}{M} \sum_{i=1}^{M} f_i(\boldsymbol{\phi}).$$

Bagging is especially well-suited for learning algorithms that have low bias and high variance. Because they comply with these requirements, regression trees are natural candidates to be used with bagging.

The claim of bagging is that the aggregated model $f_{\text{avg}}(\cdot)$ is better than the individual models $f_i(\cdot)$. In order to understand why bagging is useful, we report Hastie et al. (2009)'s explanation. Let us assume that the output of a model for a given input $\boldsymbol{\phi}$ can be regarded as a random variable with mean $\mu$ and variance $\sigma^2$. The real output corresponding to input $\boldsymbol{\phi}$ is approximately equal to $\mu$ since we assume that the models have low bias. Assume now that we have $M$ different models $f_i(\cdot)$, with $i = 1, \ldots, M$, whose outputs, for the same input $\boldsymbol{\phi}$, all have the same mean $\mu$ and the same variance $\sigma^2$. The average of the $f_i(\cdot)$, i.e., $f_{\text{avg}}(\cdot)$, is a random variable since the sum of $M$ random variables is still a random variable. The mean of the aggregated model $f_{\text{avg}}(\boldsymbol{\phi})$ remains equal to $\mu$, which is close to the real output. However, the theory of statistics tells us that the variance of $f_{\text{avg}}(\boldsymbol{\phi})$, denoted $\sigma^2_{\text{avg}}$, is now given by

$$\sigma^2_{\text{avg}} = \rho\sigma^2 + \frac{1-\rho}{M}\sigma^2,$$

where $\rho$ is the (positive) pairwise correlation between the $M$ individual models. If the models are independent from each other, the correlation $\rho$ is equal to 0 and the variance of the aggregated model converges to 0 as the number of models increases. However, since it is in general not possible to assume that the models are uncorrelated, there is a residual amount of variance ($\rho\sigma^2$) that bagging cannot eliminate, even if an infinite number of models are aggregated.

The expression of $\sigma^2_{\text{avg}}$ clearly shows that, when $M$ increases, the resulting variance of the aggregated model decreases. Since the predictive power of a model is usually a decreasing function of the variance, it is easy to see why the aggregated model obtained with the bagging procedure is indeed an improvement over the individual models. In the general case where the bootstrap models are correlated, decreasing their correlation is a way to further improve the efficiency of bagging.

### Random forests

The 'random forests' method (Breiman, 2001) proposes to combine bagging with regression trees. A random forests model is thus composed of $M$ regression trees, each one of them trained on a different bootstrap sample. The prediction is obtained by averaging the predictions of the individual trees. The major innovation of the random forests lies in a simple trick

that decorrelates the different bootstrap models in order to reduce the correlation $\rho$ between the individual models. Random forests are simple to train and to tune and their performance is traditionally very good (Hastie et al., 2009).

As stated, the random forests try to build $M$ uncorrelated trees. This uncorrelation is achieved through randomization of the training process. Indeed, the random forests, unlike traditional regression trees, consider, at each node, only a random subset of the features as splitting candidates (see Section 3.2.2). More specifically, only $m \leq d$ candidates are considered at each node, and these $m$ candidates are randomly selected among the $d$ features. In the regression case, a typical value for $m$ is $\sqrt{d}$. Apart from the bootstrap sample and the random selection of the candidate splitting variables, all other aspects of the tree growing procedure are analogous to the growing procedure of traditional regression trees. In particular, the split point is chosen so as to minimize some loss (see Equation (3.7), Section 3.2.2).

**Remark** The bagging approach and the randomized training procedure participate together in the reduction of the variance $\sigma^2_{\text{avg}}$ of the global method through the increase of $M$ and decrease of $\rho$, respectively. It is to be noted, however, that reducing the correlation is achieved at the expense of the variance of the individual models. Indeed, randomizing the tree growing procedure actually increases the variance of the individual regression trees. If the variance of a traditional regression tree is $\sigma^2_{\text{trad}}$ and the variance of a randomized regression tree is $\sigma^2_{\text{rand}}$, the relationship

$$\sigma^2_{\text{trad}} \leq \sigma^2_{\text{rand}}$$

holds in general (Hastie et al., 2009). The hope is naturally that this variance increase is counterbalanced by the reduction of $\rho$ in order to yield a global method that is better than the one obtained when no randomization is performed.

#### Extremely randomized trees

The 'extremely randomized trees' (Geurts et al., 2006) take the founding ideas behind random forests even further. As their name indicates, extremely randomized trees, or ExtraTrees, are even more randomized than random forests. There are two main differences between ExtraTrees and random forests that we detail in what follows.

First, in addition to randomly selecting the splitting candidates, the ExtraTrees also randomize the choice of the split point for each splitting variable. More specifically, when a random subset of the features is generated at each node of the individual trees, a random split point is drawn for each feature in the random subset from a uniform distribution between $\phi_i^-$ and $\phi_i^+$, where $\phi_i^-$ and $\phi_i^+$ represent the minimum and maximum values of feature $i$ in the training set available at the current node, respectively. The splitting variable is then chosen among the randomly selected candidates as the variable for which the random split minimizes a given loss.

The second difference between both methods is that each individual tree is grown using the entire training set in the ExtraTrees approach, while the individual trees are trained on bootstrap replicas in the case of random forests. The motivation of this choice lies in the impact that using bootstrap replicas has on the accuracy of ExtraTrees. The performance of

ExtraTrees tends indeed to decrease when bootstrap replicas are used instead of the complete learning dataset to grow the individual trees (Geurts et al., 2006).

Note that the remark formulated for the random forests (regarding the variance increase) also applies to extremely randomized trees.

# Chapter 4

# Machine learning for variable branching in branch-and-bound

The contents of this chapter are mainly reproductions of two pieces of work published online (Marcos Alvarez et al., 2014, 2016).

## 4.1 Introduction

In this chapter, we address binary mixed-integer linear programming (MILP) problems of the form (2.7). As detailed in Section 2.3.2, the branch-and-bound algorithm (Land and Doig, 1960), also referred to as B&B, is the method of choice when it comes to solving such problems. The description given in Section 2.3.2 focuses on the essential components of B&B. However, as briefly mentioned, there exist numerous additional features, such as cutting planes, presolve, heuristics, and advanced branching strategies, that can be added to the algorithm in order to improve its performance (Achterberg and Wunderling, 2013). Among those additional features, branching, i.e., the process that divides the feasible region into two or more subproblems, is probably the key component that most conditions the efficiency of the solver (Achterberg and Wunderling, 2013).

Branching strategies have been extensively studied in the literature and we briefly review here some key contributions to that field. The simplest criterion, known as *most-infeasible branching*, consists in branching on the variable that has the greatest fractional part, i.e., the variable whose fractional part is closest to 0.5. However, most-infeasible branching is known to perform poorly in practice and other methods, such as *pseudocost branching* (Benichou et al., 1971), have been developed later. Pseudocost branching keeps a history of the dual bound increases observed during previous branchings and uses this information to estimate the dual bound improvements for each candidate variable at the current node. Although pseudocost branching is very efficient in terms of computation time, the branchings performed at the very beginning of the B&B tree might be inefficient, as no reliable history has been recorded at that time. Later, Applegate et al. (1995) proposed a strategy, known as *strong branching*, that overcomes this limitation. Strong branching explicitly evaluates the dual bound increase for

each fractional variable by actually computing the LP relaxations resulting from the branching on that variable. The variable that leads to the largest increase is chosen as branching variable at the current node. Despite its apparent simplicity, strong branching is, up to now, the most efficient branching strategy in terms of the number of nodes in the B&B tree. However, this efficiency is achieved at the expense of computation time and strong branching is unfortunately intractable in practice. More recently, Achterberg et al. (2005) proposed to combine the advantages of both pseudocost and strong branching in a branching strategy called *reliability branching*. Many other branching strategies have been developed for the past 15 years, such as *inference branching* (Li and Anbulagan, 1997), *non-chimerical branching* (Fischetti and Monaci, 2012b), *active constraint branching* (Patel and Chinneck, 2007), and *cloud branching* (Berthold and Salvagnin, 2013), but their thorough description is beyond the scope of this work. Finally, let us mention *hybrid branching* (Achterberg and Berthold, 2009), which is probably today's state-of-the-art branching strategy. Hybrid branching efficiently combines five scores obtained from other common branching strategies and is used as the main branching strategy in CPLEX 12.5 (Achterberg and Wunderling, 2013).

Following the ideas introduced by pseudocost branching, researchers have recently started investigating branching strategies that rely on information collected through multiple B&B restarts. *Backdoor branching* (Fischetti and Monaci, 2012a) and *information-based branching* (Karzan et al., 2009) are two key contributions to this aspect. The mechanism of these strategies is two-phased. During the first phase, the optimization of the current problem is restarted from the beginning multiple times and the algorithm collects some statistics about each run. In the second phase, the real optimization starts, and the harvested information is used to take efficient branching decisions. The idea behind those methods is to quickly and superficially explore different parts of the B&B tree and to decide, based on those shallow explorations, which part it is better to focus on. Finally, let us mention the work of Di Liberto et al. (2013) who recently proposed a two-phased method that uses machine learning techniques to switch between several branching heuristics. More specifically, their method learns, during a first phase, when it is better to apply a given heuristic. During the second phase, i.e., the optimization per se, their algorithm switches between several branching heuristics based on the knowledge acquired during the first phase.

There exist numerous branching strategies and choosing one among all that are available is not easy. In general, the main criterion is the time. Indeed, the primary objective when solving an optimization problem is to solve it as quickly as possible. When using B&B, the time $T(P)$ required to solve a problem $P$ is approximately proportional to the number of nodes explored by the algorithm, i.e.,

$$T(P) = t_u N(P),$$

where $t_u$ and $N(P)$ represent the time required to process one node and the number of nodes in the optimization tree, respectively. In other words, in order to minimize the optimization time $T(P)$, it is important to minimize both the time to process one node and the number of explored nodes. The 'efficiency' of a branching strategy is thus a function of the time needed to take a branching decision and the size of the resulting B&B tree. In general, there is often a tradeoff between the time spent to process one node and the 'quality' of the resulting decision, which influences the size of the optimization tree. For instance, the time required to process one node with most-infeasible branching is very short, i.e., $t_u$ is very small, but the

resulting branching decisions are usually poor, i.e., $N(P)$ is large. On the other hand, strong branching tends to produce small trees, but the processing time of a node is very large.

The idea proposed, detailed, and assessed in this chapter consists in using machine learning techniques to approximate strong branching decisions without the computational cost of the normal strong branching approach.

## 4.2 Proposed approach

### 4.2.1 Foundation and motivation

A branching heuristic, i.e., a procedure that splits a given problem into several subproblems, can be formulated in a generic functional form $\mathcal{B}$ such that

$$\mathcal{B} : (i, \cdot) \mapsto \mathbb{R}, \tag{4.1}$$

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

$$i^* = \underset{i \in F}{\operatorname{argmax}} \, \mathcal{B}\left(i, \cdot\right),$$

where $F$ represents the set of fractional variables (see Section 2.3.2 for a description of the notations used here).

The functional form $\mathcal{B}$ is different for every branching criterion and proposing a new branching heuristic merely consists in providing a new $\mathcal{B}$, including its implementation and the specification of its arguments. For instance, in the case of most-infeasible branching (MIB), $\mathcal{B}_{\text{mib}}$ only requires the current fractional solution to output a score for a variable. Then, the functional form of MIB is written $\mathcal{B}_{\text{mib}}\left(i, \boldsymbol{x}'\right) = \min\left(1 - x'_i, x'_i\right)$. Another example is strong branching, which requires more input arguments, as it needs more information to take a decision. The functional form of strong branching (SB) is $\mathcal{B}_{\text{sb}}\left(i, \boldsymbol{c}, \boldsymbol{A}, \boldsymbol{b}, \boldsymbol{x}', \boldsymbol{l}', \boldsymbol{u}'\right)$, where $\boldsymbol{l}'$ and $\boldsymbol{u}'$ represent the lower and upper bounds of the variables at the current node, respectively. The implementation of $\mathcal{B}_{\text{sb}}$ consists in creating two subproblems by changing the upper and lower bounds of variable $i$ in the current problem to $\lfloor x'_i \rfloor$ and $\lceil x'_i \rceil$, respectively. The LP-relaxations of the subproblems thus created are then solved, and, for each subproblem, the difference between the objective value of the subproblem and the objective value of the current problem is computed. The differences computed for each subproblem represent the objective increases observed between the current node and the child nodes when tighter bounds are used for the variable $i$. The output of $\mathcal{B}_{\text{sb}}$ is finally given by the product[2] of the computed differences.

The strength of machine learning (ML) relies on its ability to generalize behaviors observed on data with very few assumptions needed. This makes it a powerful tool when one wants

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

[2] Other functions like the sum or the maximum of both objective increases can be used, but the product generally yields better results.

to imitate unknown functions for which no, or very little, information is available. The main requirement is that the machine learning procedure needs a dataset containing input-output pairs obtained from the function that the ML algorithm is trying to imitate. Those input-output pairs can be obtained by simulation or through a black-box function, there is no need to actually know the functional representation of the function to be learned.

Since machine learning techniques (see Section 3.1.1) can be used to approximate (learn) functions, and since branching can be seen as a function, it seems natural to consider machine learning as a reasonable way to build a branching strategy. We must stress here that machine learning does not provide a completely new branching criterion, as it requires the observation of real branching decisions to actually build a branching function. Rather than providing a new branching heuristic, machine learning constructs a branching strategy that imitates the decisions of the branching strategy that generated the dataset $D$, in our case, strong branching.

Our goal is to create an efficient approximation of strong branching that could be used in practice. In other words, we propose to create a branching heuristic $\hat{\mathcal{B}}(i, \boldsymbol{\phi}_i)$ such that

$$\hat{\mathcal{B}}(i, \boldsymbol{\phi}_i) \approx \mathcal{B}_{\text{sb}}\left(i, \boldsymbol{c}, \boldsymbol{A}, \boldsymbol{b}, \boldsymbol{x}', \boldsymbol{l}', \boldsymbol{u}'\right), \tag{4.2}$$

where $\boldsymbol{\phi}_i$ is a feature vector describing the state of variable $i$ at the current node of the optimization tree.

The functions in (4.2) differ in two ways: (i) $\hat{\mathcal{B}}$ is not based on a determined function, but rather learned from data, and (ii) $\hat{\mathcal{B}}$ takes as input a feature vector $\boldsymbol{\phi}$ describing the current state of the optimization problem. The success of such an approach relies on two aspects: (i) the learned branching strategy needs to closely approximate the strong branching function, and (ii) the function $\hat{\mathcal{B}}$ (including the generation of its inputs) needs to be fast to evaluate.

Ideally, the input of the branching function $\hat{\mathcal{B}}$ should be the complete state of the optimization problem at the current node plus the current state of the B&B. However, representing the complete state is often a difficult problem, e.g., because its dimensionality is too big, or because it contains a lot of irrelevant information. For this reason, in the machine learning community, the inputs $\boldsymbol{\phi}$ are usually 'features' representing a simplified version of the considered input. We refer the reader to Section 3.1.5 for further details on the features. The features are a fundamental component of our approach since they critically determine the efficiency of learning methods. As they represent only part of the current state of the task, it is important that the parts described by the features are indeed correlated with the desired output. For this reason, the features need to be carefully designed and tailored to the problem of interest. In our case, the feature vector $\boldsymbol{\phi}_i$ does not describe the current node of the B&B tree, but rather describes variable $i$ in the current node. Those features need to be computationally efficient and have to well represent the problem at the current B&B node from the perspective of variable $i$. Section 4.3 explains in more detail how the features are designed.

### 4.2.2 Description of the proposed approaches

Our goal is to overcome the large computational overhead resulting from a strong branching decision by creating a function $\hat{\mathcal{B}}$ that yields branching scores that are close to the real SB

scores and that is fast to evaluate. Speeding up strong branching-like decisions is not a new idea, as it is already behind other branching heuristics, such as reliability branching or non-chimerical branching. In this work, we propose two alternative approaches to create $\hat{\mathcal{B}}$ that use machine learning techniques in order to imitate the strong branching decisions in an efficient way.

### Offline learning of strong branching decisions

First, we propose a two-phased approach that yields a 'learned' branching strategy denoted by $\mathcal{B}_{\text{learned}}$ that can be used within B&B as a proxy to strong branching (SB). The first phase involves optimizing a set of training problems with strong branching as branching heuristic in order to generate a set of branching decisions. During this phase, each branching decision is recorded in a dataset, called training set, that will then be used by a machine learning algorithm to learn a function imitating SB decisions, i.e., $\mathcal{B}_{\text{learned}}$. In the second phase, we introduce, as any other branching heuristic, $\mathcal{B}_{\text{learned}}$ into B&B and evaluate its efficiency on a set of test problems. An important characteristic of our approach is that the first phase needs only to be done once. Indeed, once the branching heuristic $\mathcal{B}_{\text{learned}}$ is learned from the training set, it can be included directly into B&B, without requiring a new training phase each time a problem needs to be solved. In that sense, we can say that the learning phase can be done in an offline fashion, thus avoiding useless computational overhead at the beginning of each optimization.

The idea behind our approach can be seen as a combination of the ideas behind reliability branching and information-based branching. On the one hand, our approach is similar to reliability branching in the sense that we want to find a fast proxy to strong branching. However, the approaches differ in the means by which the information is collected, and in how the information is used. On the other hand, our approach is similar to information-based branching in the sense that it tries to use general information collected during a preliminary phase in order to improve the performance of the current optimization. In both cases, the idea is to gain a wide overview of the optimization and to use this overview to take sensible branching decisions. The main difference appears in the way information is collected: information-based branching harvests information through multiple restarts on the same problem, while our method collects optimization information obtained through the optimization of a set of different optimization problems. Another important difference is that, in the case of information-based branching, the collection phase needs to be done for each problem to optimize, while our approach only requires the information to be collected once. This method is detailed in Section 4.4.

### Online learning of strong branching decisions

Second, in an attempt to push further the ideas proposed by the previous information-based branching strategies, we present an approach that combines the ideas behind our first proposed approach, i.e., learned branching $\mathcal{B}_{\text{learned}}$, and reliability branching (Achterberg et al., 2005). More specifically, the proposed method uses online learning algorithms to learn a proxy to strong branching (SB). The idea is similar to the first approach that we propose in the sense that the goal is to take SB-like decisions without the computational cost induced by the

evaluation of the real SB scores. However, in this case, the learning is performed in an online manner, during the course of the optimization, which alleviates some important shortcomings of the previous approach. Indeed, since the learning is made online, no preliminary phase is required in order to learn the proxy and no computing time is thus wasted in such a step. Additionally, the learned function is really tailored to the studied problem rendering some previous machine learning (ML) concerns obsolete. Moreover, our method uses a reliability mechanism similar to that of reliability branching (Achterberg et al., 2005) to decide how the SB score is going to be computed.

Technically speaking, this method uses SB scores in order to rank the possible branching variables. The candidate with the largest score is chosen as branching variable. These scores can be computed in two ways: either through the normal SB procedure, i.e., by computing $\mathcal{B}_{\text{sb}}\left(i, \boldsymbol{c}, \boldsymbol{A}, \boldsymbol{b}, \boldsymbol{x}', \boldsymbol{l}', \boldsymbol{u}'\right)$, or thanks to our learned proxy function $\mathcal{B}_{\text{olb}}\left(i, \boldsymbol{\phi}_i\right)$. More specifically, if the approximation of the SB score for a candidate variable, i.e., $\mathcal{B}_{\text{olb}}\left(i, \boldsymbol{\phi}_i\right)$, is deemed unreliable, the real SB score is used and, conversely, if the approximation is trusted for that variable, the learned proxy is used to generate the score in place of the real SB function. The proxy function $\mathcal{B}_{\text{olb}}\left(i, \boldsymbol{\phi}_i\right)$ is expected to yield approximate SB scores that are close enough to the real scores, i.e.,

$$\mathcal{B}_{\text{olb}}\left(i, \boldsymbol{\phi}_i\right) \approx \mathcal{B}_{\text{sb}}\left(i, \boldsymbol{c}, \boldsymbol{A}, \boldsymbol{b}, \boldsymbol{x}', \boldsymbol{l}', \boldsymbol{u}'\right),$$

but in a much shorter amount of time than the real SB procedure. The SB proxy is learned in an online fashion because the data used to train the function is generated during the course of the B&B. We detail this idea in Section 4.5.

## 4.3 Feature description of an optimization variable

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

The features $\boldsymbol{\phi}_i$ that we describe assume that the problem is in the canonical form (2.7). Each feature vector $\boldsymbol{\phi}_i$ is computed for variable $i$ at the current node and represents the proper input for the functions $\mathcal{B}_{\text{learned}}$ and $\mathcal{B}_{\text{olb}}$. The features are divided into three subsets representing different aspects of the state of the current problem in the optimization, namely 'static problem features', 'dynamic problem features' and 'dynamic optimization features'.

Before describing the features that we use, we need to emphasize the three properties that these features should have. The proposed features might seem a bit twisted and counter-intuitive, but they have been designed such that the following requirements are fulfilled. If they are not, the methods will in general yield poor performance (the requirements are less crucial in the case of the online method $\mathcal{B}_{\text{olb}}$ that we propose).

These requirements are as follows:

1. the number of features needs to be independent of the size of the problem instance. Indeed, most learning algorithms can cope only with datasets in which all the feature vectors $\boldsymbol{\phi}_i$ have the same number of elements. If this number depends on the size of the problem, a different branching strategy must be learned for each problem size. This is of course an impractical situation, and enforcing the size-independency is the best way to obtain a single learned branching strategy that can be used for any problem size. This might seem a straightforward requirement, but the size-independency is not trivial to achieve. Elementary features such as $\boldsymbol{c}$, $\boldsymbol{A}$ or $\boldsymbol{b}$ cannot be used directly in this case.

2. the features should be invariant with respect to irrelevant changes in the problem, such as row or column permutation.

3. the developed features need to be independent of the scale of the problem, i.e., if the parameters ($\boldsymbol{c}$, $\boldsymbol{A}$, and $\boldsymbol{b}$) are multiplied by some factor, the features should remain identical.

Note that Hutter et al. (2014) recently introduced a certain number of features describing MIP problems. Beside a few of them, their features are by nature different and do not relate directly to the ones proposed in this paper. Additionally, since ML in this paper and in Hutter et al. (2014)'s paper targets distinct goals, the features that are relevant for each task are likely to be different.

### 4.3.1 Static problem features

The first set of features is computed from the sole parameters $\boldsymbol{c}$, $\boldsymbol{A}$, and $\boldsymbol{b}$. These features are calculated once and for all and they represent some static information about the problem. Their goal is to give an overall description of variable $i$ in the problem. These features are designed such that the aforementioned requirements are fulfilled.

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

The second class of static features is meant to represent the influence of the coefficients of variable $i$ in the coefficient matrix $A$. We develop three measures, namely $M_j^1$, $M_j^2$, and $M_j^3$, that describe variable $i$ within the problem in terms of the constraint $j$. Once the values of the measures $M_j$ are computed for variable $i$ and each constraint $j$, the corresponding features added to the feature vector $\boldsymbol{\phi}$ are given by $\min_j M_j$ and $\max_j M_j$ (these values are not included as is, but rather as two features: one representing the sign of the value and the other one being its absolute value). The rationale behind this choice is that, when it comes to describing the constraints of a given problem, only the extreme values are relevant.

The first measure $M_j^1$ is composed of two parts: $M_j^{1+}$ computed by $A_{ji}/\,|b_j|$, $\forall j$ such that $b_j \geq 0$, and $M_j^{1-}$ computed by $A_{ji}/\,|b_j|$, $\forall j$ such that $b_j < 0$. The minimum and maximum

values of $M_j^{1+}$ and $M_j^{1-}$ are used as features, to indicate by how much a variable contributes to the constraint violations.

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

Finally, the third measure $M_j^3$ represents the inter-variable relationships within the constraints. The measure is split into two parts $M^{3+}$ and $M^{3-}$ that are given by

$$M_j^{3+} = \frac{|A_{ji}|}{\sum_{k:A_{jk}\geq 0} |A_{jk}|}, \text{ and } M_j^{3-} = \frac{|A_{ji}|}{\sum_{k:A_{jk}<0} |A_{jk}|}.$$

$M_j^{3+}$ is in turn divided in $M_j^{3++}$ and $M_j^{3+-}$ that are calculated using the formula of $M_j^{3+}$ for $A_{ji} \geq 0$ and $A_{ji} < 0$, respectively. The same splitting is performed for $M_j^{3-}$. Again, the minimum and maximum of the four $M_j^3$ computed for all constraints are added to the features.

The static features are listed in Table 4.1. Measures $M^1$, $M^2$, and $M^3$ correspond to the features 4-11, 12-19, and 20-35 of Table 4.1, respectively.

### 4.3.2 Dynamic problem features

The second type of features is related to the solution of the problem at the current B&B node. These features are listed in Table 4.2.

These features contain the proportion of fixed variables at the current solution, the fractionality of variable $i$, the up and down Driebeek penalties (Driebeek, 1966) corresponding to variable $i$, normalized by the objective value $o'$ at the current node. Furthermore, the sensitivity range of the objective function coefficient of variable $i$, normalized by $|c_i|$, as well as the sensitivity range of the right hand size coefficient $b_r$ (also normalized) are added to the features, where $r$ is the index of the row corresponding to the leaving basic variable when variable $i$ is entering the basis. Finally, the slack variable and the dual variable corresponding to row $r$ are also added to the features.

Note that some values are added both in their natural scale and in a logarithmic scale to cope with different amplitudes that depend on the problem.

### 4.3.3 Dynamic optimization features

The last set of features used to describe variable $i$ within the B&B tree is meant to represent the overall impact of variable $i$ on the optimization. These features summarize global information that is not available from the single current node. They are listed in Table 4.3.

When branching is performed on a variable, the objective increases are stored for that variable. From these numbers, we extract statistics for each variable: the minimum, the maximum, the mean, the standard deviation, and the quartiles of the objective increases. These

| Feat. # | Description |
|---|---|
| 1 | $\operatorname{sign} c_i$ |
| 2-3 | $\lvert c_i\rvert / \sum_{k=1\ldots n:c_k>0} \lvert c_k\rvert$ ; $\lvert c_i\rvert / \sum_{k=1\ldots n:c_k<0} \lvert c_k\rvert$ |
| 4-5 | $\operatorname{sign}\left\{ [\ \min;\ \max\ ]_{j=1\ldots m:b_j\geq 0} A_{ji}/\lvert b_j\rvert \right\}$ |
| 6-7 | $\left\lvert [\ \min;\ \max\ ]_{j=1\ldots m:b_j\geq 0} A_{ji}/\lvert b_j\rvert \right\rvert$ |
| 8-9 | $\operatorname{sign}\left\{ [\ \min;\ \max\ ]_{j=1\ldots m:b_j<0} A_{ji}/\lvert b_j\rvert \right\}$ |
| 10-11 | $\left\lvert [\ \min;\ \max\ ]_{j=1\ldots m:b_j<0} A_{ji}/\lvert b_j\rvert \right\rvert$ |
| 12-13 | $\operatorname{sign}\left\{ [\ \min;\ \max\ ]_{j=1\ldots m} \lvert c_i\rvert/A_{ji} \right\}$ (if $c_i \geq 0$, else [ 0 ; 0 ]) |
| 14-15 | $\left\lvert [\ \min;\ \max\ ]_{j=1\ldots m} \lvert c_i\rvert/A_{ji} \right\rvert$ (if $c_i \geq 0$, else [ -1 ; -1 ]) |
| 16-17 | $\operatorname{sign}\left\{ [\ \min;\ \max\ ]_{j=1\ldots m} \lvert c_i\rvert/A_{ji} \right\}$ (if $c_i < 0$, else [ 0 ; 0 ]) |
| 18-19 | $\left\lvert [\ \min;\ \max\ ]_{j=1\ldots m} \lvert c_i\rvert/A_{ji} \right\rvert$ (if $c_i < 0$, else [ -1 ; -1 ]) |
| 20-21 | $\operatorname{sign}\left\{ [\ \min;\ \max\ ]_{j=1\ldots m:A_{ji}\geq 0} A_{ji}/\left(\sum_{k=1\ldots n:A_{jk}\geq 0} \lvert A_{jk}\rvert\right) \right\}$ |
| 22-23 | $\left\lvert [\ \min;\ \max\ ]_{j=1\ldots m:A_{ji}\geq 0} A_{ji}/\left(\sum_{k=1\ldots n:A_{jk}\geq 0} \lvert A_{jk}\rvert\right) \right\rvert$ |
| 24-25 | $\operatorname{sign}\left\{ [\ \min;\ \max\ ]_{j=1\ldots m:A_{ji}\geq 0} A_{ji}/\left(\sum_{k=1\ldots n:A_{jk}<0} \lvert A_{jk}\rvert\right) \right\}$ |
| 26-27 | $\left\lvert [\ \min;\ \max\ ]_{j=1\ldots m:A_{ji}\geq 0} A_{ji}/\left(\sum_{k=1\ldots n:A_{jk}<0} \lvert A_{jk}\rvert\right) \right\rvert$ |
| 28-29 | $\operatorname{sign}\left\{ [\ \min;\ \max\ ]_{j=1\ldots m:A_{ji}<0} A_{ji}/\left(\sum_{k=1\ldots n:A_{jk}\geq 0} \lvert A_{jk}\rvert\right) \right\}$ |
| 30-31 | $\left\lvert [\ \min;\ \max\ ]_{j=1\ldots m:A_{ji}<0} A_{ji}/\left(\sum_{k=1\ldots n:A_{jk}\geq 0} \lvert A_{jk}\rvert\right) \right\rvert$ |
| 32-33 | $\operatorname{sign}\left\{ [\ \min;\ \max\ ]_{j=1\ldots m:A_{ji}<0} A_{ji}/\left(\sum_{k=1\ldots n:A_{jk}<0} \lvert A_{jk}\rvert\right) \right\}$ |
| 34-35 | $\left\lvert [\ \min;\ \max\ ]_{j=1\ldots m:A_{ji}<0} A_{ji}/\left(\sum_{k=1\ldots n:A_{jk}<0} \lvert A_{jk}\rvert\right) \right\rvert$ |

Table 4.1: Static features used to describe variable $i$ at a given node of the B&B. To avoid repetitions, we use brackets to indicate that several features are obtained with the same expression by simply changing a small element. For instance, $3 + [\ a;\ b\ ]$ indicates that two features are computed with this expression: one is $3 + a$ and the other is $3 + b$.

| Feat. # | Description |
|---|---|
| 36 | depth of the current node / number of integer variables |
| 37 | $\min\left(x'_i - \lfloor x'_i \rfloor, \lceil x'_i \rceil - x'_i\right)$ |
| 38-40 | $\log(\,[$ down; up; down+up Driebeek penalties normalized by $o'\,])$ |
| 41-43 | $[$ down; up; down $\times$ up Driebeek penalties normalized by $o'\,]$ |
| 44-46 | $\text{sign}\left(\left[\, c_i^{\text{sen}-};\, c_i;\, c_i^{\text{sen}+}\,\right]\right)$ |
| 47-48 | $\log\left((c_i - c_i^{\text{sen}-})/\lvert c_i\rvert\right)$ ; $\log\left((c_i^{\text{sen}+} - c_i)/\lvert c_i\rvert\right)$ |
| 49-51 | $\text{sign}\left(\left[\, b_r^{\text{sen}-};\, b_r;\, b_r^{\text{sen}+}\,\right]\right)$ |
| 52-53 | $\log\left((b_r - b_r^{\text{sen}-})/\lvert b_r\rvert\right)$ ; $\log\left((b_r^{\text{sen}+} - b_r)/\lvert b_r\rvert\right)$ |
| 54 | slack variable used in constraint $r$ |
| 55 | dual variable corresponding to constraint $r$ |

Table 4.2: Dynamic problem features used to describe variable $i$ at a given node of the B&B. Index $r$ represents the row corresponding to the leaving basic variable when variable $i$ is entering the basis. To avoid repetitions, we use brackets to indicate that several features are obtained with the same expression by simply changing a small element. For instance, $3 + [\, a;\, b\,]$ indicates that two features are computed with this expression: one is $3 + a$ and the other is $3 + b$.

statistics are used as features to describe the variable for which they were computed. As those features should be independent of the scale of the problem, we divide each objective increase by the objective value at the current node, such that the computed statistics correspond to the relative objective increase for each variable.

Finally, the last feature added to this subset is the number of times variable $i$ has been chosen as branching variable, normalized by the total number of branchings performed.

| Feat. # | Description |
|---|---|
| 56-59 | [ mean; std; min; max ] of the observed objective increases for variable $i$ |
| 60 | number of times B&B branched on $i$ / total number of branchings |
| 61-64 | quartiles of the observed objective increases for variable $i$ |

Table 4.3: Dynamic optimization features used to describe variable $i$ at a given node of the B&B. To avoid repetitions, we use brackets to indicate that several features are obtained with the same expression by simply changing a small element. For instance, $3 + [\ a;\ b\ ]$ indicates that two features are computed with this expression: one is $3 + a$ and the other is $3 + b$.

## 4.4 Batch learning of strong branching decisions

This section details how strong branching can be approximated with a batch learning procedure. We first describe the method per se and then present some experiments and results. We finally discuss some aspects of the method in the last part of the section.

### 4.4.1 Learning branching decisions in a batch setting

Creating a new branching heuristic merely consists in providing an implementation of the branching function $\mathcal{B}$ (see Equation (4.1)). The first method that we propose provides a function $\mathcal{B}_{\text{learned}}(i, \boldsymbol{\phi}_i)$ that is learned in an offline fashion (batch learning) and then inserted into B&B to select the branching variables. The definition of the features $\boldsymbol{\phi}$ used to describe a given variable is a prerequisite in order to apply this method. These features are detailed in the previous section.

Once the features are known, the procedure to create $\mathcal{B}_{\text{learned}}$ starts with the generation of a dataset containing input-output observations of the system we want to model. In our case, we want the input to be a variable (or, more specifically, a feature description of a variable) and the output to be the corresponding strong branching score. When the dataset is generated, it can be fed to a machine learning algorithm that learns a model able to predict the output corresponding to a given input. The branching heuristic $\mathcal{B}_{\text{learned}}$ corresponds to this model trained on the generated dataset.

We now detail the dataset generation procedure and give some details regarding the learning algorithm used on the generated dataset.

#### Dataset generation

As explained in Section 3.1.1, supervised learning algorithms need a dataset in order to learn a function. Such a dataset is not available in our case, one must thus be generated so that our procedure can be applied. The first step towards the creation of $\mathcal{B}_{\text{learned}}$ with a supervised machine learning technique is thus to generate such a dataset from which the function can be learned.

In order to create a training set of pairs $(\boldsymbol{\phi}_i, y_i)$ that can be used for learning, we optimize the problems contained in some sets, which we call training problems, with B&B using strong

branching as branching heuristic. At each node explored by B&B during the optimization of those problems, the strong branching score $\mathcal{B}_{\text{sb}}\left(i, \boldsymbol{c}, \boldsymbol{A}, \boldsymbol{b}, \boldsymbol{x}', \boldsymbol{l}', \boldsymbol{u}'\right) = y_i$ is computed for each fractional variable $i \in F$ together with the features $\boldsymbol{\phi}_i$ associated with that variable. These features-score pairs are then saved in a dataset $D_{\text{sb}}$, which is then used as input of the learning algorithm to generate the learned branching strategy. Note that, for the sake of simplicity, we explain the method as if we predict the strong branching score. This is not entirely true. Indeed, for scaling reasons imposed by the learning procedure, we rather include in the dataset (and, thus, predict) the logarithm of the relative SB score. The relative SB score is obtained by taking the product of the objective increases induced by a given branching, divided by the absolute value of the LP objective at the current node. This transformation does not change the branching decisions of SB, but renders the learning problem much easier.

The problems that are used to train $\mathcal{B}_{\text{learned}}$ are detailed in Section 4.4.2.

#### Machine learning algorithm

Once the features are designed to correctly describe each variable of the problem, and once a dataset of input-output pairs is available, a machine learning algorithm can be applied to learn a function from the available data.

In this work, we use *Extremely Randomized Trees* (Geurts et al., 2006), or ExtraTrees, that are a learning method based on an ensemble of regression trees. Our choice is motivated by the simplicity and the robustness of ExtraTrees. Indeed, the performance of ExtraTrees is very robust against the choice of their parameters and the default parameter values work very well in practice (see Geurts et al., 2006). We refer the reader to Section 3.2.3 for a complete description of ExtraTrees.

Applying the ExtraTrees to the available dataset is straightforward and yields $\mathcal{B}_{\text{learned}}$, a function that takes as argument an input vector describing a variable in an optimization problem and that outputs an approximation of the SB score. The function $\mathcal{B}_{\text{learned}}$ can then be used without effort within B&B in place of the normal strong branching function $\mathcal{B}_{\text{sb}}$.

### 4.4.2 Experiments

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

#### Problem sets

We use two types of problem sets, namely randomly generated problem sets and standard benchmark problems from the MIPLIB (Bixby et al., 1996; Achterberg et al., 2006). The random problem sets are used for both learning (steps 1 and 2) and assessing (step 3) the

branching strategies, whereas MIPLIB problems are only used for assessment (step 3). The reasons for using random problems are two-fold.

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

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

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

**Random problems** We randomly generate three sets of binary and mixed (binary-)integer minimization problems that each contain two different types of constraints. The possible constraints are chosen among set covering (SC), multi-knapsack (MKN), bin packing (BP), and equality constraints (EQ). We generate problems that contain constraints of type BP-EQ, BP-SC, and MKN-SC. The number of variables, the number of constraints, and the values of the elements in the matrices $\boldsymbol{c}$, $\boldsymbol{A}$, and $\boldsymbol{b}$ are randomly generated. More specifically, we first arbitrarily set some bounds on the number of variables, on the number of constraints, and on the elements contained in the matrices $\boldsymbol{c}$, $\boldsymbol{A}$, and $\boldsymbol{b}$. Then, we generate the parameters defining the problem with a uniform distribution between the chosen bounds.

The number of variables in these problems is of the order of a couple of hundreds and the number of constraints is of the order of one hundred. As some of those problems are going to be used to generate the training set, we randomly split each problem family into a 'train' and

a 'test' set. In the end, we have six problem sets: 'BPEQ_train', 'BPEQ_test', 'BPSC_train', 'BPSC_test', 'MKNSC_train', and 'MKNSC_test'. The test sets contain 50 problems each, while the train sets each contain 25. Tables 4.4 and 4.5 summarize statistics about the randomly generated problems. More specifically, those tables respectively contain the bounds on the number of variables and on the number of constraints of the problems belonging to each randomly generated problem set. Remember that, as explained in the previous section, the generated problems are intentionally kept small. The problem sets are available online[3], or can be provided upon request.

**MIPLIB** We also compare the proposed branching strategy to other branching strategies on a problem set composed of benchmark problems from MIPLIB3 (Bixby et al., 1996) and MIPLIB2003 (Achterberg et al., 2006). We eliminated the non-binary problems and the problems that were too big in order to match the size of the randomly generated problems (training problems). The set finally contains 44 problems. Table 4.6 lists the test problems taken from MIPLIB3 and MIPLIB2003.

### Experimental setup

The computations have been performed on a 16-core computer, equipped with two Intel Xeon E5520 (2.27GHz, 8 cores, and 8MB cache) and 32GB RAM, running CentOS 5.4 and CPLEX 12.2. We ran our experiments on our selection of the MIPLIB problems and on the problem sets 'BPEQ_test', 'BPSC_test', and 'MKNSC_test', described in Section 4.4.2. The training set $D_{\text{sb}}$ is generated from the problem sets 'BPEQ_train', 'BPSC_train', and 'MKNSC_train'.

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

The generated training set $D_{\text{sb}}$ contains around $7 \times 10^7$ learning examples. This number is too large and we thus use only $10^5$ of them, randomly selected without replacement from the original dataset $D_{\text{sb}}$. The chosen parameters of ExtraTrees are $N = 100$, $k = |\boldsymbol{\phi}|$, and $n_{\min} = 20$. More details about the parameters of the ExtraTrees can be found in the appendices (see Appendix A.2). In this setup and on the considered computers (without parallelization), learning the branching heuristic takes around 6 minutes. Naturally, this time can be amortized if the same heuristic is used to optimize several problems and even

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

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

Table 4.4: Randomly generated problem sets. 'all', 'bin', and 'cont' indicate the total number, and the number of binary and continuous variables in the problems, respectively.

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

Table 4.5: Randomly generated problem sets. 'all', 'EQ', 'BP', 'SC', and 'MKN' specify the total number, and the number of equality, bin packing, set covering, and multi-knapsack constraints in the problem sets, respectively.

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

Table 4.6: List of problems (44) from MIPLIB3 and MIPLIB2003.

reduced by slightly changing the parameters of the ExtraTrees (at the potential expense of performance).

Two types of experiments are performed: one where the optimization is stopped early based on a limit either on the number of explored nodes or on the time spent, and another one where the problems are solved until optimality however long it takes. For the first experiment, the rationale behind the two considered limits is to evaluate different aspects of the branching strategies. When the optimization is limited by the number of explored nodes, we can compare the branching strategies based on the closed gap[4] and on the time spent to actually explore that given number of nodes. This sheds some light on how good a branching strategy is compared to other branching strategies. In these conditions, FSB is usually the best in terms of closed gap and the worst in terms of time spent. On the other hand, the time limit is useful to assess different strategies in practical conditions where the number of nodes matters less than the time required to solve a problem. The closed gap is also used in that experiment to assess how far from the optimum the optimization is after a given amount of time. In this case, FSB is typically outperformed, in terms of closed gap, by other strategies. The second experiment (without limit) is used to assess all branching strategies in a practical situation where the problems need to be solved to optimality.

### 4.4.3 Results

We now present a selection of results comparing our approach to other branching strategies (random, MIB, NCB, FSB, and RB). Tables 4.7, 4.8, and 4.10 report these results. In these tables, 'Cl. Gap' refers to the closed gap, and 'S/T' indicates the number of problems solved within the provided nodes or time limit, versus the total number of problems. 'Nodes' and 'Time' respectively represent the number of explored nodes and the time spent (in seconds) before the optimization either finds the optimal solution, or stops earlier because of one stopping criterion. Those values are measured separately for each problem and are averaged in the tables.

In addition to the optimization results reported here, we also present some results related to the learning aspects of our method. These results are given in Section 4.6.

#### Experiments with limits

Table 4.7 first shows the results achieved on the random test problem sets for both stopping criteria. Those results show that our approach succeeds in efficiently imitating FSB. Indeed, the experiments performed with a limit on the number of nodes show that the closed gap is only 9% smaller, while the time spent is reduced by 85% compared to FSB. The experiments with a time limit show that the reduced time required to take a decision allows the learned strategy to explore more nodes, and to thus further close the gap than FSB. While these results are encouraging, they are still slightly worse than the results obtained with RB, which is both closer to FSB and faster than our approach.

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

In addition to the previously presented branching strategies, Table 4.7 contains one extra experiment for each family of random problems. The normal learned branching strategy, i.e., ($n_{\min} = 20$, all), is learned based on a dataset containing samples from the three types of training problems, i.e., BPSC, BPEQ, and MKNSC. We also investigate, for each type of random problem, the effect of learning the branching strategy from a dataset generated with training problems of the same type as the target test problems. In other words, when we test this strategy on the BPSC test problems, the branching rule is learned based on samples generated with BPSC train problems only, which is indicated in the table by ($n_{\min} = 20$, BPSC only). Overall, the results show that the strategies learned only on the type of problem on which they are tested perform a bit better than the strategy learned from a dataset containing a mixture of the three types of problems. This indicates that the approach can benefit from learning on a specific problem type and that the performance of the learned branching strategy improves when the problems to optimize are aligned with the problems that are used to generate the training set.

Table 4.8 then shows the results obtained with a node limit and a time limit on the MIPLIB problems. For these experiments, we separated the problems that were solved by all methods from the problems that were not solved by at least one of the compared methods. Similarly to the results obtained on the random problem sets, the proposed branching strategy compares favorably with strong branching both on the node limit and time limit experiments. Nonetheless, the results obtained with the learned branching strategy are still a little below the results obtained with reliability branching. The results presented here are averaged over all considered problems. The detailed results for all problems in the MIPLIB set are available in Appendix A.3.

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

Table 4.7: Optimization results for the random problems. 'Cl. Gap' refers to the closed gap, and 'S/T' indicates the number of problems solved within the provided nodes or time limit, versus the total number of problems. 'Nodes' and 'Time' respectively represent the number of explored nodes and the time spent (in seconds) before the optimization either finds the optimal solution, or stops earlier because of one stopping criterion.

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

Table 4.8: Optimization results for the MIPLIB problems of Table 4.6. 'Cl. Gap' refers to the closed gap, and 'S/T' indicates the number of problems solved within the provided nodes or time limit, versus the total number of problems. 'Nodes' and 'Time' respectively represent the number of explored nodes and the time spent (in seconds) before the optimization either finds the optimal solution, or stops earlier because of one stopping criterion.

**Experiments without limits**

Finally, Tables 4.9 and 4.10 report the results form our last set of experiments. We apply all branching heuristics on all MIPLIB problems initially contained in Table 4.6, and let the computers solve the problem for five days. After this time limit, the problems that are not solved by all branching methods are discarded. We create thus a new set of MIPLIB problems (reported in Table 4.9) that is used to compare different branching strategies until the end of the optimization procedure. Additionally, in another experiment, we let CPLEX use cuts and heuristics (with default CPLEX parameters) in the course of the optimization in order to observe their impact on the efficiency of each branching strategy. The optimization results are shown in Table 4.10.

Overall, our method compares favorably to its competitors when cuts and heuristics are used by CPLEX. Indeed, in that case, our learned branching strategy is the fastest (almost three times faster than the second fastest method, i.e., MIB) to solve all the 30 considered problems. Note that the apparent bad results of RB are due to three problems that are especially hard for that branching heuristic (air04, air05, and mod011). If we remove them from the computation of the average optimization time, both RB and the learned branching strategy take 74 sec on average to solve the remaining 27 problems. That average time is still 40% smaller than the average time of the runner-up (MIB). These experiments show that our strategy behaves very well when cuts and heuristics are used by CPLEX to solve the problems. The detailed results are available in Appendix A.3.

Things appear to be different when cuts and heuristics are not used. Indeed, based on the results of Table 4.10, our method seems to be very slow, but the large number of nodes and the large amount of time is actually due to a small number of problems for which the method does not work well. These problems artificially increase the average number of nodes and average amount of time reported in the table. A finer analysis can be conducted by interpreting the detailed results reported in Appendix A. When done, we see that our method is faster than RB in 11/30 cases and faster than FSB in 21/30 cases, thus alleviating the a priori bad performance of the learned branching strategy. A possible explanation for why our method does not perform well on those problems can be that these problems, because too large, are not well represented in the dataset that we use in order to learn the branching strategy. This shows the importance of considering a large and diverse training set for the learning of an efficient branching strategy.

### 4.4.4 Discussion of the proposed method

We describe, in this section, a new approach to design branching strategies for MILP problems. It consists in observing branching decisions taken by a supposedly good strategy, FSB in this case, and to imitate those decisions with a strategy obtained through offline supervised machine learning techniques. The learned strategy can then be used in place of the usual branching strategies. The experiments show promising results.

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

Table 4.9: Updated list of problems from MIPLIB3 and MIPLIB2003. This list contains the problems from Table 4.6 that are solved to optimality with each branching heuristic in less than five days.

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

Table 4.10: Optimization results (until termination) for the updated list of the MIPLIB problems (Table 4.9). 'Cl. Gap' refers to the closed gap, and 'S/T' indicates the number of problems solved within the provided nodes or time limit, versus the total number of problems. 'Nodes' and 'Time' respectively represent the number of explored nodes and the time spent (in seconds) before the optimization either finds the optimal solution, or stops earlier because of one stopping criterion.

**Comparison with other branching strategies**

The idea behind our approach can be seen as a combination of the ideas behind reliability branching and information-based branching. On the one hand, our approach is similar to reliability branching in the sense that we want to find a fast proxy to strong branching. However, the approaches differ in the means by which the information is collected, and in how the information is used. On the other hand, our approach is similar to information-based branching in the sense that it tries to use general information collected during a preliminary phase in order to speed up the current optimization. In both cases, the idea is to obtain a broad overview of the B&B process and to use this overview to take sensible branching decisions. The main difference lies in the way information is collected: information-based branching harvests information through multiple restarts on the same problem, while our method collects optimization information obtained through the optimization of a set of different optimization problems. Another important difference is that, in the case of information-based branching, the collection phase needs to be performed for each problem to optimize, while our approach only requires the information to be collected once. In the light of Table 4.7, it is to be noted, however, that customizing the branching strategy to the problem being optimized could supposedly yield better results, which would be in favor of the approach that involves collecting some data for each problem to solve.

**Connections with the no free lunch theorems**

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

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

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

## 4.5 Online learning of strong branching decisions

The second approach that we develop to speed up strong branching proposes to use online machine learning techniques in place of offline algorithms. The advantage of online learning techniques relies on the fact that they do not require a full dataset in order to train a model. They rather build the model incrementally as the data arrives. We refer the reader to Section 3.1.2 for more information about online learning.

In an attempt to push further the method introduced in Section 4.4, we present an approach that combines the ideas behind the branching strategy developed in the previous section, i.e., 'learned branching', and reliability branching. More specifically, this method uses online learning algorithms to learn a function that approximates strong branching (SB) and that is both fast to evaluate and accurate enough. The idea is similar to the one introduced by learned branching in the sense that the goal is to take SB-like decisions without the computational cost induced by the real SB. However, in this case, the learning is performed in an online

manner, during the course of the optimization, which alleviates some important shortcomings of the previous approach. Indeed, since the learning is made online, no preliminary phase is required in order to generate a dataset nor to learn the proxy. No computing time is thus wasted in such a step. Additionally, the learned function is really tailored to the studied problem rendering previous machine learning concerns (like, for instance, some requirements on the features) obsolete.

Similarly to the batch learning case, this section details the method that we set up to leverage online learning techniques in order to learn a proxy to strong branching. We successively describe the method, the experiments, and the results and finally conclude with some discussion of the method.

### 4.5.1 Learning branching decisions in an online setting

As already discussed in some detail, strong branching (SB) (Applegate et al., 1995) is a very popular branching heuristic that selects a branching variable by explicitly computing the dual bound improvements for each candidate. Let us recall that SB is known to perform very well in terms of the number of nodes of the resulting B&B tree, but requires a very large amount of computational effort to take a decision. In order to alleviate this problem, the method that we propose learns, during the course of the optimization, a function that approximates SB and that can be evaluated more quickly. The approximation of SB is obtained with a very simple online learning technique, namely (online) linear regression. Moreover, the method that we develop uses a reliability mechanism that is similar to the one used by reliability branching (Achterberg et al., 2005).

More specifically, we propose to use the function $\mathcal{B}_{\text{olb}}$ to select the branching variable among the set of candidate variables. The function $\mathcal{B}_{\text{olb}}$ is trained with a linear regression algorithm, which assumes that the output is obtained through an inner product between the inputs (i.e., the features) and a vector of parameters $\boldsymbol{w}$, i.e.,

$$\mathcal{B}_{\text{olb}}\left(i, \boldsymbol{\phi}_i\right) = \boldsymbol{w}^\top \boldsymbol{\phi}_i,$$

where $\boldsymbol{\phi}_i$ is a feature vector that represents variable $i$ at the given node of the B&B tree. Unlike the batch method, however, the function $\mathcal{B}_{\text{olb}}$ (more specifically the parameter vector $\boldsymbol{w}$) is not obtained through a preliminary learning phase, but trained during the optimization. As in the batch case, the function $\mathcal{B}_{\text{olb}}$ must be fast to evaluate and close enough to the real SB function, i.e.,

$$\mathcal{B}_{\text{olb}}\left(i, \boldsymbol{\phi}_i\right) \approx \mathcal{B}_{\text{sb}}\left(i, \boldsymbol{c}, \boldsymbol{A}, \boldsymbol{b}, \boldsymbol{x}', \boldsymbol{l}', \boldsymbol{u}'\right).$$

Note that the features used to describe a variable are similar to the ones used in the batch case (see Section 4.4) and are described in Section 4.3. There is one minor difference though: features 41-43 are bounded above by $10^4$ because linear regression is not suited when the features can take very large values. Features 41-43 are thus, in the online case, given by

$$\min\left(1e4, \left[\text{ down; up; down+up Driebeek penalties normalized by } o' \right]\right).$$

In this application, we only limit the magnitude of three features because the other features remain at acceptably low values. It is possible though that, in other contexts, other features take large values that could hinder the efficiency of linear regression. In general, it is good to ensure that the features do not take too large values.

### Description of the online learning branching (OLB) strategy

Technically speaking, the branching strategy that we propose (OLB) is as follows. When B&B needs to branch, a set of candidate variables is first selected among all fractional variables at the current node. A score is then computed for each variable in that set. The variable with the largest score is then chosen as branching variable.

The branching scores can be computed in two ways. On the one hand, if the function $\mathcal{B}_{\text{olb}}$ is deemed unreliable for the candidate variable, the real SB score is computed, together with a set of features describing that variable at the current node. The computed score is used to rank the candidate variable, but it is also stored in a dataset (with the computed features) to improve the approximation of the SB proxy. On the other hand, if the function $\mathcal{B}_{\text{olb}}$ is deemed reliable, the features describing the current candidate are computed and fed to the approximated version of SB, i.e., the function $\mathcal{B}_{\text{olb}}$ learned with linear regression, in order to quickly generate an approximate SB score, hopefully close to the real SB score.

The real SB scores that are generated when the approximation cannot be trusted are used to train a simple linear regression, whose goal is to predict approximate SB scores[5]. The learning is performed in an online fashion with a simple gradient descent algorithm (using line search) to allow the approximation to evolve during the course of the optimization. In order to determine whether the approximation for a given variable is reliable or not, we simply count the number of samples (i.e., computed real SB scores and features) that have been generated previously for each variable. This mechanism is comparable to the reliability mechanism of reliability branching (Achterberg et al., 2005).

The complete description of our branching strategy is given in Algorithm 2. The proposed method requires four main parameters. First, $\lambda \in \mathbb{N}_0^+$ controls the number of variables that can be considered as branching candidates at each iteration. Second, $\eta \in \mathbb{N}_0^+$ indicates how many samples are required in order to trust the approximation for a specific variable. Lastly, $\delta \in \mathbb{R}_0^+$ and $\sigma \in \mathbb{R}_0^+$ are used to limit the convergence of the gradient descent algorithm in order to avoid undesirable oscillations (cf. Algorithm 2, line 32).

### Improvements of OLB

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

[5]Similarly to the batch case (see Section 4.4.1), we do not predict exactly the SB scores, but rather predict the logarithm of the relative SB scores.

the stored features, to the learning queue in order to be processed by the learning algorithm. Using this trick allows the branching strategy to adapt over time, even if the dynamics of the problem for a given variable change.

### 4.5.2 Experiments

We assess the efficiency of online learning branching (OLB) by comparing different branching strategies on a selection of problems from MIPLIB (Bixby et al., 1996; Achterberg et al., 2006), listed in Table 4.6. These problems are the same as the ones used to assess the performance of learned branching $\mathcal{B}_{\text{learned}}$.

The experiments consist in optimizing the selected problems while plugging in different branching strategies. Each problem is optimized 10 times with 10 different random seeds (from 0 to 9). We impose a time limit of 7,200 seconds for each optimization.

The experiments are carried out on a 16-core computer, equipped with two Intel Xeon E5520 (2.27GHz, 8 cores) and 32GB RAM, running CPLEX 12.6. We disable presolve in CPLEX but leave the default values for the other parameters (except for the seed). Additionally, we disable parallelism, i.e., only one core is made available for each run of CPLEX. We compare our approach to three other branching strategies, namely full strong branching (FSB) (Applegate et al., 1995), reliability branching (RB) (Achterberg et al., 2005), and learned branching (Learned) (see Section 4.4). The default parameter values $\lambda = 4$ and $\eta = 8$ are used for RB (Achterberg et al., 2005). For FSB, the SB LP-relaxations are solved to optimality and there is no limit on the number of candidate fractional variables at each node. For learned, we use the default parameters (see Section 4.4), i.e., $N = 100$, $k = |\boldsymbol{\phi}|$, and $n_{\min} = 20$. The parameters for our methods, online learning branching (OLB) and online perpetual learning branching (OPLB), are $\lambda = 4$, $\eta = 8$, $\delta = 0.01$, and $\sigma = 500$.

### 4.5.3 Results

We use performance profiles to compare the considered branching strategies. The performance profiles are drawn considering that the pairs composed of a problem and a seed are a single problem. Consequently, the performance profiles are constructed with 440 problems (every combination of problem and seed). In addition to the performance profiles, the complete experimental results are reported in Appendix A.4.

The performance profiles (Dolan and Moré, 2002) report the probability that a solver solves a problem vs. a performance ratio. A point $(x, y)$ on a performance profile curve should be understood as 'there is a probability $y$ that the method solves a problem if it is given at most $x$ times as much budget as the best solver needs to solve the problem'. The leftmost part of the performance profiles indicates how good the solvers are, i.e., how fast they solve the problems in terms of the chosen metric (time or number of nodes, in this case). On the other hand, the rightmost part of the graphs is usually an image of the robustness of a method, i.e., this part indicates the proportion of problems that will eventually be solved by a method if enough time is granted.

**Algorithm 2** Online learning branching (OLB). Note that all variables are assumed to be global.

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

**Parameters:** $\lambda$, $\eta$, $\delta$, $\sigma$

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

Figure 4.1(a) illustrates the performance of the methods in terms of time, while Figure 4.1(b) focuses on the number of nodes required to solve the problems. The results show that FSB behaves as expected: very good in terms of nodes, but rather slow in general. The proposed method (OLB, OPLB) compares favorably to the other methods. It takes rather good branching decisions and is shown to be very fast in the beginning of the optimization (better than the others when the time ratio is less than ≈3.5). Learned branching is dominated by OLB and OPLB in the beginning but catches up quite quickly[6] (the ratio of solved instances exceeds the other methods when a large enough amount of time is provided). Learned is also very effective in terms of nodes. Overall, RB is dominated by the other approaches both in terms of time and in terms of nodes. Finally, we note that learning perpetually (OPLB) does not improve significantly the performance of the normal online learning approach (OLB).

In addition to the optimization results reported here, we discuss some learning aspects of the proposed method in Section 4.6.

### 4.5.4 Discussion of the proposed method

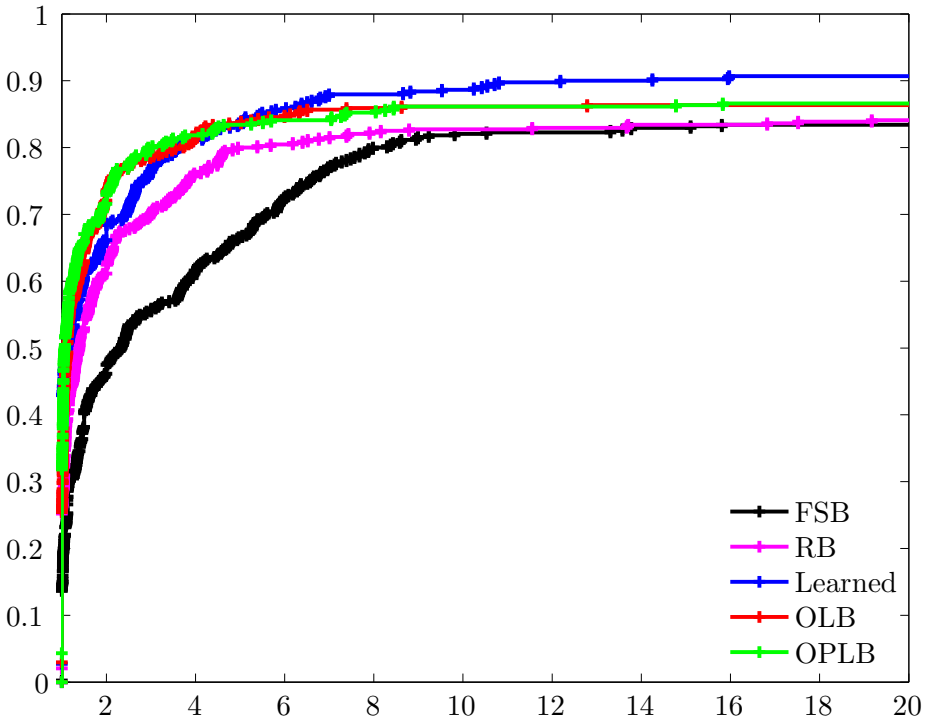
The online branching strategy described here constitutes a slight improvement over the approach proposed in the previous section. The goal of this online technique remains unchanged: find an approximation of strong branching that is fast to evaluate. The difference between both methods lies in how the approximation is obtained. In the former method, a dataset is first generated and then fed to a learning algorithm in order to train a model with the available data. This approach requires that some (possible large) amount of time is devoted to the generation of the training set and to the preliminary learning phase. The online approach alleviates this problem by using online learning techniques that do not require that a full dataset is available from the beginning. The model is instead trained online, i.e., on the fly, as the data arrives.

In addition to this advantage, the proposed online branching strategy is similar in its mechanisms to reliability branching. The method is meant to work with a supposedly good branching strategy (full strong branching in this case). During the branching procedure, if the approximation for a candidate variable is deemed unreliable, the good branching strategy is used, which increases the confidence level of the approximation for that variable. At some point in time, the variable can be trusted and, from that moment on, the approximation of the score generated by the good strategy is used instead of the real score.

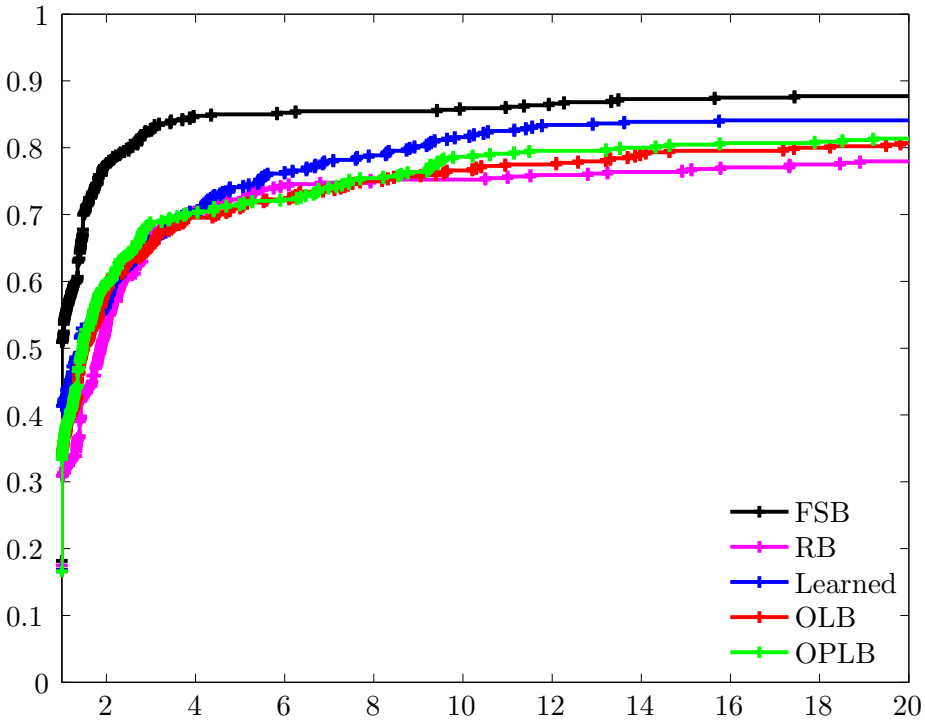
The experimental results show that our approach compares favorably to other branching strategies both in terms of time and in terms of nodes and outperforms reliability branching in both metrics. It is interesting to note though that learned branching outperforms OLB and OPLB when the performance ratio (time and number of nodes) increases. This result is against the intuition since one could have expected that online learning can tailor the strong branching approximation to the optimized problem more easily than a batch learning algorithm. A possible explanation for this behavior could be that the online learning procedure is damped too much, i.e., the parameters $\delta$ and $\sigma$ are too small.

[6]It is to be noted that the time required to train the 'Learned' branching strategy $\mathcal{B}_{\text{learned}}$ is not taken into account in the performance profiles.

Finally, note that, in this case, the features design is somewhat less critical than in the batch learning case. Indeed, in the batch case, the features constitute a common representation of any variable of any problem. The features must, in a sense, lie in a common space that allows the comparison between two variables whatever the problem. In the online case, this is not required anymore and the features can be designed specifically for a problem (in particular, they can be of different dimensionalities for different problems), since no information is carried over from one optimization to the next.




(a) Performance profile in terms of time




(b) Performance profile in terms of nodes

Figure 4.1: Performance profiles for the competing methods. The detailed experimental results used to draw the performance profiles are reported in Appendix A.4.

## 4.6 Analyzing learning performance

This section reports some results related to the learning aspects of our methods.

### 4.6.1 Learning accuracy

We first assess the performance of our learned functions to determine how close from the real scores the predicted values are. More specifically, we use the average and the median of the relative errors to check the accuracy of the methods. For a single real score and the corresponding prediction, the relative error is computed as

$$\text{RE} = \frac{|y - \hat{y}|}{\max\left(1e^{-3}, |y|\right)},$$

where $y$ and $\hat{y}$ denote the real score computed with the strong branching procedure and the prediction obtained with our learning techniques, respectively. We add the maximum in the computation of the error to ensure that the denominator is different from 0 and not too small.

The assessment procedure is as follows. From the original dataset generated in Section 4.4.2, two subsets are randomly sampled (without replacement). The first subset is a training set and the second one is a test set. The original dataset contains around $7 \times 10^7$ learning examples and is thus rather large. In order to ease the experiments, we only use a fraction of it. More specifically, the training and test set each contain $10^6$ (distinct) samples, which are randomly sampled from the original dataset. The assessment procedure then consists in training a model on the training set and in computing the relative errors induced by the model on the test set. More specifically, the inputs in the test set are used as inputs of the learned models and the corresponding outputs given by the models, i.e., the predictions, are then compared to the real outputs stored in the dataset. The relative errors are computed according to the previous formula and the learning algorithms can then be compared based on the average and median relative errors that they induce on the test set.

In this work, we learn a function that approximates strong branching with both ExtraTrees and linear regression, in the batch and online case, respectively. Although ExtraTrees typically yield better performance than linear regression, the latter is used, in the online version of the proposed approach, to train the model. This is due to the fact that ExtraTrees cannot be easily adapted to the online setting, while linear regression can. Because both learning algorithms are used, we evaluate the performance of both methods on the available data. Note that, strictly speaking, online linear regression is used in our approach, but we evaluate linear regression in the batch case. The reason is that it is typically harder to evaluate online algorithms because several issues arise (e.g., not enough test data) and dealing with all of them to fairly assess the method implies a certain amount of additional work. For the sake of convenience, we thus use the batch setting for linear regression as well. This does not impact interpretability but two aspects must be highlighted. First, batch linear regression typically performs better than online linear regression (the reported results may thus be overly optimistic). Second, the online version of our approach is applied to a single problem, i.e., the SB proxy is learned for the considered problem only. This learning task is likely to be easier than the batch approach that takes into account many different problems at the same time

and that is used to assess linear regression in batch mode. There is thus a second effect not accounted for here, but this effect tends to negatively impact the reported performance, i.e., tends to increase the reported errors. Because of these two limitations, one must be cautious when analyzing the obtained results, but using batch linear regression is, in any case, useful to better understand the dynamics of the problem and to validate the choice of that algorithm to train an online proxy to SB. In addition to the two mentioned learning algorithms, we also apply a third one to the training and test sets, namely the Lasso method (Tibshirani, 1996). The Lasso is basically a linear regression that pushes the less useful regression coefficients to 0. This method is helpful to determine which features are the most important to predict a correct output. To apply the Lasso, one needs to determine the value of one parameter $\alpha$ that controls how 'hard' the features are pushed to 0.

In these experiments, we use the same parameters for the ExtraTrees as in the rest of the chapter, i.e., $N = 100$, $k = |\boldsymbol{\phi}|$, and $n_{\min} = 20$. Batch linear regression does not have any parameter and, regarding the Lasso, several values of the parameter $\alpha$ are considered: $\alpha = \{0.01, 0.1, 0.5, 1\}$.

The learning results are reported in Table 4.11 and Figure 4.2. First, Table 4.11 indicates that, even if the ExtraTrees are a bit ahead, the three learning algorithms are able to approximate fairly well the SB score. Indeed, a relative error of 16% and 22% is achieved on average on the predictions with the ExtraTrees and linear regression, respectively. Additionally, the median shows that for 50% of the samples, the prediction error is less than 6% and 8%, respectively. This indicates that the predictions achieve a quite good accuracy in general with only a limited number of features and very simple learning algorithms (without any parameter tuning). Figure 4.2 next shows a histogram of the relative errors achieved on the test set with the ExtraTrees. The graph shows that, for a large fraction of the test set, the relative errors are rather small. More precisely, 50% of the samples are predicted with a relative error less than 5.97% (the median), and the relative error is less than 20% for 90.62% of the test data. In addition to the prediction accuracy, we report in the table the Spearman correlation coefficient that measures the rank correlation between the predictions and the real scores. The numbers show that the predictions are highly correlated with the real scores with coefficients of 0.94 and 0.89 for the ExtraTrees and linear regression, respectively. This high correlation implies that, in many cases, it is likely that the variable ranking provided by the learned strategy is similar to the ranking provided by strong branching.

Finally, it is important to emphasize that the learning accuracy is not crucial here. Indeed, the main purpose of the approach is to quickly approximate the strong branching behavior and not to predict with infinite accuracy the strong branching scores. Obviously, being able to estimate those scores accurately is a key factor to success, but a fair assessment of the method must also consider the optimization results obtained with the approach. Generally speaking, in order to learn strong branching, it is more important to learn the candidate variables ranking rather than the true scores. The reported results show that our approach succeeds in that aspect quite well.

| | | | Lasso | | | |
|---|---|---|---|---|---|---|
| | ET | LR | 0.01 | 0.1 | 0.5 | 1 |
| RE: mean | 0.1604 | 0.2207 | 0.2196 | 0.2281 | 0.2347 | 0.2488 |
| RE: median | 0.0597 | 0.0828 | 0.0833 | 0.0902 | 0.1001 | 0.1056 |
| Spearman corr. | 0.9398 | 0.8914 | 0.8903 | 0.8626 | 0.8411 | 0.8328 |

Table 4.11: Means and medians of the relative errors, and the Spearman correlation coefficients achieved on the test set by ExtraTrees (ET), linear regression (LR), and the Lasso (with different values of the main parameter of the Lasso).

### 4.6.2 Important features

Besides the accuracy of the learning methods, we also analyze the importance of each feature on the ability of the learned function to predict a correct output. There are several ways to assess the feature importance.

A first possibility consists in using the outcome of the learning procedure to determine which features matter most. For instance, in linear regression, one could assume that the larger the absolute value of a regression coefficient, the more important the corresponding feature is. This is true to some extent in the sense that the regression coefficient of a useless variable can be set to 0 without impacting the prediction. However, it is highly unlikely that the regression coefficients corresponding to meaningless features are exactly equal to 0. The learning algorithm will always try to learn a bit (even noise) from the data for all features and even useless features will consequently have non-zero coefficients. Moreover, the amplitudes of the regression coefficients depend on the amplitudes of the features. If a meaningless feature happens to take very small values, it is possible that, in order to 'learn something', the learning algorithm will set the regression coefficients to large values. Similarly, a very small regression coefficient may correspond to a very important feature that takes large values. This implies that, though interesting, the regression coefficients may not be trusted entirely when it comes to feature importances. On the other hand, the Lasso method is designed in such a way that meaningless features can be dealt with. Indeed, the Lasso encourages the regression coefficients to be 0. This alleviates a bit the previous concerns because it implies that the regression coefficients of useless features will naturally fade away. However, if the regression coefficient of an important feature is close to 0, the algorithm may push that coefficient to 0 as well thus depriving the prediction from an important component. This method is thus not exempt from flaws either. Finally, the ExtraTrees possess a nice characteristic that computes, during the learning procedure, so-called feature importances for each input feature. These feature importances are similarly helpful to understand the input-output correlations in the dataset.

A second possibility to estimate the impact of a feature on the prediction ability is to compute the cost of omission (COO) (Leyton-Brown et al., 2009). The cost of omission represents the cost, in terms of the error, that omitting a given feature during the learning and the testing phases induces. More specifically, the learning and the testing phases are applied as usual on the same data from which the considered feature is eliminated. The mean relative error (MRE) without the considered feature can then be estimated from the test set. The difference between the MRE obtained without the feature and the MRE obtained with

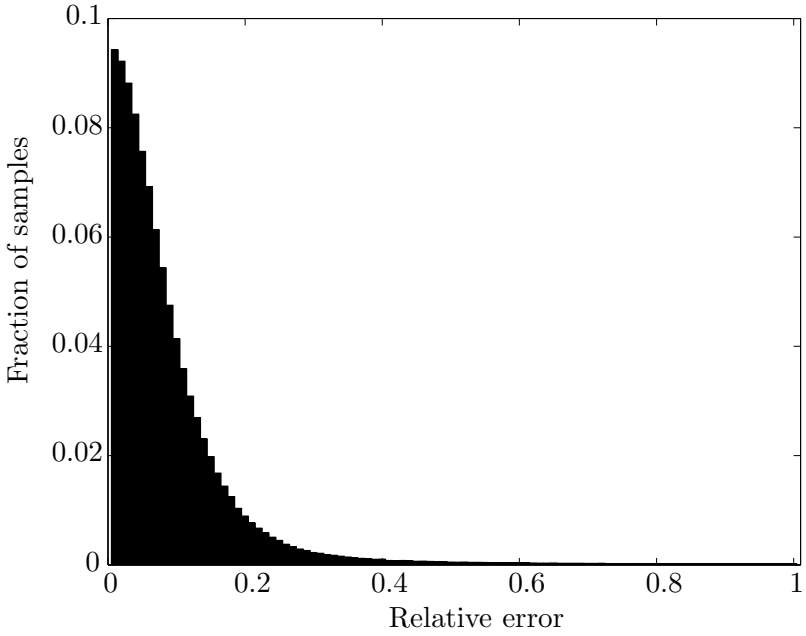



Figure 4.2: Histogram of the relative errors achieved on the test set with the ExtraTrees. The graph is limited to 1 here for readability reasons. Note that there are some samples in the test set for which the relative error is larger than 1, but their number is limited. More precisely, the relative error is larger than 1 (i.e., 100%) for only 1.34% of the test set.

all features is an image of how important the feature is. If the value of the COO is positive, this means that the feature is important for the prediction. On the other hand, when the COO is negative, it implies that the feature has a negative impact on the prediction accuracy. Small values of the COO (either positive or negative) indicate that the feature is not very important and could just be a source of noise in the prediction. Note that we use in this work the normalized COO which consists in attributing to the largest COO a value of 100, all other COOs being scaled accordingly.

The two possibilities detailed above can, to some extent, provide information about the importance of the features taken individually. This information can be used to improve the understanding of the dynamics of the problem but should not be regarded as a conclusive analysis. Moreover, the potential interactions between the features are not taken into account with either method and this (important) aspect is thus totally omitted by such evaluation procedures.

Table 4.12 reports the feature importances (for the 10 most important features) as obtained through the learning procedure (feature importances for the ExtraTrees and amplitude of the regression coefficients for the other two methods), the MRE obtained without the considered feature, and the corresponding COO. The conclusions that can be drawn from the table are twofold. First, feature # 37, i.e., the fractionality of the candidate variable, seems to be an important feature, although removing that feature does not impact very negatively the performance. Second, none of the other features seems to be important according to the feature importances nor according to the COOs. This is of course misleading. This leads us to think that what matters in this situation is not really the individual features themselves, but the

interactions between those features. Further investigating how the features are combined in order to predict a good output may largely improve the understanding of the branching procedure. The complete list of feature importances is available from Appendix A.1 in Tables A.1 and A.2.

| | ET | | | LR | | | Lasso ($\alpha = 0.01$) | | |
|---|---|---|---|---|---|---|---|---|---|
| # | FI | MRE | COO | FI | MRE | COO | FI | MRE | COO |
| 2 | 0.2875 | 0.1591 | -4 | 0.0000 | 0.2078 | -262 | 0.0000 | 0.2141 | -38 |
| 27 | 0.1297 | 0.1596 | -3 | 0.0000 | 0.2077 | -263 | 0.1213 | 0.2141 | -38 |
| 37 | 0.0497 | 0.1892 | 100 | 0.0000 | 0.2257 | 100 | 1.0000 | 0.2341 | 100 |
| 40 | 0.0486 | 0.1552 | -18 | 0.0000 | 0.2078 | -261 | 0.0451 | 0.2141 | -38 |
| 31 | 0.0449 | 0.1587 | -6 | 0.0000 | 0.2078 | -262 | 0.2423 | 0.2150 | -32 |
| 18 | 0.0413 | 0.1569 | -12 | 0.0000 | 0.2077 | -263 | 0.0009 | 0.2117 | -55 |
| 63 | 0.0289 | 0.1574 | -10 | 0.0000 | 0.2106 | -204 | 0.1037 | 0.2157 | -27 |
| 56 | 0.0287 | 0.1571 | -11 | 0.0000 | 0.2073 | -272 | 0.0000 | 0.2141 | -38 |
| 57 | 0.0286 | 0.1580 | -8 | 0.0000 | 0.2182 | -50 | 0.6523 | 0.2235 | 27 |
| 58 | 0.0273 | 0.1562 | -14 | 0.0000 | 0.2084 | -249 | 0.0052 | 0.2133 | -44 |

Table 4.12: Feature importances (of the 10 most important features) as computed by the ExtraTrees (ET), linear regression (LR), and the Lasso. Each row of the table corresponds to a feature, whose number is given in the first column. 'FI' represents the feature importance for the corresponding feature. For the ExtraTrees, the feature importances are the result of the internal procedure run during the learning phase. For LR and the Lasso, the feature importances correspond to the (normalized) absolute values of the regression coefficients. Note that, in the case of LR, a few coefficients are very large (order of $10^7$) and thus the relative importance of the other features is seemingly null. 'MRE' represents the mean relative error obtained on the test set when the considered feature is eliminated from learning and testing. The COOs are obtained by comparing the MREs obtained without the features and the values reported in Table 4.11. The complete list of feature importances is available from Appendix A.1 in Tables A.1 and A.2.

## 4.7 Concluding remarks and outlooks

In this chapter, we proposed a new approach to design branching strategies for MILP problems. It consists in observing branching decisions taken by a supposedly good strategy, strong branching in our case, and to imitate those decisions with a strategy obtained by a machine learning procedure. To this end, we develop a set of features that are used to characterize the current state of the problem in the B&B tree from the perspective of a particular variable. These features are computed for all candidate variables and used as input of the learned branching heuristic in order to predict an approximation of the strong branching score for that variable. We propose two approaches that use machine learning: a batch approach and an online approach. In the batch case, a dataset must be generated prior to the optimization and fed to a machine learning algorithm in order to train a model on the data that approximates strong branching. In the online case, it is not necessary to generate a dataset, since the online learning algorithm trains the model during the course of the optimization. The experiments performed to assess the efficiency of both methods show promising results and suggest that further research in this direction may lead to favorable improvements in MIP solvers.

The underlying mechanism of the developed approaches is not different from other branching strategies. Indeed, in all cases, features are computed from the current state of the problem (in some way or another) and then used to decide which variable to branch on. Those features can be, e.g., the fractionality of the variable in the current solution (used in most-infeasible and reliability branching), the pseudocosts of the variables (used in reliability branching), or the objective increases observed when the branching is performed (used in strong branching). In our approach, however, we can include many types of features, including those used by popular strategies. The difference lies in how these features are used. While, in traditional methods, the features are assumed to explain everything and are all used to take the branching decision, we let the learning algorithm decide which features are relevant and which are not. Indeed, learning algorithms are able to sort out which of those features are useful, and are able to automatically determine how to combine them to rank branching decisions.

In that sense, we can see our method as a very general branching strategy that can imitate any other heuristic, as long as the appropriate features are provided. For example, in its current implementation, our approach can be tied to pseudocost and reliability branching. Indeed, our features include the value of the fractionality of the variables and the observed objective increases (see Section 4.3). This means that the information that pseudocost branching uses to take a decision is provided to the learning algorithm. Our approach can thus, in principle, use only the pseudocosts and the variable fractionalities to take a branching decision, if the learning algorithm decides that these features best explain the desired output. Note, however, that the desired output that we are focusing on is the strong branching score, which is different from the score used by pseudocost branching. The learning algorithm can also discover novel heuristics by combining the features used by several popular methods with novel ones, hopefully yielding a better approximation of the desired output.

Note that, although the goal of this work is to create a fast approximation of strong branching, it would be possible to use the very same approach to approximate a branching strategy that is even more expensive and effective (in terms of the number of nodes of the tree) than strong branching.

# Chapter 5

# Machine learning for parallel branch-and-bound

The contents of this chapter are mainly reproductions of one piece of work published online (Marcos Alvarez et al., 2015).

## 5.1 Introduction

At this point, there should be no need to insist on the ubiquity of branch-and-bound (B&B) and its variants in solving mixed-integer programming (MIP) problems. In the previous chapter, we focus our attention on improving one key element of B&B, namely the branching strategy, to better and faster solve optimization problems. However, despite all improvements that can been made to all components of B&B (including, e.g., cutting planes, branching strategies, and presolve), some very large MIP problems remain nowadays still too difficult to be solved by a single sequential B&B.

Parallelizing B&B on a large number of computers is a promising way to solve those problems that remain out of reach for traditional approaches. This rationale is strongly motivated by two arguments. First, B&B is a natural candidate for parallelization since it relies on the divide-and-conquer paradigm. Parallelizing B&B is indeed conceptually quite simple and mainly consists in dividing the original optimization tree in several subtrees, or subproblems, and in letting each processor, or worker, work on its own part of the global tree. Two parallel implementations mainly differ in the way the original work is split among the available workers and by the amount of communication involved in the optimization. The second argument in favor of parallel B&B is the explosion of parallel computing and affordable massively parallel computers that has been witnessed in the last two to three decades.

Based on these observations, many researchers started developing parallel B&B algorithms. One of the first reported attempts to parallelize B&B dates back to 1975 and is summarized in a 1988 paper by Pruul et al. (1988). In that paper, Pruul et al. describe a simple approach to parallelize B&B on a shared memory serial computer. They report a set of experimental results obtained on the travelling salesman problem and analyze the efficiency of their approach.

One important finding is that the number of explored nodes might be less in the parallel case than in the serial case. As a consequence, the achieved speedup computed from the number of explored nodes might be higher than the number of processors on which the B&B has been parallelized. These findings further support the idea that parallel B&B is a front-running candidate to solve large MIP problems. It has to be noted however that there exist special cases where the parallel version of B&B performs worse than its serial counterpart (see, e.g., Lai and Sahni, 1984).

Very early, balancing the load of each worker of a parallel B&B has become a major concern. Indeed, as for any parallel algorithm, load balance is a crucial aspect that must be addressed in order to achieve interesting speedups. Throughout the years, several load balancing schemes have been proposed. For instance, El-Dessouki and Huen (1980) propose a mixed static and dynamic balancing scheme that gives each processor the responsibility to compute its own subtree and that allows them to help other processors when their own workload has been exhausted. Later, Karp and Zhang (1988) proposed a fully dynamic work distribution method that automatically balances the load of each worker by sending the newly created children to random processors. Rao and Kumar (1987) also proposed several load balancing schemes tailored to different parallel architectures (see also Kumar and Rao, 1987).

A common shortcoming of all the dynamic load balancing schemes is that they imply a large amount of communication. It became very early clear that the overhead cost induced by communication times was a major concern for all parallel B&B implementations. On the one hand, communication is desirable because it allows to better balance each processor's load by ensuring that no processor remains idle while others are working. Moreover, communication can also reduce the total amount of work to be done by all processors by sharing information about feasible solutions. But, despite its advantages, communication between processors remains very expensive and should be limited to its minimum. Laursen (1994) was one of the first to propose a method in which the processors do not communicate with each other and that allocates statically the workload to each worker. Of course, the key factor to success of this approach is to evaluate accurately enough the difficulty of the subtree given to each processor. If the workload is not well balanced between the workers, the utilization of the processors will not be optimal. One of the approaches proposed by Laursen consists in finding a function that predicts the number of nodes of a subtree, i.e., its difficulty, from a set of characteristics extracted from the subtree. Laursen carried out a series of experiments with several functions constructed from simple functional forms like the exponential or the logarithm. Unfortunately, each considered function was not able to consistently predict the difficulty of several classes of problems. Laursen concluded that it was not trivial to find such a function. Later, the idea proposed by Laursen (1994) has been further explored in a more principled approach by Otten and Dechter (2012) who used machine learning techniques to create a function that can be used to predict the difficulty of a subproblem based on easily computable features. They reported good results for a given class of MIP problems represented over graphical models solved by an AND/OR branch-and-bound.

In a slightly different fashion, Wah and Yu (1985) and Yang and Das (1994) have developed interesting approaches to evaluate the difficulty of a subproblem. Both approaches are based on probabilistic models that are used to predict the complexity of a subproblem. Despite the encouraging results that they report, the assumptions required by the probabilistic

models seem very strong and unrealistic for a wide variety of problems. The interest in parallel B&B is not limited to the field of optimization. Indeed, parallelizing B&B has also attracted some attention in the computer science community that developed several frameworks aimed at easing the implementation of personal specialized parallel B&B algorithms (see, e.g., Eckstein et al., 2001; Dorta et al., 2004).

It must be noted that the survey of related work reported here is by no means exhaustive and we refer the reader to Gendron and Crainic (1994)'s paper for a wider, though older, survey of parallel B&B techniques.

Based on the previously made observations and on previous work, and further supported by the conclusions drawn by Linderoth (1998, p. 197), we propose in this work a new approach using machine learning to balance the load between several processors and apply our approach to a set of unit commitment (UC) problems. The main contribution of this work is the development of an approach using machine learning to create and distribute several subproblems to a given number of processors such that the workloads of each processor are not too dissimilar. This is achieved through the use of learning techniques to create a difficulty estimator that is able to evaluate the difficulty (in terms of the number of nodes) of a subproblem. Moreover, we develop a set of new features that allow to represent a subproblem in order to predict its difficulty, in terms of the number of nodes. It must be emphasized that we do not propose a new parallel implementation of the B&B, but rather use a naive parallelization to illustrate how we can estimate the difficulty of subproblems through learning techniques. The experimental results show that the approach succeeds in efficiently balancing the load between several processors and that it achieves interesting results with and without communication. It is to be noted that the developed features do not depend on the class of problems used to assess our method. These features are virtually applicable to any type of MIP problem, although some adaptation might be necessary to improve the performance on a given problem class. Moreover, we must emphasize that machine learning is mainly useful when the considered problems are related to each other, otherwise it is in general difficult for the algorithm to learn something from the available data. Because of these requirements, the proposed approach is primarily applicable to the situations where similar problems have to be repeatedly solved over time. Focusing on unit commitment (UC) problems is thus a straightforward choice since generation companies, or transmission system operators, have to repeatedly solve very similar UC problems again and again.

In the remainder, we first start, in Section 5.2, by stating the addressed problem more formally and then give, in Section 5.3, a detailed description of the method that we propose. A short theoretical analysis is next carried out in Section 5.4. Sections 5.5, 5.6, and 5.7 then describe the experimental setup that we use to validate our approach, together with the experimental results that followed. Finally, Section 5.8 concludes this chapter and draws some lines of future work.

## 5.2 Problem statement

Before getting into the details of the proposed solution, let us first give a more specific description of the addressed problem. As in the previous chapter, we focus here on binary

mixed-integer linear programming (MIP) problems of the form (2.7) and, in the following, we use the notations introduced in Sections 2.3.1 and 2.3.2.

Our aim is to develop an efficient parallel version of a branch-and-bound algorithm that minimizes the amount of communication and that achieves high speedups. In order to do so, we will split the original optimization tree into several subtrees that cover together the initial tree. These subtrees constitute a partition of the initial optimization tree. Each subtree is then given to a processor (a processor can be responsible for several subtrees) that is asked to solve the subproblem defined by the subtree. Communication between processors is ideally forbidden, but a small amount can still be allowed in order to benefit from the solutions found by other processors. Because communication should be maintained at its minimum, the workload of each processor, i.e., the difficulty of the given set of subtrees, should ideally be balanced between all processors so that high speedups can be achieved. Balancing the workload is the problem tackled in this chapter.

In the context of optimization, the workload is basically the time a solver needs to find the optimal solution, but other difficulty measures are also commonly used. For instance, in the case of B&B, it is acknowledged that the number of nodes explored by the algorithm before optimality is proved favorably estimates the difficulty of a problem. In this work, we focus on the latter difficulty measure, i.e., the number of nodes, as it is more robust to perturbations during the experiments and roughly linearly dependent (up to a time factor) on the optimization time.

More specifically, the proposed method uses machine learning techniques in order to assess the difficulty of a subproblem, i.e., to determine the number of nodes required to solve it, and then uses that information to distribute the subproblems among the available processors.

## 5.3 Description of the method

In this section, we describe the method that we devise in order to balance the load of each individual processor of a parallel branch-and-bound. We first describe how we generate a set of subproblems that span the entire optimization tree. We next describe how the subproblems can be distributed among the available processors.

### 5.3.1 Generating a partition of the original optimization tree

The approach that we propose to generate a partition of the optimization tree is very much alike a traditional branch-and-bound. It is represented in Algorithm 3.

In this algorithm, we generate a partition of the original optimization tree containing at most $k$ elements. From now on, the notation $p$ represents a subproblem of the original problem, i.e., a problem for which a certain number of binary variables are fixed either to 0 or 1. In particular, $p_0$ designates the root node, i.e., a version of the original problem in which no binary variable is fixed. The algorithm first starts with $p_0$ that is added to a queue $q$. Then, the procedure is as follows. The algorithm retrieves and removes a subproblem $p$ from the queue and iteratively creates a certain number of children of $p$ by setting each unfixed binary variable in $p$ to 0 and then to 1. Thus, for each unfixed binary variable, we create

two children by fixing that variable alternately to 0 and to 1. A set of features describing each child thus created is then computed with a given function $\mathcal{C}(\cdot)$. The computed features are subsequently used as input of a learned complexity function $f_{\text{nodes}}(\cdot)$ that returns an estimate of the number of nodes required to solve the child subproblem represented by the features. The predictions of the numbers of nodes of each child are next used to compute a score according to which the unfixed binary variables in the subproblem $p$ are ranked. Once every unfixed binary variable has been scored, the two child subproblems corresponding to the variable that has the lowest score are added to the queue. The presented procedure is then repeated by removing from the queue the subproblem whose predicted number of nodes is the greatest, until the queue fulfills a given stopping criterion or until a maximum queue size has been reached.

In the end of the procedure, each element of the queue represents the root node of a subtree forming the sought partition of the original optimization tree.

The behavior of the proposed partitioning algorithm depends on three main factors: the function $f_{\text{nodes}}(\cdot)$ predicting the number of nodes of a subproblem; the way the features are computed, i.e., the implementation of function $\mathcal{C}(\cdot)$; and the implementation of the function $\text{score}(\cdot,\cdot)$. The rest of this section details how these functions were implemented in this work.

### Assessing the difficulty of a subproblem

In order to create a partition of the original optimization tree that balances well the workload of each processor, our procedure requires that a function able to predict the difficulty of a subproblem is available. As previous research indicates (Wah and Yu, 1985; Yang and Das, 1994; Laursen, 1994), it is not trivial to find a simple mathematical formulation for such a function. We thus decided to resort to machine learning in order to create that function.

In this work, we apply supervised machine learning techniques. Remember that, in the supervised learning case, a dataset containing input-output pairs is needed by the machine learning algorithm to construct the desired function. The input-output pairs should be observations of the system that the function is supposed to imitate. The inputs traditionally consist in feature vectors, i.e., vectors of scalars, that represent some characteristics of the subproblem. The output of the function is, in this case, the number of nodes that need to be explored before the subproblem is solved. The output thus gives an idea of how difficult a subproblem is.

More specifically, the function that we learn is

$$f_{\text{nodes}} : \Phi \subset \mathbb{R}^d \mapsto \mathbb{R},$$

where $\Phi$ is the feature space that is included in $\mathbb{R}^d$. In principle, the output should be an integer, but this is not guaranteed by the machine learning algorithm. We thus allow the estimated number of nodes to be a general scalar instead of an integer.

The supervised learning framework requires that a training set of input-output pairs observed from the system we are trying to imitate is available for learning. In our case, such a dataset is not available and we must thus create one so that our method can be applied. In order to do that, we first select a set of problems. We next randomly generate, for each

**Algorithm 3** Partitioning algorithm of an optimization tree.

**Inputs:** $N$, the number of processors — $k$, the maximum number of elements in the partition — $p_0$, the initial root node — $f_{\text{nodes}}(\cdot)$, a function that estimates the number of nodes of a subproblem, takes a feature vector as input — $\mathcal{C}(\cdot)$, a function that generates a feature vector that describes the subproblem given as input

1: **procedure** PARTITION($N$, $k$, $p_0$, $f_{\text{nodes}}(\cdot)$, $\mathcal{C}(\cdot)$)
2: $q = p_0$ ▷ $q$ is a queue containing subproblems
3: **while** true **do**
4: **if** $|q| \geq k$ **then**
5: **break**
6: **else if** $|q| \geq 3N$ **then**
7: **if** $\max_{p \in q} f_{\text{nodes}}(\mathcal{C}(p)) \leq \frac{3}{4N} \sum_{p \in q} f_{\text{nodes}}(\mathcal{C}(p))$ **then**
8: **break**
9: **end if**
10: **end if**
11: $p = \text{argmax}_{p' \in q} f_{\text{nodes}}(\mathcal{C}(p'))$
12: $s^* = +\infty$
13: **for** $i \in U_p$ **do** ▷ $U_p$ is the set of indices of the unfixed binary variables in $p$
14: $p_{\text{left}} = p$ with $x_i$ set to 0
15: $\hat{n}_{\text{left}} = f_{\text{nodes}}(\mathcal{C}(p_{\text{left}}))$
16: $p_{\text{right}} = p$ with $x_i$ set to 1
17: $\hat{n}_{\text{right}} = f_{\text{nodes}}(\mathcal{C}(p_{\text{right}}))$ ▷ $\hat{n}$ is an estimate of the number of nodes of $p$
18: $s = \text{score}(\hat{n}_{\text{left}}, \hat{n}_{\text{right}})$
19: **if** $s < s^*$ **then**
20: $s^* = s$
21: $p^*_{\text{left}} = p_{\text{left}}$
22: $p^*_{\text{right}} = p_{\text{right}}$
23: **end if**
24: **end for**
25: $q = q \setminus p \cup \left\{ p^*_{\text{left}}, p^*_{\text{right}} \right\}$
26: **end while**
27: **return** $q$ ▷ return a list containing the generated partition
28: **end procedure**

problem in this set, a certain number of subproblems by randomly fixing each variable in a random subset of the binary variables to 0 or 1. For instance, if the problem contains 20 binary variables, we first select a random subset of them and we subsequently randomly fix each variable in this subset to 0 or 1. The created subproblem is then optimized until optimality is reached. Solving the subproblem yields the number of nodes (and, hence, an image of its difficulty), which corresponds to the output part of a pair of the training set. The input part is given by the feature vector that is computed for the randomly generated subproblem. This procedure is repeated until enough data is generated.

Once the training set is created, we can apply a supervised machine learning algorithm in order to learn the function $f_{\text{nodes}}$ from the observed data. In this work, we use the random forests algorithm (Breiman, 2001) (see Section 3.2.3). Once the function is trained, estimating the number of nodes that solving a given subproblem requires starts with computing the features, which can then be fed to the trained function. In other words, estimating the size of the optimization tree of a subproblem $p$ is written $f_{\text{nodes}}\left(\mathcal{C}\left(p\right)\right)$. However, for the sake of conciseness, we may sometimes drop the feature computation and simply write $f_{\text{nodes}}\left(p\right)$.

### Computing the features of a subproblem

As mentioned in the previous section, the input of the function $f_{\text{nodes}}$ should be a vector. This section describes how the features are computed for a given subproblem. More specifically, we propose an implementation for the function

$$\mathcal{C} : \mathcal{P} \mapsto \Phi \subset \mathbb{R}^d,$$

where $\mathcal{P}$ is the 'space' of subproblems and $\Phi$ is the feature space. Note that the features that are detailed in this section are merely a proposal. It may be relevant, if not compulsory, to develop new features, and thus new functions $\mathcal{C}\left(\cdot\right)$, in order to apply this procedure to other classes of MIP problems than those considered in this work.

A subproblem $p$ is created by fixing a certain number of binary variables either to 0 or to 1. We denote by $F_p$ the set of the indices of the fixed binary variables and the set of unfixed binary variables by $U_p$. The indices of the variables fixed to 0 and 1 are respectively contained in the sets $F_{p0}$ and $F_{p1}$. Note that we assume that all problems are in the form (2.7) and that we use the same notations as in Sections 2.3.1 and 2.3.2.

We denote the LP solution of the root node of the original problem by $\boldsymbol{x}'_0$, and the solution at the root of the subproblem $p$ by $\boldsymbol{x}'_p$. Similarly, the value of the objective function obtained with the solutions $\boldsymbol{x}'_0$ and $\boldsymbol{x}'_p$ are denoted $o'_0$ and $o'_p$, respectively. We moreover assume that a heuristic solution is available from the beginning. The heuristic function $h(\cdot)$ applied to solution $\boldsymbol{x}'_0$ gives the solution $\boldsymbol{x}^h_0 = h(\boldsymbol{x}'_0)$, and the value of the objective function for this solution is $o^h_0$. This heuristic solution allows us to compute the initial gap $g_i$ at the root node of subproblem $p$, that is,

$$g_i = \frac{o^h_0 - o'_p}{o^h_0}.$$

Note that this gap can be negative since the LP objective of the subproblem can be greater than the objective of the heuristic solution computed at the root node of the original problem.

Additionally, we compute, for each subproblem $p$, a new right hand side $\overline{b}_i$, for each constraint $i$. Indeed, since some variables are fixed in $p$, the values of their coefficients in the constraint matrix $\boldsymbol{A}$, multiplied by their value, can be subtracted from the initial $\boldsymbol{b}$. We thus define the new right hand side as

$$\overline{b}_i = b_i - \sum_{j \in F_{p1}} A_{ij},$$

since it is not necessary to subtract the coefficients of the variables fixed to 0.

In order to compute additional features, we also optimize, for a very short period of time, the subproblem $p$ with a traditional B&B. More specifically, we allow a certain number of nodes (typically 5,000) to be explored. Note that the algorithm uses as primal bound the value of the objective function found by the heuristic at the root node, i.e., $o_0^h$. When this budget is exhausted, we extract a certain number of characteristics from the optimization. This phase is called probing and the maximum number of nodes to be explored is referred to as the probing budget. At the end of the optimization or when the probing budget is exhausted, we retrieve the dual bound $o_{dual}^{probing}$ and the new primal bound $o_{primal}^{probing}$. With these values, we can compute the final gap $g_f$ at the end of the probing phase with

$$g_f = \frac{o_{primal}^{probing} - o_{dual}^{probing}}{o_{primal}^{probing}}.$$

Besides the previous values that must be recomputed for each new subproblem $p$, we carry out some preliminary calculations whose results are subsequently used to extract characteristics from any subproblem $p$. More specifically, we compute the relative objective increase observed between the root node of the original problem and the subproblem created by fixing a specific binary variable $x_j$, with $j \in I$, to 0 or 1. We thus obtain two vectors $\boldsymbol{oi}_0$ and $\boldsymbol{oi}_1$, such that

$$oi_0(j) = \frac{o'_{p_{x_j=0}} - o'_0}{|o'_0|} \quad \text{and} \quad oi_1(j) = \frac{o'_{p_{x_j=1}} - o'_0}{|o'_0|},$$

where $p_{x_j=0}$ (respectively $p_{x_j=1}$) is the subproblem created by fixing variable $x_j$ to 0 (respectively 1), and leaving all other variables unfixed. These vectors are computed once and for all in the beginning and are used to compute some of the proposed features.

The above description merely introduces the notations and some values that are used to compute the features describing a given subproblem $p$. The complete list of features that we use in this work is given in Tables 5.1 and 5.2. In these tables, the features are separated into five categories, each one of which is meant to represent different aspects of the problem. The first category of features captures basic characteristics of the subproblem, as well as some differences between the subproblem and the original root problem, like the increase of the LP objective between the roots of both problems. The second category aims at representing the different interactions that exist between the fixed binary variables of the subproblem and the other binary variables in the cost function and in the constraints. Then, the features in the third category model the sparsity of the subproblem with different measures computed from the subproblem and the original problem. The fourth category is similar to the second one except that its goal is to evaluate the connections between all variables (fixed, unfixed binary,

and continuous variables) in the objective function as well as in the constraints. Finally, the fifth category contains the features that are computed after the probing phase. These features give a small glimpse of the optimization of the subproblem.

Tables 5.1 and 5.2 list 66 features in total. When all features are computed for a given subproblem $p$, the output of function $\mathcal{C}(p)$ is a vector composed of the 66 values, where the numbers given in the tables represent the positions of the features in the final vector.

**Scoring a variable**

In the algorithm that we propose (Algorithm 3, line 18), a score is used to determine which variable it is better to branch on in order to expand the current tree by two newly created nodes. How this score is computed influences the behavior of the tree partitioning algorithm. The first score that we propose aims at balancing the difficulty, i.e., the number of nodes, of each newly created children. The proposed score is as follows:

$$\text{score}_{\text{bal}}(\hat{n}_{\text{left}}, \hat{n}_{\text{right}}) = \frac{\hat{n}_{\text{left}} + \hat{n}_{\text{right}}}{2} + \left| \frac{\hat{n}_{\text{right}} - \hat{n}_{\text{left}}}{2} \right| + \left| \frac{\hat{n}_{\text{left}} - \hat{n}_{\text{right}}}{2} \right|,$$

where $\hat{n}_{\text{left}}$ and $\hat{n}_{\text{right}}$ respectively denote the estimated size of the left and right subproblems.

The other proposed scoring criterion is designed such that the total amount of work is minimized. It is given by

$$\text{score}_{\text{max}}(\hat{n}_{\text{left}}, \hat{n}_{\text{right}}) = \max(\hat{n}_{\text{left}}, \hat{n}_{\text{right}}).$$

This score does not take into account the difficulty equilibrium between the two created nodes. It is assumed that the balance can be achieved later when the generated subproblems are distributed to the workers.

### 5.3.2 Distributing nodes to processors

In the case where the number of generated subproblems is equal to the number of processors, the distribution of the work is trivial. However, when the number of nodes in the partition is greater than the number of processors, one must find a way to distribute the work evenly between the processors such that the workload is well balanced between each worker.

There are several ways this distribution can be done. In this work, we apply a simple greedy method although other, more formal, approaches exist. The greedy routine that we use is detailed in Algorithm 4. The output of this algorithm is an array, one element per processor, of queues specifying which subproblems have to be solved by a given processor.

| Feat. # | Description |
|---|---|
| 1 | $\left\|o'_0 - o'_p\right\| / \left\|o'_0\right\|$ |
| 2-3 | $\|F_{p0}\|/\|I\|$ ; $\|F_{p1}\|/\|I\|$ |
| 4 | $\|U_p\|/\|I\|$ |
| 5 | $\left(\sum_{j\in F_{p0}} \left\|0 - x'_0(j)\right\| + \sum_{j\in F_{p1}} \left\|1 - x'_0(j)\right\|\right) / \|F_p\|$ |
| 6-9 | $[\min\ ;\ \max\ ;\ \text{mean}\ ;\ \text{std}]_{j\in F_{p0}}\ oi_0(j)$ |
| 10-13 | $[\min\ ;\ \max\ ;\ \text{mean}\ ;\ \text{std}]_{j\in F_{p1}}\ oi_1(j)$ |
| 14-17 | $[\min\ ;\ \max\ ;\ \text{mean}\ ;\ \text{std}]_{j\in U_p}\ \frac{1}{2}oi_0(j) + \frac{1}{2}oi_1(j)$ |
| 18 | $\left(o_0^h - o'_p\right) / \left\|o_0^h\right\|$ |
| 19 | $\sum_{j\in F_p} c_j / \sum_{j\in I} c_j$ |
| 20 | $\sum_{j\in U_p} c_j / \sum_{j\in I} c_j$ |
| 21-22 | $[\min\ ;\ \max]_{i=1\ldots m} \left(b_i - \sum_{j\in F_{p1}} A_{ij}\right) / b_i$ |
| 23-24 | $[\min\ ;\ \max]_{i=1\ldots m} \sum_{j\in U_p} A_{ij} / b_i$ |
| 25-26 | $[\min\ ;\ \max]_{i=1\ldots m} \left(\sum_{j\in U_p} A_{ij} - \overline{b}_i\right) / \overline{b}_i$ |
| 27-28 | $[\min\ ;\ \max]_{i=1\ldots m} \sum_{j\in F_p} A_{ij} / \sum_{j\in I} A_{ij}$ |
| 29-30 | $[\min\ ;\ \max]_{i=1\ldots m} \sum_{j\in U_p} A_{ij} / \sum_{j\in I} A_{ij}$ |
| 31-34 | $[\text{mean}\ ;\ \min\ ;\ \max\ ;\ \text{std}]_{j\in F_p} \left\|A_{:j}\right\|_0 / m$ |
| 35-38 | $[\text{mean}\ ;\ \min\ ;\ \max\ ;\ \text{std}]_{j\in U_p} \left\|A_{:j}\right\|_0 / m$ |
| 39-42 | $[\text{mean}\ ;\ \min\ ;\ \max\ ;\ \text{std}]_{j\in C} \left\|A_{:j}\right\|_0 / m$ |

Table 5.1: Features (1/2) used to describe a subproblem. To avoid repetitions, we use brackets to indicate that several features are obtained with the same expression by simply changing a small element. For instance, $3 + [\ a;\ b\ ]$ indicates that two features are computed with this expression: one is $3 + a$ and the other is $3 + b$.

| Feat. # | Description |
|---|---|
| 43 | $\sum_{j\in F_p} c_j / \sum_{j=1}^{n} c_j$ |
| 44 | $\sum_{j\in U_p} c_j / \sum_{j=1}^{n} c_j$ |
| 45 | $\sum_{j\in C} c_j / \sum_{j=1}^{n} c_j$ |
| 46-47 | $[\min\,;\,\max]_{i=1\ldots m} \sum_{j\in F_p} A_{ij} / \sum_{j=1}^{n} A_{ij}$ |
| 48-49 | $[\min\,;\,\max]_{i=1\ldots m} \sum_{j\in U_p} A_{ij} / \sum_{j=1}^{n} A_{ij}$ |
| 50-51 | $[\min\,;\,\max]_{i=1\ldots m} \sum_{j\in C} A_{ij} / \sum_{j=1}^{n} A_{ij}$ |
| 52-53 | $[\min\,;\,\max]_{i:\overline{b}_i\geq 0} \left(\sum_{j\in U_p:A_{ij}\geq 0} A_{ij} + \sum_{j\in C:A_{ij}\geq 0} A_{ij}\right) / \overline{b}_i$ |
| 54-55 | $[\min\,;\,\max]_{i:\overline{b}_i\geq 0} \left(\sum_{j\in U_p:A_{ij}< 0} A_{ij} + \sum_{j\in C:A_{ij}< 0} A_{ij}\right) / \overline{b}_i$ |
| 56-57 | $[\min\,;\,\max]_{i:\overline{b}_i< 0} \left(\sum_{j\in U_p:A_{ij}\geq 0} A_{ij} + \sum_{j\in C:A_{ij}\geq 0} A_{ij}\right) / \overline{b}_i$ |
| 58-59 | $[\min\,;\,\max]_{i:\overline{b}_i< 0} \left(\sum_{j\in U_p:A_{ij}< 0} A_{ij} + \sum_{j\in C:A_{ij}< 0} A_{ij}\right) / \overline{b}_i$ |
| 60 | $\left(o_{dual}^{probing} - o'_p\right) / o_{dual}^{probing}$ |
| 61 | $\left(o_0^h - o_{primal}^{probing}\right) / o_0^h$ |
| 62 | ratio between the number of open nodes left after the probing budget is exhausted and the number of explored nodes |
| 63 | maximum depth of the probing tree |
| 64 | depth of the last full level (i.e., the level $l$ such that the numbers of nodes in the levels $l' = 1 \ldots l$ are $2^{l'}$) in the probing tree |
| 65 | waist of the probing tree (i.e., level $l$ with the largest number of nodes) |
| 66 | $100\frac{(g_i - g_f)}{\lvert g_i\rvert}$ |

Table 5.2: Features (2/2) used to describe a subproblem. To avoid repetitions, we use brackets to indicate that several features are obtained with the same expression by simply changing a small element. For instance, $3 + [\ a;\ b\ ]$ indicates that two features are computed with this expression: one is $3 + a$ and the other is $3 + b$.

**Algorithm 4** Greedy subproblem allocation.

**Inputs:** $N$, the number of processors — $q$, a list containing the subproblems of the generated tree partition

1: **procedure** ALLOC($N$, $q$)
2: $h(i) = 0, \forall i$ ▷ $h$ is an array of $N$ scalars
3: $l(i) = \emptyset, \forall i$ ▷ $l$ is an array of $N$ queues
4: **while** $q \neq \emptyset$ **do**
5: $p = \text{argmax}_{p' \in q} f_{\text{nodes}}(\mathcal{C}(p'))$ ▷ select the most difficult remaining subproblem
6: $j = \text{argmin}_{j'=1\ldots N} h(j')$ ▷ identify the queue whose expected workload is the least
7: $l(j) = l(j) \cup p$ ▷ add the current subproblem to the chosen queue
8: $h(j) = h(j) + f_{\text{nodes}}(\mathcal{C}(p))$ ▷ update the expected workload of the chosen queue
9: $q = q \setminus p$
10: **end while**
11: **return** $l$ ▷ $l(i)$ contains the subproblems to be solved by processor $i$
12: **end procedure**

A more formal approach to allocate the subproblems to the workers is to solve the following optimization problem:

$$
\begin{aligned}
\min \quad & \sum_{i=1}^{N} z_i \\
\text{s.t.} \quad & \sum_{j=1}^{k} a_{ij} \hat{n}_j = w_i \quad \forall i = 1 \ldots N \\
& \sum_{i=1}^{N} a_{ij} = 1 \quad \forall j = 1 \ldots k \\
& m - w_i \leq z_i \quad \forall i = 1 \ldots N \\
& m - w_i \geq -z_i \quad \forall i = 1 \ldots N \\
& a_{ij} \in \{0, 1\} \\
& w_i, z_i \in \mathbb{R}^+,
\end{aligned}
$$

where $k$ and $N$ respectively correspond to the number of subproblems to be allocated and to the number of processors, and $\hat{n}_j$ and $m$ respectively represent the predicted number of nodes of a subproblem and the ideal average load of each processor, i.e., $\frac{\sum_j \hat{n}_j}{N}$. The solution of this problem yields an optimal subproblem allocation based on the available knowledge. Obviously, since the amount of work (i.e., the number of nodes) required to solve each subproblem is not known exactly, the real allocation is likely to be suboptimal. Formulating the subproblem allocation as a stochastic or robust optimization problem is an interesting way to take uncertainty into account.

## 5.4 Theoretical analysis

In this section, we present a short theoretical analysis of our method. Indeed, one of the side advantages of using machine learning is that it is possible to get in advance, i.e., without carrying out the optimization, an estimate of the number of nodes to be processed before a subproblem is solved to optimality. Moreover, the error of this estimate can be estimated as well. In the following, we show how these estimates can be used to get an approximation of the speedup of the method before the optimization is carried out.

Thanks to machine learning, the number of nodes $n_i$ required to solve a subproblem $p_i$ can be estimated with the learned function, i.e., $\hat{n}_i = f_{\text{nodes}}(p_i)$. In practice however, the prediction is not perfect and we can assume that the real number of nodes $n_i$ required to solve the subproblem to optimality is a random variable that is distributed around the predicted value $\hat{n}_i$. Furthermore, it is in general easier to reach an equilibrium between the workloads of each processor when the chunks that have to be distributed are smaller. For this reason, we assume that the number of generated subproblems is greater than the number of processors. Each worker $j$ therefore possesses a queue $q_j$ containing the subproblems for which it is responsible.

Assuming that the mean of $n_i$ is $\hat{n}_i$ and that its standard deviation is $\hat{\sigma}_i$, the following theorems characterize a given subproblem allocation. We propose two theorems that characterize, respectively, the speedup obtained with a parallel work distribution, and its absolute duration, i.e., the maximum number of nodes over all processors. Both theorems provide information that can be used in different situations. Indeed, the speedup is useful when one wants to characterize whether the processors are efficiently used, while the absolute duration is of interest when one wants to know how quickly an optimization job will terminate. Note that, in this context, the speedup has to be understood as the ratio between the total amount of work carried out by all processors (i.e., the sum of the numbers of nodes of all processors), and the largest amount of work (i.e., the largest number of nodes over all processors).

**Theorem 1** (Speedup approximation)**.** *Let $k$ be a number of subproblems that have been generated in such a way that their union covers the entire original optimization tree, and let each subproblem $p_i$ be allocated to one queue $q_j$ of one of the $N$ available workers $w_j$, the speedup $SU$ obtained by this work distribution is bounded below by $l_{SU}$ and above by $u_{SU}$, i.e.,*

$$l_{SU} \leq SU \leq u_{SU},$$

*where $l_{SU} = 1 + \frac{l(N-1)}{u}$ and $u_{SU} = N$, with probability at least*

$$\varepsilon = \prod_{j=1}^{N} \rho\left(l, u, \mu_{w_j}, \sigma_{w_j}\right),$$

*where $\mu_{w_j} = \sum_{t:p_t \in q_j} \hat{n}_t$, $\sigma^2_{w_j} = \sum_{t:p_t \in q_j} \hat{\sigma}^2_t$, and*

$$\rho\left(l, u, \mu_{w_j}, \sigma_{w_j}\right) = \int_l^u \frac{1}{\sigma_{w_j}\sqrt{2\pi}} \exp\left[-\frac{\left(x - \mu_{w_j}\right)^2}{2\sigma^2_{w_j}}\right] \mathrm{d}x.$$

*Proof.* The main mechanisms of the proof of this theorem are based on the probability theory. Since we assume that $k > N$, each worker is responsible for the optimization of a certain number of subproblems. The total number of nodes required in order to finish a 'job' is thus the sum, over all subproblems $p_i$ optimized by a processor $w_j$, of the number of nodes $n_i$ that these subproblems require in order to be fully optimized. We define a new random variable $G_{w_j}$ for the worker $w_j$ such that

$$G_{w_j} = \sum_{i:p_i \in q_j} n_i.$$

Assuming that the central limit theorem applies in this situation, and that the variables $n_i$ are independent of each other, the random variable $G_{w_j}$, which represents the total amount of work the processor carries out, is distributed according to a normal distribution with parameters

$$\mu_{w_j} = \sum_{t:p_t \in q_j} \hat{n}_t \quad \text{and} \quad \sigma^2_{w_j} = \sum_{t:p_t \in q_j} \hat{\sigma}^2_t.$$

Then, arbitrarily choosing two values $l$ and $u$, we can compute the probability that the total number of nodes explored by one worker is comprised between $l$ and $u$:

$$\rho\left(l, u, \mu_{w_j}, \sigma_{w_j}\right) = \int_l^u \frac{1}{\sigma_{w_j}\sqrt{2\pi}} \exp\left[-\frac{\left(x - \mu_{w_j}\right)^2}{2\sigma^2_{w_j}}\right] \mathrm{d}x.$$

Given that the variables $n_i$ are independent of each other, so are the variables $G_{w_j}$. Thus, the probability $\varepsilon$ that the total amount of work carried out by each worker is comprised between $l$ and $u$ is given by

$$\varepsilon = \prod_{j=1}^{N} \rho\left(l, u, \mu_{w_j}, \sigma_{w_j}\right).$$

Finally, the lower and upper bounds $l_{\mathrm{SU}}$ and $u_{\mathrm{SU}}$ on the speedup can be computed from the bounds $l$ and $u$ on the number of nodes of each worker by

$$l_{\mathrm{SU}} = 1 + \frac{l\,(N-1)}{u} \quad \text{and} \quad u_{\mathrm{SU}} = N,$$

where $l_{\mathrm{SU}}$ and $u_{\mathrm{SU}}$ respectively represent the worst and the best case. The best bound on the speedup $u_{\mathrm{SU}}$ corresponds to the case where all workers carry out the same amount of work. On the other hand, the worst case $l_{\mathrm{SU}}$ corresponds to the case where $N-1$ workers carry out an amount of work equal to $l$, while the remaining worker is responsible for an amount of work equal to $u$. □

Note that the probability $\varepsilon$ given by Theorem 1 is actually a lower bound on the probability that the speedup falls in the range $[l_{\mathrm{SU}}, u_{\mathrm{SU}}]$. Indeed, there are situations where the amounts of work of the processors are outside the range $[l, u]$, but still yield a speedup comprised between $l_{\mathrm{SU}}$ and $u_{\mathrm{SU}}$.

Theorem 1 can be used to determine beforehand whether the chosen work distribution would lead to interesting speedups or not. It is to be noted that the previous analysis is valid when communication between processors is forbidden. If the workers are given the possibility to communicate, for example a primal bound, the expected speedup would most likely be greater than the one estimated by Theorem 1. Moreover, we must emphasize the

fact that this speedup computation assumes that the serial amount of work, i.e., when a single subproblem (the root) is optimized by a single processor, is equal to the sum of the individual amounts of work of each subproblem. This is not entirely true, but, for the sake of simplicity, this approximation is used in order to compute in advance the speedup for a given work distribution. Thus, rather than giving a real speedup, the previous theorem might be more useful to estimate the actual utilization of the processors.

In addition to the speedup, the mechanisms of the previous theorem can be used to determine the probability that a worker $w_j$ explores more than a given number of nodes.

**Theorem 2.** *Based on the same assumptions as Theorem 1, the probability $\varepsilon$ that each worker explores no more than a given number $t$ of nodes is given by*

$$\varepsilon = \prod_{j=1}^{N} \varphi\left(t, \mu_{w_j}, \sigma_{w_j}\right),$$

*with*

$$\varphi\left(t, \mu_{w_j}, \sigma_{w_j}\right) = \int_{-\infty}^{t} \frac{1}{\sigma_{w_j}\sqrt{2\pi}} \exp\left[-\frac{\left(x-\mu_{w_j}\right)^2}{2\sigma_{w_j}^2}\right] \mathrm{d}x.$$

*Proof.* The proof of this theorem follows immediately from the fact that the random variables $G_{w_j}$ are normally distributed. The probability that one of these variables is less than a given value $t$ is directly computable, and the probability that all variables are less than $t$ is obtained by computing the product of each individual probability since the $G_{w_j}$ are assumed to be independent of each other. □

Given the presented theorems, and provided that the considered learning algorithm is able to characterize the variance of a prediction (which is the case for the random forests), one can easily evaluate the performance of a given partition of the original problem and its distribution among several workers. In a similar way, the proposed theorems could be used, with some adaptations, to find the optimal work distribution, instead of evaluating a given subproblem allocation.

## 5.5 Experiments

We describe here the problems that we use to evaluate our approach and the general experimental procedure that leads to the presented results.

### 5.5.1 Problem sets

We evaluate our approach on a set of unit commitment (UC) problems. The problems that we consider are a minimalist version of UC problems. Their mathematical form is given by

$$
\begin{aligned}
\min \quad & \sum_{j=1}^{n_{NS}} c_j^{NS} \sum_{i=1}^{T} x_{ij}^{NS} + \sum_{j=1}^{n_S} c_j^S \sum_{i=1}^{T} x_{ij}^S \\
& + \sum_{j=1}^{n_{NS}} f_j^{NS} \sum_{i=1}^{T} y_{ij}^{NS} + \sum_{j=1}^{n_S} f_j^S \sum_{i=1}^{T} y_{ij}^S + \sum_{j=1}^{n_S} u_j^S \sum_{i=2}^{T} z_{ij} \\
\text{s.t.} \quad & \textstyle\sum_{j=1}^{n_{NS}} x_{ij}^{NS} + \sum_{j=1}^{n_S} x_{ij}^S \geq d_i && \forall i = 1 \dots T \\
& x_{ij}^{NS} \leq M_j^{NS} y_{ij}^{NS} && \forall i = 1 \dots T,\ \forall j = 1 \dots n_{NS} \\
& x_{ij}^S \leq M_j^S y_{ij}^S && \forall i = 1 \dots T,\ \forall j = 1 \dots n_S \\
& z_{ij} - y_{ij}^S + y_{i-1j}^S \geq 0 && \forall i = 2 \dots T,\ \forall j = 1 \dots n_S \\
& x_{ij}^{NS}, x_{ij}^S \in \mathbb{R}^+ \\
& y_{ij}^{NS}, y_{ij}^S, z_{ij} \in \{0, 1\}\,.
\end{aligned}
$$

In this formulation, $T$, $n_{NS}$, and $n_S$ represent the number of time periods, the number of power plants without startup costs, and the number of plants with startup costs, respectively. The other parameters $c_j$, $f_j$, and $u_j$ denote the variable, fixed, and startup costs of each power plant, respectively. Finally, the $M_j$ and $d_i$ denote the nominal (maximum) power of each plant and the demands that have to be satisfied at each time period, respectively.

In this work, we set the number of periods to 12, and the number of power plants with and without startup costs both to 5. Moreover, all the UC problems that we consider differ only by the demand of each period, i.e., all the parameters are identical except for the demands that are different for each problem. In order to create our problems, we generate randomly a first set of parameters including the demands, which will constitute the basis of our UC problems. Then, the demands for each problem are randomly updated by adding to the initial vector of demands $\left[\overline{d}_1, \dots, \overline{d}_T\right]$ a unique randomly drawn term $d_m$, and a random term for each time period $d'_i$. The final demand vectors are thus of the form $\left[d_m + \overline{d}_1 + d'_1, \dots, d_m + \overline{d}_T + d'_T\right]$, where the $d_m$ changes from problem to problem, and the $d'_i$ from problem to problem and from period to period. We generate one set of 300 problems that constitute a training set, and a set of 20 UC problems to evaluate our approach. Those problem sets are available online[1] and can be provided upon request.

All our experiments are performed on our randomly generated problems. There are two reasons why we decided to use such problems for our experiments. First, machine learning requires that the problems that we use for learning and for testing are similar enough. If the problems in the training set are too dissimilar from those in the test set, nothing useful for the test can be learned from the provided data. In this first study, we thus decided to focus only on a single class of problems, with similar characteristics. In principle, the approach can

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

be extended to take into account different classes of problems and more dissimilar problems, but this demands more data to be generated (and considerably more time). Moreover, the features would probably need to be adapted to capture a (most likely) larger set of problems dynamics. Second, the choice of a set of UC problems with a same cost structure and varying demands is also motivated by its similarity to practical situations. Indeed, in the real world, the generation companies, or the transmission system operators, have to repeatedly solve similar problems with a similar cost structure (the plants do not change very often), but with a varying demand. Our problem setting is thus strongly motivated by an obvious similarity with practical applications.

### 5.5.2 Experimental procedure

Once the problem sets are at our disposal, our experimental procedure can be applied. It is composed of three steps: (1) we generate a training set $D_{\text{np}}$ of pairs composed of features of subproblems and the corresponding numbers of nodes; (2) we learn from $D_{\text{np}}$ a function able to predict the size of a subproblem; and (3) we apply our partitioning algorithm in order to generate several subproblems and analyze the obtained experimental results.

Note that, in all experiments, including steps (1) and (3) of our experimental procedure, we give to B&B an upper (primal) bound on the problem. This primal bound is very loose, and is computed with a simple heuristic that merely consists in rounding up each fractional variable in the LP solution of the root node of the original problem.

#### Step 1: dataset generation

In order to create a dataset of pairs $(\boldsymbol{\phi}_l, n_l)$, we first generate, for each problem in our learning problem set, a certain number (250) of random subproblems. Each subproblem is created by randomly choosing a subset of the binary variables and by randomly fixing each variable in this subset to either 0 or 1. Then, for each subproblem, we compute the features $\boldsymbol{\phi}_l$ corresponding to this subproblem and we solve the subproblem to optimality. The number of nodes $n_l$ required to fully optimize the subproblem is added, together with the feature vector $\boldsymbol{\phi}_l$, to the dataset $D_{\text{np}}$ as a pair $(\boldsymbol{\phi}_l, n_l)$. The dataset $D_{\text{np}}$ contains around $75,000$ learning examples and is used as input of the learning algorithm to create the function $f_{\text{nodes}}(\cdot)$.

#### Step 2: learning a function predicting the number of nodes

We now apply a supervised machine learning algorithm to the training set $D_{\text{np}}$ to learn a function that predicts the number of nodes required to solve a subproblem to optimality. In this work, we use the *random forests* algorithm (Breiman, 2001), whose description is given in Section 3.2.3. Our choice is motivated by the computational efficiency (in the learning phase) and the simple mechanisms of the random forests. Another advantage is that the performance of the random forests is very robust against the choice of their parameters. The random forests actually have three main parameters: $M$, which is the number of trees in the ensemble method; $n_{\min}$, which is the number of training samples contained in a node below which that node becomes a leaf; and $m$, which is the number of candidate features

considered at each node of a tree. The number of trees is set to $M = 50$ in our experiments. The parameter $n_{\min}$ controls the complexity of the trees and is set to a value of $n_{\min} = 10$. Finally, the parameter $m$ is given the value $d$, where $d$ is the number of features used to describe a subproblem ($m = 66$ in our case).

Because the experiments show that the parameter values have little impact on the performance of the method, the values that we give to those parameters have been chosen based on our experience without any tuning. The exact understanding of these parameters is beyond the scope of this chapter and we refer the reader to Section 3.2.3 or to Breiman (2001) for a deeper explanation.

**Step 3: comparing several partitioning schemes**

After having generated the training set $D_{\text{np}}$ and applied the learning algorithm, we can compare our learned partitioning scheme to other schemes. Besides the one proposed in this work and due to the lack of clear competitors, we have imagined two extremely simple approaches that we compare our method with. The first one, that we call 'random', consists in generating a certain number of subproblems partitioning the original optimization tree completely randomly. The procedure is as follows. Imagine that there is a list that stores, such as in B&B, all open nodes. The list is first initialized with the original root node. While the number of elements in the list is less than the desired number of elements in the partition, the procedure takes one node randomly from the list. That node is examined and the unfixed binary variables are identified. Then, one unfixed binary variable is randomly chosen and two child nodes are created by fixing the randomly chosen variable to 0 and 1, respectively. This procedure yields a totally random partitioning of the original tree. The second approach that we propose is similar to the previous one, except that the next node to split into two children is not chosen randomly. Indeed, we rather open the nodes in a breadth-first manner such that the tree resulting from the random partitioning is balanced. We naturally name this approach 'balanced'.

When the number of nodes is equal to the number of processors, distributing the work among the different workers is easy. When the number of elements in the partition is greater than the number of processors, we must find a way to distribute the work between them. When the partition is generated with our learned method, we use Algorithm 4 to distribute the work between all workers. When the random or balanced schemes generate the partition, we randomly distribute the subproblems to each worker while balancing the number of subproblems that each processor is responsible for, i.e., we attribute to each processor a number $k/N$ of subproblems.

Note that the subproblems generated by our approach depend on the scoring function that is used. We propose two different scoring functions in Section 5.3.1. However, our experiments show that the second one, i.e., $\text{score}_{\max}$, is more efficient than the first one. Thus, for the sake of conciseness, we mainly focus, from now on, on the scoring function $\text{score}(\hat{n}_{\text{left}}, \hat{n}_{\text{right}}) = \text{score}_{\max}(\hat{n}_{\text{left}}, \hat{n}_{\text{right}}) = \max(\hat{n}_{\text{left}}, \hat{n}_{\text{right}})$. We report nonetheless some results generated with $\text{score}_{\text{bal}}$ to illustrate to what extent $\text{score}_{\text{bal}}$ is less effective than $\text{score}_{\max}$.

In order to assess the proposed approach, we generate, for each problem in our test set, several partitions of increasing size with the three proposed partitioning schemes. We then

gather the results and analyze them. Furthermore, we consider a setting without communication and a setting with communication where the best found primal bound is shared among the processors (updates of this primal bound are performed on a regular basis).

We evaluate our approach on the problems contained in our test set (20 UC problems). CPLEX 12.2 is used as the main B&B solver. Note that presolve is applied to each problem at the root node and then is disabled for the subproblems. Moreover, in order to assess only the performance of the partitioning strategies, we disable heuristics and cuts in CPLEX.

## 5.6 Experimental results: learning

This section focuses on the presentation and discussion of some experimental results regarding the learning procedure.

### 5.6.1 Learning to predict the number of nodes

We first report some results regarding the accuracy of the learned complexity function $f_{\text{nodes}}$. In order to quantify the precision of the function, we split the initial training set $D_{\text{np}}$ into two sets: one set, $D_{\text{np}}^{\text{train}}$, is used to train the complexity function (around 50,000 samples), and the other set, $D_{\text{np}}^{\text{test}}$, is used to test the learned function (around 25,000 samples)[2]. Once the function is trained, we predict an output for each feature vector in $D_{\text{np}}^{\text{test}}$, which can then be compared to the real output stored in the set. An element in the test set is denoted by $(\boldsymbol{\phi}_i, n_i) \in D_{\text{np}}^{\text{test}}$ and the corresponding prediction obtained with the learned function is denoted by $f_{\text{nodes}}(\boldsymbol{\phi}_i) = \hat{n}_i$.

The presented result tables illustrate the performance of our learned function. More specifically, the tables report the mean and the median of the achieved relative errors, as well as the Spearman correlation (i.e., the correlation of the ranks) between the predictions and the real values. For a sample $(\boldsymbol{\phi}_i, n_i)$ in the test set, the relative error is given by

$$\text{RE}_i = \frac{|n_i - \hat{n}_i|}{\max(\varepsilon, n_i)},$$

where $\varepsilon \in \mathbb{R}_0^+$ is added to upper bound the computed relative errors.

Computing the correlation, the mean, and the median of the relative errors for all samples in the test set is not enough to correctly assess the learning accuracy. Indeed, the distribution of the relative errors in $\mathbb{R}^+$ is not uniform: the distribution is much denser for smaller errors. For this reason, we partition the space of relative errors, i.e., $\mathbb{R}^+$, in several regions and evaluate the learned function independently on each region. The learned function is therefore assessed in several situations depending on the magnitude of the errors.

Our method is initially designed to predict the number of nodes of a subproblem. However, this learning problem is rather difficult because the output, i.e., the number of nodes,

[2]Note that the initial dataset $D_{\text{np}}$ is split with respect to the problems that generate the samples. This means that the problems used to generate the samples contained in the test set $D_{\text{np}}^{\text{test}}$ are different from the problems that generate the samples in the training set $D_{\text{np}}^{\text{train}}$.

typically follows an exponential function of the problem parameters, e.g., the number of binary variables[3]. The exponential nature of the output makes it hard for traditional learning algorithms to perform well (in terms of common performance measures, e.g., the mean relative error). In order to ease the problem from the learning point of view, we also consider the case where the output is not the number of nodes of the subproblem, but its logarithm. This modification does not change anything to the task (the logarithm is monotonous), but renders the problem slightly easier from the learning point of view, because the input-output relationship becomes more linear. In the following (and in the next section), we thus report the results in the case where the function is trained with the number of nodes, and in the case where the function is trained with the logarithm of the number of nodes. Note that the relative errors are computed with $\varepsilon = 1$ when the output is the number of nodes and with $\varepsilon = 1e^{-5}$ when the output is the logarithm of the number of nodes.

Table 5.3 reports the prediction accuracy of the random forests when the output is the number of nodes and when the output is the logarithm of the number of nodes. The table shows that, in both cases, around 28% of the test samples are predicted exactly. Those samples mostly correspond to very small or unfeasible subproblems that are very easily detected by the learning method with the chosen features. When larger errors are considered, we can make three main observations:

1. the Spearman correlation between the predictions and the real outputs is always very high (0.86 in the worst case);
2. when the number of nodes is the chosen output, only 22.35% of the samples have a relative error larger than 150% and, despite the relatively high mean relative error (738.01%), the correlation remains very good (0.90);
3. when the output is the logarithm of the number of nodes, the relative errors are much better (the mean relative error is 20.30% in the worst case) than in the former case, but the correlation between the predictions and the real outputs diminishes.

Table 5.3 suggests that, as expected, it is easier to predict the logarithm than the number of nodes. However, the correlation in the logarithm case, though still high, is less than the correlation obtained when the number of nodes is the chosen output. The next section shows that lower relative errors do not necessarily mean better optimization results. Table 5.3 also shows that there exist some problems for which the learned function is not accurate enough. This was expected since the dataset that we use is obviously too small and does not satisfactorily cover the space of considered problems (especially the largest problems). Consequently, the predictions are less accurate for those (sub)problems that are not well represented in the training set. There are two (complementary) ways to alleviate this problem: (i) increase the training set, and (ii) develop better features. However, despite the relatively bad accuracy, the computed correlations show that the predictions are still highly correlated with the real values even for the hardest problems. This is important since we are not interested only in quantitative aspects of the predictions, but also in qualitative aspects. Indeed, being able to tell whether a subproblem will take much longer to solve than another one is a valuable piece of information that can be as useful as the accurate prediction of the number of nodes.

[3]There is no rule that states that the number of nodes is indeed an exponential function of the input parameters, but such a behavior is often observed in practice.

| Output | Considered elements | % elem. | RE (%) Mean | RE (%) Median | Corr. |
|---|---|---|---|---|---|
| #nodes | | | | | |
| | $RE_i = 0$ | 28.08 | 0.00 | 0.00 | 1.0000 |
| | $0 < RE_i \leq 50$ | 31.96 | 21.39 | 19.93 | 0.9959 |
| | $50 < RE_i \leq 100$ | 12.59 | 70.22 | 68.17 | 0.9371 |
| | $100 < RE_i \leq 150$ | 5.02 | 122.92 | 121.78 | 0.9996 |
| | $150 < RE_i \leq +\infty$ | 22.35 | 738.01 | 388.87 | 0.9014 |
| log #nodes | | | | | |
| | $RE_i = 0$ | 28.10 | 0.00 | 0.00 | 1.0000 |
| | $0 < RE_i \leq 5$ | 36.44 | 2.21 | 2.08 | 0.9900 |
| | $5 < RE_i \leq 10$ | 20.11 | 7.24 | 7.16 | 0.9106 |
| | $10 < RE_i \leq 15$ | 9.29 | 12.17 | 11.98 | 0.8272 |
| | $15 < RE_i \leq +\infty$ | 6.06 | 20.30 | 18.60 | 0.8696 |

Table 5.3: Prediction accuracy of the learned complexity function when the considered output is the number of nodes and when the output is the logarithm of the number of nodes (probe size=5,000). The table reports the mean and the median of the relative errors, as well as the Spearman correlation coefficient between the real outputs and the predictions. Each line represents a case where only some samples are considered for the computations. For instance, the line labeled $0 < RE_i \leq 50$ indicates that only those samples for which the relative error is strictly greater than 0 and less than or equal to 50 are used to compute the mean, the median, and the correlation. The table also reports the proportion of elements considered in each line. Note that the samples for which the prediction is perfect, i.e., $RE_i = 0$, are separated from the others to compute the performance measures.

In addition to Table 5.3, we also report, in Table 5.4, complementary learning results. While Table 5.3 displays the learning results for the default probing budget (5,000 nodes), Table 5.4 reports the same performance measures when the probing budget varies. As a reminder, the probing budget refers to the size of the probing tree that is used in the computation of the features. As expected, the table indicates that increasing the probing size to 50,000 nodes has a positive impact on the predictions. When the output is the number of nodes, the proportion of elements that are not well predicted ($150 < RE_i \leq +\infty$) significantly decreases (from 22.35% to 13.38%) and the corresponding mean relative error falls down to 434.59%. Besides, the Spearman correlation significantly progresses. When the output is the logarithm of the number of nodes, the increase in the accuracy of the method is not as significant, but the correlation becomes much better. It is to be noted that reducing the probing budget has a serious negative impact on the learning accuracy. However, the optimization results reported in the next section show that the effect on the parallelization of B&B is limited.

Finally, it is important to recall that the learning accuracy is not crucial here. Indeed, the main purpose of the approach is to split the workload evenly among several processors and not to predict with infinite accuracy the size of an optimization tree. Obviously, being able to

estimate accurately the size of the tree remains a key factor to success, but a fair assessment of the method must also consider the optimization results obtained with the approach. It turns out that, in this problem setting, being able to correctly sort several subproblems by difficulty is at least as important as accurately predicting their hardness. The correlation results show that our approach achieves this objective quite well.

### 5.6.2 Important features

Besides the raw prediction accuracy, we also analyze the importance of each feature on the ability of the learned function to predict a correct output[4]. There are several ways to assess whether a feature matters or not. In this work, we use two techniques as in Section 4.6.2.

First, we examine the so-called 'feature importances', which are values computed by the random forests at learning time and that sum to 1 over all features. The greater the feature importance, the more relevant the feature is. Similarly, we use the cost of omission (COO) (Leyton-Brown et al., 2009) to estimate the impact of a feature on the prediction ability. The cost of omission consists in omitting a given feature during the learning and the testing phases. We can then compute the estimated mean relative error (MRE) obtained without the chosen feature. The difference between the MRE obtained without the feature and the MRE obtained with all features is an image of how important the feature is and is referred to as the cost of omission. If the value of the COO is positive, this means that the feature is important for the prediction. On the other hand, when the COO is negative, it implies that the feature has a negative impact on the prediction accuracy. Small COOs (either positive or negative) indicate that the feature is not very important and could just be a source of noise in the prediction. Note that we use in this work the normalized COO which consists in attributing to the largest positive COO a value of 100, all other COOs are then scaled accordingly. Given that there exist some large errors that have a huge impact on the MRE, and, thus, on the COO, we compute the COOs with different thresholds on the relative errors, just as in the previous section[5]. The results of the feature importances are summarized in Table 5.5 for the 10 most relevant features (according to the feature importances computed by the random forests). The entire list of feature importances is given in Appendix B.1.

The table indicates that feature #66, i.e., the gap decrease at the end of the probing phase, as well as feature #63, i.e., the maximum depth of the probing tree, are very important. Apart from feature #64, i.e., the depth of the last full level in the probing tree (which does not appear in Table 5.5, see Appendix B.1), the importance of the other features seems to be negligible compared to features #66 and #63 (the COOs are rather small). Even if they appear to be less important than the two winners, removing them typically yields positive COOs, which could indicate that their absence is detrimental to the prediction. Additionally, the importance of the features (other than features #66 and #63) tends to decrease when the errors considered in the computation of the COOs increase. This behavior is expected as, for those examples whose predictions are very poor, the features seem to not be particularly able to explain the dynamics of the difficulty estimation problem. For those examples, the features

[4] A probing budget of 5,000 nodes is used to compute the feature importances.

[5] Note that, in this case, the elements that are considered to compute the mean, and thus the COO, are those elements for which the initial relative error (i.e., the relative error obtained with all features) is included in the interval.

| Considered elements | Probe size = 50 | | | | Probe size = 500 | | | | Probe size = 50,000 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | RE (%) | | | | RE (%) | | | | RE (%) | | |
| | % | Mean | Median | Corr. | % | Mean | Median | Corr. | % | Mean | Median | Corr. |
| #nodes | | | | | | | | | | | | |
| $RE_i = 0$ | 28.02 | 0 | 0 | 1 | 28.07 | 0 | 0 | 1 | 28.08 | 0 | 0 | 1 |
| $0 < RE_i \leq 50$ | 20.15 | 22.75 | 21.72 | 0.9961 | 26.09 | 21.95 | 20.69 | 0.9966 | 41.03 | 20.81 | 19.17 | 0.9924 |
| $50 < RE_i \leq 100$ | 12.01 | 72.40 | 71.20 | 0.8758 | 12.14 | 70.79 | 69.21 | 0.9129 | 12.63 | 69.59 | 67.03 | 0.9699 |
| $100 < RE_i \leq 150$ | 4.07 | 123.55 | 123.55 | 0.9995 | 4.74 | 122.62 | 121.56 | 0.9995 | 4.88 | 122.87 | 121.76 | 0.9995 |
| $150 < RE_i \leq +\infty$ | 35.75 | 4,744.63 | 745.99 | 0.7125 | 28.97 | 1,278.33 | 559.44 | 0.8037 | 13.38 | 434.59 | 292.17 | 0.9601 |
| log #nodes | | | | | | | | | | | | |
| $RE_i = 0$ | 28.10 | 0 | 0 | 1 | 28.08 | 0 | 0 | 1 | 28.08 | 0 | 0 | 1 |
| $0 < RE_i \leq 5$ | 25.81 | 2.37 | 2.30 | 0.9859 | 31.13 | 2.30 | 2.22 | 0.9894 | 44.45 | 2.12 | 1.99 | 0.9892 |
| $5 < RE_i \leq 10$ | 19.81 | 7.39 | 7.34 | 0.8966 | 20.35 | 7.33 | 7.25 | 0.8939 | 17.65 | 7.09 | 6.90 | 0.9483 |
| $10 < RE_i \leq 15$ | 12.60 | 12.23 | 12.13 | 0.7401 | 10.88 | 12.23 | 12.10 | 0.7543 | 6.48 | 12.00 | 11.74 | 0.9497 |
| $15 < RE_i \leq +\infty$ | 13.68 | 22.26 | 20.00 | 0.4551 | 9.56 | 21.38 | 19.38 | 0.6970 | 3.33 | 20.66 | 18.54 | 0.9523 |

Table 5.4: Prediction accuracy of the learned complexity function when the probing budget varies. The table reports the mean and the median of the relative errors, as well as the Spearman correlation coefficient between the real outputs and the predictions. The table is read as Table 5.3.

are, in a sense, noise and it is thus no surprise that their importance is small. Interestingly, in the logarithm case, removing the features one at a time seems to have a positive impact on the prediction when the largest errors are considered (negative COOs when $\mathrm{RE}_i > 5$). This would mean that none of the features is important. This is of course misleading and the most likely conclusion that can be drawn from this observation is that single features alone do not explain the output and that the combination of several features is more important than the individual features themselves to yield a correct prediction.

As a closing remark, let us mention that this analysis alone is not conclusive and that further work is required to determine whether the features (apart from features #63 and #66) are really important or just model noise. Additionally, it seems that the combination of several features is rather important for the predictions, but the impact of those interactions cannot be deduced from the sole COOs. Further investigating those relationships would most likely greatly improve our understanding of the subproblem difficulty estimation task.

| Output | | | $RE_i = 0$ | | $0 < RE_i \leq 50$ | | $50 < RE_i \leq 100$ | | $100 < RE_i \leq 150$ | | $150 < RE_i \leq +\infty$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| #nodes | # | FI | MRE | COO | MRE | COO | MRE | COO | MRE | COO | MRE | COO |
| | 66 | 0.2020 | 0.00 | 0 | 100.52 | 100 | 171.40 | 100 | 348.15 | 100 | 992.00 | 100 |
| | 63 | 0.1388 | 137.42 | 100 | 40.19 | 24 | 89.48 | 19 | 153.77 | 14 | 731.56 | -3 |
| | 14 | 0.0705 | 0.00 | 0 | 25.01 | 5 | 71.46 | 1 | 126.78 | 2 | 721.52 | -6 |
| | 18 | 0.0629 | 0.00 | 0 | 26.16 | 6 | 73.29 | 3 | 131.85 | 4 | 736.73 | -1 |
| | 62 | 0.0493 | 0.00 | 0 | 25.53 | 5 | 70.98 | 1 | 127.22 | 2 | 730.60 | -3 |
| | 17 | 0.0410 | 0.00 | 0 | 24.62 | 4 | 71.32 | 1 | 126.28 | 1 | 730.19 | -3 |
| | 1 | 0.0325 | 0.00 | 0 | 24.82 | 4 | 71.43 | 1 | 127.47 | 2 | 741.18 | 1 |
| | 61 | 0.0297 | 0.00 | 0 | 26.12 | 6 | 74.54 | 4 | 135.67 | 6 | 736.86 | -0 |
| | 16 | 0.0277 | 0.00 | 0 | 24.56 | 4 | 72.22 | 2 | 125.86 | 1 | 726.41 | -5 |
| | 60 | 0.0206 | 0.00 | 0 | 25.01 | 5 | 72.23 | 2 | 130.13 | 3 | 740.92 | 1 |
| Output | | | $RE_i = 0$ | | $0 < RE_i \leq 5$ | | $5 < RE_i \leq 10$ | | $10 < RE_i \leq 15$ | | $15 < RE_i \leq +\infty$ | |
| log #nodes | # | FI | MRE | COO | MRE | COO | MRE | COO | MRE | COO | MRE | COO |
| | 63 | 0.9001 | 0.12 | 24 | 3.30 | 79 | 7.45 | 100 | 11.82 | -334 | 19.04 | -807 |
| | 66 | 0.0577 | 0.00 | 0 | 3.59 | 100 | 6.97 | -127 | 11.37 | -761 | 18.83 | -944 |
| | 62 | 0.0199 | 0.00 | 0 | 2.50 | 21 | 7.07 | -82 | 11.94 | -222 | 19.86 | -281 |
| | 18 | 0.0035 | 0.00 | 0 | 2.51 | 22 | 7.06 | -84 | 11.84 | -312 | 19.78 | -334 |
| | 65 | 0.0019 | 0.10 | 20 | 2.56 | 26 | 7.12 | -58 | 11.98 | -183 | 20.01 | -187 |
| | 61 | 0.0015 | 0.00 | 0 | 2.48 | 20 | 7.15 | -44 | 11.93 | -225 | 19.88 | -271 |
| | 40 | 0.0014 | 0.00 | 0 | 2.45 | 17 | 7.08 | -77 | 11.88 | -277 | 19.71 | -380 |
| | 14 | 0.0014 | 0.00 | 0 | 2.47 | 19 | 7.08 | -75 | 11.88 | -274 | 19.77 | -344 |
| | 17 | 0.0012 | 0.00 | 0 | 2.42 | 15 | 7.09 | -71 | 11.93 | -226 | 19.92 | -243 |
| | 59 | 0.0012 | 0.00 | 0 | 2.43 | 16 | 7.15 | -43 | 12.06 | -100 | 20.15 | -100 |

Table 5.5: Feature importances and normalized costs of omission for the 10 most important features. '#' indicates the feature number, ''FI' represents the feature importance computed by the random forests, and 'MRE' and 'COO' represent, respectively, the mean relative error (in %) and the corresponding cost of omission achieved when the feature of interest is removed from the data. The results are reported when the output of the learning model is #nodes and log #nodes. Finally, several cases are considered to compute the COOs, as detailed in Section 5.6.1. The complete list of feature importances is available in Appendix B.1.

## 5.7 Experimental results: optimization

We now turn to experimental results regarding the parallelization of B&B.

### 5.7.1 Parallel optimization

We compare the approach that we propose to the trivial ones in a real parallel optimization setting. We apply the three proposed partitioning schemes to our set of test problems and create increasingly larger partitions of the original optimization trees. We consider two settings. First, we generate partitions whose sizes $k$ are equal to the number of available workers $N$. In that case, each worker, or processor, is responsible for a single subproblem. The second setting focuses on the case where the subproblems outnumber the workers. In that case, $k$ is larger than $N$ and each processor is typically responsible for more than one subproblem.

We assess the performance of the competing partitioning methods thanks to several measures computed from the experimental results. More specifically, when problem $i$ is entirely solved (i.e., all workers are done), the number of nodes processed by each worker $j$ is recorded. The value $n_{ij}$ represents the number of nodes worker $j$ has to explore in order to solve its share of the work that is required to entirely solve problem $i$. When there is only one subproblem assigned to a processor, $n_{ij}$ represents the number of nodes required to solve that subproblem. When the worker is responsible for several subproblems, $n_{ij}$ represents the sum of the nodes required to solve each subproblem assigned to $j$. Several performance measures (average, standard deviation, minimum, maximum, and sum) are then computed for each problem from the stored $n_{ij}$. In other words, we compute, for each problem $i$, the values $\text{mean}_j\, n_{ij}$, $\text{std}_j\, n_{ij}$, $\min_j n_{ij}$, $\max_j n_{ij}$, and $\sum_j n_{ij}$. Then, these values are averaged over all test problems for each partitioning method and each partition size and finally reported in our experimental results.

We perform two types of experiments: with and without communication. In the case without communication, each worker is assigned a given number of subproblems and solves each subproblem independently of the other subproblems and independently of the other workers. In the case where communications are allowed, their sole purpose is to render the best primal bound available to all workers. The communication is thus maintained at its minimum but remains yet very useful to achieve good performance. The communication works as follows. There is, in shared memory, a single scalar that stores the objective value of the best known integral solution. Periodically, after a predefined number of nodes is explored by the worker, the shared primal bound is read and the worker updates its local primal bound accordingly. This allows all processors to be aware of the best available solution in order to early prune unpromising branches of the tree. Moreover, each time a new integral solution is found, the worker responsible for that discovery updates, if necessary, the shared primal bound. This mechanism has proved to be very useful in reducing the total amount of work carried out by all processors, while being very light in terms of communication. We arbitrarily set the default communication interval to 10,000 nodes, but the experimental results suggest that this communication interval can be increased without impacting the optimization results. Note that the optimal communication interval typically depends on

the nature of the considered problems. The proposed value for this parameter is thus not universal and should therefore be adapted based on the studied problems.

The experiments that we carry out are designed to compare several partitioning strategies and to evaluate the impact of the parameters on the optimization results. We compare the two trivial partitioning strategies (random and balanced) described in Section 5.5.2 with the proposed learned partitioning strategy. We define a default learned partitioning strategy that is used in the comparison with the trivial approaches. This default learned partitioning scheme uses $\text{score}_{\text{max}}$ as scoring function, allows 5,000 nodes to be explored in the probing phase of the feature computation, and uses the number of nodes as a difficulty estimator, i.e., the learning algorithm directly predicts the number of nodes of a given subproblem. We also carry out other experiments that are meant to measure the importance of communication and to assess the effect of the parameters of the learned partitioning strategy on the optimization results. Note that, in addition to the three partitioning methods, we also report in the results a so-called 'baseline', which corresponds to the normal optimization of a problem, i.e., starting from the root, the problem is solved to optimality by a single processor.

#### Size of the partition equal to the number of workers: $k = N$

In this first set of experiments, the size of the partition is equal to the number of processors. We generate partitions of size 2 to 24, i.e., we generate $k$ subproblems (with $k = 2 \dots 24$) for each problem to optimize, each subproblem being optimized by a single processor. The performance measures described above and averaged over all problems in our test set are reported, versus the size of the partition, in Figures 5.1-5.7. These figures are just meant to show the trends of the performance measures. The detailed results are given in the form of tables in Appendix B.2.

Figures 5.1 and 5.2 first report the results for the random, balanced, and default learned partitioning strategies without and with communication, respectively. The reported results directly show that our approach always beats the two trivial approaches in every aspect (considering the same communication setup). The results also highlight the importance of communication to achieve good performance in parallel B&B. Overall, the mean number of nodes per processor decreases for all partitioning schemes when the number of generated subproblems increases, i.e., when the size of the partition increases. The same observation can be made for the minimum of the number of nodes across all processors. The maximum of the number of nodes tends to increase when the number of elements in the partition increases, but only when communications are forbidden. This can be easily understood since the deeper the subproblem is in the optimization tree, the less likely it is to contain a good feasible solution that can prune unpromising branches. Also note that the maximum is the most interesting measure because it conditions the speedup. Indeed, even if the total amount of work required to solve a problem is not equal between the serial and the parallel case, the time needed to complete the optimization is conditioned by the processor that takes the most time. In order to analyze the potential speedups, the maximum number of nodes must thus be compared with the number of nodes in the serial case, i.e., the baseline. The speedups obtained when communications are forbidden are very modest. The situation is different when we allow the processors to communicate. Indeed, in that case, our approach achieves a very interesting 4.22 speedup with respect to the serial case when the problem is parallelized

on 24 cores. This has to be compared with the speedups of 1.36 and 1.58 obtained with the random and balanced partitioning schemes. In other words, the approach that we propose indeed achieves speedups that can have an important practical impact.

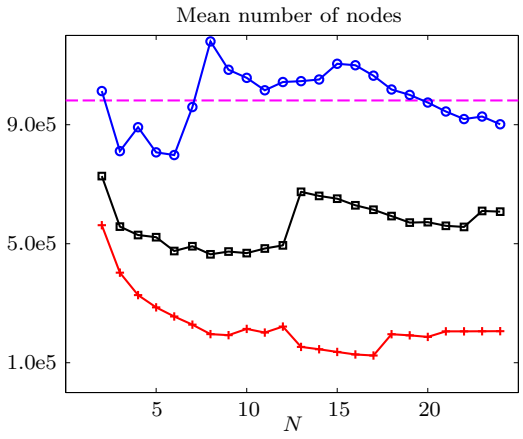
Figures 5.1 and 5.2 highlight the crucial role that communication plays when it comes to achieving interesting speedups. It is not clear, however, whether one should allow a lot of communications or if only a limited amount is enough to obtain good results. Figure 5.3 analyses this aspect. From the figure, it appears that the sensitivity of the optimization results is quite low with respect to the communication interval. These results tend to indicate that only very little communication is actually required to obtain interesting speedups.

The previous figures report the optimization results for the default learned partitioning strategy. Figures 5.4-5.7 report the optimization results when the parameters of the learned strategy vary. Figures 5.4 and 5.5 first report the impact of the scoring function used to create the partition, as well as the impact of the difficulty estimator. From these figures, it appears that $\text{score}_{\text{bal}}$ is outperformed by $\text{score}_{\text{max}}$ in every aspect, even in balancing the workload. The figures additionally show that the choice of the difficulty estimator (#nodes or log #nodes) is not very important. This is quite surprising given that the learning results clearly favor the logarithm of the number of nodes. However, these results strengthen the claim that the learning accuracy is not crucial in this application and that the ranking induced by the predictions is actually more important than the raw prediction accuracy. Finally, Figures 5.6 and 5.7 report the effect of the probing budget on the optimization results. The previous section clearly indicates that increasing the probing budget improves the learning accuracy. A similar phenomenon appears in the figures. Indeed, the larger the size of the probing tree, the better the optimization results. It is interesting to note though that, even without communication, a speedup of $\approx 4$ can be achieved when a probing budget of 50,000 nodes is allowed for the computation of the features. The speedup becomes even better ($\approx 7$) when communications are allowed. On the other hand, when the probing budget is very small (50 nodes), the optimization results are not very good without communication, but compare very nicely with the other probing sizes when communications are allowed.

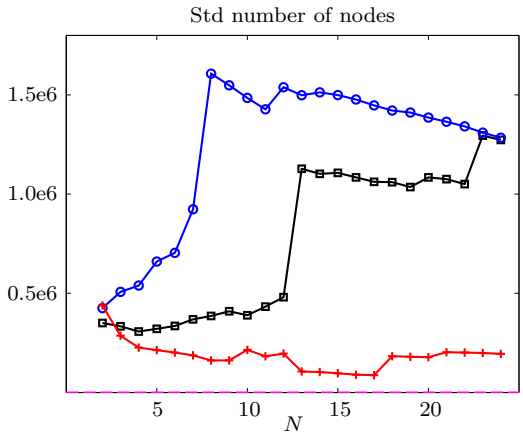
As a final remark, it is interesting to emphasize that the computed speedups are lower bounds on the attainable speedups. Indeed, in this work, we do not perform dynamic load balancing since we only distribute the work to each processor before the optimization starts. If a dynamic load balancing scheme is used to further improve the work equilibrium between the processors, it is conceivable that much higher speedups can be achieved. Indeed, in that case, other measures, such as the mean and the minimum numbers of nodes per processor, should be used to enrich our analysis. Given that the mean and the minimum workload per processor are both very low with our method, it is fair to expect greater gains in computation time if a dynamic load balancing scheme is used, together with our method, in order to redistribute the work from the busiest processor to the idle ones.

#### Size of the partition greater than the number of workers: $k > N$

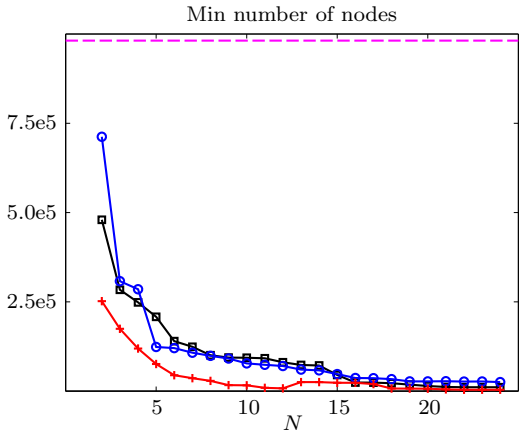
The last set of experiments that we propose focuses on the parallel optimization of our set of test problems when the number $k$ of generated subproblems is greater than the number $N$ of workers, i.e., $k > N$. In order to do so, we generate a certain number $k$ of subproblems with



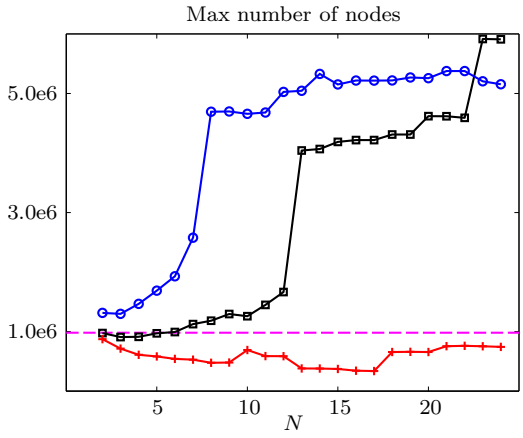

(a) Mean number of nodes per processor



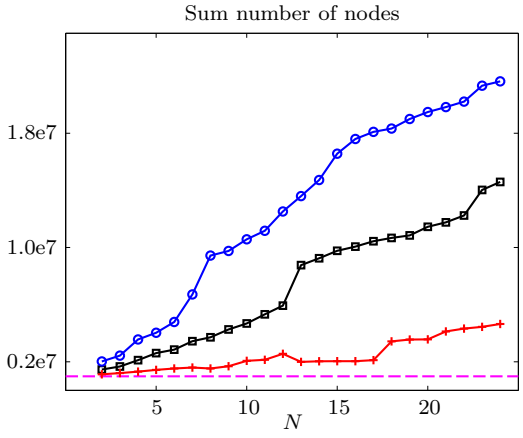

(b) Std.dev. of number of nodes per processor




(c) Minimum number of nodes per processor




(d) Maximum number of nodes per processor




(e) Total number of nodes

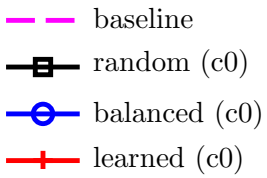
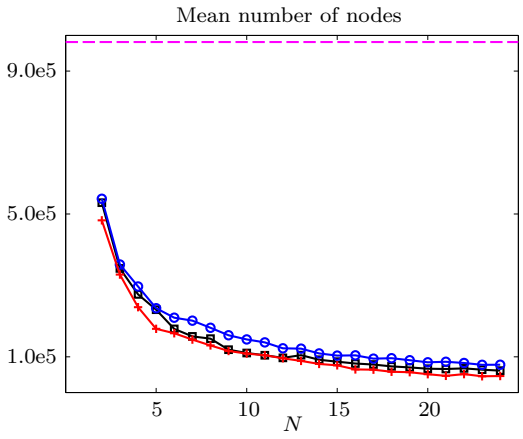
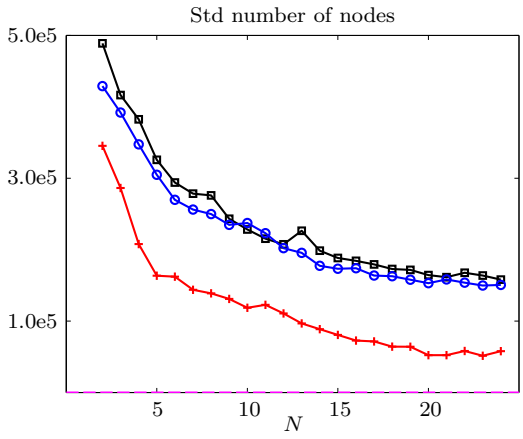


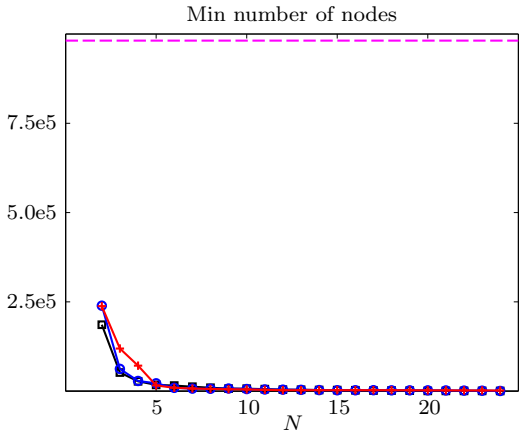

Figure 5.1: Parallel optimization results ($k = N$) without communication. Comparison between the two trivial partitioning strategies (random and balanced) and the default learned partitioning strategy (score=$\text{score}_{\max}$, probe size=5,000, and output=#nodes).



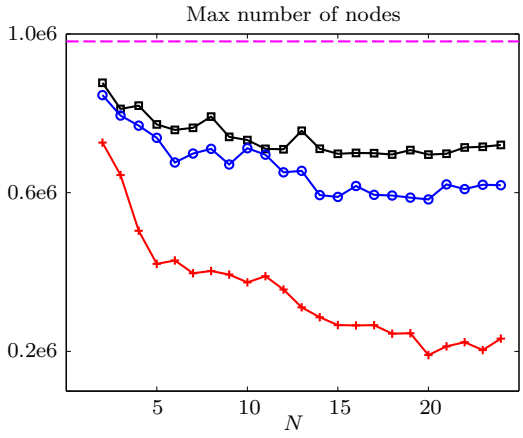

(a) Mean number of nodes per processor



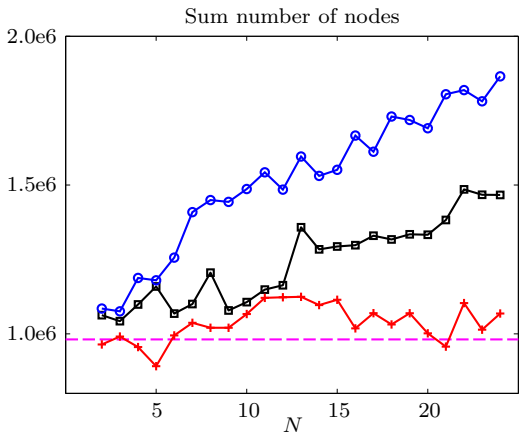

(b) Std.dev. of number of nodes per processor




(c) Minimum number of nodes per processor




(d) Maximum number of nodes per processor




(e) Total number of nodes

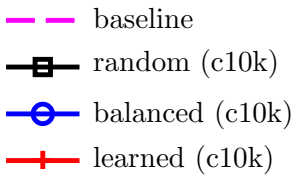
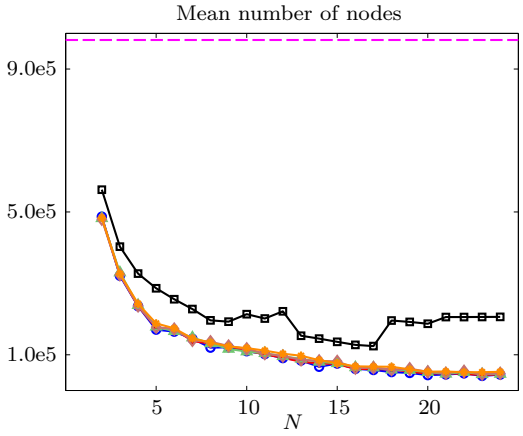
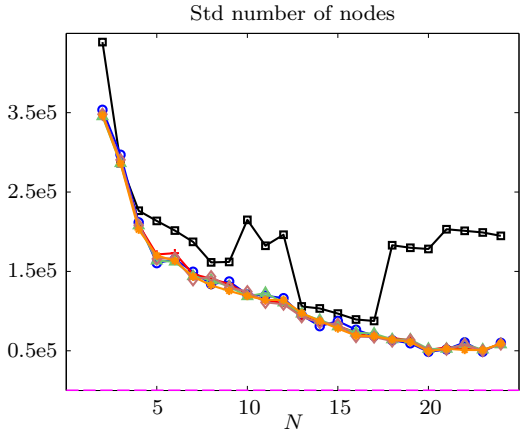


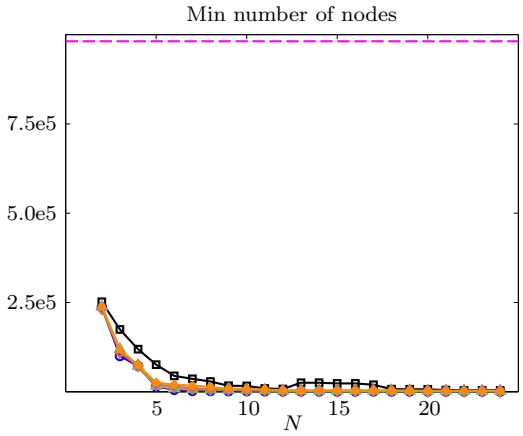

Figure 5.2: Parallel optimization results ($k = N$) with communication every 10,000 nodes. Comparison between the two trivial partitioning strategies (random and balanced) and the default learned partitioning strategy (score=$\text{score}_{\max}$, probe size=5,000, and output=#nodes).



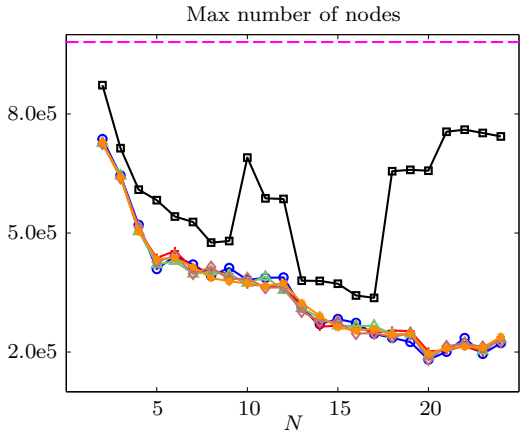

(a) Mean number of nodes per processor



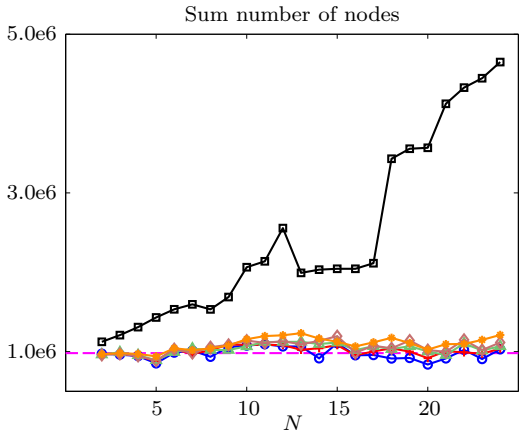

(b) Std.dev. of number of nodes per processor




(c) Minimum number of nodes per processor




(d) Maximum number of nodes per processor




(e) Total number of nodes

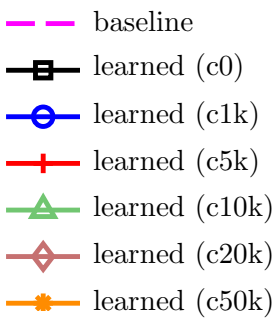
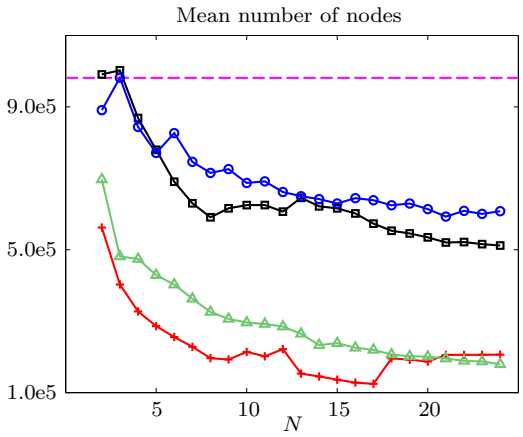
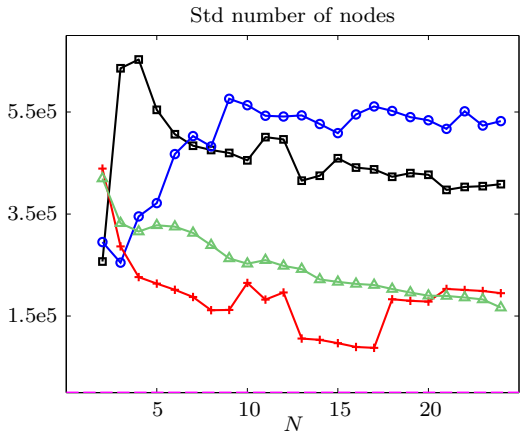


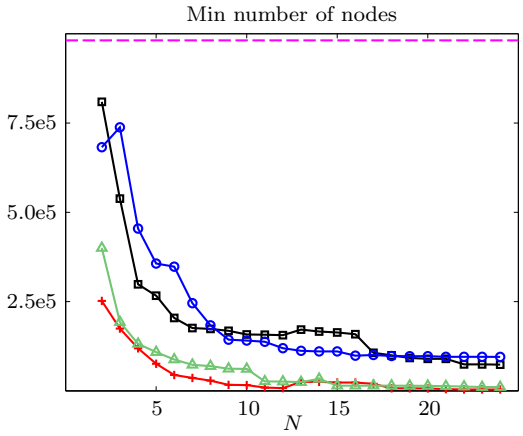

Figure 5.3: Parallel optimization results ($k = N$) for the default learned partitioning strategy (score=$\text{score}_{\max}$, probe size=5,000, and output=#nodes) with different communication intervals. 'c0' indicates no communication and 'cx' indicates communication every x nodes.



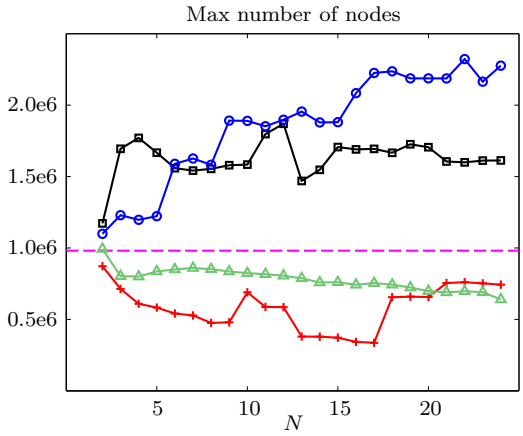

(a) Mean number of nodes per processor



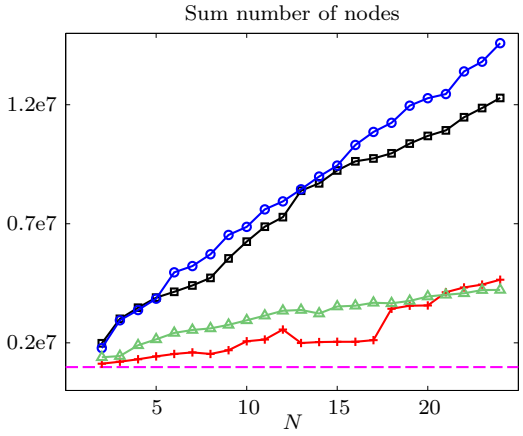

(b) Std.dev. of number of nodes per processor




(c) Minimum number of nodes per processor




(d) Maximum number of nodes per processor




(e) Total number of nodes

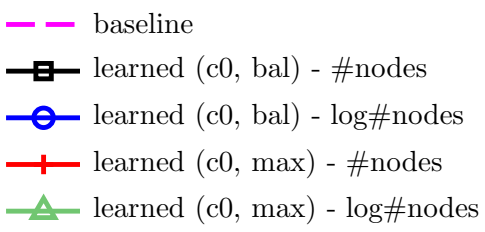
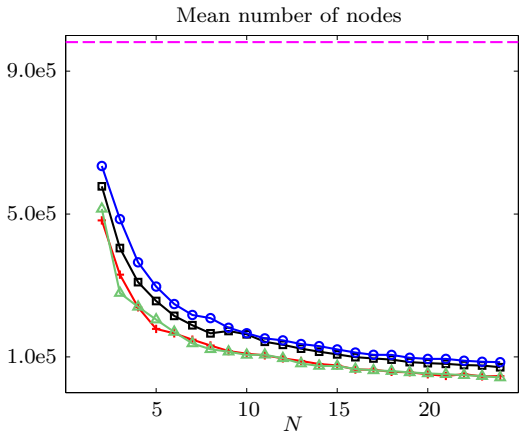
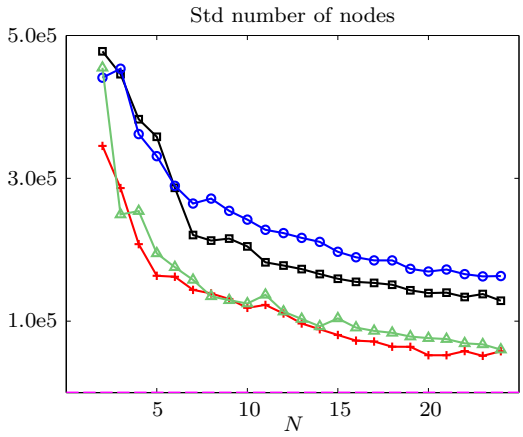


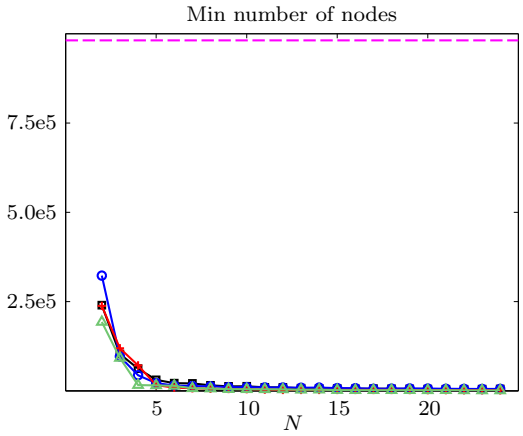

Figure 5.4: Parallel optimization results ($k = N$) without communication for the learned partitioning strategy (probe size=5,000): impact of the scoring functions $\text{score}_{\text{bal}}$ ('bal') and $\text{score}_{\text{max}}$ ('max'), and impact of the output predicted by the learning algorithm (either #nodes, or log #nodes).



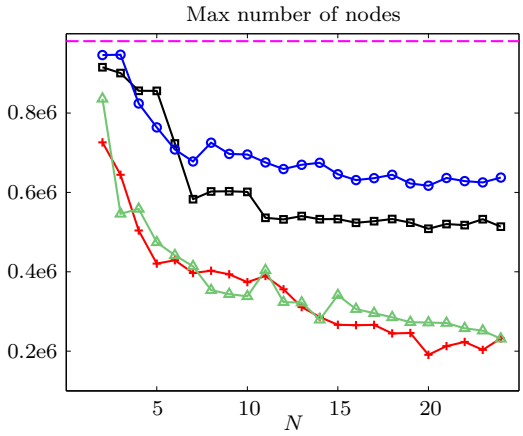

(a) Mean number of nodes per processor



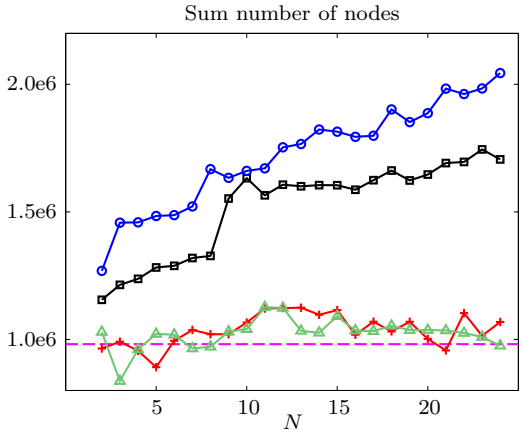

(b) Std.dev. of number of nodes per processor




(c) Minimum number of nodes per processor




(d) Maximum number of nodes per processor




(e) Total number of nodes

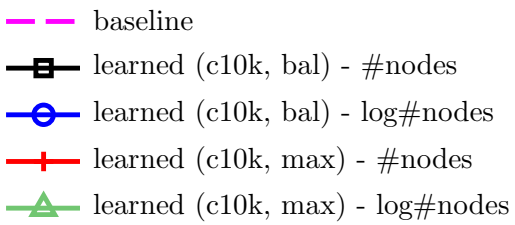
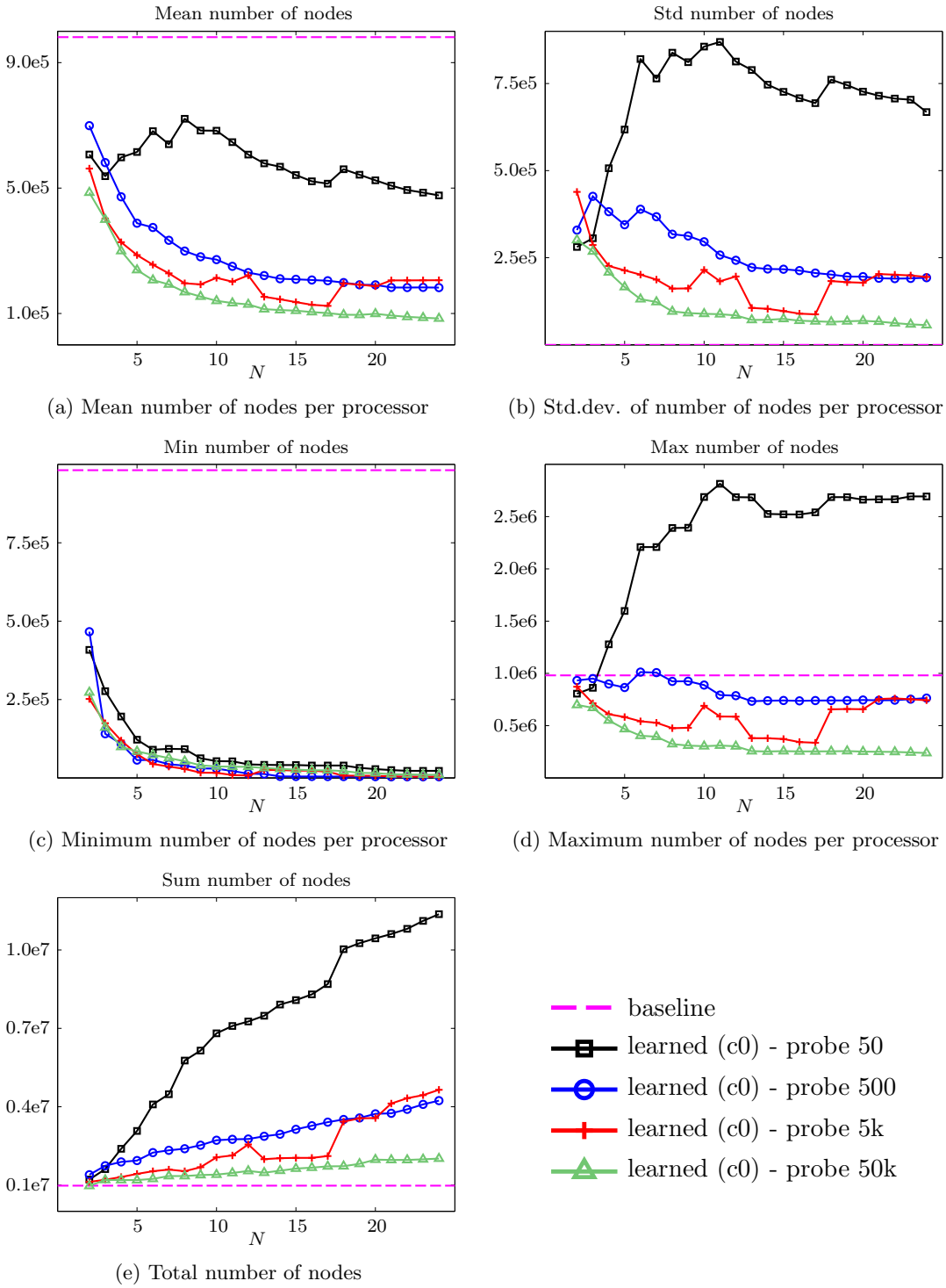



Figure 5.5: Parallel optimization results ($k = N$) with communication every 10,000 nodes for the learned partitioning strategy (probe size=5,000): impact of the scoring functions $\text{score}_{\text{bal}}$ ('bal') and $\text{score}_{\text{max}}$ ('max'), and impact of the output predicted by the learning algorithm (either #nodes, or log #nodes).




(a) Mean number of nodes per processor

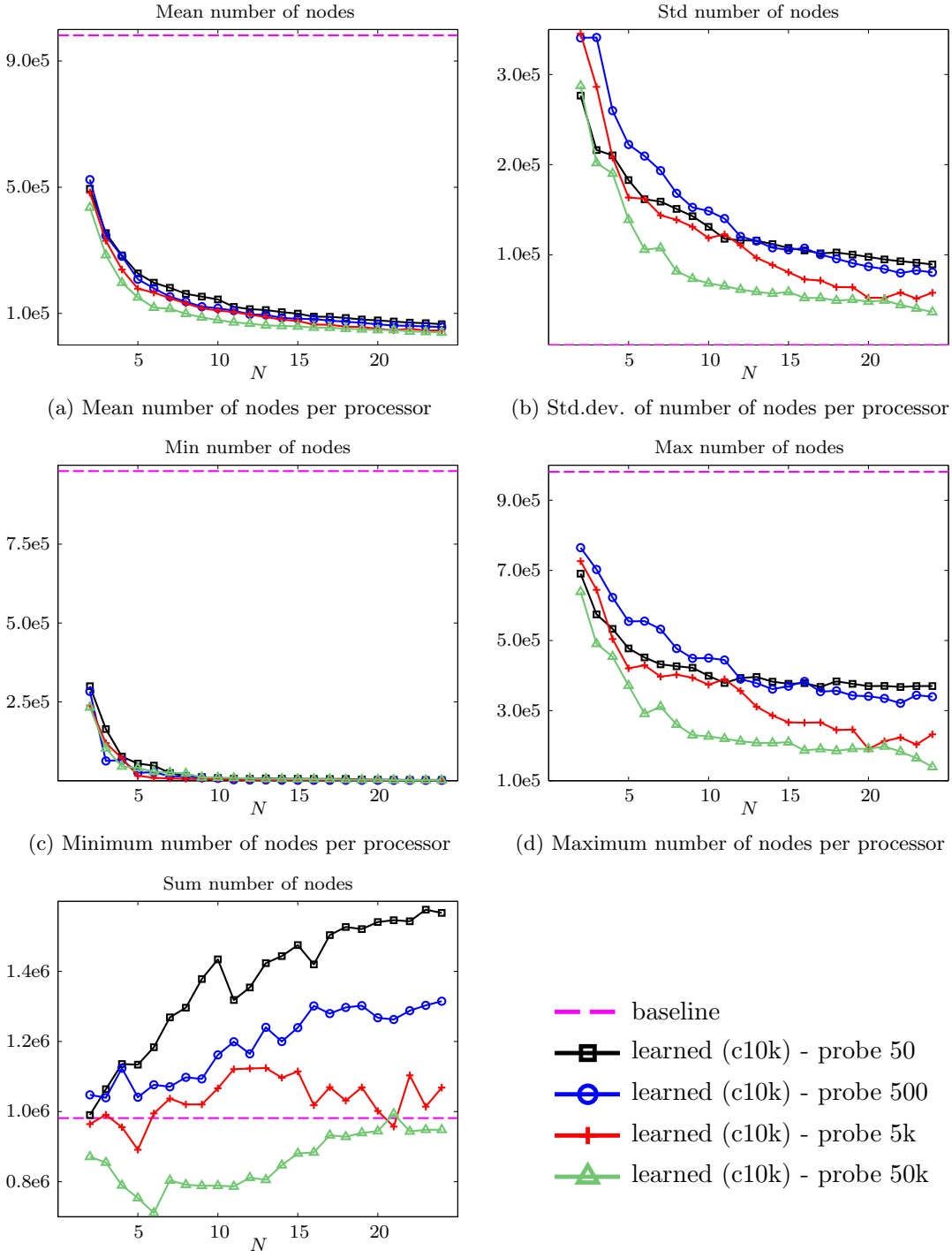
(b) Std.dev. of number of nodes per processor

(c) Minimum number of nodes per processor

(d) Maximum number of nodes per processor

(e) Total number of nodes

Figure 5.6: Parallel optimization results ($k = N$) without communication for the learned partitioning strategy (score=$\text{score}_{\max}$ and output=#nodes): impact of the probing budget.




(a) Mean number of nodes per processor

(b) Std.dev. of number of nodes per processor

(c) Minimum number of nodes per processor

(d) Maximum number of nodes per processor

(e) Total number of nodes

Figure 5.7: Parallel optimization results ($k = N$) with communication every 10,000 nodes for the learned partitioning strategy (score=$\text{score}_{\max}$ and output=#nodes): impact of the probing budget.

each considered partitioning scheme. Then, the subproblems are distributed across the $N$ available processors, either by randomly attributing $k/N$ subproblems to each processor (for the random and balanced partitioning schemes), or by using Algorithm 4 (for the learned partitioning scheme). Note that the number of subproblems generated with the learned partitioning scheme is at most $k$, but may be less given that Algorithm 3 provides another stopping criterion that can be used to stop the generation of the subproblems before the limit $k$ is reached. Table 5.6 reports, in the case where the number of generated subproblems is greater than the number of workers, the same performance measures as those presented in Figures 5.1 and 5.2. Note that, in this case, the performance measures (average, standard deviation, etc.) are computed from the total number of nodes processed by each worker, which corresponds to the sum of the number of nodes required by each subproblem attributed to the given worker. The total number of nodes processed by each worker is thus an image of the total amount of work carried out by a worker.

Table 5.6 reports the parallel optimization results when the number $k$ of generated subproblems is greater than the number $N$ of processors. We compare the two trivial approaches (random and balanced) with the default learned approach (score=$\text{score}_{\max}$, probe size=5,000, and output=#nodes). The table first indicates that the parallel optimization without communication does not compare very well with the single threaded case. Indeed, the maximum number of nodes that a worker processes is always greater than in the single threaded case. This implies that the parallel optimization does not terminate before the single threaded B&B. However, the approach that we propose compares favorably with the trivial partitioning schemes. Indeed, with our method, the mean number of nodes that a processor explores decreases, as well as the standard deviation. This implies that, on average, the number of nodes per worker is less than for a single threaded optimization and that the load is acceptably well balanced. Things are different when communications are allowed between the processors. Indeed, while the trivial partitioning schemes still perform very poorly compared to the single threaded optimization, the proposed method, on the other hand, exhibits very interesting performance. Indeed, in that case, the obtained speedups between the single threaded baseline and the learned partitioning scheme are equal to 4.07 and 5.61, for 12 and 24 processors, respectively. Similarly to the case without communication, the mean number of nodes and the standard deviation per processor are reduced compared to the trivial partitioning schemes, which shows that the discrepancies between the workloads of the workers are limited.

Finally, it is interesting to compare the situation where each worker is assigned only one subproblem, with the situation where they are responsible for multiple subproblems. This analysis can be carried out by comparing the results from Table 5.6 with those from Tables B.9-B.13 in Appendix B.2. Comparing both sets of results for 12 and 24 processors yields the following observations. First, the mean and the total number of nodes are roughly equal between both setups. Additionally, we see that the standard deviation per processor decreases when more subproblems are assigned to a single worker, which is a display of better load balance. Lastly, we observe that the difference between the maximum and the minimum number of nodes decreases when the processors are assigned more than one subproblem, which tends, again, to show that the work is better balanced in that case. Overall, the conclusion that can be drawn is that allocating more than one subproblem to each processor does not increase the total amount of work to be carried out, but sensibly reduces the unbalance between the workers.

| | $N$ | $k$ | Mean | Std | Min | Max | Sum |
|---|---|---|---|---|---|---|---|
| baseline | 1 | 1 | 9.81e+05 | - | 9.81e+05 | 9.81e+05 | 9.81e+05 |
| Without communication (c0) | | | | | | | |
| random | 12 | 240 | 7.94e+06 | 5.70e+06 | 1.73e+06 | 1.97e+07 | 9.53e+07 |
| random | 24 | 480 | 6.29e+06 | 5.76e+06 | 1.00e+06 | 2.14e+07 | 1.51e+08 |
| balanced | 12 | 240 | 1.55e+07 | 8.81e+06 | 4.53e+06 | 3.33e+07 | 1.86e+08 |
| balanced | 24 | 480 | 1.25e+07 | 8.15e+06 | 3.28e+06 | 3.79e+07 | 3.00e+08 |
| learned | 12 | 240 | 5.96e+05 | 2.38e+05 | 1.85e+05 | 9.89e+05 | 7.16e+06 |
| learned | 24 | 480 | 5.54e+05 | 3.03e+05 | 1.22e+05 | 1.18e+06 | 1.33e+07 |
| With communication every 10,000 nodes (c10k) | | | | | | | |
| random | 12 | 240 | 1.67e+05 | 1.89e+05 | 4.22e+04 | 6.94e+05 | 2.00e+06 |
| random | 24 | 480 | 7.98e+04 | 1.59e+05 | 1.03e+04 | 7.60e+05 | 1.91e+06 |
| balanced | 12 | 240 | 1.99e+05 | 1.50e+05 | 5.22e+04 | 5.47e+05 | 2.39e+06 |
| balanced | 24 | 480 | 1.44e+05 | 1.27e+05 | 3.29e+04 | 5.73e+05 | 3.46e+06 |
| learned | 12 | 240 | 9.38e+04 | 6.14e+04 | 1.90e+04 | 2.41e+05 | 1.12e+06 |
| learned | 24 | 480 | 4.58e+04 | 3.95e+04 | 6.80e+03 | 1.75e+05 | 1.10e+06 |

Table 5.6: Parallel optimization results with and without communication when the number of generated subproblems ($k$) is greater than the number of processors ($N$), i.e., $k > N$. The table compares the two trivial approaches (random and balanced) with the default learned partitioning approach (score=score$_{\max}$, probe size=5,000, and output=#nodes). The values reported are computed in the same way as in the previous section. Note that, for the learned partitioning scheme, the $k$ indicates the maximum number of elements in the partition. The real number of generated subproblems is different for each problem and depends on the stopping criterion given in Algorithm 3.

## 5.8 Concluding remarks and outlooks

In this chapter, we propose a new approach to split the optimization of a single problem into several independent parts that can be solved by several workers in parallel, with the goal that the amount of work given to each processor is well balanced between the workers. The approach consists in creating a function, with the use of machine learning techniques, that is able to estimate the number of nodes, hence the amount of work, that a solver needs to process in order to solve to optimality a subproblem of the original problem. To this end, we develop a set of features that are used to characterize a given subproblem in the B&B tree and use these features as input of the learned function in order to predict the expected number of nodes required to solve the given subproblem to optimality. These estimates are then used to create a partition of the original optimization tree so that one or several elements of the partition can be given to each worker. The experiments show that our approach succeeds in balancing the amount of work between the processors and that interesting speedups can be achieved with little effort.

Further research orientations include the development of more relevant features that would better grasp the dynamics of the considered problems in order to better predict the subproblem difficulty. Another research direction is to implement the proposed framework on massively parallel computers to better understand how the speedups and processor utilizations evolve when the original work is split into a very large number of independent parts.

Finally, let us emphasize the fact that the same framework can be transposed to any problem class with only minor adaptations to the proposed method (e.g., adaptation to the features). Indeed, although this work totally focuses on a single class of MIP problems, namely the unit commitment problems, no specific information about the problem class is used in the development of the method. This implies that the method is entirely generic and could be applied to any problem class with great success. If modifications should be made to the method (and modifications may not be compulsory for the method to work), they would mainly concentrate on the features and would most likely be minor.

# Chapter 6

# Machine learning and the theory of linear optimization

## 6.1 Introduction

In the previous chapters, we have shown how machine learning techniques can be used to improve several components of a traditional optimization algorithm, namely the branch-and-bound algorithm. The proposed approaches are practical in the sense that they are meant to address pragmatic issues faced by the solvers. However, even if the proposed methods are mainly driven by practical goals, it is possible to use them to gain theoretical insights about the solution process. For instance, in the case of the branching strategy, it is possible to use our techniques to determine which features are important so as to better theoretically understand what conditions good branchings for specific problems. In this chapter, we push this idea further and, leaving aside the practical aspects, we turn our attention to theoretical challenges in optimization. We show how learning techniques can provide interesting tools to theoretically study optimization questions. More specifically, we theoretically investigate the worst case complexity of the simplex algorithm, which is a central matter in the theory of linear optimization.

In this chapter, we consider linear programming problems (LP problems) (see Chapter 2, Equation (2.2)) solved by the simplex algorithm and we study the worst case complexity of the algorithm in very general situations. It is well known that linear programming problems can be solved in polynomial time using specific algorithms (see, e.g., Khachiyan, 1980; Karmarkar, 1984; Nesterov and Nemirovskii, 1994). These algorithms are known to have a 'weakly polynomial time complexity', which is defined by the fact that the number of iterations of the algorithm polynomially depends on the size of the problem (number of variables and constraints) and on the logarithm of the numbers that appear in the problem. On the contrary, the number of iterations required by 'strongly polynomial time' algorithms only depends on the size of the problem (number of variables and constraints) and not on the magnitude of the coefficients that appear in the problem (Megiddo, 1986). For the time being, strongly polynomial algorithms are not known for linear programming in general (some algorithms are strongly polynomial for certain classes of LP problems).

In this work, we investigate whether a strongly polynomial version of the simplex method exists for any linear programming problem. To this end, we try to find so-called polynomial pivoting rules that, when used within the simplex skeleton, will prove that the complexity of the algorithm is polynomial in the dimensions of the problem. Note that a pivoting rule is said to be polynomial if the pivoting rule induces a run of the algorithm that terminates after a polynomial number of iterations and if each iteration is itself polynomial in the dimensions of the problem. In order to find such polynomial pivoting rules, we propose to formulate the simplex algorithm applied to an optimization problem as a Markov decision process (MDP). We then cast the search for a polynomial pivoting rule as a reinforcement learning problem, which is a learning framework that provides tools to solve MDPs. Once this parallel is drawn, the theory of reinforcement learning can be used to analyze from a theoretical perspective the proposed approach.

Without going too much into the details here, let us briefly explain why we formulate the simplex algorithm as an MDP and let us concisely lay down the main steps of the proposed idea. The simplex algorithm is an iterative algorithm that goes from basic feasible solution to basic feasible solution until the optimal solution is found. These transitions are discrete. Indeed, even if one iteration of the simplex algorithm involves continuous variables changing values, the basic feasible solutions can be described in a discrete manner by just identifying the variables that belong to the basis. Additionally, these transitions are controlled by the algorithm by means of discrete decisions: the choice of the pivoting variable. This setting is very similar to Markov decision processes (MDPs) where a system (here the simplex algorithm) experiences state changes when actions (here the pivoting decisions) are taken. MDPs are widely used in control problems where they model how the systems evolve when control actions are applied. Because of their wide application range, many different tools have been developed to solve MDPs and most of these tools have undergone extensive theoretical study, which we leverage here. Among the possible tools that can be used to solve MDPs, we focus on reinforcement learning (RL) techniques because of the richness of the available theory. Once the parallel between the simplex and MDPs is drawn, it is easy to formulate the simplex as an MDP and to make use of RL algorithms and of RL theory to study the simplex from the perspective of MDPs.

The purpose of this chapter is to propose a novel approach to tackle the question of the worst case complexity of the simplex. We introduce some tools and give some preliminary theoretical results, but the chapter merely presents and describes the proposed approach and does not solve the question of the worst case complexity of the simplex, which remains open. The chapter proposes a framework and gives some initial results on which a complete solution can be built, which is a required first step. The contributions of the chapter lie in the formulation of the simplex as an MDP and in the proposed idea that learning techniques can be used to gain theoretical insights about the complexity of the algorithm. Furthermore, the main stumbling blocks are identified, so that the main difficulties are already highlighted. However, we believe that a considerable amount of work is still needed in order to yield an entire and satisfactory theoretical analysis of the problem on top of what we present here. Moreover, it is our belief that close collaboration between the optimization and learning research communities is the key to bringing the proposed approach to a complete solution. Hopefully, the preliminary results that we present here will encourage such collaborations.

In the following, we first redefine, in Section 6.2, LP problems and the simplex algorithm from a geometric perspective. Then, Section 6.3 discusses the complexity of the simplex method. Section 6.4 next defines Markov decision processes (MDP) as well as the reinforcement learning paradigm and casts the simplex algorithm as an MDP. Section 6.5 dives deep into the problem and describes how the theory of reinforcement learning can provide interesting tools to study theoretical questions about the simplex method. Later on, we move a bit away from theory in Section 6.6 where we push the analysis a bit further and also discuss some practical aspects of the proposed approach. Finally, Section 6.7 draws the conclusions of this chapter.

## 6.2 Linear programming problems, simplex algorithm, and geometry

In this section, we remind the basic definitions regarding linear optimization already given in Chapter 2. We nonetheless reintroduce them here because they are presented in a slightly different way that serves the purpose of this chapter. Indeed, besides their algebraic formulation, optimization problems can be analyzed and studied under the light of elementary geometry. Actually, the algebraic components of the LP problem (e.g., the objective function, the constraints) as well as the algebraic objects handled during the solution of that problem (e.g., the basic feasible solutions) have a clear counterpart in the geometric world. In what follows, we give a simplified introduction to the important algebraic concepts, together with their geometric twins, that are required to understand this work, as well as a description of the simplex algorithm based on these definitions.

Note that the discussion is voluntarily simplified here in order to focus on the most important concepts only. We refer the reader to appropriate literature for further details and formal introduction of the mentioned mathematical concepts (see, e.g., Bertsimas and Tsitsiklis, 1997).

### 6.2.1 Linear programming problems and their geometry

As a reminder, let us redefine linear programming (LP) problems, which are problems of the form

$$\begin{aligned} \min \quad & \boldsymbol{c}^\top \boldsymbol{x} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} \\ & \boldsymbol{x} \in \mathbb{R}^n, \end{aligned} \tag{6.1}$$

where $\boldsymbol{c} \in \mathbb{R}^n$, $\boldsymbol{A} \in \mathbb{R}^{m \times n}$, and $\boldsymbol{b} \in \mathbb{R}^m$ denote the cost coefficients, the coefficient matrix, and the right-hand side, respectively. Note that the number of constraints $m$ is assumed to be larger than the number of variables $n$, i.e., $m > n$.

The concept of polyhedron is central to linear programming. Formally, polyhedra are defined as follows.

**Definition 1** (Polyhedron)**.** *A polyhedron $P$ is defined as a set of $\mathbb{R}^n$ bounded by $m$ hyperplanes, i.e.,*

$$P = \{\boldsymbol{x} : \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}\}.$$

Because polyhedra represent the sets of feasible solutions of the LP problems, it is easy to see why polyhedra are crucial to linear programming. This nice parallel that can be drawn between a geometric object (a polyhedron) and algebraic elements (the constraints that define the set of feasible solutions) is at the root of the geometric-algebraic interpretation of linear programming. The polyhedra may be unbounded or bounded, in which case they are referred to as polytopes (see, e.g., Nemhauser and Wolsey, 1988). If the polyhedra are non-empty and bounded, i.e., when they are non-empty polytopes, there exists in general an optimal solution to the corresponding optimization problems. When a polyhedron is unbounded, a solution may not exist depending on the cost function and, more specifically, on the orientation of the cost function with respect to the polyhedron. In this work, we solely consider non-empty bounded polyhedra.

In addition to the previous definition, we also define an extreme point of the polyhedron, or vertex, as follows.

**Definition 2** (Vertex or extreme point of a polyhedron)**.** *A point $\boldsymbol{x}$ of a polyhedron $P$ is a vertex if it cannot be expressed as a convex combination of two other elements of the polyhedron, i.e.,*

$$\nexists\ \boldsymbol{y}, \boldsymbol{z} \in P \setminus \boldsymbol{x}, \lambda \in [0,1] \ \text{ such that } \boldsymbol{x} = \lambda\boldsymbol{y} + (1-\lambda)\boldsymbol{z}.$$

From the geometric point of view, the vertices are located at the intersection of $n$ facets of the polyhedron (i.e., at the intersection of $n$ constraints). Vertices are important objects to understand the mechanisms of the simplex and are related to the concept of basic feasible solution, whose definition is given now.

**Definition 3** (Basic feasible solution)**.** *Let $\boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}$ define a polyhedron, a basic feasible solution (BFS) is a point $\boldsymbol{x} \in \mathbb{R}^n$ such that*

1. *$n$ linearly independent inequalities are satisfied with equality, i.e., $\boldsymbol{x}$ satisfies*

$$\boldsymbol{A}_{i:}\boldsymbol{x} = b_i,\ \forall i \in V;$$

2. *all inequalities are satisfied, i.e., $\boldsymbol{x}$ satisfies*

$$\boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b};$$

*where $V$ is a set containing the $n$ indices of the constraints satisfied with equality by $\boldsymbol{x}$, and $\boldsymbol{A}_{i:}$ represents the $i$th row of matrix $\boldsymbol{A}$. For a given BFS, the constraints $\boldsymbol{A}_{i:}\boldsymbol{x} = b_i$ with $i \in V$ are said to be tight or active.*

From the previous definition, it appears that a BFS is defined by $n$ active linearly independent inequalities. If more than $n$ inequalities are active at a given BFS (they are not linearly independent anymore), we say that the BFS is degenerate. Degeneracy happens when more than $n$ inequalities are active at one or several BFS. Without loss of generality of our analysis, we can safely assume that the considered polyhedra are non-degenerate (i.e., all BFS are non-degenerate) and, from now on, we thus focus on non-degenerate polyhedra only (De Loera,

2011). Note that, because we assume that there is no degeneracy in the studied problems, the theory of optimization guarantees that there is a one-to-one correspondence between a basic feasible solution and a vertex of the polyhedron. This implies, in particular, that an optimal solution of the problem corresponds to a vertex of the polyhedron. In the following, we use indistinguishably vertex, extreme point, and BFS to refer to the same elements.

**Definition 4** (Number of vertices of a polyhedron)**.** *The number $\mathcal{V}$ of vertices of a polyhedron is bounded by*

$$\mathcal{V} \leq \binom{m}{n} = \frac{m!}{n!(m-n)!} \leq m^n.$$

In particular, the number $\mathcal{V}$ of vertices of a polyhedron is typically an exponential function of the parameters $n$ and $m$ of the problem. To illustrate this, let us consider one of the simplest polytopes: the unit hypercube, which corresponds to the $n$-dimensional generalization of the three-dimensional cube. In mathematical terms, the unit hypercube of dimension $n$ is defined by $2n$ hyperplanes given by

$$0 \leq x_i \leq 1, \ \forall i = 1 \dots n,$$

where the number $m$ of constraints is equal to $2n$. The number of vertices of a hypercube is equal to $2^n$ and, equivalently, to $2^{\frac{m}{2}}$. This example illustrates that, even with very simple constructions, the number of vertices can grow exponentially fast as a function of the parameters of the problem (either $n$ or $m$). We describe later why this growth is an issue when it comes to solving LP problems.

In the theory of optimization, the concept of adjacent basic feasible solutions is of primary importance. Let us now define this algebraic notion.

**Definition 5** (Adjacent basic feasible solutions)**.** *Two basic feasible solutions (defined by the sets $V_1$ and $V_2$ of active constraints, respectively) are said to be adjacent if and only if*

$$|V_1 \cap V_2| = n - 1,$$

*i.e., the two BFS share the same $n-1$ active linearly independent inequalities and differ only by one active constraint.*

Since there is a strict correspondence between BFS and vertices (the polyhedra are non-degenerate), two adjacent vertices are defined in a similar way as two adjacent BFS. From a geometric perspective, two adjacent vertices are also said to be 'neighbors' and the neighborhood of a vertex refers to the set of its adjacent vertices (or adjacent BFS). As mentioned above, the vertices are located at the intersection of $n$ linearly independent constraints. In order to better understand the notion of adjacent vertices, let us define the edges of a polyhedron.

**Definition 6** (Edge of a polyhedron)**.** *An edge of a polyhedron corresponds to the intersection of $n-1$ linearly independent constraints. In algebraic terms, an edge corresponds to*

$$\{\boldsymbol{x} : \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} \wedge \boldsymbol{A}_{i:}\boldsymbol{x} = b_i, \ \forall i \in E\},$$

*where $E$ is a set containing the $n-1$ indices of the constraints satisfied with equality, and $\boldsymbol{A}_{i:}$ represents the $i$th row of matrix $\boldsymbol{A}$.*

In particular, the above definition implies that two adjacent vertices belong to the same edge. An edge can thus be seen as a 'link' between two adjacent vertices.

With these definitions at hand, it appears quite directly that it is possible to travel through all vertices of a polyhedron (and, hence, through all BFS) by following only the edges of the polyhedron. It is thus possible to find a path, between any arbitrary pair of vertices, that follows only the edges of the polyhedron. This leads to the following definitions of the path between two vertices and of the diameter of a polyhedron.

**Definition 7** (Path between two vertices)**.** *In a polyhedron, a path between a vertex $V_1$ and a vertex $V_2$ is defined as a sequence of edges that connect a sequence of adjacent vertices that starts with $V_1$ and ends with $V_2$. The length of a path is equal to the number of edges in the path.*

**Definition 8** (Diameter of a polyhedron)**.** *The diameter of a polyhedron is defined as the maximum length of the shortest path between any two arbitrary vertices of the polyhedron.*

The diameter of a polyhedron is central to the concepts and theory developed in this chapter. The diameter has attracted a lot of attention from the research community because it is related to the efficiency of optimization algorithms (see next section). There exist many studies that investigate the diameters of well known polytopes, but these are not discussed here. We merely report a very important conjecture, the Hirsch conjecture, which is the basis of our theoretical reasoning. This conjecture, formulated by Warren Hirsch in 1957, gives an upper bound on the diameter of any polyhedron (Bertsimas and Tsitsiklis, 1997; De Loera, 2011). This conjecture has been recently disproved (Santos, 2012).

**Conjecture 1** (Hirsch conjecture)**.** *The diameter of any polyhedron is no more than*

$$m - n.$$

Even if the Hirsch conjecture is now known to be false, the research community still believes that a modified version of the Hirsch conjecture is true (see, e.g., De Loera, 2011). A modified version of the Hirsch conjecture could be as follows.

**Conjecture 2** (Modified Hirsch conjecture)**.** *The diameter of any polyhedron is no more than*

$$p(m, n),$$

*where $p(\cdot, \cdot)$ is a polynomial of the input arguments.*

If the proposed modified Hirsch conjecture is true, this implies that it is possible to reach any vertex from any starting vertex by traversing at most a polynomial number of edges (polynomial in the dimensions $m$ and $n$ of the polyhedron). From now on, we make the working assumption that the proposed modified Hirsch conjecture is true.

**Assumption 1.** *The modified Hirsch conjecture as presented in Conjecture 2 is true, i.e., the diameter of any polyhedron is bounded by a polynomial in the dimensions $n$ and $m$ of the polyhedron.*

### 6.2.2 The simplex algorithm

The simplex algorithm (Dantzig, 1987), originally proposed by Dantzig in 1947, is one of the most efficient and most widely used optimization algorithms to solve LP problems. This section does not provide a detailed description of the algorithm, but merely presents its mechanisms from the geometrical perspective. We refer the reader to Chapter 2 of this document or to other appropriate material for a more complete description of the method (see, e.g., Chvatal, 1983; Bertsimas and Tsitsiklis, 1997; Vanderbei, 2014; Nemhauser and Wolsey, 1988).

The simplex method is an iterative algorithm that travels through the vertices (or BFS) of a given polyhedron until the vertex corresponding to the optimal basic feasible solution is found. The method starts at an arbitrary vertex (or BFS). Then, each iteration consists in choosing a neighboring vertex that, ideally, has not been visited yet. The algorithm next moves to that vertex and the process is repeated until the optimal vertex is reached. Note that the algorithm follows the edges of the polyhedron, since it iteratively visits adjacent vertices (adjacent BFS).

The only decision that has to be taken by the algorithm consists in choosing which adjacent vertex to process next. Deciding which vertex of the polyhedron will be the next vertex to explore is called 'pivoting'.

**Definition 9** (Pivoting rule)**.** *A pivoting rule is a rule that is applied at each iteration of the simplex algorithm and that determines the next vertex to explore. For any starting vertex (i.e., any starting point of the simplex), a pivoting rule induces a path through the vertices of the polyhedron (the path follows the edges).*

Pivoting rules are designed in such a way that they define, for any starting vertex, a path that (hopefully) ends at the optimal vertex of the problem. Pivoting rules may cycle, but such a behavior is undesirable. There exist many different pivoting rules and some are better than others (in terms of the length of the induced paths). In the worst (pathological) case, a pivoting rule may visit all vertices of the polyhedron before finding the optimal vertex (Klee and Minty, 1972).

It must be emphasized that the definition of pivoting rules used in this work slightly differs from other definitions found in the literature. Traditionally, pivoting rules are indeed defined as rules that choose adjacent vertices (adjacent basic feasible solutions) with better objective values (see, e.g., De Loera, 2011). We shall refer to these rules as greedy pivoting rules. In this work, we do not require the next vertex to have a better objective value. We refer the reader to Section 6.6.2 for more details on this matter.

#### Theoretical challenges regarding the simplex

The search for a pivoting rule whose worst case complexity is polynomial (in terms of $n$ and $m$) is an important theoretical aspect of the study of the simplex algorithm. Assuming that the proposed modified Hirsch conjecture is true, such a rule exists. On the contrary, if the proposed modified Hirsch conjecture is false, it makes no sense to search for a polynomial

pivoting rule. This justifies Assumption 1 (i.e., the diameter is polynomial in the dimensions of the problem).

This work is devoted to the description of a methodology that endeavors to construct, in polynomial complexity, a pivoting rule that is guaranteed to yield paths of polynomial length when applied to restricted classes of problems. In particular, this work will use the theory of reinforcement learning to prove, or at least give hints, that such rules exist and can be found. Note that, for the simplex algorithm to be polynomial in the worst case, it is not sufficient that the length of the path is polynomial, each iteration of the algorithm must be polynomial as well. This implies that the pivoting rule must

1. yield paths of polynomial length between any starting vertex and the optimal vertex;
2. be applicable in polynomial time (i.e., the time needed to take a pivoting decision must be polynomial).

## 6.3 Complexity of the simplex algorithm

The performance of the simplex method is measured in terms of the number of vertices that are explored before the algorithm finds the (assumed) unique optimal solution. This number of vertices corresponds to the length of the path followed by the algorithm through the vertices of the polyhedron, i.e., the number of iterations of the simplex. The traveled path depends on the chosen implementation of the simplex and, more specifically, on the pivoting rule that is used within the algorithm as well as on the starting vertex. In this section, we discuss the performance, or time complexity, of the simplex in several situations. Note that this discussion is not intended to be a thorough review of the literature in the field, nor is it meant to go into the details of computational complexity theory.

Before discussing the performance of the simplex algorithm, let us briefly remind that there exist three types of performance measures: the best-, average-, and worst-case performance, respectively. The best-case performance represents a lower bound on the performance of any execution of the algorithm. The best-case performance is achieved when all components of the solution process are such that it is not possible to solve the problem more efficiently with the considered algorithm. For instance, in the case of the simplex, if a procedure (assumed to be an oracle whose time complexity is a constant) initializes the algorithm with the optimal vertex for each possible problem, then the simplex does not even need to perform any pivot and the problem is solved directly, hence, in constant time. The average-case performance represents the expected performance of the algorithm when it solves a problem drawn from a given distribution of problems. The average-case performance is intended to represent the behavior of the algorithm in practice when the problem to solve is not known beforehand. The average-case performance is typically much harder to compute than the best- and worst-case performance because it involves computing expectations over problem distributions. Finally, the worst-case performance is used to model the worst scenario that an algorithm can face. It is of primary interest in critical applications when an algorithm is required to produce a solution in a given amount of time.

Usually, the best-case performance is not the most informative complexity measure because it represents situations that are unlikely to happen in practice. The average- and worst-case

performance are generally preferred to study an algorithm as they model the average (likely to be observed in practice) and worst situations that the examined algorithm can face. When upgrades are made to an algorithm, they generally focus on improving the average- and worst-case behaviors. We therefore restrict, in the following, our discussion to these two performance measures.

Let $P(n, m)$ denote a family of optimization problems of size $(n, m)$ represented in the form of Equation (6.1). Let $S$ denote a particular implementation of the simplex method including the pivoting rule and let $I$ be an initialization rule that chooses the starting vertex of the algorithm. For a specific tuple $(S, I, P(\cdot, \cdot))$, the average-case and worst-case performance of the simplex, which are expressed in terms of the number of iterations of the algorithm, are defined as follows.

**Definition 10** (Average-case performance of the simplex)**.** *For a given tuple $(S, I, P(\cdot, \cdot))$, the average-case performance $K_{avg}(S, I, P(n, m))$ of the simplex is defined as the expectation of the number $K(S, I, p(n, m))$ of iterations required in order to solve any problem $p(n, m) \in P(n, m)$, distributed according to a given distribution $\mathcal{D}_{P(n,m)}$. In mathematical terms, the average-case performance is given by*

$$K_{avg}(S, I, P(n, m)) = \mathop{\mathrm{E}}_{p(n,m)\sim\mathcal{D}_{P(n,m)}} \Big[ K(S, I, p(n, m)) \Big], \ \forall (n, m).$$

*The average-case performance is often written in asymptotic notation, i.e.,*

$$K_{avg}(S, I, P(n, m)) = \mathcal{O}\Big( \mathcal{F}_{avg}(S, I, P(n, m)) \Big),$$

*where $\mathcal{O}$ designates the big-O notation and $\mathcal{F}_{avg}(S, I, P(n, m))$ is a function that characterizes the average-case performance for increasing problem sizes.*

**Definition 11** (Worst-case performance of the simplex)**.** *For a given tuple $(S, I, P(\cdot, \cdot))$, the worst-case performance $K_{worst}(S, I, P(n, m))$ of the simplex is defined as the maximum of the number $K(S, I, p(n, m))$ of iterations required in order to solve any problem $p(n, m) \in P(n, m)$. In mathematical terms, the worst-case performance is given by*

$$K_{worst}(S, I, P(n, m)) = \max_{p(n,m)\in P(n,m)} \Big[ K(S, I, p(n, m)) \Big], \ \forall (n, m).$$

*The worst-case performance is often written in asymptotic notation, i.e.,*

$$K_{worst}(S, I, P(n, m)) = \mathcal{O}\Big( \mathcal{F}_{worst}(S, I, P(n, m)) \Big),$$

*where $\mathcal{O}$ designates the big-O notation and $\mathcal{F}_{worst}(S, I, P(n, m))$ is a function that characterizes the worst-case performance for increasing problem sizes.*

The average-case performance of the simplex is known to be polynomial in terms of the dimensions $n$ and $m$ of the problems, i.e.,

$$K_{\text{avg}} = \mathcal{O}\left(\lambda(n, m)\right),$$

where $\lambda(\cdot, \cdot)$ denotes a polynomial. Such a result has been proved for different implementations of the simplex algorithm and for different assumptions on the distributions of the

problems (see, e.g., Borgwardt, 1982; Smale, 1983; Adler and Megiddo, 1985; Borgwardt, 1988). A formal proof of average polynomial complexity may not exist for every possible implementation of the simplex method, but it is widely acknowledged that the complexity of the simplex is indeed polynomial on average under rather general assumptions (Vanderbei, 2014; Nemhauser and Wolsey, 1988).

Unfortunately, despite the fact that the simplex works quite well in practice (i.e., polynomial time on average), the worst-case performance remains exponential (see, e.g., Klee and Minty, 1972). This can be easily understood intuitively. Indeed, given Definition 4, we know that the number of vertices of a polyhedron can be as large as an exponential function of the dimensions ($n$ and $m$) of the problem. If a pivoting rule is initialized with a very bad (with respect to the pivoting rule) starting vertex, then it is possible that the algorithm visits all possible vertices before finding the optimal solution. The corresponding complexity is thus exponential. Stated in mathematical terms, the worst-case performance for the simplex algorithm is given by

$$K_{\text{worst}} = \mathcal{O}\left(\exp(n, m)\right),$$

where $\exp(n, m)$ denotes an exponential function of the parameters.

In the case of the simplex, the complexity is mainly controlled by the pivoting rule used within the algorithm and by the starting vertex. There exist pivoting rules that are guaranteed to run in polynomial time for certain problem classes, even in the worst case (see, e.g., Ye, 2011). However, there exists at least one counter example (polytope and starting vertex) for most pivoting rules for which the algorithm requires an exponential number of iterations (Amenta and Ziegler, 1999; De Loera, 2011). It turns out that a pivoting rule, i.e., an implementation of the simplex, whose worst-case performance is polynomial for any possible LP problem class has not been found yet.

## 6.4 The simplex algorithm and sequential decision making problems

We claim that the simplex algorithm can be seen as a sequential decision making problem. We show in the following how. We first define more formally the so-called Markov decision processes (a framework to study sequential decision making problems), as well as the reinforcement learning paradigm. Then, we detail why a polyhedron can be regarded as a state space and make explicit the relationships between the simplex algorithm and the reinforcement learning task.

### 6.4.1 Markov decision processes and reinforcement learning

Reinforcement learning is a field of machine learning that can be defined as the problem of controlling a Markov decision process so that some performance criterion is maximized (see, e.g., Sutton and Barto, 1998). In the following, we give a more detailed description and definition of these elements.

**Definition 12.** *A finite Markov decision process (MDP) is a 4-tuple* $(\mathcal{S}, \mathcal{A}, p, r)$*, where*

- $\mathcal{S} = \{1, \ldots, n_{\mathcal{S}}\}$ *is a finite set of states;*
- $\mathcal{A} = \{1, \ldots, n_{\mathcal{A}}\}$ *is a finite set of actions;*
- $p(a, s, s') = \mathrm{P}[s_{t+1} = s' \,|s_t = s, a_t = a]$ *is a probability distribution representing the probability that the state at time* $t + 1$ *is* $s'$*, when the state at time* $t$ *and the action chosen at time* $t$ *are* $s$ *and* $a$*, respectively;*
- $r(s, s')$ *is a reward function*[1] *representing the reward received when state* $s'$ *is reached from state* $s$*.*

An MDP is an iterative process that features a player, or agent, that takes actions at successive time steps. Given the initial state $s_0$, and the sequence of actions $a_0$, $a_1$, $a_2$, ... chosen by the player, the states of the MDP follow the sequence $s_0$, $s_1$, $s_2$, ..., where each state $s_{t+1}$ is a realization of a random variable distributed according to the probability distribution

$$p(a_t, s_t, \cdot).$$

The player chooses the sequence of actions according to a so-called policy $\pi : \mathcal{S} \mapsto \mathcal{A}$, which maps a state to an action. The goal of the player is to choose the sequence of actions so as to maximize the sequence of rewards $r(s_0, s_1)$, $r(s_1, s_2)$, $r(s_2, s_3)$, ... that the player receives. The sequence of received rewards depends on the sequence of states, which itself depends on the sequence of actions and thus on the policy played by the agent. In general, a policy $\pi$ is given a score by summing the (discounted) rewards received by the player at each time step. In mathematical terms, this amounts to computing the sum

$$\sum_{t=0}^{T} \gamma^t r(s_t, s_{t+1}),$$

where $0 < \gamma \leq 1$ is a so-called discount factor and $T$ represents the time horizon. With this definition at hand, we can compute the score of the policy $\pi$ that produced the sequence of states $s_t$ for any initial state $s_0$. This score, known as the value function, is given by

$$J^{\pi}(i) = \lim_{T \to \infty} \mathrm{E}\left[\sum_{t=0}^{T} \gamma^t r(s_t, s_{t+1}) \,|s_0 = i\right]$$

in the infinite horizon case, where $E[\cdot]$ represents the expectation over the sequence of visited states $s_0$, $s_1$, $s_2$, ..., $s_T$.

**Definition 13** (Reinforcement learning)**.** *The reinforcement learning (RL) task is defined as the problem of finding a policy* $\pi^*$ *according to which the sequence of chosen actions yields a sequence of rewards that is maximum. In other words, reinforcement learning tries to find the policy* $\pi^*$ *that maximizes the value function* $J^{\pi^*}(i)$ *for every state* $i$*.*

Note that the definition hereabove is given so as to keep the discussion relatively simple. More formally, the term 'reinforcement learning' is usually limited to the problems for which the dynamics (transition probabilities and rewards) are unknown. For those cases where the model is known, the problem of finding an optimal policy is often referred to as dynamic

[1] Note that, in general, the reward function also depends on the chosen action. We omit this dependency here since it is not required in our analysis and $r(s, s')$ is essentially a special case of the more general $r(s, a, s')$.

programming (DP) (Busoniu et al., 2010). Instead of juggling with both names and in order to simplify the discussion, we restrict our wording to reinforcement learning because RL is more general than dynamic programming (RL techniques can indeed be applied to problems for which the dynamics are known, whereas DP techniques cannot be used when a model is not available).

### 6.4.2 Simplex as a sequential decision making problem

Since the simplex algorithm mainly handles vertices, and because each vertex is unique, the vertices can be seen as states of a state space. The current vertex processed by the simplex can thus be seen as the current state of an MDP. Additionally, the simplex algorithm merely jumps from one vertex to the next and the possible transitions are limited by the neighborhood of the current vertex in the considered polyhedron. These possible transitions can be reflected in the MDP through the transition probability matrix that defines the probability to go from one state to another when a given action is taken. Figure 6.1 illustrates a mapping from a polyhedron to a graph with matching nodes and edges.

In the following, we cast the simplex algorithm as a probabilistic MDP. Moving from a deterministic setting to a probabilistic one may seem odd, but this conversion is motivated by the fact that RL techniques are traditionally used with probabilistic settings. Applying RL techniques to deterministic problems is possible, but most of the machinery of RL algorithms are designed for the probabilistic case. Additionally, probabilistic settings are more general than deterministic ones and, in particular, encompass the deterministic case. Since there is no loss of generality when formulating the problem in a probabilistic manner and because RL mainly deals with probabilistic problems, formulating the simplex as a probabilistic MDP is a reasonable choice.

### 6.4.3 MDP formulation of the simplex

Let us consider a state space $\mathcal{S} = \{1, 2, \ldots, \mathcal{V}\}$ composed of a finite number $\mathcal{V}$ of states. The state space $\mathcal{S}$ corresponds to the set of vertices of the polyhedron of the studied LP problem. Let us also consider a set of actions $\mathcal{A} = \{(i, j)\}, \forall i, j \in \mathcal{S}$ that model the actions allowing the MDP to go from one state (vertex) to the next.

#### Simplex as a finite horizon MDP

An MDP formulation generally involves a probability distribution that governs the transitions between the states according to the chosen actions. Since, in the simplex, not all transitions are allowed, the probability distribution must reflect these constraints in the MDP. More specifically, there are two constraints that govern the path of the simplex through the algorithm: (i) it is not possible to reach a non-neighboring vertex with one step, and (ii) when the current state is $i$, following an edge $(i, j)$ with tail $i$ inevitably leads to node $j$. Stated in MDP terms, this amounts to imposing the following constraints on the transitions of the MDP: when the current state (vertex) is $i$,

1. an action $a = (i, j)$ leads deterministically to state $j$ provided that $j$ is a neighbor of $i$;

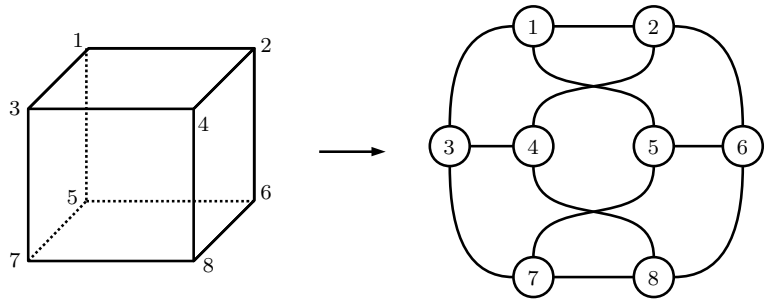



Figure 6.1: A polyhedron can be considered as a graph whose nodes are the vertices of the polyhedron and whose edges correspond to the edges of the polyhedron. In the graph, the circles represent the states of the problem and the edges represent the allowed transitions from one state to the next.

2. if an action $a = (i, j)$ is chosen and $j$ is not a neighbor, the MDP cannot move to $j$ and must remain in state $i$;
3. an action $a = (i', j')$ with tail $i' \neq i$ cannot be chosen and the MDP must also remain in state $i$.

If we denote by $\mathcal{N}(i)$ the set of neighbors of the vertex $i$ in the polyhedron, whose cardinality $|\mathcal{N}(i)|$ is bounded above by $m - n$ (i.e., the number of neighbors of state $i$ is bounded by $m - n$), the probability distribution $p\,(a, i, j)$ that models the transitions in the state space (and, hence, in the polyhedron) is given by

$$\forall a \in \mathcal{A}, i, j \in \mathcal{S},\ p\left(a = (i', j'), i, j\right) = \begin{cases} 1 & \text{if } i' = i \wedge j' \in \mathcal{N}(i) \wedge j = j', \\ 1 & \text{if } i' = i \wedge j' \notin \mathcal{N}(i) \wedge j = i, \\ 1 & \text{if } i' \neq i \wedge j = i, \\ 0 & \text{otherwise.} \end{cases} \tag{6.2}$$

In addition to the previous elements, a reward function needs to be added to the MDP formulation. If the reward function is chosen appropriately, the solution (i.e., the optimal policy) of the corresponding reinforcement learning problem will yield a shortest path from any state (vertex) to the optimal state (i.e., the optimal vertex). Indeed, the optimal policy, according to Definition 13, maximizes the value function for each state. If the chosen reward is such that the value function for any initial state is a (strictly) decreasing function of the length of the path between the initial and final states, maximizing the value function becomes equivalent to minimizing the length of the path. In particular, the optimal (maximum) value function is achieved for minimum path lengths. One possible reward function for which the optimal policy yields such shortest paths from any state to the optimal one (when $\gamma < 1$) is $\mathcal{R}_1$, which is defined as

$$\mathcal{R}_1(i, j) = \begin{cases} 1 \text{ if } j \text{ is the optimal vertex,} \\ 0 \text{ otherwise.} \end{cases}$$

Note that it is crucial that the discount factor $\gamma$ is strictly less than 1 for $\mathcal{R}_1$ to yield optimal paths. It is possible to devise other reward functions that similarly yield shortest paths. The proposed reward $\mathcal{R}_1$ is a simple example of such reward functions.

Finally, for the MDP formulation to be complete, a policy according to which the decisions are taken needs to be defined. Once the optimal value function $J^{\pi^*}$ is known, the optimal policy $\pi^*$ is obtained directly with

$$\pi^*(i) = \underset{a \in \mathcal{A}}{\operatorname{argmax}} \left\{ \underset{j \sim p(a,i,\cdot)}{\mathrm{E}} \left[ r(i,j) + \gamma J^{\pi^*}(j) \right] \right\}, \tag{6.3}$$

where $p(a, i, \cdot)$ is the probability distribution that models the allowed transitions in the MDP (see Definition 12) and $\mathrm{E}[\cdot]$ is the expectation operator. In our case, this expression can be simplified to

$$\pi^*_{\mathcal{R}_1}(i) = \left( i, \underset{j \in \mathcal{N}(i)}{\operatorname{argmax}} J^*_{\mathcal{R}_1}(j) \right) \in \mathcal{A},$$

where $J^*_{\mathcal{R}_1}$ denotes the optimal value function for reward $\mathcal{R}_1$. This definition implies that, at a given vertex $i$ of the polytope, the next vertex to process corresponds to the neighboring vertex that maximizes the optimal value function. Since the optimal value function for a node is equal to the discount factor ($\gamma$) to the power of the length of the shortest path between that node and the optimal node, following the maximum value function yields shortest paths through the vertices of the polytope. Note that, if $J^*_{\mathcal{R}_1}(j)$ can be evaluated in polynomial time for any state $j$, the optimal policy $\pi^*_{\mathcal{R}_1}(\cdot)$ is polynomial as well since the argmax only considers $|\mathcal{N}(i)| \leq m - n$ elements.

**From finite to infinite horizon**

The MDP formulation of the simplex given above holds in the finite horizon setting. Since applying the theory of reinforcement learning is usually easier in the infinite horizon case, we transform the previous MDP formulation of the simplex into an infinite horizon formulation, which, fortunately, requires only a minor modification. Indeed, it suffices to add a new transition from the optimal vertex to each other vertex in the polytope in such a way that the vertex following the optimal one is randomly chosen with uniform distribution. When the current state of the MDP is the optimal vertex, the transition probability becomes

$$\forall a \in \mathcal{A}, j \in \mathcal{S},\ p\left(a = (i', j'), i^*, j\right) = \begin{cases} \frac{1}{\mathcal{V}} & \text{if } i' = i^*, \\ 1 & \text{if } i' \neq i^* \wedge j = i^*, \\ 0 & \text{otherwise,} \end{cases}$$

where $i^*$ is the optimal vertex and $\mathcal{V}$ represents the number of vertices in the polytope.

The modification is really minor but it has an important implication. Indeed, adding a transition after the optimal vertex changes the optimal value function for all vertices. The optimal value function quits being equal to the discount factor to the power of the shortest path between the current node and the optimal one. However, this is not a major concern. Indeed, the optimal value function in the infinite case actually just corresponds to the sum of the optimal value in the finite horizon setting and of a 'restart term' due to the added transition after the optimal vertex. The optimal value function for a node $j$ in the infinite

horizon case is thus given by

$$\begin{aligned}
J_\infty^{\pi^*}(j) &= \mathrm{E}\left[\sum_{t=0}^{\infty}\gamma^t r(s_t, s_{t+1})\,|s_0 = j\right] \\
&= \mathrm{E}\left[\sum_{t=0}^{T_j}\gamma^t r(s_t, s_{t+1})\,|s_0 = j\right] + \mathrm{E}\left[\sum_{t=T_j+1}^{\infty}\gamma^t r(s_t, s_{t+1})\,\big|s_{T_j+1} = i^*\right] \\
&= \gamma^{T_j} r(s_{T_j}, s_{T_j+1}) + \gamma^{T_j+1}\underbrace{\mathrm{E}\left[\sum_{t=T_j+1}^{\infty}\gamma^{t-T_j-1} r(s_t, s_{t+1})\,\big|s_{T_j+1} = i^*\right]}_{\alpha} \\
&= \gamma^{T_j}\big(r(s_{T_j}, s_{T_j+1}) + \gamma\alpha\big)
\end{aligned}$$

where $T_j$ represents the length of the shortest path between vertex $j$ and the optimal vertex $i^*$ and $\alpha$ represents the restart term. Note that it is possible to write $J_\infty^{\pi^*}(j)$ in that way because all rewards are 0 except for the reward that leads to the optimal vertex and because the restart transition renders the sequence of states $s_{T_j+1}, s_{T_j+2}, \ldots$ independent from the sequence $s_1, s_2, \ldots, s_{T_j}$. In this special case, the reward leading to the optimal vertex is equal to 1 (see definition of $\mathcal{R}_1$), which yields an optimal value function equal to

$$J_\infty^{\pi^*}(j) = \gamma^{T_j}(1 + \gamma\alpha). \tag{6.4}$$

As mentioned, the restart term has almost no impact: a greedy policy (i.e., a policy of the form of Equation (6.3)) starting from an arbitrary vertex still follows the shortest path until the optimal vertex is reached.

Starting from its definition, it is possible to give a simple expression for $\alpha$.

$$\begin{aligned}
\alpha &= \mathrm{E}\left[\sum_{t=T_j+1}^{\infty}\gamma^{t-T_j-1} r(s_t, s_{t+1})\,\big|s_{T_j+1} = i^*\right] \\
&= \sum_{i\in\mathcal{S}}\frac{1}{\mathcal{V}}\left\{r(i^*, i) + \mathrm{E}\left[\sum_{t=T_j+2}^{\infty}\gamma^{t-T_j-1} r(s_t, s_{t+1})\,\big|s_{T_j+2} = i\right]\right\} \\
&= \frac{1}{\mathcal{V}}\sum_{i\in\mathcal{S}}\left\{r(i^*, i) + \gamma\,\mathrm{E}\left[\sum_{t=T_j+2}^{\infty}\gamma^{t-T_j-2} r(s_t, s_{t+1})\,\big|s_{T_j+2} = i\right]\right\} \\
&= \frac{1}{\mathcal{V}}\sum_{i\in\mathcal{S}}\gamma\,\mathrm{E}\left[\sum_{t'=0}^{\infty}\gamma^{t'} r(s_{t'}, s_{t'+1})\,|s_0 = i\right] \\
&= \frac{1}{\mathcal{V}}\sum_{i\in\mathcal{S}}\gamma^{T_i+1}(1 + \gamma\alpha),
\end{aligned}$$

where we use Equation (6.4), the fact that $r(i^*, i) = 0$, and the transition probabilities defined for the simplex. Further simplifying this expression yields

$$\begin{aligned}
\alpha &= \frac{1}{\mathcal{V}} \sum_{i \in \mathcal{S}} \gamma^{T_i+1}(1 + \gamma\alpha) \\
\alpha &= \frac{1}{\mathcal{V}} \sum_{i \in \mathcal{S}} \gamma^{T_i+1} + \frac{1}{\mathcal{V}} \sum_{i \in \mathcal{S}} \gamma^{T_i+2}\alpha \\
\alpha - \frac{1}{\mathcal{V}} \sum_{i \in \mathcal{S}} \gamma^{T_i+2}\alpha &= \frac{1}{\mathcal{V}} \sum_{i \in \mathcal{S}} \gamma^{T_i+1} \\
\alpha &= \frac{\frac{1}{\mathcal{V}} \sum_{i \in \mathcal{S}} \gamma^{T_i+1}}{1 - \frac{1}{\mathcal{V}} \sum_{i \in \mathcal{S}} \gamma^{T_i+2}}. \qquad (6.5)
\end{aligned}$$

The optimal value function possesses several interesting properties. The following lemma illustrates one of them.

**Lemma 1** (Optimal value function of neighboring vertices)**.** *Let $i$ be an arbitrary vertex of the polytope and let $\mathcal{N}(i)$ denote the set of neighbors of $i$. The optimal value function $J^*$ obeys the following equation*

$$\forall j \in \mathcal{N}(i),\ J^*(j) = \begin{cases} \gamma J^*(i), & or \\ J^*(i), & or \\ \frac{J^*(i)}{\gamma}. \end{cases}$$

*Proof.* Equation (6.4) indicates that the optimal value function is equal to the product of the factors $\gamma^{T_i}$ and $(1+\gamma\alpha)$, where $T_i$ represents the length of the shortest path between vertex $i$ and the optimal vertex. For a vertex $j \in \mathcal{N}(i)$, i.e., a neighbor of $i$, the length of the shortest path to the optimal vertex is either the same, one less, or one more than the distance from $i$ to the same optimal vertex. Writing the optimal value function as Equation (6.4) for $T_i$, $T_i - 1$, and $T_i + 1$ proves the lemma. □

### Discussion

Given the previous MDP formulation, it appears clearly that the simplex can be regarded as an agent that seeks a path in the state space from an arbitrary starting state to the optimal state (corresponding to the optimal vertex of the LP problem). In this formulation, the simplex pivoting rules are represented by the policies of the MDP. Making the simplex as efficient as possible implies that the number of iterations is as small as possible and, hence, that the path travelled by the simplex is as short as possible between the initial and final states (which, by the way, forbids the loops in the path). However, this is rarely achieved, and even more rarely guaranteed, with traditional pivoting rules defined within the optimization framework. On the other hand, the RL framework, together with the MDP formulation of the simplex, provides the theoretical tools to study such pivoting rules. Indeed, applying the ad hoc algorithm, and under certain assumptions, RL is guaranteed to find the optimal policies, which correspond to the optimal (shortest) pivoting rules. Since the shortest paths are polynomial due to our working assumption (the modified Hirsch conjecture is true), polynomial pivoting rules can be found for any polytope.

Actually, the MDP formulation of the simplex replaces the search for a polynomial pivoting rule by the search for an optimal policy that maximizes an appropriately chosen reward. The assumption that the diameter of the polytope is polynomial (i.e., the modified Hirsch conjecture is true) ensures that the shortest path induced by the optimal policy is actually polynomial in the dimensions $m$ and $n$ of the LP problem.

It is important to emphasize here that the length of the path traveled by the simplex is only one component of the complexity of the proposed approach. Indeed, in the MDP framework, several other elements have to be taken into account in order to claim that the entire approach is polynomial. More specifically, the time complexity of the entire approach is composed of

1. the time required to find the optimal policy with RL algorithms. This component has to be taken into account only once, since the optimal policy remains the same for the entire optimization once it is found;

2. the time required to apply the chosen policy, which has to be multiplied by the number of times the policy is applied (roughly equal to number of iterations of the simplex up to a multiplicative factor). The time required to apply a policy is itself composed of several components. More details about the complexity of applying a policy are given later in the chapter.

For the MDP-powered simplex to be polynomial, the complexity of every component of the approach must be polynomial, in addition to yielding polynomial paths through the polytope. Moreover, the space required by the different components of the approach need to be polynomial as well in the dimension of the optimization problem. These aspects are discussed in greater detail in the remainder of the chapter.

For the sake of readability, we may in the following use interchangeably the terms pivoting rule, policy, and value function in an abuse of terminology.

## 6.5 Using reinforcement learning theory to find polynomial pivoting rules

Casting the simplex algorithm as a Markov decision process allows us to use the theory of reinforcement learning to shed light onto some theoretical aspects of the simplex method from a radically new perspective. We now discuss how reinforcement learning methods can be applied to the simplex problem.

In this section, we start by briefly describing how reinforcement learning algorithms work and then give some theoretical results that tie the efficiency of RL methods to the complexity of the simplex algorithm. We then consider two specific RL algorithms and analyze how their outcomes can be used to derive theoretical results for the simplex.

### 6.5.1 Brief description of reinforcement learning algorithms

Before going any further, it is worth mentioning here that traditional RL algorithms, such as value iteration or policy iteration, are, as their names indicate, iterative. They typically start with an arbitrary value function (in the case of value iteration) or an arbitrary policy (in the case of policy iteration) and try to improve the current approximation as the algorithm progresses.

In the following, we give brief descriptions of the value iteration, the policy iteration, and the approximate value iteration methods. Note that we voluntarily omit the details regarding the internal mechanisms of the RL algorithms. We refer the reader to, for instance, Bertsekas and Tsitsiklis (1996) or Sutton and Barto (1998) for a complete description of these algorithms.

#### Value iteration

Value iteration (VI) is a RL technique whose goal is to find the optimal value function of a MDP. More specifically, the method starts with an arbitrary approximation $\tilde{J}_0$ of the optimal value function $J^*$, e.g.,

$$\tilde{J}_0(s) = 0, \, \forall s \in \mathcal{S},$$

and provides, after $L$ iterations, a new approximation $\tilde{J}_L$ that (hopefully) approximates $J^*$ closely enough, i.e.,

$$\tilde{J}_L(s) \approx J^*(s), \, \forall s \in \mathcal{S}.$$

At each iteration, VI applies a specific operator $\mathcal{T}$, known as the Bellman operator, that provides a new approximation of the optimal value function. The Bellman operator is defined as follows:

$$\tilde{J}_{k+1}(s) = \mathcal{T}\tilde{J}_k(s) = \max_{a \in \mathcal{A}} \left\{ \sum_{s' \in \mathcal{S}} p(a, s, s') \left( r(s, s') + \gamma \tilde{J}_k(s') \right) \right\}$$

which, in our case, can be simplified to

$$\tilde{J}_{k+1}(s) = \max_{s' \in \mathcal{N}(s)} \left\{ r(s, s') + \gamma \tilde{J}_k(s') \right\}, \tag{6.6}$$

since the probability distribution of the MDP formulation of the simplex (see Equation (6.2)) greatly simplifies the first equation. In the end, adding all steps of the algorithm together, VI sums up to only a few operations represented in Algorithm 5.

#### Policy iteration

For the sake of completeness, let us briefly highlight the main features of policy iteration (PI) that differs from VI in that PI tries to find the optimal policy instead of the optimal value function. The algorithm initializes its working policy $\tilde{\pi}$ to some initial arbitrary policy, say $\tilde{\pi}_0$, and then iterates to bring the working policy closer and closer to the optimal policy

**Algorithm 5** Value iteration (VI)

1: $\tilde{J}_0(s) = 0, \forall s \in \mathcal{S}$
2: **for** $k = 0, \ldots, L-1$ **do**
3: **for** $s \in \mathcal{S}$ **do**
4: $\tilde{J}_{k+1}(s) = \max_{s' \in \mathcal{N}(s)} \left\{ r(s, s') + \gamma \tilde{J}_k(s') \right\}$
5: **end for**
6: **end for**

$\pi^*$. When the algorithm terminates after $L$ iterations, the working policy is hopefully a fair representation of the optimal policy, i.e.,

$$\tilde{\pi}_L(s) \approx \pi^*(s), \forall s \in \mathcal{S}.$$

Unlike VI, we do not enter into the particulars of PI since the remainder of this work relies on VI techniques only.

### Approximate value iteration

Pure VI as presented above suffers two main drawbacks:

1. each iteration of the algorithm requires a number of (sub)iterations that is proportional to the size of the state space to update the current approximation of $J^*$ (because each component of $\tilde{J}$ is updated independently, see lines 3-5 in Algorithm 5);
2. storing $\tilde{J}$ requires $\mathcal{O}(\mathcal{V})$ space.

These two drawbacks rapidly become a main concern. Indeed, the requirements regarding both storage space and computation time may prevent the algorithm from being applied to interesting problems (interesting from a practical point of view). To alleviate both shortcomings of VI, approximate value iteration (AVI) methods can be used.

Approximate value iteration differs from the theoretical version of VI in that the value function is not stored as a vector (with one component equal to the value of the value function for a given state). Instead, in AVI, each iterate of the value function $\tilde{J}_k$ is approximated, typically by choosing $\tilde{J}_k$ from a given set of possible functions $\mathcal{F}$. In that case, it is possible that one iterate of the value function, which is projected onto $\mathcal{F}$, cannot be represented exactly, hence the qualifier 'approximate'. In mathematical terms, the value function computed by an AVI method after $k$ iterations is thus

$$\tilde{J}_k(s) = f(s) \in \mathcal{F},$$

where $\mathcal{F} = \{f(\cdot)\}$ is a set of functions that map states to scalars, i.e., $f : \mathcal{S} \mapsto \mathbb{R}$. For a choice of the function set $\mathcal{F}$, the task of value iteration, which aims at finding the optimal value function $J^*(s)$, translates, in terms of the function set $\mathcal{F}$, into finding $f^*$ such that

$$f^* = \underset{f \in \mathcal{F}}{\operatorname{argmin}} \left\| J^* - f(\cdot) \right\|_p,$$

where $\|\cdot\|_p$ represents a $p$-norm and $f(\cdot)$ a vector with one component per state $s \in \mathcal{S}$. In other words, AVI aims at finding the function $f^*$ that minimizes some distance (in terms of a $p$-norm) between the function set and the optimal value function.

Restricting $\tilde{J}_k$ to a given function set $\mathcal{F}$ alleviates the storage issue discussed earlier. Indeed, the function set can be chosen so that the required storage is limited. Choosing a restricted function set thus solves the space complexity issue of VI. The second issue, the update operation with the Bellman operator, remains open. We now discuss fitted value iteration (FVI) which is a RL algorithm that implements the AVI idea (see, e.g., Munos, 2007). In FVI, the Bellman operator is not used as is to find the next iterate. Instead, a supervised learning algorithm is used to provide the next function $f = \tilde{J}_{k+1}$ (hence the term fitted in FVI). More specifically, at each iteration, FVI

1. generates a set $\{s_i\}_{i=1}^H$ of $H$ states drawn for an arbitrary distribution $\mu$;
2. applies the Bellman operator (Equation (6.6)) to those states only to obtain a set of values $\{u_i\}_{i=1}^H$;
3. computes some chosen features $\{\boldsymbol{\phi}_i = \mathcal{C}(s_i)\}_{i=1}^H$ from the generated states;
4. applies the supervised learning algorithm to the training set $\{(\boldsymbol{\phi}_i, u_i)\}_{i=1}^H$ to obtain the next iterate $\tilde{J}_{k+1}$.

FVI is summarized in Algorithm 6.

FVI smartly gets round the storage and time issues of VI. Obviously, this comes at a price and FVI achieves both memory and computation time efficiency at the expense of accuracy (which is typically worse for FVI) and at the expense of the guaranteed convergence of VI.

### 6.5.2 Tying the efficiency of RL to the complexity of the simplex

The success of a RL algorithm can be measured based on how close the approximation is from the optimal value function or policy and based on how many iterations $L$ are needed to obtain this result. This aspect is important and must be taken into account in our analysis. In this section, we show how the efficiency of reinforcement learning algorithms can be tied to the complexity of the simplex. More specifically, we focus here on the value iteration algorithm.

As mentioned before, value iteration typically yields an approximation $\tilde{J}$ of the optimal value function. We saw earlier that the optimal value function $J^*$ indeed finds polynomial paths from any vertex of the polytope to the optimal one. We now prove that it is possible to obtain similar results if the approximated value function $\tilde{J}$ is used instead of $J^*$.

**Theorem 3** (Local shortest paths with approximated value functions)**.** *Let $\tilde{J}$ be an approximation of the real value function $J^*$ such that*

$$\left\| J^* - \tilde{J} \right\|_\infty \leq \varepsilon,$$

*and let $T_i$ be the length of the shortest path from an arbitrary vertex $i$ to the optimal vertex $o$. For any vertex $i$ such that*

$$T_i < 1 + \log_\gamma \frac{2\varepsilon}{(1+\gamma\alpha)(1-\gamma)},$$

**Algorithm 6** Fitted value iteration (FVI)

$\tilde{J}_0 = 0$
**for** $k = 0, \ldots, L-1$ **do**
  $\{s_i\}_{i=1}^{H} \sim \mu$ ▷ generate a set of states from distribution $\mu$
  $u_i = \max_{s' \in \mathcal{N}(s_i)} \left\{ r(s_i, s') + \gamma \tilde{J}_k(s') \right\}, i = 1, \ldots, H$ ▷ Bellman operator
  $\{\boldsymbol{\phi}_i = \mathcal{C}(s_i)\}_{i=1}^{H}$ ▷ compute the feature representation of the generated states
  $\tilde{J}_{k+1} = \text{train}\left( \{(\boldsymbol{\phi}_i, u_i)\}_{i=1}^{H} \right)$ ▷ compute features and apply learning algorithm
**end for**

*then a policy that is greedy with respect to the approximate value function $\tilde{J}$ yields shortest paths between the vertex $i$ and the optimal vertex $o$.*

*Proof.* Let us assume that the simplex algorithm starts at vertex $i$, whose optimal value function is given by $J^*(i)$. Using Lemma 1, it is clear that any vertex $k \in \mathcal{N}(i)$ that is a neighbor of $i$ is either

1. at the same distance as $i$ to the optimal node, in which case $J^*(k) = J^*(i)$;
2. one step further to the optimal node, in which case $J^*(k) = \gamma J^*(i)$;
3. one step closer to the optimal node, in which case $J^*(k) = \gamma^{-1} J^*(i)$.

We denote by $\mathcal{N}_0(i)$, $\mathcal{N}_{+1}(i)$, and $\mathcal{N}_{-1}(i)$ the sets of vertices that are at the same distance, one step further, and one step closer to the optimal node, respectively.

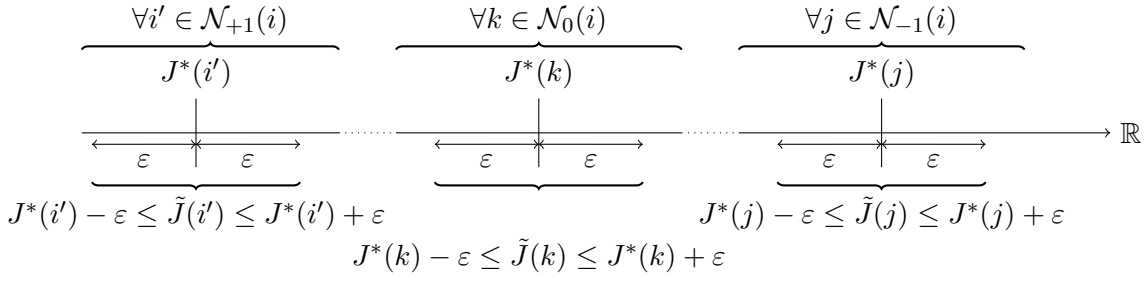
Considering that the simplex is at vertex $i$, the next state that lies on the shortest path and that is one step closer to the optimal vertex belongs to the set $\mathcal{N}_{-1}(i)$. The simplex thus follows the shortest path if the next state that is explored by the algorithm belongs to that set. In order for such a vertex to be chosen by the greedy policy, we need

$$\tilde{J}(i') < \tilde{J}(j'), \forall i' \in \mathcal{N}_0(i) \vee \mathcal{N}_{+1}(i), j' \in \mathcal{N}_{-1}(i),$$

which, provided that $\left\| J^* - \tilde{J} \right\|_\infty \leq \varepsilon$, becomes

$$J^*(k) + \varepsilon < J^*(j) - \varepsilon,$$

where $k \in \mathcal{N}_0(i)$ and $j \in \mathcal{N}_{-1}(i)$. The following schema illustrates graphically why this inequality can be written.



Using Equation 6.4, we can write

$$\begin{aligned} J^*(k) + \varepsilon &< J^*(j) - \varepsilon \\ \gamma^{T_i}(1+\gamma\alpha) + \varepsilon &< \gamma^{T_i-1}(1+\gamma\alpha) - \varepsilon \\ 2\varepsilon &< \gamma^{T_i-1}(1+\gamma\alpha) - \gamma^{T_i}(1+\gamma\alpha), \end{aligned}$$

which means that if $\varepsilon$ satisfies this inequality, then the next state indeed belongs to one optimal path towards the optimal vertex.

Going back to the hypothesis, we now have

$$\begin{aligned} T_i &< 1 + \log_\gamma \frac{2\varepsilon}{(1+\gamma\alpha)(1-\gamma)} \\ T_i - 1 &< \log_\gamma \frac{2\varepsilon}{(1+\gamma\alpha)(1-\gamma)} \\ \log_\gamma \left(\gamma^{T_i-1}\right) &< \log_\gamma \frac{2\varepsilon}{(1+\gamma\alpha)(1-\gamma)} \\ \frac{2\varepsilon}{(1+\gamma\alpha)(1-\gamma)} &< \gamma^{T_i-1} \\ 2\varepsilon &< \gamma^{T_i-1}(1+\gamma\alpha)(1-\gamma), \end{aligned}$$

which shows that, for a given error in the approximation of the value function, the approximation still leads optimal decisions as long as the vertices are not too far away from the optimal vertex. Indeed, this bound holds for all vertices whose distance to the optimal vertex is less than $T_i$ and, in particular, the bound holds for all vertices that lie on the shortest path between $i$ and $o$.

This last inequality shows that, if the shortest path from a vertex $i$ and the optimal vertex is less than $1 + \log_\gamma \frac{2\varepsilon}{(1+\gamma\alpha)(1-\gamma)}$, then an approximate value function $\tilde{J}$, which is such that $\left\| J^* - \tilde{J} \right\|_\infty \leq \varepsilon$, yields an optimal path through the edges of the polytope from $i$ to $o$. □

Theorem 3 shows that, for a given approximation of the value function, it is still possible to recover the shortest paths for those vertices that are not too far away from the optimal vertex. Theorem 3 also indicates that it is not necessary to approximate with infinite accuracy the real value function in order for the procedure to yield shortest (polynomial) paths through the polytope. This result is extended to the entire polytope by the following corollary.

**Corollary 1** (Global shortest paths with approximated value functions)**.** *Let $\tilde{J}$ be an approximation of the optimal value function $J^*$ such that*

$$\left\| J^* - \tilde{J} \right\|_\infty \leq \varepsilon,$$

*and let $T_P$ be the diameter of the polytope $P$, i.e., the maximum length of all shortest paths from arbitrary vertices to the optimal vertex. If the approximation error $\varepsilon$ is such that*

$$\varepsilon < \frac{1}{2}\gamma^{T_P-1}(1+\gamma\alpha)(1-\gamma),$$

*then the approximate value function $\tilde{J}$ yields shortest paths between any starting vertex $i \in \mathcal{S}$ and the optimal vertex $o$ in the polytope $P$.*

*Proof.* The proof is straightforward. Indeed, applying Theorem 3 leads to the conclusion that the current approximation of the optimal value function yields shortest paths through the polytope for all vertices that are not further away than $T_P$ from the optimal vertex. Since, by definition, $T_P$ is the largest distance between any two vertices in the polytope, we conclude that $\tilde{J}$ yields shortest paths for any starting vertex in the polytope. □

Corollary 1 proves that an approximate value function can still yield shortest paths through the entire polytope provided that the approximation is close enough to the real value function.

### 6.5.3 Value iteration to compute the optimal value function

Let us first consider the naive value iteration algorithm to find the optimal policy and let us show how the theory of RL can provide interesting insights about the optimization problem. We use here theoretical results presented in Bartlett (2003) with the necessary adaptations of notation.

Consider that the value function is represented as a vector, with one component per state of the MDP. Such a representation allows us to represent exactly all possible value functions that can be derived for the current MDP. With such a representation of the value function, it can be shown (see, e.g., Bartlett, 2003, Theorem 2) that value iteration converges to the optimal value function after a finite number of iterations. In particular, after $L$ iterations, value iteration yields an approximation $\tilde{J}_L$ of the value function that satisfies

$$\left\| J^* - \tilde{J}_L \right\|_\infty \leq \frac{2\gamma^{L+1}}{(1-\gamma)^2} \|r\|_\infty .$$

This result can be used to determine the number of iterations required by value iteration to produce an approximation of the value function that is close enough to the optimal value function so as to yield shortest paths through the polytope.

**Theorem 4** (Sufficient convergence of value iteration to the optimal value function)**.** *After $L$ iterations, where*

$$L > T_P - 2 + \log_\gamma \frac{1}{4} + \log_\gamma (1+\gamma\alpha)(1-\gamma)^3, \tag{6.7}$$

*value iteration produces an approximation of the value function that is close enough to the optimal value function and, hence, yields shortest paths through polytope $P$.*

*Proof.* Combining the convergence results of value iteration (Equation (6.7)) with Corollary 1 yields

$$\begin{aligned}
\left\| J^* - \tilde{J}_L \right\|_\infty \leq \frac{2\gamma^{L+1}}{(1-\gamma)^2} \|r\|_\infty &< \frac{1}{2}\gamma^{T_P-1}(1+\gamma\alpha)(1-\gamma), \\
\gamma^{L+1} \|r\|_\infty &< \frac{1}{4}\gamma^{T_P-1}(1+\gamma\alpha)(1-\gamma)^3, \\
L+1 &> T_P - 1 + \log_\gamma \frac{1}{4} + \log_\gamma (1+\gamma\alpha)(1-\gamma)^3,
\end{aligned}$$

where $\|r\|_\infty = 1$ by definition of the reward function $\mathcal{R}_1$. □

Theorem 4 shows that the convergence of value iteration is linear with the diameter of the polytope and, hence, linear with the number of variables and constraints. This guarantees a polynomial convergence of the algorithm towards the optimal value function and, hence, towards the optimal policy (pivoting rule).

However, despite the polynomial convergence of value iteration, this approach does not answer the question of the polynomial complexity of the simplex. Indeed, as discussed in Section 6.5.1, VI suffers two main drawbacks regarding both the required storage space and the training time. These two drawbacks are major concerns in this situation:

1. since each iteration of the training procedure requires iterating over all states of the MDP and since the number of states may be exponential, the training procedure is potentially exponential as well;
2. storing the approximate value function $\tilde{J}$ requires $\mathcal{O}(\mathcal{V})$ space, where $\mathcal{V}$ is potentially exponential.

These two drawbacks render the proposed solution method not satisfactory when the approach is analyzed as a whole. Indeed, the learning part of the approach, which is dedicated to finding the optimal value function, is not polynomial in the dimensions of the optimization problem. Additionally, the space required to apply this solution (due to the storage of the approximate value function) is proportional to the number of vertices in the polyhedron, which is potentially exponential.

### 6.5.4 Fitted value iteration to compute the optimal value function

To get round the limitations of the 'theoretical' approach detailed in Section 6.5.3, we must turn to other techniques than VI. We now consider the so-called fitted value iteration (FVI) method that differs from VI in that the value function is not stored as a vector (with one component equal to the value of the value function for a given state). Instead, in FVI, the possible value functions are restricted to a fixed set of candidate functions. Additionally, the computational complexity of the update operations can be controlled by an appropriate choice of the algorithm parameters. We now discuss the different aspects of FVI applied to the simplex problem, show how the space and time can be controlled, and discuss how the FVI theoretical guarantees can be transposed to the problem of finding good pivoting rules for the simplex algorithm.

#### FVI space complexity

The total space required to execute FVI is composed of two parts: the space required to store the current iterate $\tilde{J}_k$ and the space required to store the data generated at each iteration, drawn from distribution $\mu$ (see line 3 in Algorithm 6).

In FVI, each iterate belongs to a predefined function set $\mathcal{F}$. Constraining the set of possible functions to a given function set has one important advantage: if the function set admits a compact representation, for instance by means of a parametric representation of the elements of the set, the space required to store one element of the set can be easily controlled. For

example, if $\mathcal{F}$ is constrained to the set of linear functions from the inputs to the output, $\mathcal{F}$ can be expressed as

$$\mathcal{F} = \left\{ f(x) = x^\top \theta \,|\theta \in \mathbb{R}^q \right\}.$$

In this case, storing one single element $f \in \mathcal{F}$ amounts to storing $q$ scalars. Hence, under the assumption that the elements of the function set can be represented in a compact way, the space requirements of FVI can be controlled effectively by controlling the richness of $\mathcal{F}$. Clearly, since the space of representable functions is restricted, FVI may not converge to the optimal value function while VI is guaranteed to. Note that the function set $\mathcal{F}$ is typically directly dependent on the learning algorithm that is used as part of FVI.

The second component of the space required to run FVI arises from the data that is generated and fed to the learning algorithm to create the next iterate $\tilde{J}_{k+1}$. The total amount of space required to store the generated data is thus equal to the number of observations drawn from the distribution multiplied by the space required to store one state. In typical situations, one state can be represented in a compact way. For instance, in this work, each state is uniquely identified by $n$ values (where the $n$ values represent the basic feasible solution corresponding to the state). Since the space required to represent one state is rather small, the space required to store the data to learn from is mainly dependent on the number of observations that are drawn from the distribution. Fortunately, this number is a parameter of FVI and the required space can thus be kept within acceptable limits.

### FVI time complexity

The time complexity of FVI depends on four components:

1. the number $L$ of iterations of the algorithm;
2. the number $H$ of states drawn from distribution $\mu$;
3. the time to compute the features;
4. the time taken by the learning algorithm to learn a model from the generated data.

Both the number of iterations of the algorithm and the size of the generated data are parameters of FVI and can therefore be chosen so that the time complexity does not get out of control. On top of the data generation, the features that are used as input of the learning algorithm must also be computable efficiently. This is a matter of major importance since it is possible to design very interesting features, but that can be too time consuming to compute and thus not usable for our purposes. This aspect is not discussed here since the choice of the good features remains an open question and depends on the particular polytope on which we want to apply the proposed approach (a more detailed discussion is proposed further below).

Finally, the last component of the time complexity directly depends on the chosen learning algorithm. Once the features are computed for the data drawn from the target distribution, they can be given as input to the learning algorithm that is then responsible for finding the best candidate in $\mathcal{F}$ to minimize the error between the values computed by the approximate Bellman operator and the prediction obtained with the target model (see line 6 in Algorithm 6). The time complexity of the algorithm has to be taken into account to evaluate the total time complexity of FVI. Since this time complexity depends on the particular algorithm

that is used within FVI, no specific information can be given here. For the sake of illustration, let us just mention that a simple learning algorithm like linear regression has a time complexity of (approximately) $\mathcal{O}(H^3)$, where $H$ corresponds to the number of observations in the sampled data.

Summing up the previous points, the total time complexity of FVI is given by

$$K_{\text{FVI}} = \mathcal{O}\left(LH^2 K_{\mathcal{C}} K_{\text{train}}(H)\right),$$

where $K_{\mathcal{C}}$ and $K_{\text{train}}(H)$ represent the time complexity needed to compute the features for one state and the time complexity of the learning algorithm that (typically) depends on $H$. Clearly, the time complexity of FVI remains under control (i.e., not exponential) provided that both $K_{\mathcal{C}}$ and $K_{\text{train}}(H)$ are kept at acceptable values.

### Theoretical guarantees on the FVI solution

We have seen that FVI offers simple means to control the time and space complexity of the training procedure. While FVI gets round the main limitations of VI, FVI loses the guaranteed convergence. Despite this shortcoming, there exist theoretical guarantees on the quality of the obtained solution. For this analysis, we rely on Munos and Szepesvári (2008) who propose a thorough study of the performance of FVI with respect to the considered learning algorithm, to the number $L$ of iterations, and to the number $H$ of samples generated at each iteration.

Before stating the main result of Munos and Szepesvári (2008) that we use in the following, let us first define two assumptions that are required in order to apply their theorem. Munos and Szepesvári (2008, Assumption A0)

**Assumption 2** (MDP regularity, Assumption A0 (Munos and Szepesvári, 2008))**.** *The MDP satisfies the following:*

1. *the state space $\mathcal{S}$ of the MDP is a bounded closed subset of some Euclidian space;*
2. *the action space $\mathcal{A}$ is finite;*
3. *the discount factor is lower and upper bounded: $0 < \gamma < 1$;*
4. *the reward function is a bounded measurable function with bound $R_{max}$;*
5. *the support of the 'reward kernel' $S(\cdot\,|s,a)$ (the probability distribution of the reward conditioned on the state $s$ and action $a$) is included in $\left[-\hat{R}_{max}, \hat{R}_{max}\right]$, independently of $(s,a) \in \mathcal{S} \times \mathcal{A}$.*

**Assumption 3** (Uniformly stochastic transitions, Assumption A1 (Munos and Szepesvári, 2008))**.** *For all $s \in \mathcal{S}$ and $a \in \mathcal{A}$, the probability transitions $p(a,s,s') = \mathrm{P}\left[s_{t+1} = s' \,|s_t = s, a_t = a\right]$ are such that*

$$p(a,s,\cdot) \leq C_\mu \mu(\cdot).$$

In the case of the MDP formulation of the simplex, we conjecture that both assumptions are satisfied (see the following for a discussion of the possible pitfalls). In that case, provided that

Assumptions A0 and A1 are satisfied, we can make use of Theorem 2 in Munos and Szepesvári (2008) that states the following (with some modifications in the notations).

**Theorem 5** (Theorem 2 (Munos and Szepesvári, 2008))**.** *For a fixed $p \geq 1$, a given distribution $\mu$ over the state space $\mathcal{S}$, and a given $J_0 \in \mathcal{F}$, then, for any $\varepsilon > 0$ and $\delta > 0$, there exist integers*

- *$L$ that is linear in $\log \frac{1}{\varepsilon}$, $\log J_{max}$, and $\log \frac{1}{1-\gamma}$,*
- *$H$ and $M$ that are polynomial in $\frac{1}{\varepsilon}$, $\log \frac{1}{\delta}$, $\log \frac{1}{1-\gamma}$, $J_{max}$, $\hat{R}_{max}$, $\log |\mathcal{A}|$, and $\log \mathcal{N}\left(\frac{c\varepsilon(1-\gamma)^2}{\gamma C_{\rho,\mu}^{\frac{1}{p}}}, \mathcal{F}, H, \mu\right)$, for some constant $c > 0$,*

*such that, if FVI is applied to the MDP and $\pi_L$ is a greedy policy with respect to the $L$-th iterate of FVI, i.e., $\tilde{J}_L$, then with probability at least $1 - \delta$,*

$$\left\| J^* - \tilde{J}_{\pi_L} \right\|_\infty \leq \frac{2\gamma}{(1-\gamma)^2} C_\mu^{\frac{1}{p}} d_{p,\mu}(\mathcal{TF}, \mathcal{F}) + \varepsilon.$$

We see directly that moving from the simple VI to the more realistic FVI considerably complicates the analysis of the error bounds. In order to analyze this result in the simplest way, let us first focus on the quality bound and then move on to the discussion of the assumptions and of the different parameters appearing in the theorem.

Before discussing how this theorem can be applied to our problem, let us specify the values and meaning of some variables appearing in the theorem. To start with, $J_{\max}$ represents the maximum value that the value function can take, which is equal to the restart term $\alpha$. $\hat{R}_{\max}$ represents the bounds on the support of the reward kernel, which, in our case, is equal to 1. Then, by definition (see Munos and Szepesvári (2008)), $C_{\rho,\mu}^{\frac{1}{p}}$ satisfies $C_{\rho,\mu}^{\frac{1}{p}} \leq (1-\gamma)^2 \sum_{m \geq 1} m\gamma^{m-1} C_\mu$. The value $d_{p,\mu}(\mathcal{TF}, \mathcal{F})$ is referred to as the 'inherent Bellman error' and represents a bound on the error of the individual iterations (Munos and Szepesvári, 2008). Finally, $\mathcal{N}(\cdot, \cdot, \cdot, \cdot)$ represents the expectation of the covering number of a set of identically distributed points whose description is omitted here (see Munos and Szepesvári (2008) for more information on this). Applying Theorem 5 to our problem requires a precise understanding of all the parameters and variables that appear here. Completing the proposed analysis requires paying careful attention to all these elements that may seem of minor importance, but that could have a huge impact on the final conclusions.

### FVI bounds applied to the simplex MDP

Theorem 5 as described above allows us to study how FVI behaves on the simplex MDP.

We first assume that one can find a feature description of the states and a learning algorithm that are both easy to compute (i.e., polynomial time) and that approximate accurately enough the consecutive iterates of the value function, i.e., $d_{p,\mu}(\mathcal{TF}, \mathcal{F}) = 0$. This equation can be expressed as

$$d_{p,\mu}(\mathcal{TF}, \mathcal{F}) = \sup_{g \in \mathcal{F}} \inf_{f \in \mathcal{F}} \|\mathcal{T}g - f\|_{p,\mu} = 0,$$

which, in words, means that applying the Bellman operator to a function that belongs to the function set $\mathcal{F}$ yields a new function that belongs to the same set.

If the inherent Bellman error term $d_{p,\mu}(\mathcal{TF}, \mathcal{F})$ can be set to 0, Corollary 1 gives us the maximum allowed error for an approximate value function to yield shortest paths in the polytope. We can then state the following conjecture.

**Conjecture 3** (Polynomial complexity of FVI)**.** *Provided that all the above assumptions are satisfied and that the inherent Bellman error is equal to 0, FVI finds optimal paths in the MDP if*

$$\varepsilon < \frac{1}{2}\gamma^{T_P - 1}(1 + \gamma\alpha)(1 - \gamma).$$

*Then, with this fixed value of $\varepsilon$, the integers $L$, $H$, and $M$ yield a complexity of FVI that is*

$$K_{FVI} = \mathcal{O}\left(LH^2 K_{\mathcal{C}} K_{train}(H)\right),$$

*which is polynomial in the initial dimensions of the optimization problem.*

Given the expressions of the integer parameters of FVI as stated by Theorem 5, it is realistic to assume that polynomial bounds for the training time of FVI could be derived while guaranteeing that the resulting pivoting rules are also polynomial. However, there remain a number of unanswered questions in this approach and we are not able, at this time, to prove the conjecture.

### Discussion of the assumptions

Conjecture 3 indicates that it is possible to find polynomial pivoting rules in polynomial time. However, caution must be taken in several ways. Indeed, some shortcuts have been used to derive this conjecture and these shortcuts must be thoroughly studied before one can state a theorem based on that conjecture. The following is a discussion of some aspects and of some shortcuts that led us to state the conjecture. This is by no means an exhaustive list of all the elements that must be taken into account in order to convert the conjecture into a theorem.

1. Munos and Szepesvári (2008) study FVI with the assumption that the state space $\mathcal{S}$ is continuous. This is not the case with the simplex MDP since we consider that the state space corresponds to the set of vertices of the polytope. The main question is thus: can similar performance bounds be derived for discrete state spaces?

2. Stochasticity is another aspect to be taken into account. Indeed, the setting in Munos and Szepesvári (2008) considers that both transitions and rewards are stochastic. In the simplex MDP, neither is. The deterministic nature of the simplex MDP must thus be factored in when considering Theorem 5 to investigate the role of stochasticity in the results.

3. Another difference between the main theorem on which the conjecture relies and the simplex MDP lies in the fact that the Bellman operator is estimated in Munos and Szepesvári (2008) (because of reward stochasticity, $M$ states are drawn from the distribution $p(a, s, \cdot)$ and averaged to estimate each argument of the max function in Equation 6.6). However, since the transitions and rewards are deterministic in the

simplex MDP, a single sample is sufficient to compute exactly each argument of the maximum. It is not clear how this can affect the results.

These are just a few aspects that we highlight here and that could be detrimental to the application of the proposed approach. Clearly all assumptions should be double-checked to make sure that no shortcut is left unstudied. Obviously, the rather complicated nature of the considered bounds renders this task cumbersome and far from trivial.

It is also worth noticing that all theorems depend on the restart term $\alpha$ (defined in Equation (6.5)). This term however does not depend on the method used to find the pivoting rules (it depends on the formulation of the simplex MDP), but appears in all the bounds. This indicates that some polytopes may be better suited for some solution methods depending on the corresponding values of their restart term.

Finally, in addition to checking that all assumptions are indeed satisfied, it is important to keep in mind that the method works only if

1. a training algorithm with a function set $\mathcal{F}$ rich enough can be found and if the training time of such an algorithm is polynomial in the dimensions of the problem;

2. features can be designed such that, together with the learning algorithm, they are able to approximate the sought value functions accurately enough and if they can be computed in polynomial time with respect to the dimensions of the problem.

Actually both elements need to be considered simultaneously. Indeed, choosing a representation for the value function goes hand in hand with the choice of the features. A representation will be able to accurately enough approximate the real value function only if appropriate features are given. And the features need to be taken into account (in particular their computational complexity) when a representation is chosen. A solution to one matter cannot be given without providing a solution to the other. Typically, a good choice for the value function representation and for the features is a choice that minimizes the inherent Bellman error $d_{p,\mu}(\mathcal{T}\mathcal{F}, \mathcal{F})$ (ideally sets it to 0). In words, minimizing the inherent Bellman error means that, when a new update of the approximate value function is computed, the projection error of that approximation onto the space $\mathcal{F}$ of candidate value functions is minimum. The difficulty of finding such a representation/features combo comes from the fact that the error needs to be 0 (or at least very small) at any iteration of FVI. This may seem out of reach in general cases, but, considering the very special structure of the value functions in the simplex MDP, finding a representation/features pair that satisfies this requirement in our case does not seem foolish. However, we want to emphasize here that this task is, in our opinion, far from being trivial.

All in all, we are very aware that meeting all requirements is rather hard. However, the primary purpose of this theoretical study is to be exploratory. Making everything work together may be hard, but we nonetheless think that the proposed methodology clearly points towards novel research directions in the fields of linear programming and of the simplex algorithm.

## 6.6 Discussion, comments, and remarks

From the theoretical point of view, the approach proposed above shows that MDPs are a promising way to deliver interesting insights about the theory of the simplex algorithm. However, we only discuss some theoretical aspects of the idea. This section is devoted to pushing the analysis a bit further and to giving some additional details about the developed theory.

### 6.6.1 Generate data to learn from

From a very practical perspective, applying FVI techniques to learn an optimal pivoting rule requires a training set to learn from. Generating a useful training set is not a simple task. Indeed, the quality of the training set critically conditions the quality of the learned policy (i.e., the learned pivoting rule). Actually, the learning task of RL algorithms is much easier if the training set contains good trajectories[2]. If such good trajectories do not appear in the training set, the ability of the RL algorithm to learn something useful may be hindered. Some care should thus be taken when generating a training set to learn from.

One possibility to generate a good training set from a practical point of view is as follows. Given a polytope $P$,

1. choose an arbitrary vertex (BFS) $\boldsymbol{x}' \in P$;
2. make that vertex optimal by appropriately choosing the objective function, i.e., choose a vector $\boldsymbol{c}'$ such that $\boldsymbol{x}'$ is an optimal solution to the problem;
3. from the (now) optimal vertex $\boldsymbol{x}'$, explore backwards the graph (this generates a trajectory that passes through the optimal vertex);
4. repeat several times Point 3 in order to generate multiple trajectories (a few long trajectories are preferable than many short trajectories).

This idea can be used to generate useful training sets that can be fed to the RL algorithm. This procedure merely describes a means by which a training set can be created. The procedure (size of the training set, length of the trajectories, etc.) needs to be tuned in accordance with the particular characteristics of the studied problem and of the considered RL algorithm.

### 6.6.2 Note on the greediness of pivoting rules

In this work, we relax the constraint that a pivoting rule must choose an improving (in the sense of the objective function) basic feasible solution as the next vertex to process. If the greediness is enforced, the graph corresponding to the polytope becomes a directed graph, where the directions of the edges depend on the cost function. It happens that, while the diameter of a polytope is independent of the cost function, the diameter (or its equivalent) of the greedy-directed graph does depend on $\boldsymbol{c}$. Because not all transitions are allowed in the

[2]The goodness of a trajectory usually depends on the considered reward function. For instance, in the case of $\mathcal{R}_1$ proposed hereabove, a good trajectory to learn from is one that goes through the optimal state.

directed graph, requiring the pivoting rule to be greedy may actually forbid the existence of a polynomial path between any vertex and the optimal one. Indeed, even the assumption that the Hirsch conjecture is true does not guarantee that a polynomial path exists between any two vertices in the corresponding directed graph.

If one is interested in the polynomiality of greedy pivoting rules, the modified Hirsch conjecture is no longer enough as a working assumption. For the greedy pivoting rules to have a chance to be polynomial on general polytopes, one must assume (or prove) that there exists a path of polynomial length in the directed graph between any vertex and the optimal vertex. This assumption is likely to be stronger than the modified Hirsch conjecture. Additionally, it is even possible that, for a given cost function and a given polyhedron, no polynomial pivoting rule exists. This leads us to think that the existence of polynomial greedy pivoting rules depends on the cost function at least as much as on the polytope itself and is a problem much harder than its non-greedy counterpart.

However, if it is conceivable that all paths are polynomial in a greedy-directed polytope, applying the same framework as the one proposed hereabove leads to the optimal (polynomial) solution. Our choice to relax the greediness of pivoting rules is thus only motivated by the fact that the diameter of any polytope is strongly believed to be polynomial, while no such statement can be formulated for the greedy-directed polytopes. Relaxing the greediness thus allows us to consider any polytope as potential candidate to apply our method, while the list of candidate polytopes would have been much shorter if the greediness was enforced.

## 6.7 Conclusion

In this chapter, we study an important theoretical question in the field of linear optimization: the worst case complexity of the simplex algorithm. To this end, we use the theory of reinforcement learning, a machine learning paradigm, and show how this theory can provide interesting tools to improve the theoretical analysis of the simplex algorithm.

Reinforcement learning is a learning framework whose goal is to teach an agent to take decisions in Markovian environments. In order to apply this framework to the simplex, we formulate the solution procedure of the simplex as a Markov decision process (MDP). More specifically, we regard the simplex as an agent that takes decisions in a state space (the vertices of the polytope) and whose goal is to maximize some criterion (a decreasing function of the number of iterations required to reach the optimal vertex). Once the simplex is formulated as an MDP, the theory of reinforcement learning provides theoretical tools to study the worst case complexity of the simplex.

Having modeled the search for a simplex pivoting rule as an MDP, we use several reinforcement learning algorithms and the corresponding theory to shed some light onto the time complexity of the simplex. In particular, we consider the value iteration (VI) and the fitted value iteration (FVI) algorithms. The theoretical results obtained with the value iteration analysis show that, in a polynomial number of iterations, the algorithm finds a pivoting rule that, when used within simplex, is also polynomial in the dimensions of the problem. The drawback of this approach is that the space complexity is exponential and so is the training time required to find the pivoting rule (because of the number of subiterations). In order to

alleviate these two issues, the fitted value iteration algorithm is considered. We show that using fitted value iteration both the space and training complexity are kept under control (i.e., are not exponential), but this is achieved at the expense of the accuracy. Moreover, complete study of the search for pivoting rules with FVI involves some advanced mathematical concepts that have left some questions unanswered and that lead to a conjecture instead of a theorem.

As such, the solutions that we propose are certainly not the end of the road since many points are left open. However, we believe that this work, although still in its early stages, can have a significant impact on the theory of the simplex algorithm. Indeed, we propose to tackle the search for a polynomial pivoting rule from a radically different perspective: rather than trying to find a pivoting rule and then prove that it is polynomial, we conjecture that, under some (possibly strong) assumptions, learning algorithms can find themselves polynomial pivoting rules with minimal human intervention. This could have a major impact on the way the community approaches the problem.

As a final note, let us emphasize that the theory of machine learning provides other theoretical tools that could be used to study this problem. In particular, reinforcement learning theory may not be the most suitable framework to prove that the time complexity of the simplex is polynomial in the dimensions of the problem. Other ML frameworks could provide the appropriate tools for such a study such as, for instance, the 'probably approximately correct learning' framework (Valiant, 1984).

# Chapter 7

# Conclusions and outlooks

Put briefly, the idea that we develop in this work consists in using learning techniques to improve the performance and understanding of optimization algorithms. This approach is not new and has been referred to by many names, one of which is 'learning to search' (Langley, 1985). Throughout the years, numerous methods leveraging this principle have been developed, but they were in general tailored to specific tasks. In this thesis, we try to tackle the problem from a higher perspective and, instead of focusing on specific applications, we turn our attention to common optimization algorithms. More precisely, we implement the idea on two of them, namely the simplex and the branch-and-bound algorithms, and use learning techniques to improve some of their mechanisms.

In the following, we first summarize the conclusions of each individual chapter. We then detail the objectives of the thesis and explain how, according to the obtained results, they have been achieved. We next present a few pragmatic research directions that directly follow the contents of this manuscript and finally conclude this work with some general remarks.

## 7.1 Conclusions of the individual chapters

### 7.1.1 Learning to branch in branch-and-bound

Our first attempt to use learning techniques to improve optimization algorithms focuses on branching strategies used within the branch-and-bound algorithm. The idea is to create a cheap approximation of a good branching strategy that is usually expensive to evaluate.

The proposed approach consists in observing branching decisions taken by a supposedly good strategy, strong branching in our case, and to imitate those decisions with a strategy obtained by a machine learning procedure. To this end, we develop a set of features that are used to characterize the current state of the problem in the B&B tree from the perspective of a particular variable. These features are computed for all candidate variables and used as input of the learned branching heuristic in order to predict an approximation of the strong branching score for that variable. The underlying mechanism of the developed approach is not different from other branching strategies. Indeed, in all cases, features are computed from the

current state of the problem (in some way or another) and then used to decide which variable to branch on. In our approach, however, we can include many types of features, including those used by popular strategies, and let the learning algorithm decide which features are the best to predict a given branching score. In that sense, we can see our method as a very general branching strategy that can imitate any other heuristic, as long as the appropriate features are provided.

We propose two approaches that use machine learning: a batch approach and an online approach. In the batch case, a dataset must be generated prior to the optimization and fed to a machine learning algorithm in order to train a model on the data that approximates strong branching. In the online case, it is not necessary to generate a dataset, since the online learning algorithm trains the model during the course of the optimization. The experiments performed to assess the efficiency of both methods show promising results and suggest that further research in this direction may lead to favorable improvements in MIP solvers.

### 7.1.2 Learning to estimate the difficulty of a MIP problem and its use in parallelization

The next research direction that we investigate consists in using learning techniques to estimate the difficulty of a MIP problem. As an application of the created difficulty estimator, we illustrate how evaluating the difficulty of a problem can be used to improve the parallelization of a naive parallel branch-and-bound.

More specifically, we propose an approach to split the optimization of a single problem into several independent parts that can be solved by several workers in parallel, with the goal that the amount of work given to each processor is well balanced between the workers. The parallelization scheme is very naive and the innovation mostly lies in how the difficulty of the subproblems is estimated. To do this, we create a function, with the use of learning techniques, that is able to estimate the number of nodes, hence the amount of work, that a solver needs to process in order to solve to optimality a subproblem of the original problem. To this end, we develop a set of features that are used to characterize a given subproblem in the B&B tree and use these features as input of the learned function in order to predict the expected number of nodes required to solve the given subproblem to optimality. The estimates of the numbers of nodes are then used to create a partition of the original optimization tree so that one or several elements of the partition can be given to each worker. The experiments show that our approach succeeds in balancing the amount of work between the processors and that interesting speedups can be achieved with little effort.

### 7.1.3 Learning theory applied to the simplex algorithm

In the last part of this work, we show how the theory of machine learning can be used to improve the theoretical analysis of optimization algorithms. More specifically, we tackle an important question in the field of linear optimization: the worst case complexity of the simplex algorithm.

More precisely, we use the theory of reinforcement learning (RL), which is a learning framework whose goal is to teach an agent to take decisions in a Markovian environment.

The basic idea of our development is to formulate the simplex algorithm as a Markov decision process (MDP) on which the reinforcement learning framework can be applied. In this case, the decisions taken by the agent determine the path of the simplex algorithm (through the basic feasible solutions of the problem). Once it is formulated as an MDP, the theory of reinforcement learning provides new tools to study the worst case complexity of the simplex algorithm.

We tackle the problem with two different RL algorithms that have been studied theoretically quite extensively. We first consider a naive RL algorithm (value iteration or VI) and show that this algorithm is able to find pivoting rules that are guaranteed to be polynomial if enough time is given to the algorithm. The found pivoting rule is indeed guaranteed to be polynomial when used to solve a problem, but the time required to find the pivoting rule (the training time of the algorithm) is not. Additionally, the space required by the approach is proportional to the number of vertices of the polytope. This proof is thus not satisfactory since the learning step (and the required space) renders the entire procedure not polynomial. To get round this issue, we consider another RL algorithm (fitted value iteration or FVI) whose fundamental mechanisms are meant to render the training time tractable. This algorithm has been studied theoretically and there exist, similarly to the simpler VI, theoretical guarantees on the quality of the solutions found by the algorithm. Using both the theory of FVI and the fact that the training complexity can be controlled, we show how such an approach can get us one step closer to finding polynomial pivoting rules for the simplex algorithm. Note that we present here only a draft of a global proof and further work is required in order to leverage the full power of RL theory to answer optimization-related questions.

We believe that this work, although still in its early stages, can have a significant impact on the theory of the simplex algorithm. Indeed, we propose to tackle the search for a polynomial pivoting rule from a radically different perspective, which is, in our opinion, easier to approach. Rather than trying to find a pivoting rule and then prove that it is polynomial, we conjecture that, under some assumptions, learning algorithms can find themselves polynomial pivoting rules without human intervention. We believe that this new approach of the subject may lead to major breakthroughs in the field.

## 7.2 Objectives of the thesis

The raison d'être of this thesis is threefold.

First, we show that massive amounts of data are generated for free when one solves an optimization problem. It turns out that, currently, only little, if any, knowledge is gained from this available data. Therefore, we additionally propose several ways to leverage this data through appropriate techniques in the context of optimization algorithms.

Second, we show that the information gathered and interpreted by learning techniques can not only be used to improve the performance of optimization algorithms, but also to better understand how they work and what conditions their efficiency. For instance, in the case of variable branching, an analysis of the feature importances highlights which features are important and which are irrelevant. This information sheds some light on how branching

works and can improve the understanding that researchers have on that crucial step of branch-and-bound.

Lastly, mathematical optimization is a field of science that has been primarily governed by pure mathematicians who tend to be reluctant when it comes to not-so-theoretical disciplines like machine learning. In this work, we believe that we have produced convincing evidence that machine learning should not be disregarded and that, quite to the contrary, machine learning should be considered as a front-running candidate to improve optimization algorithms. We hope that this modest contribution will, at least, draw attention to such methodologies and, at best, lead to major breakthroughs that will dramatically improve the field of mathematical optimization.

## 7.3 Outlooks and future research

There is still a lot to be done in the research direction that we have investigated. We mention in the following a few elements that deserve, in our opinion, some attention.

In this work, a large part of the research was devoted to the design of appropriate features. Some of them have proved to be crucial to the methods, but others turned out to be much less important. One obvious first step on the path of improvement would be to develop more advanced (tailored?) features describing the studied problems. This could be done by hand, as in this thesis, or through automatic methods. This is indeed rendered possible by the recent developments of machine learning. For instance, in the past few years, the learning community has witnessed the rapid growth of deep learning methods. These methods have been applied successfully in many applications (including speech recognition and image classification), but they are also known to automatically extract important features from the raw data they are fed with. Using this kind of methods to automatize the feature representation is a promising research direction in the present case where the features are not trivial to identify nor design.

Second, it is important to emphasize that this work completely, and voluntarily, omits the search for the best learning algorithm and its optimal parameters. We did not consider this part as a crucial component of our research and consequently used algorithms that are known to perform well while being robust with respect to the choice of the parameter values. However, if the methodology that we propose is meant to be used in competitive environments, it is clear that considerable effort should be put into identifying which learning algorithm performs best for a dedicated task. Achieving state-of-the-art performance most definitely requires such a (possibly cumbersome) preliminary step.

Finally, it goes without saying that applying the proposed methodology to other problem families and other optimization algorithms than the ones considered in this work is of great practical interest.

## 7.4 Final word

In this work, we believe that two aspects clearly surface. On the one hand, free data is generated when one optimizes a problem and, on the other hand, that data contains a lot

of helpful information that can be used to improve the internal machinery of an optimization algorithm and its general behavior. With that in mind, one can regard optimization algorithms as objects that, in addition to their direct goal, which is to solve an optimization problem, are given the possibility to evolve and improve. It would be interesting to study optimization algorithms under the light of the well-known exploration-exploitation tradeoff. Under the scope of this framework, the development of optimization algorithms would be totally different. Indeed, one would need to entirely rethink the way these algorithms work to account for the fact that they are given short-term and long-term goals. The short-term goals would correspond to the solution of a single optimization problem, while the long-term goals would push the algorithm to learn a lot from the past in order to be more efficient in the future. Making optimization decisions in that context would be entirely different since one would need to take into account several objectives, among which some of them are most likely stochastic. The advent of such algorithms would probably imply the obsolescence of current optimization researchers, but, fortunately for us, it is unlikely to happen overnight.

With this work, we barely scratch the surface of the general idea that optimization can benefit from learning, but we believe that the potential for improving optimization techniques with machine learning is huge. Overall, we hope that this work will foster collaborations between optimizers and machine learners and will further bridge the gap that still separates both fields.

Finally, let us emphasize again that this thesis is intended as a proof-of-concept and merely focuses on the simplest optimization problems (linear and mixed-integer linear optimization). We strongly believe, however, that the same machine learning reasoning can be used much more efficiently in the situations that are studied in this work and that similar approaches can also be successfully applied to other, more difficult, optimization problems, such as convex, stochastic, and global optimization. It is just a matter of figuring out how.

# Appendix A

# Machine learning for variable branching: appendix

This appendix brings additional information about the methodology deployed to improve the design of branching strategies with learning techniques as detailed in Chapter 4.

## A.1 Detailed feature importance results

This appendix reports the detailed feature importances and costs of omission for all variables developed for this application. The results are given in Tables A.1 and A.2. For an analysis of the tables, we refer the reader to Section 4.6.2 of the main document.

| | ET | | | LR | | | Lasso ($\alpha = 0.01$) | | |
|---|---|---|---|---|---|---|---|---|---|
| # | FI | MRE | COO | FI | MRE | COO | FI | MRE | COO |
| 2 | 0.2875 | 0.1591 | -4 | 0.0000 | 0.2078 | -262 | 0.0000 | 0.2141 | -38 |
| 27 | 0.1297 | 0.1596 | -3 | 0.0000 | 0.2077 | -263 | 0.1213 | 0.2141 | -38 |
| 37 | 0.0497 | 0.1892 | 100 | 0.0000 | 0.2257 | 100 | 1.0000 | 0.2341 | 100 |
| 40 | 0.0486 | 0.1552 | -18 | 0.0000 | 0.2078 | -261 | 0.0451 | 0.2141 | -38 |
| 31 | 0.0449 | 0.1587 | -6 | 0.0000 | 0.2078 | -262 | 0.2423 | 0.2150 | -32 |
| 18 | 0.0413 | 0.1569 | -12 | 0.0000 | 0.2077 | -263 | 0.0009 | 0.2117 | -55 |
| 63 | 0.0289 | 0.1574 | -10 | 0.0000 | 0.2106 | -204 | 0.1037 | 0.2157 | -27 |
| 56 | 0.0287 | 0.1571 | -11 | 0.0000 | 0.2073 | -272 | 0.0000 | 0.2141 | -38 |
| 57 | 0.0286 | 0.1580 | -8 | 0.0000 | 0.2182 | -50 | 0.6523 | 0.2235 | 27 |
| 58 | 0.0273 | 0.1562 | -14 | 0.0000 | 0.2084 | -249 | 0.0052 | 0.2133 | -44 |
| 30 | 0.0213 | 0.1574 | -10 | 0.0000 | 0.2078 | -260 | 0.0953 | 0.2141 | -38 |
| 62 | 0.0207 | 0.1573 | -11 | 0.0000 | 0.2077 | -263 | 0.0302 | 0.2143 | -37 |
| 61 | 0.0198 | 0.1571 | -11 | 0.0000 | 0.2058 | -302 | 0.2220 | 0.2158 | -27 |
| 39 | 0.0191 | 0.1591 | -5 | 0.0000 | 0.2078 | -261 | 0.0000 | 0.2141 | -38 |
| 38 | 0.0180 | 0.1568 | -12 | 0.0000 | 0.2078 | -261 | 0.0000 | 0.2141 | -38 |
| 36 | 0.0152 | 0.1657 | 18 | 0.0000 | 0.2067 | -283 | 0.0415 | 0.2141 | -38 |
| 59 | 0.0146 | 0.1585 | -7 | 0.0000 | 0.2078 | -262 | 0.0142 | 0.2141 | -38 |
| 43 | 0.0144 | 0.1557 | -16 | 0.0000 | 0.1967 | -486 | 0.0003 | 0.1991 | -142 |
| 64 | 0.0118 | 0.1599 | -2 | 0.0000 | 0.2078 | -262 | 0.0023 | 0.2141 | -38 |
| 41 | 0.0116 | 0.1564 | -14 | 0.0000 | 0.2089 | -239 | 0.0008 | 0.2150 | -32 |
| 42 | 0.0105 | 0.1575 | -10 | 0.0000 | 0.2086 | -245 | 0.0006 | 0.2155 | -29 |
| 23 | 0.0102 | 0.1563 | -14 | 0.0000 | 0.2079 | -259 | 0.0000 | 0.2141 | -38 |
| 46 | 0.0097 | 0.1579 | -9 | 0.0000 | 0.2071 | -275 | 0.0846 | 0.2139 | -40 |
| 55 | 0.0071 | 0.1576 | -10 | 0.0000 | 0.2078 | -261 | 0.0000 | 0.2141 | -38 |
| 60 | 0.0061 | 0.1602 | -1 | 0.0000 | 0.2084 | -248 | 0.0000 | 0.2141 | -38 |
| 35 | 0.0056 | 0.1583 | -7 | 0.0000 | 0.2078 | -260 | 0.1511 | 0.2141 | -38 |
| 7 | 0.0056 | 0.1580 | -8 | 0.0000 | 0.2078 | -262 | 0.0001 | 0.2141 | -38 |
| 47 | 0.0049 | 0.1583 | -7 | 0.0000 | 0.2078 | -261 | 0.0013 | 0.2141 | -38 |
| 48 | 0.0044 | 0.1574 | -10 | 0.0000 | 0.2078 | -262 | 0.0048 | 0.2141 | -39 |
| 4 | 0.0043 | 0.1574 | -10 | 0.0000 | 0.2078 | -262 | 0.0358 | 0.2141 | -38 |
| 34 | 0.0043 | 0.1585 | -6 | 0.0000 | 0.2080 | -258 | 0.0000 | 0.2141 | -38 |
| 3 | 0.0043 | 0.1572 | -11 | 0.0000 | 0.2077 | -264 | 0.0000 | 0.2141 | -38 |

Table A.1: Feature importances (1/2) as computed by the ExtraTrees (ET), linear regression (LR), and the Lasso. Each row of the table corresponds to a feature, whose number is given in the first column. 'FI' represents the feature importance for the corresponding feature. For the ExtraTrees, the feature importances are the result of the internal procedure run during the learning phase. For LR and the Lasso, the feature importances correspond to the (normalized) absolute values of the regression coefficients. Note that, in the case of LR, a few coefficients are very large (order of $10^7$) and thus the relative importance of the other features is seemingly null. 'MRE' represents the mean relative error obtained on the test set when the considered feature is eliminated from learning and testing. The COOs are obtained by comparing the MREs obtained without the features and the values reported in Table 4.11.

| | ET | | | LR | | | Lasso ($\alpha = 0.01$) | | |
|---|---|---|---|---|---|---|---|---|---|
| # | FI | MRE | COO | FI | MRE | COO | FI | MRE | COO |
| 11 | 0.0042 | 0.1571 | -11 | 0.0000 | 0.2078 | -262 | 0.0001 | 0.2141 | -38 |
| 10 | 0.0041 | 0.1550 | -19 | 0.0000 | 0.2078 | -261 | 0.0002 | 0.2142 | -38 |
| 54 | 0.0040 | 0.1562 | -14 | 0.0000 | 0.2078 | -261 | 0.0000 | 0.2141 | -38 |
| 52 | 0.0038 | 0.1589 | -5 | 0.0000 | 0.2078 | -262 | 0.0037 | 0.2141 | -38 |
| 6 | 0.0038 | 0.1582 | -8 | 0.0000 | 0.2078 | -261 | 0.0001 | 0.2141 | -38 |
| 8 | 0.0028 | 0.1602 | -0 | 0.0000 | 0.2078 | -262 | 0.0000 | 0.2141 | -38 |
| 12 | 0.0027 | 0.1604 | 0 | 0.3204 | 0.2078 | -262 | 0.0000 | 0.2141 | -38 |
| 44 | 0.0021 | 0.1594 | -3 | 0.0000 | 0.2078 | -262 | 0.0000 | 0.2141 | -38 |
| 13 | 0.0020 | 0.1556 | -16 | 0.3204 | 0.2078 | -262 | 0.0000 | 0.2141 | -38 |
| 53 | 0.0020 | 0.1580 | -8 | 0.0000 | 0.2078 | -262 | 0.0001 | 0.2141 | -38 |
| 51 | 0.0015 | 0.1587 | -6 | 0.0000 | 0.2078 | -262 | 0.0000 | 0.2141 | -38 |
| 50 | 0.0014 | 0.1594 | -4 | 0.0000 | 0.2078 | -261 | 0.0042 | 0.2141 | -38 |
| 49 | 0.0014 | 0.1570 | -12 | 0.0000 | 0.2077 | -263 | 0.0044 | 0.2141 | -39 |
| 9 | 0.0013 | 0.1570 | -12 | 0.0000 | 0.2078 | -261 | 0.0123 | 0.2142 | -38 |
| 45 | 0.0010 | 0.1582 | -8 | 1.0000 | 0.2078 | -262 | 0.0013 | 0.2141 | -38 |
| 14 | 0.0010 | 0.1571 | -11 | 0.0000 | 0.2081 | -254 | 0.0005 | 0.2144 | -36 |
| 1 | 0.0006 | 0.1566 | -13 | 0.3959 | 0.2078 | -262 | 0.0222 | 0.2141 | -38 |
| 15 | 0.0005 | 0.1582 | -8 | 0.0367 | 0.2078 | -262 | 0.0000 | 0.2141 | -38 |
| 17 | 0.0002 | 0.1565 | -13 | 0.7171 | 0.2078 | -262 | 0.0000 | 0.2141 | -38 |
| 16 | 0.0002 | 0.1563 | -14 | 0.1496 | 0.2078 | -262 | 0.0000 | 0.2141 | -38 |
| 5 | 0.0001 | 0.1570 | -12 | 0.0000 | 0.2078 | -262 | 0.0000 | 0.2141 | -38 |
| 21 | 0.0001 | 0.1568 | -13 | 0.0000 | 0.2078 | -262 | 0.0000 | 0.2141 | -38 |
| 25 | 0.0000 | 0.1572 | -11 | 0.0000 | 0.2078 | -262 | 0.0000 | 0.2141 | -38 |
| 33 | 0.0000 | 0.1565 | -13 | 0.0749 | 0.2078 | -262 | 0.0000 | 0.2141 | -38 |
| 28 | 0.0000 | 0.1575 | -10 | 0.0749 | 0.2078 | -262 | 0.0000 | 0.2141 | -38 |
| 29 | 0.0000 | 0.1593 | -4 | 0.0746 | 0.2078 | -262 | 0.0000 | 0.2141 | -38 |
| 32 | 0.0000 | 0.1577 | -9 | 0.0749 | 0.2078 | -262 | 0.0000 | 0.2141 | -38 |
| 19 | 0.0000 | 0.1567 | -13 | 0.0000 | 0.2078 | -262 | 0.0000 | 0.2141 | -38 |
| 20 | 0.0000 | 0.1561 | -15 | 0.0000 | 0.2078 | -262 | 0.0000 | 0.2141 | -38 |
| 22 | 0.0000 | 0.1585 | -6 | 0.0000 | 0.2078 | -262 | 0.0000 | 0.2141 | -38 |
| 24 | 0.0000 | 0.1586 | -6 | 0.0000 | 0.2078 | -262 | 0.0000 | 0.2141 | -38 |
| 26 | 0.0000 | 0.1596 | -3 | 0.0000 | 0.2078 | -262 | 0.0000 | 0.2141 | -38 |

Table A.2: Feature importances (2/2) as computed by the ExtraTrees (ET), linear regression (LR), and the Lasso. Each row of the table corresponds to a feature, whose number is given in the first column. 'FI' represents the feature importance for the corresponding feature. For the ExtraTrees, the feature importances are the result of the internal procedure run during the learning phase. For LR and the Lasso, the feature importances correspond to the (normalized) absolute values of the regression coefficients. Note that, in the case of LR, a few coefficients are very large (order of $10^7$) and thus the relative importance of the other features is seemingly null. 'MRE' represents the mean relative error obtained on the test set when the considered feature is eliminated from learning and testing. The COOs are obtained by comparing the MREs obtained without the features and the values reported in Table 4.11.

## A.2 About ExtraTrees parameters

The performance of ExtraTrees (Geurts et al., 2006) is very robust with respect to the choice of their parameters. ExtraTrees actually have three parameters: $N$, which is the number of trees in the method; $k$, which is the number of features evaluated at each node during the creation of the trees; and $n_{\min}$, which is the number of learning samples contained in a node below which that node becomes a leaf. The number of trees is set to the default value of $N = 100$ in our experiments. The parameter $k$, which represents the number of features that are considered for the creation of the next node in the ExtraTrees, is also set to a default value of $k = |\phi|$. The exact understanding of these parameters is beyond the scope of this appendix and we refer the reader to the paper of Geurts et al. (2006) for a deeper explanation.

Tables A.3, A.4, A.5, and A.6 compare the influence of the parameter $n_{\min}$ on the performance of the method. The experiments are the same as those detailed in Section 4.4.3 of the main document. The main observation that can be made from those tables is that the parameter $n_{\min}$ influences the computational time, but not really the accuracy of the taken decisions. Indeed, the greater the $n_{\min}$, the faster the method. This behavior was expected, as a large $n_{\min}$ produces smaller trees, which generally reduces the computational time required to take a branching decision. On the other hand, the third column of Tables A.3 and A.6, and the sixth column of Table A.4, all of which correspond to the gap closed after the node limit has been reached, show that the accuracy of the taken decisions is not influenced by $n_{\min}$ in the range of tested values. These observations illustrate that the method is actually robust to the choice of $n_{\min}$.

Besides the experiments included in this appendix, further work should focus on a more detailed study of the influence of the different parameters of the ExtraTrees on the performance of the optimization procedure.

| | Node limit ($10^5$ nodes) | | | Time limit (10 min.) | | |
|---|---|---|---|---|---|---|
| | S/T | Cl. Gap | Time | S/T | Cl. Gap | Nodes |
| Learned - $n_{\min} = 1$ | 0/150 | 0.62 | 72.23 | 16/150 | 0.81 | 104,090 |
| Learned - $n_{\min} = 5$ | 0/150 | 0.62 | 63.28 | 19/150 | 0.82 | 114,389 |
| Learned - $n_{\min} = 10$ | 0/150 | 0.62 | 61.91 | 21/150 | 0.83 | 122,038 |
| Learned - $n_{\min} = 20$ | 0/150 | 0.62 | 54.23 | 23/150 | 0.84 | 131,994 |

Table A.3: Optimization results for the problems of BPEQ_test, BPSC_test and MKNSC_test.

| | Solved by all methods | | | Not solved by at least one method | | | |
|---|---|---|---|---|---|---|---|
| | S/T | Nodes | Time | S/T | Cl. Gap | Nodes | Time |
| Learned - $n_{\min} = 1$ | 9/44 | 1,436 | 3.42 | 11/44 | 0.63 | 8,084 | 110.70 |
| Learned - $n_{\min} = 5$ | 9/44 | 1,229 | 3.25 | 10/44 | 0.62 | 8,176 | 103.34 |
| Learned - $n_{\min} = 10$ | 9/44 | 1,437 | 5.45 | 10/44 | 0.63 | 8,083 | 84.75 |
| Learned - $n_{\min} = 20$ | 9/44 | 1,194 | 2.73 | 10/44 | 0.62 | 8,073 | 162.87 |

Table A.4: Optimization results for the MIPLIB problems. Node limit = $10^5$ nodes.

| | Solved by all methods | | | Not solved by at least one method | | | |
|---|---|---|---|---|---|---|---|
| | S/T | Nodes | Time | S/T | Cl. Gap | Nodes | Time |
| Learned - $n_{\min} = 1$ | 19/44 | 13,887 | 36.19 | 5/44 | 0.64 | 112,810 | 510.72 |
| Learned - $n_{\min} = 5$ | 19/44 | 15,310 | 38.46 | 5/44 | 0.64 | 116,493 | 525.46 |
| Learned - $n_{\min} = 10$ | 19/44 | 14,647 | 37.42 | 7/44 | 0.65 | 124,346 | 499.55 |
| Learned - $n_{\min} = 20$ | 19/44 | 14,008 | 34.12 | 5/44 | 0.63 | 130,081 | 512.72 |

Table A.5: Optimization results for the MIPLIB problems. Time limit = 10 min.

| | Node limit ($10^5$ nodes) | | | Time limit (10 min.) | | |
|---|---|---|---|---|---|---|
| | S/T | Cl. Gap | Time | S/T | Cl. Gap | Nodes |
| Learned - $n_{\min} = 1$ | 0/50 | 0.47 | 76.20 | 0/50 | 0.63 | 83,938 |
| Learned - $n_{\min} = 5$ | 0/50 | 0.47 | 67.31 | 0/50 | 0.64 | 94,826 |
| Learned - $n_{\min} = 10$ | 0/50 | 0.47 | 68.58 | 0/50 | 0.65 | 102,941 |
| Learned - $n_{\min} = 20$ | 0/50 | 0.48 | 56.36 | 0/50 | 0.67 | 112,918 |
| Learned - $n_{\min} = 1$ - BPSC only | 0/50 | 0.51 | 90.69 | 0/50 | 0.67 | 77,174 |
| Learned - $n_{\min} = 5$ - BPSC only | 0/50 | 0.51 | 79.53 | 0/50 | 0.67 | 82,535 |
| Learned - $n_{\min} = 10$ - BPSC only | 0/50 | 0.51 | 72.79 | 0/50 | 0.68 | 92,586 |
| Learned - $n_{\min} = 20$ - BPSC only | 0/50 | 0.51 | 60.54 | 0/50 | 0.70 | 109,066 |

Table A.6: Optimization results for the problems from BPSC_test. Comparison between the strategy learned on the entire training set and the strategy learned only from BPSC_train examples.

## A.3 Batch learning of branching decisions: complete experimental results

This section contains the detailed experimental results for the MIPLIB problems used in our experiments. Averaging the following results over all problems gives the aggregated results shown in the main document. The detailed results are given in Tables A.7 through A.20.

The first set of tables, i.e., Tables A.7 through A.14, reports the results for the MIPLIB problems contained in Table 4.6 when limits are set either on the number of nodes or on the time spent. The B&B version that is used for these experiments is pure, i.e., no cuts nor heuristics are used.

The second set of tables, i.e., Tables A.15 through A.20, reports the optimization results for the problems contained in Table 4.9. These problems are obtained by keeping from the initial list of problems those that are solved within a 5 days time limit with all considered branching strategies. Tables A.15 through A.20 then report the optimization results with no time (or node) limit on the second list of MIPLIB problems. The first three tables correspond to the results obtained with a pure version of B&B, while the second half of the tables contains the results when CPLEX's cuts and heuristics are used during the optimization.

| | Problem names | 10teams | aflow30a | aflow40b | air03 | air04 | air05 | cap6000 | dcmulti | egout | fiber | fixnet6 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | LP Obj. | 917 | 983.17 | 1,005.66 | 338,864 | 55,535.40 | 25,877.60 | -2,450,000 | 183,976 | 149.59 | 156,083 | 1,200.88 |
| | True Obj. | 924 | 1,158 | 1,170 | 340,160 | 56,100 | 26,374 | -2,450,000 | 188,182 | 568.10 | 405,935 | 3,983 |
| | Fin. by all | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Random | Fin. | 4 | 5 | 5 | 0 | 4 | 4 | 5 | 5 | 4 | 5 | 5 |
| | Obj. | 917 | 1,051.18 | 1,043.31 | 340,160 | 55,946.50 | 26,234.50 | -2,450,000 | 187,298 | 562.60 | 189,746 | 1,573.11 |
| | Nodes | 10,000 | 10,000 | 10,000 | 25 | 10,000 | 10,000 | 10,000 | 10,000 | 10,000 | 10,000 | 10,000 |
| | Time | 364.40 | 9.35 | 44.47 | 1.45 | 2,219.56 | 1,173.48 | 30.70 | 5.61 | 1.12 | 5.39 | 4 |
| | Cl. Gap | 0 | 0.39 | 0.23 | 1 | 0.68 | 0.72 | 0.41 | 0.79 | 0.99 | 0.13 | 0.13 |
| MIB | Fin. | 4 | 5 | 5 | 0 | 4 | 5 | 5 | 0 | 0 | 5 | 5 |
| | Obj. | 920 | 1,059.97 | 1,038.70 | 340,160 | 55,980.80 | 26,270.30 | -2,450,000 | 188,182 | 568.10 | 184,391 | 2,004.30 |
| | Nodes | 10,000 | 10,000 | 10,000 | 7 | 10,000 | 10,000 | 10,000 | 9,493 | 7,015 | 10,000 | 10,000 |
| | Time | 251.55 | 10.44 | 46.41 | 0.60 | 5,965.28 | 1,516.76 | 29.02 | 4.44 | 0.79 | 5.91 | 4.12 |
| | Cl. Gap | 0.43 | 0.44 | 0.20 | 1 | 0.74 | 0.79 | 0.41 | 1 | 1 | 0.11 | 0.29 |
| NCB | Fin. | 0 | 5 | 5 | 0 | 0 | 0 | 5 | 0 | 0 | 5 | 5 |
| | Obj. | 924 | 1,117.99 | 1,099.12 | 340,160 | 56,137 | 26,374 | -2,450,000 | 188,182 | 568.10 | 359,328 | 2,528.38 |
| | Nodes | 1,453 | 10,000 | 10,000 | 3 | 131 | 215 | 10,000 | 1,065 | 5,713 | 10,000 | 10,000 |
| | Time | 456.18 | 98.16 | 354.31 | 2.04 | 1,146.13 | 772.90 | 147.50 | 2.99 | 1.94 | 52.82 | 74.06 |
| | Cl. Gap | 1 | 0.77 | 0.58 | 1 | 1 | 1 | 0.44 | 1 | 1 | 0.81 | 0.48 |
| FSB | Fin. | 0 | 5 | 5 | 0 | 0 | 0 | 5 | 0 | 0 | 5 | 5 |
| | Obj. | 924 | 1,120.14 | 1,099.87 | 340,160 | 56,137 | 26,400 | -2,450,000 | 188,182 | 568 | 362,329 | 2,482.51 |
| | Nodes | 259 | 10,000 | 10,000 | 3 | 111 | 177 | 10,000 | 927 | 4,730 | 10,000 | 10,000 |
| | Time | 1,215.41 | 217.34 | 979.50 | 2.75 | 3,249.42 | 3,840 | 159 | 4.83 | 2.16 | 267.72 | 246.11 |
| | Cl. Gap | 1 | 0.78 | 0.58 | 1 | 1 | 1 | 0.44 | 1 | 1 | 0.83 | 0.46 |
| RB | Fin. | 5 | 5 | 5 | 0 | 0 | 0 | 5 | 0 | 0 | 5 | 5 |
| | Obj. | 917 | 1,103.69 | 1,089.99 | 340,160 | 56,137 | 26,374 | -2,450,000 | 188,182 | 568.10 | 287,517 | 2,322.90 |
| | Nodes | 10,000 | 10,000 | 10,000 | 3 | 3,601 | 1,489 | 10,000 | 1,013 | 9,143 | 10,000 | 10,000 |
| | Time | 1,200.43 | 37.74 | 136.16 | 0.53 | 2,786.84 | 1,461.74 | 118 | 1.79 | 2.37 | 18.77 | 18.36 |
| | Cl. Gap | 0 | 0.69 | 0.52 | 1 | 1 | 1 | 0.44 | 1 | 1 | 0.53 | 0.40 |
| Learned | Fin. | 0 | 5 | 5 | 0 | 0 | 4 | 5 | 0 | 0 | 5 | 5 |
| | Obj. | 924 | 1,064.88 | 1,039.41 | 340,160 | 56,137 | 26,284.40 | -2,450,000 | 188,182 | 568.10 | 313,565 | 2,239.68 |
| | Nodes | 1,739 | 10,000 | 10,000 | 3 | 3,833 | 10,000 | 10,000 | 1,565 | 3,007 | 10,000 | 10,000 |
| | Time | 108.16 | 57.10 | 146.12 | 0.48 | 1,648.75 | 2,237.31 | 45.80 | 2.76 | 1.31 | 42.17 | 61.90 |
| | Cl. Gap | 1 | 0.47 | 0.21 | 1 | 1 | 0.82 | 0.44 | 1 | 1 | 0.63 | 0.37 |

Table A.7: Detailed results for the MIPLIB problems (1/4). Node limit = $10^5$ nodes. The row 'Cl. Gap' refers to the gap closed at the end of the optimization whether it achieved optimality or not. The row 'Fin. by all' indicates whether all methods were able to solve this problem to optimality. For each method, the 'Fin.' rows indicate the termination status of the optimization: 0 for optimality, 1 for unfeasibility, 2 for unboundedness, 3 for unfeasibility or unboundedness, 4 for another stopping criterion with a feasible solution found, and 5 for another stopping criterion with no feasible solution found.

| Problem names | | harp2 | khb05250 | l152lav | lseu | mas74 | mas76 | misc03 | misc06 | misc07 | mitre | mod008 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| LP Obj. | | -74,300,000 | 95,900,000 | 4,656.36 | 834.68 | 10,482.80 | 38,893.90 | 1,910 | 12,841.70 | 1,415 | 114,741 | 290.93 |
| True Obj. | | -73,900,000 | 107,000,000 | 4,722 | 1,120 | 11,801.20 | 40,005.10 | 3,360 | 12,850.90 | 2,810 | 115,155 | 307 |
| Fin. by all | | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| Random | Fin. | 5 | 4 | 4 | 5 | 5 | 5 | 0 | 0 | 4 | 5 | 4 |
| | Obj. | -74,300,000 | 107,000,000 | 4,700.21 | 1,005.77 | 10,860.40 | 39,280.80 | 3,360 | 12,851.10 | 2,294.29 | 115,094 | 305.69 |
| | Nodes | 10,000 | 10,000 | 10,000 | 10,000 | 10,000 | 10,000 | 1,179 | 527 | 10,000 | 10,000 | 10,000 |
| | Time | 16.92 | 6.17 | 32.30 | 1.10 | 3.37 | 3.04 | 0.51 | 0.69 | 10.44 | 182.61 | 1.50 |
| | Cl. Gap | 0.16 | 0.98 | 0.67 | 0.60 | 0.29 | 0.35 | 1 | 1.02 | 0.63 | 0.85 | 0.92 |
| MIB | Fin. | 5 | 0 | 4 | 5 | 5 | 5 | 0 | 0 | 4 | 0 | 0 |
| | Obj. | -74,300,000 | 107,000,000 | 4,705.10 | 1,062.44 | 10,877 | 39,301.10 | 3,360 | 12,850.90 | 2,574.11 | 115,155 | 307 |
| | Nodes | 10,000 | 7,477 | 10,000 | 10,000 | 10,000 | 10,000 | 629 | 551 | 10,000 | 986 | 9,091 |
| | Time | 15.67 | 4.95 | 35.04 | 1.11 | 3.42 | 3.08 | 0.32 | 0.76 | 9 | 30.06 | 1.49 |
| | Cl. Gap | 0.14 | 1 | 0.74 | 0.80 | 0.30 | 0.37 | 1 | 1 | 0.83 | 1 | 1 |
| NCB | Fin. | 5 | 0 | 0 | 4 | 5 | 5 | 0 | 0 | 4 | 0 | 0 |
| | Obj. | -74,100,000 | 107,000,000 | 4,722 | 1,093.29 | 11,041.20 | 39,371.80 | 3,360 | 12,850.90 | 2,487.50 | 115,155 | 307 |
| | Nodes | 10,000 | 1,527 | 213 | 10,000 | 10,000 | 10,000 | 579 | 56 | 10,000 | 617 | 4,605 |
| | Time | 95.59 | 4.76 | 4.53 | 5.16 | 28.23 | 23.72 | 2.65 | 0.58 | 152.62 | 49.18 | 2.85 |
| | Cl. Gap | 0.63 | 1 | 1 | 0.91 | 0.42 | 0.43 | 1 | 1 | 0.77 | 1 | 1 |
| FSB | Fin. | 5 | 0 | 0 | 5 | 5 | 5 | 0 | 0 | 4 | 0 | 0 |
| | Obj. | -74,100,000 | 107,000,000 | 4,722 | 1,088.06 | 11,041.80 | 39,410.90 | 3,360 | 12,850.90 | 2,537.50 | 115,000 | 307 |
| | Nodes | 10,000 | 1,502 | 238 | 10,000 | 10,000 | 10,000 | 371 | 55 | 10,000 | 709 | 3,333 |
| | Time | 336 | 7.27 | 47.67 | 8.37 | 44.55 | 35.20 | 2.78 | 1.20 | 378.26 | 139 | 1.77 |
| | Cl. Gap | 0.65 | 1 | 1 | 0.89 | 0.42 | 0.47 | 1 | 1 | 0.80 | 1 | 1 |
| RB | Fin. | 5 | 0 | 0 | 4 | 5 | 5 | 0 | 0 | 4 | 0 | 0 |
| | Obj. | -74,100,000 | 107,000,000 | 4,722 | 1,095.03 | 11,008 | 39,368 | 3,360 | 12,850.90 | 2,668.33 | 115,155 | 307 |
| | Nodes | 10,000 | 1,685 | 611 | 10,000 | 10,000 | 10,000 | 641 | 62 | 10,000 | 1,900 | 2,927 |
| | Time | 43.14 | 2.21 | 6.71 | 2.68 | 9.43 | 8.12 | 1.89 | 0.54 | 48.44 | 130.35 | 1.24 |
| | Cl. Gap | 0.56 | 1 | 1 | 0.91 | 0.40 | 0.43 | 1 | 1 | 0.90 | 1 | 1 |
| Learned | Fin. | 5 | 0 | 0 | 4 | 5 | 5 | 0 | 0 | 5 | 0 | 0 |
| | Obj. | -74,100,000 | 107,000,000 | 4,722 | 1,083.23 | 10,914.30 | 39,321 | 3,360 | 12,850.90 | 2,410 | 115,155 | 307 |
| | Nodes | 10,000 | 5,790 | 1,061 | 10,000 | 10,000 | 10,000 | 1,865 | 161 | 10,000 | 470 | 6,365 |
| | Time | 59.84 | 8.29 | 8.77 | 6.90 | 21.87 | 16.38 | 2.29 | 0.53 | 35.01 | 26.31 | 3.81 |
| | Cl. Gap | 0.41 | 1 | 1 | 0.87 | 0.33 | 0.38 | 1 | 1 | 0.71 | 1 | 1 |

Table A.8: Detailed results for the MIPLIB problems (2/4). Node limit = $10^5$ nodes. The row 'Cl. Gap' refers to the gap closed at the end of the optimization whether it achieved optimality or not. The row 'Fin. by all' indicates whether all methods were able to solve this problem to optimality. For each method, the 'Fin.' rows indicate the termination status of the optimization: 0 for optimality, 1 for unfeasibility, 2 for unboundedness, 3 for unfeasibility or unboundedness, 4 for another stopping criterion with a feasible solution found, and 5 for another stopping criterion with no feasible solution found.

| | Problem names | mod010 | mod011 | modglob | nw04 | opt1217 | p0033 | p0201 | p0282 | p0548 | p2756 | pk1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | LP Obj. | 6,532.08 | -62,100,000 | 20,400,000 | 16,310.70 | -20.02 | 2,520.57 | 6,875 | 176,868 | 429.68 | 2,698.95 | 0 |
| | True Obj. | 6,548 | -54,600,000 | 20,700,000 | 16,862 | -16 | 3,089 | 7,615 | 258,411 | 8,691 | 3,124 | 11 |
| | Fin. by all | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| Random | Fin. | 4 | 5 | 5 | 0 | 5 | 0 | 0 | 5 | 5 | 5 | 5 |
| | Obj. | 6,543 | -57,400,000 | 20,500,000 | 16,862 | -20.02 | 3,089 | 7,615 | 184,159 | 1,422.30 | 2,713.04 | 2.38 |
| | Nodes | 10,000 | 10,000 | 10,000 | 259 | 10,000 | 4,859 | 649 | 10,000 | 10,000 | 10,000 | 10,000 |
| | Time | 21.46 | 67.20 | 3.31 | 10.80 | 6.83 | 0.31 | 0.45 | 2.36 | 4.10 | 12.33 | 2.72 |
| | Cl. Gap | 0.69 | 0.62 | 0.33 | 1 | 0 | 1 | 1 | 0.09 | 0.12 | 0.03 | 0.22 |
| MIB | Fin. | 0 | 5 | 5 | 0 | 5 | 0 | 0 | 5 | 5 | 5 | 5 |
| | Obj. | 6,548 | -55,700,000 | 20,600,000 | 16,862 | -20.02 | 3,089 | 7,615 | 181,564 | 512.38 | 2,702.63 | 3.05 |
| | Nodes | 532 | 10,000 | 10,000 | 1,737 | 10,000 | 6,265 | 4,747 | 10,000 | 10,000 | 10,000 | 10,000 |
| | Time | 3.74 | 77.08 | 3.34 | 45.13 | 6.82 | 0.35 | 2.37 | 2.56 | 3.16 | 9.92 | 2.66 |
| | Cl. Gap | 1 | 0.84 | 0.57 | 1 | 0 | 1 | 1 | 0.06 | 0.01 | 0.01 | 0.28 |
| NCB | Fin. | 0 | 5 | 5 | 0 | 5 | 0 | 0 | 0 | 5 | 5 | 5 |
| | Obj. | 6,548 | -55,000,000 | 20,600,000 | 16,862 | -19.80 | 3,089 | 7,615 | 258,411 | 8,678.53 | 2,925.78 | 4.79 |
| | Nodes | 96 | 10,000 | 10,000 | 355 | 10,000 | 745 | 173 | 624 | 10,000 | 10,000 | 10,000 |
| | Time | 2.47 | 2,764.36 | 24.84 | 62.49 | 17 | 0.10 | 1.16 | 1.12 | 24.01 | 142.83 | 33.08 |
| | Cl. Gap | 1 | 0.94 | 0.66 | 1 | 0.06 | 1 | 1 | 1 | 1 | 0.53 | 0.44 |
| FSB | Fin. | 0 | 5 | 5 | 0 | 5 | 0 | 0 | 0 | 0 | 5 | 5 |
| | Obj. | 6,548 | -55,000,000 | 20,600,000 | 16,862 | -19.77 | 3,089 | 7,615 | 258,411 | 8,690 | 2,942.55 | 4.54 |
| | Nodes | 31 | 10,000 | 10,000 | 233 | 10,000 | 363 | 186 | 502 | 8,920 | 10,000 | 10,000 |
| | Time | 3.33 | 3,815.40 | 63.53 | 82.35 | 150.78 | 0.07 | 3.71 | 1.29 | 24.80 | 483.98 | 53.50 |
| | Cl. Gap | 1 | 0.94 | 0.70 | 1 | 0.06 | 1 | 1 | 1 | 1 | 0.57 | 0.41 |
| RB | Fin. | 0 | 5 | 5 | 0 | 5 | 0 | 0 | 0 | 5 | 5 | 5 |
| | Obj. | 6,548 | -55,300,000 | 20,600,000 | 16,862 | -19.92 | 3,089 | 7,615 | 258,411 | 4,881.29 | 2,715.33 | 4.07 |
| | Nodes | 83 | 10,000 | 10,000 | 1,201 | 10,000 | 1,150 | 359 | 767 | 10,000 | 10,000 | 10,000 |
| | Time | 3.26 | 991.97 | 9.07 | 116.97 | 16.27 | 0.15 | 1.51 | 0.97 | 20.97 | 64.81 | 9.59 |
| | Cl. Gap | 1 | 0.90 | 0.58 | 1 | 0.03 | 1 | 1 | 1 | 0.54 | 0.04 | 0.37 |
| Learned | Fin. | 0 | 5 | 5 | 0 | 5 | 0 | 0 | 0 | 5 | 5 | 5 |
| | Obj. | 6,548 | -55,900,000 | 20,600,000 | 16,862 | -19.98 | 3,090 | 7,615 | 258,411 | 8,640.78 | 2,721.47 | 2.96 |
| | Nodes | 123 | 10,000 | 10,000 | 235 | 10,000 | 291 | 612 | 8,614 | 10,000 | 10,000 | 10,000 |
| | Time | 1.58 | 127.51 | 22.36 | 10.80 | 39.96 | 0.07 | 1.78 | 7.55 | 23.56 | 110.99 | 17.70 |
| | Cl. Gap | 1 | 0.83 | 0.45 | 1 | 0.01 | 1 | 1 | 1 | 0.99 | 0.05 | 0.27 |

Table A.9: Detailed results for the MIPLIB problems (3/4). Node limit = $10^5$ nodes. The row 'Cl. Gap' refers to the gap closed at the end of the optimization whether it achieved optimality or not. The row 'Fin. by all' indicates whether all methods were able to solve this problem to optimality. For each method, the 'Fin.' rows indicate the termination status of the optimization: 0 for optimality, 1 for unfeasibility, 2 for unboundedness, 3 for unfeasibility or unboundedness, 4 for another stopping criterion with a feasible solution found, and 5 for another stopping criterion with no feasible solution found.

| | Problem names | pp08a | pp08aCUTS | qiu | rentacar | rgn | set1ch | stein27 | stein45 | tr12-30 | vpm1 | vpm2 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | LP Obj. | 2,748.35 | 5,480.61 | -931.64 | 28,800,000 | 48.80 | 32,007.70 | 13 | 22 | 14,210.40 | 15.42 | 9.89 |
| | True Obj. | 7,350 | 7,350 | -132.87 | 30,400,000 | 82.20 | 54,537.80 | 18 | 30 | 131,000 | 20 | 13.75 |
| | Fin. by all | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| Random | Fin. | 5 | 5 | 5 | 0 | 0 | 5 | 0 | 5 | 5 | 5 | 5 |
| | Obj. | 4,270.24 | 6,265.34 | -350.96 | 30,400,000 | 82.20 | 36,346.60 | 18 | 27.50 | 21,683.40 | 16.63 | 11.33 |
| | Nodes | 10,000 | 10,000 | 10,000 | 41 | 5,947 | 10,000 | 4,283 | 10,000 | 10,000 | 10,000 | 10,000 |
| | Time | 1.77 | 4.44 | 97.92 | 4.54 | 0.83 | 3.73 | 0.60 | 3.99 | 4.40 | 2.48 | 3.01 |
| | Cl. Gap | 0.33 | 0.42 | 0.73 | 1 | 1 | 0.19 | 1 | 0.69 | 0.06 | 0.27 | 0.37 |
| MIB | Fin. | 5 | 5 | 5 | 0 | 0 | 5 | 0 | 5 | 5 | 5 | 5 |
| | Obj. | 4,601.49 | 6,269.34 | -344.50 | 30,400,000 | 82.20 | 35,370.60 | 18 | 27.33 | 23,785.10 | 16.97 | 11.32 |
| | Nodes | 10,000 | 10,000 | 10,000 | 26 | 4,147 | 10,000 | 4,681 | 10,000 | 10,000 | 10,000 | 10,000 |
| | Time | 1.77 | 5.07 | 88.89 | 3.48 | 0.58 | 3.87 | 0.64 | 3.75 | 4.73 | 2.58 | 3.17 |
| | Cl. Gap | 0.40 | 0.42 | 0.74 | 1 | 1 | 0.15 | 1 | 0.67 | 0.08 | 0.34 | 0.37 |
| NCB | Fin. | 5 | 5 | 5 | 0 | 0 | 5 | 0 | 5 | 5 | 5 | 5 |
| | Obj. | 5,078.04 | 6,729.63 | -156.37 | 30,400,000 | 82.20 | 39,908.60 | 18 | 27.50 | 26,409.90 | 18.23 | 12.53 |
| | Nodes | 10,000 | 10,000 | 10,000 | 26 | 2,735 | 10,000 | 3,240 | 10,000 | 10,000 | 10,000 | 10,000 |
| | Time | 18.01 | 58.62 | 1,403.40 | 24.52 | 1.03 | 37.06 | 1.77 | 47.48 | 50.81 | 10.54 | 36.48 |
| | Cl. Gap | 0.51 | 0.67 | 0.97 | 1 | 1 | 0.35 | 1 | 0.69 | 0.10 | 0.61 | 0.69 |
| FSB | Fin. | 5 | 5 | 5 | 0 | 0 | 5 | 0 | 5 | 5 | 5 | 5 |
| | Obj. | 5,174.11 | 6,697.22 | -216.31 | 30,400,000 | 82.20 | 40,156.80 | 18 | 28.07 | 27,009.20 | 18.05 | 12.50 |
| | Nodes | 10,000 | 10,000 | 10,000 | 26 | 2,849 | 10,000 | 2,141 | 10,000 | 10,000 | 10,000 | 10,000 |
| | Time | 73.86 | 199.13 | 3,404.77 | 32.19 | 1.74 | 592 | 3.51 | 181.32 | 1,724.58 | 28.28 | 66.18 |
| | Cl. Gap | 0.53 | 0.65 | 0.90 | 1 | 1 | 0.36 | 1 | 0.76 | 0.11 | 0.57 | 0.67 |
| RB | Fin. | 5 | 5 | 4 | 0 | 0 | 5 | 0 | 5 | 5 | 5 | 5 |
| | Obj. | 4,659.35 | 6,567.33 | -143.30 | 30,400,000 | 82.20 | 40,319 | 18 | 27.50 | 26,487 | 17.78 | 11.89 |
| | Nodes | 10,000 | 10,000 | 10,000 | 21 | 2,701 | 10,000 | 3,980 | 10,000 | 10,000 | 10,000 | 10,000 |
| | Time | 6.23 | 15.49 | 436.22 | 18.23 | 0.77 | 9.60 | 1.41 | 21.27 | 13.61 | 5.67 | 19 |
| | Cl. Gap | 0.42 | 0.58 | 0.99 | 1 | 1 | 0.37 | 1 | 0.69 | 0.11 | 0.52 | 0.52 |
| Learned | Fin. | 5 | 5 | 5 | 0 | 0 | 5 | 0 | 5 | 5 | 5 | 5 |
| | Obj. | 4,730.36 | 6,490.81 | -279.63 | 30,400,000 | 82.20 | 39,600 | 18 | 27.50 | 24,065.90 | 16.95 | 11.35 |
| | Nodes | 10,000 | 10,000 | 10,000 | 36 | 3,401 | 10,000 | 4,140 | 10,000 | 10,000 | 10,000 | 10,000 |
| | Time | 42.04 | 51.12 | 152.68 | 4.32 | 1.20 | 120 | 3.13 | 31.06 | 366.50 | 16.07 | 31.62 |
| | Cl. Gap | 0.43 | 0.54 | 0.82 | 1 | 1 | 0.34 | 1 | 0.69 | 0.08 | 0.33 | 0.38 |

Table A.10: Detailed results for the MIPLIB problems (4/4). Node limit = $10^5$ nodes. The row 'Cl. Gap' refers to the gap closed at the end of the optimization whether it achieved optimality or not. The row 'Fin. by all' indicates whether all methods were able to solve this problem to optimality. For each method, the 'Fin.' rows indicate the termination status of the optimization: 0 for optimality, 1 for unfeasibility, 2 for unboundedness, 3 for unfeasibility or unboundedness, 4 for another stopping criterion with a feasible solution found, and 5 for another stopping criterion with no feasible solution found.

| | Problem names | 10teams | aflow30a | aflow40b | air03 | air04 | air05 | cap6000 | dcmulti | egout | fiber | fixnet6 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | LP Obj. | 917 | 983.17 | 1,005.66 | 338,864 | 55,535.40 | 25,877.60 | -2,451,540 | 183,976 | 149.59 | 156,083 | 1,200.88 |
| | True Obj. | 924 | 1,158 | 1,168 | 340,160 | 56,137 | 26,374 | -2,451,200 | 188,182 | 568.10 | 405,935 | 3,983 |
| | Fin. by all | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| Random | Fin. | 4 | 5 | 5 | 0 | 5 | 4 | 0 | 0 | 0 | 5 | 5 |
| | Obj. | 917 | 1,105.30 | 1,061.77 | 340,160 | 55,798.10 | 26,181.90 | -2,451,380 | 188,182 | 568.10 | 217,681 | 1,837.57 |
| | Nodes | 20,337 | 603,128 | 141,714 | 25 | 762 | 3,742 | 81,127 | 87,985 | 11,855 | 1,076,478 | 1,304,269 |
| | Time | 600.01 | 600.01 | 600.01 | 1.45 | 600.02 | 600.02 | 263.52 | 46.17 | 1.31 | 600.01 | 600.01 |
| | Cl. Gap | 0 | 0.70 | 0.35 | 1 | 0.44 | 0.61 | 0.47 | 1 | 1 | 0.25 | 0.23 |
| MIB | Fin. | 4 | 5 | 5 | 0 | 5 | 5 | 0 | 0 | 0 | 5 | 5 |
| | Obj. | 920.27 | 1,114.89 | 1,057.98 | 340,160 | 55,849.90 | 26,221 | -2,451,340 | 188,182 | 568.10 | 216,400 | 2,481.05 |
| | Nodes | 28,721 | 565,662 | 133,573 | 7 | 872 | 3,877 | 22,637 | 9,493 | 7,015 | 966,294 | 1,250,501 |
| | Time | 600.01 | 600.01 | 600.01 | 0.60 | 600.10 | 600.02 | 64.25 | 4.48 | 0.80 | 600.01 | 600.01 |
| | Cl. Gap | 0.47 | 0.75 | 0.32 | 1 | 0.52 | 0.69 | 0.59 | 1 | 1 | 0.24 | 0.46 |
| NCB | Fin. | 0 | 5 | 4 | 0 | 5 | 5 | 0 | 0 | 0 | 0 | 5 |
| | Obj. | 924 | 1,103.67 | 1,141.45 | 340,160 | 55,787.30 | 26,285.90 | -2,451,340 | 188,182 | 568.10 | 405,935 | 2,766.69 |
| | Nodes | 1,453 | 16,178 | 52,727 | 3 | 12 | 104 | 17,214 | 1,065 | 5,713 | 88,143 | 69,931 |
| | Time | 456.95 | 600.01 | 600.01 | 2.04 | 600.06 | 600.01 | 261.12 | 3.04 | 1.99 | 275.17 | 600.01 |
| | Cl. Gap | 1 | 0.69 | 0.84 | 1 | 0.42 | 0.82 | 0.59 | 1 | 1 | 1 | 0.56 |
| FSB | Fin. | 5 | 4 | 5 | 0 | 5 | 5 | 0 | 0 | 0 | 4 | 5 |
| | Obj. | 923 | 1,135.33 | 1,095.06 | 340,160 | 55,632.60 | 26,139.10 | -2,451,340 | 188,182 | 568.10 | 380,621 | 2,593.05 |
| | Nodes | 102 | 27,814 | 6,023 | 3 | 6 | 23 | 15,732 | 927 | 4,725 | 24,604 | 25,867 |
| | Time | 600.10 | 600.01 | 600.01 | 2.76 | 600.18 | 600.16 | 254.21 | 4.87 | 2.23 | 600.01 | 600.01 |
| | Cl. Gap | 0.86 | 0.87 | 0.55 | 1 | 0.16 | 0.53 | 0.59 | 1 | 1 | 0.90 | 0.50 |
| RB | Fin. | 5 | 4 | 5 | 0 | 5 | 5 | 0 | 0 | 0 | 4 | 5 |
| | Obj. | 917 | 1,149.05 | 1,102.58 | 340,160 | 55,823.60 | 26,227.20 | -2,451,340 | 188,182 | 568.10 | 382,623 | 2,773.55 |
| | Nodes | 4,018 | 147,308 | 44,452 | 3 | 247 | 612 | 18,989 | 1,013 | 9,143 | 289,900 | 343,403 |
| | Time | 600.01 | 600.01 | 600.01 | 0.52 | 600.03 | 600.02 | 225.13 | 1.80 | 2.39 | 600.01 | 600.01 |
| | Cl. Gap | 0 | 0.95 | 0.60 | 1 | 0.48 | 0.70 | 0.59 | 1 | 1 | 0.91 | 0.57 |
| Learned | Fin. | 0 | 5 | 5 | 0 | 5 | 5 | 0 | 0 | 0 | 5 | 5 |
| | Obj. | 924 | 1,104.13 | 1,046.45 | 340,160 | 55,992.20 | 26,224.70 | -2,451,380 | 188,182 | 568.10 | 372,086 | 2,411.64 |
| | Nodes | 1,739 | 120,872 | 43,281 | 3 | 553 | 2,083 | 66,587 | 1,565 | 3,007 | 150,084 | 104,029 |
| | Time | 108.28 | 600.01 | 600.01 | 0.49 | 600.01 | 600.01 | 314.71 | 2.74 | 1.31 | 600.01 | 600.01 |
| | Cl. Gap | 1 | 0.69 | 0.25 | 1 | 0.76 | 0.70 | 0.47 | 1 | 1 | 0.86 | 0.44 |

Table A.11: Detailed results for the MIPLIB problems (1/4). Time limit = 600 seconds. The row 'Cl. Gap' refers to the gap closed at the end of the optimization whether it achieved optimality or not. The row 'Fin. by all' indicates whether all methods were able to solve this problem to optimality. For each method, the 'Fin.' rows indicate the termination status of the optimization: 0 for optimality, 1 for unfeasibility, 2 for unboundedness, 3 for unfeasibility or unboundedness, 4 for another stopping criterion with a feasible solution found, and 5 for another stopping criterion with no feasible solution found.

| Problem names | | harp2 | khb05250 | l152lav | lseu | mas74 | mas76 | misc03 | misc06 | misc07 | mitre | mod008 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| LP Obj. | | -74,325,200 | 95,919,500 | 4,656.36 | 834.68 | 10,482.80 | 38,893.90 | 1,910 | 12,841.70 | 1,415 | 114,741 | 290.93 |
| True Obj. | | -73,899,300 | 106,940,000 | 4,722 | 1,120 | 11,801.20 | 40,005.10 | 3,360 | 12,850.90 | 2,810 | 115,155 | 307 |
| Fin. by all | | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| Random | Fin. | 5 | 0 | 0 | 0 | 5 | 4 | 0 | 0 | 0 | 5 | 0 |
| | Obj. | -74,231,300 | 106,940,000 | 4,722 | 1,120 | 11,227.40 | 39,899.60 | 3,360 | 12,851.10 | 2,810 | 115,131 | 307 |
| | Nodes | 219,404 | 11,555 | 57,281 | 112,035 | 1,092,445 | 1,662,754 | 1,179 | 527 | 82,081 | 63,300 | 10,307 |
| | Time | 600.03 | 7.15 | 107.70 | 10.23 | 600.01 | 600.01 | 0.52 | 0.70 | 53.43 | 600.01 | 1.54 |
| | Cl. Gap | 0.22 | 1 | 1 | 1 | 0.56 | 0.91 | 1 | 1.02 | 1 | 0.94 | 1 |
| MIB | Fin. | 5 | 0 | 0 | 0 | 5 | 4 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | -74,236,200 | 106,940,000 | 4,722 | 1,120 | 11,245.90 | 39,888.20 | 3,360 | 12,850.90 | 2,810 | 115,155 | 307 |
| | Nodes | 260,188 | 7,477 | 42,037 | 52,413 | 1,046,064 | 1,230,909 | 629 | 551 | 24,005 | 986 | 9,091 |
| | Time | 600.03 | 4.96 | 96.91 | 6.10 | 600.01 | 600.01 | 0.33 | 0.77 | 15.88 | 30.04 | 1.51 |
| | Cl. Gap | 0.21 | 1 | 1 | 1 | 0.58 | 0.89 | 1 | 1 | 1 | 1 | 1 |
| NCB | Fin. | 5 | 0 | 0 | 0 | 5 | 4 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | -74,019,800 | 106,940,000 | 4,722 | 1,120 | 11,349.60 | 39,784.90 | 3,360 | 12,850.90 | 2,810 | 115,155 | 307 |
| | Nodes | 52,988 | 1,527 | 213 | 18,533 | 171,360 | 234,705 | 579 | 56 | 32,873 | 617 | 4,605 |
| | Time | 600.01 | 4.79 | 4.57 | 7.24 | 600.01 | 600.01 | 2.71 | 0.59 | 254.98 | 49.25 | 2.91 |
| | Cl. Gap | 0.72 | 1 | 1 | 1 | 0.66 | 0.80 | 1 | 1 | 1 | 1 | 1 |
| FSB | Fin. | 5 | 0 | 0 | 0 | 5 | 4 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | -74,040,700 | 106,940,000 | 4,722 | 1,120 | 11,310.80 | 39,786.60 | 3,360 | 12,850.90 | 2,810 | 115,155 | 307 |
| | Nodes | 17,259 | 1,502 | 238 | 24,957 | 132,769 | 178,936 | 371 | 55 | 25,551 | 709 | 3,333 |
| | Time | 600.02 | 7.40 | 48.17 | 16.65 | 600.01 | 600.01 | 2.83 | 1.22 | 591.65 | 139.18 | 1.81 |
| | Cl. Gap | 0.67 | 1 | 1 | 1 | 0.63 | 0.80 | 1 | 1 | 1 | 1 | 1 |
| RB | Fin. | 5 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | -74,034,800 | 106,940,000 | 4,722 | 1,120 | 11,481.90 | 40,005.10 | 3,360 | 12,850.90 | 2,810 | 115,155 | 307 |
| | Nodes | 115,774 | 1,685 | 611 | 16,933 | 547,986 | 621,181 | 641 | 62 | 19,179 | 1,900 | 2,927 |
| | Time | 600.03 | 2.24 | 6.76 | 3.99 | 600.01 | 496.77 | 1.92 | 0.53 | 59.87 | 130.55 | 1.25 |
| | Cl. Gap | 0.68 | 1 | 1 | 1 | 0.76 | 1 | 1 | 1 | 1 | 1 | 1 |
| Learned | Fin. | 5 | 0 | 0 | 0 | 5 | 4 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | -74,115,700 | 106,940,000 | 4,722 | 1,120 | 11,230.50 | 39,753 | 3,360 | 12,850.90 | 2,810 | 115,155 | 307 |
| | Nodes | 90,166 | 5,790 | 1,061 | 23,707 | 301,782 | 403,888 | 1,865 | 161 | 97,197 | 470 | 6,365 |
| | Time | 600.01 | 8.24 | 8.66 | 11.39 | 600.01 | 600.01 | 2.24 | 0.53 | 174.27 | 26.21 | 3.59 |
| | Cl. Gap | 0.49 | 1 | 1 | 1 | 0.57 | 0.77 | 1 | 1 | 1 | 1 | 1 |

Table A.12: Detailed results for the MIPLIB problems (2/4). Time limit = 600 seconds. The row 'Cl. Gap' refers to the gap closed at the end of the optimization whether it achieved optimality or not. The row 'Fin. by all' indicates whether all methods were able to solve this problem to optimality. For each method, the 'Fin.' rows indicate the termination status of the optimization: 0 for optimality, 1 for unfeasibility, 2 for unboundedness, 3 for unfeasibility or unboundedness, 4 for another stopping criterion with a feasible solution found, and 5 for another stopping criterion with no feasible solution found.

| | Problem names | mod010 | mod011 | modglob | nw04 | opt1217 | p0033 | p0201 | p0282 | p0548 | p2756 | pk1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | LP Obj. | 6,532.08 | -62,122,000 | 20,430,900 | 16,310.70 | -20.02 | 2,520.57 | 6,875 | 176,868 | 429.68 | 2,698.95 | 0 |
| | True Obj. | 6,548 | -54,558,500 | 20,740,500 | 16,862 | -16 | 3,089 | 7,615 | 258,411 | 8,691 | 3,124 | 11 |
| | Fin. by all | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| Random | Fin. | 0 | 5 | 5 | 0 | 5 | 0 | 0 | 5 | 5 | 5 | 4 |
| | Obj. | 6,548 | -55,956,000 | 20,607,600 | 16,862 | -20.02 | 3,089 | 7,615 | 192,124 | 2,085.24 | 2,733.15 | 10.92 |
| | Nodes | 27,055 | 82,469 | 1,293,324 | 259 | 687,665 | 4,859 | 649 | 1,712,728 | 1,285,332 | 529,728 | 1,346,758 |
| | Time | 47.98 | 600.01 | 600.01 | 10.81 | 600.01 | 0.32 | 0.45 | 600.01 | 600.01 | 600.01 | 600.01 |
| | Cl. Gap | 1 | 0.82 | 0.57 | 1 | 0 | 1 | 1 | 0.19 | 0.20 | 0.08 | 0.99 |
| MIB | Fin. | 0 | 0 | 5 | 0 | 5 | 0 | 0 | 5 | 5 | 5 | 0 |
| | Obj. | 6,548 | -54,558,500 | 20,706,800 | 16,862 | -20.02 | 3,089 | 7,615 | 184,470 | 524.95 | 2,705.50 | 11 |
| | Nodes | 532 | 49,533 | 989,483 | 1,737 | 670,264 | 6,265 | 4,747 | 1,504,144 | 1,121,079 | 634,588 | 927,663 |
| | Time | 3.76 | 419.22 | 600.01 | 44.77 | 600.01 | 0.37 | 2.43 | 600.01 | 600.01 | 600.01 | 382.10 |
| | Cl. Gap | 1 | 1 | 0.89 | 1 | 0 | 1 | 1 | 0.09 | 0.01 | 0.02 | 1 |
| NCB | Fin. | 0 | 5 | 5 | 0 | 5 | 0 | 0 | 0 | 0 | 5 | 5 |
| | Obj. | 6,548 | -56,157,900 | 20,682,800 | 16,862 | -19.65 | 3,089 | 7,615 | 258,411 | 8,691 | 2,984.96 | 9.95 |
| | Nodes | 96 | 2,518 | 176,901 | 355 | 260,272 | 745 | 173 | 624 | 18,782 | 34,910 | 151,768 |
| | Time | 2.50 | 600.05 | 600.01 | 62.35 | 600.01 | 0.12 | 1.17 | 1.14 | 42.31 | 600.01 | 600.01 |
| | Cl. Gap | 1 | 0.79 | 0.81 | 1 | 0.09 | 1 | 1 | 1 | 1 | 0.67 | 0.90 |
| FSB | Fin. | 0 | 5 | 5 | 0 | 5 | 0 | 0 | 0 | 0 | 5 | 5 |
| | Obj. | 6,548 | -56,661,700 | 20,686,000 | 16,862 | -19.66 | 3,089 | 7,615 | 258,411 | 8,691 | 2,950.75 | 9.23 |
| | Nodes | 31 | 1,260 | 101,090 | 233 | 38,913 | 363 | 186 | 502 | 8,920 | 12,364 | 125,603 |
| | Time | 3.36 | 600.03 | 600.01 | 82.23 | 600.01 | 0.07 | 3.78 | 1.31 | 25.15 | 600.01 | 600.01 |
| | Cl. Gap | 1 | 0.72 | 0.82 | 1 | 0.09 | 1 | 1 | 1 | 1 | 0.59 | 0.84 |
| RB | Fin. | 0 | 5 | 5 | 0 | 5 | 0 | 0 | 0 | 0 | 5 | 0 |
| | Obj. | 6,548 | -55,719,100 | 20,686,900 | 16,862 | -19.76 | 3,089 | 7,615 | 258,411 | 8,691 | 2,763.77 | 11 |
| | Nodes | 83 | 6,180 | 493,119 | 1,201 | 389,380 | 1,145 | 359 | 767 | 145,693 | 92,008 | 366,151 |
| | Time | 3.27 | 600.06 | 600.01 | 117.90 | 600.01 | 0.15 | 1.52 | 0.97 | 174.10 | 600.01 | 358.78 |
| | Cl. Gap | 1 | 0.85 | 0.83 | 1 | 0.07 | 1 | 1 | 1 | 1 | 0.15 | 1 |
| Learned | Fin. | 0 | 0 | 5 | 0 | 5 | 0 | 0 | 0 | 0 | 5 | 5 |
| | Obj. | 6,548 | -54,558,500 | 20,600,400 | 16,862 | -19.93 | 3,089 | 7,615 | 258,411 | 8,691 | 2,728.89 | 8.60 |
| | Nodes | 123 | 46,907 | 259,394 | 235 | 135,183 | 291 | 612 | 8,614 | 86,603 | 53,100 | 362,383 |
| | Time | 1.59 | 564.95 | 600.01 | 10.68 | 600.01 | 0.07 | 1.73 | 7.56 | 110.72 | 600.01 | 600.01 |
| | Cl. Gap | 1 | 1 | 0.55 | 1 | 0.02 | 1 | 1 | 1 | 1 | 0.07 | 0.78 |

Table A.13: Detailed results for the MIPLIB problems (3/4). Time limit = 600 seconds. The row 'Cl. Gap' refers to the gap closed at the end of the optimization whether it achieved optimality or not. The row 'Fin. by all' indicates whether all methods were able to solve this problem to optimality. For each method, the 'Fin.' rows indicate the termination status of the optimization: 0 for optimality, 1 for unfeasibility, 2 for unboundedness, 3 for unfeasibility or unboundedness, 4 for another stopping criterion with a feasible solution found, and 5 for another stopping criterion with no feasible solution found.

| Problem names | | pp08a | pp08aCUTS | qiu | rentacar | rgn | set1ch | stein27 | stein45 | tr12-30 | vpm1 | vpm2 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| LP Obj. | | 2,748.35 | 5,480.61 | -931.64 | 28,806,100 | 48.80 | 32,007.70 | 13 | 22 | 14,210.40 | 15.42 | 9.89 |
| True Obj. | | 7,350 | 7,350 | -132.87 | 30,356,800 | 82.20 | 54,537.80 | 18 | 30 | 130,596 | 20 | 13.75 |
| Fin. by all | | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| Random | Fin. | 5 | 5 | 4 | 0 | 0 | 5 | 0 | 0 | 5 | 5 | 5 |
| | Obj. | 5,198.76 | 6,756.75 | -226.26 | 30,356,800 | 82.20 | 37,522.60 | 18 | 30 | 24,059.40 | 18.25 | 12.60 |
| | Nodes | 1,725,290 | 1,103,681 | 74,246 | 41 | 5,947 | 1,351,176 | 4,283 | 63,115 | 1,251,955 | 1,608,231 | 1,455,001 |
| | Time | 600.01 | 600.01 | 600.01 | 4.54 | 0.85 | 600.01 | 0.60 | 20.25 | 600.01 | 600.01 | 600.05 |
| | Cl. Gap | 0.53 | 0.68 | 0.88 | 1 | 1 | 0.24 | 1 | 1 | 0.08 | 0.62 | 0.70 |
| MIB | Fin. | 5 | 5 | 4 | 0 | 0 | 5 | 0 | 0 | 5 | 5 | 5 |
| | Obj. | 5,625.42 | 6,817.68 | -211.52 | 30,356,800 | 82.20 | 35,814.40 | 18 | 30 | 26,437.20 | 18.55 | 12.60 |
| | Nodes | 1,525,690 | 1,001,129 | 80,056 | 26 | 4,147 | 1,174,846 | 4,681 | 86,199 | 1,049,709 | 1,563,746 | 1,331,407 |
| | Time | 600.01 | 600.01 | 600.01 | 3.49 | 0.60 | 600.01 | 0.66 | 26.17 | 600.02 | 600.01 | 600 |
| | Cl. Gap | 0.63 | 0.72 | 0.90 | 1 | 1 | 0.17 | 1 | 1 | 0.11 | 0.68 | 0.70 |
| NCB | Fin. | 5 | 5 | 5 | 0 | 0 | 5 | 0 | 0 | 5 | 5 | 5 |
| | Obj. | 5,800.71 | 6,988.93 | -229.11 | 30,356,800 | 82.20 | 41,291.10 | 18 | 30 | 28,983.30 | 19.25 | 13.16 |
| | Nodes | 223,945 | 86,240 | 4,592 | 26 | 2,735 | 122,404 | 3,239 | 44,219 | 120,874 | 501,139 | 142,007 |
| | Time | 600.01 | 600.01 | 600.01 | 24.52 | 1.06 | 600.01 | 1.80 | 149.92 | 600.01 | 600.01 | 600 |
| | Cl. Gap | 0.66 | 0.81 | 0.88 | 1 | 1 | 0.41 | 1 | 1 | 0.13 | 0.84 | 0.85 |
| FSB | Fin. | 5 | 5 | 5 | 0 | 0 | 5 | 0 | 0 | 5 | 5 | 5 |
| | Obj. | 5,666.07 | 6,864.18 | -399.43 | 30,356,800 | 82.20 | 40,157.30 | 18 | 30 | 25,847.50 | 19 | 13.06 |
| | Nodes | 87,459 | 32,806 | 1,360 | 26 | 2,849 | 10,008 | 2,141 | 24,835 | 3,496 | 282,230 | 105,079 |
| | Time | 600.01 | 600.01 | 600.01 | 32.21 | 1.80 | 600.02 | 3.61 | 285.03 | 600.02 | 600.01 | 600.01 |
| | Cl. Gap | 0.63 | 0.74 | 0.67 | 1 | 1 | 0.36 | 1 | 1 | 0.10 | 0.78 | 0.82 |
| RB | Fin. | 5 | 5 | 0 | 0 | 0 | 5 | 0 | 0 | 5 | 0 | 5 |
| | Obj. | 5,649.88 | 7,082.74 | -132.87 | 30,356,800 | 82.20 | 42,397.10 | 18 | 30 | 30,376.10 | 20 | 12.79 |
| | Nodes | 643,995 | 305,396 | 14,923 | 21 | 2,701 | 492,097 | 3,975 | 50,335 | 418,805 | 649,531 | 299,538 |
| | Time | 600.01 | 600.01 | 477.28 | 18.20 | 0.78 | 600.01 | 1.44 | 70.49 | 600.01 | 446.18 | 600.01 |
| | Cl. Gap | 0.63 | 0.86 | 1 | 1 | 1 | 0.46 | 1 | 1 | 0.14 | 1 | 0.75 |
| Learned | Fin. | 5 | 5 | 5 | 0 | 0 | 5 | 0 | 0 | 5 | 5 | 5 |
| | Obj. | 5,318.51 | 6,828.05 | -179.74 | 30,356,800 | 82.20 | 40,650.20 | 18 | 30 | 24,425.10 | 18.18 | 12.24 |
| | Nodes | 166,483 | 129,426 | 47,914 | 36 | 3,401 | 52,779 | 4,141 | 50,013 | 16,200 | 448,168 | 219,927 |
| | Time | 600.01 | 600.01 | 600.01 | 4.32 | 1.20 | 600.01 | 3.12 | 97.31 | 600.03 | 600.01 | 600 |
| | Cl. Gap | 0.56 | 0.72 | 0.94 | 1 | 1 | 0.38 | 1 | 1 | 0.09 | 0.60 | 0.61 |

Table A.14: Detailed results for the MIPLIB problems (4/4). Time limit = 600 seconds. The row 'Cl. Gap' refers to the gap closed at the end of the optimization whether it achieved optimality or not. The row 'Fin. by all' indicates whether all methods were able to solve this problem to optimality. For each method, the 'Fin.' rows indicate the termination status of the optimization: 0 for optimality, 1 for unfeasibility, 2 for unboundedness, 3 for unfeasibility or unboundedness, 4 for another stopping criterion with a feasible solution found, and 5 for another stopping criterion with no feasible solution found.

| | Problem names | aflow30a | air03 | air04 | air05 | cap6000 | dcmulti | egout | khb05250 | l152lav | lseu |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | LP Obj. | 983.17 | 338,864 | 55,535.40 | 25,877.60 | -2,451,540 | 183,976 | 149.59 | 95,919,500 | 4,656.36 | 834.68 |
| | True Obj. | 1,158 | 340,160 | 56,137 | 26,374 | -2,451,200 | 188,182 | 568.10 | 106,940,000 | 4,722 | 1,120 |
| | Fin. by all | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Random | Fin. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | 1,158 | 340,160 | 56,137 | 26,374 | -2,450,000 | 188,182 | 568.10 | 107,000,000 | 4,722 | 1,120 |
| | Nodes | 15,800,801 | 25 | 130,471 | 88,241 | 81,127 | 87,985 | 11,855 | 11,555 | 57,281 | 112,035 |
| | Time | 19,276.30 | 1.44 | 9,228.73 | 4,613 | 259.32 | 45.45 | 1.27 | 6.85 | 106.19 | 9.82 |
| | Cl. Gap | 1 | 1 | 1 | 1 | 4.53 | 1 | 1 | 1.01 | 1 | 1 |
| MIB | Fin. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | 1,158 | 340,160 | 56,137 | 26,374 | -2,451,340 | 188,182 | 568.10 | 106,940,000 | 4,722 | 1,120 |
| | Nodes | 7,691,239 | 7 | 105,821 | 66,991 | 22,637 | 9,493 | 7,015 | 7,477 | 42,037 | 52,413 |
| | Time | 9,109.39 | 0.59 | 14,008.90 | 5,642.01 | 63.34 | 4.44 | 0.77 | 4.77 | 95.61 | 5.78 |
| | Cl. Gap | 1 | 1 | 1 | 1 | 0.59 | 1 | 1 | 1 | 1 | 1 |
| NCB | Fin. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | 1,158 | 340,160 | 56,137 | 26,374 | -2,451,340 | 188,182 | 568.10 | 106,940,000 | 4,722 | 1,120 |
| | Nodes | 137,895 | 3 | 131 | 215 | 17,214 | 1,065 | 5,713 | 1,527 | 213 | 18,533 |
| | Time | 1,296.89 | 2.02 | 1,133.76 | 768.24 | 256.47 | 2.98 | 1.93 | 4.71 | 4.51 | 6.93 |
| | Cl. Gap | 1 | 1 | 1 | 1 | 0.59 | 1 | 1 | 1 | 1 | 1 |
| FSB | Fin. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | 1,158 | 340,160 | 56,137 | 26,374 | -2,451,340 | 188,182 | 568.10 | 106,940,000 | 4,722 | 1,120 |
| | Nodes | 118,057 | 3 | 111 | 177 | 15,732 | 927 | 4,725 | 1,502 | 238 | 24,957 |
| | Time | 2,073.98 | 2.73 | 3,224.13 | 3,812.73 | 250.59 | 4.81 | 2.13 | 7.22 | 47.27 | 15.98 |
| | Cl. Gap | 1 | 1 | 1 | 1 | 0.59 | 1 | 1 | 1 | 1 | 1 |
| RB | Fin. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | 1,158 | 340,160 | 56,137 | 26,374 | -2,451,340 | 188,182 | 568.10 | 106,940,000 | 4,722 | 1,120 |
| | Nodes | 230,038 | 3 | 3,601 | 1,489 | 18,989 | 1,013 | 9,143 | 1,685 | 611 | 16,933 |
| | Time | 898.36 | 0.52 | 2,760.38 | 1,449.85 | 222.89 | 1.77 | 2.31 | 2.17 | 6.66 | 3.83 |
| | Cl. Gap | 1 | 1 | 1 | 1 | 0.59 | 1 | 1 | 1 | 1 | 1 |
| Learned | Fin. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | 1,158 | 340,160 | 56,137 | 26,374 | -2,451,380 | 188,182 | 568.10 | 106,940,000 | 4,722 | 1,120 |
| | Nodes | 2,825,981 | 3 | 3,833 | 65,305 | 66,587 | 1,565 | 3,007 | 5,790 | 1,061 | 23,707 |
| | Time | 8,755.66 | 0.48 | 1,613.37 | 5,141.27 | 310.41 | 2.30 | 1.07 | 7.13 | 7.88 | 8.99 |
| | Cl. Gap | 1 | 1 | 1 | 1 | 0.47 | 1 | 1 | 1 | 1 | 1 |

Table A.15: Detailed results for the updated list of MIPLIB problems (1/3). Time limit = none. The row 'Cl. Gap' refers to the gap closed at the end of the optimization whether it achieved optimality or not. The row 'Fin. by all' indicates whether all methods were able to solve this problem to optimality. For each method, the 'Fin.' rows indicate the termination status of the optimization: 0 for optimality, 1 for unfeasibility, 2 for unboundedness, 3 for unfeasibility or unboundedness, 4 for another stopping criterion with a feasible solution found, and 5 for another stopping criterion with no feasible solution found.

| | Problem names | mas76 | misc03 | misc06 | misc07 | mitre | mod008 | mod010 | mod011 | nw04 | p0033 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | LP Obj. | 38,893.90 | 1,910 | 12,841.70 | 1,415 | 114,741 | 290.93 | 6,532.08 | -62,122,000 | 16,310.70 | 2,520.57 |
| | True Obj. | 40,005.10 | 3,360 | 12,850.90 | 2,810 | 115,155 | 307 | 6,548 | -54,558,500 | 16,862 | 3,089 |
| | Fin. by all | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Random | Fin. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | 40,005.10 | 3,360 | 12,851.10 | 2,810 | 115,155 | 307 | 6,548 | -54,600,000 | 16,862 | 3,089 |
| | Nodes | 2,306,195 | 1,179 | 527 | 82,081 | 288,265 | 10,307 | 27,055 | 517,237 | 259 | 4,859 |
| | Time | 693.85 | 0.50 | 0.70 | 52.66 | 1,606.37 | 1.49 | 47.67 | 4,072.82 | 10.63 | 0.30 |
| | Cl. Gap | 1 | 1 | 1.02 | 1 | 1 | 1 | 1 | 0.99 | 1 | 1 |
| MIB | Fin. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | 40,005.10 | 3,360 | 12,850.90 | 2,810 | 115,155 | 307 | 6,548 | -54,558,500 | 16,862 | 3,089 |
| | Nodes | 2,097,725 | 629 | 551 | 24,005 | 986 | 9,091 | 532 | 49,533 | 1,737 | 6,265 |
| | Time | 760.55 | 0.32 | 0.77 | 15.67 | 29.70 | 1.45 | 3.70 | 410.90 | 44.66 | 0.35 |
| | Cl. Gap | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| NCB | Fin. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | 40,005.10 | 3,360 | 12,850.90 | 2,810 | 115,155 | 307 | 6,548 | -54,558,500 | 16,862 | 3,089 |
| | Nodes | 783,139 | 579 | 56 | 32,873 | 617 | 4,605 | 96 | 19,617 | 355 | 745 |
| | Time | 1,491.35 | 2.65 | 0.58 | 250.44 | 48.30 | 2.82 | 2.47 | 5,422.18 | 61.96 | 0.10 |
| | Cl. Gap | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| FSB | Fin. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | 40,005.10 | 3,360 | 12,850.90 | 2,810 | 115,155 | 307 | 6,548 | -54,558,500 | 16,862 | 3,089 |
| | Nodes | 597,109 | 371 | 55 | 25,551 | 709 | 3,333 | 31 | 18,283 | 233 | 363 |
| | Time | 1,466.68 | 2.78 | 1.19 | 581.55 | 137.07 | 1.75 | 3.28 | 6,409.24 | 81.40 | 0.06 |
| | Cl. Gap | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| RB | Fin. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | 40,005.10 | 3,360 | 12,850.90 | 2,810 | 115,155 | 307 | 6,548 | -54,558,500 | 16,862 | 3,089 |
| | Nodes | 621,181 | 641 | 62 | 19,179 | 1,900 | 2,927 | 83 | 25,873 | 1,201 | 1,145 |
| | Time | 477.96 | 1.89 | 0.53 | 59.32 | 128.88 | 1.22 | 3.23 | 2,715.15 | 116.54 | 0.13 |
| | Cl. Gap | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Learned | Fin. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | 40,005.10 | 3,360 | 12,850.90 | 2,810 | 115,155 | 307 | 6,548 | -54,558,500 | 16,862 | 3,089 |
| | Nodes | 1,722,679 | 1,865 | 161 | 97,197 | 470 | 6,365 | 123 | 46,907 | 235 | 291 |
| | Time | 2,232.96 | 1.90 | 0.50 | 153.05 | 24.88 | 2.94 | 1.51 | 544.07 | 10.78 | 0.05 |
| | Cl. Gap | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Table A.16: Detailed results for the updated list of MIPLIB problems (2/3). Time limit = none. The row 'Cl. Gap' refers to the gap closed at the end of the optimization whether it achieved optimality or not. The row 'Fin. by all' indicates whether all methods were able to solve this problem to optimality. For each method, the 'Fin.' rows indicate the termination status of the optimization: 0 for optimality, 1 for unfeasibility, 2 for unboundedness, 3 for unfeasibility or unboundedness, 4 for another stopping criterion with a feasible solution found, and 5 for another stopping criterion with no feasible solution found.

| | Problem names | p0201 | pk1 | pp08aCUTS | qiu | rentacar | rgn | stein27 | stein45 | vpm1 | vpm2 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | LP Obj. | 6,875 | 0 | 5,480.61 | -931.64 | 28,806,100 | 48.80 | 13 | 22 | 15.42 | 9.89 |
| | True Obj. | 7,615 | 11 | 7,350 | -132.87 | 30,356,800 | 82.20 | 18 | 30 | 20 | 13.75 |
| | Fin. by all | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Random | Fin. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | 7,615 | 11 | 7,350 | -132.87 | 30,400,000 | 82.20 | 18 | 30 | 20 | 13.75 |
| | Nodes | 649 | 1,400,487 | 172,918,239 | 411,879 | 41 | 5,947 | 4,283 | 63,115 | 14,838,614 | 25,017,647 |
| | Time | 0.44 | 585.92 | 793,141 | 2,523.43 | 4.48 | 0.81 | 0.58 | 19.91 | 17,681.30 | 27,319.80 |
| | Cl. Gap | 1 | 1 | 1 | 1 | 1.03 | 1 | 1 | 1 | 1 | 1 |
| MIB | Fin. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | 7,615 | 11 | 7,350 | -132.87 | 30,356,800 | 82.20 | 18 | 30 | 20 | 13.75 |
| | Nodes | 4,747 | 927,663 | 63,144,303 | 337,813 | 26 | 4,147 | 4,681 | 86,199 | 8,953,981 | 20,513,193 |
| | Time | 2.36 | 365.08 | 161,744 | 2,058.33 | 3.44 | 0.56 | 0.63 | 25.79 | 7,629.32 | 19,579.40 |
| | Cl. Gap | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| NCB | Fin. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | 7,615 | 11 | 7,350 | -132.87 | 30,356,800 | 82.20 | 18 | 30 | 20 | 13.75 |
| | Nodes | 173 | 257,925 | 1,744,935 | 16,865 | 26 | 2,735 | 3,239 | 44,219 | 621,984 | 640,044 |
| | Time | 1.15 | 957.74 | 17,663.10 | 1,610.02 | 24.15 | 1.02 | 1.76 | 146.05 | 735.40 | 2,188.64 |
| | Cl. Gap | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| FSB | Fin. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | 7,615 | 11 | 7,350 | -132.87 | 30,356,800 | 82.20 | 18 | 30 | 20 | 13.75 |
| | Nodes | 186 | 306,805 | 1,561,719 | 49,693 | 26 | 2,849 | 2,141 | 24,835 | 465,770 | 644,925 |
| | Time | 3.69 | 1,041.17 | 17,645.50 | 7,136.80 | 31.84 | 1.73 | 3.49 | 276.55 | 954.07 | 2,692.21 |
| | Cl. Gap | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| RB | Fin. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | 7,615 | 11 | 7,350 | -132.87 | 30,356,800 | 82.20 | 18 | 30 | 20 | 13.75 |
| | Nodes | 359 | 366,151 | 2,668,423 | 14,923 | 21 | 2,701 | 3,975 | 50,335 | 649,531 | 4,837,431 |
| | Time | 1.49 | 344.92 | 4,815.89 | 476.17 | 17.96 | 0.75 | 1.39 | 69.08 | 431.22 | 11,571.10 |
| | Cl. Gap | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Learned | Fin. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | 7,615 | 11 | 7,350 | -132.87 | 30,356,800 | 82.20 | 18 | 30 | 20 | 13.75 |
| | Nodes | 612 | 1,545,753 | 7,272,423 | 116,593 | 36 | 3,401 | 4,141 | 50,013 | 6,606,414 | 10,639,147 |
| | Time | 1.43 | 2,187.65 | 22,281.40 | 1,074.32 | 4.26 | 1.01 | 2.42 | 76.64 | 10,218.50 | 36,031.50 |
| | Cl. Gap | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Table A.17: Detailed results for the updated list of MIPLIB problems (3/3). Time limit = none. The row 'Cl. Gap' refers to the gap closed at the end of the optimization whether it achieved optimality or not. The row 'Fin. by all' indicates whether all methods were able to solve this problem to optimality. For each method, the 'Fin.' rows indicate the termination status of the optimization: 0 for optimality, 1 for unfeasibility, 2 for unboundedness, 3 for unfeasibility or unboundedness, 4 for another stopping criterion with a feasible solution found, and 5 for another stopping criterion with no feasible solution found.

| | Problem names | aflow30a | air03 | air04 | air05 | cap6000 | dcmulti | egout | khb05250 | l152lav | lseu |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | LP Obj. | 983.17 | 338,864 | 55,535.40 | 25,877.60 | -2,451,540 | 183,976 | 149.59 | 95,919,500 | 4,656.36 | 834.68 |
| | True Obj. | 1,158 | 340,160 | 56,137 | 26,374 | -2,451,200 | 188,182 | 568.10 | 106,940,000 | 4,722 | 1,120 |
| | Fin. by all | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Random | Fin. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | 1,158 | 340,160 | 56,137 | 26,374 | -2,450,000 | 188,188 | 568.10 | 107,000,000 | 4,722 | 1,120 |
| | Nodes | 58,369 | 0 | 50,847 | 29,149 | 180 | 641 | 3 | 5 | 45,433 | 185 |
| | Time | 256.88 | 0.59 | 3,956.43 | 1,839.59 | 4.06 | 1.58 | 0.01 | 0.41 | 324.38 | 0.13 |
| | Cl. Gap | 1 | 1 | 1 | 1 | 4.53 | 1 | 1 | 1.01 | 1 | 1 |
| MIB | Fin. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | 1,158 | 340,160 | 56,137 | 26,374 | -2,451,350 | 188,188 | 568.10 | 106,940,000 | 4,722 | 1,120 |
| | Nodes | 35,205 | 0 | 46,139 | 38,793 | 60 | 125 | 3 | 3 | 29,545 | 97 |
| | Time | 203.06 | 0.60 | 4,403.65 | 2,607.30 | 1.30 | 0.71 | 0.01 | 0.42 | 227.48 | 0.11 |
| | Cl. Gap | 1 | 1 | 1 | 1 | 0.56 | 1 | 1 | 1 | 1 | 1 |
| NCB | Fin. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | 1,158 | 340,160 | 56,137 | 26,374 | -2,451,350 | 188,182 | 568.10 | 106,940,000 | 4,722 | 1,120 |
| | Nodes | 115,987 | 0 | 365 | 487 | 20 | 8,135 | 7 | 9 | 187 | 223 |
| | Time | 2,043.61 | 0.59 | 3,488.17 | 1,844.01 | 0.87 | 33.28 | 0.02 | 0.46 | 4.74 | 0.22 |
| | Cl. Gap | 1 | 1 | 1 | 1 | 0.56 | 1 | 1 | 1 | 1 | 1 |
| FSB | Fin. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | 1,158 | 340,160 | 56,137 | 26,374 | -2,451,180 | 188,186 | 568.10 | 106,940,000 | 4,722 | 1,120 |
| | Nodes | 31,323 | 0 | 231 | 211 | 49 | 40 | 7 | 5 | 169 | 133 |
| | Time | 1,158.81 | 0.60 | 10,229.90 | 3,941.68 | 1.53 | 0.77 | 0.02 | 0.42 | 18.73 | 0.27 |
| | Cl. Gap | 1 | 1 | 1 | 1 | 1.06 | 1 | 1 | 1 | 1 | 1 |
| RB | Fin. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | 1,158 | 340,160 | 56,137 | 26,374 | -2,451,350 | 188,187 | 568.10 | 106,940,000 | 4,722 | 1,120 |
| | Nodes | 84,014 | 0 | 336,711 | 78,761 | 20 | 429 | 7 | 7 | 459 | 99 |
| | Time | 590.75 | 0.59 | 57,934.10 | 17,733.40 | 0.93 | 1.67 | 0.02 | 0.45 | 6.45 | 0.15 |
| | Cl. Gap | 1 | 1 | 1 | 1 | 0.56 | 1 | 1 | 1 | 1 | 1 |
| Learned | Fin. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | 1,158 | 340,160 | 56,137 | 26,374 | -2,451,350 | 188,195 | 568.10 | 106,940,000 | 4,722 | 1,120 |
| | Nodes | 42,389 | 0 | 553 | 5,785 | 20 | 111 | 5 | 7 | 799 | 75 |
| | Time | 296.91 | 0.60 | 237.80 | 969.80 | 0.74 | 0.87 | 0.02 | 0.44 | 7.27 | 0.17 |
| | Closed gap | 1 | 1 | 1 | 1 | 0.56 | 1 | 1 | 1 | 1 | 1 |

Table A.18: Detailed results for the updated list of MIPLIB problems (1/3). Time limit = none and CPLEX's cuts and heuristics applied. The row 'Cl. Gap' refers to the gap closed at the end of the optimization whether it achieved optimality or not. The row 'Fin. by all' indicates whether all methods were able to solve this problem to optimality. For each method, the 'Fin.' rows indicate the termination status of the optimization: 0 for optimality, 1 for unfeasibility, 2 for unboundedness, 3 for unfeasibility or unboundedness, 4 for another stopping criterion with a feasible solution found, and 5 for another stopping criterion with no feasible solution found.

| | Problem names | mas76 | misc03 | misc06 | misc07 | mitre | mod008 | mod010 | mod011 | nw04 | p0033 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | LP Obj. | 38,893.90 | 1,910 | 12,841.70 | 1,415 | 114,741 | 290.93 | 6,532.08 | -62,122,000 | 16,310.70 | 2,520.57 |
| | True Obj. | 40,005.10 | 3,360 | 12,850.90 | 2,810 | 115,155 | 307 | 6,548 | -54,558,500 | 16,862 | 3,089 |
| | Fin. by all | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Random | Fin. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | 40,005.10 | 3,360 | 12,850.90 | 2,810 | 115,155 | 307 | 6,548 | -54,600,000 | 16,862 | 3,089 |
| | Nodes | 2,869,383 | 1,009 | 7 | 55,815 | 15 | 1,589 | 1,505 | 58,297 | 255 | 1 |
| | Time | 1,004.72 | 0.93 | 0.55 | 48.61 | 4.33 | 0.47 | 3.96 | 3,726.60 | 46.53 | 0.01 |
| | Cl. Gap | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0.99 | 1 | 1 |
| MIB | Fin. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | 40,005.10 | 3,360 | 12,850.90 | 2,810 | 115,155 | 307 | 6,548 | -54,558,500 | 16,862 | 3,089 |
| | Nodes | 1,742,607 | 279 | 11 | 13,707 | 29 | 1,537 | 39 | 4,027 | 337 | 1 |
| | Time | 518.99 | 0.47 | 0.56 | 13.28 | 4.27 | 0.47 | 0.65 | 333.87 | 51.31 | 0.01 |
| | Cl. Gap | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| NCB | Fin. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | 40,005.10 | 3,360 | 12,850.90 | 2,810 | 115,155 | 307 | 6,548 | -54,558,500 | 16,862 | 3,089 |
| | Nodes | 400,115 | 455 | 7 | 26,769 | 20 | 783 | 13 | 160,639 | 39 | 1 |
| | Time | 311.09 | 1.99 | 0.59 | 222.85 | 4.26 | 0.47 | 0.64 | 34,024.60 | 34.68 | 0.01 |
| | Cl. Gap | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| FSB | Fin. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | 40,005.10 | 3,360 | 12,850.90 | 2,810 | 115,155 | 307 | 6,548 | -54,558,500 | 16,862 | 3,089 |
| | Nodes | 355,507 | 301 | 7 | 17,299 | 20 | 423 | 11 | 19,789 | 55 | 1 |
| | Time | 394.68 | 3.60 | 0.61 | 410.78 | 4.44 | 0.52 | 0.91 | 6,564.28 | 47.72 | 0.01 |
| | Cl. Gap | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| RB | Fin. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | 40,005.10 | 3,360 | 12,850.90 | 2,810 | 115,155 | 307 | 6,548 | -54,558,500 | 16,862 | 3,089 |
| | Nodes | 561,667 | 305 | 9 | 19,231 | 15 | 1,755 | 115 | 64,870 | 1,213 | 1 |
| | Time | 188.54 | 1.02 | 0.61 | 61.48 | 4.30 | 0.78 | 1.24 | 7,436.06 | 143.11 | 0.01 |
| | Cl. Gap | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Learned | Fin. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | 40,005.10 | 3,360 | 12,850.90 | 2,810 | 115,155 | 307 | 6,548 | -54,558,500 | 16,862 | 3,089 |
| | Nodes | 564,679 | 1,979 | 9 | 110,751 | 10 | 993 | 99 | 5,064 | 55 | 1 |
| | Time | 307.45 | 3.43 | 0.56 | 212.24 | 4.05 | 0.67 | 1.03 | 541.92 | 34.02 | 0.01 |
| | Closed gap | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Table A.19: Detailed results for the updated list of MIPLIB problems (2/3). Time limit = none and CPLEX's cuts and heuristics applied. The row 'Cl. Gap' refers to the gap closed at the end of the optimization whether it achieved optimality or not. The row 'Fin. by all' indicates whether all methods were able to solve this problem to optimality. For each method, the 'Fin.' rows indicate the termination status of the optimization: 0 for optimality, 1 for unfeasibility, 2 for unboundedness, 3 for unfeasibility or unboundedness, 4 for another stopping criterion with a feasible solution found, and 5 for another stopping criterion with no feasible solution found.

| | Problem names | p0201 | pk1 | pp08aCUTS | qiu | rentacar | rgn | stein27 | stein45 | vpm1 | vpm2 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | LP Obj. | 6,875 | 0 | 5,480.61 | -931.64 | 28,806,100 | 48.80 | 13 | 22 | 15.42 | 9.89 |
| | True Obj. | 7,615 | 11 | 7,350 | -132.87 | 30,356,800 | 82.20 | 18 | 30 | 20 | 13.75 |
| | Fin. by all | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Random | Fin. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | 7,615 | 11 | 7,350 | -132.87 | 30,400,000 | 82.20 | 18 | 30 | 20 | 13.75 |
| | Nodes | 40 | 861,991 | 4,965 | 462,433 | 9 | 2,605 | 4,499 | 61,825 | 0 | 5,869 |
| | Time | 0.73 | 265 | 8.08 | 3,565.69 | 12.75 | 0.60 | 0.78 | 22.17 | 0.02 | 4.95 |
| | Cl. Gap | 1 | 1 | 1 | 1 | 1.03 | 1 | 1 | 1 | 1 | 1 |
| MIB | Fin. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | 7,615 | 11 | 7,350 | -132.87 | 30,356,800 | 82.20 | 18 | 30 | 20 | 13.75 |
| | Nodes | 57 | 892,187 | 5,157 | 265,859 | 7 | 2,233 | 4,633 | 82,377 | 0 | 5,721 |
| | Time | 0.83 | 272.01 | 7.98 | 1,999.22 | 12.93 | 0.51 | 0.80 | 27.61 | 0.01 | 5.15 |
| | Cl. Gap | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| NCB | Fin. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | 7,615 | 11 | 7,350 | -132.87 | 30,356,800 | 82.20 | 18 | 30 | 20 | 13.75 |
| | Nodes | 67 | 243,581 | 12,497 | 12,541 | 7 | 3,007 | 3,499 | 39,787 | 0 | 5,781 |
| | Time | 1.31 | 430.17 | 68.16 | 913.26 | 14.27 | 1.80 | 1.79 | 90.07 | 0.01 | 14.19 |
| | Cl. Gap | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| FSB | Fin. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | 7,615 | 11 | 7,350 | -132.87 | 30,356,800 | 82.20 | 18 | 30 | 20 | 13.75 |
| | Nodes | 67 | 284,851 | 7,523 | 27,779 | 7 | 1,489 | 2,153 | 24,949 | 0 | 3,847 |
| | Time | 2.60 | 651.97 | 70.38 | 3,074.14 | 14.23 | 0.82 | 3.46 | 244.19 | 0.02 | 18.59 |
| | Cl. Gap | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| RB | Fin. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | 7,615 | 11 | 7,350 | -132.87 | 30,356,800 | 82.20 | 18 | 30 | 20 | 13.75 |
| | Nodes | 113 | 308,491 | 9,515 | 26,277 | 11 | 2,271 | 4,231 | 53,623 | 0 | 3,197 |
| | Time | 1.13 | 153.21 | 22.80 | 745.02 | 14.98 | 0.83 | 1.58 | 58.58 | 0.01 | 4.18 |
| | Cl. Gap | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Learned | Fin. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Obj. | 7,615 | 11 | 7,350 | -132.87 | 30,356,800 | 82.20 | 18 | 30 | 20 | 13.75 |
| | Nodes | 43 | 897,129 | 3,131 | 25,065 | 7 | 1,619 | 4,449 | 54,541 | 0 | 10,207 |
| | Time | 0.88 | 651.99 | 8.18 | 363.04 | 13.03 | 0.65 | 2.78 | 69.59 | 0.01 | 18.03 |
| | Closed gap | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Table A.20: Detailed results for the updated list of MIPLIB problems (3/3). Time limit = none and CPLEX's cuts and heuristics applied. The row 'Cl. Gap' refers to the gap closed at the end of the optimization whether it achieved optimality or not. The row 'Fin. by all' indicates whether all methods were able to solve this problem to optimality. For each method, the 'Fin.' rows indicate the termination status of the optimization: 0 for optimality, 1 for unfeasibility, 2 for unboundedness, 3 for unfeasibility or unboundedness, 4 for another stopping criterion with a feasible solution found, and 5 for another stopping criterion with no feasible solution found.

## A.4 Online learning of branching decisions: complete experimental results

The next tables report the detailed optimization results presented in Section 4.5.3.

Tables A.21 through A.30 report the detailed optimization times obtained with the competing branching heuristics on the considered MIPLIB problems. Tables A.31 through A.40 report the corresponding numbers of nodes processed by B&B before the optimization terminates (either because optimality is proved or because the time budget is exhausted) for the same MIPLIB problems.

We refer the reader to Section 4.5.3 of the main document for an analysis and a clearer presentation of the results.

| | Random seed | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Prob. name | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 10teams | 725 | 363 | 1,355 | 1 | 1,156 | 3,505 | 290 | 332 | 326 | 240 |
| aflow30a | 925 | 763 | 786 | 888 | 782 | 774 | 879 | 955 | 651 | 668 |
| aflow40b | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 |
| air03 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| air04 | 866 | 7,200 | 1,598 | 4,572 | 7,200 | 7,200 | 1,670 | 7,200 | 6,309 | 7,200 |
| air05 | 3,964 | 3,098 | 3,602 | 3,035 | 3,334 | 3,017 | 3,496 | 2,786 | 3,617 | 2,426 |
| cap6000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| dcmulti | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| egout | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| fiber | 11 | 100 | 9 | 10 | 18 | 7 | 11 | 13 | 9 | 15 |
| fixnet6 | 8 | 8 | 8 | 4 | 8 | 6 | 7 | 6 | 8 | 6 |
| harp2 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 |
| khb05250 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| l152lav | 24 | 23 | 21 | 28 | 23 | 26 | 18 | 17 | 12 | 15 |
| lseu | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| mas74 | 4,560 | 3,894 | 3,890 | 3,886 | 3,864 | 3,887 | 3,879 | 5,407 | 3,888 | 3,865 |
| mas76 | 350 | 347 | 510 | 346 | 568 | 350 | 349 | 346 | 569 | 350 |
| misc03 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| misc06 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| misc07 | 393 | 475 | 470 | 487 | 525 | 402 | 508 | 506 | 432 | 506 |
| mitre | 5 | 4 | 5 | 5 | 5 | 6 | 5 | 5 | 4 | 3 |
| mod008 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Table A.21: Strong branching optimization times: detailed optimization results for the considered MIPLIB problems and for each random seed (1/2). The table reports the optimization times (in seconds) taken by B&B before the optimization terminates. A value of 7,200 indicates that the problem has not been solved within the provided time limit.

| | Random seed | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Prob. name | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| mod010 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| mod011 | 1,986 | 1,213 | 1,555 | 2,022 | 2,453 | 2,332 | 2,264 | 2,250 | 2,258 | 1,803 |
| modglob | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| nw04 | 70 | 84 | 90 | 85 | 111 | 75 | 96 | 103 | 65 | 63 |
| opt1217 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 |
| p0033 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| p0201 | 2 | 3 | 2 | 3 | 2 | 3 | 3 | 6 | 2 | 3 |
| p0282 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| p0548 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| p2756 | 1,418 | 69 | 52 | 4,594 | 2,407 | 576 | 160 | 2,831 | 2,388 | 2,651 |
| pk1 | 638 | 593 | 526 | 435 | 425 | 737 | 599 | 710 | 521 | 615 |
| pp08a | 6 | 5 | 6 | 6 | 7 | 7 | 5 | 6 | 6 | 5 |
| pp08aCUTS | 14 | 14 | 17 | 14 | 14 | 17 | 14 | 14 | 17 | 14 |
| qiu | 3,615 | 4,195 | 4,859 | 3,729 | 4,176 | 3,746 | 3,564 | 3,905 | 2,722 | 4,086 |
| rentacar | 14 | 14 | 15 | 16 | 14 | 15 | 15 | 14 | 14 | 15 |
| rgn | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| set1ch | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| stein27 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 3 | 4 |
| stein45 | 197 | 224 | 216 | 197 | 195 | 198 | 251 | 197 | 193 | 217 |
| tr12-30 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 |
| vpm1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| vpm2 | 11 | 16 | 11 | 11 | 12 | 11 | 12 | 11 | 11 | 11 |

Table A.22: Strong branching optimization times: detailed optimization results for the considered MIPLIB problems and for each random seed (2/2). The table reports the optimization times (in seconds) taken by B&B before the optimization terminates. A value of 7,200 indicates that the problem has not been solved within the provided time limit.

| | Random seed | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Prob. name | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 10teams | 52 | 28 | 42 | 1 | 172 | 26 | 191 | 200 | 113 | 40 |
| aflow30a | 435 | 494 | 298 | 649 | 196 | 388 | 542 | 355 | 409 | 352 |
| aflow40b | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 |
| air03 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| air04 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 |
| air05 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 |
| cap6000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| dcmulti | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| egout | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| fiber | 4 | 39 | 2 | 3 | 4 | 6 | 3 | 4 | 5 | 5 |
| fixnet6 | 3,444 | 700 | 1,429 | 591 | 1,456 | 5,319 | 483 | 1,560 | 1,724 | 999 |
| harp2 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 |
| khb05250 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| l152lav | 4 | 4 | 7 | 7 | 7 | 7 | 7 | 6 | 7 | 7 |
| lseu | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| mas74 | 4,782 | 2,883 | 2,901 | 3,708 | 2,870 | 2,886 | 3,879 | 3,592 | 3,871 | 3,872 |
| mas76 | 140 | 140 | 140 | 140 | 144 | 144 | 140 | 140 | 140 | 144 |
| misc03 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| misc06 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| misc07 | 70 | 61 | 57 | 62 | 58 | 77 | 57 | 91 | 53 | 75 |
| mitre | 5 | 4 | 5 | 5 | 5 | 6 | 5 | 5 | 4 | 3 |
| mod008 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 |

Table A.23: Reliability branching optimization times: detailed optimization results for the considered MIPLIB problems and for each random seed (1/2). The table reports the optimization times (in seconds) taken by B&B before the optimization terminates. A value of 7,200 indicates that the problem has not been solved within the provided time limit.

| | Random seed | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Prob. name | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| mod010 | 4 | 1 | 1 | 4 | 1 | 1 | 1 | 1 | 1 | 1 |
| mod011 | 1,599 | 900 | 972 | 1,593 | 2,686 | 1,638 | 1,746 | 3,065 | 2,372 | 1,709 |
| modglob | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| nw04 | 113 | 124 | 96 | 95 | 148 | 121 | 161 | 149 | 121 | 142 |
| opt1217 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 |
| p0033 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| p0201 | 6 | 6 | 5 | 3 | 2 | 5 | 5 | 3 | 6 | 4 |
| p0282 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| p0548 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| p2756 | 7,200 | 7,200 | 3,915 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 |
| pk1 | 202 | 174 | 181 | 193 | 160 | 137 | 164 | 189 | 224 | 224 |
| pp08a | 3 | 4 | 3 | 3 | 3 | 5 | 4 | 3 | 3 | 4 |
| pp08aCUTS | 3 | 3 | 3 | 3 | 3 | 3 | 4 | 3 | 3 | 3 |
| qiu | 603 | 602 | 611 | 493 | 397 | 417 | 703 | 1,779 | 1,827 | 1,793 |
| rentacar | 15 | 15 | 16 | 17 | 15 | 16 | 16 | 15 | 15 | 16 |
| rgn | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| set1ch | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| stein27 | 2 | 2 | 1 | 2 | 2 | 2 | 1 | 2 | 2 | 2 |
| stein45 | 61 | 60 | 60 | 54 | 59 | 58 | 57 | 57 | 63 | 57 |
| tr12-30 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 |
| vpm1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| vpm2 | 3 | 4 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |

Table A.24: Reliability branching optimization times: detailed optimization results for the considered MIPLIB problems and for each random seed (2/2). The table reports the optimization times (in seconds) taken by B&B before the optimization terminates. A value of 7,200 indicates that the problem has not been solved within the provided time limit.

| | Random seed | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Prob. name | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 10teams | 66 | 9 | 71 | 1 | 22 | 17 | 12 | 24 | 8 | 12 |
| aflow30a | 461 | 501 | 488 | 660 | 349 | 393 | 488 | 296 | 368 | 280 |
| aflow40b | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 |
| air03 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| air04 | 103 | 975 | 116 | 166 | 291 | 121 | 251 | 289 | 256 | 243 |
| air05 | 2,166 | 525 | 271 | 2,420 | 1,027 | 1,266 | 623 | 428 | 1,164 | 2,556 |
| cap6000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| dcmulti | 4 | 4 | 5 | 4 | 5 | 4 | 5 | 4 | 6 | 5 |
| egout | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| fiber | 1 | 2 | 1 | 1 | 2 | 1 | 1 | 2 | 1 | 1 |
| fixnet6 | 11 | 17 | 12 | 17 | 15 | 12 | 17 | 17 | 17 | 9 |
| harp2 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 |
| khb05250 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| l152lav | 5 | 127 | 3 | 8 | 4 | 6 | 10 | 5 | 9 | 6 |
| lseu | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| mas74 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 |
| mas76 | 1,214 | 799 | 896 | 1,237 | 900 | 732 | 772 | 947 | 896 | 776 |
| misc03 | 3 | 4 | 3 | 5 | 4 | 4 | 3 | 5 | 6 | 5 |
| misc06 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| misc07 | 233 | 193 | 161 | 279 | 146 | 274 | 164 | 180 | 169 | 169 |
| mitre | 5 | 4 | 5 | 5 | 6 | 5 | 5 | 5 | 4 | 3 |
| mod008 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Table A.25: Batch learned branching optimization times: detailed optimization results for the considered MIPLIB problems and for each random seed (1/2). The table reports the optimization times (in seconds) taken by B&B before the optimization terminates. A value of 7,200 indicates that the problem has not been solved within the provided time limit.

| | Random seed | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Prob. name | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| mod010 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| mod011 | 541 | 403 | 443 | 445 | 362 | 415 | 623 | 453 | 461 | 463 |
| modglob | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| nw04 | 26 | 31 | 38 | 28 | 30 | 26 | 36 | 39 | 31 | 31 |
| opt1217 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 |
| p0033 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| p0201 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 1 | 2 | 2 |
| p0282 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| p0548 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| p2756 | 4 | 2 | 3 | 1 | 4 | 2 | 5 | 3 | 1 | 7 |
| pk1 | 1,170 | 831 | 1,266 | 861 | 1,038 | 903 | 1,133 | 1,142 | 1,021 | 1,520 |
| pp08a | 4 | 4 | 4 | 4 | 3 | 3 | 4 | 4 | 3 | 4 |
| pp08aCUTS | 3 | 3 | 3 | 2 | 2 | 3 | 3 | 3 | 2 | 3 |
| qiu | 862 | 931 | 526 | 867 | 664 | 768 | 600 | 592 | 399 | 571 |
| rentacar | 14 | 14 | 15 | 15 | 14 | 15 | 15 | 14 | 14 | 15 |
| rgn | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| set1ch | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| stein27 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| stein45 | 78 | 77 | 81 | 79 | 93 | 74 | 69 | 73 | 74 | 89 |
| tr12-30 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 |
| vpm1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| vpm2 | 5 | 5 | 6 | 6 | 5 | 5 | 5 | 4 | 6 | 6 |

Table A.26: Batch learned branching optimization times: detailed optimization results for the considered MIPLIB problems and for each random seed (2/2). The table reports the optimization times (in seconds) taken by B&B before the optimization terminates. A value of 7,200 indicates that the problem has not been solved within the provided time limit.

| | Random seed | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Prob. name | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 10teams | 59 | 19 | 74 | 1 | 39 | 46 | 11 | 52 | 45 | 11 |
| aflow30a | 202 | 188 | 233 | 152 | 124 | 144 | 144 | 158 | 144 | 219 |
| aflow40b | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 |
| air03 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| air04 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 |
| air05 | 3,867 | 3,418 | 6,816 | 6,317 | 5,800 | 5,003 | 5,375 | 5,472 | 4,597 | 5,097 |
| cap6000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| dcmulti | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| egout | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| fiber | 3 | 11 | 3 | 2 | 4 | 3 | 3 | 4 | 2 | 3 |
| fixnet6 | 6 | 9 | 10 | 3 | 9 | 5 | 7 | 7 | 9 | 5 |
| harp2 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 |
| khb05250 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| l152lav | 5 | 5 | 4 | 3 | 4 | 4 | 4 | 3 | 5 | 5 |
| lseu | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| mas74 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 |
| mas76 | 547 | 496 | 563 | 563 | 912 | 912 | 562 | 455 | 564 | 440 |
| misc03 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| misc06 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| misc07 | 16 | 16 | 16 | 17 | 16 | 17 | 16 | 17 | 16 | 17 |
| mitre | 5 | 4 | 5 | 5 | 5 | 6 | 5 | 5 | 4 | 3 |
| mod008 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Table A.27: Online learning branching optimization times: detailed optimization results for the considered MIPLIB problems and for each random seed (1/2). The table reports the optimization times (in seconds) taken by B&B before the optimization terminates. A value of 7,200 indicates that the problem has not been solved within the provided time limit.

| | Random seed | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Prob. name | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| mod010 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| mod011 | 497 | 410 | 455 | 567 | 642 | 558 | 718 | 590 | 544 | 574 |
| modglob | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| nw04 | 109 | 118 | 89 | 95 | 66 | 83 | 124 | 117 | 69 | 88 |
| opt1217 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 |
| p0033 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| p0201 | 2 | 2 | 2 | 1 | 1 | 2 | 2 | 2 | 2 | 2 |
| p0282 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| p0548 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| p2756 | 4,322 | 1,201 | 3,074 | 7,200 | 4,226 | 3,513 | 3,062 | 3,450 | 4,107 | 7,200 |
| pk1 | 265 | 239 | 292 | 393 | 302 | 313 | 338 | 358 | 307 | 271 |
| pp08a | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 4 | 3 | 3 |
| pp08aCUTS | 3 | 4 | 4 | 3 | 3 | 4 | 4 | 4 | 4 | 4 |
| qiu | 2,252 | 2,527 | 2,502 | 2,474 | 2,429 | 1,745 | 2,307 | 2,598 | 2,396 | 2,411 |
| rentacar | 14 | 14 | 15 | 16 | 14 | 15 | 15 | 14 | 14 | 15 |
| rgn | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| set1ch | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| stein27 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| stein45 | 28 | 30 | 31 | 28 | 28 | 33 | 32 | 29 | 30 | 29 |
| tr12-30 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 |
| vpm1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| vpm2 | 4 | 2 | 4 | 2 | 2 | 4 | 2 | 4 | 4 | 4 |

Table A.28: Online learning branching optimization times: detailed optimization results for the considered MIPLIB problems and for each random seed (2/2). The table reports the optimization times (in seconds) taken by B&B before the optimization terminates. A value of 7,200 indicates that the problem has not been solved within the provided time limit.

| | Random seed | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Prob. name | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 10teams | 46 | 19 | 82 | 1 | 47 | 48 | 11 | 55 | 46 | 11 |
| aflow30a | 180 | 144 | 165 | 154 | 136 | 164 | 185 | 154 | 151 | 210 |
| aflow40b | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 |
| air03 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| air04 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 |
| air05 | 6,291 | 4,225 | 4,284 | 7,108 | 5,854 | 5,645 | 4,582 | 6,325 | 5,393 | 7,031 |
| cap6000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| dcmulti | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| egout | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| fiber | 3 | 11 | 3 | 2 | 4 | 3 | 3 | 4 | 2 | 3 |
| fixnet6 | 4 | 6 | 9 | 4 | 6 | 6 | 6 | 6 | 6 | 4 |
| harp2 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 |
| khb05250 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| l152lav | 5 | 5 | 5 | 3 | 4 | 4 | 4 | 3 | 5 | 5 |
| lseu | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| mas74 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 |
| mas76 | 633 | 986 | 615 | 613 | 638 | 1,187 | 1,186 | 987 | 1,187 | 1,045 |
| misc03 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| misc06 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| misc07 | 17 | 16 | 15 | 18 | 15 | 18 | 16 | 17 | 16 | 16 |
| mitre | 5 | 4 | 5 | 5 | 5 | 6 | 5 | 5 | 4 | 3 |
| mod008 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Table A.29: Online perpetual learning branching optimization times: detailed optimization results for the considered MIPLIB problems and for each random seed (1/2). The table reports the optimization times (in seconds) taken by B&B before the optimization terminates. A value of 7,200 indicates that the problem has not been solved within the provided time limit.

| | Random seed | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Prob. name | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| mod010 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| mod011 | 423 | 342 | 267 | 410 | 421 | 355 | 424 | 521 | 364 | 380 |
| modglob | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| nw04 | 110 | 110 | 97 | 93 | 66 | 86 | 124 | 114 | 69 | 81 |
| opt1217 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 |
| p0033 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| p0201 | 3 | 2 | 2 | 2 | 1 | 2 | 2 | 2 | 2 | 2 |
| p0282 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| p0548 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| p2756 | 3,123 | 2,420 | 2,985 | 3,773 | 6,399 | 2,747 | 7,200 | 1,720 | 2,246 | 3,166 |
| pk1 | 288 | 219 | 244 | 246 | 236 | 249 | 231 | 261 | 239 | 235 |
| pp08a | 3 | 3 | 3 | 3 | 3 | 4 | 3 | 3 | 3 | 3 |
| pp08aCUTS | 3 | 4 | 4 | 3 | 3 | 4 | 4 | 4 | 4 | 4 |
| qiu | 1,318 | 1,249 | 1,402 | 1,495 | 1,323 | 1,548 | 1,341 | 1,571 | 1,504 | 1,464 |
| rentacar | 14 | 14 | 15 | 16 | 14 | 15 | 15 | 14 | 14 | 15 |
| rgn | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| set1ch | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| stein27 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| stein45 | 29 | 28 | 28 | 26 | 27 | 28 | 28 | 28 | 28 | 28 |
| tr12-30 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 | 7,200 |
| vpm1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| vpm2 | 2 | 3 | 5 | 5 | 3 | 2 | 2 | 3 | 5 | 5 |

Table A.30: Online perpetual learning branching optimization times: detailed optimization results for the considered MIPLIB problems and for each random seed (2/2). The table reports the optimization times (in seconds) taken by B&B before the optimization terminates. A value of 7,200 indicates that the problem has not been solved within the provided time limit.

| | Random seed | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Prob. name | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 10teams | 211 | 100 | 554 | 0 | 470 | 1,025 | 91 | 80 | 100 | 60 |
| aflow30a | 23,971 | 20,111 | 21,515 | 24,113 | 18,711 | 19,267 | 22,213 | 23,725 | 16,493 | 17,437 |
| aflow40b | 18,526 | 16,468 | 17,897 | 17,812 | 18,000 | 15,533 | 16,810 | 17,886 | 19,565 | 16,475 |
| air03 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| air04 | 67 | 114 | 95 | 166 | 87 | 94 | 113 | 76 | 157 | 94 |
| air05 | 359 | 249 | 361 | 153 | 255 | 249 | 189 | 279 | 361 | 241 |
| cap6000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| dcmulti | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 |
| egout | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| fiber | 493 | 4,721 | 273 | 465 | 871 | 309 | 515 | 557 | 381 | 575 |
| fixnet6 | 729 | 745 | 767 | 135 | 745 | 457 | 379 | 339 | 745 | 273 |
| harp2 | 123,687 | 123,760 | 123,658 | 123,768 | 116,308 | 112,598 | 95,257 | 107,238 | 114,844 | 112,379 |
| khb05250 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| l152lav | 183 | 197 | 181 | 201 | 191 | 206 | 196 | 236 | 123 | 169 |
| lseu | 107 | 95 | 99 | 107 | 95 | 117 | 105 | 129 | 123 | 95 |
| mas74 | 2,825,401 | 2,520,219 | 2,520,219 | 2,520,219 | 2,499,433 | 2,518,417 | 2,499,433 | 3,239,257 | 2,518,417 | 2,499,433 |
| mas76 | 310,911 | 309,689 | 424,841 | 309,697 | 456,823 | 310,895 | 310,911 | 309,697 | 456,823 | 310,895 |
| misc03 | 289 | 285 | 289 | 287 | 285 | 291 | 297 | 289 | 269 | 300 |
| misc06 | 7 | 7 | 10 | 9 | 7 | 7 | 7 | 7 | 7 | 7 |
| misc07 | 15,165 | 17,633 | 20,189 | 18,773 | 22,385 | 15,041 | 21,063 | 19,905 | 17,999 | 20,009 |
| mitre | 20 | 0 | 9 | 0 | 10 | 14 | 11 | 20 | 0 | 0 |
| mod008 | 281 | 241 | 215 | 281 | 241 | 323 | 281 | 241 | 323 | 281 |

Table A.31: Strong branching number of nodes: detailed optimization results for the considered MIPLIB problems and for each random seed (1/2). The table reports the number of nodes processed by B&B before the optimization terminates. To determine if, at termination, the problem is solved to optimality, check the corresponding table reporting the optimization times (a value of 7,200 indicates that the problem has not been solved within the provided time limit).

| | Random seed | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Prob. name | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| mod010 | 21 | 17 | 7 | 13 | 17 | 11 | 11 | 23 | 15 | 11 |
| mod011 | 5,779 | 3,247 | 4,355 | 5,441 | 6,420 | 6,600 | 5,831 | 6,509 | 6,399 | 5,697 |
| modglob | 71 | 71 | 86 | 82 | 74 | 71 | 71 | 71 | 85 | 86 |
| nw04 | 88 | 123 | 155 | 125 | 169 | 179 | 159 | 147 | 93 | 139 |
| opt1217 | 390,267 | 389,230 | 382,698 | 379,904 | 378,225 | 391,932 | 387,309 | 401,350 | 397,100 | 381,622 |
| p0033 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| p0201 | 65 | 81 | 83 | 65 | 42 | 81 | 87 | 237 | 77 | 87 |
| p0282 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| p0548 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| p2756 | 40,563 | 2,347 | 2,095 | 135,909 | 66,103 | 16,680 | 4,359 | 86,422 | 67,177 | 82,367 |
| pk1 | 284,367 | 267,361 | 244,103 | 202,817 | 202,099 | 299,349 | 267,435 | 298,037 | 239,693 | 270,275 |
| pp08a | 1,269 | 1,197 | 1,279 | 1,279 | 1,303 | 1,437 | 1,215 | 1,299 | 1,387 | 1,219 |
| pp08aCUTS | 1,381 | 1,449 | 1,563 | 1,381 | 1,381 | 1,563 | 1,449 | 1,371 | 1,563 | 1,371 |
| qiu | 30,991 | 30,309 | 35,919 | 28,339 | 34,697 | 31,073 | 31,683 | 33,997 | 23,105 | 32,391 |
| rentacar | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| rgn | 0 | 819 | 0 | 891 | 0 | 49 | 49 | 0 | 0 | 0 |
| set1ch | 19 | 19 | 19 | 19 | 19 | 19 | 19 | 19 | 19 | 19 |
| stein27 | 2,117 | 2,157 | 2,093 | 2,171 | 2,189 | 2,135 | 2,115 | 2,181 | 2,115 | 2,169 |
| stein45 | 23,189 | 25,057 | 24,619 | 23,373 | 23,339 | 23,611 | 25,187 | 23,551 | 22,793 | 24,795 |
| tr12-30 | 221,522 | 223,714 | 222,754 | 217,439 | 224,220 | 218,883 | 222,089 | 219,664 | 215,561 | 223,800 |
| vpm1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| vpm2 | 1,770 | 2,770 | 1,690 | 1,690 | 2,029 | 1,770 | 1,960 | 1,770 | 1,690 | 1,690 |

Table A.32: Strong branching number of nodes: detailed optimization results for the considered MIPLIB problems and for each random seed (2/2). The table reports the number of nodes processed by B&B before the optimization terminates. To determine if, at termination, the problem is solved to optimality, check the corresponding table reporting the optimization times (a value of 7,200 indicates that the problem has not been solved within the provided time limit).

| | Random seed | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Prob. name | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 10teams | 169 | 85 | 143 | 0 | 679 | 76 | 858 | 1,860 | 426 | 133 |
| aflow30a | 49,891 | 67,953 | 42,377 | 77,391 | 26,645 | 51,131 | 63,557 | 47,031 | 44,039 | 47,495 |
| aflow40b | 101,718 | 94,664 | 99,227 | 108,641 | 98,841 | 97,955 | 89,130 | 111,120 | 113,714 | 112,081 |
| air03 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| air04 | 17,047 | 13,934 | 12,779 | 18,136 | 16,891 | 11,970 | 16,520 | 14,726 | 12,180 | 16,328 |
| air05 | 19,923 | 29,059 | 15,860 | 12,634 | 16,377 | 13,764 | 35,759 | 23,498 | 12,951 | 13,811 |
| cap6000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| dcmulti | 59 | 59 | 59 | 59 | 59 | 59 | 59 | 59 | 59 | 59 |
| egout | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| fiber | 490 | 16,853 | 89 | 309 | 563 | 1,449 | 251 | 741 | 1,305 | 697 |
| fixnet6 | 938,437 | 209,431 | 429,955 | 178,605 | 387,975 | 1,285,429 | 144,945 | 460,795 | 506,325 | 270,121 |
| harp2 | 473,249 | 482,744 | 472,000 | 532,215 | 483,414 | 483,376 | 466,795 | 534,081 | 488,480 | 482,721 |
| khb05250 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| l152lav | 169 | 311 | 623 | 623 | 623 | 623 | 593 | 467 | 623 | 623 |
| lseu | 201 | 159 | 87 | 181 | 117 | 219 | 147 | 135 | 123 | 101 |
| mas74 | 6,309,775 | 5,101,537 | 5,092,723 | 5,852,027 | 5,087,621 | 5,101,537 | 5,903,643 | 5,787,073 | 5,903,643 | 5,903,643 |
| mas76 | 412,445 | 412,445 | 412,445 | 412,445 | 417,981 | 417,981 | 412,445 | 412,445 | 412,445 | 417,981 |
| misc03 | 293 | 273 | 281 | 249 | 287 | 287 | 305 | 269 | 253 | 287 |
| misc06 | 7 | 5 | 20 | 5 | 5 | 7 | 7 | 5 | 7 | 7 |
| misc07 | 22,895 | 20,089 | 21,247 | 18,367 | 20,613 | 22,503 | 20,263 | 30,161 | 19,237 | 25,827 |
| mitre | 8 | 0 | 9 | 0 | 7 | 10 | 7 | 8 | 0 | 0 |
| mod008 | 1,191 | 637 | 637 | 1,191 | 637 | 703 | 1,191 | 637 | 703 | 1,191 |

Table A.33: Reliability branching number of nodes: detailed optimization results for the considered MIPLIB problems and for each random seed (1/2). The table reports the number of nodes processed by B&B before the optimization terminates. To determine if, at termination, the problem is solved to optimality, check the corresponding table reporting the optimization times (a value of 7,200 indicates that the problem has not been solved within the provided time limit).

| | Random seed | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Prob. name | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| mod010 | 825 | 69 | 31 | 547 | 65 | 51 | 67 | 89 | 53 | 37 |
| mod011 | 11,123 | 6,979 | 8,791 | 11,727 | 19,643 | 13,375 | 12,261 | 22,238 | 18,037 | 13,877 |
| modglob | 119 | 119 | 150 | 150 | 127 | 119 | 119 | 119 | 150 | 150 |
| nw04 | 917 | 1,059 | 764 | 715 | 1,335 | 1,089 | 1,291 | 1,111 | 953 | 1,216 |
| opt1217 | 2,212,674 | 2,238,835 | 2,458,968 | 2,340,720 | 2,378,320 | 2,397,980 | 2,326,890 | 2,402,908 | 2,306,149 | 2,386,198 |
| p0033 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| p0201 | 1,927 | 2,325 | 2,101 | 495 | 441 | 1,661 | 1,645 | 469 | 1,695 | 1,507 |
| p0282 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| p0548 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| p2756 | 290,701 | 281,108 | 223,111 | 284,562 | 196,323 | 278,667 | 258,132 | 271,985 | 333,552 | 224,810 |
| pk1 | 387,537 | 318,705 | 342,101 | 336,935 | 304,993 | 254,095 | 294,313 | 306,927 | 358,031 | 377,519 |
| pp08a | 927 | 1,449 | 897 | 893 | 925 | 1,951 | 1,449 | 893 | 1,091 | 1,441 |
| pp08aCUTS | 853 | 853 | 849 | 833 | 853 | 853 | 897 | 849 | 853 | 853 |
| qiu | 20,695 | 20,715 | 20,643 | 17,755 | 15,155 | 14,977 | 24,915 | 63,909 | 63,677 | 65,117 |
| rentacar | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |
| rgn | 0 | 509 | 0 | 949 | 0 | 89 | 181 | 0 | 0 | 0 |
| set1ch | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 |
| stein27 | 4,219 | 4,311 | 3,907 | 4,237 | 4,245 | 4,051 | 3,941 | 4,145 | 4,145 | 4,279 |
| stein45 | 53,615 | 54,001 | 52,283 | 48,997 | 53,477 | 53,137 | 52,847 | 53,489 | 50,933 | 52,597 |
| tr12-30 | 942,594 | 973,305 | 931,317 | 913,917 | 968,784 | 922,522 | 921,355 | 947,224 | 900,898 | 935,389 |
| vpm1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| vpm2 | 1,297 | 1,497 | 1,149 | 1,155 | 1,069 | 1,297 | 1,073 | 1,297 | 1,149 | 1,149 |

Table A.34: Reliability branching number of nodes: detailed optimization results for the considered MIPLIB problems and for each random seed (2/2). The table reports the number of nodes processed by B&B before the optimization terminates. To determine if, at termination, the problem is solved to optimality, check the corresponding table reporting the optimization times (a value of 7,200 indicates that the problem has not been solved within the provided time limit).

| | Random seed | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Prob. name | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 10teams | 916 | 29 | 858 | 0 | 201 | 106 | 66 | 215 | 27 | 70 |
| aflow30a | 58,533 | 66,893 | 56,765 | 76,579 | 53,181 | 49,855 | 61,685 | 37,127 | 48,401 | 37,923 |
| aflow40b | 96,655 | 83,409 | 81,481 | 78,570 | 93,822 | 89,635 | 90,406 | 86,753 | 96,657 | 93,032 |
| air03 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| air04 | 291 | 4,223 | 355 | 353 | 1,123 | 197 | 1,109 | 1,035 | 715 | 327 |
| air05 | 22,385 | 2,401 | 2,409 | 26,655 | 8,077 | 13,895 | 4,425 | 4,393 | 11,947 | 23,949 |
| cap6000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| dcmulti | 643 | 845 | 1,065 | 691 | 1,029 | 807 | 919 | 847 | 1,253 | 1,013 |
| egout | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| fiber | 37 | 39 | 29 | 39 | 71 | 71 | 47 | 49 | 61 | 27 |
| fixnet6 | 2,103 | 3,555 | 2,067 | 5,027 | 3,265 | 2,373 | 3,549 | 3,547 | 3,547 | 1,577 |
| harp2 | 323,700 | 311,955 | 362,799 | 301,929 | 317,628 | 322,138 | 313,877 | 280,947 | 286,326 | 326,986 |
| khb05250 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| l152lav | 647 | 16,021 | 235 | 684 | 381 | 529 | 1,077 | 659 | 1,025 | 527 |
| lseu | 89 | 79 | 81 | 73 | 103 | 85 | 55 | 60 | 91 | 71 |
| mas74 | 4,449,609 | 4,122,173 | 4,753,425 | 4,162,462 | 4,152,510 | 4,156,326 | 4,734,180 | 4,152,510 | 4,156,014 | 4,753,425 |
| mas76 | 1,703,949 | 1,308,977 | 1,403,103 | 1,671,361 | 1,423,877 | 1,203,693 | 1,284,023 | 1,475,395 | 1,403,103 | 1,284,023 |
| misc03 | 1,967 | 2,257 | 1,935 | 2,777 | 2,467 | 2,035 | 2,035 | 2,729 | 2,887 | 2,941 |
| misc06 | 7 | 5 | 10 | 14 | 5 | 7 | 7 | 5 | 7 | 7 |
| misc07 | 113,107 | 98,073 | 92,913 | 113,527 | 84,981 | 115,257 | 89,613 | 93,407 | 95,355 | 86,377 |
| mitre | 16 | 0 | 10 | 0 | 17 | 10 | 6 | 11 | 0 | 0 |
| mod008 | 407 | 265 | 309 | 407 | 265 | 201 | 407 | 265 | 201 | 407 |

Table A.35: Batch learned branching number of nodes: detailed optimization results for the considered MIPLIB problems and for each random seed (1/2). The table reports the number of nodes processed by B&B before the optimization terminates. To determine if, at termination, the problem is solved to optimality, check the corresponding table reporting the optimization times (a value of 7,200 indicates that the problem has not been solved within the provided time limit).

| | Random seed | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Prob. name | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| mod010 | 13 | 12 | 7 | 25 | 23 | 13 | 13 | 17 | 13 | 23 |
| mod011 | 3,759 | 3,361 | 4,359 | 4,143 | 3,735 | 4,181 | 4,079 | 4,040 | 3,821 | 3,705 |
| modglob | 84 | 84 | 92 | 90 | 86 | 84 | 84 | 82 | 87 | 86 |
| nw04 | 41 | 71 | 147 | 65 | 29 | 59 | 59 | 53 | 55 | 95 |
| opt1217 | 1,286,540 | 1,330,458 | 1,291,136 | 1,353,453 | 1,323,995 | 1,355,451 | 1,328,831 | 1,357,157 | 1,269,414 | 1,325,864 |
| p0033 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| p0201 | 221 | 203 | 187 | 43 | 65 | 289 | 309 | 69 | 251 | 267 |
| p0282 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| p0548 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| p2756 | 190 | 150 | 211 | 40 | 251 | 80 | 250 | 160 | 58 | 340 |
| pk1 | 1,441,723 | 1,081,771 | 1,585,061 | 1,127,951 | 1,352,205 | 1,072,369 | 1,502,559 | 1,373,567 | 1,299,899 | 1,670,309 |
| pp08a | 1,089 | 1,285 | 1,089 | 1,089 | 954 | 1,009 | 1,285 | 1,089 | 1,115 | 1,285 |
| pp08aCUTS | 507 | 507 | 507 | 543 | 543 | 507 | 507 | 507 | 543 | 507 |
| qiu | 73,325 | 88,657 | 41,299 | 74,583 | 55,921 | 66,277 | 49,719 | 46,877 | 29,269 | 45,693 |
| rentacar | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |
| rgn | 0 | 470 | 0 | 449 | 0 | 83 | 181 | 0 | 0 | 0 |
| set1ch | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 |
| stein27 | 4,395 | 4,393 | 4,277 | 4,417 | 4,661 | 4,441 | 4,459 | 4,255 | 4,027 | 4,235 |
| stein45 | 60,437 | 58,365 | 64,285 | 60,873 | 59,815 | 54,821 | 51,319 | 55,515 | 54,819 | 67,469 |
| tr12-30 | 418,734 | 400,744 | 412,018 | 451,465 | 419,794 | 430,662 | 423,467 | 427,947 | 442,390 | 424,902 |
| vpm1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| vpm2 | 1,623 | 1,987 | 2,091 | 2,091 | 1,577 | 1,623 | 1,795 | 1,565 | 2,091 | 2,091 |

Table A.36: Batch learned branching number of nodes: detailed optimization results for the considered MIPLIB problems and for each random seed (2/2). The table reports the number of nodes processed by B&B before the optimization terminates. To determine if, at termination, the problem is solved to optimality, check the corresponding table reporting the optimization times (a value of 7,200 indicates that the problem has not been solved within the provided time limit).

| | Random seed | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Prob. name | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 10teams | 340 | 57 | 427 | 0 | 171 | 193 | 23 | 281 | 196 | 25 |
| aflow30a | 45,355 | 29,549 | 33,811 | 32,549 | 22,833 | 27,151 | 29,323 | 30,629 | 24,069 | 33,373 |
| aflow40b | 182,707 | 143,386 | 161,169 | 150,425 | 184,727 | 160,406 | 149,374 | 196,470 | 195,216 | 155,724 |
| air03 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| air04 | 76,363 | 38,489 | 64,197 | 59,956 | 60,136 | 60,589 | 60,871 | 46,286 | 50,696 | 47,845 |
| air05 | 73,665 | 69,165 | 88,098 | 90,039 | 94,832 | 84,977 | 87,389 | 96,945 | 79,511 | 83,443 |
| cap6000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| dcmulti | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 |
| egout | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| fiber | 381 | 3,617 | 213 | 427 | 725 | 311 | 453 | 401 | 381 | 543 |
| fixnet6 | 1,017 | 2,153 | 2,453 | 153 | 2,027 | 667 | 1,172 | 1,172 | 2,085 | 331 |
| harp2 | 653,597 | 608,991 | 608,751 | 635,776 | 628,475 | 632,974 | 608,072 | 589,834 | 624,265 | 624,637 |
| khb05250 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| l152lav | 272 | 279 | 239 | 167 | 213 | 217 | 245 | 209 | 289 | 299 |
| lseu | 137 | 111 | 85 | 95 | 119 | 125 | 87 | 145 | 127 | 129 |
| mas74 | 5,690,446 | 5,564,087 | 5,625,559 | 5,780,219 | 5,558,741 | 5,843,693 | 5,781,819 | 5,696,710 | 5,629,088 | 5,696,055 |
| mas76 | 1,896,737 | 1,730,777 | 1,965,413 | 1,965,413 | 2,584,783 | 2,584,805 | 1,965,423 | 1,596,987 | 1,965,423 | 1,576,739 |
| misc03 | 259 | 273 | 309 | 277 | 267 | 305 | 297 | 299 | 299 | 289 |
| misc06 | 7 | 7 | 10 | 9 | 7 | 7 | 7 | 7 | 7 | 7 |
| misc07 | 10,823 | 10,563 | 11,101 | 9,827 | 11,265 | 9,791 | 10,811 | 10,481 | 11,263 | 11,193 |
| mitre | 30 | 0 | 9 | 0 | 10 | 14 | 11 | 17 | 0 | 0 |
| mod008 | 461 | 363 | 363 | 461 | 363 | 415 | 465 | 363 | 415 | 465 |

Table A.37: Online learning branching number of nodes: detailed optimization results for the considered MIPLIB problems and for each random seed (1/2). The table reports the number of nodes processed by B&B before the optimization terminates. To determine if, at termination, the problem is solved to optimality, check the corresponding table reporting the optimization times (a value of 7,200 indicates that the problem has not been solved within the provided time limit).

| | Random seed | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Prob. name | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| mod010 | 10 | 17 | 9 | 21 | 11 | 15 | 13 | 23 | 19 | 10 |
| mod011 | 6,553 | 4,699 | 6,799 | 6,745 | 8,169 | 7,921 | 8,613 | 6,827 | 7,837 | 7,835 |
| modglob | 68 | 68 | 83 | 82 | 79 | 68 | 68 | 68 | 82 | 83 |
| nw04 | 1,047 | 1,237 | 813 | 901 | 383 | 785 | 1,169 | 1,033 | 423 | 803 |
| opt1217 | 3,033,528 | 2,824,800 | 3,059,437 | 3,050,524 | 2,832,733 | 2,796,839 | 2,907,355 | 2,839,749 | 2,815,289 | 2,643,170 |
| p0033 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| p0201 | 951 | 601 | 751 | 295 | 197 | 775 | 619 | 345 | 579 | 817 |
| p0282 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| p0548 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| p2756 | 191,897 | 63,551 | 134,821 | 266,457 | 163,751 | 160,439 | 138,103 | 181,331 | 183,739 | 309,367 |
| pk1 | 807,985 | 741,927 | 906,195 | 1,025,529 | 911,525 | 921,561 | 1,002,973 | 1,009,643 | 904,315 | 833,593 |
| pp08a | 1,015 | 1,013 | 1,007 | 1,011 | 1,089 | 1,129 | 999 | 1,144 | 1,135 | 999 |
| pp08aCUTS | 1,039 | 1,205 | 1,231 | 1,039 | 1,039 | 1,231 | 1,205 | 1,207 | 1,379 | 1,219 |
| qiu | 289,237 | 355,985 | 376,483 | 377,571 | 346,415 | 205,087 | 313,473 | 411,161 | 325,727 | 337,235 |
| rentacar | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| rgn | 0 | 577 | 0 | 825 | 0 | 53 | 49 | 0 | 0 | 0 |
| set1ch | 19 | 19 | 19 | 19 | 19 | 19 | 19 | 19 | 19 | 19 |
| stein27 | 4,389 | 4,215 | 4,155 | 3,777 | 3,939 | 4,163 | 4,241 | 3,851 | 4,115 | 4,061 |
| stein45 | 70,329 | 73,195 | 77,819 | 72,435 | 68,059 | 82,213 | 77,583 | 74,419 | 72,401 | 73,219 |
| tr12-30 | 1,075,950 | 1,023,556 | 1,116,385 | 1,081,820 | 1,088,009 | 1,123,411 | 1,098,694 | 1,064,005 | 991,731 | 1,041,402 |
| vpm1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| vpm2 | 1,800 | 923 | 1,570 | 957 | 917 | 1,800 | 861 | 1,800 | 1,570 | 1,570 |

Table A.38: Online learning branching number of nodes: detailed optimization results for the considered MIPLIB problems and for each random seed (2/2). The table reports the number of nodes processed by B&B before the optimization terminates. To determine if, at termination, the problem is solved to optimality, check the corresponding table reporting the optimization times (a value of 7,200 indicates that the problem has not been solved within the provided time limit).

| | Random seed | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Prob. name | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 10teams | 230 | 57 | 486 | 0 | 218 | 204 | 23 | 303 | 201 | 25 |
| aflow30a | 38,507 | 25,194 | 32,429 | 33,935 | 27,499 | 32,741 | 35,479 | 30,557 | 27,159 | 38,381 |
| aflow40b | 154,289 | 146,389 | 173,073 | 167,729 | 169,425 | 162,006 | 146,245 | 182,755 | 168,344 | 143,815 |
| air03 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| air04 | 75,604 | 55,827 | 62,701 | 51,585 | 53,982 | 45,935 | 63,091 | 41,049 | 38,099 | 47,101 |
| air05 | 80,145 | 72,591 | 72,259 | 89,907 | 79,885 | 86,009 | 75,797 | 85,245 | 80,909 | 85,611 |
| cap6000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| dcmulti | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 |
| egout | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| fiber | 425 | 3,611 | 216 | 373 | 753 | 313 | 405 | 357 | 395 | 483 |
| fixnet6 | 229 | 1,097 | 1,679 | 321 | 1,097 | 1,015 | 1,097 | 1,099 | 1,099 | 243 |
| harp2 | 660,094 | 861,048 | 853,743 | 661,491 | 809,722 | 632,217 | 646,046 | 619,399 | 637,294 | 655,857 |
| khb05250 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| l152lav | 273 | 276 | 273 | 161 | 213 | 227 | 247 | 211 | 297 | 283 |
| lseu | 131 | 87 | 87 | 95 | 119 | 125 | 115 | 160 | 118 | 129 |
| mas74 | 5,602,185 | 5,599,212 | 5,601,323 | 5,601,941 | 5,610,080 | 5,590,050 | 5,593,556 | 5,597,803 | 5,598,497 | 5,588,481 |
| mas76 | 2,083,793 | 2,921,749 | 2,059,101 | 2,059,101 | 2,124,669 | 2,881,594 | 2,881,594 | 2,921,749 | 2,881,594 | 3,040,381 |
| misc03 | 259 | 277 | 297 | 283 | 279 | 305 | 299 | 299 | 299 | 297 |
| misc06 | 7 | 7 | 10 | 9 | 7 | 7 | 7 | 7 | 7 | 7 |
| misc07 | 10,711 | 10,751 | 10,635 | 10,061 | 11,149 | 10,301 | 11,487 | 10,707 | 10,833 | 10,955 |
| mitre | 30 | 0 | 9 | 0 | 10 | 14 | 11 | 17 | 0 | 0 |
| mod008 | 467 | 367 | 367 | 467 | 367 | 359 | 489 | 367 | 359 | 489 |

Table A.39: Online perpetual learning branching number of nodes: detailed optimization results for the considered MIPLIB problems and for each random seed (1/2). The table reports the number of nodes processed by B&B before the optimization terminates. To determine if, at termination, the problem is solved to optimality, check the corresponding table reporting the optimization times (a value of 7,200 indicates that the problem has not been solved within the provided time limit).

| | Random seed | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Prob. name | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| mod010 | 10 | 17 | 9 | 21 | 11 | 15 | 13 | 23 | 19 | 10 |
| mod011 | 5,601 | 3,413 | 3,741 | 4,921 | 5,325 | 4,477 | 5,345 | 5,878 | 4,645 | 4,355 |
| modglob | 68 | 68 | 85 | 82 | 70 | 68 | 68 | 68 | 82 | 85 |
| nw04 | 1,125 | 1,119 | 993 | 879 | 383 | 845 | 1,133 | 981 | 423 | 707 |
| opt1217 | 3,003,949 | 3,102,807 | 3,000,605 | 2,838,296 | 3,050,352 | 2,899,494 | 3,080,053 | 3,043,389 | 2,963,036 | 2,779,932 |
| p0033 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| p0201 | 951 | 745 | 925 | 353 | 197 | 747 | 623 | 279 | 733 | 805 |
| p0282 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| p0548 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| p2756 | 139,735 | 123,383 | 150,563 | 158,883 | 300,005 | 108,173 | 330,323 | 91,649 | 104,695 | 157,591 |
| pk1 | 791,179 | 661,943 | 727,101 | 693,979 | 725,943 | 717,445 | 692,993 | 753,555 | 677,729 | 715,343 |
| pp08a | 1,073 | 1,031 | 1,105 | 1,112 | 1,167 | 1,175 | 1,009 | 1,103 | 1,119 | 985 |
| pp08aCUTS | 919 | 1,163 | 1,147 | 919 | 919 | 1,147 | 1,163 | 1,147 | 1,191 | 1,191 |
| qiu | 129,805 | 122,269 | 139,421 | 151,793 | 132,747 | 153,049 | 131,425 | 149,727 | 147,079 | 143,119 |
| rentacar | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| rgn | 0 | 609 | 0 | 753 | 0 | 53 | 49 | 0 | 0 | 0 |
| set1ch | 19 | 19 | 19 | 19 | 19 | 19 | 19 | 19 | 19 | 19 |
| stein27 | 4,123 | 4,041 | 3,955 | 3,805 | 4,059 | 4,277 | 4,127 | 3,779 | 4,203 | 3,847 |
| stein45 | 69,257 | 67,031 | 68,647 | 65,203 | 64,095 | 68,931 | 64,859 | 67,389 | 63,315 | 67,999 |
| tr12-30 | 1,065,077 | 1,090,952 | 1,164,260 | 1,148,385 | 1,023,133 | 1,066,605 | 1,096,171 | 1,030,625 | 1,103,400 | 1,102,236 |
| vpm1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| vpm2 | 979 | 1,380 | 2,471 | 2,471 | 1,327 | 979 | 845 | 979 | 2,471 | 2,471 |

Table A.40: Online perpetual learning branching number of nodes: detailed optimization results for the considered MIPLIB problems and for each random seed (2/2). The table reports the number of nodes processed by B&B before the optimization terminates. To determine if, at termination, the problem is solved to optimality, check the corresponding table reporting the optimization times (a value of 7,200 indicates that the problem has not been solved within the provided time limit).

# Appendix B

# Machine learning for parallel branch-and-bound: appendix

## B.1 Detailed feature importances results

This appendix reports the detailed feature importances and costs of omission for all variables developed for this application. The results are given in Tables B.1, B.2, B.3, and B.4 when the considered output is the number of nodes and in Tables B.5, B.6, B.7, and B.8 when the considered output is the logarithm of the number of nodes. For an analysis of the tables, we refer the reader to Section 5.6.2 of the main document.

Output: #nodes

| | | $RE_i = 0$ | | $0 < RE_i \leq 50$ | | $50 < RE_i \leq 100$ | | $100 < RE_i \leq 150$ | | $150 < RE_i \leq +\infty$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| # | FI | MRE | COO | MRE | COO | MRE | COO | MRE | COO | MRE | COO |
| 66 | 0.2020 | 0.00 | 0 | 100.52 | 100 | 171.40 | 100 | 348.15 | 100 | 992.00 | 100 |
| 63 | 0.1388 | 137.42 | 100 | 40.19 | 24 | 89.48 | 19 | 153.77 | 14 | 731.56 | -3 |
| 14 | 0.0705 | 0.00 | 0 | 25.01 | 5 | 71.46 | 1 | 126.78 | 2 | 721.52 | -6 |
| 18 | 0.0629 | 0.00 | 0 | 26.16 | 6 | 73.29 | 3 | 131.85 | 4 | 736.73 | -1 |
| 62 | 0.0493 | 0.00 | 0 | 25.53 | 5 | 70.98 | 1 | 127.22 | 2 | 730.60 | -3 |
| 17 | 0.0410 | 0.00 | 0 | 24.62 | 4 | 71.32 | 1 | 126.28 | 1 | 730.19 | -3 |
| 1 | 0.0325 | 0.00 | 0 | 24.82 | 4 | 71.43 | 1 | 127.47 | 2 | 741.18 | 1 |
| 61 | 0.0297 | 0.00 | 0 | 26.12 | 6 | 74.54 | 4 | 135.67 | 6 | 736.86 | -0 |
| 16 | 0.0277 | 0.00 | 0 | 24.56 | 4 | 72.22 | 2 | 125.86 | 1 | 726.41 | -5 |
| 60 | 0.0206 | 0.00 | 0 | 25.01 | 5 | 72.23 | 2 | 130.13 | 3 | 740.92 | 1 |
| 15 | 0.0191 | 0.00 | 0 | 24.87 | 4 | 71.68 | 1 | 128.50 | 2 | 750.32 | 5 |
| 59 | 0.0184 | 0.00 | 0 | 24.52 | 4 | 70.62 | 0 | 128.64 | 3 | 751.62 | 5 |
| 65 | 0.0184 | 0.17 | 0 | 25.03 | 5 | 72.51 | 2 | 128.59 | 3 | 738.93 | 0 |
| 64 | 0.0174 | 0.49 | 0 | 26.67 | 7 | 72.10 | 2 | 129.30 | 3 | 736.46 | -1 |
| 34 | 0.0169 | 0.00 | 0 | 24.13 | 3 | 71.49 | 1 | 125.12 | 1 | 738.19 | 0 |
| 31 | 0.0167 | 0.00 | 0 | 24.62 | 4 | 72.73 | 2 | 128.19 | 2 | 735.42 | -1 |
| 38 | 0.0162 | 0.00 | 0 | 24.56 | 4 | 71.49 | 1 | 127.95 | 2 | 732.52 | -2 |

Table B.1: Feature importances (1/4) as computed by the random forests algorithm and normalized costs of omission when the considered output is the number of nodes. The features are sorted in descending order of the feature importances computed by the random forests. The first column indicates the feature number. ''FI' represents the feature importance computed by the random forests for the given feature. The columns labeled 'MRE' and 'COO' represent the mean relative errors (in %) and the corresponding normalized costs of omission achieved when the corresponding feature is removed from the data. Additionally, several cases are considered to compute the MREs and COOs, as detailed in Section 5.6.1.

Output: #nodes

| | | $RE_i = 0$ | | $0 < RE_i \leq 50$ | | $50 < RE_i \leq 100$ | | $100 < RE_i \leq 150$ | | $150 < RE_i \leq +\infty$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| # | FI | MRE | COO | MRE | COO | MRE | COO | MRE | COO | MRE | COO |
| 40 | 0.0155 | 0.00 | 0 | 26.32 | 6 | 73.55 | 3 | 135.58 | 6 | 755.62 | 7 |
| 46 | 0.0153 | 0.00 | 0 | 24.65 | 4 | 73.00 | 3 | 130.72 | 3 | 744.47 | 3 |
| 12 | 0.0143 | 0.00 | 0 | 24.63 | 4 | 71.39 | 1 | 127.91 | 2 | 745.68 | 3 |
| 5 | 0.0139 | 0.00 | 0 | 24.41 | 4 | 71.44 | 1 | 128.23 | 2 | 738.33 | 0 |
| 9 | 0.0128 | 0.00 | 0 | 24.24 | 4 | 72.01 | 2 | 128.07 | 2 | 740.63 | 1 |
| 13 | 0.0106 | 0.00 | 0 | 24.20 | 4 | 71.07 | 1 | 126.80 | 2 | 741.37 | 1 |
| 8 | 0.0104 | 0.00 | 0 | 24.28 | 4 | 73.02 | 3 | 128.12 | 2 | 740.67 | 1 |
| 2 | 0.0101 | 0.00 | 0 | 24.56 | 4 | 71.82 | 2 | 129.53 | 3 | 746.00 | 3 |
| 7 | 0.0099 | 0.00 | 0 | 24.68 | 4 | 72.25 | 2 | 127.82 | 2 | 754.33 | 6 |
| 11 | 0.0098 | 0.00 | 0 | 24.11 | 3 | 70.62 | 0 | 127.66 | 2 | 747.29 | 4 |
| 35 | 0.0097 | 0.00 | 0 | 24.29 | 4 | 71.50 | 1 | 127.99 | 2 | 732.77 | -2 |
| 3 | 0.0084 | 0.00 | 0 | 24.40 | 4 | 71.74 | 2 | 129.71 | 3 | 738.94 | 0 |
| 26 | 0.0076 | 0.00 | 0 | 24.40 | 4 | 71.95 | 2 | 124.44 | 1 | 737.89 | -0 |
| 4 | 0.0069 | 0.00 | 0 | 24.12 | 3 | 71.58 | 1 | 127.13 | 2 | 740.93 | 1 |
| 6 | 0.0064 | 0.00 | 0 | 24.61 | 4 | 71.26 | 1 | 128.02 | 2 | 726.78 | -4 |
| 33 | 0.0060 | 0.00 | 0 | 24.62 | 4 | 71.81 | 2 | 129.01 | 3 | 747.31 | 4 |
| 10 | 0.0053 | 0.00 | 0 | 24.24 | 4 | 71.16 | 1 | 129.07 | 3 | 743.51 | 2 |

Table B.2: Feature importances (2/4) as computed by the random forests algorithm and normalized costs of omission when the considered output is the number of nodes. The features are sorted in descending order of the feature importances computed by the random forests. The first column indicates the feature number. ''FI' represents the feature importance computed by the random forests for the given feature. The columns labeled 'MRE' and 'COO' represent the mean relative errors (in %) and the corresponding normalized costs of omission achieved when the corresponding feature is removed from the data. Additionally, several cases are considered to compute the MREs and COOs, as detailed in Section 5.6.1.

Output: #nodes

| | | $RE_i = 0$ | | $0 < RE_i \leq 50$ | | $50 < RE_i \leq 100$ | | $100 < RE_i \leq 150$ | | $150 < RE_i \leq +\infty$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| # | FI | MRE | COO | MRE | COO | MRE | COO | MRE | COO | MRE | COO |
| 19 | 0.0041 | 0.00 | 0 | 24.04 | 3 | 71.51 | 1 | 127.04 | 2 | 735.96 | -1 |
| 58 | 0.0037 | 0.00 | 0 | 24.01 | 3 | 71.38 | 1 | 126.67 | 2 | 729.51 | -3 |
| 32 | 0.0035 | 0.00 | 0 | 24.36 | 4 | 71.40 | 1 | 128.90 | 3 | 735.60 | -1 |
| 20 | 0.0031 | 0.00 | 0 | 24.42 | 4 | 71.55 | 1 | 125.56 | 1 | 739.98 | 1 |
| 43 | 0.0028 | 0.00 | 0 | 24.17 | 4 | 71.57 | 1 | 128.28 | 2 | 739.56 | 1 |
| 44 | 0.0027 | 0.00 | 0 | 24.53 | 4 | 74.45 | 4 | 129.75 | 3 | 742.78 | 2 |
| 52 | 0.0021 | 0.00 | 0 | 24.50 | 4 | 72.37 | 2 | 128.19 | 2 | 735.65 | -1 |
| 29 | 0.0012 | 0.00 | 0 | 24.58 | 4 | 72.37 | 2 | 128.68 | 3 | 738.24 | 0 |
| 28 | 0.0011 | 0.00 | 0 | 24.34 | 4 | 71.89 | 2 | 128.03 | 2 | 732.96 | -2 |
| 42 | 0.0009 | 0.00 | 0 | 24.58 | 4 | 73.08 | 3 | 131.07 | 4 | 752.62 | 6 |
| 49 | 0.0008 | 0.00 | 0 | 24.62 | 4 | 71.59 | 1 | 130.50 | 3 | 738.91 | 0 |
| 45 | 0.0006 | 0.00 | 0 | 24.27 | 4 | 70.65 | 0 | 126.63 | 2 | 724.21 | -5 |
| 24 | 0.0005 | 0.00 | 0 | 24.16 | 3 | 71.99 | 2 | 126.54 | 2 | 747.04 | 4 |
| 41 | 0.0004 | 0.00 | 0 | 24.57 | 4 | 71.55 | 1 | 128.91 | 3 | 743.03 | 2 |
| 39 | 0.0004 | 0.00 | 0 | 24.72 | 4 | 72.26 | 2 | 127.84 | 2 | 748.47 | 4 |
| 37 | 0.0002 | 0.00 | 0 | 24.43 | 4 | 71.66 | 1 | 127.12 | 2 | 748.89 | 4 |

Table B.3: Feature importances (3/4) as computed by the random forests algorithm and normalized costs of omission when the considered output is the number of nodes. The features are sorted in descending order of the feature importances computed by the random forests. The first column indicates the feature number. ''FI' represents the feature importance computed by the random forests for the given feature. The columns labeled 'MRE' and 'COO' represent the mean relative errors (in %) and the corresponding normalized costs of omission achieved when the corresponding feature is removed from the data. Additionally, several cases are considered to compute the MREs and COOs, as detailed in Section 5.6.1.

Output: #nodes

| | | $RE_i = 0$ | | $0 < RE_i \leq 50$ | | $50 < RE_i \leq 100$ | | $100 < RE_i \leq 150$ | | $150 < RE_i \leq +\infty$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| # | FI | MRE | COO | MRE | COO | MRE | COO | MRE | COO | MRE | COO |
| 36 | 0.0002 | 0.00 | 0 | 24.16 | 4 | 71.59 | 1 | 129.91 | 3 | 739.42 | 1 |
| 22 | 0.0001 | 0.00 | 0 | 24.33 | 4 | 71.86 | 2 | 127.50 | 2 | 744.12 | 2 |
| 55 | 0.0001 | 0.00 | 0 | 24.00 | 3 | 71.19 | 1 | 127.45 | 2 | 747.74 | 4 |
| 30 | 0.0001 | 0.00 | 0 | 24.52 | 4 | 72.08 | 2 | 129.60 | 3 | 731.41 | -3 |
| 48 | 0.0001 | 0.00 | 0 | 24.69 | 4 | 71.95 | 2 | 127.72 | 2 | 747.20 | 4 |
| 51 | 0.0000 | 0.00 | 0 | 24.35 | 4 | 71.77 | 2 | 127.39 | 2 | 751.37 | 5 |
| 27 | 0.0000 | 0.00 | 0 | 24.38 | 4 | 71.95 | 2 | 126.11 | 1 | 737.64 | -0 |
| 21 | 0.0000 | 0.00 | 0 | 24.27 | 4 | 70.28 | 0 | 125.53 | 1 | 737.86 | -0 |
| 47 | 0.0000 | 0.00 | 0 | 24.42 | 4 | 71.72 | 1 | 129.28 | 3 | 734.04 | -2 |
| 23 | 0.0000 | 0.00 | 0 | 24.28 | 4 | 71.69 | 1 | 128.79 | 3 | 744.49 | 3 |
| 25 | 0.0000 | 0.00 | 0 | 24.51 | 4 | 72.38 | 2 | 130.41 | 3 | 749.99 | 5 |
| 50 | 0.0000 | 0.00 | 0 | 24.32 | 4 | 71.21 | 1 | 127.00 | 2 | 743.11 | 2 |
| 53 | 0.0000 | 0.00 | 0 | 24.34 | 4 | 70.50 | 0 | 125.87 | 1 | 736.29 | -1 |
| 54 | 0.0000 | 0.00 | 0 | 24.43 | 4 | 71.21 | 1 | 127.36 | 2 | 739.12 | 0 |
| 56 | 0.0000 | 0.00 | 0 | 24.57 | 4 | 71.86 | 2 | 127.88 | 2 | 731.44 | -3 |
| 57 | 0.0000 | 0.00 | 0 | 24.16 | 4 | 71.47 | 1 | 128.15 | 2 | 754.64 | 7 |

Table B.4: Feature importances (4/4) as computed by the random forests algorithm and normalized costs of omission when the considered output is the number of nodes. The features are sorted in descending order of the feature importances computed by the random forests. The first column indicates the feature number. ''FI' represents the feature importance computed by the random forests for the given feature. The columns labeled 'MRE' and 'COO' represent the mean relative errors (in %) and the corresponding normalized costs of omission achieved when the corresponding feature is removed from the data. Additionally, several cases are considered to compute the MREs and COOs, as detailed in Section 5.6.1.

Output: log #nodes

| | | $RE_i = 0$ | | $0 < RE_i \leq 5$ | | $5 < RE_i \leq 10$ | | $10 < RE_i \leq 15$ | | $15 < RE_i \leq +\infty$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| # | FI | MRE | COO | MRE | COO | MRE | COO | MRE | COO | MRE | COO |
| 63 | 0.9001 | 0.12 | 24 | 3.30 | 79 | 7.45 | 100 | 11.82 | -334 | 19.04 | -807 |
| 66 | 0.0577 | 0.00 | 0 | 3.59 | 100 | 6.97 | -127 | 11.37 | -761 | 18.83 | -944 |
| 62 | 0.0199 | 0.00 | 0 | 2.50 | 21 | 7.07 | -82 | 11.94 | -222 | 19.86 | -281 |
| 18 | 0.0035 | 0.00 | 0 | 2.51 | 22 | 7.06 | -84 | 11.84 | -312 | 19.78 | -334 |
| 65 | 0.0019 | 0.10 | 20 | 2.56 | 26 | 7.12 | -58 | 11.98 | -183 | 20.01 | -187 |
| 61 | 0.0015 | 0.00 | 0 | 2.48 | 20 | 7.15 | -44 | 11.93 | -225 | 19.88 | -271 |
| 40 | 0.0014 | 0.00 | 0 | 2.45 | 17 | 7.08 | -77 | 11.88 | -277 | 19.71 | -380 |
| 14 | 0.0014 | 0.00 | 0 | 2.47 | 19 | 7.08 | -75 | 11.88 | -274 | 19.77 | -344 |
| 17 | 0.0012 | 0.00 | 0 | 2.42 | 15 | 7.09 | -71 | 11.93 | -226 | 19.92 | -243 |
| 59 | 0.0012 | 0.00 | 0 | 2.43 | 16 | 7.15 | -43 | 12.06 | -100 | 20.15 | -100 |
| 1 | 0.0011 | 0.00 | 0 | 2.41 | 15 | 7.09 | -72 | 11.90 | -252 | 19.87 | -279 |
| 15 | 0.0011 | 0.00 | 0 | 2.41 | 15 | 7.11 | -62 | 11.92 | -235 | 19.84 | -297 |
| 16 | 0.0008 | 0.00 | 0 | 2.40 | 14 | 7.08 | -75 | 12.02 | -139 | 19.89 | -261 |
| 60 | 0.0007 | 0.00 | 0 | 2.37 | 12 | 7.09 | -71 | 11.94 | -222 | 19.96 | -220 |
| 64 | 0.0006 | 0.50 | 100 | 2.79 | 42 | 7.30 | 31 | 12.00 | -163 | 19.95 | -224 |
| 7 | 0.0006 | 0.00 | 0 | 2.40 | 14 | 7.12 | -56 | 11.96 | -198 | 19.81 | -315 |
| 38 | 0.0004 | 0.00 | 0 | 2.37 | 12 | 7.11 | -63 | 11.99 | -175 | 19.85 | -289 |

Table B.5: Feature importances (1/4) as computed by the random forests algorithm and normalized costs of omission when the considered output is the logarithm of the number of nodes. The features are sorted in descending order of the feature importances computed by the random forests. The first column indicates the feature number. ''FI' represents the feature importance computed by the random forests for the given feature. The columns labeled 'MRE' and 'COO' represent the mean relative errors (in %) and the corresponding normalized costs of omission achieved when the corresponding feature is removed from the data. Additionally, several cases are considered to compute the MREs and COOs, as detailed in Section 5.6.1.

Output: log #nodes

| | | $RE_i = 0$ | | $0 < RE_i \leq 5$ | | $5 < RE_i \leq 10$ | | $10 < RE_i \leq 15$ | | $15 < RE_i \leq +\infty$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| # | FI | MRE | COO | MRE | COO | MRE | COO | MRE | COO | MRE | COO |
| 11 | 0.0004 | 0.00 | 0 | 2.39 | 13 | 7.13 | -51 | 12.01 | -154 | 19.88 | -273 |
| 5 | 0.0004 | 0.00 | 0 | 2.40 | 14 | 7.09 | -72 | 11.95 | -209 | 19.85 | -291 |
| 35 | 0.0003 | 0.00 | 0 | 2.37 | 12 | 7.08 | -78 | 11.97 | -190 | 19.92 | -247 |
| 13 | 0.0003 | 0.00 | 0 | 2.38 | 12 | 7.08 | -77 | 11.92 | -241 | 19.95 | -224 |
| 12 | 0.0003 | 0.00 | 0 | 2.38 | 13 | 7.09 | -70 | 11.93 | -228 | 19.82 | -307 |
| 3 | 0.0003 | 0.00 | 0 | 2.35 | 11 | 7.09 | -72 | 11.93 | -227 | 19.88 | -269 |
| 8 | 0.0003 | 0.00 | 0 | 2.38 | 12 | 7.12 | -58 | 11.99 | -174 | 19.87 | -274 |
| 9 | 0.0003 | 0.00 | 0 | 2.38 | 12 | 7.09 | -71 | 11.91 | -249 | 19.77 | -343 |
| 34 | 0.0003 | 0.00 | 0 | 2.37 | 12 | 7.10 | -66 | 11.94 | -220 | 19.92 | -245 |
| 31 | 0.0003 | 0.00 | 0 | 2.40 | 14 | 7.14 | -49 | 12.00 | -156 | 19.81 | -314 |
| 2 | 0.0002 | 0.00 | 0 | 2.39 | 13 | 7.07 | -82 | 12.00 | -164 | 19.91 | -252 |
| 58 | 0.0002 | 0.00 | 0 | 2.38 | 13 | 7.13 | -50 | 12.00 | -158 | 19.93 | -239 |
| 4 | 0.0002 | 0.00 | 0 | 2.39 | 13 | 7.08 | -75 | 11.95 | -210 | 19.93 | -236 |
| 52 | 0.0002 | 0.00 | 0 | 2.39 | 13 | 7.09 | -69 | 11.99 | -175 | 19.83 | -301 |
| 46 | 0.0002 | 0.00 | 0 | 2.36 | 11 | 7.12 | -57 | 11.97 | -191 | 20.00 | -194 |
| 44 | 0.0001 | 0.00 | 0 | 2.39 | 13 | 7.07 | -80 | 11.90 | -253 | 19.83 | -300 |
| 19 | 0.0001 | 0.00 | 0 | 2.38 | 13 | 7.11 | -63 | 11.96 | -201 | 19.92 | -244 |

Table B.6: Feature importances (2/4) as computed by the random forests algorithm and normalized costs of omission when the considered output is the logarithm of the number of nodes. The features are sorted in descending order of the feature importances computed by the random forests. The first column indicates the feature number. ''FI' represents the feature importance computed by the random forests for the given feature. The columns labeled 'MRE' and 'COO' represent the mean relative errors (in %) and the corresponding normalized costs of omission achieved when the corresponding feature is removed from the data. Additionally, several cases are considered to compute the MREs and COOs, as detailed in Section 5.6.1.

Output: log #nodes

| | | $RE_i = 0$ | | $0 < RE_i \leq 5$ | | $5 < RE_i \leq 10$ | | $10 < RE_i \leq 15$ | | $15 < RE_i \leq +\infty$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| # | FI | MRE | COO | MRE | COO | MRE | COO | MRE | COO | MRE | COO |
| 43 | 0.0001 | 0.00 | 0 | 2.37 | 12 | 7.12 | -58 | 11.99 | -169 | 20.04 | -171 |
| 10 | 0.0001 | 0.00 | 0 | 2.36 | 11 | 7.11 | -61 | 11.96 | -198 | 19.88 | -269 |
| 20 | 0.0001 | 0.00 | 0 | 2.39 | 13 | 7.11 | -60 | 11.98 | -176 | 19.93 | -237 |
| 33 | 0.0001 | 0.00 | 0 | 2.36 | 11 | 7.11 | -62 | 12.02 | -140 | 19.93 | -236 |
| 42 | 0.0001 | 0.00 | 0 | 2.37 | 11 | 7.11 | -60 | 12.01 | -152 | 19.92 | -243 |
| 6 | 0.0001 | 0.00 | 0 | 2.39 | 13 | 7.12 | -56 | 12.02 | -142 | 20.00 | -192 |
| 32 | 0.0001 | 0.00 | 0 | 2.36 | 11 | 7.10 | -69 | 11.97 | -192 | 19.91 | -253 |
| 37 | 0.0000 | 0.00 | 0 | 2.38 | 12 | 7.13 | -50 | 12.00 | -156 | 19.93 | -237 |
| 45 | 0.0000 | 0.00 | 0 | 2.37 | 12 | 7.09 | -73 | 11.93 | -228 | 19.82 | -312 |
| 39 | 0.0000 | 0.00 | 0 | 2.39 | 13 | 7.11 | -61 | 11.96 | -197 | 19.89 | -265 |
| 26 | 0.0000 | 0.00 | 0 | 2.38 | 13 | 7.12 | -59 | 11.97 | -190 | 19.95 | -223 |
| 36 | 0.0000 | 0.00 | 0 | 2.37 | 12 | 7.12 | -57 | 11.97 | -193 | 19.86 | -285 |
| 41 | 0.0000 | 0.00 | 0 | 2.38 | 13 | 7.12 | -55 | 11.92 | -234 | 19.89 | -262 |
| 24 | 0.0000 | 0.00 | 0 | 2.39 | 13 | 7.11 | -60 | 11.99 | -175 | 19.88 | -271 |
| 29 | 0.0000 | 0.00 | 0 | 2.40 | 14 | 7.10 | -65 | 11.99 | -166 | 19.95 | -227 |
| 49 | 0.0000 | 0.00 | 0 | 2.38 | 12 | 7.10 | -67 | 11.95 | -204 | 19.90 | -261 |

Table B.7: Feature importances (3/4) as computed by the random forests algorithm and normalized costs of omission when the considered output is the logarithm of the number of nodes. The features are sorted in descending order of the feature importances computed by the random forests. The first column indicates the feature number. ''FI' represents the feature importance computed by the random forests for the given feature. The columns labeled 'MRE' and 'COO' represent the mean relative errors (in %) and the corresponding normalized costs of omission achieved when the corresponding feature is removed from the data. Additionally, several cases are considered to compute the MREs and COOs, as detailed in Section 5.6.1.

Output: log #nodes

| | | $RE_i = 0$ | | $0 < RE_i \leq 5$ | | $5 < RE_i \leq 10$ | | $10 < RE_i \leq 15$ | | $15 < RE_i \leq +\infty$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| # | FI | MRE | COO | MRE | COO | MRE | COO | MRE | COO | MRE | COO |
| 21 | 0.0000 | 0.00 | 0 | 2.39 | 13 | 7.06 | -84 | 11.96 | -197 | 19.92 | -242 |
| 22 | 0.0000 | 0.00 | 0 | 2.38 | 12 | 7.11 | -61 | 11.97 | -188 | 19.95 | -224 |
| 28 | 0.0000 | 0.00 | 0 | 2.38 | 12 | 7.10 | -66 | 12.00 | -162 | 20.00 | -192 |
| 55 | 0.0000 | 0.00 | 0 | 2.38 | 12 | 7.13 | -54 | 11.96 | -200 | 19.88 | -269 |
| 48 | 0.0000 | 0.00 | 0 | 2.38 | 12 | 7.11 | -61 | 11.96 | -195 | 19.95 | -225 |
| 47 | 0.0000 | 0.00 | 0 | 2.37 | 11 | 7.12 | -55 | 11.95 | -205 | 19.93 | -237 |
| 30 | 0.0000 | 0.00 | 0 | 2.38 | 12 | 7.11 | -63 | 11.93 | -224 | 19.86 | -285 |
| 27 | 0.0000 | 0.00 | 0 | 2.37 | 12 | 7.09 | -69 | 11.97 | -187 | 19.84 | -294 |
| 51 | 0.0000 | 0.00 | 0 | 2.38 | 13 | 7.11 | -63 | 11.98 | -177 | 19.90 | -257 |
| 23 | 0.0000 | 0.00 | 0 | 2.39 | 13 | 7.08 | -76 | 11.95 | -213 | 19.92 | -244 |
| 25 | 0.0000 | 0.00 | 0 | 2.39 | 13 | 7.11 | -61 | 11.93 | -224 | 19.84 | -299 |
| 50 | 0.0000 | 0.00 | 0 | 2.37 | 11 | 7.09 | -70 | 11.95 | -209 | 19.89 | -264 |
| 53 | 0.0000 | 0.00 | 0 | 2.38 | 13 | 7.13 | -54 | 12.02 | -144 | 20.00 | -197 |
| 54 | 0.0000 | 0.00 | 0 | 2.38 | 13 | 7.12 | -57 | 12.02 | -146 | 19.93 | -236 |
| 56 | 0.0000 | 0.00 | 0 | 2.37 | 12 | 7.08 | -75 | 11.94 | -220 | 19.86 | -282 |
| 57 | 0.0000 | 0.00 | 0 | 2.38 | 13 | 7.11 | -64 | 11.95 | -210 | 19.87 | -275 |

Table B.8: Feature importances (4/4) as computed by the random forests algorithm and normalized costs of omission when the considered output is the logarithm of the number of nodes. The features are sorted in descending order of the feature importances computed by the random forests. The first column indicates the feature number. ''FI' represents the feature importance computed by the random forests for the given feature. The columns labeled 'MRE' and 'COO' represent the mean relative errors (in %) and the corresponding normalized costs of omission achieved when the corresponding feature is removed from the data. Additionally, several cases are considered to compute the MREs and COOs, as detailed in Section 5.6.1.

## B.2 Complete experimental results

This appendix contains the detailed experimental results used to draw Figures 5.1 through 5.7 from Chapter 5.

Tables B.9 through B.28 report the experimental results obtained when our test problems are optimized by a parallel B&B. The number of nodes required to solve each subproblem on each processor is saved. The mean, standard deviation, minimum, and maximum numbers of nodes observed on each processor, together with the sum of the number of nodes of each processor are then computed for each problem, averaged over all problems, and finally reported in the following tables. Note that, in this case, the number $k$ of generated subproblems is equal to the number $N$ of processors. We refer the reader to Section 5.7 for a more detailed explanation on how the results are obtained.

Tables B.9 through B.13 report the detailed experimental results comparing the two trivial partitioning strategies (random and balanced) with the default learned partitioning strategy (score=$\text{score}_{\max}$, probe size=5,000, and output=#nodes), with and without communication (see also Figures 5.1 and 5.2). Tables B.14 through B.18 then illustrate the importance of communication for the default learned partitioning strategy (see Figure 5.3). Next, Tables B.19 through B.23 report the experimental results when different learning outputs are considered (#nodes or logarithm of #nodes) and when different scores are used to combine the predictions of the difficulties of the subproblems during the partitioning procedure (see also Figures 5.4 and 5.5). Finally, Tables B.24 through B.28 show the experimental results when different probing sizes are considered to compute the features used to describe the subproblems (see Figures 5.6 and 5.7).

This appendix omits the analysis and merely reports the obtained results. We refer the reader to Section 5.7 for a detailed discussion.

| | | Parallelized | | | | | |
|---|---|---|---|---|---|---|---|
| | | Without comm. (c0) | | | Comm. every 10k nodes (c10k) | | |
| $N$ | baseline | random | balanced | learned | random | balanced | learned |
| 2 | 9.81e+05 | 7.27e+05 | 1.01e+06 | 5.62e+05 | 5.31e+05 | 5.42e+05 | 4.82e+05 |
| 3 | 9.81e+05 | 5.58e+05 | 8.11e+05 | 4.03e+05 | 3.48e+05 | 3.59e+05 | 3.30e+05 |
| 4 | 9.81e+05 | 5.29e+05 | 8.91e+05 | 3.27e+05 | 2.75e+05 | 2.97e+05 | 2.39e+05 |
| 5 | 9.81e+05 | 5.22e+05 | 8.07e+05 | 2.86e+05 | 2.32e+05 | 2.36e+05 | 1.78e+05 |
| 6 | 9.81e+05 | 4.75e+05 | 7.98e+05 | 2.56e+05 | 1.78e+05 | 2.09e+05 | 1.66e+05 |
| 7 | 9.81e+05 | 4.91e+05 | 9.59e+05 | 2.28e+05 | 1.57e+05 | 2.01e+05 | 1.48e+05 |
| 8 | 9.81e+05 | 4.64e+05 | 1.18e+06 | 1.96e+05 | 1.51e+05 | 1.81e+05 | 1.32e+05 |
| 9 | 9.81e+05 | 4.74e+05 | 1.08e+06 | 1.93e+05 | 1.20e+05 | 1.60e+05 | 1.16e+05 |
| 10 | 9.81e+05 | 4.68e+05 | 1.06e+06 | 2.14e+05 | 1.11e+05 | 1.49e+05 | 1.09e+05 |
| 11 | 9.81e+05 | 4.84e+05 | 1.02e+06 | 2.02e+05 | 1.04e+05 | 1.40e+05 | 1.04e+05 |
| 12 | 9.81e+05 | 4.94e+05 | 1.04e+06 | 2.22e+05 | 9.69e+04 | 1.24e+05 | 9.57e+04 |
| 13 | 9.81e+05 | 6.74e+05 | 1.05e+06 | 1.53e+05 | 1.04e+05 | 1.23e+05 | 8.86e+04 |
| 14 | 9.81e+05 | 6.60e+05 | 1.05e+06 | 1.45e+05 | 9.17e+04 | 1.09e+05 | 7.99e+04 |
| 15 | 9.81e+05 | 6.51e+05 | 1.10e+06 | 1.36e+05 | 8.62e+04 | 1.03e+05 | 7.60e+04 |
| 16 | 9.81e+05 | 6.29e+05 | 1.10e+06 | 1.28e+05 | 8.11e+04 | 1.04e+05 | 6.46e+04 |
| 17 | 9.81e+05 | 6.14e+05 | 1.06e+06 | 1.24e+05 | 7.82e+04 | 9.48e+04 | 6.40e+04 |
| 18 | 9.81e+05 | 5.93e+05 | 1.02e+06 | 1.96e+05 | 7.32e+04 | 9.61e+04 | 5.82e+04 |
| 19 | 9.81e+05 | 5.71e+05 | 1.00e+06 | 1.92e+05 | 7.02e+04 | 9.05e+04 | 5.70e+04 |
| 20 | 9.81e+05 | 5.72e+05 | 9.74e+05 | 1.87e+05 | 6.66e+04 | 8.45e+04 | 5.12e+04 |
| 21 | 9.81e+05 | 5.60e+05 | 9.44e+05 | 2.06e+05 | 6.58e+04 | 8.59e+04 | 4.67e+04 |
| 22 | 9.81e+05 | 5.56e+05 | 9.19e+05 | 2.06e+05 | 6.75e+04 | 8.27e+04 | 5.14e+04 |
| 23 | 9.81e+05 | 6.10e+05 | 9.27e+05 | 2.06e+05 | 6.38e+04 | 7.74e+04 | 4.54e+04 |
| 24 | 9.81e+05 | 6.08e+05 | 9.01e+05 | 2.06e+05 | 6.11e+04 | 7.77e+04 | 4.61e+04 |

Table B.9: Mean number of nodes: trivial partitioning strategies (random and balanced) vs. default learned partitioning strategy (score=score$_{\max}$, probe size=5,000, and output=#nodes).

| | | Parallelized | | | | | |
|---|---|---|---|---|---|---|---|
| | | Without comm. (c0) | | | Comm. every 10k nodes (c10k) | | |
| $N$ | baseline | random | balanced | learned | random | balanced | learned |
| 2 | 0 | 3.50e+05 | 4.25e+05 | 4.39e+05 | 4.89e+05 | 4.29e+05 | 3.45e+05 |
| 3 | 0 | 3.34e+05 | 5.07e+05 | 2.87e+05 | 4.16e+05 | 3.92e+05 | 2.86e+05 |
| 4 | 0 | 3.07e+05 | 5.39e+05 | 2.26e+05 | 3.82e+05 | 3.47e+05 | 2.08e+05 |
| 5 | 0 | 3.21e+05 | 6.60e+05 | 2.14e+05 | 3.26e+05 | 3.05e+05 | 1.64e+05 |
| 6 | 0 | 3.35e+05 | 7.04e+05 | 2.01e+05 | 2.94e+05 | 2.70e+05 | 1.62e+05 |
| 7 | 0 | 3.69e+05 | 9.24e+05 | 1.87e+05 | 2.78e+05 | 2.56e+05 | 1.44e+05 |
| 8 | 0 | 3.86e+05 | 1.61e+06 | 1.61e+05 | 2.76e+05 | 2.50e+05 | 1.39e+05 |
| 9 | 0 | 4.09e+05 | 1.55e+06 | 1.62e+05 | 2.43e+05 | 2.35e+05 | 1.31e+05 |
| 10 | 0 | 3.90e+05 | 1.48e+06 | 2.15e+05 | 2.28e+05 | 2.37e+05 | 1.19e+05 |
| 11 | 0 | 4.32e+05 | 1.43e+06 | 1.82e+05 | 2.15e+05 | 2.23e+05 | 1.23e+05 |
| 12 | 0 | 4.81e+05 | 1.54e+06 | 1.96e+05 | 2.07e+05 | 2.02e+05 | 1.11e+05 |
| 13 | 0 | 1.13e+06 | 1.50e+06 | 1.06e+05 | 2.26e+05 | 1.96e+05 | 9.67e+04 |
| 14 | 0 | 1.10e+06 | 1.51e+06 | 1.03e+05 | 1.99e+05 | 1.77e+05 | 8.87e+04 |
| 15 | 0 | 1.11e+06 | 1.50e+06 | 9.69e+04 | 1.88e+05 | 1.73e+05 | 8.06e+04 |
| 16 | 0 | 1.08e+06 | 1.48e+06 | 8.94e+04 | 1.84e+05 | 1.74e+05 | 7.27e+04 |
| 17 | 0 | 1.06e+06 | 1.45e+06 | 8.76e+04 | 1.79e+05 | 1.64e+05 | 7.16e+04 |
| 18 | 0 | 1.06e+06 | 1.42e+06 | 1.83e+05 | 1.73e+05 | 1.63e+05 | 6.42e+04 |
| 19 | 0 | 1.04e+06 | 1.41e+06 | 1.80e+05 | 1.72e+05 | 1.58e+05 | 6.40e+04 |
| 20 | 0 | 1.08e+06 | 1.39e+06 | 1.78e+05 | 1.64e+05 | 1.53e+05 | 5.23e+04 |
| 21 | 0 | 1.08e+06 | 1.36e+06 | 2.03e+05 | 1.61e+05 | 1.58e+05 | 5.23e+04 |
| 22 | 0 | 1.05e+06 | 1.34e+06 | 2.01e+05 | 1.68e+05 | 1.54e+05 | 5.81e+04 |
| 23 | 0 | 1.29e+06 | 1.31e+06 | 1.99e+05 | 1.64e+05 | 1.50e+05 | 5.14e+04 |
| 24 | 0 | 1.27e+06 | 1.28e+06 | 1.95e+05 | 1.58e+05 | 1.51e+05 | 5.79e+04 |

Table B.10: Standard deviation of the number of nodes: trivial partitioning strategies (random and balanced) vs. default learned partitioning strategy (score=$\text{score}_{\max}$, probe size=5,000, and output=#nodes).

| | | Parallelized | | | | | |
|---|---|---|---|---|---|---|---|
| | | Without comm. (c0) | | | Comm. every 10k nodes (c10k) | | |
| $N$ | baseline | random | balanced | learned | random | balanced | learned |
| 2 | 9.81e+05 | 4.80e+05 | 7.12e+05 | 2.52e+05 | 1.86e+05 | 2.39e+05 | 2.38e+05 |
| 3 | 9.81e+05 | 2.84e+05 | 3.08e+05 | 1.74e+05 | 5.20e+04 | 6.23e+04 | 1.19e+05 |
| 4 | 9.81e+05 | 2.48e+05 | 2.85e+05 | 1.19e+05 | 2.75e+04 | 2.83e+04 | 7.11e+04 |
| 5 | 9.81e+05 | 2.08e+05 | 1.24e+05 | 7.58e+04 | 1.74e+04 | 2.17e+04 | 1.60e+04 |
| 6 | 9.81e+05 | 1.40e+05 | 1.20e+05 | 4.45e+04 | 1.55e+04 | 9.18e+03 | 9.09e+03 |
| 7 | 9.81e+05 | 1.24e+05 | 1.07e+05 | 3.61e+04 | 1.24e+04 | 6.88e+03 | 7.27e+03 |
| 8 | 9.81e+05 | 1.00e+05 | 9.87e+04 | 2.86e+04 | 9.40e+03 | 6.35e+03 | 5.87e+03 |
| 9 | 9.81e+05 | 9.38e+04 | 9.11e+04 | 1.66e+04 | 6.88e+03 | 7.11e+03 | 5.32e+03 |
| 10 | 9.81e+05 | 9.33e+04 | 7.77e+04 | 1.61e+04 | 6.71e+03 | 5.84e+03 | 4.73e+03 |
| 11 | 9.81e+05 | 9.18e+04 | 7.37e+04 | 9.23e+03 | 5.52e+03 | 4.60e+03 | 3.53e+03 |
| 12 | 9.81e+05 | 8.07e+04 | 7.03e+04 | 7.70e+03 | 4.00e+03 | 4.00e+03 | 2.67e+03 |
| 13 | 9.81e+05 | 7.32e+04 | 6.03e+04 | 2.53e+04 | 3.71e+03 | 3.49e+03 | 2.58e+03 |
| 14 | 9.81e+05 | 7.18e+04 | 5.80e+04 | 2.52e+04 | 3.03e+03 | 2.86e+03 | 2.52e+03 |
| 15 | 9.81e+05 | 4.51e+04 | 4.77e+04 | 2.32e+04 | 2.75e+03 | 2.39e+03 | 2.49e+03 |
| 16 | 9.81e+05 | 2.40e+04 | 3.66e+04 | 2.31e+04 | 2.27e+03 | 2.30e+03 | 2.53e+03 |
| 17 | 9.81e+05 | 2.38e+04 | 3.61e+04 | 1.96e+04 | 1.65e+03 | 2.52e+03 | 2.44e+03 |
| 18 | 9.81e+05 | 2.20e+04 | 3.36e+04 | 7.06e+03 | 1.54e+03 | 2.10e+03 | 2.49e+03 |
| 19 | 9.81e+05 | 1.80e+04 | 2.72e+04 | 6.91e+03 | 1.24e+03 | 2.09e+03 | 2.30e+03 |
| 20 | 9.81e+05 | 1.42e+04 | 2.72e+04 | 6.66e+03 | 540 | 2.10e+03 | 2.02e+03 |
| 21 | 9.81e+05 | 1.16e+04 | 2.75e+04 | 4.77e+03 | 611 | 1.87e+03 | 1.99e+03 |
| 22 | 9.81e+05 | 1.09e+04 | 2.65e+04 | 3.74e+03 | 609 | 1.31e+03 | 1.71e+03 |
| 23 | 9.81e+05 | 1.09e+04 | 2.69e+04 | 3.74e+03 | 606 | 1.17e+03 | 1.79e+03 |
| 24 | 9.81e+05 | 1.09e+04 | 2.51e+04 | 3.69e+03 | 463 | 377 | 1.62e+03 |

Table B.11: Minimum number of nodes: trivial partitioning strategies (random and balanced) vs. default learned partitioning strategy (score=$\text{score}_{\max}$, probe size=5,000, and output=#nodes).

| | | Parallelized | | | | | |
|---|---|---|---|---|---|---|---|
| | | Without comm. (c0) | | | Comm. every 10k nodes (c10k) | | |
| $N$ | baseline | random | balanced | learned | random | balanced | learned |
| 2 | 9.81e+05 | 9.75e+05 | 1.31e+06 | 8.73e+05 | 8.77e+05 | 8.46e+05 | 7.26e+05 |
| 3 | 9.81e+05 | 9.08e+05 | 1.30e+06 | 7.14e+05 | 8.11e+05 | 7.94e+05 | 6.45e+05 |
| 4 | 9.81e+05 | 9.11e+05 | 1.47e+06 | 6.09e+05 | 8.19e+05 | 7.69e+05 | 5.04e+05 |
| 5 | 9.81e+05 | 9.71e+05 | 1.69e+06 | 5.83e+05 | 7.72e+05 | 7.38e+05 | 4.21e+05 |
| 6 | 9.81e+05 | 9.93e+05 | 1.93e+06 | 5.41e+05 | 7.58e+05 | 6.76e+05 | 4.29e+05 |
| 7 | 9.81e+05 | 1.13e+06 | 2.58e+06 | 5.28e+05 | 7.64e+05 | 6.99e+05 | 3.97e+05 |
| 8 | 9.81e+05 | 1.18e+06 | 4.69e+06 | 4.76e+05 | 7.92e+05 | 7.11e+05 | 4.03e+05 |
| 9 | 9.81e+05 | 1.29e+06 | 4.70e+06 | 4.80e+05 | 7.41e+05 | 6.71e+05 | 3.94e+05 |
| 10 | 9.81e+05 | 1.26e+06 | 4.66e+06 | 6.90e+05 | 7.33e+05 | 7.12e+05 | 3.74e+05 |
| 11 | 9.81e+05 | 1.45e+06 | 4.68e+06 | 5.87e+05 | 7.10e+05 | 6.96e+05 | 3.89e+05 |
| 12 | 9.81e+05 | 1.66e+06 | 5.03e+06 | 5.86e+05 | 7.09e+05 | 6.51e+05 | 3.56e+05 |
| 13 | 9.81e+05 | 4.04e+06 | 5.05e+06 | 3.80e+05 | 7.56e+05 | 6.55e+05 | 3.11e+05 |
| 14 | 9.81e+05 | 4.07e+06 | 5.33e+06 | 3.79e+05 | 7.11e+05 | 5.94e+05 | 2.86e+05 |
| 15 | 9.81e+05 | 4.19e+06 | 5.15e+06 | 3.72e+05 | 6.98e+05 | 5.89e+05 | 2.66e+05 |
| 16 | 9.81e+05 | 4.22e+06 | 5.22e+06 | 3.43e+05 | 7.00e+05 | 6.17e+05 | 2.66e+05 |
| 17 | 9.81e+05 | 4.22e+06 | 5.22e+06 | 3.36e+05 | 7.00e+05 | 5.95e+05 | 2.66e+05 |
| 18 | 9.81e+05 | 4.31e+06 | 5.22e+06 | 6.56e+05 | 6.96e+05 | 5.93e+05 | 2.45e+05 |
| 19 | 9.81e+05 | 4.31e+06 | 5.27e+06 | 6.59e+05 | 7.07e+05 | 5.87e+05 | 2.46e+05 |
| 20 | 9.81e+05 | 4.62e+06 | 5.26e+06 | 6.57e+05 | 6.96e+05 | 5.83e+05 | 1.91e+05 |
| 21 | 9.81e+05 | 4.62e+06 | 5.38e+06 | 7.55e+05 | 6.98e+05 | 6.21e+05 | 2.13e+05 |
| 22 | 9.81e+05 | 4.59e+06 | 5.38e+06 | 7.60e+05 | 7.14e+05 | 6.09e+05 | 2.23e+05 |
| 23 | 9.81e+05 | 5.92e+06 | 5.21e+06 | 7.52e+05 | 7.16e+05 | 6.20e+05 | 2.03e+05 |
| 24 | 9.81e+05 | 5.91e+06 | 5.16e+06 | 7.43e+05 | 7.20e+05 | 6.19e+05 | 2.32e+05 |

Table B.12: Maximum number of nodes: trivial partitioning strategies (random and balanced) vs. default learned partitioning strategy (score=$\text{score}_{\max}$, probe size=5,000, and output=#nodes).

| | | Parallelized | | | | | |
|---|---|---|---|---|---|---|---|
| | | Without comm. (c0) | | | Comm. every 10k nodes (c10k) | | |
| $N$ | baseline | random | balanced | learned | random | balanced | learned |
| 2 | 9.81e+05 | 1.45e+06 | 2.03e+06 | 1.12e+06 | 1.06e+06 | 1.08e+06 | 9.64e+05 |
| 3 | 9.81e+05 | 1.67e+06 | 2.43e+06 | 1.21e+06 | 1.04e+06 | 1.08e+06 | 9.91e+05 |
| 4 | 9.81e+05 | 2.12e+06 | 3.57e+06 | 1.31e+06 | 1.10e+06 | 1.19e+06 | 9.56e+05 |
| 5 | 9.81e+05 | 2.61e+06 | 4.03e+06 | 1.43e+06 | 1.16e+06 | 1.18e+06 | 8.91e+05 |
| 6 | 9.81e+05 | 2.85e+06 | 4.79e+06 | 1.53e+06 | 1.07e+06 | 1.26e+06 | 9.95e+05 |
| 7 | 9.81e+05 | 3.44e+06 | 6.71e+06 | 1.60e+06 | 1.10e+06 | 1.41e+06 | 1.04e+06 |
| 8 | 9.81e+05 | 3.71e+06 | 9.44e+06 | 1.53e+06 | 1.21e+06 | 1.45e+06 | 1.02e+06 |
| 9 | 9.81e+05 | 4.26e+06 | 9.76e+06 | 1.69e+06 | 1.08e+06 | 1.44e+06 | 1.02e+06 |
| 10 | 9.81e+05 | 4.68e+06 | 1.06e+07 | 2.06e+06 | 1.11e+06 | 1.49e+06 | 1.07e+06 |
| 11 | 9.81e+05 | 5.32e+06 | 1.12e+07 | 2.14e+06 | 1.15e+06 | 1.54e+06 | 1.12e+06 |
| 12 | 9.81e+05 | 5.93e+06 | 1.25e+07 | 2.56e+06 | 1.16e+06 | 1.48e+06 | 1.12e+06 |
| 13 | 9.81e+05 | 8.77e+06 | 1.36e+07 | 1.99e+06 | 1.36e+06 | 1.60e+06 | 1.12e+06 |
| 14 | 9.81e+05 | 9.25e+06 | 1.47e+07 | 2.03e+06 | 1.28e+06 | 1.53e+06 | 1.10e+06 |
| 15 | 9.81e+05 | 9.77e+06 | 1.66e+07 | 2.04e+06 | 1.29e+06 | 1.55e+06 | 1.11e+06 |
| 16 | 9.81e+05 | 1.01e+07 | 1.76e+07 | 2.04e+06 | 1.30e+06 | 1.67e+06 | 1.02e+06 |
| 17 | 9.81e+05 | 1.04e+07 | 1.81e+07 | 2.11e+06 | 1.33e+06 | 1.61e+06 | 1.07e+06 |
| 18 | 9.81e+05 | 1.07e+07 | 1.83e+07 | 3.43e+06 | 1.32e+06 | 1.73e+06 | 1.03e+06 |
| 19 | 9.81e+05 | 1.08e+07 | 1.90e+07 | 3.56e+06 | 1.33e+06 | 1.72e+06 | 1.07e+06 |
| 20 | 9.81e+05 | 1.14e+07 | 1.95e+07 | 3.57e+06 | 1.33e+06 | 1.69e+06 | 1.00e+06 |
| 21 | 9.81e+05 | 1.18e+07 | 1.98e+07 | 4.12e+06 | 1.38e+06 | 1.80e+06 | 9.57e+05 |
| 22 | 9.81e+05 | 1.22e+07 | 2.02e+07 | 4.33e+06 | 1.49e+06 | 1.82e+06 | 1.10e+06 |
| 23 | 9.81e+05 | 1.40e+07 | 2.13e+07 | 4.45e+06 | 1.47e+06 | 1.78e+06 | 1.01e+06 |
| 24 | 9.81e+05 | 1.46e+07 | 2.16e+07 | 4.65e+06 | 1.47e+06 | 1.87e+06 | 1.07e+06 |

Table B.13: Total number of nodes: trivial partitioning strategies (random and balanced) vs. default learned partitioning strategy (score=$\text{score}_{\max}$, probe size=5,000, and output=#nodes).

| | | Parallelized: learned partitioning strategy | | | | | |
|---|---|---|---|---|---|---|---|
| $N$ | baseline | c0 | c1k | c5k | c10k | c20k | c50k |
| 2 | 9.81e+05 | 5.62e+05 | 4.87e+05 | 4.84e+05 | 4.82e+05 | 4.79e+05 | 4.83e+05 |
| 3 | 9.81e+05 | 4.03e+05 | 3.21e+05 | 3.24e+05 | 3.30e+05 | 3.24e+05 | 3.27e+05 |
| 4 | 9.81e+05 | 3.27e+05 | 2.36e+05 | 2.36e+05 | 2.39e+05 | 2.37e+05 | 2.43e+05 |
| 5 | 9.81e+05 | 2.86e+05 | 1.70e+05 | 1.76e+05 | 1.78e+05 | 1.77e+05 | 1.88e+05 |
| 6 | 9.81e+05 | 2.56e+05 | 1.65e+05 | 1.72e+05 | 1.66e+05 | 1.72e+05 | 1.74e+05 |
| 7 | 9.81e+05 | 2.28e+05 | 1.45e+05 | 1.38e+05 | 1.48e+05 | 1.40e+05 | 1.46e+05 |
| 8 | 9.81e+05 | 1.96e+05 | 1.20e+05 | 1.32e+05 | 1.32e+05 | 1.36e+05 | 1.34e+05 |
| 9 | 9.81e+05 | 1.93e+05 | 1.21e+05 | 1.25e+05 | 1.16e+05 | 1.24e+05 | 1.23e+05 |
| 10 | 9.81e+05 | 2.14e+05 | 1.10e+05 | 1.12e+05 | 1.09e+05 | 1.17e+05 | 1.19e+05 |
| 11 | 9.81e+05 | 2.02e+05 | 1.02e+05 | 1.01e+05 | 1.04e+05 | 1.03e+05 | 1.12e+05 |
| 12 | 9.81e+05 | 2.22e+05 | 9.06e+04 | 9.20e+04 | 9.57e+04 | 9.63e+04 | 1.03e+05 |
| 13 | 9.81e+05 | 1.53e+05 | 8.38e+04 | 8.02e+04 | 8.86e+04 | 8.59e+04 | 9.72e+04 |
| 14 | 9.81e+05 | 1.45e+05 | 6.64e+04 | 7.61e+04 | 7.99e+04 | 8.28e+04 | 8.49e+04 |
| 15 | 9.81e+05 | 1.36e+05 | 7.54e+04 | 7.38e+04 | 7.60e+04 | 8.15e+04 | 7.64e+04 |
| 16 | 9.81e+05 | 1.28e+05 | 6.06e+04 | 6.04e+04 | 6.46e+04 | 6.25e+04 | 6.76e+04 |
| 17 | 9.81e+05 | 1.24e+05 | 5.72e+04 | 6.00e+04 | 6.40e+04 | 6.42e+04 | 6.68e+04 |
| 18 | 9.81e+05 | 1.96e+05 | 5.14e+04 | 5.88e+04 | 5.82e+04 | 5.86e+04 | 6.63e+04 |
| 19 | 9.81e+05 | 1.92e+05 | 4.90e+04 | 5.33e+04 | 5.70e+04 | 6.09e+04 | 5.91e+04 |
| 20 | 9.81e+05 | 1.87e+05 | 4.29e+04 | 4.69e+04 | 5.12e+04 | 5.22e+04 | 5.24e+04 |
| 21 | 9.81e+05 | 2.06e+05 | 4.47e+04 | 4.89e+04 | 4.67e+04 | 4.86e+04 | 5.33e+04 |
| 22 | 9.81e+05 | 2.06e+05 | 4.79e+04 | 4.60e+04 | 5.14e+04 | 5.38e+04 | 5.09e+04 |
| 23 | 9.81e+05 | 2.06e+05 | 4.08e+04 | 4.36e+04 | 4.54e+04 | 4.57e+04 | 5.13e+04 |
| 24 | 9.81e+05 | 2.06e+05 | 4.44e+04 | 4.48e+04 | 4.61e+04 | 4.80e+04 | 5.22e+04 |

Table B.14: Mean number of nodes: default learned partitioning strategy (score=$\text{score}_{\max}$, probe size=5,000, and output=#nodes) for different communication intervals. 'c0' indicates no communication and 'cx' indicates communication every x nodes (e.g., 'c1k' indicates communication every 1,000 nodes).

| | | Parallelized: learned partitioning strategy | | | | | |
|---|---|---|---|---|---|---|---|
| $N$ | baseline | c0 | c1k | c5k | c10k | c20k | c50k |
| 2 | 0 | 4.39e+05 | 3.54e+05 | 3.48e+05 | 3.45e+05 | 3.48e+05 | 3.47e+05 |
| 3 | 0 | 2.87e+05 | 2.97e+05 | 2.89e+05 | 2.86e+05 | 2.91e+05 | 2.86e+05 |
| 4 | 0 | 2.26e+05 | 2.12e+05 | 2.09e+05 | 2.08e+05 | 2.08e+05 | 2.03e+05 |
| 5 | 0 | 2.14e+05 | 1.60e+05 | 1.71e+05 | 1.64e+05 | 1.66e+05 | 1.70e+05 |
| 6 | 0 | 2.01e+05 | 1.65e+05 | 1.73e+05 | 1.62e+05 | 1.67e+05 | 1.63e+05 |
| 7 | 0 | 1.87e+05 | 1.50e+05 | 1.47e+05 | 1.44e+05 | 1.39e+05 | 1.43e+05 |
| 8 | 0 | 1.61e+05 | 1.34e+05 | 1.41e+05 | 1.39e+05 | 1.43e+05 | 1.33e+05 |
| 9 | 0 | 1.62e+05 | 1.37e+05 | 1.34e+05 | 1.31e+05 | 1.30e+05 | 1.25e+05 |
| 10 | 0 | 2.15e+05 | 1.21e+05 | 1.22e+05 | 1.19e+05 | 1.25e+05 | 1.19e+05 |
| 11 | 0 | 1.82e+05 | 1.19e+05 | 1.13e+05 | 1.23e+05 | 1.11e+05 | 1.14e+05 |
| 12 | 0 | 1.96e+05 | 1.16e+05 | 1.10e+05 | 1.11e+05 | 1.09e+05 | 1.14e+05 |
| 13 | 0 | 1.06e+05 | 9.50e+04 | 9.41e+04 | 9.67e+04 | 9.28e+04 | 9.76e+04 |
| 14 | 0 | 1.03e+05 | 8.10e+04 | 8.57e+04 | 8.87e+04 | 8.55e+04 | 8.76e+04 |
| 15 | 0 | 9.69e+04 | 8.74e+04 | 8.14e+04 | 8.06e+04 | 8.35e+04 | 7.81e+04 |
| 16 | 0 | 8.94e+04 | 7.65e+04 | 7.29e+04 | 7.27e+04 | 6.79e+04 | 6.91e+04 |
| 17 | 0 | 8.76e+04 | 6.88e+04 | 6.64e+04 | 7.16e+04 | 6.73e+04 | 6.84e+04 |
| 18 | 0 | 1.83e+05 | 6.33e+04 | 6.61e+04 | 6.42e+04 | 6.22e+04 | 6.41e+04 |
| 19 | 0 | 1.80e+05 | 5.94e+04 | 6.32e+04 | 6.40e+04 | 6.35e+04 | 6.09e+04 |
| 20 | 0 | 1.78e+05 | 4.87e+04 | 5.09e+04 | 5.23e+04 | 5.04e+04 | 4.93e+04 |
| 21 | 0 | 2.03e+05 | 5.17e+04 | 5.45e+04 | 5.23e+04 | 5.20e+04 | 5.26e+04 |
| 22 | 0 | 2.01e+05 | 6.06e+04 | 5.26e+04 | 5.81e+04 | 5.79e+04 | 5.10e+04 |
| 23 | 0 | 1.99e+05 | 4.88e+04 | 5.04e+04 | 5.14e+04 | 5.01e+04 | 5.09e+04 |
| 24 | 0 | 1.95e+05 | 6.00e+04 | 5.90e+04 | 5.79e+04 | 5.85e+04 | 5.92e+04 |

Table B.15: Standard deviation of the number of nodes: default learned partitioning strategy (score=$\text{score}_{\max}$, probe size=5,000, and output=#nodes) for different communication intervals. 'c0' indicates no communication and 'cx' indicates communication every x nodes (e.g., 'c1k' indicates communication every 1,000 nodes).

| | | Parallelized: learned partitioning strategy | | | | | |
|---|---|---|---|---|---|---|---|
| $N$ | baseline | c0 | c1k | c5k | c10k | c20k | c50k |
| 2 | 9.81e+05 | 2.52e+05 | 2.37e+05 | 2.38e+05 | 2.38e+05 | 2.33e+05 | 2.38e+05 |
| 3 | 9.81e+05 | 1.74e+05 | 9.99e+04 | 1.10e+05 | 1.19e+05 | 1.10e+05 | 1.18e+05 |
| 4 | 9.81e+05 | 1.19e+05 | 7.20e+04 | 6.98e+04 | 7.11e+04 | 7.34e+04 | 7.44e+04 |
| 5 | 9.81e+05 | 7.58e+04 | 1.45e+04 | 1.49e+04 | 1.60e+04 | 2.04e+04 | 2.60e+04 |
| 6 | 9.81e+05 | 4.45e+04 | 5.21e+03 | 7.75e+03 | 9.09e+03 | 1.47e+04 | 1.95e+04 |
| 7 | 9.81e+05 | 3.61e+04 | 2.04e+03 | 4.46e+03 | 7.27e+03 | 9.82e+03 | 1.73e+04 |
| 8 | 9.81e+05 | 2.86e+04 | 2.05e+03 | 4.57e+03 | 5.87e+03 | 7.26e+03 | 1.30e+04 |
| 9 | 9.81e+05 | 1.66e+04 | 1.68e+03 | 3.97e+03 | 5.32e+03 | 6.59e+03 | 8.49e+03 |
| 10 | 9.81e+05 | 1.61e+04 | 1.55e+03 | 3.10e+03 | 4.73e+03 | 4.96e+03 | 7.80e+03 |
| 11 | 9.81e+05 | 9.23e+03 | 1.17e+03 | 2.56e+03 | 3.53e+03 | 4.60e+03 | 6.47e+03 |
| 12 | 9.81e+05 | 7.70e+03 | 6.08e+02 | 1.89e+03 | 2.67e+03 | 3.15e+03 | 3.94e+03 |
| 13 | 9.81e+05 | 2.53e+04 | 6.30e+02 | 1.87e+03 | 2.58e+03 | 3.16e+03 | 3.30e+03 |
| 14 | 9.81e+05 | 2.52e+04 | 5.79e+02 | 1.80e+03 | 2.52e+03 | 2.96e+03 | 3.11e+03 |
| 15 | 9.81e+05 | 2.32e+04 | 5.90e+02 | 1.78e+03 | 2.49e+03 | 3.12e+03 | 3.27e+03 |
| 16 | 9.81e+05 | 2.31e+04 | 5.78e+02 | 1.79e+03 | 2.53e+03 | 3.25e+03 | 3.08e+03 |
| 17 | 9.81e+05 | 1.96e+04 | 5.77e+02 | 1.74e+03 | 2.44e+03 | 3.21e+03 | 3.39e+03 |
| 18 | 9.81e+05 | 7.06e+03 | 5.72e+02 | 1.89e+03 | 2.49e+03 | 3.16e+03 | 3.09e+03 |
| 19 | 9.81e+05 | 6.91e+03 | 5.49e+02 | 1.59e+03 | 2.30e+03 | 2.96e+03 | 2.88e+03 |
| 20 | 9.81e+05 | 6.66e+03 | 4.98e+02 | 1.36e+03 | 2.02e+03 | 2.25e+03 | 2.65e+03 |
| 21 | 9.81e+05 | 4.77e+03 | 5.09e+02 | 1.35e+03 | 1.99e+03 | 2.67e+03 | 2.64e+03 |
| 22 | 9.81e+05 | 3.74e+03 | 5.01e+02 | 1.30e+03 | 1.71e+03 | 2.16e+03 | 2.17e+03 |
| 23 | 9.81e+05 | 3.74e+03 | 5.06e+02 | 1.34e+03 | 1.79e+03 | 2.31e+03 | 2.15e+03 |
| 24 | 9.81e+05 | 3.69e+03 | 4.48e+02 | 1.12e+03 | 1.62e+03 | 1.99e+03 | 1.96e+03 |

Table B.16: Minimum number of nodes: default learned partitioning strategy (score=$\text{score}_{\max}$, probe size=5,000, and output=#nodes) for different communication intervals. 'c0' indicates no communication and 'cx' indicates communication every x nodes (e.g., 'c1k' indicates communication every 1,000 nodes).

| | | Parallelized: learned partitioning strategy | | | | | |
|---|---|---|---|---|---|---|---|
| $N$ | baseline | c0 | c1k | c5k | c10k | c20k | c50k |
| 2 | 9.81e+05 | 8.73e+05 | 7.37e+05 | 7.30e+05 | 7.26e+05 | 7.25e+05 | 7.29e+05 |
| 3 | 9.81e+05 | 7.14e+05 | 6.45e+05 | 6.41e+05 | 6.45e+05 | 6.39e+05 | 6.39e+05 |
| 4 | 9.81e+05 | 6.09e+05 | 5.21e+05 | 5.06e+05 | 5.04e+05 | 5.15e+05 | 5.04e+05 |
| 5 | 9.81e+05 | 5.83e+05 | 4.09e+05 | 4.37e+05 | 4.21e+05 | 4.28e+05 | 4.34e+05 |
| 6 | 9.81e+05 | 5.41e+05 | 4.41e+05 | 4.55e+05 | 4.29e+05 | 4.42e+05 | 4.37e+05 |
| 7 | 9.81e+05 | 5.28e+05 | 4.21e+05 | 4.10e+05 | 3.97e+05 | 3.98e+05 | 4.15e+05 |
| 8 | 9.81e+05 | 4.76e+05 | 3.92e+05 | 4.04e+05 | 4.03e+05 | 4.14e+05 | 3.87e+05 |
| 9 | 9.81e+05 | 4.80e+05 | 4.12e+05 | 3.97e+05 | 3.94e+05 | 3.83e+05 | 3.78e+05 |
| 10 | 9.81e+05 | 6.90e+05 | 3.82e+05 | 3.70e+05 | 3.74e+05 | 3.86e+05 | 3.73e+05 |
| 11 | 9.81e+05 | 5.87e+05 | 3.87e+05 | 3.67e+05 | 3.89e+05 | 3.62e+05 | 3.66e+05 |
| 12 | 9.81e+05 | 5.86e+05 | 3.88e+05 | 3.64e+05 | 3.56e+05 | 3.63e+05 | 3.74e+05 |
| 13 | 9.81e+05 | 3.80e+05 | 3.13e+05 | 3.14e+05 | 3.11e+05 | 3.02e+05 | 3.24e+05 |
| 14 | 9.81e+05 | 3.79e+05 | 2.70e+05 | 2.64e+05 | 2.86e+05 | 2.78e+05 | 2.91e+05 |
| 15 | 9.81e+05 | 3.72e+05 | 2.83e+05 | 2.67e+05 | 2.66e+05 | 2.76e+05 | 2.62e+05 |
| 16 | 9.81e+05 | 3.43e+05 | 2.73e+05 | 2.64e+05 | 2.66e+05 | 2.47e+05 | 2.55e+05 |
| 17 | 9.81e+05 | 3.36e+05 | 2.46e+05 | 2.48e+05 | 2.66e+05 | 2.48e+05 | 2.58e+05 |
| 18 | 9.81e+05 | 6.56e+05 | 2.36e+05 | 2.54e+05 | 2.45e+05 | 2.39e+05 | 2.43e+05 |
| 19 | 9.81e+05 | 6.59e+05 | 2.25e+05 | 2.53e+05 | 2.46e+05 | 2.47e+05 | 2.47e+05 |
| 20 | 9.81e+05 | 6.57e+05 | 1.81e+05 | 2.01e+05 | 1.91e+05 | 1.82e+05 | 1.95e+05 |
| 21 | 9.81e+05 | 7.55e+05 | 2.01e+05 | 2.04e+05 | 2.13e+05 | 2.14e+05 | 2.09e+05 |
| 22 | 9.81e+05 | 7.60e+05 | 2.35e+05 | 2.15e+05 | 2.23e+05 | 2.21e+05 | 2.13e+05 |
| 23 | 9.81e+05 | 7.52e+05 | 1.95e+05 | 1.98e+05 | 2.03e+05 | 2.13e+05 | 2.11e+05 |
| 24 | 9.81e+05 | 7.43e+05 | 2.22e+05 | 2.37e+05 | 2.32e+05 | 2.27e+05 | 2.38e+05 |

Table B.17: Maximum number of nodes: default learned partitioning strategy (score=$\text{score}_{\max}$, probe size=5,000, and output=#nodes) for different communication intervals. 'c0' indicates no communication and 'cx' indicates communication every x nodes (e.g., 'c1k' indicates communication every 1,000 nodes).

| | | Parallelized: learned partitioning strategy | | | | | |
|---|---|---|---|---|---|---|---|
| $N$ | baseline | c0 | c1k | c5k | c10k | c20k | c50k |
| 2 | 9.81e+05 | 1.12e+06 | 9.74e+05 | 9.68e+05 | 9.64e+05 | 9.59e+05 | 9.67e+05 |
| 3 | 9.81e+05 | 1.21e+06 | 9.63e+05 | 9.73e+05 | 9.91e+05 | 9.71e+05 | 9.80e+05 |
| 4 | 9.81e+05 | 1.31e+06 | 9.44e+05 | 9.44e+05 | 9.56e+05 | 9.47e+05 | 9.71e+05 |
| 5 | 9.81e+05 | 1.43e+06 | 8.52e+05 | 8.80e+05 | 8.91e+05 | 8.87e+05 | 9.39e+05 |
| 6 | 9.81e+05 | 1.53e+06 | 9.88e+05 | 1.03e+06 | 9.95e+05 | 1.03e+06 | 1.04e+06 |
| 7 | 9.81e+05 | 1.60e+06 | 1.01e+06 | 9.69e+05 | 1.04e+06 | 9.83e+05 | 1.02e+06 |
| 8 | 9.81e+05 | 1.53e+06 | 9.36e+05 | 1.02e+06 | 1.02e+06 | 1.06e+06 | 1.04e+06 |
| 9 | 9.81e+05 | 1.69e+06 | 1.06e+06 | 1.09e+06 | 1.02e+06 | 1.09e+06 | 1.08e+06 |
| 10 | 9.81e+05 | 2.06e+06 | 1.08e+06 | 1.09e+06 | 1.07e+06 | 1.14e+06 | 1.16e+06 |
| 11 | 9.81e+05 | 2.14e+06 | 1.09e+06 | 1.09e+06 | 1.12e+06 | 1.11e+06 | 1.20e+06 |
| 12 | 9.81e+05 | 2.56e+06 | 1.07e+06 | 1.08e+06 | 1.12e+06 | 1.13e+06 | 1.21e+06 |
| 13 | 9.81e+05 | 1.99e+06 | 1.06e+06 | 1.02e+06 | 1.12e+06 | 1.09e+06 | 1.23e+06 |
| 14 | 9.81e+05 | 2.03e+06 | 9.15e+05 | 1.04e+06 | 1.10e+06 | 1.14e+06 | 1.17e+06 |
| 15 | 9.81e+05 | 2.04e+06 | 1.11e+06 | 1.08e+06 | 1.11e+06 | 1.19e+06 | 1.12e+06 |
| 16 | 9.81e+05 | 2.04e+06 | 9.55e+05 | 9.53e+05 | 1.02e+06 | 9.87e+05 | 1.07e+06 |
| 17 | 9.81e+05 | 2.11e+06 | 9.57e+05 | 1.00e+06 | 1.07e+06 | 1.07e+06 | 1.12e+06 |
| 18 | 9.81e+05 | 3.43e+06 | 9.13e+05 | 1.04e+06 | 1.03e+06 | 1.04e+06 | 1.17e+06 |
| 19 | 9.81e+05 | 3.56e+06 | 9.20e+05 | 1.00e+06 | 1.07e+06 | 1.14e+06 | 1.11e+06 |
| 20 | 9.81e+05 | 3.57e+06 | 8.37e+05 | 9.19e+05 | 1.00e+06 | 1.02e+06 | 1.03e+06 |
| 21 | 9.81e+05 | 4.12e+06 | 9.15e+05 | 1.01e+06 | 9.57e+05 | 9.98e+05 | 1.09e+06 |
| 22 | 9.81e+05 | 4.33e+06 | 1.02e+06 | 9.91e+05 | 1.10e+06 | 1.15e+06 | 1.10e+06 |
| 23 | 9.81e+05 | 4.45e+06 | 9.07e+05 | 9.73e+05 | 1.01e+06 | 1.02e+06 | 1.15e+06 |
| 24 | 9.81e+05 | 4.65e+06 | 1.03e+06 | 1.04e+06 | 1.07e+06 | 1.11e+06 | 1.21e+06 |

Table B.18: Total number of nodes: default learned partitioning strategy (score=$\text{score}_{\max}$, probe size=5,000, and output=#nodes) for different communication intervals. 'c0' indicates no communication and 'cx' indicates communication every x nodes (e.g., 'c1k' indicates communication every 1,000 nodes).

| | | Parallelized: learned partitioning strategy | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Without comm. (c0) | | | | Comm. every 10k nodes (c10k) | | | |
| | | $\text{score}_{\text{bal}}$ | | $\text{score}_{\text{max}}$ | | $\text{score}_{\text{bal}}$ | | $\text{score}_{\text{max}}$ | |
| $N$ | baseline | #nodes | log #nodes | #nodes | log #nodes | #nodes | log #nodes | #nodes | log #nodes |
| 2 | 9.81e+05 | 9.91e+05 | 8.91e+05 | 5.62e+05 | 6.97e+05 | 5.78e+05 | 6.34e+05 | 4.82e+05 | 5.15e+05 |
| 3 | 9.81e+05 | 1.00e+06 | 9.81e+05 | 4.03e+05 | 4.81e+05 | 4.05e+05 | 4.86e+05 | 3.30e+05 | 2.79e+05 |
| 4 | 9.81e+05 | 8.69e+05 | 8.44e+05 | 3.27e+05 | 4.74e+05 | 3.09e+05 | 3.65e+05 | 2.39e+05 | 2.40e+05 |
| 5 | 9.81e+05 | 7.80e+05 | 7.71e+05 | 2.86e+05 | 4.29e+05 | 2.56e+05 | 2.97e+05 | 1.78e+05 | 2.04e+05 |
| 6 | 9.81e+05 | 6.90e+05 | 8.27e+05 | 2.56e+05 | 4.02e+05 | 2.15e+05 | 2.48e+05 | 1.66e+05 | 1.70e+05 |
| 7 | 9.81e+05 | 6.30e+05 | 7.46e+05 | 2.28e+05 | 3.63e+05 | 1.88e+05 | 2.17e+05 | 1.48e+05 | 1.38e+05 |
| 8 | 9.81e+05 | 5.91e+05 | 7.15e+05 | 1.96e+05 | 3.26e+05 | 1.66e+05 | 2.08e+05 | 1.32e+05 | 1.21e+05 |
| 9 | 9.81e+05 | 6.16e+05 | 7.26e+05 | 1.93e+05 | 3.06e+05 | 1.72e+05 | 1.81e+05 | 1.16e+05 | 1.14e+05 |
| 10 | 9.81e+05 | 6.25e+05 | 6.87e+05 | 2.14e+05 | 2.96e+05 | 1.63e+05 | 1.66e+05 | 1.09e+05 | 1.06e+05 |
| 11 | 9.81e+05 | 6.25e+05 | 6.91e+05 | 2.02e+05 | 2.92e+05 | 1.42e+05 | 1.52e+05 | 1.04e+05 | 1.06e+05 |
| 12 | 9.81e+05 | 6.07e+05 | 6.62e+05 | 2.22e+05 | 2.85e+05 | 1.34e+05 | 1.46e+05 | 9.57e+04 | 9.61e+04 |
| 13 | 9.81e+05 | 6.45e+05 | 6.50e+05 | 1.53e+05 | 2.65e+05 | 1.23e+05 | 1.36e+05 | 8.86e+04 | 8.15e+04 |
| 14 | 9.81e+05 | 6.21e+05 | 6.42e+05 | 1.45e+05 | 2.33e+05 | 1.15e+05 | 1.30e+05 | 7.99e+04 | 7.46e+04 |
| 15 | 9.81e+05 | 6.16e+05 | 6.30e+05 | 1.36e+05 | 2.39e+05 | 1.07e+05 | 1.21e+05 | 7.60e+04 | 7.44e+04 |
| 16 | 9.81e+05 | 6.01e+05 | 6.44e+05 | 1.28e+05 | 2.26e+05 | 9.92e+04 | 1.12e+05 | 6.46e+04 | 6.62e+04 |
| 17 | 9.81e+05 | 5.73e+05 | 6.39e+05 | 1.24e+05 | 2.20e+05 | 9.56e+04 | 1.06e+05 | 6.40e+04 | 6.21e+04 |
| 18 | 9.81e+05 | 5.53e+05 | 6.24e+05 | 1.96e+05 | 2.07e+05 | 9.23e+04 | 1.06e+05 | 5.82e+04 | 6.03e+04 |
| 19 | 9.81e+05 | 5.46e+05 | 6.29e+05 | 1.92e+05 | 2.02e+05 | 8.54e+04 | 9.75e+04 | 5.70e+04 | 5.67e+04 |
| 20 | 9.81e+05 | 5.35e+05 | 6.14e+05 | 1.87e+05 | 2.01e+05 | 8.23e+04 | 9.44e+04 | 5.12e+04 | 5.37e+04 |
| 21 | 9.81e+05 | 5.20e+05 | 5.93e+05 | 2.06e+05 | 1.95e+05 | 8.05e+04 | 9.44e+04 | 4.67e+04 | 5.16e+04 |
| 22 | 9.81e+05 | 5.22e+05 | 6.09e+05 | 2.06e+05 | 1.89e+05 | 7.71e+04 | 8.92e+04 | 5.14e+04 | 4.87e+04 |
| 23 | 9.81e+05 | 5.16e+05 | 6.00e+05 | 2.06e+05 | 1.87e+05 | 7.59e+04 | 8.62e+04 | 4.54e+04 | 4.60e+04 |
| 24 | 9.81e+05 | 5.12e+05 | 6.08e+05 | 2.06e+05 | 1.79e+05 | 7.11e+04 | 8.52e+04 | 4.61e+04 | 4.22e+04 |

Table B.19: Mean number of nodes for the learned partitioning strategy: comparison of the scores $\text{score}_{\text{bal}}$ and $\text{score}_{\text{max}}$, and the impact of the output predicted by the learning algorithm (either the number of nodes, or the logarithm of the number of nodes).

| | | Parallelized: learned partitioning strategy | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Without comm. (c0) | | | | Comm. every 10k nodes (c10k) | | | |
| | | $\text{score}_{\text{bal}}$ | | $\text{score}_{\text{max}}$ | | $\text{score}_{\text{bal}}$ | | $\text{score}_{\text{max}}$ | |
| $N$ | baseline | #nodes | log #nodes | #nodes | log #nodes | #nodes | log #nodes | #nodes | log #nodes |
| 2 | 0 | 2.57e+05 | 2.95e+05 | 4.39e+05 | 4.20e+05 | 4.78e+05 | 4.41e+05 | 3.45e+05 | 4.55e+05 |
| 3 | 0 | 6.36e+05 | 2.55e+05 | 2.87e+05 | 3.32e+05 | 4.46e+05 | 4.53e+05 | 2.86e+05 | 2.49e+05 |
| 4 | 0 | 6.53e+05 | 3.45e+05 | 2.26e+05 | 3.16e+05 | 3.83e+05 | 3.62e+05 | 2.08e+05 | 2.54e+05 |
| 5 | 0 | 5.54e+05 | 3.71e+05 | 2.14e+05 | 3.28e+05 | 3.58e+05 | 3.31e+05 | 1.64e+05 | 1.95e+05 |
| 6 | 0 | 5.06e+05 | 4.68e+05 | 2.01e+05 | 3.25e+05 | 2.87e+05 | 2.90e+05 | 1.62e+05 | 1.76e+05 |
| 7 | 0 | 4.84e+05 | 5.03e+05 | 1.87e+05 | 3.13e+05 | 2.21e+05 | 2.65e+05 | 1.44e+05 | 1.58e+05 |
| 8 | 0 | 4.75e+05 | 4.82e+05 | 1.61e+05 | 2.89e+05 | 2.13e+05 | 2.72e+05 | 1.39e+05 | 1.35e+05 |
| 9 | 0 | 4.70e+05 | 5.76e+05 | 1.62e+05 | 2.63e+05 | 2.16e+05 | 2.54e+05 | 1.31e+05 | 1.29e+05 |
| 10 | 0 | 4.55e+05 | 5.63e+05 | 2.15e+05 | 2.53e+05 | 2.05e+05 | 2.42e+05 | 1.19e+05 | 1.25e+05 |
| 11 | 0 | 5.01e+05 | 5.43e+05 | 1.82e+05 | 2.60e+05 | 1.82e+05 | 2.28e+05 | 1.23e+05 | 1.37e+05 |
| 12 | 0 | 4.96e+05 | 5.41e+05 | 1.96e+05 | 2.48e+05 | 1.78e+05 | 2.23e+05 | 1.11e+05 | 1.13e+05 |
| 13 | 0 | 4.15e+05 | 5.43e+05 | 1.06e+05 | 2.42e+05 | 1.73e+05 | 2.17e+05 | 9.67e+04 | 1.03e+05 |
| 14 | 0 | 4.25e+05 | 5.26e+05 | 1.03e+05 | 2.22e+05 | 1.66e+05 | 2.11e+05 | 8.87e+04 | 9.27e+04 |
| 15 | 0 | 4.59e+05 | 5.09e+05 | 9.69e+04 | 2.17e+05 | 1.59e+05 | 1.97e+05 | 8.06e+04 | 1.04e+05 |
| 16 | 0 | 4.41e+05 | 5.45e+05 | 8.94e+04 | 2.13e+05 | 1.55e+05 | 1.89e+05 | 7.27e+04 | 9.10e+04 |
| 17 | 0 | 4.37e+05 | 5.61e+05 | 8.76e+04 | 2.11e+05 | 1.54e+05 | 1.85e+05 | 7.16e+04 | 8.64e+04 |
| 18 | 0 | 4.23e+05 | 5.52e+05 | 1.83e+05 | 2.03e+05 | 1.51e+05 | 1.85e+05 | 6.42e+04 | 8.37e+04 |
| 19 | 0 | 4.30e+05 | 5.40e+05 | 1.80e+05 | 1.96e+05 | 1.43e+05 | 1.73e+05 | 6.40e+04 | 7.85e+04 |
| 20 | 0 | 4.27e+05 | 5.34e+05 | 1.78e+05 | 1.90e+05 | 1.39e+05 | 1.69e+05 | 5.23e+04 | 7.62e+04 |
| 21 | 0 | 3.97e+05 | 5.17e+05 | 2.03e+05 | 1.89e+05 | 1.40e+05 | 1.72e+05 | 5.23e+04 | 7.49e+04 |
| 22 | 0 | 4.03e+05 | 5.51e+05 | 2.01e+05 | 1.86e+05 | 1.34e+05 | 1.66e+05 | 5.81e+04 | 6.90e+04 |
| 23 | 0 | 4.05e+05 | 5.23e+05 | 1.99e+05 | 1.83e+05 | 1.38e+05 | 1.63e+05 | 5.14e+04 | 6.75e+04 |
| 24 | 0 | 4.09e+05 | 5.32e+05 | 1.95e+05 | 1.67e+05 | 1.29e+05 | 1.63e+05 | 5.79e+04 | 6.02e+04 |

Table B.20: Standard deviation of the number of nodes for the learned partitioning strategy: comparison of the scores $\text{score}_{\text{bal}}$ and $\text{score}_{\text{max}}$, and the impact of the output predicted by the learning algorithm (either the number of nodes, or the logarithm of the number of nodes).

| | | Parallelized: learned partitioning strategy | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Without comm. (c0) | | | | Comm. every 10k nodes (c10k) | | | |
| | | $\text{score}_{\text{bal}}$ | | $\text{score}_{\text{max}}$ | | $\text{score}_{\text{bal}}$ | | $\text{score}_{\text{max}}$ | |
| $N$ | baseline | #nodes | log #nodes | #nodes | log #nodes | #nodes | log #nodes | #nodes | log #nodes |
| 2 | 9.81e+05 | 8.09e+05 | 6.82e+05 | 2.52e+05 | 4.00e+05 | 2.40e+05 | 3.23e+05 | 2.38e+05 | 1.93e+05 |
| 3 | 9.81e+05 | 5.38e+05 | 7.38e+05 | 1.74e+05 | 1.93e+05 | 1.03e+05 | 9.63e+04 | 1.19e+05 | 9.21e+04 |
| 4 | 9.81e+05 | 2.98e+05 | 4.54e+05 | 1.19e+05 | 1.32e+05 | 6.21e+04 | 4.36e+04 | 7.11e+04 | 1.63e+04 |
| 5 | 9.81e+05 | 2.66e+05 | 3.56e+05 | 7.58e+04 | 1.08e+05 | 3.07e+04 | 2.16e+04 | 1.60e+04 | 1.53e+04 |
| 6 | 9.81e+05 | 2.04e+05 | 3.48e+05 | 4.45e+04 | 8.78e+04 | 2.15e+04 | 1.49e+04 | 9.09e+03 | 1.39e+04 |
| 7 | 9.81e+05 | 1.76e+05 | 2.46e+05 | 3.61e+04 | 7.29e+04 | 2.09e+04 | 1.32e+04 | 7.27e+03 | 7.08e+03 |
| 8 | 9.81e+05 | 1.73e+05 | 1.83e+05 | 2.86e+04 | 6.92e+04 | 1.55e+04 | 1.28e+04 | 5.87e+03 | 6.55e+03 |
| 9 | 9.81e+05 | 1.68e+05 | 1.43e+05 | 1.66e+04 | 6.20e+04 | 1.25e+04 | 1.02e+04 | 5.32e+03 | 4.82e+03 |
| 10 | 9.81e+05 | 1.58e+05 | 1.41e+05 | 1.61e+04 | 6.14e+04 | 1.28e+04 | 1.04e+04 | 4.73e+03 | 4.55e+03 |
| 11 | 9.81e+05 | 1.57e+05 | 1.37e+05 | 9.23e+03 | 2.63e+04 | 9.97e+03 | 9.63e+03 | 3.53e+03 | 4.21e+03 |
| 12 | 9.81e+05 | 1.56e+05 | 1.19e+05 | 7.70e+03 | 2.60e+04 | 9.64e+03 | 9.34e+03 | 2.67e+03 | 3.93e+03 |
| 13 | 9.81e+05 | 1.72e+05 | 1.12e+05 | 2.53e+04 | 2.43e+04 | 7.34e+03 | 8.73e+03 | 2.58e+03 | 3.14e+03 |
| 14 | 9.81e+05 | 1.66e+05 | 1.10e+05 | 2.52e+04 | 3.35e+04 | 6.60e+03 | 9.11e+03 | 2.52e+03 | 3.36e+03 |
| 15 | 9.81e+05 | 1.63e+05 | 1.10e+05 | 2.32e+04 | 1.46e+04 | 6.42e+03 | 7.78e+03 | 2.49e+03 | 1.98e+03 |
| 16 | 9.81e+05 | 1.59e+05 | 9.85e+04 | 2.31e+04 | 1.44e+04 | 5.83e+03 | 7.33e+03 | 2.53e+03 | 1.92e+03 |
| 17 | 9.81e+05 | 1.07e+05 | 1.00e+05 | 1.96e+04 | 1.44e+04 | 6.17e+03 | 5.68e+03 | 2.44e+03 | 1.87e+03 |
| 18 | 9.81e+05 | 9.95e+04 | 9.75e+04 | 7.06e+03 | 1.44e+04 | 5.70e+03 | 6.25e+03 | 2.49e+03 | 1.86e+03 |
| 19 | 9.81e+05 | 9.21e+04 | 9.71e+04 | 6.91e+03 | 1.43e+04 | 5.56e+03 | 6.79e+03 | 2.30e+03 | 1.76e+03 |
| 20 | 9.81e+05 | 8.98e+04 | 9.71e+04 | 6.66e+03 | 1.39e+04 | 4.91e+03 | 6.53e+03 | 2.02e+03 | 1.50e+03 |
| 21 | 9.81e+05 | 8.98e+04 | 9.53e+04 | 4.77e+03 | 1.31e+04 | 4.40e+03 | 5.78e+03 | 1.99e+03 | 1.73e+03 |
| 22 | 9.81e+05 | 7.42e+04 | 9.53e+04 | 3.74e+03 | 1.21e+04 | 4.49e+03 | 5.41e+03 | 1.71e+03 | 1.23e+03 |
| 23 | 9.81e+05 | 7.42e+04 | 9.53e+04 | 3.74e+03 | 1.07e+04 | 4.84e+03 | 5.26e+03 | 1.79e+03 | 988 |
| 24 | 9.81e+05 | 7.34e+04 | 9.50e+04 | 3.69e+03 | 1.07e+04 | 5.21e+03 | 5.38e+03 | 1.62e+03 | 1.19e+03 |

Table B.21: Minimum number of nodes for the learned partitioning strategy: comparison of the scores $\text{score}_{\text{bal}}$ and $\text{score}_{\text{max}}$, and the impact of the output predicted by the learning algorithm (either the number of nodes, or the logarithm of the number of nodes).

| | | Parallelized: learned partitioning strategy | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Without comm. (c0) | | | | Comm. every 10k nodes (c10k) | | | |
| | | $\text{score}_{\text{bal}}$ | | $\text{score}_{\text{max}}$ | | $\text{score}_{\text{bal}}$ | | $\text{score}_{\text{max}}$ | |
| $N$ | baseline | #nodes | log #nodes | #nodes | log #nodes | #nodes | log #nodes | #nodes | log #nodes |
| 2 | 9.81e+05 | 1.17e+06 | 1.10e+06 | 8.73e+05 | 9.94e+05 | 9.15e+05 | 9.46e+05 | 7.26e+05 | 8.36e+05 |
| 3 | 9.81e+05 | 1.69e+06 | 1.23e+06 | 7.14e+05 | 8.03e+05 | 9.01e+05 | 9.47e+05 | 6.45e+05 | 5.46e+05 |
| 4 | 9.81e+05 | 1.77e+06 | 1.20e+06 | 6.09e+05 | 8.00e+05 | 8.56e+05 | 8.24e+05 | 5.04e+05 | 5.58e+05 |
| 5 | 9.81e+05 | 1.67e+06 | 1.22e+06 | 5.83e+05 | 8.36e+05 | 8.56e+05 | 7.64e+05 | 4.21e+05 | 4.74e+05 |
| 6 | 9.81e+05 | 1.56e+06 | 1.59e+06 | 5.41e+05 | 8.50e+05 | 7.23e+05 | 7.08e+05 | 4.29e+05 | 4.42e+05 |
| 7 | 9.81e+05 | 1.54e+06 | 1.63e+06 | 5.28e+05 | 8.60e+05 | 5.83e+05 | 6.78e+05 | 3.97e+05 | 4.14e+05 |
| 8 | 9.81e+05 | 1.55e+06 | 1.58e+06 | 4.76e+05 | 8.52e+05 | 6.02e+05 | 7.25e+05 | 4.03e+05 | 3.54e+05 |
| 9 | 9.81e+05 | 1.58e+06 | 1.89e+06 | 4.80e+05 | 8.36e+05 | 6.03e+05 | 6.97e+05 | 3.94e+05 | 3.44e+05 |
| 10 | 9.81e+05 | 1.58e+06 | 1.89e+06 | 6.90e+05 | 8.24e+05 | 6.01e+05 | 6.96e+05 | 3.74e+05 | 3.38e+05 |
| 11 | 9.81e+05 | 1.80e+06 | 1.85e+06 | 5.87e+05 | 8.16e+05 | 5.36e+05 | 6.76e+05 | 3.89e+05 | 4.04e+05 |
| 12 | 9.81e+05 | 1.87e+06 | 1.90e+06 | 5.86e+05 | 8.06e+05 | 5.32e+05 | 6.59e+05 | 3.56e+05 | 3.24e+05 |
| 13 | 9.81e+05 | 1.47e+06 | 1.96e+06 | 3.80e+05 | 7.88e+05 | 5.40e+05 | 6.70e+05 | 3.11e+05 | 3.22e+05 |
| 14 | 9.81e+05 | 1.55e+06 | 1.88e+06 | 3.79e+05 | 7.59e+05 | 5.33e+05 | 6.75e+05 | 2.86e+05 | 2.79e+05 |
| 15 | 9.81e+05 | 1.71e+06 | 1.88e+06 | 3.72e+05 | 7.60e+05 | 5.33e+05 | 6.46e+05 | 2.66e+05 | 3.41e+05 |
| 16 | 9.81e+05 | 1.69e+06 | 2.08e+06 | 3.43e+05 | 7.43e+05 | 5.24e+05 | 6.31e+05 | 2.66e+05 | 3.06e+05 |
| 17 | 9.81e+05 | 1.69e+06 | 2.23e+06 | 3.36e+05 | 7.54e+05 | 5.27e+05 | 6.36e+05 | 2.66e+05 | 2.96e+05 |
| 18 | 9.81e+05 | 1.67e+06 | 2.24e+06 | 6.56e+05 | 7.43e+05 | 5.33e+05 | 6.45e+05 | 2.45e+05 | 2.85e+05 |
| 19 | 9.81e+05 | 1.73e+06 | 2.19e+06 | 6.59e+05 | 7.23e+05 | 5.24e+05 | 6.23e+05 | 2.46e+05 | 2.73e+05 |
| 20 | 9.81e+05 | 1.71e+06 | 2.19e+06 | 6.57e+05 | 6.98e+05 | 5.09e+05 | 6.17e+05 | 1.91e+05 | 2.72e+05 |
| 21 | 9.81e+05 | 1.61e+06 | 2.19e+06 | 7.55e+05 | 6.90e+05 | 5.20e+05 | 6.36e+05 | 2.13e+05 | 2.71e+05 |
| 22 | 9.81e+05 | 1.60e+06 | 2.32e+06 | 7.60e+05 | 6.98e+05 | 5.18e+05 | 6.29e+05 | 2.23e+05 | 2.58e+05 |
| 23 | 9.81e+05 | 1.61e+06 | 2.16e+06 | 7.52e+05 | 6.90e+05 | 5.33e+05 | 6.25e+05 | 2.03e+05 | 2.52e+05 |
| 24 | 9.81e+05 | 1.61e+06 | 2.28e+06 | 7.43e+05 | 6.40e+05 | 5.14e+05 | 6.37e+05 | 2.32e+05 | 2.31e+05 |

Table B.22: Maximum number of nodes for the learned partitioning strategy: comparison of the scores $\text{score}_{\text{bal}}$ and $\text{score}_{\text{max}}$, and the impact of the output predicted by the learning algorithm (either the number of nodes, or the logarithm of the number of nodes).

| | | Parallelized: learned partitioning strategy | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Without comm. (c0) | | | | Comm. every 10k nodes (c10k) | | | |
| | | $\text{score}_{\text{bal}}$ | | $\text{score}_{\text{max}}$ | | $\text{score}_{\text{bal}}$ | | $\text{score}_{\text{max}}$ | |
| $N$ | baseline | #nodes | log #nodes | #nodes | log #nodes | #nodes | log #nodes | #nodes | log #nodes |
| 2 | 9.81e+05 | 1.98e+06 | 1.78e+06 | 1.12e+06 | 1.39e+06 | 1.16e+06 | 1.27e+06 | 9.64e+05 | 1.03e+06 |
| 3 | 9.81e+05 | 3.01e+06 | 2.94e+06 | 1.21e+06 | 1.44e+06 | 1.21e+06 | 1.46e+06 | 9.91e+05 | 8.37e+05 |
| 4 | 9.81e+05 | 3.48e+06 | 3.38e+06 | 1.31e+06 | 1.90e+06 | 1.24e+06 | 1.46e+06 | 9.56e+05 | 9.61e+05 |
| 5 | 9.81e+05 | 3.90e+06 | 3.86e+06 | 1.43e+06 | 2.14e+06 | 1.28e+06 | 1.48e+06 | 8.91e+05 | 1.02e+06 |
| 6 | 9.81e+05 | 4.14e+06 | 4.96e+06 | 1.53e+06 | 2.41e+06 | 1.29e+06 | 1.49e+06 | 9.95e+05 | 1.02e+06 |
| 7 | 9.81e+05 | 4.41e+06 | 5.22e+06 | 1.60e+06 | 2.54e+06 | 1.32e+06 | 1.52e+06 | 1.04e+06 | 9.64e+05 |
| 8 | 9.81e+05 | 4.73e+06 | 5.72e+06 | 1.53e+06 | 2.61e+06 | 1.33e+06 | 1.67e+06 | 1.02e+06 | 9.71e+05 |
| 9 | 9.81e+05 | 5.54e+06 | 6.53e+06 | 1.69e+06 | 2.76e+06 | 1.55e+06 | 1.63e+06 | 1.02e+06 | 1.03e+06 |
| 10 | 9.81e+05 | 6.25e+06 | 6.87e+06 | 2.06e+06 | 2.94e+06 | 1.63e+06 | 1.66e+06 | 1.07e+06 | 1.04e+06 |
| 11 | 9.81e+05 | 6.88e+06 | 7.61e+06 | 2.14e+06 | 3.14e+06 | 1.56e+06 | 1.67e+06 | 1.12e+06 | 1.13e+06 |
| 12 | 9.81e+05 | 7.28e+06 | 7.94e+06 | 2.56e+06 | 3.35e+06 | 1.61e+06 | 1.75e+06 | 1.12e+06 | 1.12e+06 |
| 13 | 9.81e+05 | 8.39e+06 | 8.45e+06 | 1.99e+06 | 3.38e+06 | 1.60e+06 | 1.77e+06 | 1.12e+06 | 1.03e+06 |
| 14 | 9.81e+05 | 8.70e+06 | 8.98e+06 | 2.03e+06 | 3.23e+06 | 1.60e+06 | 1.82e+06 | 1.10e+06 | 1.03e+06 |
| 15 | 9.81e+05 | 9.24e+06 | 9.44e+06 | 2.04e+06 | 3.52e+06 | 1.60e+06 | 1.81e+06 | 1.11e+06 | 1.09e+06 |
| 16 | 9.81e+05 | 9.62e+06 | 1.03e+07 | 2.04e+06 | 3.56e+06 | 1.59e+06 | 1.79e+06 | 1.02e+06 | 1.03e+06 |
| 17 | 9.81e+05 | 9.74e+06 | 1.09e+07 | 2.11e+06 | 3.68e+06 | 1.62e+06 | 1.80e+06 | 1.07e+06 | 1.03e+06 |
| 18 | 9.81e+05 | 9.96e+06 | 1.12e+07 | 3.43e+06 | 3.66e+06 | 1.66e+06 | 1.90e+06 | 1.03e+06 | 1.05e+06 |
| 19 | 9.81e+05 | 1.04e+07 | 1.20e+07 | 3.56e+06 | 3.76e+06 | 1.62e+06 | 1.85e+06 | 1.07e+06 | 1.04e+06 |
| 20 | 9.81e+05 | 1.07e+07 | 1.23e+07 | 3.57e+06 | 3.94e+06 | 1.65e+06 | 1.89e+06 | 1.00e+06 | 1.04e+06 |
| 21 | 9.81e+05 | 1.09e+07 | 1.25e+07 | 4.12e+06 | 4.02e+06 | 1.69e+06 | 1.98e+06 | 9.57e+05 | 1.04e+06 |
| 22 | 9.81e+05 | 1.15e+07 | 1.34e+07 | 4.33e+06 | 4.08e+06 | 1.70e+06 | 1.96e+06 | 1.10e+06 | 1.03e+06 |
| 23 | 9.81e+05 | 1.19e+07 | 1.38e+07 | 4.45e+06 | 4.21e+06 | 1.74e+06 | 1.98e+06 | 1.01e+06 | 1.01e+06 |
| 24 | 9.81e+05 | 1.23e+07 | 1.46e+07 | 4.65e+06 | 4.22e+06 | 1.71e+06 | 2.04e+06 | 1.07e+06 | 9.75e+05 |

Table B.23: Total number of nodes for the learned partitioning strategy: comparison of the scores $\text{score}_{\text{bal}}$ and $\text{score}_{\text{max}}$, and the impact of the output predicted by the learning algorithm (either the number of nodes, or the logarithm of the number of nodes).

| | | Parallelized: learned partitioning strategy | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Without comm. (c0) | | | | Comm. every 10k nodes (c10k) | | | |
| $N$ | baseline | probe 50 | probe 500 | probe 5k | probe 50k | probe 50 | probe 500 | probe 5k | probe 50k |
| 2 | 9.81e+05 | 6.07e+05 | 6.99e+05 | 5.62e+05 | 4.86e+05 | 4.95e+05 | 5.24e+05 | 4.82e+05 | 4.36e+05 |
| 3 | 9.81e+05 | 5.38e+05 | 5.81e+05 | 4.03e+05 | 4.00e+05 | 3.55e+05 | 3.46e+05 | 3.30e+05 | 2.85e+05 |
| 4 | 9.81e+05 | 5.98e+05 | 4.72e+05 | 3.27e+05 | 2.99e+05 | 2.84e+05 | 2.81e+05 | 2.39e+05 | 1.97e+05 |
| 5 | 9.81e+05 | 6.15e+05 | 3.88e+05 | 2.86e+05 | 2.39e+05 | 2.27e+05 | 2.08e+05 | 1.78e+05 | 1.51e+05 |
| 6 | 9.81e+05 | 6.81e+05 | 3.74e+05 | 2.56e+05 | 2.07e+05 | 1.97e+05 | 1.79e+05 | 1.66e+05 | 1.18e+05 |
| 7 | 9.81e+05 | 6.40e+05 | 3.33e+05 | 2.28e+05 | 1.93e+05 | 1.81e+05 | 1.53e+05 | 1.48e+05 | 1.15e+05 |
| 8 | 9.81e+05 | 7.21e+05 | 2.99e+05 | 1.96e+05 | 1.68e+05 | 1.62e+05 | 1.37e+05 | 1.32e+05 | 9.89e+04 |
| 9 | 9.81e+05 | 6.83e+05 | 2.81e+05 | 1.93e+05 | 1.54e+05 | 1.53e+05 | 1.21e+05 | 1.16e+05 | 8.76e+04 |
| 10 | 9.81e+05 | 6.83e+05 | 2.72e+05 | 2.14e+05 | 1.40e+05 | 1.44e+05 | 1.16e+05 | 1.09e+05 | 7.89e+04 |
| 11 | 9.81e+05 | 6.46e+05 | 2.50e+05 | 2.02e+05 | 1.33e+05 | 1.21e+05 | 1.09e+05 | 1.04e+05 | 7.16e+04 |
| 12 | 9.81e+05 | 6.07e+05 | 2.30e+05 | 2.22e+05 | 1.29e+05 | 1.13e+05 | 9.71e+04 | 9.57e+04 | 6.77e+04 |
| 13 | 9.81e+05 | 5.79e+05 | 2.21e+05 | 1.53e+05 | 1.14e+05 | 1.11e+05 | 9.54e+04 | 8.86e+04 | 6.20e+04 |
| 14 | 9.81e+05 | 5.68e+05 | 2.11e+05 | 1.45e+05 | 1.11e+05 | 1.04e+05 | 8.57e+04 | 7.99e+04 | 6.06e+04 |
| 15 | 9.81e+05 | 5.41e+05 | 2.09e+05 | 1.36e+05 | 1.09e+05 | 9.93e+04 | 8.26e+04 | 7.60e+04 | 5.88e+04 |
| 16 | 9.81e+05 | 5.21e+05 | 2.07e+05 | 1.28e+05 | 1.04e+05 | 8.97e+04 | 8.22e+04 | 6.46e+04 | 5.53e+04 |
| 17 | 9.81e+05 | 5.14e+05 | 2.04e+05 | 1.24e+05 | 1.01e+05 | 8.92e+04 | 7.70e+04 | 6.40e+04 | 5.49e+04 |
| 18 | 9.81e+05 | 5.60e+05 | 1.99e+05 | 1.96e+05 | 9.59e+04 | 8.56e+04 | 7.36e+04 | 5.82e+04 | 5.16e+04 |
| 19 | 9.81e+05 | 5.43e+05 | 1.92e+05 | 1.92e+05 | 9.57e+04 | 8.08e+04 | 7.01e+04 | 5.70e+04 | 4.95e+04 |
| 20 | 9.81e+05 | 5.25e+05 | 1.91e+05 | 1.87e+05 | 9.89e+04 | 7.77e+04 | 6.51e+04 | 5.12e+04 | 4.73e+04 |
| 21 | 9.81e+05 | 5.08e+05 | 1.83e+05 | 2.06e+05 | 9.37e+04 | 7.43e+04 | 6.17e+04 | 4.67e+04 | 4.74e+04 |
| 22 | 9.81e+05 | 4.94e+05 | 1.83e+05 | 2.06e+05 | 8.93e+04 | 7.08e+04 | 6.05e+04 | 5.14e+04 | 4.29e+04 |
| 23 | 9.81e+05 | 4.85e+05 | 1.83e+05 | 2.06e+05 | 8.66e+04 | 6.91e+04 | 5.87e+04 | 4.54e+04 | 4.13e+04 |
| 24 | 9.81e+05 | 4.76e+05 | 1.83e+05 | 2.06e+05 | 8.42e+04 | 6.59e+04 | 5.68e+04 | 4.61e+04 | 3.96e+04 |

Table B.24: Mean number of nodes for the learned partitioning strategy: impact of the probing size (i.e., the number of nodes explored in order to compute the features describing the subproblem). We compare probing budgets of 50, 500, $5e^3$, and $5e^4$.

| | | Parallelized: learned partitioning strategy | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Without comm. (c0) | | | | Comm. every 10k nodes (c10k) | | | |
| $N$ | baseline | probe 50 | probe 500 | probe 5k | probe 50k | probe 50 | probe 500 | probe 5k | probe 50k |
| 2 | 0 | 2.81e+05 | 3.29e+05 | 4.39e+05 | 3.01e+05 | 2.77e+05 | 3.41e+05 | 3.45e+05 | 2.88e+05 |
| 3 | 0 | 3.06e+05 | 4.26e+05 | 2.87e+05 | 2.68e+05 | 2.16e+05 | 3.41e+05 | 2.86e+05 | 2.02e+05 |
| 4 | 0 | 5.07e+05 | 3.82e+05 | 2.26e+05 | 2.08e+05 | 2.10e+05 | 2.60e+05 | 2.08e+05 | 1.90e+05 |
| 5 | 0 | 6.18e+05 | 3.45e+05 | 2.14e+05 | 1.66e+05 | 1.83e+05 | 2.22e+05 | 1.64e+05 | 1.39e+05 |
| 6 | 0 | 8.20e+05 | 3.89e+05 | 2.01e+05 | 1.31e+05 | 1.62e+05 | 2.10e+05 | 1.62e+05 | 1.06e+05 |
| 7 | 0 | 7.64e+05 | 3.68e+05 | 1.87e+05 | 1.23e+05 | 1.59e+05 | 1.93e+05 | 1.44e+05 | 1.08e+05 |
| 8 | 0 | 8.39e+05 | 3.17e+05 | 1.61e+05 | 9.58e+04 | 1.51e+05 | 1.68e+05 | 1.39e+05 | 8.19e+04 |
| 9 | 0 | 8.11e+05 | 3.12e+05 | 1.62e+05 | 9.13e+04 | 1.43e+05 | 1.53e+05 | 1.31e+05 | 7.35e+04 |
| 10 | 0 | 8.56e+05 | 2.96e+05 | 2.15e+05 | 8.92e+04 | 1.31e+05 | 1.49e+05 | 1.19e+05 | 6.84e+04 |
| 11 | 0 | 8.69e+05 | 2.58e+05 | 1.82e+05 | 8.80e+04 | 1.18e+05 | 1.40e+05 | 1.23e+05 | 6.53e+04 |
| 12 | 0 | 8.13e+05 | 2.43e+05 | 1.96e+05 | 8.45e+04 | 1.16e+05 | 1.20e+05 | 1.11e+05 | 6.15e+04 |
| 13 | 0 | 7.89e+05 | 2.22e+05 | 1.06e+05 | 7.15e+04 | 1.16e+05 | 1.15e+05 | 9.67e+04 | 5.89e+04 |
| 14 | 0 | 7.47e+05 | 2.18e+05 | 1.03e+05 | 7.18e+04 | 1.12e+05 | 1.08e+05 | 8.87e+04 | 5.69e+04 |
| 15 | 0 | 7.26e+05 | 2.17e+05 | 9.69e+04 | 7.45e+04 | 1.08e+05 | 1.05e+05 | 8.06e+04 | 5.87e+04 |
| 16 | 0 | 7.08e+05 | 2.13e+05 | 8.94e+04 | 6.92e+04 | 1.05e+05 | 1.07e+05 | 7.27e+04 | 5.22e+04 |
| 17 | 0 | 6.94e+05 | 2.06e+05 | 8.76e+04 | 6.77e+04 | 1.02e+05 | 1.00e+05 | 7.16e+04 | 5.21e+04 |
| 18 | 0 | 7.61e+05 | 2.02e+05 | 1.83e+05 | 6.59e+04 | 1.03e+05 | 9.56e+04 | 6.42e+04 | 4.92e+04 |
| 19 | 0 | 7.45e+05 | 1.96e+05 | 1.80e+05 | 6.75e+04 | 1.00e+05 | 9.08e+04 | 6.40e+04 | 5.02e+04 |
| 20 | 0 | 7.27e+05 | 1.96e+05 | 1.78e+05 | 6.89e+04 | 9.77e+04 | 8.69e+04 | 5.23e+04 | 4.81e+04 |
| 21 | 0 | 7.15e+05 | 1.91e+05 | 2.03e+05 | 6.66e+04 | 9.49e+04 | 8.42e+04 | 5.23e+04 | 4.94e+04 |
| 22 | 0 | 7.07e+05 | 1.90e+05 | 2.01e+05 | 6.23e+04 | 9.28e+04 | 7.98e+04 | 5.81e+04 | 4.45e+04 |
| 23 | 0 | 7.04e+05 | 1.91e+05 | 1.99e+05 | 5.88e+04 | 9.13e+04 | 8.27e+04 | 5.14e+04 | 4.05e+04 |
| 24 | 0 | 6.68e+05 | 1.93e+05 | 1.95e+05 | 5.67e+04 | 8.93e+04 | 8.05e+04 | 5.79e+04 | 3.65e+04 |

Table B.25: Standard deviation of the number of nodes for the learned partitioning strategy: impact of the probing size (i.e., the number of nodes explored in order to compute the features describing the subproblem). We compare probing budgets of 50, 500, $5e^3$, and $5e^4$.

| | | Parallelized: learned partitioning strategy | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Without comm. (c0) | | | | Comm. every 10k nodes (c10k) | | | |
| $N$ | baseline | probe 50 | probe 500 | probe 5k | probe 50k | probe 50 | probe 500 | probe 5k | probe 50k |
| 2 | 9.81e+05 | 4.08e+05 | 4.66e+05 | 2.52e+05 | 2.73e+05 | 3.00e+05 | 2.83e+05 | 2.38e+05 | 2.32e+05 |
| 3 | 9.81e+05 | 2.77e+05 | 1.41e+05 | 1.74e+05 | 1.61e+05 | 1.64e+05 | 6.27e+04 | 1.19e+05 | 1.02e+05 |
| 4 | 9.81e+05 | 1.96e+05 | 1.08e+05 | 1.19e+05 | 9.89e+04 | 7.62e+04 | 6.56e+04 | 7.11e+04 | 4.51e+04 |
| 5 | 9.81e+05 | 1.22e+05 | 5.68e+04 | 7.58e+04 | 8.46e+04 | 5.44e+04 | 2.55e+04 | 1.60e+04 | 4.02e+04 |
| 6 | 9.81e+05 | 8.99e+04 | 5.73e+04 | 4.45e+04 | 7.37e+04 | 4.79e+04 | 2.66e+04 | 9.09e+03 | 2.85e+04 |
| 7 | 9.81e+05 | 9.26e+04 | 4.35e+04 | 3.61e+04 | 6.31e+04 | 2.42e+04 | 1.44e+04 | 7.27e+03 | 2.60e+04 |
| 8 | 9.81e+05 | 9.19e+04 | 3.93e+04 | 2.86e+04 | 5.31e+04 | 1.53e+04 | 1.32e+04 | 5.87e+03 | 2.30e+04 |
| 9 | 9.81e+05 | 6.28e+04 | 3.03e+04 | 1.66e+04 | 4.00e+04 | 1.17e+04 | 9.58e+03 | 5.32e+03 | 1.06e+04 |
| 10 | 9.81e+05 | 5.35e+04 | 2.97e+04 | 1.61e+04 | 3.66e+04 | 8.72e+03 | 7.31e+03 | 4.73e+03 | 9.83e+03 |
| 11 | 9.81e+05 | 5.29e+04 | 2.21e+04 | 9.23e+03 | 3.56e+04 | 6.37e+03 | 3.04e+03 | 3.53e+03 | 8.61e+03 |
| 12 | 9.81e+05 | 4.31e+04 | 1.35e+04 | 7.70e+03 | 3.54e+04 | 6.30e+03 | 2.35e+03 | 2.67e+03 | 6.85e+03 |
| 13 | 9.81e+05 | 4.16e+04 | 1.25e+04 | 2.53e+04 | 3.17e+04 | 7.26e+03 | 2.42e+03 | 2.58e+03 | 5.74e+03 |
| 14 | 9.81e+05 | 4.10e+04 | 4.65e+03 | 2.52e+04 | 2.87e+04 | 6.09e+03 | 1.48e+03 | 2.52e+03 | 5.94e+03 |
| 15 | 9.81e+05 | 4.10e+04 | 4.41e+03 | 2.32e+04 | 2.63e+04 | 6.05e+03 | 1.44e+03 | 2.49e+03 | 5.22e+03 |
| 16 | 9.81e+05 | 3.96e+04 | 4.40e+03 | 2.31e+04 | 2.44e+04 | 5.72e+03 | 1.42e+03 | 2.53e+03 | 4.81e+03 |
| 17 | 9.81e+05 | 3.96e+04 | 4.34e+03 | 1.96e+04 | 2.43e+04 | 5.38e+03 | 1.42e+03 | 2.44e+03 | 4.96e+03 |
| 18 | 9.81e+05 | 3.96e+04 | 4.34e+03 | 7.06e+03 | 2.17e+04 | 5.82e+03 | 1.42e+03 | 2.49e+03 | 3.43e+03 |
| 19 | 9.81e+05 | 3.20e+04 | 4.23e+03 | 6.91e+03 | 1.59e+04 | 4.04e+03 | 1.44e+03 | 2.30e+03 | 3.03e+03 |
| 20 | 9.81e+05 | 2.80e+04 | 4.07e+03 | 6.66e+03 | 1.56e+04 | 2.99e+03 | 1.45e+03 | 2.02e+03 | 3.55e+03 |
| 21 | 9.81e+05 | 2.49e+04 | 3.49e+03 | 4.77e+03 | 1.16e+04 | 2.86e+03 | 1.40e+03 | 1.99e+03 | 1.76e+03 |
| 22 | 9.81e+05 | 2.26e+04 | 3.49e+03 | 3.74e+03 | 1.12e+04 | 2.28e+03 | 1.39e+03 | 1.71e+03 | 1.82e+03 |
| 23 | 9.81e+05 | 2.26e+04 | 3.48e+03 | 3.74e+03 | 1.01e+04 | 2.26e+03 | 1.38e+03 | 1.79e+03 | 1.70e+03 |
| 24 | 9.81e+05 | 2.26e+04 | 3.48e+03 | 3.69e+03 | 9.90e+03 | 2.21e+03 | 1.38e+03 | 1.62e+03 | 1.68e+03 |

Table B.26: Minimum number of nodes for the learned partitioning strategy: impact of the probing size (i.e., the number of nodes explored in order to compute the features describing the subproblem). We compare probing budgets of 50, 500, $5e^3$, and $5e^4$.

| | | Parallelized: learned partitioning strategy | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Without comm. (c0) | | | | Comm. every 10k nodes (c10k) | | | |
| $N$ | baseline | probe 50 | probe 500 | probe 5k | probe 50k | probe 50 | probe 500 | probe 5k | probe 50k |
| 2 | 9.81e+05 | 8.06e+05 | 9.32e+05 | 8.73e+05 | 6.98e+05 | 6.91e+05 | 7.65e+05 | 7.26e+05 | 6.39e+05 |
| 3 | 9.81e+05 | 8.62e+05 | 9.51e+05 | 7.14e+05 | 6.70e+05 | 5.74e+05 | 7.03e+05 | 6.45e+05 | 4.91e+05 |
| 4 | 9.81e+05 | 1.28e+06 | 8.99e+05 | 6.09e+05 | 5.50e+05 | 5.33e+05 | 6.23e+05 | 5.04e+05 | 4.54e+05 |
| 5 | 9.81e+05 | 1.60e+06 | 8.65e+05 | 5.83e+05 | 4.67e+05 | 4.77e+05 | 5.54e+05 | 4.21e+05 | 3.71e+05 |
| 6 | 9.81e+05 | 2.21e+06 | 1.01e+06 | 5.41e+05 | 4.03e+05 | 4.52e+05 | 5.55e+05 | 4.29e+05 | 2.91e+05 |
| 7 | 9.81e+05 | 2.21e+06 | 1.01e+06 | 5.28e+05 | 3.94e+05 | 4.32e+05 | 5.32e+05 | 3.97e+05 | 3.11e+05 |
| 8 | 9.81e+05 | 2.39e+06 | 9.24e+05 | 4.76e+05 | 3.24e+05 | 4.26e+05 | 4.77e+05 | 4.03e+05 | 2.60e+05 |
| 9 | 9.81e+05 | 2.39e+06 | 9.25e+05 | 4.80e+05 | 3.09e+05 | 4.22e+05 | 4.49e+05 | 3.94e+05 | 2.30e+05 |
| 10 | 9.81e+05 | 2.69e+06 | 8.89e+05 | 6.90e+05 | 3.03e+05 | 3.99e+05 | 4.50e+05 | 3.74e+05 | 2.26e+05 |
| 11 | 9.81e+05 | 2.81e+06 | 7.93e+05 | 5.87e+05 | 3.10e+05 | 3.79e+05 | 4.44e+05 | 3.89e+05 | 2.20e+05 |
| 12 | 9.81e+05 | 2.69e+06 | 7.87e+05 | 5.86e+05 | 3.02e+05 | 3.93e+05 | 3.90e+05 | 3.56e+05 | 2.13e+05 |
| 13 | 9.81e+05 | 2.68e+06 | 7.34e+05 | 3.80e+05 | 2.56e+05 | 3.96e+05 | 3.78e+05 | 3.11e+05 | 2.08e+05 |
| 14 | 9.81e+05 | 2.53e+06 | 7.39e+05 | 3.79e+05 | 2.53e+05 | 3.82e+05 | 3.61e+05 | 2.86e+05 | 2.08e+05 |
| 15 | 9.81e+05 | 2.52e+06 | 7.41e+05 | 3.72e+05 | 2.56e+05 | 3.77e+05 | 3.70e+05 | 2.66e+05 | 2.10e+05 |
| 16 | 9.81e+05 | 2.52e+06 | 7.37e+05 | 3.43e+05 | 2.54e+05 | 3.78e+05 | 3.83e+05 | 2.66e+05 | 1.86e+05 |
| 17 | 9.81e+05 | 2.54e+06 | 7.40e+05 | 3.36e+05 | 2.54e+05 | 3.68e+05 | 3.54e+05 | 2.66e+05 | 1.91e+05 |
| 18 | 9.81e+05 | 2.69e+06 | 7.41e+05 | 6.56e+05 | 2.54e+05 | 3.83e+05 | 3.57e+05 | 2.45e+05 | 1.85e+05 |
| 19 | 9.81e+05 | 2.69e+06 | 7.41e+05 | 6.59e+05 | 2.57e+05 | 3.76e+05 | 3.43e+05 | 2.46e+05 | 1.91e+05 |
| 20 | 9.81e+05 | 2.66e+06 | 7.46e+05 | 6.57e+05 | 2.52e+05 | 3.70e+05 | 3.41e+05 | 1.91e+05 | 1.91e+05 |
| 21 | 9.81e+05 | 2.67e+06 | 7.42e+05 | 7.55e+05 | 2.50e+05 | 3.71e+05 | 3.35e+05 | 2.13e+05 | 1.98e+05 |
| 22 | 9.81e+05 | 2.67e+06 | 7.44e+05 | 7.60e+05 | 2.49e+05 | 3.67e+05 | 3.21e+05 | 2.23e+05 | 1.83e+05 |
| 23 | 9.81e+05 | 2.69e+06 | 7.53e+05 | 7.52e+05 | 2.43e+05 | 3.70e+05 | 3.44e+05 | 2.03e+05 | 1.64e+05 |
| 24 | 9.81e+05 | 2.69e+06 | 7.64e+05 | 7.43e+05 | 2.38e+05 | 3.70e+05 | 3.40e+05 | 2.32e+05 | 1.39e+05 |

Table B.27: Maximum number of nodes for the learned partitioning strategy: impact of the probing size (i.e., the number of nodes explored in order to compute the features describing the subproblem). We compare probing budgets of 50, 500, $5e^3$, and $5e^4$.

| | | Parallelized: learned partitioning strategy | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Without comm. (c0) | | | | Comm. every 10k nodes (c10k) | | | |
| $N$ | baseline | probe 50 | probe 500 | probe 5k | probe 50k | probe 50 | probe 500 | probe 5k | probe 50k |
| 2 | 9.81e+05 | 1.21e+06 | 1.40e+06 | 1.12e+06 | 9.71e+05 | 9.90e+05 | 1.05e+06 | 9.64e+05 | 8.71e+05 |
| 3 | 9.81e+05 | 1.61e+06 | 1.74e+06 | 1.21e+06 | 1.20e+06 | 1.06e+06 | 1.04e+06 | 9.91e+05 | 8.55e+05 |
| 4 | 9.81e+05 | 2.39e+06 | 1.89e+06 | 1.31e+06 | 1.20e+06 | 1.14e+06 | 1.12e+06 | 9.56e+05 | 7.89e+05 |
| 5 | 9.81e+05 | 3.07e+06 | 1.94e+06 | 1.43e+06 | 1.19e+06 | 1.13e+06 | 1.04e+06 | 8.91e+05 | 7.53e+05 |
| 6 | 9.81e+05 | 4.09e+06 | 2.25e+06 | 1.53e+06 | 1.24e+06 | 1.18e+06 | 1.08e+06 | 9.95e+05 | 7.11e+05 |
| 7 | 9.81e+05 | 4.48e+06 | 2.33e+06 | 1.60e+06 | 1.35e+06 | 1.27e+06 | 1.07e+06 | 1.04e+06 | 8.04e+05 |
| 8 | 9.81e+05 | 5.77e+06 | 2.39e+06 | 1.53e+06 | 1.35e+06 | 1.30e+06 | 1.10e+06 | 1.02e+06 | 7.91e+05 |
| 9 | 9.81e+05 | 6.15e+06 | 2.53e+06 | 1.69e+06 | 1.39e+06 | 1.38e+06 | 1.09e+06 | 1.02e+06 | 7.88e+05 |
| 10 | 9.81e+05 | 6.81e+06 | 2.72e+06 | 2.06e+06 | 1.40e+06 | 1.43e+06 | 1.16e+06 | 1.07e+06 | 7.89e+05 |
| 11 | 9.81e+05 | 7.09e+06 | 2.75e+06 | 2.14e+06 | 1.46e+06 | 1.32e+06 | 1.20e+06 | 1.12e+06 | 7.86e+05 |
| 12 | 9.81e+05 | 7.26e+06 | 2.76e+06 | 2.56e+06 | 1.55e+06 | 1.35e+06 | 1.16e+06 | 1.12e+06 | 8.11e+05 |
| 13 | 9.81e+05 | 7.48e+06 | 2.87e+06 | 1.99e+06 | 1.48e+06 | 1.42e+06 | 1.24e+06 | 1.12e+06 | 8.05e+05 |
| 14 | 9.81e+05 | 7.91e+06 | 2.95e+06 | 2.03e+06 | 1.55e+06 | 1.44e+06 | 1.20e+06 | 1.10e+06 | 8.47e+05 |
| 15 | 9.81e+05 | 8.08e+06 | 3.14e+06 | 2.04e+06 | 1.63e+06 | 1.47e+06 | 1.24e+06 | 1.11e+06 | 8.81e+05 |
| 16 | 9.81e+05 | 8.30e+06 | 3.27e+06 | 2.04e+06 | 1.67e+06 | 1.42e+06 | 1.30e+06 | 1.02e+06 | 8.84e+05 |
| 17 | 9.81e+05 | 8.69e+06 | 3.41e+06 | 2.11e+06 | 1.72e+06 | 1.50e+06 | 1.28e+06 | 1.07e+06 | 9.33e+05 |
| 18 | 9.81e+05 | 1.00e+07 | 3.51e+06 | 3.43e+06 | 1.73e+06 | 1.53e+06 | 1.30e+06 | 1.03e+06 | 9.28e+05 |
| 19 | 9.81e+05 | 1.03e+07 | 3.57e+06 | 3.56e+06 | 1.82e+06 | 1.52e+06 | 1.30e+06 | 1.07e+06 | 9.39e+05 |
| 20 | 9.81e+05 | 1.04e+07 | 3.73e+06 | 3.57e+06 | 1.98e+06 | 1.54e+06 | 1.27e+06 | 1.00e+06 | 9.44e+05 |
| 21 | 9.81e+05 | 1.06e+07 | 3.75e+06 | 4.12e+06 | 1.96e+06 | 1.55e+06 | 1.26e+06 | 9.57e+05 | 9.94e+05 |
| 22 | 9.81e+05 | 1.08e+07 | 3.90e+06 | 4.33e+06 | 1.96e+06 | 1.54e+06 | 1.29e+06 | 1.10e+06 | 9.43e+05 |
| 23 | 9.81e+05 | 1.11e+07 | 4.09e+06 | 4.45e+06 | 1.99e+06 | 1.58e+06 | 1.30e+06 | 1.01e+06 | 9.48e+05 |
| 24 | 9.81e+05 | 1.14e+07 | 4.23e+06 | 4.65e+06 | 2.02e+06 | 1.57e+06 | 1.31e+06 | 1.07e+06 | 9.48e+05 |

Table B.28: Total number of nodes for the learned partitioning strategy: impact of the probing size (i.e., the number of nodes explored in order to compute the features describing the subproblem). We compare probing budgets of 50, 500, $5e^3$, and $5e^4$.

# References


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

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

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

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

Adler, I. and Megiddo, N. (1985). A simplex algorithm whose average number of steps is bounded between two quadratic functions of the smaller dimension. *Journal of the ACM*, 32(4):871–895.

Amenta, N. and Ziegler, G. M. (1999). Deformed products and maximal shadows of polytopes. *Contemporary Mathematics*, 223:57–90.

Applegate, D., Bixby, R., Chvátal, V., and Cook, W. (1995). Finding cuts in the TSP (A preliminary report). Technical Report 05, DIMACS.

Bartlett, P. L. (2003). An introduction to reinforcement learning theory: Value function methods. In *Advanced lectures on machine learning*, pages 184–202. Springer.

Battiti, R., Brunato, M., and Mascia, F. (2008). *Reactive Search and Intelligent Optimization*. Springer Verlag.

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

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

Bertsekas, D. P. and Tsitsiklis, J. N. (1996). *Neuro-Dynamic Programming*. Athena Scientific.

Bertsimas, D. and Tsitsiklis, J. N. (1997). *Introduction to linear optimization*. Athena Scientific.

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

Blanzieri, E. and Bryl, A. (2008). A survey of learning-based techniques of email spam filtering. *Artificial Intelligence Review*, 29(1):63–92.

Borgwardt, K. H. (1982). The average number of pivot steps required by the simplex-method is polynomial. *Zeitschrift für Operations Research*, 26(1):157–177.

Borgwardt, K. H. (1988). Probabilistic analysis of the simplex method. In *Papers of the 16th Annual Meeting of DGOR in Cooperation with NSOR*, pages 564–575.

Boyan, J. and Moore, A. (2000). Learning evaluation functions to improve optimization by local search. *Journal of Machine Learning Research*, pages 77–112.

Breiman, L. (1996). Bagging predictors. *Machine learning*, 24(2):123–140.

Breiman, L. (2001). Random forests. *Machine learning*, 45(1):5–32.

Busoniu, L., Babuska, R., De Schutter, B., and Ernst, D. (2010). *Reinforcement learning and dynamic programming using function approximators*, volume 39. CRC press.

Chvatal, V. (1983). *Linear programming*. Macmillan.

Cong, J., He, L., Koh, C.-K., and Madden, P. H. (1996). Performance optimization of VLSI interconnect layout. *Integration, the VLSI journal*, 21(1):1–94.

Cruz, J. A. and Wishart, D. S. (2006). Applications of machine learning in cancer prediction and prognosis. *Cancer informatics*, 2:59.

Dalal, N. and Triggs, B. (2005). Histograms of oriented gradients for human detection. In *Computer Vision and Pattern Recognition, IEEE Conference on*, volume 1, pages 886–893.

Dantzig, G. B. (1987). Origins of the simplex method. Technical report, Stanford University.

De Loera, J. A. (2011). New insights into the complexity and geometry of linear optimization. *Optima, newsletter of the Mathematical Optimization Society*, 87:1–13.

Di Liberto, G., Kadioglu, S., Leo, K., and Malitsky, Y. (2013). Dash: Dynamic approach for switching heuristics. arXiv.

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

Dorta, I., Leon, C., and Rodriguez, C. (2004). Parallel branch-and-bound skeletons: Message passing and shared memory implementations. In *Parallel Processing and Applied Mathematics*, pages 286–291. Springer.

Driebeek, N. (1966). An algorithm for the solution of mixed integer programming problems. *Management Science*, 12(7):576–587.

Eckstein, J., Phillips, C. A., and Hart, W. E. (2001). PICO: An object-oriented framework for parallel branch-and-bound. *Studies in Computational Mathematics*, 8:219–265.

El-Dessouki, O. I. and Huen, W. H. (1980). Distributed enumeration on between computers. *Computers, IEEE Transactions on*, 29(9):818–825.

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

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

Gendron, B. and Crainic, T. G. (1994). Parallel branch-and-bound algorithms: Survey and synthesis. *Operations Research*, 42(6):1042–1066.

Geurts, P., Ernst, D., and Wehenkel, L. (2006). Extremely randomized trees. *Machine learning*, 63(1):3–42.

Gonzalez, J. L., Dimoulkas, I., and Amelin, M. (2014). Operation planning of a CSP plant in the spanish day-ahead electricity market. In *European Energy Market, IEEE International Conference on the*, pages 1–5.

Graham, R. L., Lawler, E. L., Lenstra, J. K., and Kan, A. R. (1979). Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of discrete mathematics*, 5:287–326.

Gutierrez, D. D. (2015). *Machine Learning and Data Science: An Introduction to Statistical Learning Methods with R.* Technics Publications.

Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Springer.

Hutter, F., Hamadi, Y., Hoos, H., and Leyton-Brown, K. (2006). Performance prediction and automated tuning of randomized and parametric algorithms. In *International Conference on Principles and Practice of Constraint Programming*, pages 213–228.

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

Josz, C., Maeght, J., Panciatici, P., and Gilbert, J. C. (2015). Application of the moment-SOS approach to global optimization of the OPF problem. *Power Systems, IEEE Transactions on*, 30(1):463–470.

Karmarkar, N. (1984). A new polynomial-time algorithm for linear programming. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 302–311.

Karp, R. and Zhang, Y. (1988). A randomized parallel branch-and-bound procedure. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 290–300.

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

Khachiyan, L. G. (1980). Polynomial algorithms in linear programming. *USSR Computational Mathematics and Mathematical Physics*, 20(1):53–72.

Kimura, A., Ishiguro, K., Yamada, M., Marcos Alvarez, A., Kataoka, K., and Murasaki, K. (2013). Image context discovery from socially curated contents. In *Proceedings of the 21st ACM international conference on Multimedia*, pages 565–568.

Klee, V. and Minty, G. J. (1972). How good is the simplex algorithm? In *Inequalities, III (Proc. Third Sympos., Univ. California, Los Angeles, Calif., 1969; dedicated to the memory of Theodore S. Motzkin)*, pages 159–175.

Koch, T., Achterberg, T., Andersen, E., Bastert, O., Berthold, T., Bixby, R., Danna, E., Gamrath, G., Gleixner, A., Heinz, S., et al. (2011). MIPLIB 2010. *Mathematical Programming Computation*, 3(2):103–163.

Kumar, V. and Rao, V. N. (1987). Parallel depth first search. Part II. Analysis. *International Journal of Parallel Programming*, 16(6):501–519.

Lai, T.-H. and Sahni, S. (1984). Anomalies in parallel branch-and-bound algorithms. *Communications of the ACM*, 27(6):594–602.

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

Langley, P. (1985). Learning to search: From weak methods to domain-specific heuristics. *Cognitive Science*, 9(2):217–260.

Laursen, P. S. (1994). Can parallel branch-and-bound without communication be effective? *SIAM Journal on Optimization*, 4(2):288–296.

Leyton-Brown, K., Nudelman, E., and Shoham, Y. (2009). Empirical hardness models: Methodology and a case study on combinatorial auctions. *Journal of the ACM (JACM)*, 56(4):22.

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

Linderoth, J. T. (1998). *Topics in parallel integer optimization*. PhD thesis, Georgia Institute of Technology.

Liwo, A., Lee, J., Ripoll, D. R., Pillardy, J., and Scheraga, H. A. (1999). Protein structure prediction by global optimization of a potential energy function. *Proceedings of the National Academy of Sciences*, 96(10):5482–5485.

Lowe, D. G. (1999). Object recognition from local scale-invariant features. In *Computer vision, IEEE International Conference on*, volume 2, pages 1150–1157.

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

Marcos Alvarez, A., Maes, F., and Wehenkel, L. (2012). Supervised learning to tune simulated annealing for in silico protein structure prediction. In *ESANN 2012 proceedings, 20th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*, pages 49–54.

Marcos Alvarez, A., Wehenkel, L., and Louveaux, Q. (2015). Machine learning to balance the load in parallel branch-and-bound. Technical report, Université de Liège.

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

Marcos Alvarez, A., Yamada, M., and Kimura, A. (2013a). Exploiting socially-generated side information in dimensionality reduction. In *Proceedings of the 2nd international workshop on Socially-Aware Multimedia*, pages 9–12.

Marcos Alvarez, A., Yamada, M., Kimura, A., and Iwata, T. (2013b). Clustering-based anomaly detection in multi-view data. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 1545–1548.

Megiddo, N. (1986). *On the complexity of linear programming.* IBM Thomas J. Watson Research Division.

Moll, R., Barto, A. G., Perkins, T. J., and Sutton, R. S. (1998). Learning instance-independent value functions to enhance local search. In *Advances in Neural Information Processing Systems*, pages 1017–1023.

Munos, R. (2007). Performance bounds in l_p-norm for approximate value iteration. *SIAM journal on control and optimization*, 46(2):541–561.

Munos, R. and Szepesvári, C. (2008). Finite-time bounds for fitted value iteration. *Journal of Machine Learning Research*, 9(May):815–857.

Nemhauser, G. L. and Wolsey, L. A. (1988). *Integer and Combinatorial Optimization.* Wiley-Interscience.

Nesterov, Y. and Nemirovskii, A. (1994). *Interior-Point Polynomial Algorithms in Convex Programming.* Society for Industrial and Applied Mathematics.

Oliva, A. and Torralba, A. (2001). Modeling the shape of the scene: A holistic representation of the spatial envelope. *International Journal of Computer Vision*, 42(3):145–175.

Otten, L. and Dechter, R. (2012). A case study in complexity estimation: Towards parallel branch-and-bound over graphical models. In *Proceedings of the Twenty-Eighth Conference on Uncertainty in Artificial Intelligence*, pages 665–674.

Patel, J. and Chinneck, J. (2007). Active-constraint variable ordering for faster feasibility of mixed integer linear programs. *Mathematical Programming*, 110(3):445–474.

Piérard, S., Marcos Alvarez, A., Lejeune, A., and Van Droogenbroeck, M. (2014). On-the-fly domain adaptation of binary classifiers. In *23rd Belgian-Dutch Conference on Machine Learning (BENELEARN)*, pages 20–28.

Plamondon, R. and Srihari, S. N. (2000). Online and off-line handwriting recognition: a comprehensive survey. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 22(1):63–84.

Pruul, E., Nemhauser, G., and Rushmeier, R. (1988). Branch-and-bound and parallel computation: A historical note. *Operations Research Letters*, 7(2):65–69.

Rao, V. N. and Kumar, V. (1987). Parallel depth first search. Part I. Implementation. *International Journal of Parallel Programming*, 16(6):479–499.

Ricci, F., Rokach, L., and Shapira, B. (2011). *Introduction to recommender systems handbook.* Springer.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3):210–229.

Santos, F. (2012). A counterexample to the Hirsch Conjecture. *Annals of Mathematics*, 176(1):383–412.

Smale, S. (1983). On the average number of steps of the simplex method of linear programming. *Mathematical Programming*, 27(3):241–262.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement learning: An introduction*. MIT press.

Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288.

Valiant, L. G. (1984). A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142.

Vanderbei, R. J. (2014). *Linear programming*. Springer.

Vapnik, V. (2013). *The nature of statistical learning theory*. Springer.

Veloso, M., Carbonell, J., Pérez, A., Borrajo, D., and Blythe, J. (1995). Integrating Planning and Learning: The PRODIGY Architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, pages 81–120.

Wah, B. and Yu, C. F. (1985). Stochastic modeling of branch-and-bound algorithms with best-first search. *Software Engineering, IEEE Transactions on*, 11(9):922–934.

Wolpert, D. and Macready, W. (1997). No free lunch theorems for optimization. *Evolutionary Computation, IEEE Transactions on*, 1(1):67–82.

Xu, Y., Fern, A., and Yoon, S. (2007). Discriminative learning of beam-search heuristics for planning. In *International Joint Conference on Artificial intelligence*, pages 2041–2046.

Yang, M. K. and Das, C. R. (1994). Evaluation of a parallel branch-and-bound algorithm on a class of multiprocessors. *Parallel and Distributed Systems, IEEE Transactions on*, 5(1):74–86.

Ye, Y. (2011). The simplex and policy-iteration methods are strongly polynomial for the markov decision problem with a fixed discount rate. *Mathematics of Operations Research*, 36(4):593–603.

Yu, D. and Deng, L. (2012). *Automatic Speech Recognition*. Springer.

Zhang, W. and Dietterich, T. G. (1995). A reinforcement learning approach to job-shop scheduling. In *International Joint Conference on Artificial Intelligence*, pages 1114–1120.