Towards smarter MILP solvers: A data-driven approach to branch-and-bound

Scavuzzo Montaña, L.V.

DOI
10.4233/uuid:8e239ff5-96ba-451b-a1b1-138eb139c390

Publication date
2024

Document Version
Final published version

Citation (APA)
Scavuzzo Montaña, L. V. (2024). *Towards smarter MILP solvers: A data-driven approach to branch-and-bound*. [Dissertation (TU Delft), Delft University of Technology]. https://doi.org/10.4233/uuid:8e239ff5-96ba-451b-a1b1-138eb139c390

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

TOWARDS SMARTER MILP SOLVERS: A DATA-DRIVEN APPROACH TO BRANCH-AND-BOUND

TOWARDS SMARTER MILP SOLVERS: A DATA-DRIVEN APPROACH TO BRANCH-AND-BOUND

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus Prof. dr. ir. T.H.J.J. van der Hagen,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op donderdag 30, januari 2025 om 12:30 uur

door

Lara Victoria SCAVUZZO MONTAÑA

Master of Science in Applied Mathematics,
Technische Universiteit Delft,
geboren te Buenos Aires, Argentina.

Dit proefschrift is goedgekeurd door de

promotor: Prof. dr. ir. K.I. Aardal

Samenstelling promotiecommissie:

Rector Magnificus,	voorzitter
Prof. dr. ir. K.I. Aardal,	Technische Universiteit Delft, promotor
Dr. N. Yorke-Smith,	Technische Universiteit Delft, copromotor
Onafhankelijke leden:	
Prof. dr. S.I. Birbil	Universiteit van Amsterdam
Dr. C. Hojny	Technische Universiteit Eindhoven
Dr. P. Le Bodic	Monash University
Prof. dr. M.M. de Weerdt,	Technische Universiteit Delft
Overige leden:	
Prof. dr. A. Lodi	Cornell Tech
Prof. dr. D.C. Gijswijt	Technische Universiteit Delft, reservelid

Het onderzoek beschreven in dit proefschrift is mede gefinancierd door de Nederlandse Organisatie voor Wetenschappelijk Onderzoek (NWO), onder project OCENW.GROOT.2019.015.

Keywords: Integer Programming, Branch-and-bound, Machine Learning.

ISBN 000-00-0000-000-0

An electronic version of this dissertation is available at
http://repository.tudelft.nl/.

To all the cats of Delft, whatever
their secret professions may be.

Contents

SUMMARY

The available technology to solve **Mixed Integer Linear Programs** (MILPs) has experienced dramatic improvements in the past two decades. Pushing this algorithmic progress further is essential for solving even more complex optimization problems that arise in practice. This thesis examines various methods to enhance **Branch-and-Bound** (B&B) based MILP solvers, focusing on areas such as branching and Machine Learning (ML) assisted rules. Through our analysis of current methodologies and the introduction of novel techniques, this thesis contributes to the development of more efficient and adaptive MILP solvers. We divide the discussion into four chapters, which are preceded by an introduction to the topics of this thesis (Chapter 1).

Chapter 2 explores the synergy between Machine Learning and B&B-based MILP solvers. In particular, we are interested in the case where these two technologies cooperate, enhancing rather than substituting each other. We survey the literature that falls into this category by first defining a number of abstract **learning tasks** where ML can play a key role. We point out methodological trends both in terms of how to frame and to solve the learning tasks. Further, we highlight some core directions and choices in terms of problem representation, benchmarks and software.

Chapter 3 dives into the problem of **variable selection**, i.e., the problem of choosing a variable that will be used for branching. We start with a discussion of the current challenges faced by some popular methods for branching, both classical and ML-based. We then propose our new formulation, the **tree Markov Decision Process** (tree MDP), which is a generalization of the well-known MDP framework. We show that the variable selection problem can be cast as a tree MDP and that this allows us to more efficiently learn a variable selection strategy without the need for expert demonstrations.

Chapter 4 also addresses branching but from a broader perspective. We study **lattice-based reformulations** of the feasible set that transform the shape and sometimes also the dimension of the problem. The variables in the reformulation are hyperplanes in the original space, hence can be interpreted as a way to generate general branching directions for the original problem. We study the reformulations from different perspectives, trying to uncover why and when the reformulations are effective. Our results show that these techniques have a wide applicability and high potential for speeding-up the solution of difficult Integer Programs.

Chapter 5 takes up again the perspective of ML-enhanced MILP solvers. This time, we ask a simple question: can we predict the **optimal objective value** of an MILP? The answer to this question is of great relevance to MILP solving, with several sub-routines and solver strategies having the potential to benefit from such a prediction. We propose

both a static and a dynamic method which outperform the previously proposed models. These results open the door for more dynamically configurable solvers that automatically adapt their strategy as more information becomes available.

SAMENVATTING

De beschikbare technologie om Mixed Integer Linear Programs (MILPs) op te lossen is de afgelopen twintig jaar enorm verbeterd. Het doorzetten van deze algoritmische vooruitgang is essentieel voor het oplossen van nog complexere optimalisatieproblemen die in de praktijk voorkomen. Deze dissertatie onderzoekt verschillende methoden om Branch-and-Bound (B&B) gebaseerde MILP oplossers te verbeteren, met de nadruk op gebieden zoals branching en regels met ondersteuning van Machine Learning (ML). Door onze analyse van huidige methodologieën en de introductie van nieuwe technieken, draagt dit proefschrift bij aan de ontwikkeling van efficiëntere en adaptievere MILP solvers. We verdelen de discussie in vier hoofdstukken, die worden voorafgegaan door een inleiding op de onderwerpen van dit proefschrift (Hoofdstuk 1).

Hoofdstuk 2 verkent de synergie tussen Machine Learning en MILP solvers op basis van B&B. In het bijzonder zijn we geïnteresseerd in het geval waarin deze twee technologieën samenwerken en elkaar versterken in plaats van vervangen. We geven een overzicht van de literatuur die in deze categorie valt door eerst een aantal abstracte leertaken te definiëren waarbij ML een sleutelrol kan spelen. We wijzen op methodologische trends, zowel wat betreft het kader als het oplossen van de leertaken. Verder belichten we enkele belangrijke richtingen en keuzes in termen van probleemrepresentatie, benchmarks en software.

Hoofdstuk 3 duikt in het probleem van variabelenselectie, d.w.z. het probleem van het kiezen van een variabele die gebruikt zal worden voor branching. We beginnen met een bespreking van de huidige uitdagingen van enkele populaire methoden voor branching, zowel klassiek als ML-gebaseerd. Vervolgens stellen we onze nieuwe formulering voor, het tree Markov Decision Process (tree MDP), dat een veralgemening is van het bekende MDP raamwerk. We laten zien dat het variabelenselectieprobleem kan worden geformuleerd als een tree MDP en dat dit ons in staat stelt om efficiënter een variabelenselectiestrategie te leren zonder de noodzaak van expertdemonstraties.

Hoofdstuk 4 gaat ook in op branching, maar vanuit een breder perspectief. We bestuderen op roosters gebaseerde herformuleringen van de haalbare verzameling die de vorm en soms ook de dimensie van het probleem transformeren. De variabelen in de herformulering zijn hypervlakken in de oorspronkelijke ruimte, en kunnen dus geïnterpreteerd worden als een manier om algemene branchingrichtingen voor het oorspronkelijke probleem te genereren. We bestuderen de herformuleringen vanuit verschillende perspectieven en proberen te ontdekken waarom en wanneer de herformuleringen effectief zijn. Onze resultaten tonen aan dat deze technieken breed toepasbaar zijn en een groot potentieel hebben om de oplossing van moeilijke integer programma's te versnellen.

Hoofdstuk 5 neemt het perspectief van ML-ondersteunde MILP solvers weer op. Deze keer stellen we een eenvoudige vraag: kunnen we de optimale doelfunctiewaarde van een MILP voorspellen? Het antwoord op deze vraag is van groot belang voor het oplossen van MILP's, waarbij verschillende subroutines en oplossingsstrategieën het potentieel hebben om te profiteren van een dergelijke voorspelling. We stellen zowel een statische als een dynamische methode voor die beter presteren dan de eerder voorgestelde modellen. Deze resultaten openen de deur voor meer dynamisch configureerbare solvers die hun strategie automatisch aanpassen als er meer informatie beschikbaar komt.

NOTATION

Vectors are denoted with lowercase in bold: $\boldsymbol{x}$.

Matrices are denoted with uppercase in bold: $\boldsymbol{A}$.

Elements of matrices and vectors are denoted with a subscript: A_{ij}, x_i.

Euclidean norm: $||\boldsymbol{x}|| := \sqrt{\sum_{i=1}^{n} x_i^2}$.

Vector multiplication: for two vectors $\boldsymbol{x}$ and $\boldsymbol{y} \in \mathbb{R}^n$ we use both the notation $\boldsymbol{x}^\mathsf{T}\boldsymbol{y}$ and $\langle \boldsymbol{x}, \boldsymbol{y} \rangle$ to denote the vector multiplication $\sum_{i=1}^{n} x_i y_i$.

An indexed set of vectors: $\{\boldsymbol{x}^1, \ldots, \boldsymbol{x}^K\} = \{\boldsymbol{x}^i\}_{i=1}^K$.

Columns of a matrix: $\boldsymbol{A} = [\boldsymbol{a}^i]_{i=1}^K$

Kernel of a matrix $\boldsymbol{A}$: ker $\boldsymbol{A}$

Rank of a matrix $\boldsymbol{A}$: rk $\boldsymbol{A}$

Standard basis vectors: $\boldsymbol{e}^i$ is a vector whose coordinates are all zero, except the i-th coordinate which takes value one.

Span of vectors: $\text{span}\{\boldsymbol{x}^1, \boldsymbol{x}^2, \ldots, \boldsymbol{x}^K\} = \{\boldsymbol{y} \in \mathbb{R}^n \mid \boldsymbol{y} = \sum_{i=1}^{K} \alpha_i \boldsymbol{x}^i, \alpha_i \in \mathbb{R}\}$

Orthogonal complement of a subspace $W \subseteq \mathbb{R}^n$: $W^\perp = \{\boldsymbol{x} \mid \boldsymbol{x}^\mathsf{T}\boldsymbol{y} = 0 \text{ for all } \boldsymbol{y} \in W\}$

Independence of two random variables X and Y: $X \perp\!\!\!\perp Y$

$[n] := \{1, ..., n\}$ is the set of integer numbers from 1 to n.

$\propto$ means “proportional to".

1

INTRODUCTION

1.1. GOALS OF THIS THESIS

Mixed Integer Linear Programs (MILPs) offer a powerful modeling tool for optimization problems with a combinatorial nature, and are therefore extensively used in many real-world applications [123]. Since the advent of the cutting-plane algorithm [69] and the branch-and-bound algorithm [97, 45], we have witnessed enormous advances in our capabilities to solve MILPs to optimality. Undoubtedly, improvements in the available hardware have played a key role, but algorithmic advancements have been equally pivotal. Most commercial solvers use the branch-and-bound algorithm with cuts (also known as branch-and-cut). This means that the backbone of MILP solving is the same as it was back in the 60s, when the field was just starting. Decades of research have built upon that, with remarkable advances in, e.g., cutting plane generation, domain reduction techniques and, notably, smart decision-making rules. An implementation of the branch-and-bound (or branch-and-cut) algorithm requires, in fact, making a great number of choices. From the node processing order, to the branching strategy, these choices have a critical impact in the performance of the algorithm [7]. This thesis revisits different aspects of decision-making within solvers. In particular, the contributions of this thesis can be summarized in the following four perspectives.

- **Identification of decision-making tasks for machine learning.** We revisit different components of the solver, evaluating the potential of machine learning tools for aiding the decision-making process. We present the advances in this intersection, providing a methodological overview as well as our own contributions.
- **Advances in branching rules.** We present our contribution to branching rules. We study the problem of learning to branch without expert demonstrations, as well as different methodologies for branching on general disjunctions.
- **Reflection on solver metrics.** The process of solving MILPs generates plenty of data. There is a need to better understand how to make effective use of this in-

formation. In this line, we study different MILP representation methods that curate this data in a way that a machine learning model can use. We further study a number of solver metrics that can be used to make predictions about the optimal objective value.

- **Technological contributions.** The aim of this thesis is to also provide accessible code and resources that contribute to the growing community that creates the tools for development, testing and deployment of MILP solvers.

1.2. STRUCTURE OF THIS THESIS

We start with an introduction to MILP and MILP solvers (Section 1.3 of this chapter), which also introduces important notation that will be used throughout the thesis. Also in this chapter, Section 1.4 provides a short introduction to concepts regarding machine learning methods that will be of relevance to the thesis. Chapter 2 dives into different decision-making tasks within MILP solving, and how they can be addressed from a machine learning perspective. Chapter 3 presents our methodology to learning branching rules without demonstrations. Chapter 4 also addresses the problem of branching, but with a wider viewpoint: we study different methodologies that allow us to generate branching directions that are not necessarily single variables. Finally, Chapter 5 presents our approach to predicting optimal objective values, with the aim of contributing towards smarter, dynamically configurable MILP solvers.

1.3. INTRODUCTION TO MILP

1.3.1. MIXED INTEGER LINEAR PROGRAMMING

Let us start by formally defining the MILP formulation. We are given a matrix $\boldsymbol{A} \in \mathbb{Q}^{m \times n}$, vectors $\boldsymbol{c} \in \mathbb{Q}^n$ and $\boldsymbol{b} \in \mathbb{Q}^m$, and a partition $(\mathscr{A}, \mathscr{B}, \mathscr{C})$ of the variable index set $[n]$. A Mixed Integer Linear Program is the problem of finding

$$\begin{aligned} z^* = \min \ & \boldsymbol{c}^\top \boldsymbol{x} \\ \text{subject to } & A\boldsymbol{x} \geq \boldsymbol{b}, \\ & x_j \in \mathbb{Z}_{\geq 0} \quad \forall j \in \mathscr{A}, \\ & x_j \in \{0,1\} \quad \forall j \in \mathscr{B}, \\ & x_j \geq 0 \quad \forall j \in \mathscr{C}. \end{aligned} \tag{1.1}$$

We call z^* the *optimal value*, and we say $\boldsymbol{x}^*$ is an *optimal solution* if $\boldsymbol{c}^\top \boldsymbol{x}^* = z^*$. We write $\mathscr{I} := \mathscr{A} \cup \mathscr{B}$ to denote the set of integer and binary variables. When $\mathscr{C} = \emptyset$, and with a slight abuse of notation, we speak of Integer Programming (IP). An MILP is said to be mixed-binary if $\mathscr{A} = \emptyset$ and binary if $\mathscr{A} = \mathscr{C} = \emptyset$. Relaxing the integrality constraints for variables in $\mathscr{I}$ yields a Linear Program (LP), known as the LP relaxation of the MILP.

The *feasible set* of (1.1) is defined as

$$X := \left\{ \boldsymbol{x} \in \mathbb{R}^n_{\geq 0} \mid \boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b}, \quad x_j \in \mathbb{Z}_{\geq 0} \ \forall j \in \mathscr{A}, \quad x_j \in \{0,1\} \ \forall j \in \mathscr{B} \right\}. \tag{1.2}$$

Table 1.1: Summary of the B&B notation.

z^*	optimal value.
$\boldsymbol{x}^*$	optimal solution.
z^{LP}	LP value (global).
$\boldsymbol{x}^{LP}$	LP solution (global).
z^{LP_i}	LP value of node i.
$\boldsymbol{x}^{LP_i}$	LP solution of node i.
$\bar{z}$	primal bound.
$\bar{\boldsymbol{x}}$	incumbent solution.
$\underline{z}$	dual bound.

The integer variables are key to the wide applicability of the MILP model, since they allow practitioners to model indivisible entities or yes/no decisions. At the same time, it is because of the integrality constraints that the decision problem related to (1.1) is in $\mathcal{NP}$ [40]. Indeed, if we allow all variables to take continuous variables, the problem becomes a Linear Program (LP) for which we have algorithms that are efficient in theory and in practice [146, 153, 75]. This does not mean that MILPs cannot be solved. In fact, MILPs of practical importance are routinely solved, and research in MILP solution methods has fostered remarkable advances [7]. At the backbone of these solvers is the branch-and-bound algorithm, which is the topic of the next section.

1.3.2. THE BRANCH-AND-BOUND ALGORITHM

The Branch-and-Bound (B&B) algorithm was first developed in the 1960s as a general purpose algorithm for tackling discrete optimization problems [97, 45]. The core idea behind it is the successive partition of the feasible set into smaller problems. This is combined with the use of relaxations, which provide bounds that can be used to discard unpromising sub-problems, thus avoiding complete enumeration. The key components of the algorithm are explained in the following, and the algorithm as a whole is summarized with pseudocode in Alg. 1. We note that this constitutes a textbook version of the B&B algorithm, which we call *vanilla B&B*. In practice, the software dedicated to solving MILPs uses a plethora of specialized rules and additional components (which will be mentioned in Section 1.3.3). A summary of the key terminology and notation introduced in the current section can be found in Table 1.1.

The LP relaxation

Let $P := \{\boldsymbol{x} \in \mathbb{R}^n_{\geq 0} \mid \boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b}\}$. This is, we obtain P from the feasible set X by dropping the integrality requirements. Since $X \subset P$, we have that the problem

$$\begin{aligned} &\text{minimize } \boldsymbol{c}^\top \boldsymbol{x} \\ &\text{subject to } \boldsymbol{x} \in P \end{aligned} \tag{1.3}$$

is a relaxation of (1.1). Problem (1.3) receives the name of *LP relaxation*. Solving (1.3), which as mentioned above can be done efficiently, provides a lower bound to (1.1). In other words, letting $z^{LP} := \min\{\boldsymbol{c}^\top \boldsymbol{x} \mid \boldsymbol{x} \in P\}$, we have that $z^{LP} \leq z^*$.

Branch...

Solving the LP relaxation of an MILP might yield a solution $\boldsymbol{x}^{LP}$ that satisfies the integrality constraints. However, this is not always the case. In general, there will be a variable $j \in \mathscr{I}$ such that $x_j^{LP} \notin \mathbb{Z}$. Notice that for any $\boldsymbol{x} \in X$ and $j \in \mathscr{I}$ it is true that

$$x_j \leq \lfloor x_j^{LP} \rfloor \qquad \text{or} \qquad x_j \geq \lceil x_j^{LP} \rceil . \tag{1.4}$$

These inequalities allow us to partition the feasible set into two subproblems (or branches), whose LP-relaxations do not contain $\boldsymbol{x}^{LP}$

$$X_1 = X \cap \{\boldsymbol{x} \,|\, x_j \leq \lfloor x_j^{LP} \rfloor\} \qquad \text{and} \qquad X_2 = X \cap \{\boldsymbol{x} \,|\, x_j \geq \lceil x_j^{LP} \rceil\} .$$

This partitioning step is known as branching. It is also possible to split the problem using other types of inequalities, which will be discussed in Chapter 4. The B&B algorithm recursively applies branching, thereby building a search tree. Each node in the tree has an associated index i and an associated local sub-MILP, with a local LP solution $\boldsymbol{x}^{LP_i}$ and associated LP value z^{LP_i} which acts as a local lower bound.

... and bound

The bounds z^{LP_i} obtained at each node of the tree play an important role in the B&B algorithm. Most importantly, whenever a feasible solution $\boldsymbol{x} \in X$ is found, all nodes such that $z^{LP_i} \geq \boldsymbol{c}^\mathsf{T} \boldsymbol{x}$ can be *pruned*, i.e., discarded. Indeed, if this is the case, any feasible solution $\boldsymbol{x}'$ contained in that node will surely satisfy $\boldsymbol{c}^\mathsf{T} \boldsymbol{x}' \geq z^{LP_i} \geq \boldsymbol{c}^\mathsf{T} \boldsymbol{x}$, and therefore cannot be better than $\boldsymbol{x}$. This procedure avoids complete enumeration of the feasible region. In practice, these bounds are used for a number of other purposes, such as node selection.

The termination criterion

At any point during the search, the value of an integer feasible solution provides an upper bound on z^*. The best, i.e., smallest, known upper bound value is called *primal bound*, and we denote it by $\bar{z}$. The corresponding best known feasible solution $\bar{\boldsymbol{x}}$ is called *incumbent solution*. Similarly, the bounds z^{LP_i} of unprocessed nodes can be used to obtain a lower bound. In particular, $\underline{z} := \min_{i:i \text{ unprocessed}} \{z^{LP_i}\}$ provides a lower bound on z^* and is called *dual bound*. The B&B algorithm ends when all nodes have been processed, when $\bar{z} = \underline{z}$, or when another termination condition (e.g., timeout) applies.

1.3.3. MILP SOLVING

In the previous section, we described the basic principles of the B&B algorithm as implemented in commercial MILP solvers, such as CPLEX [84], Gurobi [75] or Xpress [57], as well as in academic solvers like SCIP [28]. In practice, the execution of the B&B algorithm revolves around some key solver components that handle the different aspects of the solving process. The most important components are preprocessing, the branching rule, the cut management and the primal heuristics.

Algorithm 1 A (vanilla) Branch and Bound algorithm

Input: The root node N_0 associated with the original MILP.

1: Initialization: $\mathscr{L} = \{N_0\}$, $\bar{z} = +\infty$, $\bar{\boldsymbol{x}} = \emptyset$
2: **if** $\mathscr{L} = \emptyset$ **then return** $\bar{z}$ and $\bar{\boldsymbol{x}}$
3: **end if**
4: Choose $N_k \in \mathscr{L}$
5: $\mathscr{L} \leftarrow \mathscr{L} \setminus \{N_k\}$
6: Solve the LP relaxation of N_k
7: **if** infeasible **then goto** 2
8: **end if**
9: Let $\boldsymbol{x}^{LP_k}$ be the LP solution and z^{LP_k} the LP value
10: **if** $z^{LP_k} \geq \bar{z}$ **then goto** 2
11: **end if**
12: # here we must have $z^{LP_k} < \bar{z}$
13: **if** $\boldsymbol{x}^{LP_k}$ is integer feasible **then**
14: $\quad \bar{z} \leftarrow z^{LP_k}$
15: $\quad \bar{\boldsymbol{x}} \leftarrow \boldsymbol{x}^{LP_k}$
16: **else** branch:
17: $\quad$ Choose $i \in I$ such that $x_i^{LP_k} \notin \mathbb{Z}$.
18: $\quad$ Split N_k into N^+ and N^- using the inequalities in Eq 1.4 on variable i
19: $\quad \mathscr{L} \leftarrow \mathscr{L} \cup \{N^+, N^-\}$
20: **end if**
21: **goto** 2

Notice that we use the term 'MILP solver' rather than the, sometimes more common, 'MIP solver', being that we focus especially on the learning developed for solving mixed integer *linear* programs, not including the nonlinear case. Nevertheless, most of the technology is used in the solvers for both linear and nonlinear programs. Notably, ML has also been applied to the nonlinear case (see, e.g., [30]).

Preprocessing. Most solvers implement a number of procedures that try to reduce the size of the problem and its difficulty, for example by identifying substructures or ways to strengthen the LP relaxation. These routines have been shown to be of crucial importance in speeding up the solution process. For an overview of these methods, together with a computational analysis of their effectiveness see [10].

Branching and node selection. The procedure of dividing the feasible region is called *branching*. There is a choice to be made with respect to which disjunction is used for branching. The standard[1] is to use single variable disjunctions of the type

$$x_j \leq \lfloor x_j^{LP} \rfloor \vee x_j \geq \lceil x_j^{LP} \rceil,$$

[1] More generally, it is also possible to split the problem using multi-variable disjunctions (see, e.g., Karamanov and Cornuéjols [85]).

for some $j \in \mathcal{I}$ such that $x_j^{LP} \notin \mathbb{Z}$. Still, the solution to the LP relaxation is likely to violate more than one integrality constraint, which means there is more than one candidate variable for branching. The so-called *branching rule* is the strategy that the solver uses to select a variable for branching. Computational studies have shown that this choice has a critical impact on solver performance [7]. Another important decision, though less critical in terms of solver performance, is *node selection*, where the question is which unprocessed B&B node to consider next.

Cutting planes. The LP relaxation can be strengthened by adding valid linear inequalities. These are inequalities that cut off parts of the relaxation but do not exclude any integer feasible solutions. This can, in principle, be done in any node of the B&B tree representing a non-empty feasible region, yet it is standard practice to use cutting planes more heavily, or even exclusively, at the root node. A B&B routine where cutting planes are added in other nodes than the root node is typically referred to as branch-and-cut [122]. There is a vast amount of knowledge on cutting planes (or cuts, for short) for MILP (see, e.g., [118, 152, 40]) and most solvers implement a plethora of efficient separation[2] algorithms. The wide availability of cuts can create an issue: while the addition of cuts strengthens the local relaxation, a large amount of cuts can slow down LP solving and lead to numerical instability [46]. For this reason, a judicious *cut management* strategy, which includes separation, selection and removal, is of utmost importance.

Primal heuristics. We use the term primal heuristics to refer to routines that try to find feasible solutions in a short amount of time without a success guarantee of doing so. Relying solely on integer feasible LP relaxations to find solutions is most often inefficient. Primal heuristics can provide good solutions early on and help bring down the primal bound $\bar{z}$ more effectively. As with cutting planes, primal heuristics can be used in any node of the tree if desired.

1.3.4. EVALUATION METRICS FOR MILP

In this section, we define a number of metrics that quantify the progress of a branch-and-bound run. We extend the notation just presented in Section 1.3.2 with a variable $t \geq 0$ that represents solving time. We can then define the primal bound $\bar{z}(t) : [0, T_{\max}] \mapsto \mathbb{R}$ and the dual bound $\underline{z}(t) : [0, T_{\max}] \mapsto \mathbb{R}$ as functions of time. We define $\bar{z}(t)$ to be infinity if no integer feasible solution has yet been found at time t. We also make use of a small tolerance value ϵ that is introduced for numerical stability, typically $\epsilon = 10^{-6}$. Notice that many of these metrics make use of the optimal solution value z^* and must therefore be calculated a posteriori, once the instance is solved.

OPTIMALITY GAP

We define the *optimality gap* as

$$g(t) := \begin{cases} 1 & \text{if no solution has been found yet or } \bar{z}(t) \cdot \underline{z}(t) < 0, \\ \frac{|\bar{z}(t) - \underline{z}(t)|}{\max\{|\bar{z}(t)|, |\underline{z}(t)|, \epsilon\}} & \text{otherwise.} \end{cases} \tag{1.5}$$

[2] The term *separation* refers to the fact that a cutting plane has an effect when it separates a fractional solution of an LP relaxation from the convex hull of (mixed) integer feasible solutions.

Alternatively, one can track the *integrality gap* defined as $g'(t) = |z^* - \underline{z}(t)|$.

PRIMAL GAP AND INTEGRAL

For a given feasible solution $\boldsymbol{x}$, we define the *primal gap* $\gamma(\boldsymbol{x})$ as

$$\gamma(\boldsymbol{x}) := \begin{cases} 1 & \text{if } z^* \cdot \boldsymbol{c}^\top \boldsymbol{x} < 0, \\ \frac{|z^* - \boldsymbol{c}^\top \boldsymbol{x}|}{\max\{|z^*|, |\boldsymbol{c}^\top \boldsymbol{x}|, \epsilon\}} & \text{otherwise.} \end{cases} \tag{1.6}$$

We can define a *primal gap function* that maps the solving time to the primal gap of the best solution found up until that point. In particular, denoting by $\boldsymbol{x}(t)$ the best solution found at time t, we define

$$p(t) := \begin{cases} 1 & \text{if no solution has been found at time } t, \\ \gamma(\boldsymbol{x}(t)) & \text{otherwise.} \end{cases} \tag{1.7}$$

The *primal integral* [22] of a process with time limit $T_{max} > 0$ is defined as

$$P(T_{max}) := \int_0^{T_{max}} p(t)\,dt. \tag{1.8}$$

An extension of the primal integral is the *confined primal integral* [23] which integrates over an exponentially dampened primal gap function $p_\alpha(t) = p(t)e^{t/\alpha}$ for some $\alpha < 0$. This puts emphasis on the early part of the solution process and avoids a high dependence on the chosen time limit.

PRIMAL-DUAL INTEGRAL

One can extend the concept of primal integral to also account for improvements in the dual bound. For this purpose, we integrate over the optimality gap instead of the primal gap, to obtain

$$PD(T_{max}) := \int_0^{T_{max}} g(t)\,dt. \tag{1.9}$$

1.4. A BRIEF INTRODUCTION TO MACHINE LEARNING

This section provides a brief introduction to the key concepts of machine learning that are necessary in order to follow this survey. For a more detailed introduction to machine learning we refer to Mitchell [115].

We are interested in the problem of constructing a mapping from some input data to a desired output space. Let $\mathcal{X}$ be the input data space and let $\mathcal{Y}$ be the output space. It is common to refer to $\mathcal{X}$ as the *feature* space, as it basically represents a set of descriptors of the data samples. The output can be, for example, a prediction based on the input data. The mapping will be constructed by choosing among a family of parameterized functions $f(\cdot, \boldsymbol{\theta}) : \mathcal{X} \to \mathcal{Y}$ with parameters $\boldsymbol{\theta} \in \Theta$. In short, the objective is to optimize the behavior of the mapping $f(\cdot, \boldsymbol{\theta})$ by carefully tuning the parameters, based on sampled observations and a progress metric of choice.

To formalize this, we follow standard practices in machine learning and distinguish the following two settings.

Supervised learning. The learner has access to a finite collection of pairs $\{(X_i, Y_i)\}_{i=1}^N$, where $X_i \in \mathscr{X}$ and $Y_i \in \mathscr{Y}$. Here, Y_i is the *desired output* to input X_i. It is common to refer to Y_i as the *ground truth* or *label*. The goal is to minimize the *loss function* $l : \mathscr{Y} \times \mathscr{Y} \mapsto \mathbb{R}$, a metric that represents the dissimilarity between the prediction and the desired output. An example is the mean square error loss (MSE) $\frac{1}{N}\sum_{i=1}^N \left(f(X_i, \boldsymbol{\theta}) - Y_i\right)^2$. Since the true sample distribution is unknown, the loss is minimized with respect to the observed samples

$$\underset{\theta \in \Theta}{\text{minimize}} \sum_{i=1}^{N} l(Y_i, f(X_i, \theta)) .$$

Reinforcement learning. Reinforcement learning (RL) is defined in the context of sequential decision making, where actions have long-term consequences and the optimal action is either unknown or too expensive to compute. The learning agent has no access to the ground truth. This methodology is formalized under the framework of Markov Decision Processes (MDPs), see Figure 1.1 for a diagram summarizing MDPs. In an MDP, an *agent* interacts with an *environment*. The environment has an associated *state* representing its internal configuration. We denote the state space $\mathscr{S}$. The agent acts on the environment by choosing an action from the action space $\mathscr{A}$ using a *policy* $\pi : \mathscr{S} \mapsto \mathscr{A}$. The agent's action changes the state of the environment, which corresponds to the transition to a new state. The transitioning mechanism is unknown to the agent.

Apart from the environment's state, the agent can observe a *reward function*. We consider the episodic case, where this interaction between agent and environment happens sequentially in discrete time-steps until a *terminal state* is reached and the interaction ends. A realization of such agent-environment interaction is called trajectory

$$\tau := (S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T) ,$$

where T denotes the episode length. The goal of the agent is to find a policy that maximizes the expected cumulative reward, known as the *value function* and formally defined as

$$V_\pi := \mathbb{E}_\tau \left[\sum_{t=1}^{T} \gamma^{t-1} R_t \right] , \tag{1.10}$$

where $\gamma \in [0, 1]$ is the *discount factor*. This parameter controls the greediness of the policy: for γ close to zero, the agent will prioritize obtaining immediate rewards, whereas for γ close to one the agent is encouraged to follow a strategy that pays off in the long term. There is a variety of RL methodologies (see, e.g., [137]) that provide ways to train a parametrized function $\pi(s, \theta)$ with the goal of maximizing V_π. Notice that the trajectory distribution depends both on the agent's policy and the unknown transition mechanism of the environment. For this reason, RL algorithms use trajectory sampling as a way to estimate this expectation. Note that $-V_\pi$ can be seen as a loss function.

The MDP formulation can also be used in a supervised learning setting. In this case, the reward signal is substituted by an expert that tells the agent what is the optimal action. The expert is typically expensive to query, and for this reason we want the agent to learn a policy that imitates the expert, but at a lower cost. This methodology is known as *imitation learning*.

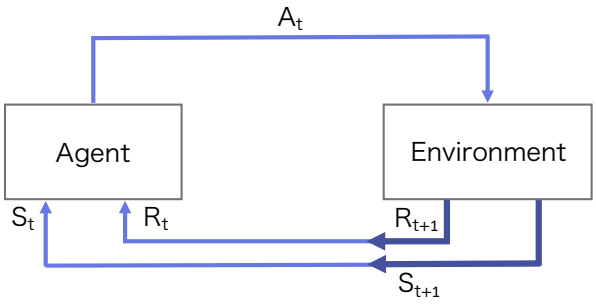


Figure 1.1: Markov Decision Process

1.4.1. MAPPING FEATURES TO PREDICTIONS

A fundamental step in the design of an ML scheme is the choice of function space parameterized by $\boldsymbol{\theta}$. This is commonly known as the *architecture*, and it is independent of the choice of learning methodology. Modern machine learning favors neural networks as the computational representation and mechanism of f. A simple form of neural network is the feed-forward neural network that consists of a series of linear transformations, each followed by a non-linear transformation called an *activation function*.

Definition 1 (Feed-forward Neural Network)**.** *Let $X \in \mathbb{R}^d$ be the input data and let d^L be the desired output dimension. An L-layer* feed-forward neural network *is a function $f : \mathbb{R}^d \mapsto \mathbb{R}^{d^L}$ that defines a mapping from X to $a^L \in \mathbb{R}^{d^L}$ through the recursive relation*

$$z^l = W^l a^{l-1} + b^l$$

$$a^l = \sigma(z^l)$$

for $l = 1, ..., L$, where

- *$a^0 = X$, $d^0 = d$,*
- *$W^l \in \mathbb{R}^{d^l \times d^{l-1}}$ and $b^l \in \mathbb{R}^{d^l}$ are learnable parameters, and*
- *σ is the activation function.*

Some common activation functions are the Rectified Linear Unit (ReLU) or the sigmoid function, defined, respectively, as

$$\sigma_{\text{ReLu}}(x) = \max(0, x)\,,$$

$$\sigma_s(x) = \frac{1}{1 + e^{-x}}\,.$$

Notice that these functions are applied componentwise.

Another prominent architecture is the graph neural network (GNN) that will be of particular interest to the discussion in Section 2.2. Consider an undirected graph $G = (V, E)$ with vertex set V and edge set E. For a node $v \in V$, we denote $N(v) \subseteq V$ the set of neighbors of v.

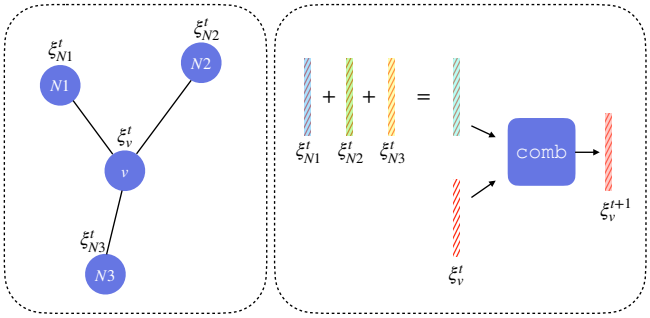


Figure 1.2: Embedding computation with a Graph Neural Network. Here, we use the abbreviation ξ^t_v for $\xi^t(G,v)$. To update the embedding ξ^t_v of node v at time t, the embeddings of neighboring nodes ($N1$, $N2$ and $N3$) are added and then combined with the current embedding using the `comb` function. The result is ξ^{t+1}_v.

Definition 2 (Graph embedding)**.** *A (d-dimensional)* graph embedding ξ *is a function that takes in a graph* $G=(V,E)$ *and a node* $v \in V$ *and returns an element* $\xi(G,v) \in \mathbb{R}^d$.

Definition 3 (Graph Neural Network)**.** *A* Graph Neural Network *is a function that takes as input a graph* $G=(V,E)$ *and an initial embedding* ξ^0 *and defines a recursive embedding* ξ^t *over the vertices of* G*. This function is characterized by*

- *A combination function* `comb`: $\mathbb{R}^{2d} \to \mathbb{R}^d$, *and*
- *An update rule* $\xi^{t+1}(G,v) = \texttt{comb}\left(\xi^t(G,v), \sum_{u \in N(v)} \xi^t(G,u)\right)$.

Figure 1.2 depicts one iteration of this recursive embedding mechanism. It is common to refer to a single iteration as a *message-passing operation*. A possible combination function is a feed-forward neural network over the two concatenated vectors. We can extend graph embeddings to edges. This is, for $e \in E$, we can define $\xi(G,e) \in \mathbb{R}^d$. In the presence of edge embeddings, we can redefine a GNN update rule as

$$\xi^{t+1}(G,v) = \texttt{comb}\left(\xi^t(G,v), \sum_{u \in N(v)} \texttt{aggr}\left(\xi^t(G,u), \xi(G,\{u,v\})\right)\right) \tag{1.11}$$

with an aggregation function `aggr`: $\mathbb{R}^{2d} \to \mathbb{R}^d$ that handles the combination of node and edge embeddings. An example of aggregation function is a simple component-wise sum.

1.4.2. ELEMENTS OF THE LEARNING PROCESS

The learning process is itself one of (inexact) optimization, termed *training*.

Optimization algorithms. The goal of the training process is to optimize the behavior of the mapping $f(\cdot,\boldsymbol{\theta})$ over the parameter space Θ with respect to the loss function. This is done with algorithms for continuous non-convex optimization, which take gradient descent as their base. Some examples of commonly-used algorithms are AdaGrad [52] or Adam [92]. For reinforcement learning, the situation is more complex, in the sense

that several learning paradigms are available for approximating the value function and learning the optimal policy.[3]

Hyper-parameters. There is a number of additional parameters that influence the learning process that are not part of the parameters $\boldsymbol{\theta}$ to be learned. These meta-level parameters are called *hyper-parameters*. Examples of hyper-parameters are the number of iterations of the optimization algorithm, the learning rate, the size of the learned function, etc. Tools exist to automatically tune hyper-parameters given a validation dataset (see below).

Train, validation and test datasets. The data used to tune the parameters of mapping f is called the *training data*. The data used to estimate the performance of f in an unbiased way, and to tune the hyper-parameters, is called the *validation data*. Finally, the data used to evaluate the final trained and optimised mapping is called the *test set*. These sets must be disjoint.

Data collection. The learner makes use of the available training data to improve its behavior. This data can be gathered all at once during an initial data collection phase, before training starts. Alternatively, several data collection and training steps can be run in an alternating fashion.

Overfitting. We say a learned model suffers from *overfitting* when the performance on the training set is distinctively better than the performance on the test set. In other words, the model is unable to generalise to data unseen during the training phase. This phenomenon occurs when the chosen model is too large (in terms of number of parameters) for the task, or the amount of relevant data is too limited for the size of the model being learned. See Goodfellow et al. [71] for a discussion of methods to avoid overfitting.

Online versus offline learning. Whenever we can distinguish separate training and execution phases we speak of *offline learning*. That is, in offline learning, we perform the data collection and learning as a separate, preliminary, once and for all process. After that, the learned function is fixed and used without further tuning. This training phase can be computationally costly but this cost is not reflected in the execution time (often called inference time) of the algorithm, which is typically the metric of interest to the end user. In fact, this can be seen as doing the heavy work upfront while alleviating the effort at the moment of execution.

In contrast, in an *online learning* setting, data becomes available during execution. Learning must therefore happen dynamically as each data sample becomes available, and in parallel to the main process. This has the great advantage of generating a function that is highly adaptive. On the other hand, it entails adding the cost of learning to the cost of execution and further there is usually less available data.

[3] A general discussion of those paradigms is outside the scope of this thesis and special cases will be discussed while surveying specific results from the literature, see Section 2.1.

2

MACHINE LEARNING ASSISTED B&B

The B&B algorithm is the skeleton upon which MILP software builds its solving capability, which has seen remarkable progress in the last two decades [7]. Continuing these waves of algorithmic progress is key to unlocking new application domains and larger scales, as many areas of potential improvement still exist. For example, several algorithmic decisions that must be made within B&B are made by heuristic rules that have been developed and tuned via computational studies to yield good average performance. For some of these decisions we know that more successful heuristics exist, which are, however, too computationally expensive. In the past decade, Machine Learning (ML) methodologies have been explored as a potential tool to mimic such time-consuming heuristics for improved performance. This strategy is particularly promising given the increase in data availability, not only in terms of problem instances but also data collected by the solver during the solution process.

This chapter addresses precisely the perspective of enhancing key components of B&B by ML so as to integrate machine learning and mathematical optimization as complementary technologies, not as competing ones. We note that collecting data about the solving process and exploiting it to make informed decisions within the algorithm is standard practice in MILP solving. We survey methods that take this idea one step further, in the sense that the mapping between the collected data and the decision is not fixed a priori by an expert, but instead automatically constructed by 'learning algorithms' that try to optimize a predefined metric of efficiency.

The survey by Bengio et al. [20] presents the first developments of using machine learning in the context of optimization, with a focus on combinatorial optimization. In this chapter, we further zoom in on the B&B framework in an MILP solver for solving a (class of) MILP. The methods we discuss do not assume any particular problem structure a priori, in the sense that the structure is instead learned. Our focus is on using ML for

elements of B&B where choices are being made, such as how to navigate the search tree, how to quickly find good feasible solutions, and how to improve linear relaxations. We do not cover solution methods that replace B&B, such as end-to-end ML approaches. By restricting the attention to the MILP context and its integration with ML, we are able to make a significant step forward in the characterization of choices like the problem representation and of specific aspects like benchmarks and software.

The chapter is organized as follows. Section 2.1 proceeds to survey the tasks in MILP solving where ML can be useful. Section 2.2 concerns representations of MILP, and Section 2.3 rounds out the survey by collating instance benchmarks and software used in the literature. Some final remarks and perspectives are given in Section 2.4. Note that the contents of this chapter are based on our published work [136].

2.1. LEARNING TASKS

This section presents different ways to formulate a learning task within the B&B algorithm. We split this discussion by considering each major solver component separately.

Before getting into the details, two considerations are required. First, throughout the section we will use the symbol X to abstractly denote a representation of an MILP instance. This representation may include data coming from the problem description (such as in Equation 1.1) and from the B&B process. Section 2.2 discusses, in more detail, approaches to build such a representation. Second, it is important to highlight that the learning tasks that we describe next might be associated with different levels of required generalization. More precisely, the ML models are often trained on data belonging to MILP instances in the same class, for example, set covering, knapsack, etc. In other words, the characteristics of the constraint matrix in (1.1) are leveraged to obtain accurate predictions, so the generalization power of the resulting models is generally restricted to that specific MILP class used for training. However, we will also review cases in which learning happens instance by instance or extends outside of a known MILP distribution. We will highlight these differences throughout the entire review of the learning tasks.

2.1.1. PRIMAL HEURISTICS

Primal heuristics play a crucial role in quickly finding feasible solutions and consequently improving the primal bound $\bar{z}$ in the early stages of the solution process. In the context of primal heuristics, ML techniques can be of interest to leverage common structures in the instances. A number of methodologies have been proposed for this purpose. Conceptually, they can be split into three main approaches: (a) guiding a heuristic search with a starting predicted solution, (b) solution improvement via a learned neighborhood selection criterion, and (c) learning a schedule to pre-existing heuristic routines. These methodologies are summarized in Figure 2.1 and will be discussed in more detail in the following. At the end of the section, Table 2.2 presents a summary of the reviewed work and their main characteristics.

An important class of primal heuristics are the so-called Large Neighborhood Search

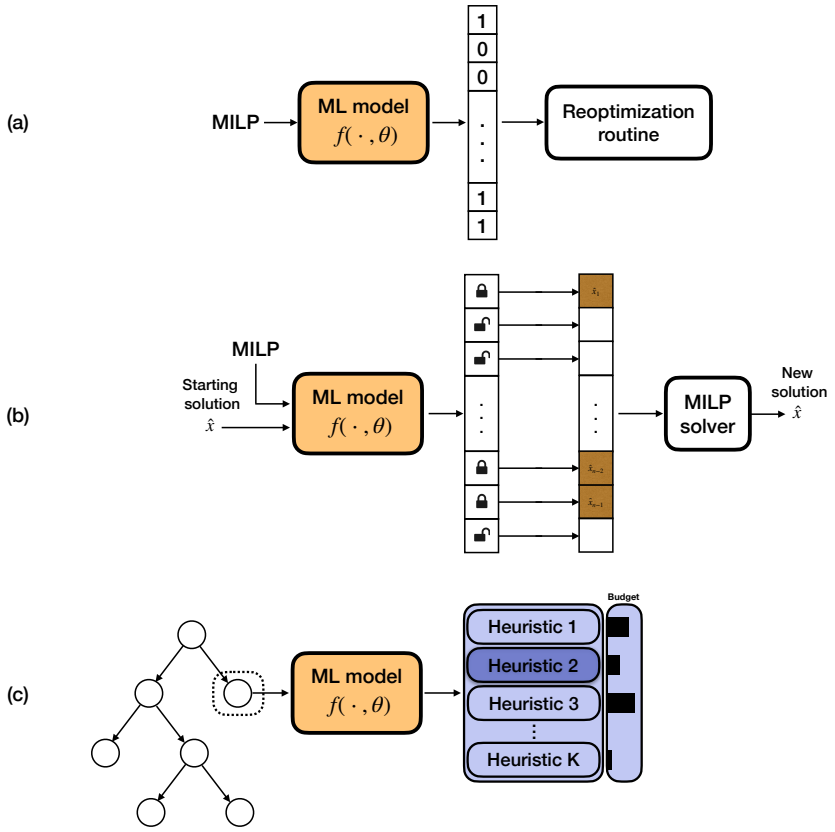


Figure 2.1: Three learning problems related to primal heuristics: (a) predict a reference solution and search in its neighborhood, (b) neighborhood selection – which and/or how many variables to unfix and re-optimize, (c) heuristic scheduling – which heuristics to run and/or for how long.

(LNS) [139] heuristics. The idea is to optimize an auxiliary MILP of smaller size, constructed by reducing the feasible region of the original MILP. This is typically done by fixing the value of some of the variables and optimizing the rest. Another strategy is to search over the neighborhood of a solution $\hat{\boldsymbol{x}}$ by imposing the constraint

$$\sum_{j:\hat{\boldsymbol{x}}_j=0} x_j + \sum_{j:\hat{\boldsymbol{x}}_j=1} (1 - x_j) \leq \eta, \tag{2.1}$$

with η the parameter that controls the neighborhood size. This is known as *local branching* [58]. The reader is referred to Fischetti and Lodi [59] for a survey on the use of LNS heuristics in MILP.

Table 2.1: Three methods to define the target probability distribution p_T, given a collection $D(X) = \{\hat{\boldsymbol{x}}^{(1)}, \hat{\boldsymbol{x}}^{(2)}, ..., \hat{\boldsymbol{x}}^{(K)}\}$ of feasible solutions to problem X. The goal is to learn a function $p_\theta(x_j = 1|X)$ that represents the probability that variable x_j takes value 1 given problem X by imitating the target probablity $p_T(x_j = 1|X)$.

Authors	Target
Ding et al. [51]	$p_T(x_j = 1\|X) = \hat{x}_j^{(1)}$ for every j s.t. $\hat{x}_j^{(k)} = \hat{x}_j^{(1)}$ for all $k \in \{1, ..., K\}$
Nair et al. [117]	$p_T(\hat{\boldsymbol{x}}^{(k)}\|X) = e^{-\boldsymbol{c}^T \hat{\boldsymbol{x}}^{(k)}} / \sum_{i=1}^{K} e^{-\boldsymbol{c}^T \hat{\boldsymbol{x}}^{(i)}}$ and $p_\theta(\hat{\boldsymbol{x}}\|X) := \prod_{j=1}^{n} p_\theta(x_j = \hat{x}_j\|X)$
Khalil et al. [89]	$p_T(x_j = 1\|X) = \frac{1}{K} \sum_{k=1}^{K} \hat{x}_j^{(k)}$

SOLUTION PREDICTION TO GUIDE THE SEARCH

Some authors explore the idea of using predictions of the optimal solution. The goal is to produce a (partial) assignment of the binary variables[1] in a binary or mixed binary MILP, that can then be used to guide the search. This idea has been implemented in different ways, both in terms of how to obtain the predictions and how to use them. Let us start by discussing the latter.

Ding et al. [51] use a local branching constraint (see Eq. 2.1) with respect to predicted values of a subset $J \subseteq \mathcal{B}$ of the binary variables. This restricts the search to a neighborhood of the predicted partial solution. In contrast, both Nair et al. [117] and Khalil et al. [89] propose to fix the variables in J to their predicted value and hand over this partial assignment to an MILP solver that optimizes over the remaining variables. In MILP terminology, this corresponds to a warm start. Khalil et al. [89] further use the predictions to guide node selection (see Section 2.1.4).

The question still remains as to how to obtain these predictions. This problem is naturally formulated as a supervised learning task, where the desired output is the optimal solution. However, optimal solutions to be used as labels in the training phase are costly to obtain and do not capture information about the region where they lie. The aforementioned work of Ding et al. [51], Nair et al. [117], Khalil et al. [89] make use of a set of feasible (not necessarily optimal) solutions to learn predictions. This is, the learning process starts with a data collection phase that, for each instance X, gathers a set of feasible solutions $D(X) = \{\hat{\boldsymbol{x}}^{(1)}, \hat{\boldsymbol{x}}^{(2)}, ..., \hat{\boldsymbol{x}}^{(K)}\}$. Once the data collection phase is finished, the prediction task can be formulated as learning a probability distribution $p_\theta(x_j = 1|X)$ that represents the probability that variable x_j takes value 1 given problem X, and is parametrized by θ. The parameters are tuned to make the behavior of these functions resemble as closely as possible that of a target probability distribution $p_T(x_j = 1|X)$. Table 2.1 summarizes the proposed targets.

[1] Note that the focus on binary variables is justified because, in most well-established instance collections, binary variables account for the great majority of integer variables. As an example, more than 68% of the instances in the MIPLIB [68] are purely binary, and in the remaining ones, more than 90% of the integer variables are in fact binary. See, e.g., Nair et al. [117] for extensions to integer variables.

SOLUTION IMPROVEMENT VIA NEIGHBORHOOD SELECTION

Alternatively to solution prediction, one may be interested in learning a *destroy heuristic* criterion. That is, given an initial feasible solution $\hat{\boldsymbol{x}}$, we select a subset of the integer variables to be re-optimized, leaving the remaining integer variables fixed. This process can be run iteratively. The goal is to identify substructures of the problem that can be used to decompose it into smaller, more manageable subproblems.

Following this line, Song et al. [141] learn to partition the set of integer variables $\mathcal{I}$ into K disjoint subsets $\{S_i\}_{i=1}^{K}$ such that $\mathcal{I} = S_1 \cup ... \cup S_K$. They iteratively unfix and re-optimize the variables in each subset, fixing the rest to the value in the best known solution. The variable subsets are of fixed size, which means that the variables simply need to be classified into the subset they belong to. Alternatively, Wu et al. [154], Sonnerat et al. [142] and Huang et al. [82] propose a method to select a single flexible-sized subset of variables to unfix. Their approach resembles the value-prediction methodology discussed in the previous section. However, instead of predicting the value that a variable takes in an optimal solution, they aim at predicting whether the variable is already assigned to its optimal value in the current best solution $\hat{\boldsymbol{x}}$. The number of unfixings is learnt implicitly through the conditionally-independent probabilities $p_\theta(\hat{x}_j = x_j^* | X)$ from which they sample variable unfixings. While Wu et al. [154] use a reinforcement learning algorithm to train their policy, Sonnerat et al. [142] explicitly calculate the best solution at most η unfixings away from $\hat{\boldsymbol{x}}$ and use it as a target unfixing policy. Huang et al. [82] follow a similar approach, while also providing the learner with negative examples. This means variable unfixings that do not lead to a sufficiently large improvement, as compared to the best. This provides additional information for the learner to distinguish good and bad unfixings.

The methods of Song et al. [141], Wu et al. [154], Sonnerat et al. [142] and Huang et al. [82] were compared against continuously running the MILP solvers they use as a subroutine. Given the same amount of time, the ML-assisted methods are able to find better solutions. Interestingly, using a random selection of the variables to unfix often results in a performance improvement compared to running the solver continuously. This highlights the fact that MILP solvers are typically tuned to optimize different performance metrics, such as the optimality gap, which means that a significant part of the computational effort is spent in, e.g., obtaining good dual bounds. Nonetheless, the proposed methods provide a relevant methodology to make use of the MILP solver in a black-box manner when the goal is to obtain high-quality but not necessarily optimal solutions in a short amount of time.

Related to this line of work, Liu et al. [106] point out that the optimal value of the neighborhood size parameter η is strongly dependent on the class of instances being solved. They work in the context of local branching and propose to learn two policies. A first policy $f_{init}(X)$, obtained in a supervised way, determines an appropriate neighborhood size η_0 for a first iteration. Another policy $f_a(X)$, obtained by RL, decides how to *adapt* the neighborhood size in successive iterations given the previous solver statistics. Compared to the simple neighborhood size selection proposed in Fischetti and Lodi [58], these two policies close the primal gap (see Eq. 1.6) faster, especially when combined together. Their computational experiments demonstrate that these selection rules have great generalization ability across different instance sizes and types, especially for

$f_a(X)$, also across different classes of instances, proving the wide applicability of their approach.

LEARNING TO SCHEDULE HEURISTICS

Whether they use ML or not, many successful primal heuristics for MILP have been proposed. Experimentally, we observe that no single heuristic dominates the others on all problems [79]. Their performance is highly dependent on the problem, and even on the solving stage. A new decision problem arises: given a collection of primal heuristics, which one should be run? In a broader sense, heuristic scheduling also tries to answer questions such as how frequently should heuristics be run, or under what computational budget. In this section, we review the line of research on using learning methods to answer such questions.

Hendel [78] proposes Adaptive Large Neighborhood Search (ALNS), a heuristic that, whenever called, makes a choice among eight Large Neighborhood Search methods. This choice is framed as a Multi Armed Bandit problem [32], an online learning methodology that learns a selection policy per instance. The learned policy is based on the observed (a posteriori) performance of the chosen heuristics. This simple formulation encapsulates the classical exploitation versus exploration dilemma: we must balance running heuristics that have performed well in the past with running heuristics whose performance is unknown because they have not been selected a sufficient number of times. The author uses the α-UCB algorithm [32] with a reward function that combines several aspects of a heuristic's performance: whether an incumbent was found, the objective improvement and the computational time the heuristic needs. The experiments on MIPLIB 2017 [68] show a considerable improvement in the primal integral (see Eq. 1.8), as well as a speed-up. The Multi Armed Bandit formulation was also used by Hendel et al. [79] and Chmiela et al. [38], who extend it to new types of heuristics.

Chmiela et al. [37] take a different perspective: instead of a selection policy per instance, they create a prioritization order that applies to a given class of instances. In fact, these authors' method constructs a schedule that assigns both a priority and a computational budget to each heuristic. In contrast with the abovementioned work of Hendel et al. [79] and Chmiela et al. [38], which use online learning, the schedule is now crafted offline, after a data collection phase where different heuristics are evaluated. They formulate a MIQP that minimizes the computational budget while finding feasible solutions for a large fraction of the B&B nodes, and they solve this using a greedy heuristic. The proposed method produces schedules for a number of different heuristics, which result in improvements in the primal integral compared to default SCIP and even a version of the solver that has been manually tuned for the considered class of instances.

Other than deciding which heuristics to run and for how long, there is the question of *when* to run them. The work discussed so far uses conventional rules to decide at which nodes heuristics will be run. The methodology proposed by Khalil et al. [88] is to instead build a mapping between B&B nodes and a yes/no decision for running each heuristic. This decision problem is challenging. Even with perfect knowledge of when a heuristic will be successful, a run-when-successful rule does not necessarily minimize solving time. Khalil et al. [88] analyze the competitive ratio of such a rule compared to an optimal offline decision policy. They then imitate the imperfect run-when-successful

Table 2.2: Summary of different learning approaches for primal heuristic management. We use the acronyms SL for supervised learning and RL for reinforcement learning.

	ML paradigm	Online/offline	Model
Ding et al. [51]	SL	Offline	GNN
Nair et al. [117]	SL	Offline	GNN
Khalil et al. [89]	SL	Offline	GNN
Song et al. [141]	SL & RL	Offline	PCA[72]+NN
Wu et al. [154]	RL, policy gradient	Offline	GNN
Sonnerat et al. [142]	SL, imitation	Offline	GNN
Huang et al. [82]	SL, contrastive	Offline	GNN
Liu et al. [106]	SL & RL	Offline	GNN
Hendel [78]	Multi-Armed Bandit	Online	-
Chmiela et al. [38]	Multi-Armed Bandit	Online	-
Chmiela et al. [37]	Greedy heuristic	Offline	-
Khalil et al. [88]	SL, classification	Offline	Logistic regression

rule by learning to predict when a heuristic will succeed. Their computational study shows the effectiveness of this method: heuristics are called less often, but with a higher success rate, resulting in a better primal integral. This effect is even stronger when the policy is trained on data coming from the same instance class.

FUTURE OUTLOOK

The success of most of the approaches discussed in Section 2.1.1 is rooted in the ability of learning within the distribution of specific classes of MILP instances. Generalizing outside of a specific class, has proven difficult so far. This is certainly the main challenge for the integration of ML-augmented primal heuristics within MILP solvers. Studies like the one in Liu et al. [106] indicate that some significant generalization is achievable, especially in the context of algorithms that sequentially adapt while exploring the solution space as in classical RL schemes. This is a promising direction that, however, conflicts with the MILP solver's general need of executing (extremely) fast primal heuristics. Further, in the context of solution prediction, the case of general integer variables has been comparatively less studied; see Nair et al. [117] for a discussion of how to treat this case.

2.1.2. BRANCHING

Branching is one of the core mechanisms on which the B&B algorithm operates. The branching rule, i.e., the criterion used to select a variable for branching, has been identified as having a critical impact on performance [7, 105]. Computational studies have served to identify a number of metrics that are good indicators of how a variable will perform. Notably, state-of-the-art branching rules look at the change in objective value in the resulting children nodes. More specifically, let z^{LP_i} be the objective value of the current node i and let $z^{LP_{i+}}$ and $z^{LP_{i-}}$ be the objective values of the two nodes resulting from

branching on candidate variable x_k. This variable will be scored using a combination[2] of $\Delta^+ := z^{LP_{i+}} - z^{LP_i}$ and $\Delta^- := z^{LP_{i-}} - z^{LP_i}$. These values can be explicitly calculated for each candidate at the moment of branching (thus solving two LPs per candidate). Such strategy, known as *strong branching*, was introduced in the context of the traveling salesman problem [15] and later standardised by CPLEX.

Strong branching is known to produce small B&B trees, but at high computational cost per branching. Alternatively, one can attempt to estimate the objective change based on past values, once they become naturally available through node processing. In particular, solvers typically store the values of Δ^+ and Δ^- normalized by the variable's fractionality, and keep track of the per-variable average, known as *pseudocosts*. The so-called *reliability branching* rule [8] performs strong branching at the top of the tree as an initialization phase, and then switches to using pseudocosts as soon as a variable has been branched on enough times. The initialization phase is not only important to build a branching history, but also because branching decisions have the most impact at the top of the tree. This is because judicious branching decisions here can lead to much smaller tree sizes, due to the earlier finding of feasible solutions and the stronger pruning of nodes.

In this section, we will discuss different approaches to learning to branch. Their common goal is to learn a function that maps a description of the candidate variables to scores. We note that alternative strategies exist, such as learning a weighting scheme for a portfolio of branching rules (see, e.g., Balcan et al. [18]). Ultimately, the objective is to minimize the solving time, which typically entails a favorable balance among different sub-targets. For example, (i) being computationally cheap, (ii) generating small trees as a result of their scoring, and (iii) adapting to the different situations that may arise. These objectives are often at odds with each other. At the end of the section, Table 2.3 presents a summary of the reviewed work and their main characteristics.

A First Approach to Learning from Strong Branching

There is a well-established body of research on fast approximations of the strong branching rule. This idea was first explored by Alvarez et al. [12] who propose to learn a prediction of the strong branching score of each variable. The predictor is learned offline (see Section 1.4) and tested on both heterogeneous and homogeneous instance collections. The experiments of Alvarez et al. [12] show that the method achieves the desired objective of imitating strong branching decisions without the large computational overhead. Indeed, when compared with strong branching on a fixed number of nodes, the closed gap is only moderately worse, indicating that the branching decisions are of high quality. Simultaneously, for a fixed time, the method explores a much larger number of nodes and closes a greater proportion of the gap, therefore achieving an overall better trade-off. However, reliability branching still outperforms the learned branching rule in closed optimality gap. The experimental results seem to indicate that the former makes smarter decisions, and is on average faster in making them. It is interesting to note that the method of Alvarez et al. [12] performs better on homogeneous instance collections, i.e., when the problems in the training and the test set are of the same type.

[2] For example, the variable's score can be computed as $\max\{\Delta^+, 10^{-8}\} \cdot \max\{\Delta^-, 10^{-8}\}$. See, e.g. Achterberg [5] for a discussion.

ONLINE LEARNING TO BRANCH

The results presented in Alvarez et al. [12] call for several reflections. First, experiments seem to indicate that performing strong branching at the top of the tree, where branching decisions have the most impact, is highly advantageous. Second, the authors already point towards more adaptation to the problem structure as a promising direction of improvement.

These ideas are studied in Khalil et al. [87] and the follow-up work of Marcos Alvarez et al. [112]. Independently, these authors proposed to use online learning with a strong branching initialization at the top of the tree. One key difference is that, while Marcos Alvarez et al. [112] continue to frame learning to branch as a prediction task (i.e., predicting the strong branching score of a variable), Khalil et al. [87] formulate the task as that of *ranking*. Indeed, one does not necessarily need to accurately predict the variables' scores, but rather which variables have relatively better ones. The latter task is easier from a learning perspective.

Framing an online learning task allows to learn a specialized ML model per instance, in this case a simple linear function. Khalil et al. [87] analyse their learned models by studying the weight assigned to each of the features. The first question they ask is: are the learned models obtained for each of the instances similar? Their analysis concludes that there is only a weak correlation among the models. This supports the idea that adaptation plays a key role. In spite of the learned models being quite different, they were able to identify some features that are often given high importance (large absolute weight), such as the product of the pseudocosts, and data related to the constraint matrix. For a more detailed discussion of this analysis see Khalil [86].

OFFLINE LEARNING WITH STRUCTURE SPECIALIZATION

Marcos Alvarez et al. [112] and Khalil et al. [87] use a linear mapping from variable features to output. This offers the advantage of interpretability and low computational cost. The more complex GNN model proposed by Gasse et al. [66] (see Section 2.2 for a description) has proven to be remarkably effective in representing MILPs for the task of variable selection and beyond. The GNN architecture that we presented in Section 1.4.1 consists of a number of parametric function compositions that enable learning more complex relations between input and output. They also require a larger amount of training data, which rules out the online learning methodology previously discussed. In order to still ensure some level of specialization, Gasse et al. [66] propose a middle point: they argue that in many realistic cases instances of the same *class* are routinely solved. This justifies learning a branching rule per instance type, as a sensible trade-off between a completely general rule (such as the one in Alvarez et al. [12]) and a completely instance-tailored rule (like in Marcos Alvarez et al. [112], Khalil et al. [87]).

Gasse et al. [66] propose training this GNN to imitate strong branching via behavioral cloning [127]. In short, this means that we again disregard the actual variable scores and focus on learning relative magnitudes among them. Through this approach the authors were able to outperform reliability branching, marking a breakthrough in the learning to branch literature. Building on this work, Gupta et al. [73, 74] propose modifications of the original loss function that further improve the performance of the learned model. Nair et al. [117] test the methodology of Gasse et al. [66] on a variety of instance types,

including heterogeneous sets. Seyfi et al. [138] additionally propose a mechanism that incorporates information about past branchings into the scoring system.

TOWARDS A GENERAL BRANCHING RULE

We have discussed methodologies that specialize to certain combinatorial structures (such as Gasse et al. [66]) or that yield a custom strategy per instance (e.g., Khalil et al. [87]). As discussed at the beginning of this section, these methodologies are a response to the great difficulty of learning one unique policy on a heterogeneous set of instances. Zarpellon et al. [156] argue that, while learning such a general policy poses a big challenge, it is possible to overcome it by using a representation of the search tree to inform variable selection. Their hypothesis is that there is a higher-order shared structure among MILPs, even among those with different combinatorial structure, and that this shared structure can be captured in the space of B&B trees. To test this they define a set of features describing the state of the search which, together with variable descriptors, are mapped into scores. We discuss these features in more detail in Section 2.2. The experiments in Zarpellon et al. [156] show that, while the model of Gasse et al. [66] struggles to learn over a heterogeneous data distribution, adding the tree context is beneficial to the generalisation performance of the model. Lin et al. [104] take this idea one step further and propose to keep a record of the features in [156] at each node where branching was performed. At every step, branching decisions are informed by this historical data, which is carefully aggregated and combined with the descriptions of the variables.

The work in [156, 104] highlights the potential of using solver statistics to influence branching decisions. It is unclear how the features proposed in Zarpellon et al. [156] are used to score branching candidates, since neural networks lack explainability. Nonetheless, this information certainly opens the door to branching rules that switch among different behaviors at different parts of the tree, or stages of the solving process. It is important to note that reliability branching still outperforms both Zarpellon et al. [156] and Lin et al. [104], perhaps because of the overhead of computing and processing these comprehensive descriptors. Nonetheless, this work calls for further research on exploiting tree information.

EXPERT-FREE LEARNING TO BRANCH

So far, we have discussed methods for building fast approximations of strong branching. The general consensus is that strong branching yields relatively small B&B trees compared to other classic branching strategies, and it is therefore advantageous to try to imitate it. This idea can be challenged with two arguments. First, as pointed out by Gamrath and Schubert [62], standard implementations of strong branching benefit from using data obtained as a by-product of the score calculation, such as bound tightenings and other statistics. A rule that *imitates* strong branching cannot profit from such side-effects, which means that, even in the case of perfect emulation, the imitator's performance would be worse than expected. Second, strong branching relies on the LP relaxation for scoring variables, which can provide little information in cases when the optimal LP objective value does not change with branching. In such cases strong branching cannot be considered a reliable expert. In the absence of a better alternative to strong branching, the following question arises: can we learn branching rules without expert knowledge? This question is addressed by Etheve et al. [55] and Scavuzzo et al. [135],

Table 2.3: Summary of different learning approaches for branching. We use the acronym SL for supervised learning and RL for reinforcement learning.

	ML paradigm	Online/offline	Model
Alvarez et al. [12]	SL, regression	Offline	ExtraTrees
Marcos Alvarez et al. [112]	SL, regression	Online	Linear
Khalil et al. [87]	SL, ranking	Online	Linear
Gasse et al. [66]	SL, imitation	Offline	GNN
Etheve et al. [55]	RL, Q-learning	Offline	NN
Scavuzzo et al. [135]	RL, policy gradient	Offline	GNN
Zarpellon et al. [156]	SL, imitation	Offline	NN
Lin et al. [104]	SL, imitation	Offline	Transformer

who use RL to learn branching rules from scratch. The computational study in Scavuzzo et al. compares one such RL methodology against the imitation learning method of Gasse et al. [66]. The experiments show that on instances where strong branching performs very well, the imitation method of Gasse et al. is superior. Yet, when strong branching struggles, the RL-based method is able to find a better branching strategy, proving the point that expert-free learning is of great interest in certain cases. A more in-depth discussion of this topic is presented in Chapter 3.

FUTURE OUTLOOK

Adaptiveness. The computational studies suggest that no single branching rule outperforms others universally across all instances. Consequently, a desirable approach involves a rule that dynamically adapts its behavior to the specific characteristics of each situation. One way in which one can introduce such adaptiveness is by controlling the distribution of data samples that the model uses for learning. Some authors propose to specialize to each given instance. This means that a set of parameters $\boldsymbol{\theta}$ is generated for each new instance, obtained by allowing the ML model to only see data coming from that instance. Another possibility is to use data samples coming from instances of the same problem class. This gives us parameters $\boldsymbol{\theta}$ that specialize to a given class and work well, on average, on different instances with shared combinatorial structure. We can therefore achieve adaptiveness on different scales. We can also understand adaptiveness of a branching rule as some sort of mechanism to change its behavior on different parts of the search process. Some progress has also been made in this regard by investigating different statistical measures that can inform branching (see Section 2.1.2). Yet, little is understood about how these metrics are used or in which ways we can further enhance performance without sacrificing speed.

Expert guidance. The strong branching heuristic has been used by many as an expert from which we can learn effective decision-making. The claim that strong branching is a desirable strategy to follow has recently been challenged, with some notable examples of instances where strong branching scores provide no useful information. Interesting research directions include identifying new experts, new strategies to better imitate them, or, conversely, more efficient approaches to learning without expert knowledge.

The latter case is what is referred to in Bengio et al. [20] as *experience*: there is no clear mathematical understanding of what should be statistically learned (the expert), so exploration should be performed. In turn, this clearly calls for RL methods that are also natural candidates to extend the work in Zarpellon et al. [156], Lin et al. [104] and exploit tree information.

New directions. The work we surveyed showcases the potential in mixing the extensive body of domain knowledge in variable selection with new learning techniques. Still a lot of open questions remain. For example, little attention has been directed towards highlighting important subsets of variables, as opposed to choosing a single one at each node. Khalil et al. [90] propose an approach to finding such important subsets, in this case the so-called *backdoors* [50], and show promise in using them as prioritized branching candidates. Another relevant gap is the absence of expert knowledge for certain classes of MILPs. In any case, it is clear that new ML-based methods need to build upon the pre-existing knowledge on variable selection to achieve a fruitful combination.

2.1.3. CUTTING PLANES

Cutting plane routines are another essential part of modern MILP solvers. They tend to work in *rounds*, also called *separation rounds*. Given an LP-relaxation solution $\boldsymbol{x}^{LP}$, one round consists of generating a number of valid cuts from different families, selecting a subset of them via a selection criterion, adding them and finally resolving the LP relaxation, where $\boldsymbol{x}^{LP}$ will now be infeasible. A good selection criterion is critical to improving the LP relaxation while avoiding an excessive number of cuts, which would slow down LP solving as well as lead to numerical instability [14, 46]. Several metrics have been proposed for the purpose of scoring cuts (see, e.g., Wesselmann and Stuhl [150] for an overview). For example, the *objective parallelism*, measured as the cosine of the angle between the objective function and the cut, or the *cutoff distance*, measured as the Euclidean distance between the cut and the LP-relaxation solution. More recently the question of cut selection has been addressed with ML-driven predictions, which is the topic of this section. We review different work in this area and, at the end, present in Table 2.4 a summary of the different methodologies and their main characteristics. For further discussion of ML for cut selection in MILP and beyond we refer to Deza and Khalil [48].

As noted in Section 1.3.3, cuts are typically more heavily applied at the root node, and for this reason the work that we survey focuses on cut selection at the root node. Still, there is no obstacle to applying these methods in other nodes. However, it is unclear whether using cuts outside the root node is computationally beneficial (see Berthold et al. [26] for a discussion of this topic).

SINGLE-CUT SELECTION

Tang et al. [144] and Paulus et al. [126] frame the task of cut selection as an MDP. At each step k, a single cut c_k is selected from a cutpool $\mathscr{C}_k$, after which the LP relaxation is resolved. In particular, let $\mathscr{C}$ be a collection of cuts, and let $z(\mathscr{C})$ be the result of solving the root LP relaxation after adding all cuts in $\mathscr{C}$. The metric of interest to these authors

is the LP-bound improvement attained by a cut c at step k, defined as

$$\Delta_k(c) := z(\{c_1, ..., c_{k-1}, c\}) - z(\{c_1, ..., c_{k-1}\}) .$$

This MDP model is summarized in Figure 2.2a, where we use the abbreviation $z_k = z(\{c_1, ..., c_k\})$.

Paulus et al. [126] use imitation learning. Their expert is the result of explicitly calculating $\Delta_k(c)$ for each possible cut c in the cutpool $\mathscr{C}_k$, and then picking the cut with highest Δ_k. This means that their approach is greedy: they look at bound improvement one step ahead. Their computational studies show that this greedy heuristic that they aim to imitate is in fact very effective in improving the LP bound after T cuts have been added, compared to other selection heuristics.

Another approach is that of Tang et al. [144], who use RL with reward $R_k = \Delta_k(c_k)$. Due to to the discount factor (see Equation 1.10), this RL strategy offers the possibility to learn less greedy policies, and doing so without explicitly computing $\Delta_k(c)$ for each k and $c \in \mathscr{C}_k$. However, many RL algorithms are known to suffer from sample inefficiency and lack of generalization [53, 93]. Paulus et al. [126] compare their approach to the one of Tang et al. [144] and to SCIP's v.7.0.2 default rule. They use the sum of gaps as a metric (lower value is better)

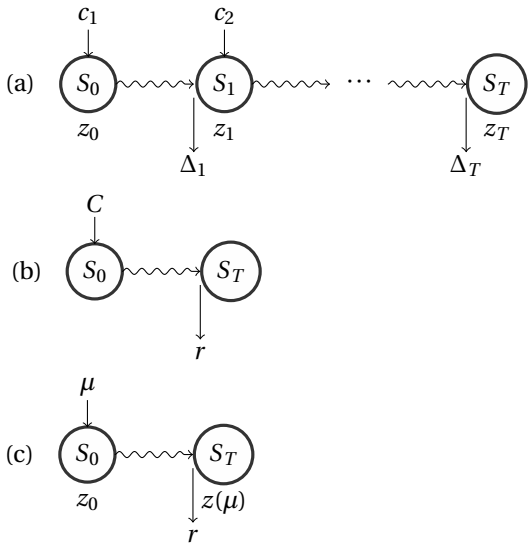
$$SG := \sum_{k=1}^{T} \frac{z^* - z_k}{z^* - z_0}, \tag{2.2}$$

with $T = 30$ being the total number of cuts added, and z^* being the pre-computed optimal solution. Notice that this metric is constructed in a way that might favour greedy policies. The computational results favor the method of Paulus et al. [126]. The authors also show promising results in solving time when their method is incorporated into SCIP and instances are solved to optimality, though they do not include root node processing time.

MULTI-CUT SELECTION

A potential criticism to the cut selection models is that solvers usually add more than one cut per round, in order to reduce the number of times the LP needs to be resolved. In fact, historically this proved crucial to the efficiency of cutting plane routines [17, 42]. Having information about the LP solution after the addition of each cut is therefore unrealistic. Paulus et al. [126] do not include root node processing times in their report, a metric under which their method is likely unfavored. Furthermore, metrics like the one in Eq. 2.2 encourage greedy bound improvements, whereas in practice cuts can work together to achieve a better bound improvement at the end of the round. In other words, the optimality gap closed after a full separation round is likely a more informative metric.

To address the potential interactions among cuts, Wang et al. [149] propose a policy that decides the fraction of cuts from the pool to be selected, and scores ordered subsets of this size. See Figure 2.2b for a summary of this selection model. The authors train the policy with an RL algorithm and use end-of-run statistics like solving time as the reward. This requires solving an MILP to optimality for each training sample. While this allows to learn a mapping from cut selection to actual solver performance (instead of the root node bound improvement, which is just a proxy for performance) this sample collection comes at a great computational cost.



Action: choose a cut c_k from the cut pool $\mathscr{C}_k$
Transition: apply cut, resolve LP
Objective: $\Delta_k = z_k - z_{k-1}$

Action: an ordered subset C of the cut pool $\mathscr{C}$
Transition: apply the cuts and solve the instance
Objective: $r = -PD(t_{max})$

Action: parameters $\mu \in \mathbb{R}^4$
Transition: N separation rounds with parameters μ
Objective: $r = \frac{z(\mu) - z(\mu_{def})}{z^* - z(\mu_{def})}$

Figure 2.2: Three models for learning to cut: (a) Tang et al. [144], Paulus et al. [126], (b) Wang et al. [149], (c) Turner et al. [145]. Here $PD(t_{max})$ refers to the primal-dual integral (see Section 1.3.4).

A third model to learning cut selection is proposed in the work of Turner et al. [145] (see Fig 2.2c). The procedure builds upon SCIP's default strategy, a rule that has been carefully curated through computational studies [5, 150]. This rule combines four cut scoring functions $s_i : \mathbb{R}^{n+1} \to \mathbb{R}_+,\ i = 1,\dots,4$ via a convex combination

$$s(c,\mu) = \mu_1 s_1(c) + \mu_2 s_2(c) + \mu_3 s_3(c) + \mu_4 s_4(c)\,,$$

$$\sum_{i=1}^{4} \mu_i = 1, \quad \mu_i \geq 0, \quad i \in \{1,2,3,4\}\,.$$

Cuts are then selected sequentially based on their score (the higher the better) and their parallelism to already selected cuts (too 'parallel' cuts are discarded). We refer to Turner et al. [145] for the definition of the scoring functions s_i, $i = 1,...,4$. The problem of choosing a good set of parameters $\mu \in \mathbb{R}^4$ has been studied from a learning theoretical perspective by Balcan et al. [19]. As a next step, Turner et al. [145] argue, through both theoretical and computational arguments, in favor of adapting the coefficients μ to the instance being solved, as opposed to a unique, static choice. One of their experiments consists of finding a custom set of parameters μ per instance through grid search. While impractical, this experiment uncovers the large potential for improvement when adapting the value of μ. In order to exploit this potential in a more realistic way, Turner et al. [145] devise a policy that, given an instance, chooses custom parameters μ, and they train it via RL. The computational study shows that the learned policy is competent in its task, outperforming random selection in terms of closed optimality gap at the root node. However, and perhaps surprisingly, their policy does not perform consistently better than SCIP's default settings when it comes to final solving time.

Table 2.4: Summary of different learning approaches for cut management. We use the acronym SL for supervised learning and RL for reinforcement learning.

	ML paradigm	Online/offline	Model
Tang et al. [144]	RL, evolutionary strategies	Offline	Attention + LSTM
Paulus et al. [126]	SL, imitation	Online	GNN
Wang et al. [149]	RL, policy gradient	Offline	LSTM + NN + Pointer
Turner et al. [145]	RL, policy gradient	Offline	GNN
Li et al. [103]	Contextual bandits	Offline	GNN

BEYOND SCORING

Cut scoring for selection is an essential part of cut management. Yet, there are other important decisions. Wang et al. [149] incorporate the number of cuts added as a decision that the policy must make. Very recently, Li et al. [103] defined a learning task for separator configuration. The objective is to select a subset of the available separators. Only the selected separators will then be active and contribute to the cutpool, meaning that this selection step happens *before* the cut selection phase. Li et al. [103] propose a methodology to overcome the high dimensionality of the configurations space, which is $2^{M \cdot R}$, with M the number of separators and R the number of cutting rounds. Their experimental findings show a lot of promise. More research into adapting other parametric choices could provide further insights.

FUTURE OUTLOOK

Measuring performance. What is the purpose of the cutting routines? One is inclined to believe that cuts should strengthen the LP relaxation, hence bringing the LP bound closer to the optimal value. However, will this result in a faster solve? Turner et al. [145] experimentally measure the (kind of folklore) fact that a better root LP bound does not always translate into a shorter solving time. Other than the clear influence of the number of added cuts, many solver components can be affected by the cut choice resulting in different performance. Wang et al. [149] address this by incorporating solving time as a reward signal, instead of root LP bound. However, observing final performance comes at great computational cost, which could be prohibitive for larger instances. More research is needed on how to efficiently navigate this trade-off.

Multi-cut rounds. The selection model of Tang et al. [144] and Paulus et al. [126] adds one cut at a time, resolving the LP at each iteration. Their work constitutes an important step towards learning cut selection rules. However, in a practical setting, such procedure is unlikely to outperform models that do limited LP resolving by adding multiple cuts at once. Going forward, models like the one of Wang et al. [149] or Turner et al. [145] have more potential for improvement.

2.1.4. NODE SELECTION

Primal heuristics have the clear goal of improving the best known MILP-objective value (primal bound). Analogously, branching rules and cutting routines are typically designed to improve the dual bound. Node selection policies have the difficult task of balancing both goals, which are often at odds. As usual, a better node selection rule is one that results in the shortest solving time. This is typically associated with smaller search trees. For that, one needs to avoid processing nodes that could be pruned if the optimal solution was known in advance. Finding a good (or even optimal) solution fast makes that task easier to accomplish.

One strategy is to first process nodes with the best (lowest) known lower bound. This is called best first search (BFS) and has the benefit of quickly improving the dual bound. Note that for all B&B trees there is a node selection policy of BFS type[3] that minimizes the number of processed nodes [5]. However, it is well known that such an approach delays finding good (or even optimal) solutions. The depth first search (DFS) strategy prioritizes children or siblings of the node that was last processed. This approach sequentially provides more constrained sub-MIPs therefore increasing the chances of finding feasible solutions both naturally and via primal heuristics. Furthermore, DFS has the added benefit of faster node processing times, on account of the similarity between sub-problems that are solved consecutively, which usually differ in one variable bound change. In practice, node selection rules alternate between both behaviors, while considering other statistics about branching that allow for estimating the cost of integrality.[4]

Different methodologies have been proposed for learning a node selection strategy over a homogeneous instance set (see Table 2.5 for an overview). He et al. [77] propose to learn from a node selection oracle that always chooses to process the node that is on a path to the optimal solution. This requires knowledge of the optimal solution during the training process, but not at test time. Similarly, Yilmaz and Yorke-Smith [155] prioritize nodes that contain high-quality solutions using a policy that always picks a child of the current node and uses an ML-based prediction to choose among these children. Once the dive is finished, they propose different ways in which the next node can be selected. Labassi et al. [96] learn a function that compares any two nodes in the tree. This can be used to substitute lower bounds as the proxy of a node's potential. The learned comparison function can be used in combination with different selection strategies, such as picking the child node with highest potential. Finally, Khalil et al. [89] guide the search based on a prediction of the optimal solution. They look at the fixed variables at each node and measure the similarity between the fixed and predicted values. This favors nodes where the partial solution aligns with the predicted solution.

FUTURE OUTLOOK

The experimental results of the papers surveyed above show promise, yet the margins of improvement remain small. Often, an effective heuristic schedule and branching strategy are much more crucial and, when chosen correctly, make the impact of the node selection strategy relatively small. Still, it is interesting to observe that ML has opened new opportunities in an area where research has been pretty much inexistent for decades.

[3] There can be more than one BFS policy because of ties.

[4] The literature in node selection is not extensive, good discussions can be found for example in [5, 9].

Table 2.5: Summary of different learning approaches for node selection. We use the acronyms SL for supervised learning.

	ML paradigm	Online/offline	Model
He et al. [77]	SL, imitation	Offline	Linear
Yilmaz and Yorke-Smith [155]	SL, imitation	Offline	NN
Labassi et al. [96]	SL, classification	Offline	GNN
Khalil et al. [89]	SL, classification	Offline	GNN

This suggests that there is some potential for looking at an “old" problem from a different perspective and with new tools. For example, one could pair different node selection strategies with restarts [13], i.e., changing the node selection in a more dramatic way over time.

2.1.5. CONFIGURATION DECISIONS

MILP solvers are highly parametric. To illustrate this, consider SCIP version 8.0.0: it has more than 2000 parameters that the user can tune. A good parameter configuration that suits the instances being solved (for example a certain class of instances) can have a crucial effect on the solving process. Again, we can look at this problem with a Machine Learning lens: we can base certain parametric decisions on a prediction given by an ML model. One can see this as falling under the realm of Automatic Algorithm Configuration (AAC). However, AAC methods typically entail configuring a large number of parameters at the same time (see, e.g. Hutter et al. [83]). Here, we are interested in the use of ML to answer a single parametric question or, at least, one question at a time. Furthermore, and contrary to other AAC methodologies, the ML models make use of a description of the instance. These models are trained on a heterogeneous collection of instances and allow for instance-dependent parameter prescriptions, as opposed to a single configuration for the given instance distribution. Some work in this area is summarized in Table 2.6. Notice that the aforementioned work of Turner et al. [145] on cut selection can also be seen in this light. The work of Hendel et al. [79] can be further highlighted as a dynamic parameter setting scheme. They update the pricing rule throughout the search depending on the observed runtime in the already processed nodes.

This avenue of research has already fostered considerable success. Notably, the method presented in Berthold and Hendel [24] is used by default in FICO Xpress v.8.9. The great potential of customised configurations is highlighted in problems where the preprocessing techniques have a broader impact, such as in mixed integer nonlinear programming. A prime example of this is Bonami et al. [30, 31], where the authors prescribe for each mixed integer quadratic programming instance if the quadratic objective function should be linearised or not. However, note that preprocessing has been shown to be the single most impactful component of MILP solvers [7, 10], hence the use of ML to configure the MILP algorithmic decisions based on the characteristics of an instance or a class of instances has a strong potential.

Table 2.6: Five examples of using ML to set a solver parameter. The ML model is responsible for answering a single parametric question by choosing one of the available options.

	Component	Question	Options
Kruber et al. [95]	General	Should the Dantzig-Wolfe decomposition be used?	yes / no
Hendel et al. [79]	LP solver	Which simplex pricing rule to use?	devex / steepest / quick-start steepest
Berthold et al. [24]	Presolve	Which scaling method to apply?	Standard / Curtis-Reid
Berthold et al. [26]	Cutting	Should cuts be applied outside the root node?	yes / no
Turner et al. [145]	Cutting	How should cut scores be weighted?	$\mu \in \mathbb{R}^4_{\geq 0}$

FUTURE OUTLOOK

Related to the work discussed in this section is the line of research dedicated to predicting search completion (see, e.g., Fischetti et al. [60], Hendel et al. [80]) by using a range of solver statistics. This is an established field with contributions to state-of-the-art solvers, e.g., the incorporation of a search completion estimation in SCIP v.7.0. These predictions can be used to trigger a restart [13], a technique that has gained a lot of attention and that allows to *re*configure algorithmic decisions based on the evolution of the solving process. Following this line of thought, one can envision that, in the future, ML models such as the ones in Table 2.6 can be used not just during preprocessing, but also to prescribe a change in strategy during the solve. This is especially appealing because before a restart some (sometimes) extensive data collection has happened, data to be leveraged by ML. Along the same lines, it is worth mentioning the recent attempts to leverage data to better solve sequences of MILP instances that differ very little from each other. This has been the focus of the 2023 MIP challenge [29], and again it pertains to effectively configure an MILP solver, i.e., its algorithmic decisions, by exploiting data associated with instance similarities and data collected from previous runs. For the challenge, several classes of instances were proposed, where the instances in each class differ very little, for example, only in the coefficients of the objective function. The solver that won the competition [125] was able to leverage the data of the (previous) runs, for example, the pseudocosts for making better branching decisions.

Many open questions in this vast research area still exist, making this a promising area of future work.

2.2. PROBLEM REPRESENTATION

The standard form of Mixed Integer Linear Programs is the one presented in Equation 1.1. Given $A \in \mathbb{Q}^{m \times n}$, $\boldsymbol{c} \in \mathbb{Q}^n$, $\boldsymbol{b} \in \mathbb{Q}^m$ and the partition $(\mathscr{A}, \mathscr{B}, \mathscr{C})$ of the variables, an MILP solver can start solving the instance. The ML contributions to the solving process surveyed in Section 2.1 also require information about the problem, but the data $\{A, \boldsymbol{c}, \boldsymbol{b}, (\mathscr{A}, \mathscr{B}, \mathscr{C})\}$ may be insufficient or unfit for the task at hand. In this section we review different methodologies to construct a representation X of the problem being solved. This

representation is the input to a parameterized policy $f(X,\theta)$ to be trained for a specific task.

Let us start by listing the desirable properties of a representation.

1. Permutation invariance:[5] permuting the order of the variables and/or constraints should leave the representation unchanged.

2. Scale invariance: scale invariance is preferred to keep values within controlled ranges, which helps the learning process. This can be achieved with a normalization step.

3. Size invariance: the size of the representation should not depend on the size of the instance. This is, we require a fixed-sized description of each element that needs representation, e.g., each variable or each node.

4. Low computational cost: low cost of extracting, storing and processing the data.

In the following, we will make a distinction with respect to descriptors that represent general properties of the MILP and descriptors that relate to a specific variable or constraint. These descriptors may be static in nature or they may dynamically change during the solving process. We will also discuss global descriptions of the process versus local (subproblem) properties. Note that some approaches use no description of the instance, and instead learn exclusively from the performance metric (see, e.g., Chmiela et al. [37], Hendel [78]).

2.2.1. THE BIPARTITE GRAPH REPRESENTATION

MILPs can be represented as a bipartite graph, as shown in Figure 2.3. This graph is constructed as follows: each constraint and each variable have a corresponding representative node. A constraint node is connected to a variable node if the corresponding variable has a non-zero coefficient in the corresponding constraint. Each node has an associated vector descriptor. The advantage of using a graph representation is that this data structure can be parsed by a Graph Neural Network (GNN, see Section 1.4.1 for a formal definition). This type of architecture automatically handles inputs of different sizes. The data aggregation step that ensures size invariance is one of the learnable mappings (function `comb` in Definition 3). This is, instead of manually engineering a mechanism to aggregate information, this mechanism is automatically *learned*. The use of GNNs for combinatorial optimization has experienced a rise in popularity in recent years [34] because of their ability to capture the structural properties of the instances without excessive engineering.

2.2.2. REPRESENTING VARIABLES INDIVIDUALLY

Some of the learning tasks discussed in Section 2.1 require a description of each variable individually. Clearly, branching rules fall under this category. This is also the case for

[5] Note that permutation invariance is an issue beyond the ML context. The performance of MILP solvers can be affected by a change in the order of the variables or constraints. Such seemingly irrelevant changes that have an impact on the solution process are a known issue called *performance variability* (see Lodi and Tramontani [107]).

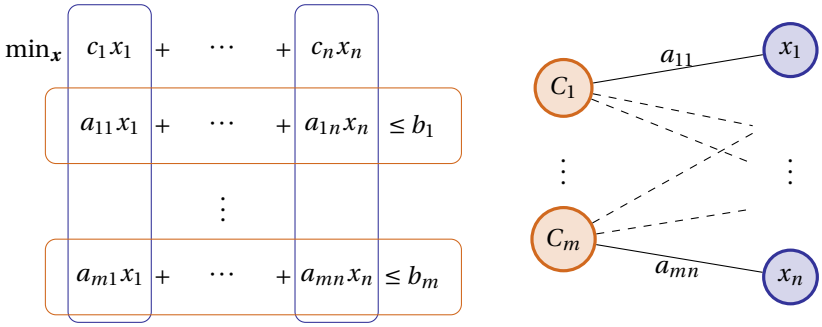


Figure 2.3: The bipartite graph representation of an MILP.

prediction-driven heuristics and neighborhood selection policies, for which variables are mapped to values or probabilities. In this section, we will discuss three important approaches to building variable representations and how they relate to the different approaches of Section 2.1.

A straightforward approach to building variable representations is to gather a number of descriptors into a *vector representation* for each variable. Khalil et al. [87] propose a number of such descriptors, including different statistics about the set of constraints in which each variable participates. These statistics aggregate information whose length would otherwise depend on the problem size. For example, for a variable $j \in \mathcal{I}$, using the constraint coefficients $\{a_{ij}\}_{i=1}^{m}$ would yield a vector whose length depends on m, which is undesirable. On the contrary, using the average of these coefficients gives a size-independent descriptor. This is a necessary step but calls to question which statistics should be included or excluded in this feature engineering step.

Alternatively, one can use the bipartite graph representation of the MILP as described in Section 2.2.1. Gasse et al. [66] were the first to use such representation to make predictions about variables, using a GNN as a mapping. The descriptors associated to the elements in the graph include structural information, such as constraint coefficients. This data is not given in the form of aggregated statistics like before. Instead, the aggregation function is part of the learnable mappings of the GNN (see Definition 3). Apart from the structural information, both Khalil et al. [87] and Gasse et al. [66] include information about the LP solution and other basic variable features such as their objective coefficient or variable type. Zarpellon et al. [156] take a different perspective, stressing the importance of historical data collected during the B&B tree. This strategy resembles SCIP's default branching rule, which considers information about past branchings, collected conflicts or cutoffs. The representation used in Zarpellon et al. [156] includes this variable information and, additionally, global information about the search tree. They argue that such a description can uncover shared structures among very diverse MILPs (see Section 2.1.2). The information collected includes statistics about the node being processed, tree composition and shape, and bound statistics, with a particular focus on unprocessed nodes.

These three approaches to MILP variable representation are compared in Table 2.7. We report on features that describe variables individually, therefore excluding the com-

plete list of tree features of Zarpellon et al. [156] (which can be found in their appendix). Table 2.7 showcases that, while some common features exist, the different representations have distinct focuses.

Most of the work surveyed in Section 2.1 uses the graph representation of Gasse et al. [66], either exactly (e.g., [142, 74, 135]) or with small modifications of the variable descriptors (e.g., [154, 106, 89]). Hybrid models also exist. In particular, Gupta et al. [73] propose extracting the Gasse et al. representation at the root node and the Khalil et al. [87] representation in the rest of the nodes of the search tree. The reason is that, while the graph representation is rich, it is also computationally expensive. Gasse et al. overcome this by using a GPU (graphics processing unit) to accelerate the execution of their learned function. Such computation on GPUs is common practice in the ML community, but one could argue that it is unrealistic to require the availability of a GPU. The hybrid model of Gupta et al. [73] reuses the rich but expensive representation of the root node in combination with the features of Khalil et al. [87] that are cheap and update the description at every node. This proves very effective, with their best performing model outperforming both reliability branching and the model of Gasse et al. when executed without GPU acceleration.

2.2.3. REPRESENTING CONSTRAINTS INDIVIDUALLY

Analogously to variables, a description of the problem's constraints may be needed. Here, we refer both to original problem constraints and additional valid constraints that can be added as cuts. A constraint representation is necessary in two cases. First and undoubtedly, whenever the task requires a decision over said constraints (e.g., which cuts to add). Second, the bipartite graph representation of MILPs discussed in the previous section (see Figure 2.3) also calls for a description of the constraints, even in the case where they are to be aggregated at a later stage.

Gasse et al. [66] first proposed the bipartite graph representation using a small number of descriptors for the constraint nodes. In particular, they use the cosine similarity[6] with the objective coefficients, the constraint right-hand-side, and LP information such as basis status and dual bound. This type of concise descriptions of the constraints is frequently used for the learning tasks associated with branching or primal heuristics.

In the case of cut selection, a more detailed description is preferred. Paulus et al. [126] extend the graph representation of Gasse et al. [66] with metrics that are typically considered in cut selection, such as violation, objective parallelism or sparsity. Wang et al. [149] describe each cut with a single vector of classical cut scores (see. e.g., [150]). In the case of Turner et al. [145], the classical cut scores are intrinsically taken into account in the definition of their learning task (see Section 2.1.3). For this reason, they use a graph representation of the model with a small amount of variable and constraint descriptors.

2.2.4. REPRESENTING A (SUB-)MILP

To conclude, it is worth observing that some decision tasks are formulated at an instance or node level, therefore needing a global representation of the MILP and perhaps also the solving process. One possible approach is to aggregate variable and constraint descrip-

[6] The cosine similarity between two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ of the same length is defined as $\boldsymbol{a} \cdot \boldsymbol{b} / ||\boldsymbol{a}||_2 ||\boldsymbol{b}||_2$, that is, their dot product divided by the product of their norms.

tors coming from the bipartite graph representation. Liu et al. [106] use the average of the variable descriptors, while Labassi et al. [96] concatenate the average variable descriptors and the average constraint descriptors.

Other approaches include the one of Khalil et al. [88], who, in the context of scheduling of primal heuristics, build a vector representation of the current node. This representation includes comparisons to the root node and context on the node's position within the tree. Great focus is put on information coming from the LP solution, such as the objective value, average fractionalities, and statistics on the constraint activity. Berthold et al. [24, 26] build representations that are more specialized to the particular configuration task. We refer to their work for a more detailed discussion of the problem descriptors.

2.2.5. OUTLOOK

In this section, we have reviewed different methodologies that build a suitable representation of MILPs that can be parsed by learning models. This includes both the selection of relevant data and data structures. At the beginning of the section, we anticipated the desired properties of such representations. All of the presented methodologies ensure permutation invariance and, to some degree, try to normalize data to establish some scale invariance without excluding important information. We also discussed the importance of the graph representation to handle input of different sizes. In the following, we elaborate on the importance of Graph Neural Networks for MILP representation, as well as the determining trade-off between model expressivity and speed.

Expressivity versus speed. In the context of machine learning, model size refers to the number of parameters and operations that define the mapping function $f(\cdot, \boldsymbol{\theta})$. Larger models allow us to learn more complex relationships between input and output. However, there is a clear trade-off between model size and computational cost of execution and training. In stark contrast to other fields of application of ML, such as large language models [120] or strategic game playing [147] where the goal is super-human performance, the computational cost per execution of the ML model is decisive to whether or not it will beat its competitors, i.e., already highly efficient optimization software. It is desirable for the MILP representation to ensure low computational cost, both in terms of data extraction and processing. The work discussed in Section 2.1 shows that navigating this trade-off is an active field of research, with promising results in both small (e.g., [87]) and large models (e.g., [66]), as well as in compressing ML models without compromising the accuracy of their predictions (e.g., [73]).

The case for Graph Neural Networks. GNNs offer a powerful representation tool for MILP. They enable instance parsing with less feature engineering, as well as size and permutation invariance. Computationally, they have shown excellent performance across different tasks. Yet, we have limited understanding of the reasons behind this success. Some recent studies have uncovered some of the factors that contribute to the success of GNNs.

Chen et al. [36] study the separation and representation power of GNNs for LP. In particular, they study this in the context of three prediction tasks: predicting feasibility, boundedness, and the optimal solution vector for LP. The separation power is the

ability of GNNs to distinguish different instances, i.e., their ability to output different results when given two different instances as input. Chen et al. [36] prove that given two LPs, if no GNN[7] can distinguish them, then both LPs have the same status in terms of feasibility and boundedness. Furthermore, they both have the same minimum-l_2-norm optimal solution up to a permutation. Finally, they also show that the three tasks mentioned above can in fact be approximated using GNNs. Continuing this line of work, Qian et al. [129] prove that GNNs can be used to reproduce interior point methods. In particular, they show that there exists a GNN using $\mathcal{O}(m)$ message-passing operations (see Definition 3), with m the number of constraints, that can replicate any one iteration of the algorithm by Nocedal and Wright [119]. They show the same result for the more practical algorithm by Gondzio [70]. Notably, this is true when representing the LP using a modified version of the aforementioned bipartite graph representation (see Figure 2.3), where a new node is added and connected to all variable and constraint nodes. This global node adds alternative routes of communication among constraint and variable nodes and is said to represent the objective function. Qian et al. [129] also provide a computational comparison of different GNN implementations, i.e., different `comb` and `aggr` functions (see Eq. 1.11). This is also an active area of research, with recent works advocating for the so-called graph attention networks (e.g., [104, 138]) and other sophisticated architectures and training methods.

These results shed some light on the representation power of GNNs for MILP and strengthen the case for using them in the context of optimization problems.

2.3. DATASETS AND SOFTWARE

2.3.1. DATASETS

The modeling power of MILP makes it a suitable language for a large range of applications. With the goal of measuring the performance of different algorithms, the MILP research community has curated large benchmarks, such as MIPLIB [68], that provide a heterogeneous set of instances coming from diverse applications. It should be noted that these benchmarks are considered large for MILP standards, but are orders of magnitude smaller than typical ML benchmarks. In the light of the methods surveyed in Section 2.1, which combine an ML component with classical optimization, there is a renewed need for collections of MILP instances. In this section we provide an overview of the collections that have been used in the growing body of literature. We restrict our discussion to benchmarks that are publicly available or whose generation code is easily accessible.

ML methodologies usually require vast amounts of data. For this reason, it is common to resort to instance generators, which complement the existing instance collections. Tables 2.8 and 2.9 provide a summary of both commonly used instance collections (with their size specification) and instance generators.

Apart from the size, there is the consideration of the composition of instances. When implementing a learning-augmented solver component a specification needs to be made regarding the instances of interest. In machine learning terms, we typically talk of an *instance distribution*, where the instances that are outside the scope of interest are as-

[7] The authors consider the family of functions defined in Equation 1.11, where the combination and aggregation functions are feed-forward neural networks.

signed a zero probability of occurring. For some applications, it can be assumed that the representative instances have a shared combinatorial structure. The machine learning model is then expected to specialize to this structure. Conversely, some approaches are designed to detect patterns across instances of any class. Throughout Section 2.1 we have surveyed examples of both situations.

Table 2.10 summarizes which of the discussed approaches uses a collection of instances with mixed structures (mixed), and which use the assumption that all instances belong to the same class (homogeneous). From this we can observe that configuration decisions are more naturally framed over mixed instance collections than other tasks, like the more complex matter of branching, where some instance specialization seems valuable. Table 2.11 further shows an overview of the homogeneous datasets used in the work presented in Section 2.1. We can observe a pattern that highlights differences in the instances, based on which task is more challenging. Instances like GISP or FCMNF are more commonly chosen as a challenging test bed for primal heuristics, indicating that for these problems the difficulty lies in finding (optimal) solutions. On the contrary, branching is usually tested on instances where proving optimality is the main challenge, such as the ones provided by Ecole.

2.3.2. SOFTWARE

In connection to instance generators, there has been increasing interest in developing libraries that help the process of data generation, training and testing in the context of ML-augmented MILP solving. Some examples of these are the library Ecole [128], or the more recent MIPLearn [130]. Their goal is to provide a standardized platform for the research community for fast prototyping and testing by removing the barrier of challenging software implementation. These libraries provide ways to easily implement learning tasks, such as branching or warm-starting. For an up-to-date specification of the features they provide we refer to their documentation.

We end the section by remarking that developing software of this type is far from trivial and present some structural challenges. Depending on the problem representation (see Section 2.2), one needs to provide an increasingly tight interface with the MILP solver in order to collect the required data. This is both in the case of offline and (even more so) online learning. For example, in the case of Ecole, the development in strict correlation with SCIP has been instrumental. Clearly, in the case one wants to use a commercial solver, a sufficiently tight interface and integration might be impossible to reach, thus limiting the type of methods one can implement. We extend the discussion on the challenges in the following section.

2.4. CONCLUSIONS, PERSPECTIVE AND CHALLENGES

The work covered in this chapter testifies to the growing interest in the integration of ML methodologies within MILP solvers. This is an emerging technology that has already fostered remarkable success within its short history,[8] and is likely to play a key role in future algorithmic developments. Beyond the discussion of the literature, we have highlighted

[8] This is the case of methods that have been included in commercial (e.g., [31, 24]) or non-commercial (e.g., [80, 13]) MILP solvers.

some methodological trends and characterised the common grounds with respect to instance representation, learning algorithms and benchmarking.

Meaningful steps forward have been taken in answering the more pressing research questions. For example, the literature shows that some learning tasks seem to be formulated more naturally than others over heterogeneous instance collections. In other cases, an argument can be made in favour of the applicability of specializing to a certain structure, which makes the learning task easier. Studies like the ones in Zarpellon et al. [156] and Fischetti et al. [60] indicate that instance representations that describe the global solution process allow to more easily recognize patters across different combinatorial structures. This is especially promising because a key challenge for the integration of ML-augmented methods into MILP solvers is their generalization properties, at least as long as solvers are conceived as one-configuration-fits-all software.

It is also interesting to note the various efforts to define efficient success metrics for the different learning tasks, such as imitation targets or reward functions. Solving instances to optimality is to be avoided because of the computational effort, but performance proxies that substitute solving time must be carefully chosen.

Already substantial progress has also been made in creating the right environment for easily implementing and testing ML models for MILP solving. Existing software infrastructure includes, for example, curated instance generators, code that simplifies the solver interface and standardized testing procedures. This further helps in evaluating and comparing the different methodologies. Other efforts to bring the research community together are competitions, like ML4CO [67], which encourage progress in well-defined tasks as well as fair comparisons among the proposed methods. A significant challenge resides on the software versus hardware side: many learning methods, e.g., those relying on neural networks, especially benefit from the use of GPUs, while MILP technology is inherently CPU based. The CPU versus GPU interaction is currently a relevant obstacle for ML-augmented MILP.

Overall, a key trend seems to be building more dynamic solving strategies. MILP solvers generate a plethora of statistics during execution that often go unused. This is fertile ground for learning algorithms, which can unlock more dynamic solvers that automatically adapt the solving strategy based on prescriptions derived from such solving statistics. This poses exciting new questions and challenges.

Finally, we remark that the phase the field is entering now is that of a more systematic transfer of the proofs of concept discussed in this survey to the MILP software. The successful transfers reviewed in this chapter – and recalled at the beginning of this section – give solid evidence but do not define yet a mature path for such a transfer to happen because of the challenges above. This is the next, in our opinion achievable, step to make.

Table 2.7: Three key approaches to variable descriptors. Features marked as 'implied' do not appear explicitly as a variable descriptor, but can be inferred from the bipartite graph representation.

	Khalil et al.	Gasse et al.	Zarpellon et al.
Basic			
objective coefficient	✔	✔	
upper/lower bound		✔	
type		✔	
Structural			
# of constraints the variable is in	✔	implied	
min and max ratios a_{ij} to b_i	✔	implied	
min and max ratios a_{ij} to $\sum_k a_{kj}$	✔	implied	
stats for constraint degrees	✔	implied	
stats for constraint coefficients	✔	implied	
stats for active constraints	✔	implied	
LP solution			
LP basis status		✔	
LP sol value		✔	✔
LP sol at bound		✔	
LP sol fractionality	✔	✔	
LP sol reduced cost		✔	
LP sol age		✔	
Root LP sol value		✔	
Incumbent			
incumbent value		✔	
average incumbent value		✔	✔
Tree statistics			
average branching depth			✔
conflict score			✔
conflict length score			✔
average conflict length			✔
pseudocost score	✔		✔
pseudocost count stats			✔
average inference score			✔
average number of inferences			✔
average cutoff score			✔
average cutoffs of variable	✔		✔
# of implications derived			✔
# of cliques of variable			✔

Table 2.8: A list of some of the most relevant instance collections

Benchmark	Composition	Size	Source	URL
MIPLIB 2017	Mixed	1065	[68]	
Cor@l	Mixed	364	-	
NN verification	Homogeneous	3692	[117]	
ML4CO_1	Homogeneous	10,000	[67]	
ML4CO_2	Homogeneous	10,000	[67]	
ML4CO_3	Homogeneous	118	[67]	

Table 2.9: A list of some of the most relevant instance generators

Benchmark	Problem type(s)	Source
Tang et al.	Max-cut, Planning, Packing, Bin packing	[144]
Ecole	Set cover, Combinatorial auctions, Maximum independent set, Capacitated facility location	[128]
MIPLearn	Bin packing, Multi-dimensional knapsack, Capacitated p-median, Set cover, Set packing, Stable set, Traveling salesman, Unit commitment, Vertex cover	[130]
GISP	Generalized independent set	[39]
FCMNF	Capacitated fixed-charge network flow	[81]

Table 2.10: Classification of the literature based on whether they use a mixed or an homogeneous instance collection (or both).

	Mixed	Homogeneous
Ding et al. [51]		✓
Nair et al. [117]	✓	✓
Khalil et al. [89]		✓
Song et al. [141]		✓
Wu et al. [154]		✓
Sonnerat et al. [142]	✓	✓
Liu et al. [106]	✓	✓
Huang et al. [82]		✓
Khalil et al. [88]	✓	✓
Chmiela et al. [37]		✓
Hendel [78]	✓	

Primal heuristics

	Mixed	Homogeneous
Khalil et al. [87]	✓	
Gasse et al. [66]		✓
Gupta et al. [73]		✓
Etheve et al. [55]		✓
Nair et al. [117]	✓	✓
Zarpellon et al. [156]	✓	
Gupta et al. [74]		✓
Scavuzzo et al. [135]		✓

Branching

	Mixed	Homogeneous
Tang et al. [144]		✓
Paulus et al. [126]		✓
Wang et al. [149]	✓	✓
Turner et al. [145]	✓	
Li et al. [103]	✓	✓

Cut selection

	Mixed	Homogeneous
He et al. [77]		✓
Yilmaz and Yorke-Smith [155]		✓
Labassi et al. [96]		✓

Node selection

	Mixed	Homogeneous
Kruber et al. [95]	✓	
Hendel et al. [79]	✓	
Berthold and Hendel [24]	✓	
Berthold et al. [26]	✓	
Turner et al. [145]	✓	

Configuration decisions

Table 2.11: Common homogeneous instance collections and where they are used.

	Ecole	NNv	GISP	FCMNF	Tang et al.	Other
Ding et al. [51]						✓
Nair et al. [117]		✓				✓
Khalil et al. [89]			✓	✓		
Song et al. [141]						✓
Wu et al. [154]						✓
Sonnerat et al. [142]		✓				✓
Liu et al. [106]	✓		✓			
Huang et al. [82]	✓					✓
Khalil et al. [88]			✓			
Chmiela et al. [37]			✓			

Primal heuristics

	Ecole	NNv	GISP	FCMNF	Tang et al.	Other
Gasse et al. [66]	✓					
Gupta et al. [73]	✓					
Etheve et al. [55]						✓
Nair et al. [117]		✓				✓
Gupta et al. [74]	✓					
Scavuzzo et al. [135]	✓					

Branching

	Ecole	NNv	GISP	FCMNF	Tang et al.	Other
Tang et al. [144]					✓	
Paulus et al. [126]		✓			✓	
Wang et al. [149]	✓					✓
Li et al. [103]	✓	✓			✓	✓

Cut selection

	Ecole	NNv	GISP	FCMNF	Tang et al.	Other
He et al. [77]						✓
Yilmaz et al. [155]	✓					
Labassi et al. [96]			✓	✓		✓

Node selection

3

EXPERT-FREE LEARNING TO BRANCH

Branching is the core mechanism on which the B&B algorithm operates. There is a choice to be made with respect to which disjunction is used for branching. The standard is to use single variable disjunctions such as the ones in Eq. 1.4. General disjunctions for branching will be the topic of Chapter 4. Still, the solution to the LP relaxation is likely to violate more than one integrality constraint. This means that there are several candidate variables that can be used for branching.

The branching strategy, i.e., the rule that selects among branching candidates, has a critical impact on the efficiency of the B&B algorithm. In fact, Achterberg and Wunderling [7] point to branching rules as one of the most important components of an MILP solver. For this reason, this topic has been heavily studied in the literature. However, given the complex nature of the branching process, analytical studies of branching work only under simplified models (see, e.g., [98]). In practice, branching strategies are tested computationally.

In Chapter 2 we offered a high-level overview of the topic of branching strategies and the different approaches to learning to branch. In the present chapter, we dive deeper into the topic, and discuss in detail our methodology to learning to branch without expert demonstrations. We start with an overview of classical branching rules, followed by a discussion of learning paradigms applied to branching. We present the MDP framework in more detail, to then introduce our tree MDP formulation in contrast. This chapter presents our theoretical and computational contributions and highlights the great potential of tree MDP for branching and beyond. The contents of this chapter are based on our published work [135]. The code for reproducing all experiments is available online [131].

3.1. CLASSICAL BRANCHING RULES

As introduced in Section 1.3.2, branching consists in partitioning the search space into smaller subproblems. For the purpose of this chapter we will consider the most standard approach to branching: single-variable disjunctions that create two subproblems. That is, for some $j \in \mathcal{I}$ such that $x_j^{LP} \notin \mathbb{Z}$ we impose

$$x_j \leq \lfloor x_j^{LP} \rfloor \qquad \text{or} \qquad x_j \geq \lceil x_j^{LP} \rceil . \tag{3.1}$$

This creates the left child and the right child respectively. The variable selection step, i.e., the selection of $j \in \mathcal{I}$, is of utmost importance. As a result of the computational studies, we know that certain metrics tend to be good indicators of high-quality branching decisions. The prime example is LP bound degradation.

Definition 4 (Left/right LP bound degradation)**.** *Consider the LP bound of the current node z^{LP} and assume variable $j \in \mathcal{I}$ takes a fractional value in the LP solution, which means it is a branching candidate. If we branch on j by imposing (3.1), two nodes are created with associated LP bounds z^{LP+} and z^{LP-}, respectively. The left and right LP bound degradation associated with branching on j at the current node are respectively defined as*

$$\Delta^- = z^{LP-} - z^{LP} \quad \textit{and} \quad \Delta^+ = z^{LP+} - z^{LP}$$

Intuitively, a good branching candidate is one that proves a better global lower bound. This is, the one that maximizes $\min\{z^{LP-}, z^{LP+}\}$. It is advantageous to make improvements on both sides. Considering this, variables are typically scored using a function that combines Δ^- and Δ^+. For example, SCIP v.7.0.0 [64] uses the following scoring function by default

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

with ϵ a small number.

STRONG BRANCHING

LP bound degradation is a popular proxy for branching potential, but this metric comes with a caveat. At the time of making the branching decision, the LP bounds of all the potential children are not known. This can be remedied in two ways. The first one is to explicitly compute these bounds for all candidates. This strategy is known as *strong branching* and requires solving two LPs per candidate. Undoubtedly, this comes at great computational cost. Yet, the gathered information can be used for more purposes. For example, tightening the LP relaxation whenever infeasible children are found. The alternative to explicit calculation is estimating the bound degradation using past data, a strategy called *pseudocost branching*.

PSEUDOCOST BRANCHING

Pseudocost branching relies on the fact that the values Δ^+ and Δ^- can be obtained retrospectively at no extra cost once that branching has taken place and the children nodes

have been processed. Every time this happens, the solver records

$$P_j^- = \frac{\Delta^-}{x_j^{LP} - \lfloor x_j^{LP} \rfloor} \quad \text{and} \quad P_j^+ = \frac{\Delta^+}{\lceil x_j^{LP} \rceil - x_j^{LP}}$$

where j is the variable used for branching. The recorded values of P_j^- and P_j^+ obtained throughout the search are averaged to obtain $\bar{P}_j^-$ and $\bar{P}_j^+$. The latter are known as the *pseudocosts* of variable j. Pseudocosts can be used to calculate an estimate of the LP bound degradation on the next branching where j is a candidate. In particular, the estimates are

$$\bar{\Delta}^- = \bar{P}_j^- \cdot (x_j^{LP} - \lfloor x_j^{LP} \rfloor) \quad \text{and} \quad \bar{\Delta}^+ = \bar{P}_j^+ (\lceil x_j^{LP} \rceil - x_j^{LP})$$

RELIABILITY BRANCHING

The strong branching rule has been shown (computationally) to produce very small trees compared to other rules [7]. While the quality of the decisions seems to be very high, the cost of making each of these decisions outweighs the benefits of a smaller tree. For this reason, strong branching is typically discarded as a viable option in practice. On the other hand, the pseudocost estimation strategy has proven effective in predicting good branching candidates, while maintaining a low cost of deciding. However, this strategy relies on data collected throughout the tree, which poses a problem at the beginning of the search. It is in fact the first branching decisions that have the most impact on the solution process, which is unfortunate, as it is precisely at this moment that the estimates are the least reliable. This is the motivation behind the *reliability branching* rule [8], which smartly combines strong and pseudocost branching. The idea is to apply the explicit calculation of strong branching until enough samples of $P_j^\pm$ are obtained, for each variable $j \in \mathcal{I}$. Once a variable is associated with enough samples, the pseudocost $\bar{P}_j^\pm$ is deemed reliable and henceforth trusted to make branching decisions.

SCIP'S DEFAULT BRANCHING RULE

SCIP v.7.0.0 [64] assigns maximum priority by default to the so-called hybrid branching rule [6]. This means that strong branching is used for initialization, but further, the choice of branching variable is based on a weighted sum of different criteria. The largest weight is placed on the variable's pseudocosts. Other than pseudocosts, SCIP also considers information about the implied reductions of other variables' domains and conflicts where the variable is involved, though with smaller importance. It is important to consider that a call to strong branching can trigger a series of side-effects within the solver that are not accounted for in the node count. This was first observed by Gamrath and Schubert [62], who point out that this gives an unfair advantage to methods that use strong branching when comparing branching rules according to the final tree size.

3.2. LEARNING TO BRANCH

Approaches that frame branching as a learning problem have gained significant attention recently. Section 2.1.2 presented a discussion of the literature in this field. As seen in Table 2.3 of that section, it is common to frame learning to branch using either imitation learning (IL) or reinforcement learning (RL). Both paradigms, introduced in Section

1.4, are formalized using the MDP formulation (see Figure 1.1). In this section, we start by presenting the necessary notation to formulate the branching task as an MDP. We then direct our attention towards the two paradigms of interest, i.e., imitation and reinforcement learning, and point out some trends and key challenges with respect to this distinction.

LEARNING TO BRANCH AS A MARKOV DECISION PROCESS

The methodology that will be discussed in this chapter requires a formal model of branching based on Markov Decision Processes (MDPs). We start by extending the brief definion of Chapter 1. We consider the episodic case, of the form $M = (\mathcal{S}, \mathcal{A}, p_{init}, p_{trans}, r)$, with states $s \in \mathcal{S}$, actions $a \in \mathcal{A}$, initial state distribution $p_{init}(s_0)$, state transition distribution $p_{trans}(s_{t+1}|s_t, a_t)$, and reward function $r : \mathcal{S} \to \mathbb{R}$. For simplicity, we assume finite episodes of length T, that is, $\tau = (s_0, a_0, \dots, s_T)$, $|\tau| = T$. Together with a control mechanism represented by a stochastic policy $\pi(a_t|s_t) : \mathcal{A} \times \mathcal{S} \to [0,1]$, with $\sum_{a \in \mathcal{A}} \pi(a|s) = 1$, the MDP defines a probability distribution over trajectories, namely

$$p_\pi(\tau) = p_{init}(s_0) \prod_{t=0}^{|\tau|-1} \pi(a_t|s_t)\, p_{trans}(s_{t+1}|s_t, a_t).$$

The MDP control problem is to find a policy that maximizes the expected cumulative reward, $\pi^\star \in \arg\max_\pi V^\pi$, with

$$V^\pi := \mathop{\mathbb{E}}_{\tau \sim p_\pi} \left[\sum_{t=0}^{|\tau|} r(s_t) \right]. \tag{3.2}$$

Here the subscript $\tau \sim p_\pi$ denotes that the expected value is computed with respect to trajectories τ which are drawn from the probability distribution p_π. The policy is in fact a parameterized function $\pi_\theta(a_t|s_t)$ with parameters $\boldsymbol{\theta}$. With a slight abuse of notation and for ease of reading, we omit the dependence on $\boldsymbol{\theta}$.

A key property of MDPs is the Markov property: future states are independent from states visited in the past conditional to the current state or action. We denote this $S_{t+1} \perp\!\!\!\perp S_{<t} \mid S_t, A_t, \forall t$. One consequence of this property is the policy gradient theorem , which forms the basis of policy gradient methods.

Theorem 1 (Policy gradient [143])**.** *Let the expected cumulative reward V^π be defined as in 3.2. Then,*

$$\nabla_\theta V^\pi \propto \mathop{\mathbb{E}}_{\tau \sim p_\pi} \left[\sum_{t=0}^{|\tau|-1} \nabla_\theta \log \pi(a_t|s_t) \sum_{t'=t+1}^{|\tau|} r(s_{t'}) \right].$$

The policy gradient theorem is key to optimizing V^π. Even in the case where p_π is unknown (because p_{init} and p_{trans} are unknown) one can compute an approximation of the gradient by sampling from this probability and then using gradient decent to optimize over the policy space.

Let us now consider the problem of learning a branching policy in a B&B solver. As noted by He et al. [77] and Gasse et al. [66], branching decisions are made sequentially, thus the problem can naturally be regarded as an MDP. In this form, the states s_t consist of the entire state of the B&B process (that is, the solver) at time t, which includes the whole tree structure, all sub-MILPs and LP solutions, and all upper and lower bounds. The actions a_t are the branching decisions, and the reward is chosen so that the return matches an objective function of interest (e.g., total solving time or the final B&B tree size). In practice however, the MDP states are complex objects of growing size, which are impractical to handle. For this reason, alternative representations are used (see Section 2.2) which turn the problem into a partially observable MDP.

LEARNING TO BRANCH WITH IMITATION LEARNING

The literature shows that successful branching rules can be obtained by learning fast approximations of strong branching [87, 112, 76, 66, 73, 156, 117]. This approach can be framed as imitation learning. While this can lead to improvements over state-of-the-art branching rules on various benchmarks, it also has several drawbacks.

First, strong branching implementations are known to trigger side effects (such as early detection of an infeasible child) that do not map well within the branching MDP framework [62]. This means that branching decisions collected from the strong branching expert might not line up with the environment of the learning agent, which might result in a performance gap between the learning agent and the expert, even if the agent manages to successfully reproduce the expert decisions.

Second, other than these side-effects, strong branching relies on dual bound changes to make branching decisions. This can be ineffective in problems where the LP relaxation is not very informative or suffers from dual degeneracy, as pointed out in Gamrath et al. [65]. As an extreme case, Dey et al. [47] provide a case where a strong-branching based B&B tree can have exponentially more nodes than the tree obtained with a problem-specific rule. This exemplifies that strong branching is not actually an expert. The performance ceiling that this “expert" sets on the learning agent can therefore be problematic.

Finally, regardless of the expert quality, obtaining strong branching samples can become prohibitively expensive for large instances. This means that non-trivial engineering solutions and scaling strategies are needed to allow training on larger problems, such as heavy computational parallelization [117].

LEARNING TO BRANCH WITH REINFORCEMENT LEARNING

The reasons listed above suggest the need for an alternative approach to finding a good branching policy; an approach that does not rely on the strong branching rule. One could perform imitation learning on another expert, but no other plausible imitation target is known, and the same performance ceiling issue would remain. A natural alternative is reinforcement learning, but despite its theoretical appeal for learning to branch, it also comes with its own challenges.

First, common evaluation metrics in B&B, such as solving time or final tree size, are inconvenient for RL. Both require to run episodes to completion, that is, to solve MILP instances to optimality, which leads to very long episodes even for moderately hard instances. Second, in contrast to many RL tasks studied in the literature, in B&B the worse the policy is, the longer the episodes are. These two factors combined give rise to the following problems:

1. Collecting training data is computationally expensive. In particular, training from scratch from a randomly initialized policy can be prohibitive for large MILPs.

2. Due to the length of the episodes, training signals are particularly sparse. This exacerbates the so-called *credit assignment problem* [114], i.e., the problem of determining which actions should be given credit for a certain outcome.

Etheve et al. [55] address the credit assignment problem by showing that, when depth-first-search (DFS) is used as the node selection strategy, minimization of the total B&B tree size can be achieved by taking decisions that minimize subtree size at each node. Based on this result, they propose a Q-learning-type algorithm [116] where the learned Q-function approximates the local subtree size. They report improvements over a state-of-the-art branching rule on collections of small, fixed-size instances. However, they only evaluate the learned policy with DFS node selection, which matches their training setting, but is not a realistic B&B setting. In the next section we will show that the method proposed by Etheve et al. [55] can be interpreted as a specific instantiation of a more general *tree MDP* framework, which effectively simplifies the credit assignment problem in learning to branch (point 2 above). Furthermore, we propose an alternative condition to the one of Etheve et al. [55] to ensure a tree MDP setting, which results in shorter episodes, hence reducing the cost of data collection (point 1 above).

3.3. THE TREE MDP FORMULATION

We now detail our tree MDP framework, and show how the branching problem can be cast as a tree MDP control problem, under some conditions. We then explain how this structure, when present, can be exploited for more efficient reinforcement learning. For further distinction of the classical MDP setting and tree MDPs, we sometimes refer to the former as *temporal MDPs*. Proofs are deferred to the appendix.

3.3.1. TREE MDPS

We define tree MDPs as augmented Markov Decision Processes $tM = (\mathcal{S}, \mathcal{A}, p_{init}, p^-_{ch}, p^+_{ch}, r, l)$, with states $s \in \mathcal{S}$, actions $a \in \mathcal{A}$, initial state distribution $p_{init}(s_0)$, respectively left and right child transition distributions $p^-_{ch}(s_{ch^-_i}|s_i, a_i)$ and $p^+_{ch}(s_{ch^+_i}|s_i, a_i)$[1], reward function $r : \mathcal{S} \to \mathbb{R}$ and leaf indicator $l : \mathcal{S} \to \{0, 1\}$. The central concept behind tree MDPs is that each non-leaf state s_i (i.e., each state such that $l(s_i) = 0$), together with an action a_i, produces two new states $s_{ch^-_i}$ (its left child) and $s_{ch^+_i}$ (its right child). As a result, the tree MDP generative process results in episodes τ that follow a tree structure

[1] In the case of branching, these transitions are deterministic.

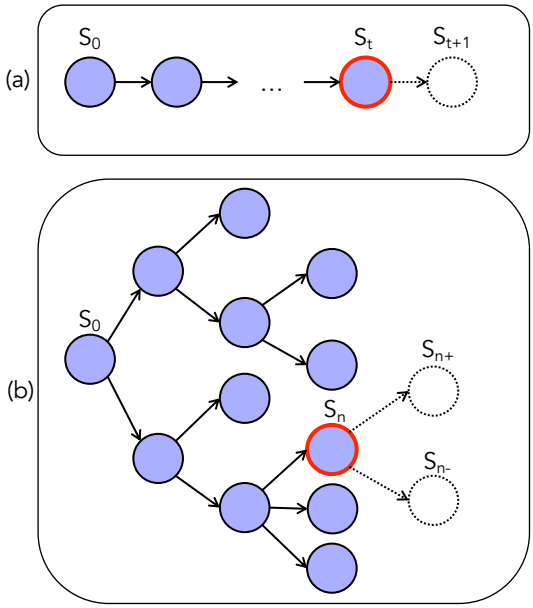


Figure 3.1: Schematic of (a) a Markov Decision Process and (b) a tree Markov Decision Process. From the current state (marked in red) we transition into a single state in the MDP and a set of states in the tree MDP.

(see Figure 3.1), where leaf states (i.e., such that $l(s_i) = 1$) are the leaf nodes of the tree, below which no action can be taken and no children state will be created. For simplicity, just like for MDPs, we assume the tree-like trajectories have some finite size that we denote $T = |\tau|$.

A tree MDP episode τ consists of a binary[2] tree with nodes $\mathcal{N} = \{0, \dots, |\tau|\}$ and leaf nodes $\mathcal{L} = \{i \in \mathcal{N} \mid l(s_i) = 1\}$, which embeds a state s_i at every node $i \in \mathcal{N}$ and an action a_i at every non-leaf node $i \in \mathcal{N} \setminus \mathcal{L}$. For convenience, in the following we will use pa_i, ch_i^- and ch_i^+ to denote the nodes that are respectively parent, left child and right child of a node i if any, as well as d_i and nd_i to denote respectively the set of all descendants and non-descendants of a node i in the tree. Together with a control mechanism $\pi(a_t|s_t)$, a tree MDP defines a probability distribution over trajectories,

$$p_\pi(\tau) = p_{init}(s_0) \prod_{i \in \mathcal{N} \setminus \mathcal{L}} \pi(a_i|s_i) p_{ch}^-(s_{ch_i^-}|s_i, a_i) p_{ch}^+(s_{ch_i^+}|s_i, a_i).$$

As in temporal MDPs, the tree MDP control problem is to find a policy that maximizes the expected cumulative reward, as defined by (3.2). Due to their specific generative process, a key characteristic of tree MDPs is the *tree Markov property*,

$$S_{ch_i^-}, S_{ch_i^+} \perp\!\!\!\perp S_{nd_i} \mid S_i, A_i \, \forall i,$$

[2] The concept can easily be extended to non-binary trees.

MDP

A (temporal) MDP process:	$(\mathscr{S}, \mathscr{A}, p_{init}, p_{trans}, r)$
Probability of trajectory τ:	$p_\pi(\tau) = p_{init}(s_0) \prod_{t=0}^{\lvert\tau\rvert-1} \pi(a_t \lvert s_t)\, p_{trans}(s_{t+1} \lvert s_t, a_t)$
Markov property:	$S_{t+1} \perp\!\!\!\perp S_{<t} \mid S_t, A_t, \forall t$

Tree MDP

A tree MDP process:	$(\mathscr{S}, \mathscr{A}, p_{init}, p^-_{ch}, p^+_{ch}, r, l)$
Probability of trajectory τ:	$p_\pi(\tau) = p_{init}(s_0) \prod_{i \in \mathscr{N} \setminus \mathscr{L}} \pi(a_i \lvert s_i) \cdot$ $\cdot p^-_{ch}(s_{ch^-_i} \lvert s_i, a_i) p^+_{ch}(s_{ch^+_i} \lvert s_i, a_i)$
Markov property:	$S_{ch^-_i}, S_{ch^+_i} \perp\!\!\!\perp S_{nd_i} \mid S_i, A_i, \forall i$

Figure 3.2: A side-by-side comparison of the MDP and tMDP models.

which guarantees, similarly to the temporal Markov property, that each subtree only depends upon the immediate state and action. This results in the following *tree policy gradient* formulation.

Proposition 1. *For any tree MDP tM, the policy gradient can be expressed as*

$$\nabla_\pi V^\pi \propto \mathop{\mathbb{E}}_{\tau \sim p_\pi} \left[\sum_{i \in \mathscr{N} \setminus \mathscr{L}} \nabla_\pi \log \pi(a_t \lvert s_t) \sum_{j \in d_i} r(s_j) \right]. \tag{3.3}$$

For a side-by-side comparison of the MDP and tree MDP formulations see Figure 3.2.

3.3.2. THE BRANCHING TREE MDP

We now show how and under which conditions the vanilla B&B algorithm can be formulated as a tree MDP. We consider episodes τ that follow exactly the B&B tree structure. Each node i in the tree embeds a state $s_i = (MILP_i, \bar{z}(i))$, where $MILP_i$ is the local sub-MILP of the node, and $\bar{z}(i)$ is the global upper bound at the time that node i is processed. Each non-leaf node also embeds an action $a_i = (j, x_j^{LP_i})$, where j is the index of the branching variable chosen by B&B, and $x_j^{LP_i}$ is the value used to branch

$$x_j \leq \lfloor x_j^{LP_i} \rfloor \vee x_j \geq \lceil x_j^{LP_i} \rceil .$$

Note that such states and actions, embedded in the B&B tree, carry enough information to unroll a vanilla B&B algorithm, as described in Section 1.3.2. We now need to make two additional assumptions in order to formulate branching as a tree MDP.

B&B TREE TRANSITIONS

First, and this is our main requirement, B&B state transitions must decompose into p^-_{ch} and p^+_{ch}.

Assumption 2. *For every non-leaf node i, the global upper bounds $\bar{z}(ch_i^-)$ and $\bar{z}(ch_i^+)$ (reached by B&B when the left and right child is processed, respectively) can be derived solely from the current state and action, (s_i, a_i).*

Proposition 2. *A vanilla B&B algorithm that satisfies Assumption 2 forms a tree MDP.*

Assumption 2 is not always satisfied, as the following counterexample shows.

Counterexample. *Consider the root problem $MILP_0 = \{\min x \, s.t. \, x \geq 0.6, x \in \mathbb{Z}\}$, with upper bound $\bar{z}(0) = \infty$. The root LP solution is $\hat{x}^\star = 0.6$, and the two sub-problems $MILP_{ch_i^-}$ and $MILP_{ch_i^+}$ follow from the (only) branching decision $x \leq 0 \vee x \geq 1$. Now, the two global upper bound $\bar{z}(ch_i^-)$ and $\bar{z}(ch_i^+)$ depend on whether the feasible solution $x = 1$ has been found in the past, which in turn depends on the node processing order. Going left first (−) will yield $(\bar{z}(ch_i^-), \bar{z}(ch_i^+)) = (\infty, \infty)$, while going right first (+) will yield $(\bar{z}(ch_i^-), \bar{z}(ch_i^+)) = (1, \infty)$.*

We now provide two conditions under which Assumption 2 is true.

Proposition 3. *In Optimal Objective Limit B&B (ObjLim B&B), that is, when the optimal solution value of the MILP is known at the start of the algorithm ($\bar{z}(0) = z^*$), Assumption 2 holds.*

Proposition 4. *In Depth-First-Search B&B (DFS B&B), that is, when nodes are processed depth-first and left-first by the algorithm, Assumption 2 holds.*

Propositions 3 and 4 provide two viable options for turning vanilla B&B into a tree MDP, where p^-_{ch} and p^+_{ch} are deterministic functions. The first variant, *ObjLim*, requires MILP instances used for training to be solved to optimality once, in order to collect their optimal objective value. The second variant, *DFS*, corresponds to the setting in Etheve et al. [55]. In this variant there is no need to pre-solve training instances to optimality, however it is expected that the collected episodes might be longer than with a standard node selection rule, which might result in slower training.

B&B TREE REWARD

Last, for branching to formulate as a control problem in a tree MDP, the objective must be compatible.

Assumption 3. *The branching objective can be decomposed over the nodes of the B&B tree, with a state-based reward function $r : \mathcal{S} \to \mathbb{R}$.*

Interestingly, a natural objective for branching is the final *B&B tree size*, which expresses naturally as $r(s_i) = -1$. Thus, it is trivially compatible with Assumption 3. We will consider this reward in our experiments. Another common objective is the total solving time, which can also be expressed as a state-based reward $r : \mathcal{S} \to \mathbb{R}$ under mild assumptions. Indeed, it suffices to consider that solving LP relaxations and making branching decisions at each node are the main contributing factors in the total running time, while other algorithmic components have a negligible cost. In vanilla B&B, both these components only depend on the local state $s_i = (MILP_i, \bar{z}(i))$ of each node.

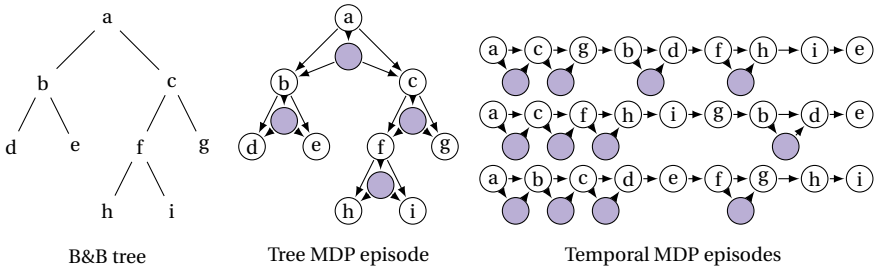


Figure 3.3: B&B process as a tree MDP episode vs. three possible temporal MDP episode. Which temporal MDP rollout is actually realized depends on the node processing order. White nodes denote states, green nodes denote actions. In the tree MDP framework, the branching decision for splitting a node f is credited two rewards, (r_h, r_i). In the temporal MDP framework, the same branching decision is credited with additional rewards which depend on the temporal order in which B&B nodes are processed, (r_h, r_i, r_e), $(r_h, r_i, r_g, r_b, r_d, r_e)$, or (r_g, r_h, r_i).

3.3.3. EFFICIENCY OF TREE MDP

Tree MDPs, when applicable, provide a convenient alternative to temporal MDPs for tackling the branching problem. First, the tree Markov property implies that branching policies in tree branching MDPs will not benefit from any information other than the local state $s_i = (MILP_i, \bar{z}(i))$ to make optimal decisions, similarly to how control policies in temporal MDP can ignore past states and rely only on the immediate state s_t. Second, the credit assignment problem in the branching tree MDP is more efficient than in the equivalent temporal MDP. This is showcased in Figure 3.3, and stems from the fact that in a tree MDP all the descendants of a node i are necessarily processed after that node temporally. As a consequence, the rewards credited to an action in the tree policy gradient (3.3), $\sum_{j \in d_i} r(s_j)$, are necessarily a subset of the rewards credited to the same action in the temporal policy gradient (Theorem 1), $\sum_{t'>t} r(s_{t'})$. Thus, it can be expected intuitively that learning branching policies within the tree MDP framework will be easier, and more sample-efficient than learning within the temporal MDP framework. We will validate this hypothesis experimentally in Section 3.4.

3.3.4. THEORETICAL LIMITATIONS

Our proposed B&B variants, *ObjLim* and *DFS*, allow for a nice formulation of the branching problem as a tree MDP, which we argue is key to unlocking a more practical and sample-efficient learning of branching policies. However, usually the end goal is to learn a branching policy that performs well in realistic B&B settings, and the fact that a branching policy performs well in one of those variants does not guarantee that it will perform well in the vanilla setting also. This discrepancy between the training environment and the evaluation environment is a recurring problem in RL, and is more generally referred to as the transfer learning problem. While there exist solutions to mitigate this problem, in this work we leave the question aside and simply assume that the transfer problem is negligible. We thus directly report the performance obtained from each training setting in the realistic evaluation setting, a default B&B solver.

3.3.5. CONNECTIONS WITH HIERARCHICAL RL

The tree MDP formulation has connections with hierarchical RL (HRL), a paradigm that aims at decomposing the learning task into a set of simpler tasks that can be solved recursively, independently of the parent task. The most related HRL approach is perhaps MAXQ [49], which decomposes the value function of an MDP recursively using a finite set of smaller constituent MDPs, each with its own action set and reward function. For example, delivering a package from a point A to a point B decomposes into: moving to A, picking up package, moving to B, dropping package. While both tree MDP and MAXQ exploit a recursive tree decomposition in order to simplify the credit assignment problem, the two frameworks also differ on several points. First, in MAXQ the hierarchical sub-task structure must be known a priori for each new task, and results in a fixed, limited tree depth, while in tree MDPs the decomposition holds by construction and can result in virtually infinite depths. Second, in MAXQ each sub-task results in a different MDP, while in tree MDPs all sub-tasks are the same. Lastly, in MAXQ the recursive decomposition must follow a temporal abstraction, where each episode is processed according to a depth-first traversal of the tree. In tree MDPs the decomposition is not tied to the temporal processing order of the episode, except for the requirement that a parent must be processed before its children. Thus, any tree traversal order is allowed (see Figure 3.2).

3.4. EXPERIMENTAL VALIDATION

We now compare the performance of four machine learning approaches: the imitation learning method of Gasse et al. [66], and three different RL methods. We also compare against the default rule of SCIP v.7.0 (see Section 3.1). Code for reproducing all experiments is available online [131].

3.4.1. SETUP

BENCHMARKS

Similarly to Gasse et al. [66], we train and evaluate each method on five NP-hard problem benchmarks, which consist of synthetic combinatorial auctions, set covering, maximum independent set, capacitated facility location and multiple knapsack instances. For each benchmark we generate a training set of 10,000 instances, along with a small set of 20 validation instances for tracking the RL performance during training. For the final evaluation, we further generate a test set of 40 instances, the same size as the training ones, and also a transfer set of 40 instances, larger and more challenging than the training ones. More information about benchmarks and instance sizes can be found in the appendix (3.6.3).

TRAINING

We use the Graph Neural Network (GNN) from Gasse et al. [66] (see Section 2.2 for a detailed description). We compare four training methods: imitation learning from strong branching (IL); RL using temporal policy gradients (MDP); RL using tree policy gradients with DFS as a node selection strategy (tMDP+DFS), which enforces the tree Markov

[3] Notice that for tree MDPs the computation of the return for each node can be computed efficiently with a bottom-up traversal that runs in $O(n)$.

Algorithm 2 REINFORCE training loop

Input: training set of MILP instances and their pre-computed optimal solution $\mathscr{D}$, maximum number of epochs K, time limit ζ, entropy bonus λ, learning rate α, sample rate β.
Initialize policy π_θ with random parameters θ.
for *epoch* from 1 to K **do**
  if elapsed time $> \zeta$ **then** break
  Sample 10 MILP instances from $\mathscr{D}$
  for each sampled instance **do**
    Collect one episode τ by running B&B to optimality
    Extract randomly $\beta \times |\tau|$ state, action, return tuples (s, a, G) from τ (with G the local subtree size for tree MDPs, and the remaining episode size for MDPs)[3]
  end for
  $n \leftarrow$ number of collected tuples, $L \leftarrow 0$
  for each collected tuple (s, a, G) **do**
    $L \leftarrow L - G\frac{1}{n}\log \pi_\theta(a|s)$ # policy gradient cost
    $L \leftarrow L - \lambda\frac{1}{n}H(\pi_\theta(\cdot|s))$ # entropy bonus
  end for
  $\theta \leftarrow \theta - \alpha\nabla_\theta L$
end for
return π_θ

property due to Proposition 4; and RL using tree policy gradients with the optimal objective value set as an objective limit (tMDP+ObjLim), which corresponds to Propositions 3. Other than that, we use default solver parameters, except for restarts and cutting planes after the root node which are deactivated. We use a plain REINFORCE [151] with entropy bonus as our RL algorithm, for simplicity. Our training procedure is summarized in Algorithm 2. We set a maximum of 15,000 epochs and a time limit of six days for training. Our implementation uses PyTorch [124] together with PyTorch Geometric [56], and Ecole [128] for interfacing to the solver SCIP [64]. All experiments are run on compute nodes equipped with a GPU.

EVALUATION

For each branching rule evaluated, we solve every validation, test or transfer instance 5 times with a different random seed. We use default solver parameters, except for restarts and cutting planes after the root node which are deactivated (same as during training), and a time limit of 1 hour for each solving run. For tMDP+DFS and tMDP+ObjLim, the specific settings used during training (DFS node selection and optimal objective limit, respectively) are not used any more, thus providing a realistic evaluation setting. We report the geometric mean of the final B&B tree size as our metric of interest, as is common practice in the MILP literature [7]. We pair this with the average per-instance standard deviation (in percentage). We only consider solving runs that finished successfully for all methods, as in [66]. Extended results including solving times are provided in the appendix (3.6.2).

Table 3.1: Solving performance of the different branching rules in terms of the final B&B tree size (lower is better). We evaluate each method on a test set with instances the same size as training, and a transfer set with larger instances. We report the geometric mean and standard deviation over 40 instances, solved 5 times with different random seeds, and we bold the best of the RL methods.

Model	Comb. Auct.	Set Cover	Max.Ind.Set	Facility Loc.	Mult. Knap.
SCIP default	7.3±39%	10.7±24%	19.3±52%	203.6±63%	267.8±96%
IL	52.2±13%	51.8±10%	35.9±36%	247.5±39%	228.0±95%
RL (MDP)	86.7±16%	196.3±20%	91.8±56%	393.2±47%	143.4±76%
RL (tMDP+DFS)	**86.1±17%**	**190.8±20%**	89.8±51%	360.4±46%	**135.8±75%**
RL (tMDP+ObjLim)	87.0±18%	193.5±23%	**85.4±53%**	**325.4±41%**	142.4±78%

Test

Model	Comb. Auct.	Set Cover	Max.Ind.Set	Facility Loc.	Mult. Knap.
SCIP default	733.9±26%	61.4±19%	2867.1±35%	344.3±57%	592.3±75%
IL	805.1±9%	145.0±6%	1774.8±38%	407.8±37%	1066.1±101%
RL (MDP)	1906.3±18%	853.3±27%	2768.5±76%	679.4±52%	518.4±79%
RL (tMDP+DFS)	**1804.6±17%**	**816.8±25%**	2970.0±76%	609.1±47%	495.1±81%
RL (tMDP+ObjLim)	1841.9±18%	826.4±26%	**2763.6±74%**	**496.0±48%**	**425.3±64%**

Transfer

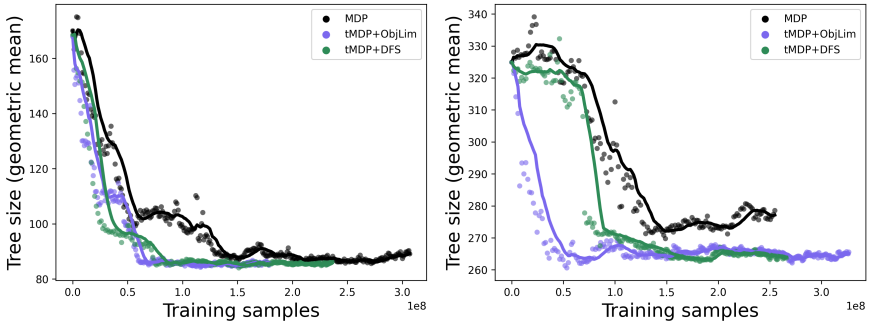
3.4.2. RESULTS

Figure 3.4 showcases the convergence of our three RL paradigms MDP, tMDP+ObjLim and tMDP+DFS during training, in terms of the final B&B tree size on the validation set (the lower the better). In order to better highlight the sample efficiency of each method, we report on the x-axis the cumulative number of collected training samples, which correlates with the length of the episodes collected during training. This provides a hardware-independent proxy for training time. As can be seen, the tree MDP paradigm clearly improves the convergence speed on these three benchmarks, with a clear domination of tMDP+ObjLim on set covering and capacitated facility location.

Table 3.1 reports the final performance of the branching rules obtained with each method, on both a held-out test set (same instance difficulty as training) and a transfer set (larger, more difficult instances than training). Despite a mismatch between the training and evaluation environments, which is required to enforce the tree Markov property, the tree MDP paradigm consistently produces equal or better branching rules than the temporal MDP paradigm on all five benchmarks.

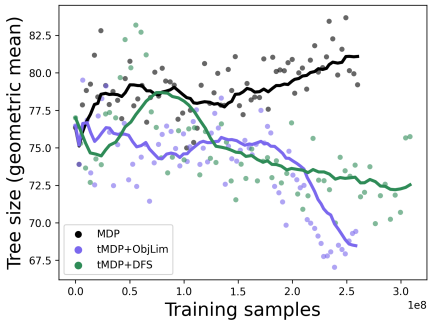
On one benchmark, multiple knapsack, the branching rules learned by RL outperform both SCIP's default branching rule and the strong branching imitation (IL) approach. The likely reason is that the MILP formulation of multiple knapsack provides a very poor linear relaxation, which often results in no dual bound change after branching. This means that strong branching scores are in most cases not discriminative, which is problematic for rules that heavily rely on this criterion (see Section 3.2), such as SCIP's default or a policy that imitates strong branching. This situation makes a strong case for the potential of RL-based methods, which can adapt and devise alternative branching strategies.

On the remaining four benchmarks, however, RL methods perform worse than IL, de-

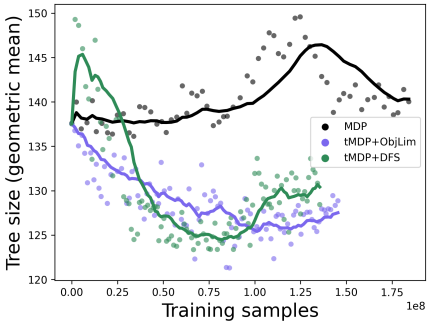


(a) Combinatorial Auctions

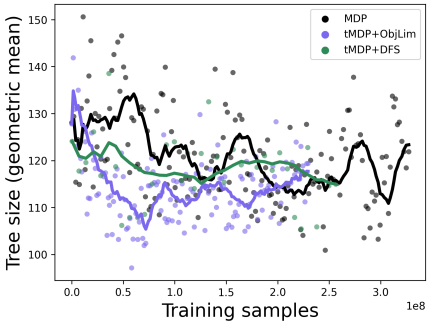
(b) Set Covering



(c) Maximum Independent Set



(d) Capacitated Facility Location



(e) Multiple Knapsack

Figure 3.4: Training curves for REINFORCE with temporal policy gradients (MDP), tree policy gradients with objective limit (tMDP+ObjLim) and DFS node selection (tMDP+DFS). We report the final B&B tree size on the validation set (geometric mean over 20 instances × 5 seeds, the lower the better), versus the number of processed training samples on the x-axis. Solid lines show the moving average.

spite being based on the same GNN architecture. This illustrates the difficulty of learning to branch via RL, even on small-scale problems, and the remaining room for improvement. We would like to reiterate that SCIP's default rule uses some iterations of strong branching (see Section 3.1) and for this reason the tree size might appear small while the solving time is larger than for the GNN-based methods. Additional evaluation criteria

(solving times and number of time limits) are available in the appendix (3.6.2).

3.5. CONCLUSIONS AND FUTURE DIRECTIONS

The work discussed in this chapter adds to a growing body of literature on using ML to assist decision-making in several key components of the B&B algorithm (see also Chapter 2). We contribute to the study of RL as a tool for learning to branch in MILP solvers. We present for the first time the tree MDP, a variant of Markov Decision Processes, and show that under some conditions, the B&B branching process is tree-Markovian. We show that the approach of Etheve et al. [55] can be naturally cast as Q-learning for tree MDPs, and we propose an alternative, more computationally appealing way to enforce the tree Markov property in B&B, using optimal objective limits. Finally, we evaluate for the first time a variety of RL-based branching rules in a comprehensive computational study, and we show that tree MDPs improve the convergence speed of RL for branching, as well as the overall performance of the learnt branching rules.

These contributions bring us closer to learning efficient branching rules from scratch using RL, which could ultimately outperform existing branching heuristics built upon decades of expert knowledge and experiment. However, despite the convergence speed-up that our method provides, training without expert knowledge remains very computationally heavy and in general still results in worse performance than its supervised learning counterpart, which reveals a significant gap that must be closed. Future work includes exploring ideas to keep improving sample efficiency, and generalization across instance size. This is necessary for RL to scale to larger, non-homogeneous benchmarks, such as MIPLIB [68], which at the moment remain out-of-reach for RL.

Finally, we would like to note that, although this chapter focuses on improving variable selection for MILP, the tree MILP construction can be useful in other applications. Branch-and-bound is a type of divide-and-conquer algorithm, and, in general, this framework can be applied to any problem where one seeks to control such algorithms more efficiently. Recently, Cameron et al. [33] apply the tree MDP formulation to proving unsatisfiablity of a SAT formula. They build upon our proof of the policy gradient theorem for tree MDP by presenting an adaptation of Monte Carlo Tree Search. Their results show further indication that the tree MDP formulation is highly advantageous in problems that decompose with a tree structure.

3.6. APPENDIX

3.6.1. PROOFS

Proof of Proposition 1. This proof draws closely to the proof of the temporal policy gradient theorem. First, let us re-write (3.2) as

$$V^\pi = \mathbb{E}_{s_0 \sim p_\pi}\left[V^\pi(s_0)\right],$$

where

$$\begin{aligned} V^\pi(s_i) &:= r(s_i) && \text{if } l(s_i) = 1 \text{ (leaf node), and} \\ V^\pi(s_i) &:= r(s_i) + \mathbb{E}_{a_i, s_{ch_i^-}, s_{ch_i^+} \sim p_\pi}\left[V^\pi(s_{ch_i^-}) + V^\pi(s_{ch_i^+})\right] && \text{if } l(s_i) = 0 \text{ (non-leaf node).} \end{aligned}$$

The corresponding gradients when $l(s_i) = 1$ and $l(s_i) = 0$ are, respectively,

$$\nabla_\pi V^\pi(s_i) = 0, \text{ and}$$

$$\nabla_\pi V^\pi(s_i) = \mathbb{E}_{a_i, s_{ch_i^-}, s_{ch_i^+} \sim p_\pi}\left[\frac{\nabla_\pi \pi(a_i|s_i)}{\pi(a_i|s_i)}\left(V^\pi(s_{ch_i^-}) + V^\pi(s_{ch_i^+})\right) \nabla_\pi V^\pi(s_{ch_i^-}) + \nabla_\pi V^\pi(s_{ch_i^+})\right].$$

Let us now write the gradient of V^π,

$$\nabla_\pi V^\pi = \mathbb{E}_{s_0 \sim p_{init}}\left[\nabla_\pi V^\pi(s_0)\right].$$

Either we have $l(s_0) = 1$ and thus $\nabla_\pi V^\pi = 0$, or we can expand $\nabla_\pi V^\pi(s_0)$ to obtain

$$\nabla_\pi V^\pi = \mathbb{E}_{s_0, a_0, s_{ch_0^-}, s_{ch_0^+} \sim p_\pi}\left[\frac{\nabla_\pi \pi(a_0|s_0)}{\pi(a_0|s_0)}(V^\pi(s_{ch_0^-}) + V^\pi(s_{ch_0^+})) + \nabla_\pi V^\pi(s_{ch_0^-}) + \nabla_\pi V^\pi(s_{ch_0^+})\right].$$

Then again, each of of the terms $\nabla_\pi V^\pi(s_{ch_0^-})$ and $\nabla_\pi V^\pi(s_{ch_0^+})$ can be replaced by 0 if the corresponding node is a leaf node, or can be expanded further in the same way if it is a non-leaf node. By applying this rule recursively, we finally obtain

$$\begin{aligned} \nabla_\pi V^\pi &= \mathbb{E}_{\tau \sim p_\pi}\left[\sum_{i \in \mathcal{N} \setminus \mathcal{L}} \frac{\nabla_\pi \pi(a_i|s_i)}{\pi(a_i|s_i)}(V^\pi(s_{ch_i^-}) + V^\pi(s_{ch_i^+}))\right] \\ &= \mathbb{E}_{\tau \sim p_\pi}\left[\sum_{i \in \mathcal{N} \setminus \mathcal{L}} \frac{\nabla_\pi \pi(a_i|s_i)}{\pi(a_i|s_i)} \sum_{j \in d_i} r(s_j)\right] \\ &= \mathbb{E}_{\tau \sim p_\pi}\left[\sum_{i \in \mathcal{N} \setminus \mathcal{L}} \nabla_\pi \log \pi(a_i|s_i) \sum_{j \in d_i} r(s_j)\right]. \end{aligned}$$

□

Lemma 4. *In B&B, both children MILPs ($MILP_{ch_i^-}$ and $MILP_{ch_i^+}$) can be derived from the local MILP $MILP_i$ and branching decision $a_i = (j, x_j^\star)$, with j the index of a variable in $MILP_i$, and $x_j^\star$ the value to be used for branching.*

Proof. From the definition of B&B in Section 1.3, $MILP_{ch_i^-}$ (resp. $MILP_{ch_i^+}$) consist of $MILP_i$ augmented with the additional constraint $x_j \leq \lfloor x_j^\star \rfloor$ (resp. $x_j \geq \lceil x_j^\star \rceil$). □

Proof of Proposition 2. We shall now prove that, under Assumption 2, the B&B process can be formulated as a tree MDP $tM = (\mathscr{S}, \mathscr{A}, p_{init}, p^-_{ch}, p^+_{ch}, r, l)$, with states $s_i = (MILP_i, \bar{z}(i))$ and actions $a_i = (j, x^\star_j)$. First, the algorithm starts at the root node with an initial MILP, $MILP_0$, and an initial global upper bound $\bar{z}(0) = \infty$. Thus, the root state s_0 follows an arbitrary, user-defined MILP distribution $p_{init}(s_0)$, which is independent of the B&B algorithm. Second, Lemma 4, together with Assumption 2, ensures the existence of (deterministic) distributions $p^-_{ch}(s_{ch^-_i}|s_i, a_i)$ and $p^+_{ch}(s_{ch^+_i}|s_i, a_i)$, from which the B&B children states $s_{ch^-_i}$ and $s_{ch^+_i}$ are generated. Third, the reward function $r(s_i)$ is not part of the B&B algorithm, and can be arbitrarily defined to match any (compatible) B&B objective. Last, the leaf node indicator $l(s_i)$ is exactly the vanilla B&B leaf node criterion, and is obtained by solving the LP relaxation of $MILP_i$ constrained with upper bound $\bar{z}(i)$, which results in either an infeasible LP (leaf node), a MILP-feasible LP solution (leaf node), or a MILP-infeasible LP solution (non-leaf node). This concludes the proof. □

Proof of Proposition 3. Because $\bar{z}(0) = z^*$, the initial global upper bound is equal to the optimal solution value to the original MILP. Then, B&B will never be able to find a feasible solution that tightens that bound, and we necessarily have $\bar{z}(i) = \bar{z}(0), \forall i$. Hence $\bar{z}(ch^-_i) = \bar{z}(ch^+_i) = \bar{z}(0)$, which means they can be directly derived from s_i. This concludes the proof. □

Proof of Proposition 4. First, it is trivial to show that $\bar{z}(ch^-_i)$ can be derived from s_i. Because node i is not a leaf node, it has not resulted in an integral solution, and hence processing node i does not change the global upper bound. And since ch^-_i is processed directly after node i, we necessarily have $\bar{z}(ch^-_i) = \bar{z}(i)$. This, combined with Lemma 4, shows that $s_{ch^-_i}$ can be inferred from s_i and a_i. Second, we show how $\bar{z}(ch^+_i)$ can be derived from s_i and a_i. Because node ch^+_i is processed right after the whole subtree below ch^-_i has been processed, $\bar{z}(ch^+_i)$ is necessarily the minimum of $\bar{z}(i)$ and the optimal solution value of $MILP_{ch^-_i}$. Now, because $s_{ch^-_i}$ can be inferred from s_i and a_i, $MILP_{ch^-_i}$ can be recovered as well, and solved to obtain its optimal solution value. Therefore, $\bar{z}(ch^+_i)$ can be recovered from s_i and a_i. □

3.6.2. EXTENDED RESULTS

Here we provide extended evaluation results (Table 3.2) with the geometric mean of the solving times in seconds (Time) and the geometric mean of the final B&B tree size (Nodes). The results are averaged over the solving runs that finished successfully for all methods. This is, if a solving run reached the time limit for any method, this is excluded from the average. Table 3.3 shows the number of solving runs that timed out per method.

3.6.3. INSTANCE COLLECTIONS

This section presents the models used to generate our instance benchmarks. The parameters used to generate each benchmark are shown in Table 3.4.

COMBINATORIAL AUCTIONS

For m items, we are given n bids $\{\mathscr{B}_j\}^n_{j=1}$. Each bid $\mathscr{B}_j$ is a subset of the items with an associated bidding price p_j. The associated combinatorial auction problem is of the

following form:

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{n} p_j x_j \\ \text{subject to} \quad & \sum_{j:i\in\mathscr{B}_j} x_j \leq 1, \quad i = 1,...,m \\ & x_j \in \{0,1\} \;\; j = 1,...,n \end{aligned}$$

where x_j represents the action of choosing bid $\mathscr{B}_j$.

SET COVERING

Given the elements $1,2,...,m$, and a collection $\mathscr{S}$ of n sets whose union equals the set of all elements, the set cover problem can be formulated as follows:

$$\begin{aligned} \text{minimize} \quad & \sum_{s\in\mathscr{S}} x_s \\ \text{subject to} \quad & \sum_{s:e\in s} x_s \geq 1, \quad e = 1,...,m \\ & x_s \in \{0,1\} \;\; \forall s \in \mathscr{S} \end{aligned}$$

MAXIMUM INDEPENDENT SET

Given a graph G the maximum independent set problem consists in finding a subset of nodes of maximum cardinality such that no two nodes in that subset are connected. We use the clique formulation from [21]. Given a collection $\mathscr{C} \subseteq 2^V$ of cliques whose union covers all the edges of the graph G, the clique cover formulation is

$$\begin{aligned} \text{maximize} \quad & \sum_{v\in V} x_v \\ \text{subject to} \quad & \sum_{v\in C} x_v \leq 1, \quad \forall C \in \mathscr{C} \\ & x_v \in \{0,1\} \quad \forall v \in V \end{aligned}$$

CAPACITATED FACILITY LOCATION WITH UNSPLITTABLE DEMAND

Given a number n of clients with demands $\{d_j\}_{j=1}^n$, and a number m of facilities with fixed operating costs $\{f_i\}_{i=1}^m$ and capacities $\{s_i\}_{i=1}^m$, let c_{ij}/d_j be the unit transportation cost between facility i and client j, and let p_{ij}/d_j be the unit profit for facility i supplying client j. We try to solve the following problem

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij}x_{ij} + \sum_{i=1}^{m} f_i y_i \\ \text{subject to} \quad & \sum_{j=1}^{n} d_j x_{ij} \leq s_i y_i, \quad i = 1,...,m \\ & \sum_{i=1}^{m} x_{ij} \geq 1, \quad j = 1,...,n \\ & x_{ij} \in \{0,1\} \quad \forall i,j \\ & y_i \in \{0,1\} \quad \forall i \end{aligned}$$

where each variable x_{ij} represents the decision of facility i supplying client j's demand, and each variable y_i representing the decision of opening facility i for operation.

MULTIPLE KNAPSACK

Given n items with respective prices $\{p_j\}_{j=1}^{n}$ and weights $\{w_j\}_{j=1}^{n}$, and m knapsacks with capacities $\{c_i\}_{i=1}^{m}$, the multiple knapsack problem consists in placing a number of items in each of the knapsacks such that the price of the selected items is maximized, while the capacity of the knapsacks is not exceeded by the total weight of the items therein. Formally:

$$\begin{aligned}
\text{maximize } & \sum_{i=1}^{m}\sum_{j=1}^{n} p_j x_{ij} \\
\text{subject to } & \sum_{j=1}^{n} w_j x_{ij} \leq c_i, \quad i = 1,...,m \\
& \sum_{i=1}^{m} x_{ij} \leq 1, \quad j = 1,...,n \\
& x_{ij} \in \{0,1\} \ \forall i,j
\end{aligned}$$

where each variable x_{ij} represents the decision of placing item j in knapsack i.

Table 3.2: Evaluation on test instances (same size as training) and transfer instances (larger size). We report the geometric mean and standard deviation of the final B&B tree size and the solving time (lower is better for both).

	Test		Transfer	
Model	Nodes	Time	Nodes	Time
SCIP default	7.3 ± 39%	3.3 ± 10%	733.9 ± 26%	27.4 ± 7%
IL	52.2 ± 13%	2.1 ± 6%	805.1 ± 9%	14.6 ± 5%
RL (MDP)	86.7 ± 16%	2.2 ± 6%	1906.3 ± 18%	20.9 ± 11%
RL (tMDP+DFS)	86.1 ± 17%	2.2 ± 6%	1804.6 ± 17%	20.1 ± 9%
RL (tMDP+ObjLim)	87.0 ± 18%	2.2 ± 6%	1841.9 ± 18%	20.4 ± 10%
Combinatorial auctions				
Model	Nodes	Time	Nodes	Time
SCIP default	10.7 ± 24%	5.8 ± 6%	61.4 ± 19%	12.6 ± 5%
IL	51.8 ± 10%	4.0 ± 5%	145.0 ± 6%	8.0 ± 4%
RL (MDP)	196.3 ± 20%	5.1 ± 8%	853.3 ± 27%	14.9 ± 13%
RL (tMDP+DFS)	190.8 ± 20%	5.1 ± 7%	816.8 ± 25%	14.6 ± 12%
RL (tMDP+ObjLim)	193.5 ± 23%	5.1 ± 8%	826.4 ± 26%	14.6 ± 13%
Set covering				
Model	Nodes	Time	Nodes	Time
SCIP default	19.3 ± 52%	13.2 ± 13%	2867.1 ± 35%	167.4 ± 23%
IL	35.9 ± 36%	8.7 ± 10%	1774.8 ± 38%	85.7 ± 22%
RL (MDP)	91.8 ± 56%	9.5 ± 16%	2768.5 ± 76%	85.6 ± 51%
RL (tMDP+DFS)	89.8 ± 51%	9.5 ± 17%	2970.0 ± 76%	90.6 ± 51%
RL (tMDP+ObjLim)	85.4 ± 53%	9.4 ± 17%	2763.6 ± 74%	86.1 ± 47%
Maximum independent set				
Model	Nodes	Time	Nodes	Time
SCIP default	203.6 ± 63%	16.9 ± 34%	344.3 ± 57%	40.3 ± 36%
IL	247.5 ± 39%	7.2 ± 26%	407.8 ± 37%	13.6 ± 24%
RL (MDP)	393.2 ± 47%	8.7 ± 29%	679.4 ± 52%	17.2 ± 33%
RL (tMDP+DFS)	360.4 ± 46%	8.3 ± 30%	609.1 ± 47%	15.9 ± 29%
RL (tMDP+ObjLim)	325.4 ± 41%	7.9 ± 26%	496.0 ± 48%	14.5 ± 28%
Facility location				
Model	Nodes	Time	Nodes	Time
SCIP default	267.8 ± 96%	1.5 ± 54%	592.3 ± 75%	3.7 ± 42%
IL	228.0 ± 95%	1.8 ± 66%	1066.1 ± 101%	7.1 ± 82%
RL (MDP)	143.4 ± 76%	1.3 ± 48%	518.4 ± 79%	4.5 ± 58%
RL (tMDP+DFS)	135.8 ± 75%	1.3 ± 48%	495.1 ± 81%	4.3 ± 59%
RL (tMDP+ObjLim)	142.4 ± 78%	1.4 ± 48%	425.3 ± 64%	3.9 ± 46%
Multiple knapsack				

Table 3.3: Number of solving runs (instance–seed pairs) out of 200 that hit the 1h time limit.

Model	C. Auct.	Set Cov.	M.Ind.Set	Fac. Loc.	M. Knap.
SCIP default	0	0	0	1	0
IL	0	0	0	0	0
RL (MDP)	0	0	1	0	0
RL (tMDP+DFS)	0	0	1	0	0
RL (tMDP+ObjLim)	0	0	1	0	0

Test

Model	C. Auct.	Set Cov.	M.Ind.Set	Fac. Loc.	M. Knap.
SCIP default	0	0	1	13	0
IL	0	0	0	0	3
RL (MDP)	0	0	20	1	2
RL (tMDP+DFS)	0	0	18	1	0
RL (tMDP+ObjLim)	0	0	16	1	0

Transfer

Table 3.4: Size of the instances used for training and evaluation, for each problem benchmark. We evaluate the final performance on instances of the same size as training (test), and also larger instances (transfer).

Benchmark	Generation method	Parameters	Train / Test	Transfer
Combinatorial auction	Leyton-Brown et al. [102] with arbitrary relationships	Items Bids	100 500	200 1000
Set covering	Balas and Ho [16]	Items Sets	400 750	500 1000
Maximum independent set	Bergman et al. [21] on Erdős-Rény graphs	Nodes Affinity	500 4	1000 4
Facility location	Cornuéjols et al. [43] with unsplittable demand	Customers Facilities	35 35	60 35
Multiple knapsack	Fukunaga [61]	Items Knapsacks	100 6	100 12

4

LATTICE REFORMULATIONS FOR IP

Branching on single variable disjunctions has become an ubiquitous strategy that most MILP solvers use by default. The more general problem of choosing a general disjunction, i.e., a hyperplane, for branching has been comparatively less studied. The reason is that, while this approach can lead to smaller search trees, several obstacles exist when it comes to their implementation, such as the generation and selection of such directions, or numerical considerations. Still, algorithms with theoretical guarantees, such as the one by Lenstra Jr. [101] and the one by Lovász and Scarf [108], make crucial use of branching on hyperplanes. Both algorithms employ lattice basis reduction. In this chapter, we discuss a line of methodologies that also make use of lattice information. These methodologies construct a reformulation of the feasible set. The reformulated variables can be seen as hyperplanes in the original space, and consequently they can be used as a way to construct lattice-informed branching directions.

In particular, we use this perspective to study the methodologies of Aardal et al. [3] (AHL) and Krishnamoorthy and Pataki [94] (KP). For the purpose of this chapter, we focus on pure integer programs (IPs). The chapter is organized as follows. We start with a brief introduction to key concepts about lattices (Section 4.1). In Section 4.2 we discuss non-standard approaches to solving IPs, with a more detailed overview of the algorithms of Lenstra Jr. [101] and Lovász and Scarf [108] which will be of importance to our study. Section 4.3 introduces a variation of the AHL reformulation and presents a useful bound on the size of the reformulation's relaxation. In Section 4.4 we present a computational analysis that compares the different formulations on different metrics and instance types, which allows us to make conclusions about the effectiveness of the generated branching directions. We also draw connections with the algorithm of Lovász and Scarf [108]. We conclude this chapter with a reflection on our findings and on future avenues of research (Section 4.5). The contents of this chapter are based on our published work [4]. The code for reproducing all experiments is available online [133].

4.1. SOME PRELIMINARIES ON LATTICES

Definition 5 (Lattice). *Let $\{\boldsymbol{b}^i\}_{i=1}^K$ be linearly independent vectors in $\mathbb{R}^n$. The set*

$$L = \left\{ \boldsymbol{x} \mid \boldsymbol{x} = \sum_{i=1}^{K} \lambda_i \boldsymbol{b}^i,\ \lambda_i \in \mathbb{Z},\ i \in [K] \right\}$$

is called a lattice. *The vectors $\{\boldsymbol{b}^i\}_{i=1}^K$ are a* basis *of L.*

Definition 6 (Gram–Schmidt process). *Given linearly independent vectors $\boldsymbol{b}^1, \ldots, \boldsymbol{b}^K$, the Gram-Schmidt process generates orthogonal vectors $\boldsymbol{b}^{1*}, \ldots, \boldsymbol{b}^{K*}$ following the recursion*

$$\boldsymbol{b}^{1*} = \boldsymbol{b}^1$$
$$\boldsymbol{b}^{j*} = \boldsymbol{b}^j - \sum_{i=1}^{j-1} \mu_{ij} \boldsymbol{b}^{i*}, \quad \text{with} \quad \mu_{ij} = \frac{\langle \boldsymbol{b}^j, \boldsymbol{b}^{i*} \rangle}{\|\boldsymbol{b}^{i*}\|^2} \quad \text{for} \quad 2 \le j \le K.$$

Definition 7 (LLL-reduced basis [99]). *Given is a basis $\boldsymbol{b}^1, \boldsymbol{b}^2, \ldots, \boldsymbol{b}^K$ of the lattice L, and the associated Gram-Schmidt vectors $\boldsymbol{b}^{1*}, \boldsymbol{b}^{2*}, \ldots, \boldsymbol{b}^{K*}$. Let $\mu_{ij} = \langle \boldsymbol{b}^i, \boldsymbol{b}^{j*} \rangle / \langle \boldsymbol{b}^{j*}, \boldsymbol{b}^{j*} \rangle$, for $1 \le j < i \le K$. The basis is* reduced *if*

$$|\mu_{ij}| \le \frac{1}{2}, \text{ for } 1 \le j < i \le K \tag{4.1}$$

and, for $y \in (\frac{1}{4}, 1)$ and $1 < i \le K$,

$$\|\boldsymbol{b}^{i*} + \mu_{i,i-1} \boldsymbol{b}^{(i-1)*}\|^2 \ge y \|\boldsymbol{b}^{(i-1)*}\|^2. \tag{4.2}$$

Notice that condition (4.2) is satisfied if $(y - \mu_{i,i-1}^2)\|\boldsymbol{b}^{(i-1)*}\|^2 \le \|\boldsymbol{b}^{i*}\|^2$, or equivalently if $\|\boldsymbol{b}^{(i-1)*}\|^2 \le c\|\boldsymbol{b}^{i*}\|^2$, where $c = \frac{1}{y-1/4}$. A *c-reduced basis* is an LLL-reduced basis for a given value of the constant c.

The LLL algorithm by Lenstra et al. [99] can reduce any given basis in polynomial time.

Definition 8 (Lattice determinant). *Given is any basis $\boldsymbol{b}^1, \boldsymbol{b}^2, \ldots, \boldsymbol{b}^K$ of the lattice L. Let $\boldsymbol{B}$ be the matrix whose columns correspond to the basis vectors. The determinant $d(L)$ of L is defined as*

$$d(L) = \sqrt{\det(\boldsymbol{B}^T \boldsymbol{B})}.$$

In the case that L is full-dimensional, this is equal to $\det(\boldsymbol{B})$.

Notice that $d(L)$ does not depend on the choice of basis.

Definition 9 (Polar lattice). *Let L be a lattice in a Euclidean vector space E with $\dim E = \mathrm{rk}\, L$. Then the polar lattice L^* of L is defined as*

$$L^* = \{\boldsymbol{x} \in E :\ \langle \boldsymbol{x}, \boldsymbol{y} \rangle \subset \mathbb{Z} \text{ for all } \boldsymbol{y} \in L\}.$$

For a lattice L and its polar L^* we have $\operatorname{rk} L = \operatorname{rk} L^*$, $L^{**} = L$, and

$$d(L) = \frac{1}{d(L^*)}. \tag{4.3}$$

Definition 10 (Pure sublattice)**.** *Let L be a lattice in a Euclidean vector space E, and let K be a subgroup of L. If there exists a subspace D of E such that $K = L \cap D$, then K is called a pure sublattice.*

Suppose that K is a pure sublattice of the lattice L. Then, the following holds:

$$d(L) = d(K) \cdot d(L/K). \tag{4.4}$$

Let L be a lattice with polar L^*, and let K be a pure sublattice of L. Then, for $K^\perp = \{\boldsymbol{x} \in L^* \mid \langle \boldsymbol{x}, K \rangle = 0\}$, and we can write

$$K^\perp = (L/K)^*. \tag{4.5}$$

A thorough treatment of this topic can be found in Lenstra [100].

4.2. NON-STANDARD ALGORITHMS FOR IP

Most implementations of the B&B algorithm described in Section 1.3.2 involve a binary tree, where branching happens as a result of a disjunction on a single (integer) variable that takes a fractional value in the LP relaxation solution $\boldsymbol{x}^{LP}$. This is, by imposing

$$x_j \le \lfloor x_j^{LP} \rfloor \qquad \text{or} \qquad x_j \ge \lceil x_j^{LP} \rceil$$

for some $j \in \mathscr{I}$. This is the standard form of the B&B algorithm for IP. More generally, one can use a tuple $(\boldsymbol{\pi}, \pi_0) \in \mathbb{Z}^n \times \mathbb{Z}$ such that $\pi_0 < \boldsymbol{\pi}^T \boldsymbol{x}^{LP} < \pi_0 + 1$. This is, the solution to the LP relaxation is cut off by imposing

$$\boldsymbol{\pi}^T \boldsymbol{x} \le \pi_0 \qquad \text{or} \qquad \boldsymbol{\pi}^T \boldsymbol{x} \ge \pi_0 + 1.$$

Computing an optimal pair $(\boldsymbol{\pi}, \pi_0) \in \mathbb{Z}^n \times \mathbb{Z}$ that minimizes the size of the search tree is $\mathscr{NP}$-hard [109]. However, several methods for computing good disjunctions have been proposed. While these general disjunctions often lead to smaller search trees, the performance in terms of solution time commonly worsens. There are two main reasons for this. First, the candidate pool is typically larger and more time is spent choosing a branching direction. On top of that, other structural issues exist related to the other solver components. For example, most implementations of domain propagation or conflict analysis benefit from branching on single-variable disjunctions. Most importantly, when using general disjunctions, LP relaxations increasingly grow with tree depth[1], and simplex warm-starting is more challenging.

In spite of these difficulties, the use of general disjunctions has proven very promising in theory, but also in practice when it comes to reducing the tree size. A deeper

[1] Notice that single-variable branching is handled as a change in the variable's bounds, and therefore does not increase the number of constraints.

analysis of disjunction-finding methods and their practical use can allow the MILP community to evaluate their applicability and give good motivation to solve the technical challenges concerning other solver components. The section at hand will present a short review of disjunction-finding algorithms for branching, followed by a more in-depth description of two algorithms that will be relevant for the purpose of this chapter.

4.2.1. DISJUNCTION-FINDING ALGORITHMS

Several methodologies to find general disjunctions for branching have been proposed. We can distinguish two main lines of research, according to the objective function of choice used as a proxy for branching quality. These are (i) thinness of the LP relaxation polytope and (ii) objective value degradation in children nodes.

Branching on thin directions is the key to unlocking algorithms that run in polynomial time with a fixed number of variables. The algorithms of Lenstra Jr. [101], and of Lovász and Scarf [108] follow this line. These algorithms are of great importance for IP history and have strong connections with the reformulations treated in this chapter. For this reason, we devote Sections 4.2.2 and 4.2.3 to discuss them in detail. Other more recent examples are the work of Mehrotra and Li [113] and of Elhedhli and Naoum-Sawaya [54]. These methods suffer from the disadvantage that finding thin directions at every node is very computationally intensive.

In line with research on the variable selection problem, one can formulate the selection of a good general disjunction in terms of the objective value degradation in the generated children nodes (see Section 3.1). However, in the single variable setting the number of candidates is limited (only integer variables that take a fractional value in the relaxation need to be considered), whereas in the general setting there is an infinite pool of branching candidates. Several approaches have been proposed to reduce the candidate pool to a subset of promising directions before probing the potential children nodes in each case. For example, by using a greedy heuristic over the set $\boldsymbol{\pi} \in \{-1, 0, 1\}^n$ [121], by considering split disjunctions coming from Gomory cuts [85], or by considering multi-aggregated variables [63].

Other approaches include the one of Mahajan and Ralphs [110], who formulate appropriate disjunction finding MILPs which need to be solved at each subproblem. They note that this comes at a great (sometimes impractical) computational cost and suggest methods for alleviating the difficulty of such formulation. Finally, let us mention the work of Mahmoud and Chinnek [111], who take a different approach and test different heuristic metrics to find branching directions that lead to feasible solutions as fast as possible. Notably, they also study the interesting question of when (in which nodes) it is effective to apply a general disjunction. They propose two metrics of interest, based on the number of candidate variables and the progress in dual bound throughout the search.

In the following, we discuss the algorithms of Lenstra and of Lovász and Scarf in more detail, as they will be of importance to the results of this chapter.

4.2.2. LENSTRA'S ALGORITHM

The algorithm by Lenstra Jr. [101] provided the first proof that IP is solvable in polynomial time in fixed dimension, a breakthrough in the history of the field. The algorithm can be applied to decide whether $X_{LP} \cap \mathbb{Z}^n = \emptyset$, with

$$X_{LP} = \{\boldsymbol{x} \in \mathbb{R}^n : A\boldsymbol{x} \leq \boldsymbol{b}\}.$$

If $X_{LP} \cap \mathbb{Z}^n \neq \emptyset$, the algorithm produces a vector in $X_{LP} \cap \mathbb{Z}^n$. We assume, without loss of generality, that X_{LP} is bounded and full-dimensional.

The first step of this algorithm consists in finding a non-sigular endomorphism $\tau : \mathbb{R}^n \to \mathbb{R}^n$ such that τX_{LP} has "spherical" appearance. In more precise terms, for $\boldsymbol{p} \in \mathbb{R}^n$, $z \in \mathbb{R}_{>0}$, let

$$B(\boldsymbol{p}, z) = \{\boldsymbol{x} \in \mathbb{R}^n : ||\boldsymbol{x} - \boldsymbol{p}|| \leq z\}$$

be the closed ball with center $\boldsymbol{p}$ and radius z. Lenstra Jr. [101] first specifies a procedure that, in polynomial time, is able to construct τ such that

$$B(\boldsymbol{p}, r) \subset \tau X_{LP} \subset B(\boldsymbol{p}, R) \tag{4.6}$$

for some $\boldsymbol{p} \in \tau X_{LP}$, and with r, R satisfying

$$\frac{R}{r} \leq C_n^1$$

with C_n^1 a constant only depending on n.

Once the endomorphism τ has been constructed, we consider the equivalent problem of determining whether $\tau X_{LP} \cap \tau \mathbb{Z}^n = \emptyset$. Notice that $L := \tau \mathbb{Z}^n$ is a lattice. Let $\{\boldsymbol{b}^i\}_{i=1}^n$ denote any basis of L with vectors indexed in such a way that $||\boldsymbol{b}^n|| = \max\{||\boldsymbol{b}^i|| : i \in [n]\}$. Let $L' := \sum_{i=1}^{n-1} \boldsymbol{b}^i$ be the lattice spanned by the first $n-1$ lattice vectors and let $H = \text{span}\{\boldsymbol{b}^1, ..., \boldsymbol{b}^{n-1}\}$. Then

$$L = L' + \mathbb{Z}\boldsymbol{b}^n \subset H + \mathbb{Z}\boldsymbol{b}^n = \cup_{k \in \mathbb{Z}}(H + k\boldsymbol{b}^n).$$

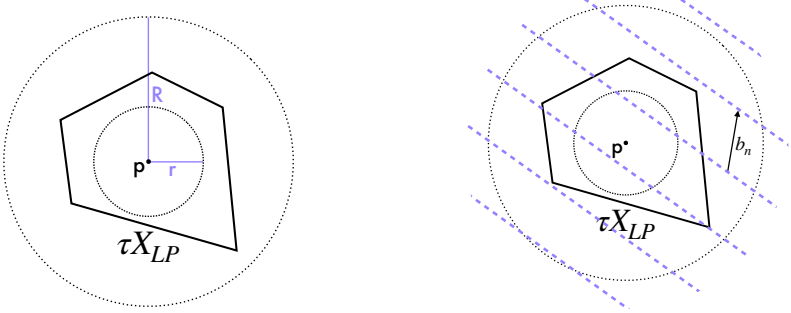
This means that the body of interest $\tau X_{LP} \cap \tau \mathbb{Z}^n$ is contained in the union of countably many hyperplanes. Let t be the number of such hyperplanes that intersect τX_{LP}. Lenstra Jr. [101] shows that if the chosen basis for L is LLL-reduced, then this number is bounded by

$$t \leq C_n^1 \cdot C_n^2 \cdot \sqrt{n} \tag{4.7}$$

where C_n^2 is another constant only depending on n.

The algorithm of Lenstra first derives, in polynomial time, a vector $\boldsymbol{x} \in L$. If $\boldsymbol{x} \in \tau X_{LP}$ then the algorithm terminates. Otherwise, the result in (4.7) is used to define a branching strategy: t subproblems in dimension $n-1$ are created by intersecting the feasible region with one of the hyperplanes. The procedure is applied recursively in each of the subproblems. Notice that the number of subproblems is bounded by a constant only

depending on n and that the corresponding decision tree can be at most n levels deep. This, together with the fact that each sub-routine runs in polynomial time, gives us the desired result.



(a) Step 1 (b) Step 2

Figure 4.1: The two steps of Lenstra's algorithm. The first step consists in finding an endomorphism τ that makes the LP-relaxation appear regular. On the second step, basis reduction is applied to find a direction for branching.

INTERPRETATION

The first step of the algorithm entails a regularization of the LP relaxation. This step ensures that, if the lattice vector x does not belong to τX_{LP}, then τX_{LP} has a small volume and a regular shape, therefore it is not wide in any direction. On the other hand, this transformation translates the lack of regularity from the LP polytope to the lattice, which needs to be handled by means of finding a suitable basis of the new lattice $\tau\mathbb{Z}^n$. Basis reduction allows us to find a direction $\boldsymbol{b}^n$ that defines good hyperplanes for branching. Because of the reduced property of the basis, the hyperplanes are guaranteed to be well spaced, in the sense that only few of them actually intersect the transformed polytope. In turn this means that we can bound the number of branchings by a constant depending only on n. The distance between two such consecutive hyperplanes is related to the norm of $\boldsymbol{b}^n$. It is in fact the norm of the projection of $\boldsymbol{b}_n$ onto the orthogonal complement of the hyperplanes. A long basis vector $\boldsymbol{b}^n$ is associated with a thin direction in the original polytope.

4.2.3. THE LOVASZ-SCARF ALGORITHM

In 1983, Lovász and Scarf [108] introduced another algorithm for IP that runs in polynomial time in fixed dimension. The driving force of this algorithm is a procedure that they call *generalized basis reduction.* While the definition of LLL-reduced basis makes use of the Euclidian distance (see Def. 7), the generalized basis reduction algorithm uses a different norm. This norm incorporates information about the shape of the polytope X_{LP}.

In particular, let us start by considering a compact convex body C in $\mathbb{R}^n$ of positive volume and symmetric about the origin, and let us define the following metric

$$F(\boldsymbol{x}) = \inf\left\{\lambda \geq 0 \,\middle|\, \frac{\boldsymbol{x}}{\lambda} \in C\right\}. \tag{4.8}$$

Associated with the body C is its polar body C^* defined as

$$C^* = \{\boldsymbol{y} \in \mathbb{R}^n \mid \boldsymbol{y}^\mathsf{T}\boldsymbol{x} \leq 1, \text{ for all } \boldsymbol{x} \in C\}. \tag{4.9}$$

We can use C^* to reformulate the distance function

$$F(\boldsymbol{x}) = \max\{\boldsymbol{x}^\mathsf{T}\boldsymbol{y} \mid \boldsymbol{y} \in C^*\}. \tag{4.10}$$

We are now ready to define the distance functions that are key to generalized basis reduction.

Definition 11. *The family of functions $F_i(\boldsymbol{x})$, for $i \in [n]$, associated with convex body C and basis $\{\boldsymbol{b}^1, ..., \boldsymbol{b}^n\}$ is defined as*

$$F_i(\boldsymbol{x}) = \max\{\boldsymbol{x}^T\boldsymbol{y} \mid \boldsymbol{y} \in C^*, \boldsymbol{y} \in span^\perp\{\boldsymbol{b}^1, ..., \boldsymbol{b}^{i-1}\}\}$$

Notice that $F_1(\boldsymbol{x}) = F(\boldsymbol{x})$. This family of functions defines an alternative notion of reduced basis.

Definition 12. *(LS-reduced basis, [108]). Given a lattice L with basis $\boldsymbol{b}^1, \ldots, \boldsymbol{b}^n$ and a convex body C, let F_i denote the family of functions associated with C, for $i \in [n]$. Fix $0 < y < 1/2$. The basis is reduced if, for $i \in [n-1]$,*

$$\begin{aligned} F_i(\boldsymbol{b}^{i+1}) &\geq (1-y)F_i(\boldsymbol{b}^i), \\ F_i(\boldsymbol{b}^{i+1} + \mu\boldsymbol{b}^i) &\geq F_i(\boldsymbol{b}^{i+1}), \text{ for all } \mu \in \mathbb{Z}. \end{aligned}$$

Lovász and Scarf [108] provide a procedure to find such a basis. This is known as the *generalized basis reduction algorithm*, and it runs in polynomial time for fixed n. Moreover, they show that the resulting basis provides a vector $\boldsymbol{b}^1$ that is an approximation of the shortest lattice vector with respect to the desired metric. This is,

$$\min_{\boldsymbol{x} \in \mathbb{Z}^n \setminus \{0\}} F(\boldsymbol{x}) \leq F(\boldsymbol{b}^1) \cdot \left(\frac{1}{2} - y\right)^{n-1}.$$

To understand how the Lovász-Scarf norm relates to the polytope we first consider a well-known fact from lattice theory. Let $L \subset E$ be a lattice of full rank in a Euclidean vector space E and let L^* be its associated polar lattice (as defined in Def. 9). For $\boldsymbol{x} \in E \setminus \{\boldsymbol{0}\}$ we have, by definition, that $\boldsymbol{x} \in L^*$ if and only if the lattice L is contained in $\{\boldsymbol{y} \in E \mid \boldsymbol{x}^\mathsf{T}\boldsymbol{y} \in \mathbb{Z}\}$, which can be written as

$$\{\boldsymbol{y} \in E \mid \boldsymbol{x}^\mathsf{T}\boldsymbol{y} \in \mathbb{Z}\} = \text{span}^\perp\{\boldsymbol{x}\} + k\boldsymbol{x}', \tag{4.11}$$

where $k \in \mathbb{Z}$, and $\boldsymbol{x}' = \boldsymbol{x}/\|\boldsymbol{x}\|^2$. So (4.11) implies that $\boldsymbol{x} \in L^*$ if and only if L is contained in the translates $\text{span}^\perp\{\boldsymbol{x}\} + k\boldsymbol{x}'$. If the vector $\boldsymbol{x}$ is short, the vector $\boldsymbol{x}'$ is long, so finding a

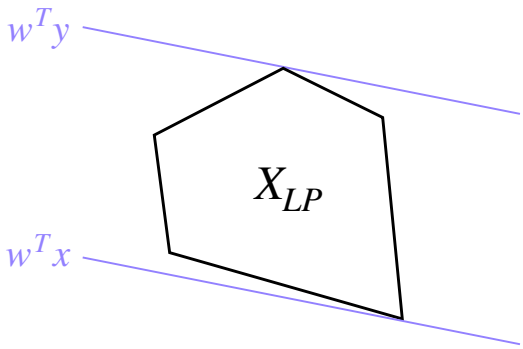


Figure 4.2: A depiction of the distance funtion $F(\boldsymbol{w})$ (see Eq 4.12). Given vector $\boldsymbol{w}$, $F(\boldsymbol{w})$ returns the width of X_{LP} in direction $\boldsymbol{w}$.

short vector in the polar lattice L^* is equivalent to finding widely spaced parallel "lattice hyperplanes".

To determine whether the full-dimensional polytope X_{LP} contains an integer vector, we observe that the sets $(X_{LP} - X_{LP})$ and $(X_{LP} - X_{LP})^*$ are compact, convex sets that are symmetric about the origin, and have positive volume. The algorithm finds an approximation of the shortest vector given the distance function (4.8) for the set $C = (X_{LP} - X_{LP})^*$, which is then the same as finding a vector $\boldsymbol{w}$ such that the lattice is contained in widely spaced translates of the orthogonal complement of $\boldsymbol{w}$, or equivalently, a direction $\boldsymbol{w}$ in which the width of X_{LP} is thin. This can be seen by re-writing $F(\boldsymbol{w})$ using its polar representation (4.10) as

$$F(\boldsymbol{w}) = \max\left\{\boldsymbol{w}^{\mathsf{T}}(\boldsymbol{x}-\boldsymbol{y}) \mid \boldsymbol{x} \in X_{LP},\ \boldsymbol{y} \in X_{LP}\right\}. \tag{4.12}$$

This is depicted in Figure 4.2.

We refer to Cook et al. [41] for a detailed description of an implementation of the generalized basis reduction algorithm and to [132] for our own implementation.

INTERPRETATION

The algorithm of Lovász and Scarf has a strong connection with Lenstra's algorithm. The latter starts by doing a regularization step, translating any "issues" from the polytope into the lattice. In a second step, an LLL-reduced basis of the transformed lattice is computed. One can interpret the algorithm of Lovász and Scarf as the combination of these two steps into one, where the basis reduction procedure considers the shape of the polytope by using a special metric. The question is whether the approximation by balls combined with polynomial basis reduction (Lenstra's algorithm) is computationally more demanding than computing the generalized reduced basis. To the best of our knowledge, no implementation of Lenstra's algorithm has been reported on, whereas the Lovász-Scarf algorithm has been implemented and successfully tested by Cook et al. [41].

An implementation advantage of the Lovász-Scarf algorithm is that the main engine of the reduction algorithm in the polyhedral case is linear programming.

4.3. LATTICE-BASED REFORMULATIONS

In this section, we first introduce the relevant reformulation procedures. In particular, we describe the procedure introduce by Aardal et al. [3] (AHL reformulation) and the one by Krishnamoorthy and Pataki [94] (KP reformulation). Section 4.3.3 presents our main result on the volume of the reformulation. For the purpose of this chapter, we consider pure integer programs.

4.3.1. THE AHL REFORMULATION

Let us first, without loss of generality, re-write the feasible set (1.2) as

$$X = \{\boldsymbol{x} \in \mathbb{Z}^n \mid \boldsymbol{A}\boldsymbol{x} = \boldsymbol{a}_0, \boldsymbol{x} \geq 0\}. \tag{4.13}$$

We assume that $\boldsymbol{A} \in \mathbb{Z}^{m\times n}$ has full row rank. The linear relaxation of X is denoted by X_{LP}.

It is well-known [148] that if the system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{a}_0$ is integer feasible, then there exist an integer vector $\bar{\boldsymbol{x}}$ and linearly independent integer vectors $\{\boldsymbol{x}^j\}_{j=1}^{n-m}$ such that

$$\left\{\boldsymbol{x} \in \mathbb{Z}^n \mid \boldsymbol{A}\boldsymbol{x} = \boldsymbol{a}_0\right\} = \bar{\boldsymbol{x}} + \sum_{j=1}^{n-m} \lambda_j \boldsymbol{x}^j$$

with $\lambda_j \in \mathbb{Z}$ for $j = 1,\dots,n-m$. In particular, we can take $\bar{\boldsymbol{x}}$ to be any vector satisfying $\boldsymbol{A}\bar{\boldsymbol{x}} = \boldsymbol{a}_0$, and $\{\boldsymbol{x}^j\}_{j=1}^{n-m}$ to be basis vectors of the lattice

$$\ker_{\mathbb{Z}} \boldsymbol{A} := \{\boldsymbol{x} \in \mathbb{Z}^n \mid \boldsymbol{A}\boldsymbol{x} = \boldsymbol{0}\}. \tag{4.14}$$

Let $\{\boldsymbol{b}^j\}_{j=1}^{n-m}$ be such a basis and let $\boldsymbol{B}$ denote the matrix $\boldsymbol{B} = [\boldsymbol{b}^j]_{j=1}^{n-m}$ whose columns are the basis vectors. Aardal et al. [3] propose to use this fact to reformulate the feasible set in the following way.

Definition 13 (AHL reformulation [3])**.** *Let X be defined as in (4.13). Let $\bar{\boldsymbol{x}}$ be any vector satisfying $\boldsymbol{A}\bar{\boldsymbol{x}} = \boldsymbol{a}_0$ and let $\boldsymbol{B}$ be a basis of $\ker_{\mathbb{Z}} \boldsymbol{A}$. Then X can be re-written as*

$$X^{\boldsymbol{\lambda}} = \{\boldsymbol{\lambda} \in \mathbb{Z}^{n-m} \mid \boldsymbol{B}\boldsymbol{\lambda} \geq -\bar{\boldsymbol{x}}\}. \tag{4.15}$$

Aardal et al. [3] use LLL reduction to derive the reformulation (4.15) of the set X. They do so by considering a higher-dimensional lattice in which the vectors $\bar{\boldsymbol{x}}$ and $\{\boldsymbol{b}^j\}_{j=1}^{n-m}$ are short and such that finding an initial basis for that lattice is trivial. The LLL algorithm then outputs such vectors or gives a certificate for integer infeasibility in polynomial time.

In the following, we denote the linear relaxation of $X^{\boldsymbol{\lambda}}$ by $X^{\boldsymbol{\lambda}}_{LP}$. Notice that the sets $X^{\boldsymbol{\lambda}}$ and $X^{\boldsymbol{\lambda}}_{LP}$ are full-dimensional. It is clear that upper bounds $\boldsymbol{u}$ on the variables $\boldsymbol{x}$ can be handled by adding the constraints $\boldsymbol{B}\boldsymbol{\lambda} \leq \boldsymbol{u} - \bar{\boldsymbol{x}}$.

4.3.2. THE KP REFORMULATION

For convenience of notation, let us start by writing

$$X = \{\boldsymbol{x} \in \mathbb{Z}^n \mid \boldsymbol{l} \leq \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{u}\}. \tag{4.16}$$

Krishnamoorthy and Pataki [94] point out that the columns of the constraint matrix $\boldsymbol{A} = [\boldsymbol{a}^i]_{i=1}^n$ can be interpreted as a basis of a lattice $L = \sum_{i=1}^n \mathbb{Z}\boldsymbol{a}^i$. Following this idea they propose the following reformulation.

Definition 14 (KP reformulation [94])**.** *Let X be defined as in (4.16). Let A' be an LLL-reduced basis of the lattice $L = \sum_{i=1}^n \mathbb{Z}\boldsymbol{a}^i$, where $[\boldsymbol{a}^i]_{i=1}^n = \boldsymbol{A}$. Then X can be re-written as*

$$X^{\boldsymbol{y}} = \{\boldsymbol{y} \in \mathbb{Z}^n \mid \boldsymbol{l} \leq \boldsymbol{A}'\boldsymbol{y} \leq \boldsymbol{u}\} \tag{4.17}$$

Notice that $\boldsymbol{A}' = \boldsymbol{A}\boldsymbol{U}$ for some unimodular matrix $\boldsymbol{U}$. The KP-reformulation can of course also be applied to an equality system, but it does not result in a dimension reduction as in the AHL-reformulation.

It is worth noticing that if the original set is full-dimensional, then the resulting KP-reformulation is equivalent to the AHL-reformulation, see the appendix. In practice there are differences, as the AHL-reformulation involves the extended matrix $(\boldsymbol{A}, \boldsymbol{a}_0)$ instead of only $\boldsymbol{A}$. When using the KP reformulation for equations, we use $\boldsymbol{A}\boldsymbol{U}\boldsymbol{y} = \boldsymbol{a}_0$.

4.3.3. THE REFORMULATED VOLUME

Here we consider the integer knapsack case, i.e., the matrix $\boldsymbol{A}$ in formulation (4.13) of X has one row $\boldsymbol{a}$ ($m = 1$). Furthermore, we assume that $a_0,\ a_1, \ldots, a_n > 0$, and that $\gcd(a_1, \ldots, a_n) = 1$.

In this section, we will study the volume of a rectangular box oriented in the coordinate directions containing the linear relaxation of the problem at hand. This is relevant because, for a given instance, it gives us an upper bound on the number of branch-and-bound nodes we need to solve the instance when branching along the coordinate directions.

It is straightforward to determine the volume of the smallest such box containing X_{LP}, which is

$$V(X_{LP}) = \frac{a_0^n}{\Pi_{i=1}^n a_i}. \tag{4.18}$$

In this section we derive an expression for an upper bound on the volume of a box containing $X_{LP}^{\boldsymbol{\lambda}}$, denoted $V(X_{LP}^{\boldsymbol{\lambda}})$. We do this by choosing an appropriate basis $\boldsymbol{B}$ for the lattice $\ker_{\mathbb{Z}} \boldsymbol{a}$. The idea comes from the approximation step in Lenstra's algorithm [101], where an appropriate polytope transformation is derived by mapping a high-volume simplex that is contained in the polytope to a regular simplex.

For our proof we will use a linear map $\boldsymbol{D}$ of $\mathbb{R}^n$ that maps X_{LP} to a regular simplex:

$$\boldsymbol{D} = \begin{pmatrix} a_1 & 0 & \cdots & 0 \\ 0 & a_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_n \end{pmatrix}. \tag{4.19}$$

Applying this map to $X_{LP} = \{\boldsymbol{x} \in \mathbb{R}^n \mid \sum_{i=1}^n a_i x_i = a_0\}$ yields a simplex that intersects each of the coordinate axes at the point $a_0 \boldsymbol{e}_i$, $i \in [n]$:

$$X_{LP}^D = \left\{ \boldsymbol{x} \in \mathbb{R}^n \,\middle|\, \sum_{i=1}^n x_i = a_0 \right\}.$$

Applying $\boldsymbol{D}$ to $\mathbb{Z}^n$ yields the lattice $\Lambda = \boldsymbol{D}\mathbb{Z}^n$ with basis $\boldsymbol{D}$. Under this map, the lattice $\ker_{\mathbb{Z}} \boldsymbol{a}$ becomes

$$\boldsymbol{D}\ker_{\mathbb{Z}} \boldsymbol{a} = \left\{ \boldsymbol{x} \in \Lambda \mid \sum_{i=1}^n x_i = 0 \right\} = \ker_{\Lambda} \mathbf{1}.$$

Notice that if $\boldsymbol{B} = [\boldsymbol{b}^i]_{i=1}^{n-1}$ is a basis for $\ker_{\mathbb{Z}} \boldsymbol{a}$, then $\hat{\boldsymbol{B}} = \boldsymbol{D}\boldsymbol{B}$ forms a basis for $\ker_{\Lambda} \mathbf{1}$. Similarly, if $\bar{\boldsymbol{x}}$ is a vector satisfying $\boldsymbol{a}\bar{\boldsymbol{x}} = a_0$, then the vector $\hat{\boldsymbol{x}} = \boldsymbol{D}\boldsymbol{x}$ satisfies $\sum_{i=1}^n \hat{x}_i = a_0$.

Remark. Notice that working in the lattice $\Lambda = \boldsymbol{D}\mathbb{Z}^n$ using the Euclidean norm $\|\boldsymbol{x}\|^2 = \sum_{i=1}^n x_i^2$ is equivalent to working in the lattice $\mathbb{Z}^n$ using the norm $q(\boldsymbol{x}) := \|\boldsymbol{x}\|_{\boldsymbol{D}}^2 = \sum_{i=1}^n a_i^2 x_i^2$.

The last ingredient we will need to prove our result is Schoof's lemma.

Lemma 5 (Schoof, see Lemma, Section 12 of [100])**.** *Given is a lattice L of rank n generated by the basis $\boldsymbol{B} = [\boldsymbol{b}^i]_{i=1}^{n-1}$, and a vector $\boldsymbol{x} \in \mathbb{R}^n$. Assume that $\boldsymbol{B}$ is a c-reduced basis for L, with $c \geq 1$, and let $\boldsymbol{b}^{1*}, \ldots, \boldsymbol{b}^{n*}$ be the Gram-Schmidt vectors corresponding to $\boldsymbol{B}$. Let $\lambda_1, \ldots, \lambda_n$ be the coefficients in the unique expression of $\boldsymbol{x}$ in terms of the basis vectors $\boldsymbol{b}^j$, i.e., $\boldsymbol{x} = \sum_{j=1}^n \lambda_j \boldsymbol{b}^j$. Then,*

$$|\lambda_j| \leq \left(\frac{3\sqrt{c}}{2}\right)^{n-j} \cdot \frac{\|\boldsymbol{x}\|}{\|\boldsymbol{b}^{j*}\|} \text{ for } j \in [n].$$

We are now ready to state our result.

Theorem 6. *Given a_0, $\boldsymbol{a} = (a_1, \ldots, a_n)$, a basis $\hat{\boldsymbol{B}}$ of $ker_{\Lambda}\mathbf{1}$, and a vector $\bar{\boldsymbol{x}} \in \mathbb{Z}^n$ satisfying $\boldsymbol{a}\bar{\boldsymbol{x}} = a_0$, assume that*

- *$a_0, a_1, \ldots, a_n > 0$,*
- *$gcd(a_1, \ldots, a_n) = 1$,*
- *$\hat{\boldsymbol{B}}$ is c-reduced with $c \geq 1$.*

Then, $X_{LP}^{\boldsymbol{\lambda}} = \{\boldsymbol{\lambda} \in \mathbb{Z}^{n-1} \mid -\boldsymbol{B}\boldsymbol{\lambda} \leq \bar{\boldsymbol{x}}\}$ with $\boldsymbol{B} = \boldsymbol{D}^{-1}\hat{\boldsymbol{B}}$ is contained in a box with volume at most

$$\left(\frac{3\sqrt{c}}{2}\right)^{\frac{(n-1)(n-2)}{2}} \cdot \frac{2^{\frac{n-1}{2}}}{\sqrt{n}} \cdot \frac{a_0^{n-1}}{\prod_{i=1}^n a_i}. \tag{4.20}$$

Proof. Let $\boldsymbol{x}^{(1)}$ and $\boldsymbol{x}^{(2)}$ be two arbitrary vectors in X_{LP}, and consider their representation in terms of $\boldsymbol{B}$, namely $\boldsymbol{x}^{(1)} = \bar{\boldsymbol{x}} + \sum_{j=1}^{n-1} \lambda_j^{(1)} \boldsymbol{b}^j$, $\boldsymbol{x}^{(2)} = \bar{\boldsymbol{x}} + \sum_{j=1}^{n-1} \lambda_j^{(2)} \boldsymbol{b}^j$. We can apply Schoof's lemma to the vector

$$\boldsymbol{D}(\boldsymbol{x}^{(1)} - \boldsymbol{x}^{(2)}) = \sum_{j=1}^{n-1} (\lambda_j^{(1)} - \lambda_j^{(2)}) \boldsymbol{D}\boldsymbol{b}^j = \sum_{j=1}^{n-1} (\lambda_j^{(1)} - \lambda_j^{(2)}) \hat{\boldsymbol{b}}^j .$$

This yields, for $j = 1, \ldots, n-1$,

$$|\lambda_j^{(1)} - \lambda_j^{(2)}| \leq \left(\frac{3\sqrt{c}}{2}\right)^{n-1-j} \cdot \frac{\|\boldsymbol{D}(\boldsymbol{x}^{(1)} - \boldsymbol{x}^{(2)})\|}{\|\hat{\boldsymbol{b}}^{j*}\|} .$$

Here $\hat{\boldsymbol{b}}^{j*}$, $j = 1, \ldots, n-1$, are the Gram-Schmidt vectors associated with the basis $\hat{\boldsymbol{B}}$. Notice that the vectors $\boldsymbol{D}\boldsymbol{x}^{(1)}$ and $\boldsymbol{D}\boldsymbol{x}^{(2)}$ are in the regular simplex X_{LP}^D and therefore $||\boldsymbol{D}\boldsymbol{x}^{(1)} - \boldsymbol{D}\boldsymbol{x}^{(2)}|| \leq \sqrt{2} \cdot a_0$. Using this and the fact that $\boldsymbol{x}^{(1)}$ and $\boldsymbol{x}^{(2)}$ were chosen arbitrarily, we can obtain an upper bound for the volume of $X_{LP}^{\boldsymbol{\lambda}}$

$$\begin{aligned} V(X_{LP}^{\boldsymbol{\lambda}}) &\leq \prod_{j=1}^{n-1} \left(\frac{3\sqrt{c}}{2}\right)^{n-1-j} \cdot \frac{\sqrt{2} \cdot a_0}{\|\hat{\boldsymbol{b}}_j^*\|} \\ &= \left(\frac{3\sqrt{c}}{2}\right)^{\frac{(n-1)(n-2)}{2}} \cdot 2^{\frac{n-1}{2}} \cdot \frac{a_0^{n-1}}{\prod_{j=1}^{n-1} \|\hat{\boldsymbol{b}}_j^*\|} . \end{aligned} \tag{4.21}$$

It is well-known that $\prod_{j=1}^{n-1} \|\hat{\boldsymbol{b}}_j^*\|$ is an expression for $d(\ker_\Lambda \mathbf{1})$ [35], so we only need to obtain an expression for $d(\ker_\Lambda \mathbf{1})$ in terms of the input.

The lattice $\ker_\Lambda \mathbf{1}$ is a pure sublattice of Λ. From known lattice formulae (see (4.3), (4.4), and (4.5)) we obtain

$$\begin{aligned} d(\ker_\Lambda \mathbf{1}) &= \frac{d(\Lambda)}{d(\Lambda/\ker_\Lambda \mathbf{1})} = d(\Lambda) \cdot d((\Lambda/\ker_\Lambda \mathbf{1})^*) \\ &= d(\Lambda) \cdot d((\ker_\Lambda \mathbf{1})^\perp) \end{aligned} \tag{4.22}$$

with $(\ker_\Lambda \mathbf{1})^\perp = \{\boldsymbol{x} \in \Lambda^* \mid \langle \boldsymbol{x}, \ker_\Lambda \mathbf{1} \rangle = 0\}$. The determinant of the lattice Λ is equal to

$$d(\Lambda) = \det(\boldsymbol{D}) = \prod_{j=1}^{n} a_j .$$

Since $\gcd(a_1, \ldots, a_n) = 1$, the lattice $(\ker_\Lambda \mathbf{1})^\perp$ is the lattice generated by the vector of all ones, i.e., $(\ker_\Lambda \mathbf{1})^\perp = \mathbb{Z}\mathbf{1}$ having determinant equal to

$$d((\ker_{\mathbb{Z}} \mathbf{1})^\perp) = \langle \mathbf{1}, \mathbf{1} \rangle^{1/2} = \sqrt{n} .$$

Expression (4.22) has now become:

$$d(\ker_\Lambda \mathbf{1}) = d(\Lambda) \cdot d((\ker_\Lambda \mathbf{1})^\perp) = \prod_{j=1}^{n} a_j \cdot \sqrt{n} .$$

We now substitute $\prod_{j=1}^{n-1} \|\hat{\boldsymbol{b}}_j^*\|$ in Expression (4.21) for $\prod_{j=1}^{n} a_j \cdot \sqrt{n}$ and obtain the desired result. □

If we compare expressions (4.18) and (4.20) we can make the following remarks. The first two factors in the bound on $V(X^{\boldsymbol{\lambda}}_{LP})$ are due to the estimate that one gets from the LLL-reduction (see the proof given in Lenstra [100]). In practice, the LLL-reduction estimates typically turn out much better than the theoretical bounds. In the last factor of the bound we have lost one power of the right-hand side a_0. It is not so clear how this gain compares with the first two factors. We investigate this in the computational study presented in Section 4.4, where we also study the question of how the basis used in the proof of Theorem 6 compares computationally to the regular AHL and KP reformulations.

4.4. COMPUTATIONAL STUDY

4.4.1. INSTANCES AND SETUP

We perform the comparison using seven sets of synthetic instances from the literature. A summary of their properties and a description of the models can be found in the appendix of this chapter. These instances were chosen to represent a range of sizes, as well as a range of variable and constraint types. Four of the collections are composed of single-row feasibility problems and three are multi-row binary problems. The single-row instances struct_s and struct_b have been generated such that the lattice $\ker_{\mathbb{Z}} \boldsymbol{a}$ has an $(n-2)$-dimensional sublattice with small determinant d, whereas $d(\ker_{\mathbb{Z}} \boldsymbol{a})$ is large. This implies that LLL-reduction will yield a basis with $n-2$ relatively short vectors and one long vector. In contrast, nostruct_s and nostruct_b are composed of instances with coefficients of the same order of magnitude as struct_s and struct_b, yielding a lattice determinant of the same order of magnitude, but with no special encoded structure. For all single-row instances, the right-hand side coefficient is the Frobenius number[2] [11], which makes them infeasible. Market split (MS) instances have been shown to benefit from the AHL reformulation in past work, see Aardal et al. [3]. Generalized assignment problems (GAP) and combinatorial auctions (CA) have not been tested in this context before. Apart from these seven synthetic benchmarks, we expand our computational study to the MIPLIB collection [68] (see Section 4.4.6). This collection is comprised of instances from a wide range of applications and with diverse sizes and constraint types.

In our computations, LLL-reductions were performed using the NTL library [140] using a reduction coefficient $y = 0.99$ (see Definition 7) in all cases except AHL$^{\text{low}}$ where we used $y = 0.3$. Each instance is solved 5 times with different randomization seeds. We use the solver SCIP v.8.0.1 [27] with default parameters. This means that we do not provide the solver with information about which variable to branch on in the reformulations, even though we have reason to believe that the last coordinate directions should be preferred. We set a time limit of one hour. Timeouts are indicated with the symbol '>'. We refer to the appendix for extended results. Code for reproducing all experiments is available at [133].

4.4.2. EXPERIMENTS WITH SINGLE-ROW INSTANCES

We compare the original formulation with four reformulations:

[2] Given positive integers $\{a_1, \ldots, a_n\}$, the Frobenius number F is the largest integer for which there is no $\boldsymbol{x} \in \mathbb{Z}^n_{\geq 0}$ such that $a_1 x_1 + a_2 x_2 + \ldots + a_n x_n = F$.

Table 4.1: Geometric mean of the number of nodes over 30 instances and 5 randomization seeds (lower is better). We compare the original formulation with the four proposed reformulations. The same results are shown in Figure 4.3.

Instance	Original	AHL	AHL$^{\boldsymbol{D}}$	AHL$^{\text{low}}$	KP
struct_s	$> 10^7$	12.70	10.81	12.63	19.19
struct_b	$> 10^7$	1.28	1.27	1.31	1.21
nostruct_s	121,456	97.10	68.64	90.43	128.98
nostruct_b	59,901	701.15	572.03	1779.06	566.99
MS	398,742	685.03	-	4111.48	1545.95
GAP	893.37	52.15	-	72.52	122.92
CA	11.26	21.03	-	95.08	16.14

Table 4.2: Geometric mean of the solving time (in seconds) over 30 instances and 5 randomization seeds (lower is better). Reduction time is not included, but we report it in Table 4.4.

Instance	Original	AHL	AHL$^{\boldsymbol{D}}$	AHL$^{\text{low}}$	KP
struct_s	> 3600	0.04	0.03	0.04	0.04
struct_b	> 3600	0.06	0.05	0.06	0.07
nostruct_s	21.74	0.12	0.10	0.11	0.10
nostruct_b	6.62	0.61	0.45	1.02	0.53
MS	37.98	0.81	-	2.40	1.64
GAP	3.77	1.05	-	1.11	1.39
CA	3.41	4.68	-	28.23	3.99

- (AHL): the AHL reformulation with $\boldsymbol{B}$ a reduced basis of $\ker_{\mathbb{Z}} \boldsymbol{a}$.
- (AHL$^{\text{low}}$): AHL-reformulation with low quality reduction coefficient $y = 0.3$.
- (AHL$^{\boldsymbol{D}}$): AHL reformulation with $\boldsymbol{B} = \boldsymbol{D}^{-1}\hat{\boldsymbol{B}}$ and $\hat{\boldsymbol{B}}$ a reduced basis of $\ker_{\Lambda} \mathbf{1}$.
- (KP): the KP reformulation.

We reiterate that the reduction coefficient is set to $y = 0.99$ unless otherwise stated. We report the averaged number of branch-and-bound nodes and solving time in the first four rows of Table 4.1 and Table 4.2, respectively. For convenience, the data in Table 4.1 can also be visualized in Figure 4.3. The results per instance can be found in Tables 4.12–4.15 in this chapter's appendix.

Our computational results show that each of the reformulations yields a remarkably easier problem. This is particularly apparent for struct_s and struct_b, which go from being unsolvable within the time limit to being remarkably easy. Notice that with higher dimension, the Frobenius number decreases quite substantially which can result in relatively easier instances. For this reason, we observe that the large structured instances become trivial for all reformulations, with most of them being solved without branching, see Table 4.14. For all structured instances, AHL$^{\boldsymbol{D}}$ gives the best results, both in terms of

Table 4.3: Average logarithm of the volume of the smallest box, oriented in the coordinate directions, that contains the LP-relaxation of the respective reformulation. We calculate this by maximizing and minimizing the value of each variable over the LP relaxation. The last column shows the logarithm of the bound in (4.20), split into two terms: (i) the logarithm of the first factor and (ii) the logarithm of the remaining factors.

Instance	Original	AHL	AHL$^{\boldsymbol{D}}$	AHL$^{\text{low}}$	KP	Thm. 6
struct_s	62.35	50.84	46.37	50.21	53.57	20.0+50.0
struct_b	412.91	449.40	401.72	447.19	449.79	2697.2+464.0
nostruct_s	25.39	17.57	14.48	17.17	18.51	20.0+16.8
nostruct_b	56.25	85.65	77.05	76.35	86.39	2697.2+110.9

nodes and time. All reductions, however, work well, even AHL$^{\text{low}}$, so it does not pay off significantly to spend time on a high-quality reduction.

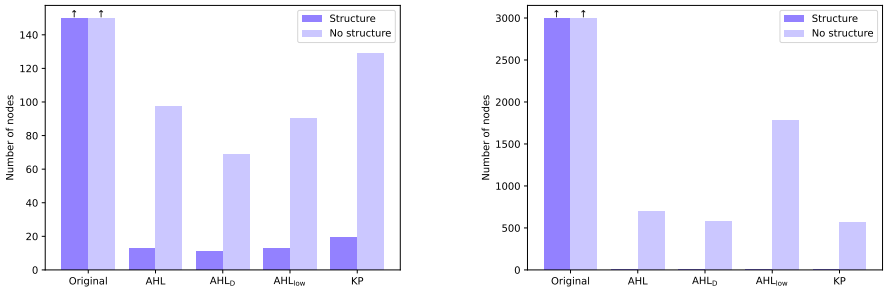
For the non-structured instances the results change depending on the instance size. For the smaller instances, AHL^{D} yields the smallest trees, followed by AHL$^{\text{low}}$ and AHL. For the larger instances the results indicate that KP gives the lowest average number of nodes, but AHL$^{\boldsymbol{D}}$ is faster. We also observe that, in particular for the smaller instances, AHL$^{\text{low}}$ yields sparser matrices than AHL, due to a smaller number of column operations on the basis matrix. Yet, they are both able to find the overall best direction (see Section 4.4.4). The computational results indicate that the solver is not always able to make use of the higher-quality basis provided by AHL and in fact a sparser constraint matrix can be beneficial.

Tables 4.1 and 4.2 suggest that the effect of the dimensionality reduction provided by all AHL variants is more pronounced for lower-dimensional problems, whereas in higher dimension the shape transformation is the main driver of improvements. This is also supported by the volume results reported in Table 4.3 (see also Figure 4.4). Here we show the average logarithm of the volume of the smallest box, oriented in the coordinate directions, that contains the LP-relaxation of the respective reformulation. AHL^{D} yields the smallest volume for all but the larger non-structured instances. For these instances it is in fact the original formulation that has the smallest relaxation volume. This means that the improvements in Table 4.1 can only be a consequence of the transformed shape. Table 4.3 also shows that the bound in Theorem 6 is far from being tight, due to the constant arising from the worst-case performance of LLL.

4.4.3. MULTI-ROW INSTANCES

We test the same formulations as in the previous section, except for AHL$^{\boldsymbol{D}}$, which is only valid for single-row problems. Results are shown in Tables 4.1 and 4.2, last three rows. For results per instance, see Tables 4.16–4.18 and Figure 4.6 in this chapter's appendix.

The market split instances are non full-dimensional instances with a dense constraint matrix with integer entries between 0 and 100. All reformulations perform sig-



(a) Small instances ($n = 10$)

(b) Big instances ($n = 100$)

Figure 4.3: Geometric mean of the number of nodes over 30 instances and 5 randomization seeds (lower is better). We compare the original formulation with the four proposed reformulations. The same results are presented in Table 4.1.

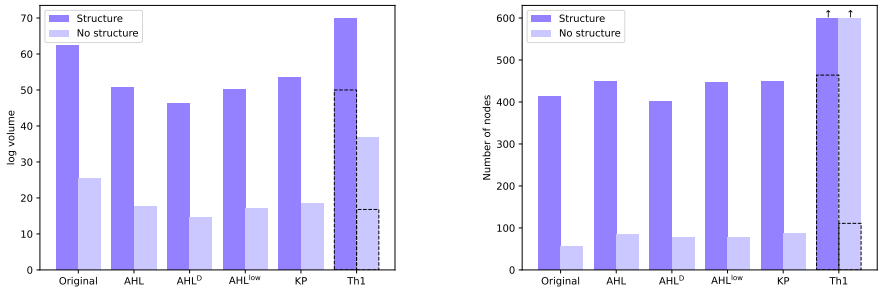
Table 4.4: Average reduction times (in seconds) of the multi-row instances. The reduction time of single-row instances is negligible.

Instance	AHL	AHL^{low}	KP
Market split	$2 \cdot 10^{-3}$	$3 \cdot 10^{-3}$	$2 \cdot 10^{-3}$
Generalized assignment problem	21.4	21.2	2.5
Combinatorial auctions	172.6	168.6	3.0

nificantly better than the original formulation, with AHL being the best. This can be an indication that dimension-reduction plays a role. We can also conclude that for these instances the quality of the reduction matters.

The GAP instances, in contrast, are binary, full-dimensional, and part of the constraint matrix contains numbers that are different from 0 and 1. This may result in an LP-relaxation having a "non-regular" shape. Again we see that all reformulations perform better than the original formulation. This supports the idea that constraint branching can be beneficial even for (some types of) binary instances. It is surprising that both AHL-reformulations perform much better than KP, since one would expect that AHL and KP would perform more or less equally well. The only difference between the two in practice is that AHL reduces $(\boldsymbol{A}, \boldsymbol{a}_0)$, whereas KP reduces only $\boldsymbol{A}$ (see Section 4.3 and the appendix).

The CA instances are full-dimensional and have 0-1 entries in the constraint matrix. Here we observe that the original formulation performs the best, even though both KP and AHL perform reasonably well in comparison. So, with no dimension reduction and a combinatorial problem structure there seems to be no gain in reformulating.



(a) Small instances ($n = 10$)

(b) Big instances ($n = 100$)

Figure 4.4: Average logarithm of the volume of the smallest box, oriented in the coordinate directions, that contains the LP-relaxation of the respective reformulation. We calculate this by maximizing and minimizing the value of each variable over the LP relaxation. 'Th1' shows the logarithm of the bound in (4.20). We further show the contribution of the last two terms with dashed lines. This is, the result of ignoring the term coming from Schoof's lemma.

Table 4.5: Average norm of the projection of $\boldsymbol{b}^1$ onto span$\{\boldsymbol{e}^2, \ldots, \boldsymbol{e}^n\}$ (lower is better). If the norm is zero, the (originally) last coordinate direction is a thin branching direction in the sense of the Lovàsz-Scarf algorithm.

Instance	AHL	AHL$^{\boldsymbol{D}}$	AHL$^{\text{low}}$	KP
struct_s	0.0	0.0	0.0	15.10
struct_b	0.0	0.0	0.0	6.63
nostruct_s	0.51	0.0	0.38	3.80
nostruct_b	0.0	0.0	0.0	5.40
MS	2.83	-	0.0	3.91
GAP	0.0	-	0.0	0.0
CA	0.0	-	0.0	0.0

4.4.4. COMPARISON WITH LOVÁSZ-SCARF

The generalized basis reduction algorithm of Lovász and Scarf gives us an approximation of a direction in which the LP relaxation is thin (see Section 4.2.3). The reformulations we study can be regarded as a heuristic way of obtaining thin directions. In particular, by construction, the last coordinate direction should indicate such a direction. We run the generalized basis reduction algorithm on the polytope X^λ, where the polytope is generated by the reformulations AHL, AHL^D, AHL$^{\text{low}}$ and KP, to investigate if indeed this last coordinate direction coincides with the direction given by the Lovász-Scarf (LS) algorithm.

For this, we reverse the order of the variables such that the last one becomes the first. We run our own implementation of the generalized basis reduction and retrieve the first basis vector $\boldsymbol{b}^1$. If $\boldsymbol{b}^1 = \boldsymbol{e}^1$ we can conclude that the (originally) last coordinate is an approximation a direction in which the LP relaxation is thin. We report the norm of the projection of $\boldsymbol{b}^1$ onto span$\{\boldsymbol{e}^2, \ldots, \boldsymbol{e}^n\}$ (the orthogonal complement of $\boldsymbol{e}^1$). This is, we

Table 4.6: Vanilla SCIP. We report the geometric mean of the number of nodes over 30 instances and 5 randomization seeds when deactivating presolve, cutting, conflict analysis and primal heuristics. We compare the original formulation with the four proposed reformulations.

Instance	Original	AHL	AHL$^{\boldsymbol{D}}$	AHL$^{\text{low}}$	KP
struct_s	$> 10^7$	37.70	22.83	32.67	82.96
struct_b	$> 10^7$	2.67	2.10	2.84	43.18
nostruct_s	141,377	197.3	129.9	193.5	293.1
nostruct_b	76,081	1,400	915.2	1,238	1,517
MS	1,041,742	989.5	-	6,493	2,101
GAP	$> 378,502$	308.0	-	354.6	$> 3,541$
CA	21.08	26.39	-	100.2	27.19

report $\sqrt{\sum_{i=2}^{2} b_i^1}$. The average over all instances is shown in Table 4.5.

These results show that AHL, in all its variants, is successful in finding such direction in many cases, and when it does not, it finds a close direction. In contrast, KP is in most cases unable to find a Lovász-Scarf direction, especially when the problem at hand is not full dimensional.

4.4.5. COMPUTATIONAL EXPERIMENTS WITH VANILLA SCIP

We repeat the experiment presented in Table 4.1 with a vanilla version of SCIP v.8.0.1: presolve, cutting, conflict analysis and primal heuristics are deactivated. This allows us to isolate the effect of branching. These additional results, shown in Table 4.6, confirm the findings discussed in in the previous sections. Additionally, we see that the benefits of AHL$^{\boldsymbol{D}}$ and the dimensionality reduction in single-row instances are even more apparent under these settings. Here, of all the reformulations, KP is the most affected by the lack of additional solver components.

4.4.6. COMPUTATIONAL EXPERIMENTS ON MIPLIB

We extend our computational study to instances in MIPLIB [68]. This instance collection is a standard test set comprised of real-world instances with varied structures. We consider a subset of these instances based on the following criteria:

- Instances with no more than 1000 variables or constraints. For larger instances the reduction process could be excessively long.
- Pure integer programs. Even though one can devise a procedure to take care of continuous variables, we focus our study on instances with only integer variables.
- Instances with integer coefficients. Again, one can conceive a procedure to transform all instances into integer form but for the purpose of this study we focus on instances that are already in this form.

After applying these criteria, we are left with 20 instances. Instance queens-30.mps had to also be eliminated from considereation because the reduction process failed to finish within the time limit of one hour, even when the reduction quality coefficient was

Table 4.7: Subset of MIPLIB instances used for the experiments.

Instance	Variables	Constraints
ej	3	1
enlight11	242	121
enlight4	32	16
enlight8	128	64
enlight9	162	81
enlight_hard	200	100
f2gap40400	400	40
fhnw-sq2	650	91
gt2	188	29
neos859080	160	164
p0201	201	133
p2m2p1m1p0n100	100	1
pb-market-split8-70-4	71	17
ponderthis0517-inf	975	78
stein15inf	15	37
stein45inf	45	332
stein9inf	9	14
supportcase14	304	234
supportcase16	319	130

decreased. The remaining 19 instances are shown in Table 4.7.

We reformulate each of the instances using the AHL ($y = 0.99$) and the KP reformulations. We solve the original instance and the two reformulations using default SCIP settings using 5 different random seeds and with a time limit of one hour. We report the results in Table 4.8. It is important to note that the presolve settings can have an influence on the results reported in this section.

We observe that for fhnw-sq2 and pb-market-split8-70-4 no formulation resulted in a solved problem within the time limit. The latter is in fact an instance for which no solution is known. Optimality gaps can also not be reported, given that fhnw-sq2 is a feasibility problem and that for pb-market-split8-70-4 no feasible solution was found throughout the whole search. We also note that the AHL reformulation process detected the infeasibility of instances enlight4, enlight9 and enlight11, so no call to the solver needed to be made.

When comparing AHL to the original formulation we can distinguish two groups of instances. For instances gt2, p0201, stein15inf and stein45inf the reformulation produced larger trees (we consider differences of less that 1 node not statistically significant). On the other hand, for instances ej, neos859080, p2m2p1m1p0n100 and ponderthis0517-inf the reformulation was trivial to solve, while the original problem was difficult or im-

Table 4.8: Results on a selection of MIPLIB instances. We report the geometric mean of the number of nodes over 5 random runs for the original formulation (Original), the AHL reformulation (AHL) and the KP reformulation (KP). The symbol > indicates that the solver failed to finish within the time limit (1h).

Instance	Original	AHL	KP
ej	124876.57	0.00	0.00
enlight11	1.0	0.00	>
enlight4	1.0	0.00	1.0
enlight8	1.00	0.00	6529.06
enlight9	1.0	0.00	65636.91
enlight_hard	1.00	0.00	18731.40
f2gap40400	1.00	1.74	1.00
fhnw-sq2	>	>	>
gt2	1.00	2703.86	104.65
neos859080	1562.77	0.0	>
p0201	7.46	123.80	74.27
p2m2p1m1p0n100	>	1.0	1.0
pb-market-split8-70-4	>	>	>
ponderthis0517-inf	>	0.0	>
stein15inf	21.71	46.50	21.77
stein45inf	698.03	724.68	708.69
stein9inf	1.00	1.00	1.00
supportcase14	1.00	1.89	1.32
supportcase16	1.15	2.00	1.15

Table 4.9: Comparison of AHL and KP to the original formulation. Green indicates better performance, orange indicates worse, and white is a tie.

Instance	AHL	KP
ej		
p2m2p1m1p0n100		
ponderthis0517-inf		
neos859080		
gt2		
p0201		
stein15inf		
stein45inf		
enlight11		
enlight8		
enlight9		
enlight_hard		

possible to solve within the time limit.

At the same time, KP produces larger trees for gt2, p0201, and stein45inf, compared to the original. Yet, the deterioration in performance is not as acute as with AHL. However, there is another group of instances where KP does worse, namely neos859080, enlight11, enlight8 and enlight9, where the performance is considerably worse. Note that all enlight instances have equality constraints. KP beats the original formulation only on two instances: ej and p2m2p1m1p0n100.

Table 4.9 compares the two reformulations to the original on instances where at least one does not tie. Compared to KP, AHL does better than the original on more instances and does worse than the original on fewer instances. There is no one instance where KP is better than the other two formulations. However, as we noted before, on those instances where both AHL and KP perform worse than the original, KP's performance is better than AHL.

4.4.7. MEASURING POTENTIAL

Observing the results presented in the previous sections it is natural to ask if it is possible to find some common characteristics of the instances that benefit from reformulations. Finding such characteristics could allow us to detect the potential of reformulating before the solve. For the AHL reformulation, the presence of equality constraints is a natural candidate. We further test three metrics as potential proxies. These metrics are based on the constraint matrix $\boldsymbol{A} \in \mathbb{Z}^{m\times n}$ and the matrix $\boldsymbol{B} \in \mathbb{Z}^{n\times l}$ resulting from the AHL reformulation defined as in Def 13. In particular, let $\alpha_{max} = \max\{\boldsymbol{A}_{ij} \mid i \in [m], j \in [n]\}$ and $\alpha_{min} = \min\{\boldsymbol{A}_{ij} \mid i \in [m], j \in [n]\}$. We define the following metrics:

$$m_1 := \frac{||\boldsymbol{b}^l||}{||\boldsymbol{b}^1||} \tag{4.23}$$

$$m_2 := \frac{\sum_{i=1}^{n}\sum_{j=1}^{l} \mathbb{1}_{B_{ij}\neq 0}}{n \cdot l} \tag{4.24}$$

$$m_3 := f(\alpha_{max}) - f(\alpha_{min}) \tag{4.25}$$

with

$$f(x) = \begin{cases} -\log_{10}(|x|) & \text{if} \quad x < 0 \\ 0 & \text{if} \quad x = 0 \\ \log_{10}(x) & \text{if} \quad x > 0\,. \end{cases}$$

We note that metrics m_1 and m_2 can only be computed after the reformulation process has been completed, whereas m_3 can be quickly computed without prior steps. The interpretation of these metrics is the following. Metric m_1 measures the relative length of the last basis vector with respect to the first one, m_2 measures the density of $\boldsymbol{B}$, and m_3 measures the range (in orders of magnitude) of the coefficients in $\boldsymbol{A}$.

Figure 4.5 shows the three metrics for all benchmarks (averaged over all instances, or per instance in the case of MIPLIB). On Table 4.10 we further show the percentage of equality constraints after presolve of the MIPLIB instances. We only show results for MIPLIB instances for which the original formulation and AHL performed differently. While

Table 4.10: Percentage of equality constraints of a selection of instances. The value is calculated after presolve.

Instance	
ej	100%
p2m2p1m1p0n100	0%
ponderthis0517-inf	33%
neos859080	32%
gt2	0%
p0201	19%
stein15inf	0%
stein45inf	0%

m_2 seemed initially like a plausible indicator of potential, we observed that its value is somewhat correlated with the instance size: smaller instances produce denser bases. This could be a consequence of the effect pointed out by Aardal and von Heymann [2]. As expected, large m_1 value seems like a reliable indicator of potential. However, a small value does not rule out good performance of AHL. The metric m_3 produces some mixed results. For example, using m_3, the instances gt2 and p0201 could become false positives. Notice that, for instance ponderthis0517-inf, none of the metrics take a high value. It is in fact a binary instance with 0-1 constraints. Yet, about a third of the constraints are equality constraints, which on its own can be a strong indication. Altogether, these results confirm our previous observations that instances with equality constraints and non-binary constraint matrices have high potential of benefiting from the reformulation.

4.5. DISCUSSION

In this chapter, we have treated the topic of lattice-based reformulations for Integer Programming. We studied the applicability of several such reformulations from different points of view: from a bound on the volume of the reformulated relaxation to a computational study that shed light on the performance of the reformulations for different types of instances. Our results demonstrate that the reformulations can be valuable tools to solve IPs. We show that they are of particular interest for instances that are non full-dimensional, even in the case of binary programs. We also provide positive results for instances where the constraint matrix is not combinatorial.

From a broader perspective, the studied problem reformulations can be seen as a heuristic way to obtain good branching disjunctions. After all, the reformulated variables can be interpreted as hyperplanes in the original space. The results presented in this chapter present a new avenue of research for branching on general disjunctions. This is an important viewpoint because, while the theoretical results of Lenstra [101] and Lovász and Scarf [108] tell us to branch on general disjunctions, most IP solvers only implement single-variable branching schemes. Some reasons are that (i) theoretically strong disjunctions are computationally costly to find and difficult to implement into standard branch-and-bound schemes, and (ii) not all instance types benefit from using

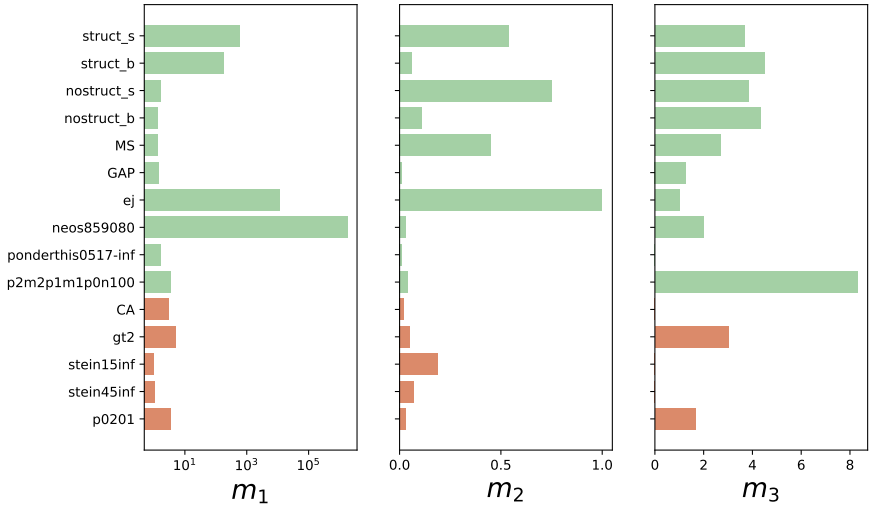


Figure 4.5: Value of m_1, m_2 and m_3 (see Eqs. 4.23, 4.24, 4.25) for all benchmarks. The results are averaged for all benchmarks, except for MIPLIB instances, for which we report the values per-instance. Instances displayed in green are those for which the AHL reformulation showed better performance. Instances displayed in red are those for which the opposite is true.

them. In this chapter, we have shown that the branching directions provided by the reformulations often coincide with the Lovász and Scarf direction. We have also pointed out some characteristics that can help us identify instances that would benefit from this scheme. The applicability of the lattice reformulations goes beyond what was originally expected, with encouraging results for binary instances, where single-variable disjunctions already provide theoretically "thin directions".

In our study, we solve the reformulated instances by standard variable branching, which, as we pointed out, is mathematically equivalent to branching on general disjunctions in the original space. However, in practice, other solver components have an impact on the performance. As an additional experiment to isolate the effect of branching we turned off presolve, cutting, heuristics and conflict analysis. These experiments support once again our findings that the reformulated variables provide good branching directions.

Overall, our results point to the great potential of the AHL and KP reformulations for tackling problems where single-variable branching fails. Still, the underlying LLL procedure (though cheaper than other schemes such as that of Lovász and Scarf [108]) can be computationally expensive as the number of variables grows (see Table 4.4 and the selection criteria for Section 4.4.6). Future directions of research can be to identify a subset of constraints to reformulate, to then add the reformulated variables as branching direc-

tions. Because of the overhead that the reformulation can cause, it is unlikely that such a procedure will become the standard for solving IPs. However, the reformulation fits perfectly into the broader perspective of dynamic solvers, which can switch to alternative strategies mid-solve after standard approaches have been tested and shown to fail.

Table 4.11: Instance collections used in our computations.

Name	Number	n	m	Ref
struct_s	30	10	1	[1]
nostruct_s	30	10	1	[1]
struct_b	30	100	1	[1]
nostruct_b	30	100	1	[1]
MS	30	30	4	[44]
GAP	30	600	606	[61]
CA	30	500	100	[102]

4.6. APPENDIX

4.6.1. EQUIVALENCE BETWEEN AHL AND KP IN THE FULL-DIMENSIONAL CASE

We demonstrate this equivalence on a set with only upper bounds, but adding lower bounds follows easily. Consider the system $\boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{a}_0$. We add slack variables and obtain $\boldsymbol{A}\boldsymbol{x} + \boldsymbol{I}\boldsymbol{s} = \boldsymbol{a}_0$. The AHL-reformulation now needs, as a starting point, a vector $(\bar{\boldsymbol{x}}, \bar{\boldsymbol{s}})^\mathsf{T}$ satisfying $\boldsymbol{A}\bar{\boldsymbol{x}} + \boldsymbol{I}\bar{\boldsymbol{s}} = \boldsymbol{a}_0$ and a basis for the lattice $\ker_{\mathbb{Z}}(\boldsymbol{A}\ \boldsymbol{I})$. We may choose $(\bar{\boldsymbol{x}},\ \bar{\boldsymbol{s}})^\mathsf{T} = (\boldsymbol{0},\ \boldsymbol{a}_0)$ and the lattice basis

$$\begin{pmatrix} \boldsymbol{I} \\ -\boldsymbol{A} \end{pmatrix}. \tag{4.26}$$

Reducing the basis (4.26) yields the AHL-reformulation, which is also precisely what Krishnamoorthy and Pataki do.

4.6.2. INSTANCE MODELS

SINGLE-ROW INSTANCES

For a given number of variables n, find a vector $\boldsymbol{x} \in \mathbb{Z}^n_{\geq 0}$ that satisfies $\boldsymbol{a}\boldsymbol{x} = a_0$. The number a_0 is chosen to be the Frobenius number corresponding to $\boldsymbol{a}$. This number is computed following the procedure described in [1]. For the structured instances, $\boldsymbol{a}$ is generated as $\boldsymbol{a} = M\boldsymbol{p} + N\boldsymbol{r}$, with $M \in [10000, 20000]$, $N \in [1000, 2000]$, $\boldsymbol{p} \in [1, 10]^n$ and $\boldsymbol{r} \in [-10, 10]^n$ sampled uniformly. In the case of instances with no structure, the vector $\boldsymbol{a}$ is sampled uniformly in $[10000, 15000]^n$.

MARKET SPLIT (MS)

For a given number m of constraints and $n = 10 \cdot (m-1)$ variables, we solve the feasibility problem of finding a vector in $\{\boldsymbol{x} \in \{0,1\}^n | \boldsymbol{A}\boldsymbol{x} = \boldsymbol{a}_0\}$. We generate the coefficients of $\boldsymbol{A}$ uniformly at random in the range $[0, 99] \cap \mathbb{Z}$ and we set $(\boldsymbol{a}_0)_i = \frac{1}{2}\sum_{j=1}^n \boldsymbol{A}_{ij}$.

GENERALIZED ASSIGNMENT PROBLEM (GAP)

Given n items with respective prices $\{p_j\}_{j=1}^n$ and weights $\{w_j\}_{j=1}^n$, and m knapsacks with capacities $\{c_i\}_{i=1}^m$, the generalized assignment problem consists in placing a number of items in each of the knapsacks such that the price of the selected items is maximized,

while the capacity of the knapsacks is not exceeded by the total weight of the items therein. Formally:

$$\begin{aligned} \text{maximize } & \sum_{i=1}^{m}\sum_{j=1}^{n} p_j x_{ij} \\ \text{subject to } & \sum_{j=1}^{n} w_j x_{ij} \le c_i, \quad i = 1,\dots,m \\ & \sum_{i=1}^{m} x_{ij} \le 1, \quad j = 1,\dots,n \\ & x_{ij} \in \{0,1\} \ \forall i,j \end{aligned}$$

where each variable x_{ij} represents the decision of placing item j in knapsack i.

COMBINATORIAL AUCTIONS (CA)

For m items, we are given n bids $\{\mathscr{B}_j\}_{j=1}^{n}$. Each bid $\mathscr{B}_j$ is a subset of the items with an associated bidding price p_j. The associated combinatorial auction problem is of the following form:

$$\begin{aligned} \text{maximize } & \sum_{j=1}^{n} p_j x_j \\ \text{subject to } & \sum_{j:i\in\mathscr{B}_j} x_j \le 1, \quad i = 1,\dots,m \\ & x_j \in \{0,1\} \ \ j = 1,\dots,n \end{aligned}$$

where x_j represents the action of choosing bid $\mathscr{B}_j$.

4.6.3. EXTENDED RESULTS

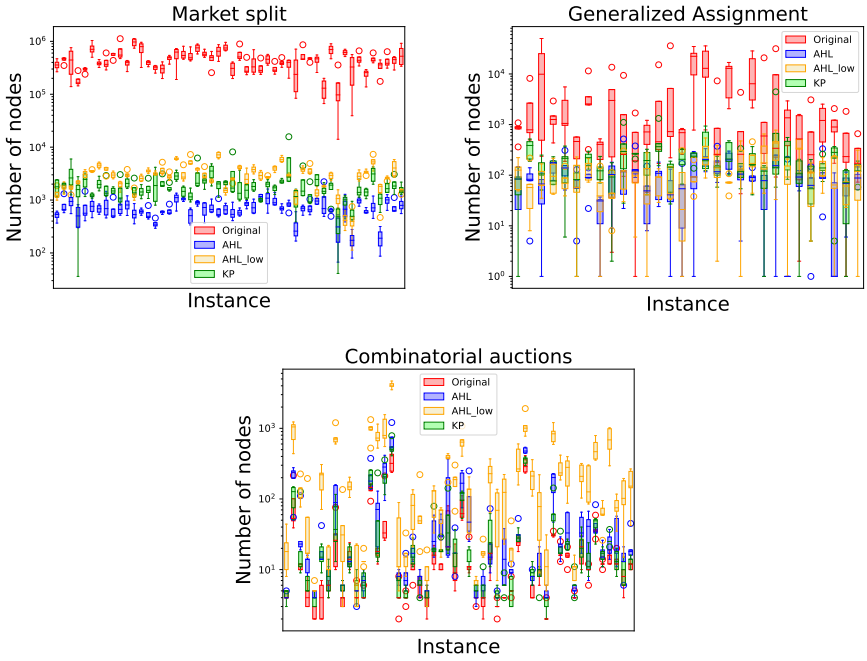


Figure 4.6: Number of explored nodes per instance for each of the formulations (lower is better). Values are averaged over 5 runs with different randomization seeds. Instances are of the type market split (top left), generalized assignment (top right) and combinatorial auctions (bottom).

Table 4.12: Extended results on small structured single-row instances. We compare the original formulation with the four proposed reformulations. We report the geometric mean of number of nodes over 5 randomization seeds.

Instance	Original	AHL	AHL^{D}	AHL$^{\text{low}}$	KP
struct_s_1	$> 10^7$	10.57	**1.00**	14.58	9.15
struct_s_2	$> 10^7$	32.56	**9.30**	23.17	18.88
struct_s_3	$> 10^7$	**7.66**	20.26	8.28	26.80
struct_s_4	$> 10^7$	22.75	**11.59**	12.31	19.54
struct_s_5	$> 10^7$	1.55	**1.00**	1.00	1.55
struct_s_6	$> 10^7$	**17.66**	21.62	20.75	44.38
struct_s_7	$> 10^7$	13.31	14.94	**13.18**	93.08
struct_s_8	$> 10^7$	21.49	**16.36**	23.79	17.86
struct_s_9	$> 10^7$	**19.12**	25.71	30.93	38.62
struct_s_10	$> 10^7$	12.63	**12.25**	16.35	12.38
struct_s_11	$> 10^7$	23.39	**17.53**	19.85	23.93
struct_s_12	$> 10^7$	21.98	27.87	26.75	**19.60**
struct_s_13	$> 10^7$	17.84	**9.98**	17.59	12.47
struct_s_14	$> 10^7$	**20.65**	22.13	25.36	126.17
struct_s_15	$> 10^7$	**1.00**	1.00	1.00	1.00
struct_s_16	$> 10^7$	22.23	19.21	**17.85**	43.81
struct_s_17	$> 10^7$	19.25	**12.45**	13.60	18.82
struct_s_18	$> 10^7$	21.22	23.97	21.99	**19.57**
struct_s_19	$> 10^7$	**15.89**	18.94	18.60	19.42
struct_s_20	$> 10^7$	**1.00**	12.33	1.00	1.58
struct_s_21	$> 10^7$	23.70	24.15	**12.94**	18.22
struct_s_22	$> 10^7$	24.15	23.93	**19.91**	75.50
struct_s_23	$> 10^7$	20.81	**13.66**	19.18	45.75
struct_s_24	$> 10^7$	**1.00**	1.00	1.25	6.56
struct_s_25	$> 10^7$	22.99	**22.09**	25.22	26.37
struct_s_26	$> 10^7$	20.81	**1.70**	17.45	9.28
struct_s_27	$> 10^7$	**16.87**	24.18	28.46	53.66
struct_s_28	$> 10^7$	**9.30**	10.10	13.60	36.25
struct_s_29	$> 10^7$	24.69	**16.86**	19.13	37.92
struct_s_30	$> 10^7$	18.91	**16.05**	20.58	23.07
Geo. mean	$> 10^7$	12.70	**10.81**	12.63	19.19
Wins	0/30	10/30	**14/30**	4/30	2/30

Table 4.13: Extended results on small random single-row instances. We compare the original formulation with the four proposed reformulations. We report the geometric mean of number of nodes over 5 randomization seeds.

Instance	Original	AHL	AHL$^{\boldsymbol{D}}$	AHL$^{\text{low}}$	KP
nostruct_s_31	117147.30	150.50	**97.44**	101.47	140.26
nostruct_s_32	35360.52	98.03	**69.74**	96.33	134.02
nostruct_s_33	77679.87	127.97	71.08	**71.06**	132.71
nostruct_s_34	59664.91	74.72	**55.96**	80.64	81.72
nostruct_s_35	470429.56	82.44	77.44	**75.05**	147.11
nostruct_s_36	162687.07	80.45	**75.76**	81.61	143.13
nostruct_s_37	79044.08	80.66	**43.49**	128.70	154.84
nostruct_s_38	126413.70	**82.99**	90.62	97.81	140.42
nostruct_s_39	271882.42	93.34	**72.66**	111.67	117.81
nostruct_s_40	57072.58	103.84	**63.85**	81.94	101.39
nostruct_s_41	166165.68	**88.74**	113.49	119.22	101.67
nostruct_s_42	225989.09	98.70	94.66	**92.23**	95.26
nostruct_s_43	199181.06	**91.84**	106.79	123.21	112.26
nostruct_s_44	122235.34	87.11	**57.07**	111.29	123.50
nostruct_s_45	102328.43	87.85	**64.62**	94.48	146.62
nostruct_s_46	99868.61	107.94	**54.96**	122.39	130.74
nostruct_s_47	102369.62	82.54	**53.36**	81.48	109.72
nostruct_s_48	79939.18	76.35	**44.38**	120.71	141.42
nostruct_s_49	285937.90	88.31	81.59	**66.65**	113.82
nostruct_s_50	116897.40	72.33	73.92	**63.07**	104.74
nostruct_s_51	108565.32	109.67	**63.67**	70.96	147.21
nostruct_s_52	159551.60	97.12	**76.30**	79.99	239.27
nostruct_s_53	251743.11	72.03	**57.34**	65.10	110.13
nostruct_s_54	187536.86	95.47	**60.53**	81.71	124.45
nostruct_s_55	58046.96	154.81	**60.35**	61.42	132.67
nostruct_s_56	152474.97	100.48	**61.88**	77.97	75.94
nostruct_s_57	147567.40	136.44	**76.11**	101.30	209.92
nostruct_s_58	51806.16	139.00	**85.97**	129.19	120.62
nostruct_s_59	150024.31	128.82	**84.94**	108.48	165.21
nostruct_s_60	70512.55	91.43	**37.22**	83.78	191.71
Geo. mean	121,456	97.10	**68.64**	90.43	128.98
Wins	0/30	3/30	**22/30**	5/30	0/30

Table 4.14: Extended results on big structured single-row instances. We compare the original formulation with the four proposed reformulations. We report the geometric mean of number of nodes over 5 randomization seeds.

Instance	Original	AHL	AHL^{D}	AHL$^{\text{low}}$	KP
struct_b_1	$> 10^7$	**1.00**	**1.00**	**1.00**	**1.00**
struct_b_2	$> 10^7$	**1.00**	**1.00**	**1.00**	**1.00**
struct_b_3	$> 10^7$	**1.00**	**1.00**	**1.00**	**1.00**
struct_b_4	$> 10^7$	**1.00**	**1.00**	**1.00**	**1.00**
struct_b_5	$> 10^7$	**1.00**	**1.00**	**1.00**	**1.00**
struct_b_6	$> 10^7$	**1.00**	**1.00**	**1.00**	**1.00**
struct_b_7	$> 10^7$	**1.00**	**1.00**	**1.00**	**1.00**
struct_b_8	$> 10^7$	**1.00**	**1.00**	**1.00**	**1.00**
struct_b_9	$> 10^7$	**1.00**	**1.00**	**1.00**	**1.00**
struct_b_10	$> 10^7$	**1.00**	**1.00**	**1.00**	**1.00**
struct_b_11	$> 10^7$	**1.00**	**1.00**	**1.00**	**1.00**
struct_b_12	$> 10^7$	1.15	**1.00**	1.38	**1.00**
struct_b_13	$> 10^7$	**1.00**	**1.00**	**1.00**	**1.00**
struct_b_14	$> 10^7$	**1.00**	**1.00**	**1.00**	**1.00**
struct_b_15	$> 10^7$	**1.00**	**1.00**	**1.00**	**1.00**
struct_b_16	$> 10^7$	**1.00**	**1.00**	**1.00**	**1.00**
struct_b_17	$> 10^7$	**1.00**	**1.00**	**1.00**	1.25
struct_b_18	$> 10^7$	**1.00**	**1.00**	**1.00**	**1.00**
struct_b_19	$> 10^7$	7.09	10.92	11.13	**2.23**
struct_b_20	$> 10^7$	**1.00**	**1.00**	**1.00**	**1.00**
struct_b_21	$> 10^7$	**1.00**	**1.00**	1.15	**1.00**
struct_b_22	$> 10^7$	12.88	12.45	13.02	**7.55**
struct_b_23	$> 10^7$	**1.00**	**1.00**	**1.00**	**1.00**
struct_b_24	$> 10^7$	**1.00**	**1.00**	**1.00**	**1.00**
struct_b_25	$> 10^7$	**1.00**	**1.00**	**1.00**	**1.00**
struct_b_26	$> 10^7$	14.55	**10.30**	15.64	14.94
struct_b_27	$> 10^7$	1.15	**1.00**	**1.00**	**1.00**
struct_b_28	$> 10^7$	**1.00**	**1.00**	**1.00**	**1.00**
struct_b_29	$> 10^7$	**1.00**	**1.00**	**1.00**	**1.00**
struct_b_30	$> 10^7$	**1.00**	**1.00**	**1.00**	**1.00**
Geo. mean	$> 10^7$	1.28	**1.27**	1.31	1.21
Wins	0/30	0/30	**1/30**	0/30	2/30

Table 4.15: Extended results on big random single-row instances. We compare the original formulation with the four proposed reformulations. We report the geometric mean of number of nodes over 5 randomization seeds.

Instance	Original	AHL	AHL^{D}	AHL$^{\text{low}}$	KP
nostruct_b_31	40066.06	499.89	**225.39**	390.18	335.75
nostruct_b_32	58912.41	768.76	**451.06**	918.01	496.08
nostruct_b_33	69796.78	1321.29	1258.45	4338.72	**712.30**
nostruct_b_34	61246.15	568.32	585.86	2186.62	**449.92**
nostruct_b_35	47761.72	**648.27**	901.42	1099.37	723.79
nostruct_b_36	75469.91	595.61	784.11	2221.29	**532.13**
nostruct_b_37	58415.69	696.93	**681.78**	1672.96	965.29
nostruct_b_38	69676.77	1144.81	**450.73**	1304.41	580.92
nostruct_b_39	65902.10	1169.01	1043.94	3588.43	**792.13**
nostruct_b_40	60974.73	667.06	**442.64**	1742.68	562.90
nostruct_b_41	55427.79	732.07	564.78	1342.24	**457.19**
nostruct_b_42	105708.90	741.30	**629.10**	2686.09	891.34
nostruct_b_43	53388.13	345.38	249.22	590.61	**143.52**
nostruct_b_44	53313.65	660.75	458.89	1067.95	**136.93**
nostruct_b_45	56414.52	550.83	661.25	851.72	**486.87**
nostruct_b_46	50542.79	494.18	**354.76**	2725.77	625.31
nostruct_b_47	32448.48	1341.72	**601.52**	1094.96	787.56
nostruct_b_48	60434.83	577.56	**484.34**	5392.68	1084.96
nostruct_b_49	62315.48	537.31	**469.06**	2535.80	487.58
nostruct_b_50	47790.45	684.91	606.55	1135.02	**144.58**
nostruct_b_51	42576.21	761.53	**626.06**	2248.45	785.46
nostruct_b_52	45216.55	443.64	**415.52**	3989.46	474.94
nostruct_b_53	52921.38	498.10	**444.15**	1174.28	542.21
nostruct_b_54	54014.03	605.54	603.97	1023.27	**539.56**
nostruct_b_55	52049.79	465.36	**445.15**	5930.21	865.73
nostruct_b_56	69288.17	774.16	**673.18**	1914.84	751.58
nostruct_b_57	85747.18	538.98	**475.09**	703.69	582.79
nostruct_b_58	87641.90	1316.64	1177.69	3758.66	**1058.79**
nostruct_b_59	59819.40	617.21	**526.40**	1988.41	597.01
nostruct_b_60	142758.31	2208.95	**1442.30**	4899.36	2050.46
Geo. mean	59,901	701.15	572.03	1779.06	**566.99**
Wins	0/30	1/30	**18/30**	0/30	11/30

Table 4.16: Extended results on market split instances. We compare the original formulation with the three proposed reformulations. We report the geometric mean of number of nodes over 5 randomization seeds.

Instance	Original	AHL	AHL^{D}	AHL$^{\text{low}}$	KP
MS_1	682632.80	**772.81**	4278.43	2842.57	
MS_2	580468.05	**808.12**	4701.41	1864.10	
MS_3	478755.61	**701.96**	4857.28	1738.94	
MS_4	529921.84	**639.50**	6368.87	2965.49	
MS_5	559298.39	**706.37**	3904.89	969.19	
MS_6	509460.40	**635.77**	7049.53	1093.95	
MS_7	90924.66	**777.72**	6602.53	1353.35	
MS_8	329578.61	**543.42**	6484.97	1942.68	
MS_9	492957.09	**605.89**	4730.02	1001.01	
MS_10	261514.02	**1141.54**	5185.66	1990.06	
MS_11	164414.90	**341.25**	3492.53	563.33	
MS_12	527807.00	**549.54**	4749.78	2100.33	
MS_13	370893.73	485.87	3398.73	**217.41**	
MS_14	981426.39	**602.94**	2929.34	1334.05	
MS_15	685585.93	**749.86**	5829.67	1617.54	
MS_16	569928.81	**787.34**	9494.30	2564.49	
MS_17	308195.50	**785.04**	5943.94	1266.77	
MS_18	356061.58	**618.53**	3284.40	1035.91	
MS_19	399377.88	**744.05**	4300.51	1981.01	
MS_20	464122.62	**1051.48**	1990.04	1945.08	
MS_21	468660.87	**675.46**	2433.52	1571.35	
MS_22	383869.78	**436.71**	2885.50	1542.10	
MS_23	502538.33	**966.24**	2744.84	1393.33	
MS_24	334804.99	**705.90**	4050.86	1887.22	
MS_25	208598.85	**943.70**	3703.26	3742.81	
MS_26	390655.79	**524.95**	4631.38	1676.14	
MS_27	279747.81	**526.48**	2422.81	2544.03	
MS_28	284524.68	**893.36**	2093.21	1325.25	
MS_29	300324.92	**721.64**	3900.22	1252.29	
MS_30	640326.92	**768.06**	3613.77	2572.34	
Geo. mean	398,742	**685.03**	4111.48	1545.95	
Wins	0/30	**29/30**	0/30	1/30	

Table 4.17: Extended results on generalized assignment problem. We compare the original formulation with the three proposed reformulations. We report the geometric mean of number of nodes over 5 randomization seeds.

Instance	Original	AHL	AHL^{D}	AHL$^{\text{low}}$	KP
GAP_1	622.36	92.85	**92.60**	219.08	
GAP_2	762.92	54.95	100.66	**33.89**	
GAP_3	589.38	145.32	**75.83**	124.92	
GAP_4	1425.92	54.49	**42.84**	324.64	
GAP_5	918.39	**2.91**	45.31	228.08	
GAP_6	9761.63	**194.82**	229.32	291.90	
GAP_7	1017.21	71.43	**66.32**	392.18	
GAP_8	4286.72	**27.75**	77.55	118.83	
GAP_9	138.85	43.92	48.94	**26.26**	
GAP_10	7342.77	187.21	**68.74**	242.12	
GAP_11	1167.24	**66.77**	251.20	303.76	
GAP_12	316.07	**60.96**	75.00	127.00	
GAP_13	209.89	82.45	**44.96**	120.48	
GAP_14	182.15	81.18	315.16	**49.51**	
GAP_15	283.40	43.72	39.34	**38.01**	
GAP_16	673.14	61.96	**34.80**	44.55	
GAP_17	2417.82	**75.90**	125.44	203.97	
GAP_18	1191.56	139.33	**91.05**	107.82	
GAP_19	435.70	**33.10**	33.37	148.83	
GAP_20	869.20	**84.40**	111.56	141.65	
GAP_21	774.89	**6.02**	45.70	131.38	
GAP_22	10823.20	**52.38**	86.75	223.37	
GAP_23	911.96	**28.98**	190.24	43.02	
GAP_24	275.78	**13.30**	24.09	118.71	
GAP_25	522.26	22.18	**20.52**	189.12	
GAP_26	356.28	**34.20**	48.35	46.80	
GAP_27	110.47	141.51	**56.04**	253.84	
GAP_28	1261.65	126.92	**71.87**	95.56	
GAP_29	1770.86	**26.29**	63.91	88.98	
GAP_30	7508.54	**93.21**	165.66	158.85	
Geo. mean	893.37	**52.15**	72.52	122.92	
Wins	0/30	**15/30**	11/30	4/30	

Table 4.18: Extended results on combinatorial auction instances. We compare the original formulation with the three proposed reformulations. We report the geometric mean of number of nodes over 5 randomization seeds.

Instance	Original	AHL	AHL^{D}	AHL$^{\text{low}}$	KP
CA_1	**3.44**	4.74	7.90	4.28	
CA_2	**4.18**	5.14	9.90	6.25	
CA_3	**10.76**	30.28	121.90	19.35	
CA_4	**7.19**	12.93	85.17	9.32	
CA_5	**298.70**	485.35	1079.77	350.94	
CA_6	**34.02**	51.74	339.22	38.92	
CA_7	**10.02**	18.06	62.91	16.49	
CA_8	**18.14**	35.72	205.20	19.63	
CA_9	**10.43**	17.26	163.22	17.58	
CA_10	**134.85**	195.28	933.42	199.42	
CA_11	**3.95**	7.97	14.92	4.54	
CA_12	**2.30**	4.54	11.42	3.10	
CA_13	**37.30**	42.72	497.02	41.21	
CA_14	**2.93**	4.18	44.24	4.28	
CA_15	**330.87**	651.44	4057.72	530.10	
CA_16	**16.91**	49.07	761.60	32.81	
CA_17	**24.01**	31.66	315.86	29.18	
CA_18	**4.47**	4.51	7.71	5.75	
CA_19	**9.36**	25.02	153.73	12.91	
CA_20	**4.00**	13.89	79.94	6.00	
CA_21	**4.28**	11.77	34.16	6.45	
CA_22	**12.59**	64.43	82.01	14.01	
CA_23	**11.93**	16.27	58.03	15.82	
CA_24	**31.14**	143.04	811.13	60.27	
CA_25	**15.00**	29.02	86.09	24.40	
CA_26	**11.51**	20.30	143.17	13.04	
CA_27	**5.72**	6.88	54.93	6.04	
CA_28	**14.76**	21.74	634.44	26.11	
CA_29	4.37	**4.18**	17.23	4.32	
CA_30	**4.70**	7.69	194.67	8.91	
Geo. mean	**11.26**	21.03	95.08	16.14	
Wins	**29/30**	1/30	0/30	0/30	

5

LEARNING OPTIMAL OBJECTIVE VALUES FOR MILP

In recent years, there has been a surge in interest in harnessing the power of machine learning tools to aid the solution process of MILPs. From solution prediction (e.g. [51, 117, 142]) to interventions on the heuristic rules used by the solvers (e.g. [66, 37, 126]) several approaches have been studied in the literature (see Chapter 2 for a more in-depth discussion of this topic). The overarching trend is to build dynamic MILP solvers that can make active use of the large amounts of data produced during the solving process.

Many of the decisions that must be made during the B&B process could be better informed were the optimal solution known from the start. In fact, even knowing the optimal *objective value* can positively influence the solver behavior. For example, once a solution is found that matches this value, any effort to find new solutions can be avoided. With perfect information of the optimal objective value, a solver can further do more aggressive pruning of nodes. In general, having this knowledge can allow the solver to adapt its configuration, putting more emphasis on different components. Even in absence of perfect information, a good prediction of the optimal objective value can still be used to change the solver settings or to devise smarter rules, such as node selection policies that account for this predicted value. Inspired by these observations we ask the two following (closely related) questions.

(Q1) How well can we predict the optimal *objective value*?

(Q2) With what accuracy can we predict, during the solution process, whether or not a given solution is optimal?

This chapter presents our proposed methodology to give an answer to the above questions. We start by defining some key concepts and notation in Section 5.1, followed by a discussion of the work most closely related to ours (Section 5.2). Section 5.3 describes our methodology in detail. The results to our computational study are presented

in Section 5.4. Finally, we conclude with some final remarks and future work in Section 5.5. The code to reproduce all experiments is available online [134].

5.1. BACKGROUND

Mixed Integer Linear Programming For ease of reading, we repeat once more the definition of an MILP. We are given a matrix $\boldsymbol{A} \in \mathbb{Q}^{m \times n}$, vectors $\boldsymbol{c} \in \mathbb{Q}^n$ and $\boldsymbol{b} \in \mathbb{Q}^m$, and a partition $(\mathscr{I}, \mathscr{C})$ of the variable index set $[n]$. A Mixed Integer Linear Program is the problem of finding

$$\begin{aligned} z^* = \min\ & \boldsymbol{c}^\top \boldsymbol{x} \\ \text{subject to } & A\boldsymbol{x} \geq \boldsymbol{b}, \\ & x_j \in \mathbb{Z}_{\geq 0} \quad \forall j \in \mathscr{I}, \\ & x_j \geq 0 \quad \forall j \in \mathscr{C}. \end{aligned} \tag{5.1}$$

Solving Mixed Integer Linear Programs The standard approach to solving MILPs is to use the LP-based branch-and-bound (B&B) algorithm. This algorithm sequentially partitions the feasible region, while using LP relaxations to obtain lower bounds on the quality of the solutions of each sub-region. This search can be represented as a binary[1] tree. At a given time t of the solution process we use T_t to denote the search tree, i.e. the set of nodes, constructed so far by the B&B algorithm. We denote by $\boldsymbol{x}^*$ the optimal solution to Problem (5.1) and z^* its corresponding optimal objective value. For a given node i of the search tree, let z_i^{LP} be the optimal objective value of the node's LP relaxation. We use the notation z^{LP} for the root node, i.e., the solution to the original problem's LP relaxation. At any point of the search, an integer feasible solution provides an upper bound on the optimal objective value. Let $\bar{\boldsymbol{x}}(t)$ be the best known solution at time t and let $\bar{z}(t) = \boldsymbol{c}^\top \bar{\boldsymbol{x}}(t)$ denote its objective value (also called the *incumbent*). Then we can prune any node i such that $z_i^{LP} \geq \bar{z}(t)$.

The nodes of T_t can be classified into three types:

- I_t is the set of inner nodes of the tree. This is, nodes that have been processed (its LP relaxation solved) and resulted in branching.
- L_t is the set of leaves of the tree. This is, the set of nodes that have been processed and resulted in pruning or in an integer feasible solution.
- O_t is the set of open nodes, i.e., nodes which have not been processed yet.

As mentioned before, the incumbent $\bar{z}(t)$ provides an upper bound on z^*. We can also obtain a global lower bound. Let $\underline{z}(t) := \min_{i \in O_t}\{z_i^{LP}\}$. Then notice that necessarily $\underline{z}(t) \leq z^*$.

In practice, MILP solvers implement a plethora of other techniques to accelerate the solution process (see Section 1.3.3). Among them, cutting planes and primal heuristics are essential parts of today's mathematical optimization software.

[1] Standard implementations of the B&B algorithm use single-variable disjunctions that partition the feasible set into two. Other approaches exist but are, to the best of our knowledge, not implemented in standard optimization software.

MILP solving phases The B&B algorithm can solve MILPs to optimality. This means that, if the algorithm terminates, it does so after having obtained a feasible solution and a proof of its optimality (or, on the contrary, proof of infeasibility). Several solver components work together for this goal, each with more or less focus on the feasibility and the optimality parts. Berthold et al. [25] point out that, typically, the optimal solution is found well before the solver can prove optimality. Following this, they propose partitioning the search process into phases, according to three target goals. These phases are the following.

1. **Feasibility.** This phase encompasses the time spanned from the beginning of the search until the first feasible solution is found.
2. **Improvement.** From the moment the first feasible solution is found until an optimal solution is found.
3. **Proving.** Spans the time elapsed from the moment the optimal solution is found until the solver terminates with a proof of optimality.

Notice that if the instance is infeasible the solver terminates while in the first phase. For the purpose of this chapter we assume that the instances are feasible.

5.2. RELATED WORK

MILP solution prediction In recent years, the topic of solution prediction for MILPs has gained momentum. The goal is to produce a (partial) assignment of the integer variables via a predictive machine learning model. This prediction can then be used to guide the search in different ways. Ding et al. [51] impose a constraint that forces search to remain in a neighborhood of the predicted optimal solution. In this way, by restricting the size of the feasible region, the authors aim to accelerate the solution process. In contrast, the approaches of Nair et al. [117] and Khalil et al. [89] consist in fixing a subset of variables to their predicted optimal value, letting the solver optimize over the remaining ones. Khalil et al. [89] further propose a solver mode that uses the predicted solution to guide the node processing order. In the present work, we take a different path by aiming to predict the *optimal objective value*, as opposed to the solution, i.e., the values that each variable takes. This task is easier from a learning perspective, yet still offers several ways in which one can exploit this information.

Phase transition predictions Berthold et al. [25] defined the three phases of MILP solving that were introduced in Section 5.1. Their goal is to adapt the solver's strategy depending on the phase. For this purpose, they propose two criteria that can be used to predict the transition between phase 2 (improvement) and phase 3 (proving) without knowledge of the optimal solution. These criteria are based on node estimates: for every node $i \in T_t$, the solver SCIP keeps an estimate $\hat{c}(i)$ of the objective value of the best solution attainable at that node (see [25] for a formal definition of how this is computed). At time t of the solving process, let $\hat{c}^{\min}(t) := \min\{\hat{c}(i) \mid i \in O_t\}$ be the minimum estimate among the open nodes. We further define $d(i)$ to be the depth[2] of node i. The first tran-

[2] We define the depth of a node as its distance to the root node. Therefore, by definition, the depth of the root node is zero.

sition criterion, the *best-estimate* criterion, indicates that the transition moment is the first time the incumbent becomes smaller than $\hat{c}^{\min}(t)$. Formally, let us define a binary classifier C^{est} that indicates if the transition has occurred using the criterion

$$C^{\text{est}} = \begin{cases} 1 & \text{if } \min_{s\in[0,t]}\{\bar{z}(s) - \hat{c}^{\min}(s)\} < 0 \\ 0 & \text{otherwise.} \end{cases} \tag{5.2}$$

The second criterion is called *rank-1* and is based on the set of open nodes with better estimate than the processed nodes at the same depth. Formally, let

$$R^1(t) := \left\{ i \in O_t \mid \hat{c}(i) \leq \inf\{\hat{c}(j) : j \in I_t \cup L_t, d(j) = d(i)\} \right\}$$

This set can be used to define a classifier $C^{\text{rank-1}}$ that indicates that the transition has occurred once the set becomes empty for the first time. This is,

$$C^{\text{rank-1}} = \begin{cases} 1 & \text{if } \min_{s\in[0,t]} |R^1(s)| = 0 \\ 0 & \text{otherwise.} \end{cases} \tag{5.3}$$

The authors use these criteria to switch between different pre-determined solver settings depending on the phase of solving. Their experiments show improved solving time, especially when using the rank-1 criterion. However, it is also clear that both criteria tend to be satisfied *before* the phase transition actually occurs, and there is some room for improvement in the accuracy of the classifiers, as we shall see from our own computational study.

B&B resolution predictions Closely related to the present work is that of Hendel et al. [80], who use a number of solver metrics to predict the final B&B tree size. They use a combination of metrics from the literature, together with their own, as input to a machine learning model that estimates the final tree size dynamically as the tree is being constructed. Their method was incorporated into version 7.0 of the solver SCIP as a progress metric for the user. In a similar fashion, Fischetti et al. [60] use a number of solver metrics to, during the solving process, predict whether or not the run will end within the given time limit. This prediction can be used to adapt the solver behavior in the case that the answer is negative.

5.3. METHODOLOGY

This section details the methodology used to answer questions Q1 (Section 5.3.1) and Q2 (Section 5.3.2). We assume we are given a space $\mathscr{X}$ of instances of interest. For some tasks, we will use the bipartite graph representation of MILPs introduced by Gasse et al. [66]. This is, given an MILP instance $X \in \mathscr{X}$ defined as in Eq. 5.1, we build a graph representation as follows: each constraint and each variable have a corresponding representative node. A constraint node is connected to a variable node if the corresponding variable has a non-zero coefficient in the corresponding constraint. Each node has an associated vector of features that describes it. We utilize the same features as Gasse et al., except that we do not include any incumbent information. In short, instead of the

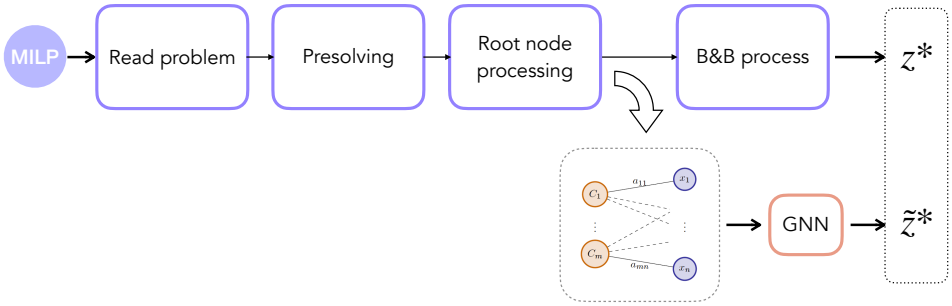


Figure 5.1: Optimal objective value prediction task. The MILP representation is computed after the root node has been processed. This serves as an input to a GNN that outputs a prediction $\tilde{z}^*$ of the optimal objective value.

raw data in $X \in \mathcal{X}$ we use the graph representation, which we denote $X_G \in \mathcal{X}_G$, and is composed of a tuple $X_G = (\boldsymbol{C}, \boldsymbol{V}, \boldsymbol{A})$, where $\boldsymbol{C} \in \mathbb{R}^{m \times d_c}$ and $\boldsymbol{V} \in \mathbb{R}^{n \times d_v}$ represent the constraint and variable features, respectively, and $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ is the adjacency matrix.

5

5.3.1. OPTIMAL VALUE PREDICTION

The first task we tackle is the one of predicting the optimal objective value (Q1). Thas is, given an MILP instance $X \in \mathcal{X}$, we want to predict the optimal objective value z^*. This prediction is computed once and for all at the root node, once the LP solution is available. We frame this as a regression task. This process is depicted in Figure 5.1.

For this regression task, we utilise the bipartite graph representation of Gasse et al. [66] defined above, which is processed using a Graph Neural Network (GNN) that performs two half-convolutions. In particular, one first pass updates the constraint descriptors using the variable descriptors, while the second pass updates the variable descriptors using the (new) constraint descriptors. The variable descriptors then go through a feedforward network with ReLU activation. Finally, average pooling is applied to obtain one single output value.

Our goal is to learn a mapping $f(X_g) : \mathcal{X}_G \mapsto \mathbb{R}$ which outputs an approximation $\tilde{z}^*$ of the optimal objective value z^*. At the moment of this prediction, the solution to the root LP relaxation is known and can be used for further context. In order to exploit that knowledge, we test three potential targets for the machine learning model, namely

$$\Theta_1 = z^*$$

$$\Theta_2 = \frac{z^*}{z^{LP}}$$

$$\Theta_3 = z^* - z^{LP}.$$

This gives rise to three models $f_1(X_g)$, $f_2(X_g)$ and $f_3(X_g)$, which we later transform into the desired output by setting either $f(X_g) = f_1(X_g)$, $f(X_g) = f_2(X_g) \cdot z^{LP}$, or $f(X_g) = f_3(X_g) + z^{LP}$.

5.3.2. PREDICTION OF PHASE TRANSITION

The second task (Q2) is predicting the transition between phases 2 (improvement) and 3 (proving). Thas is, at any point during the solution process we want to predict whether the incumbent is in fact optimal. We cast this problem as a classification task.

We test the performance of two classifiers. The first one is based on the output of the GNN model discussed in Section 5.3.1. Given an instance $X \in \mathcal{X}$ (in fact its associated graph representation X_G) and the current incumbent $\bar{z}$, we obtain a binary prediction $C_\epsilon^{GNN} : \mathcal{X}_G \times \mathbb{R} \mapsto \{0,1\}$ in the following way

$$C_\epsilon^{GNN}(X_G, \bar{z}) = \begin{cases} 1 & \text{if } \bar{z} < f(X_G) + \epsilon \cdot |f(X_G)| \\ 0 & \text{otherwise} \end{cases} \tag{5.4}$$

for some $\epsilon \in [-1,1]$. The ϵ parameter allows us to control the confidence in the prediction.

The C_ϵ^{GNN} classifier is static, in the sense that it does not make use of any information coming from the B&B process. On the contrary, the second predictor we propose, which we call C^D, is based on a set of dynamic metrics that are collected during the solving process. The metrics are the following.

Gap Following SCIP, we define the gap as

$$g(t) := \begin{cases} 1 & \text{if no solution has been found yet or } \bar{z}(t) \cdot \underline{z}(t) < 0, \\ \frac{|\bar{z}(t) - \underline{z}(t)|}{\max\{|\bar{z}(t)|, |\underline{z}(t)|, \epsilon\}} & \text{otherwise.} \end{cases} \tag{5.5}$$

Tree weight For a given node $v \in T_t$, let $d(v)$ denote the node's depth. Then, the tree weight at time t is defined as

$$\omega(t) := \sum_{v \in L_t} 2^{-d(v)}. \tag{5.6}$$

This metric was first defined by Kilby et al. [91].

Median gap Let $m(t) = \text{median}\{z_i^{LP} \mid i \in O_t\}$ and let $\bar{z}^0$ be the first incumbent found. We define the median gap as

$$\mu(t) = \frac{|\bar{z}(t) - m(t)|}{|\bar{z}^0 - z^{LP}|} \tag{5.7}$$

Trend of open nodes For a certain window size h, we store the values of $|O_k|$ for $k \in \{t-h, t-h+1, ..., t\}$. We then fit a linear function using least squares to compute the *trend* of this sequence. We denote this trend at time t as $\tau(t)$.

Ratio to GNN prediction We make use of the prediction $f(X_G)$ coming from the GNN model and include the ratio with respect to the current incumbent as a metric. In particular we use

$$\rho(t) = \frac{f(X_G)}{\bar{z}(t)} \tag{5.8}$$

Notice that, while the gap and the tree weight are metrics from the literature, the other three are our own.

The input to the classifier is therefore a tuple $X_D = (g(t), \omega(t), \mu(t), \tau(t), \rho(t))$. We train a classifier $C^D(X_D)$ that makes use of these dynamic features to make a binary prediction on whether we are in phase 2 or 3. We use a simple logistic regression, which will allow us to more easily interpret the resulting model, in contrast to more complex machine learning models.

5.4. COMPUTATIONAL RESULTS

This section describes our computational setup and results. All experiments were performed with the solver SCIP v.8.0 [27]. Code for reproducing all experiments in this section is available online [134].

5.4.1. SETUP

Benchmarks We use three NP-hard problem benchmarks from the literature: set covering, combinatorial auctions and generalized independent set problem (GISP). We create a fourth benchmark (mixed) that is comprised of instances of the three types, in equal proportion. The method and configuration used for generation of the instances is summarized in Table 5.1. For each instance type, we generate 10,000 instances for training, 2000 instances for validation and another 2000 for testing.

Phase analysis As a first approach to the instances, we run an experiment to analyze the breakdown into solving phases. We solve 100 of the training instances, each with 3 different randomization seeds, which gives us a total of 300 data points per benchmark. During the solution process we record the time when branching starts, the time when the first solution is found, the time when a solution within 5% of the optimal is found, and

Table 5.1: Method and configuration settings used to generate the instances of problem benchmark.

Benchmark	Generation method	Configuration
Set covering	Balas and Ho [16]	Items: 750 Sets: 1000
Combinatorial auctions	Leyton-Brown et al. [102] with arbitrary relationships	Items: 200 Bids: 1000
GISP	Colombi et al. [39] with Erdos-Renyi graphs	Nodes: 80 $p = 0.6$ $\alpha = 0.75$ SET2, A

the time when the optimal solution is found. This allows us to compute the percentage of time spent on each phase, and the percentage of time spent branching versus before branching (i.e., pre-processing the instance and processing the root node). We average these numbers over the 300 samples to obtain a view of the typical behavior of the solver on each benchmark. We further divide phase 2 (improvement) into two sub-phases: (2a) from the first feasible solution to the first feasible solution with objective value within 5% of the optimal, and (2b) which encompasses the rest of phase 2. The results are shown in Figure 5.2. We observe the following. For all benchmarks, obtaining a feasible solution is trivial. For set covering instances, the optimal solution is often known by the time that branching starts. In the case of combinatorial auctions, the optimal solution is typically not known at the start of B&B, but a good solution is. For GISP, finding optimal, or even good, solutions is not as easy, making the proving phase relatively shorter. We conclude that these benchmarks allow us to test our methodology on three very different settings that may arise in a real-life situation.

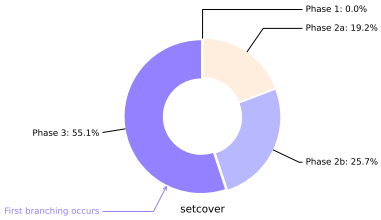
Data collection procedure For each instance, we collect information at the root node: the bipartite graph representation $X_G = (\boldsymbol{C}, \boldsymbol{V}, \boldsymbol{A})$ and the optimal root LP value z^{LP}. We then proceed to solve the instance. For the first 100 processed nodes and as long as no incumbent exists, no samples are collected. This allows us to initialize statistics as the trend of open nodes $\tau(t)$, and to ignore instances that are solved within 100 nodes which are therefore too easy. After 100 nodes have been processed and an incumbent exists, we collect samples with a probability of 0.02. At sampling time, we record the value of the dynamic features (see Section 5.3.2), as well as the incumbent value $\bar{z}(t)$. Once the instance is solved, the collected samples are completed by appending the root node information (X_G, z^{LP}) as well as the optimal objective value z^*, which will be used as a target.

Optimal objective value prediction (Q1) We test the prediction accuracy of our GNN model on the four benchmarks. We train a model for each of the targets described in Section 5.3. We measure the error as

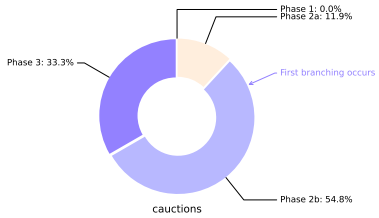
$$e = 100 \times \frac{1}{N} \sum_{i=1}^{N} \frac{|z_i^* - \tilde{z}_i^*|}{|z_i^*|} \tag{5.9}$$

where N is the number of samples, z_i^* is the optimal objective value of sample i and $\tilde{z}_i^*$ is the predicted optimal objective value of sample i. Notice that, independently of the learning target, we measure the error in the space of the original prediction we want to make.

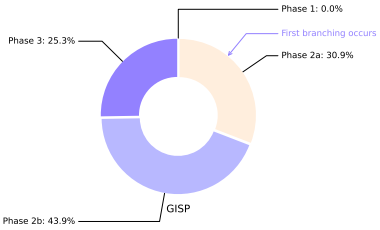
Prediction of phase transition (Q2) We make a prediction on whether we have transitioned to phase 3 (optimal solution has been found). We compare the performance of four predictors. The first two predictors are the ones proposed by Berthold et al. [25], namely C^{est} (best-estimate, see Eq. 5.2) and $C^{\text{rank-1}}$ (rank-1, see Eq. 5.3). The third predictor C_ϵ^{GNN} is based on the GNN regression model, as described in Eq. 5.4. We report the performance of this classifier with $\epsilon = 0$ and with a tuned value ϵ^* which was obtained by optimizing the accuracy with a small grid search over the range $[-0.02, 0.02]$



(a) Set covering



(b) Combinatorial auctions



(c) GISP

Figure 5.2: Phase analysis of three instance types. We divide the solution process into (1) Feasibility, in dark yellow, (2a) Improvement up to 5% to optimality, in light yellow, (2b) Improvement from 5% to optimal, in light purple, and (3) Proving, in dark purple. We also indicate when the first branching occurs. The data is averaged over 100 instances with 3 randomization seeds (i.e., 300 samples).

on the validation set. The fourth predictor C^D is based on the dynamic features, as described in Section 5.3.2.

5.4.2. RESULTS

Tables 5.2 and 5.3 show the results of the optimal objective value prediction task. The GNN models tested in Table 5.2 were trained and tested on instances of the same type. On the contrary, the results of Table 5.3 correspond to one unique model that was trained in the mixed dataset, and then tested on different benchmarks. First, we observe that using targets that include LP information (Θ_2 and Θ_3) is beneficial to performance, as opposed to directly trying to predict the optimal objective value (Θ_1). There is no clear winner among targets Θ_2 and Θ_3. Second, we observe that the generalist model, the one trained on the mixed dataset, performs comparably to the specialized models, even outperforming them in some cases.

We now select one GNN model per benchmark to be used in the next prediction task: the phase transition prediction. We select the model in the following way: we use the specialized model that achieves the best result on the validation set. Figure 5.3 (a-c) shows the results for all classifiers on the pure benchmarks (see Table 5.4 for the same results in table form). Further, we include a column that shows the classification accuracy of a dummy model that always predicts the majority class. We observe that the classifiers of [25] (best-estimate and rank-1) tend to predict the phase transition too early. This is,

Table 5.2: Average relative error (as defined in Eq. 5.9) of the GNN model. One model was trained per benchmark. The train and test instances in each case are of the same type.

Instances	Θ_1	Θ_2	Θ_3
Set covering	1.48	0.80	**0.54**
Combinatorial auctions	3.20	**0.55**	0.62
GISP	3.32	**2.35**	2.39

Table 5.3: Average relative error (as defined in Eq. 5.9) of the GNN *mixed* model. Only one model was trained on a dataset comprised of intances of all types. The test sets are comprised of instances of one type only, except for the mixed test set (last row).

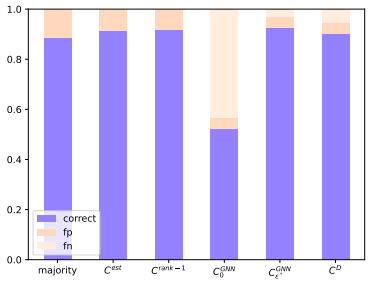
Instances	Θ_1	Θ_2	Θ_3
Set covering	1.35	**0.73**	0.82
Combinatorial auctions	3.15	1.17	**0.53**
GISP	3.17	**2.32**	2.43
Mixed test set	1.70	0.97	**0.75**

they mostly output a positive prediction, which means they believe the incumbent to be optimal. This results in the misclassified samples being almost exclusively false positives. On the contrary, the GNN model C_0^{GNN} tends to be too pessimistic, which can be fixed with the right tuning of the ϵ parameter. For all benchmarks, $C_{\epsilon^*}^{GNN}$ performs better than the classifiers of Berthold et al. [25]. At the same time, the inclusion of the dynamic features (C^D) further improves the performance, except for set covering where $C_{\epsilon^*}^{GNN}$ and C^D are close to a tie.

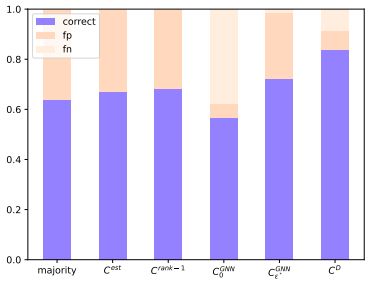
It is important to notice that, depending on the application, false positives and false negatives could have very different consequences. As an example, if the phase transition prediction is used to change the behavior of the primal heuristics (e.g. switch them off once the optimal is found) a false positive could excessively delay finding the optimal solution and therefore has a much bigger potential of harming performance than a false negative. The parameter ϵ provides an easy way to navigate this tradeoff, where one could sacrifice some accuracy to keep the rate of false positives to a minimum.

Figure 5.3d shows the same experiment but on a mixed dataset. This is, the models were trained *and* tested on a benchmark comprised of instances of all three types (in equal proportion). We observe a similar behavior compared to the specialized benchmarks. The GNN model C_0^{GNN} tends to be too pessimistic, while $C_{\epsilon^*}^{GNN}$ achieves better accuracy and better false positive rate than the classifiers of Berthold et al. [25]. Using dynamic features further improves the accuracy of the model.

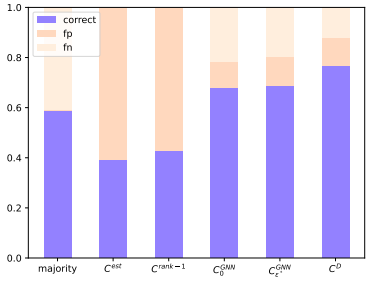
Finally, we analyze the importance of the dynamic features assigned by the C^D classifier (Figure 5.4). We see that the four learned models are in fact very different, with the GISP model mostly making decisions based on the gap and the other three considering all features more uniformly. This speaks in favour of learning on sets of instances of the



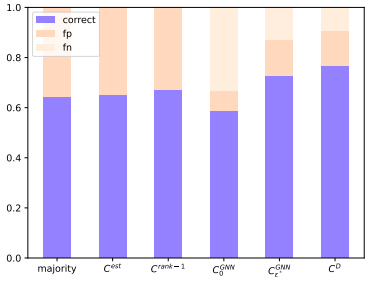
(a) Set covering



(b) Combinatorial auctions



(c) GISP



(d) Mixed

Figure 5.3: Prediction accuracy of the different classifier models. We show the fraction of correctly classified samples (correct, in purple), the fraction of false positives (fp, dark yellow) and the fraction of false negatives (fn, light yellow).

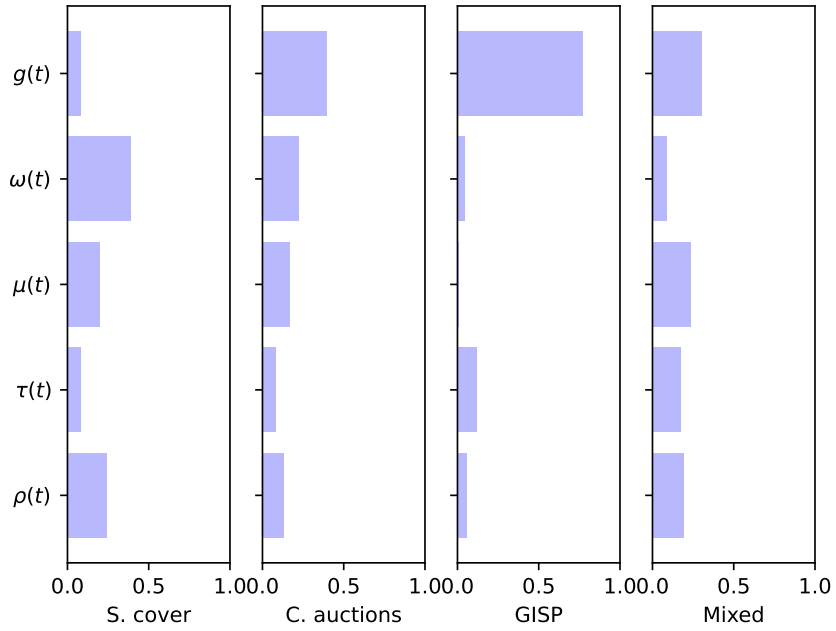


Figure 5.4: Feature importance of the dynamic models trained to predict phase transition for each of the benchmarks.

same type.

5.5. CONCLUSIONS

In this chapter, we presented our methodology for predicting the optimal objective value of MILPs. Compared to the literature on predicting optimal solutions, our learning task is easier, yet still offers a variety of possibilities for its application within MILP solvers. Our methods can be used to both predict the optimal objective value and classify a feasible solution into optimal or sub-optimal. Our computational study shows that our proposed approach outperforms the existing ones in the literature. Further, they provide more flexibility to tune the model into the desired behavior. We show that there are benefits to learning a model that specializes to an instance type, yet our model is still able to generalize well and have superior performance to other methods on mixed instance sets.

These results open the door for many possible applications. In general terms, this prediction can be used to adapt the behavior of the different solver components and rules depending on the solving phase. These applications, however, require further study and will be the subject of future work.

Table 5.4: Prediction accuracy of the different classifier models. We show the fraction of correctly classified samples, the fraction of false positives and the fraction of false negatives.

	Correct	False positives	False negatives
Majority	0.89	0.11	0.00
C^{est}	0.91	0.09	0.00
$C^{\text{rank-1}}$	0.92	0.08	0.00
C^{GNN}_{0}	0.52	0.05	0.43
$C^{\text{GNN}}_{\epsilon^*}$	0.93	0.04	0.03
C^{D}	0.90	0.05	0.05

Set covering

	Correct	False positives	False negatives
Majority	0.64	0.36	0.00
C^{est}	0.67	0.33	0.00
$C^{\text{rank-1}}$	0.68	0.32	0.00
C^{GNN}_{0}	0.57	0.06	0.37
$C^{\text{GNN}}_{\epsilon^*}$	0.72	0.27	0.01
C^{D}	0.84	0.07	0.09

Combinatorial auctions

	Correct	False positives	False negatives
Majority	0.59	0.00	0.41
C^{est}	0.39	0.61	0.00
$C^{\text{rank-1}}$	0.43	0.57	0.00
C^{GNN}_{0}	0.68	0.10	0.22
$C^{\text{GNN}}_{\epsilon^*}$	0.69	0.12	0.19
C^{D}	0.77	0.11	0.12

GISP

	Correct	False positives	False negatives
Majority	0.64	0.36	0.00
C^{est}	0.65	0.35	0.00
$C^{\text{rank-1}}$	0.67	0.33	0.00
C^{GNN}_{0}	0.59	0.08	0.34
$C^{\text{GNN}}_{\epsilon^*}$	0.73	0.14	0.13
C^{D}	0.77	0.14	0.09

Mixed

6

Conclusions

Commercial MILP solvers are typically used as a magical black box that can handle an extremely wide range of applications. The algorithmic progress that has taken us to this point is truly remarkable. Still, recent advances, such as the ones in the field of machine learning, offer a myriad of new possibilities for further improvement. In this thesis, we have explored several such perspectives. We have recognized the value of non-standard approaches. From lattice reformulations that help us solve a subset of difficult problems to learning rules that specialize to a problem type, it is clear that there is no one-fits-all strategy and that having a toolbox of different strategies is crucial. Identifying patterns and structures that allow us to pick the right strategy at the right time is equally important to make the best use of the resources. In general, the work presented in this thesis constitutes a step forward towards the goal of MILP solvers that are adaptive, dynamic and data-driven at their core.

Throughout this thesis we have also pointed out several directions of future work. For example, our methodology presented in Chapter 3 greatly improves the convergence speed of the reinforcement learning algorithm, yet it is still considerably slower than the imitation learning approach. More work is necessary to further improve the sample complexity. Similarly, the computational requirements of the LLL algorithm can hinder the applicability of the methods discussed in Chapter 4. Better techniques for identifying the potential of the reformulation, or promising sub-structures to reformulate, can allow us to overcome that challenge. For an in-depth treatment of the areas of future work, we refer to the discussions in each individual chapter.

BIBLIOGRAPHY

[1] K. Aardal and A. Lenstra. Hard equality constrained integer knapsacks. *Math. Oper. Res.*, 29(3):724–738, 2004. Erratum: Math. Oper. Res. **31**(4):846, 2006.

[2] K. Aardal and F. von Heymann. On the structure of reduced kernel lattice bases. *Mathematics of Operations Research*, 39(3):823–840, 2014.

[3] K. Aardal, C. Hurkens, and A. Lenstra. Solving a system of linear diophantine equations with lower and upper bounds on the variables. *Math. Oper. Res.*, 25(3):427–442, 2000.

[4] K. Aardal, L. Scavuzzo, and L. A. Wolsey. A study of lattice reformulations for integer programming. *Operations Research Letters*, 51(4):401–407, 2023.

[5] T. Achterberg. *Constraint integer programming.* PhD thesis, TU Berlin, 2007.

[6] T. Achterberg and T. Berthold. Hybrid branching. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 309–311. Springer, 2009.

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

[8] T. Achterberg, T. Koch, and A. Martin. Branching rules revisited. *Oper. Res. Lett.*, 33 (1):42–54, 2005.

[9] T. Achterberg, T. Berthold, S. Heinz, T. Koch, and K. Wolter. Constraint integer programming: Techniques and applications. ZIB-Report 08-43, Zuse Institute Berlin, 2008.

[10] T. Achterberg, R. E. Bixby, Z. Gu, E. Rothberg, and D. Weninger. Presolve reductions in mixed integer programming. *INFORMS Journal on Computing*, 32(2):473–506, 2020.

[11] J. R. Alfonsín. *The diophantine Frobenius problem.* OUP Oxford, 2005.

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

[13] D. Anderson, G. Hendel, P. Le Bodic, and M. Viernickel. Clairvoyant restarts in branch-and-bound search using online tree-size estimation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1427–1434, 2019.

[14] G. Andreello, A. Caprara, and M. Fischetti. Embedding cuts in a branch and cut framework: a computational study with $\{0, \frac{1}{2}\}$-cuts. *INFORMS Journal on Computing*, 19:229–238, 2007.

[15] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. Finding cuts in the TSP (a preliminary report). Technical Report 95-05, DIMACS, 1995. URL https://api.semanticscholar.org/CorpusID:972108.

[16] E. Balas and A. Ho. Set covering algorithms using cutting planes, heuristics, and subgradient optimization: a computational study. In *Combinatorial Optimization*, pages 37–60. Springer, 1980.

[17] E. Balas, S. Ceria, G. Cornuéjols, and N. Natraj. Gomory cuts revisited. *Operations Research Letters*, 19(1):1–9, 1996.

[18] M.-F. Balcan, T. Dick, T. Sandholm, and E. Vitercik. Learning to branch. In *International Conference on Machine Learning*, pages 344–353. PMLR, 2018.

[19] M.-F. F. Balcan, S. Prasad, T. Sandholm, and E. Vitercik. Sample complexity of tree search configuration: Cutting planes and beyond. *Advances in Neural Information Processing Systems*, 34:4015–4027, 2021.

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

[21] D. Bergman, A. A. Cire, W.-J. Van Hoeve, and J. Hooker. *Decision diagrams for optimization*, volume 1. Springer, 2016.

[22] T. Berthold. Measuring the impact of primal heuristics. *Operations Research Letters*, 41(6):611–614, 2013.

[23] T. Berthold and Z. Csizmadia. The confined primal integral: a measure to benchmark heuristic MINLP solvers against global MINLP solvers. *Mathematical Programming*, 188(2):523–537, 2021.

[24] T. Berthold and G. Hendel. Learning to scale mixed-integer programs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 3661–3668, 2021.

[25] T. Berthold, G. Hendel, and T. Koch. From feasibility to improvement to proof: three phases of solving mixed-integer programs. *Optimization Methods and Software*, 33(3):499–517, 2018.

[26] T. Berthold, M. Francobaldi, and G. Hendel. Learning to use local cuts. *arXiv preprint arXiv:2206.11618*, 2022.

[27] K. Bestuzheva, M. Besançon, W.-K. Chen, A. Chmiela, T. Donkiewicz, J. van Doornmalen, L. Eifler, O. Gaul, G. Gamrath, A. Gleixner, L. Gottwald, C. Graczyk, K. Halbig, A. Hoen, C. Hojny, R. van der Hulst, T. Koch, M. Lübbecke, S. J. Maher, F. Matter, E. Mühmer, B. Müller, M. E. Pfetsch, D. Rehfeldt, S. Schlein, F. Schlösser, F. Serrano, Y. Shinano, B. Sofranac, M. Turner, S. Vigerske, F. Wegscheider, P. Wellner, D. Weninger, and J. Witzig. The SCIP Optimization Suite 8.0. Technical report, Optimization Online, December 2021. URL http://www.optimization-online.org/DB_HTML/2021/12/8728.html.

[28] K. Bestuzheva, M. Besançon, W.-K. Chen, A. Chmiela, T. Donkiewicz, J. van Doornmalen, L. Eifler, O. Gaul, G. Gamrath, A. Gleixner, L. Gottwald, C. Graczyk, K. Halbig, A. Hoen, C. Hojny, R. van der Hulst, T. Koch, M. Lübbecke, S. J. Maher, F. Matter, E. Mühmer, B. Müller, M. E. Pfetsch, D. Rehfeldt, S. Schlein, F. Schlösser, F. Serrano, Y. Shinano, B. Sofranac, M. Turner, S. Vigerske, F. Wegscheider, P. Wellner, D. Weninger, and J. Witzig. Enabling research through the SCIP Optimization Suite 8.0. *ACM Trans. Math. Softw.*, 49(2), jun 2023. ISSN 0098-3500. doi: 10.1145/3585516. URL https://doi.org/10.1145/3585516.

[29] S. Bolusani, M. Besançon, A. Gleixner, T. Berthold, C. d'Ambrosio, G. Muñoz, J. Paat, and D. Thomopulos. The MIP Workshop 2023 Computational Competition on Reoptimization. *arXiv preprint arXiv:2311.14834*, 2023.

[30] P. Bonami, A. Lodi, and G. Zarpellon. Learning a classification of mixed-integer quadratic programming problems. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR)*, volume 15, pages 595–604. Springer, 2018.

[31] P. Bonami, A. Lodi, and G. Zarpellon. A classifier to decide on the linearization of mixed-integer quadratic problems in CPLEX. *Operations Research*, 70(6):3303–3320, 2022.

[32] S. Bubeck, N. Cesa-Bianchi, et al. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *Foundations and Trends® in Machine Learning*, 5 (1):1–122, 2012.

[33] C. Cameron, J. Hartford, T. Lundy, T. Truong, A. Milligan, R. Chen, and K. Leyton-Brown. UNSAT solver synthesis via monte carlo forest search. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR)*, pages 170–189, Cham, 2024. Springer Nature Switzerland.

[34] Q. Cappart, D. Chételat, E. B. Khalil, A. Lodi, C. Morris, and P. Veličković. Combinatorial optimization and reasoning with graph neural networks. *Journal of Machine Learning Research*, 24(130):1–61, 2023.

[35] J. W. S. Cassels. *An introduction to the geometry of numbers*. Springer Science & Business Media, 2012.

[36] Z. Chen, J. Liu, X. Wang, and W. Yin. On representing linear programs by graph neural networks. In *Proceedings of the International Conference on Learning Representations*, volume 11, 2023.

[37] A. Chmiela, E. Khalil, A. Gleixner, A. Lodi, and S. Pokutta. Learning to schedule heuristics in branch and bound. *Advances in Neural Information Processing Systems*, 34:24235–24246, 2021.

[38] A. Chmiela, A. Gleixner, P. Lichocki, and S. Pokutta. Online learning for scheduling MIP heuristics. In *Proceedings of the International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR)*, pages 114–123. Springer, 2023.

[39] M. Colombi, R. Mansini, and M. Savelsbergh. The generalized independent set problem: Polyhedral analysis and solution approaches. *European Journal of Operational Research*, 260(1):41–55, 2017.

[40] M. Conforti, G. Cornuéjols, G. Zambelli, M. Conforti, G. Cornuéjols, and G. Zambelli. *Integer programming models*. Springer, 2014.

[41] W. Cook, T. Rutherford, H. Scarf, and D. Shallcross. An implementation of the generalized basis reduction algorithm for integer programming. *ORSA J. Comput.*, 5(2):206–212, 1993.

[42] G. Cornuéjols. Revival of the Gomory cuts in the 1990's. *Annals of Operations Research*, 149(1):63–66, 2007.

[43] G. Cornuéjols, R. Sridharan, and J.-M. Thizy. A comparison of heuristics and relaxations for the capacitated plant location problem. *European Journal of Operational Research*, 50(3):280–297, 1991.

[44] G. Cornuéjols and M. Dawande. A class of hard small 0-1 programs. *INFORMS J. Comput.*, 11(2):205–210, 1999.

[45] R. J. Dakin. A tree-search algorithm for mixed integer programming problems. *The computer journal*, 8(3):250–255, 1965.

[46] S. S. Dey and M. Molinaro. Theoretical challenges towards cutting-plane selection. *Mathematical Programming*, 170:237–266, 2018.

[47] S. S. Dey, Y. Dubey, M. Molinaro, and P. Shah. A theoretical and computational analysis of full strong-branching. *arXiv preprint arXiv:2110.10754*, 2021.

[48] A. Deza and E. B. Khalil. Machine learning for cutting planes in integer programming: A survey. In *International Joint Conference on Artificial Intelligence*, volume 32, pages 6592–6600. ijcai.org, 2023.

[49] T. G. Dietterich. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.

[50] B. Dilkina, C. P. Gomes, Y. Malitsky, A. Sabharwal, and M. Sellmann. Backdoors to combinatorial optimization: Feasibility and optimality. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR)*, volume 5547, pages 56–70. Springer, 2009.

[51] J.-Y. Ding, C. Zhang, L. Shen, S. Li, B. Wang, Y. Xu, and L. Song. Accelerating primal solution findings for mixed integer programs based on solution prediction. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 1452–1459, 2020.

[52] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(7), 2011.

[53] G. Dulac-Arnold, N. Levine, D. J. Mankowitz, J. Li, C. Paduraru, S. Gowal, and T. Hester. Challenges of real-world reinforcement learning: definitions, benchmarks and analysis. *Machine Learning*, 110(9):2419–2468, 2021.

[54] S. Elhedhli and J. Naoum-Sawaya. Improved branching disjunctions for branch-and-bound: An analytic center approach. *European Journal of Operational Research*, 247(1):37–45, 2015.

[55] M. Etheve, Z. Alès, C. Bissuel, O. Juan, and S. Kedad-Sidhoum. Reinforcement learning for variable selection in a branch and bound algorithm. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR)*, pages 176–185. Springer, 2020.

[56] M. Fey and J. E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[57] FICO. FICO Xpress Optimizer, 2023. URL https://www.fico.com/en/products/fico-xpress-optimization.

[58] M. Fischetti and A. Lodi. Local branching. *Mathematical programming*, 98:23–47, 2003.

[59] M. Fischetti and A. Lodi. Heuristics in mixed integer programming. *Wiley Encyclopedia of Operations Research and Management Science*, 2010.

[60] M. Fischetti, A. Lodi, and G. Zarpellon. Learning MILP resolution outcomes before reaching time-limit. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR)*, volume 16, pages 275–291. Springer, 2019.

[61] A. Fukunaga. A branch-and-bound algorithm for hard multiple knapsack problems. *Ann. Oper. Res.*, 184(1):97–119, 2011.

[62] G. Gamrath and C. Schubert. Measuring the impact of branching rules for mixed-integer programming. In *Operations Research Proceedings 2017: Selected Papers of the Annual International Conference of the German Operations Research Society (GOR), Freie Universiät Berlin, Germany, September 6-8, 2017*, pages 165–170. Springer, 2018.

[63] G. Gamrath, A. Melchiori, T. Berthold, A. M. Gleixner, and D. Salvagnin. Branching on multi-aggregated variables. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR)*, pages 141–156. Springer, 2015.

[64] G. Gamrath, D. Anderson, K. Bestuzheva, W.-K. Chen, L. Eifler, M. Gasse, P. Gemander, A. Gleixner, L. Gottwald, K. Halbig, et al. The SCIP optimization suite 7.0. 2020.

[65] G. Gamrath, T. Berthold, and D. Salvagnin. An exploratory computational analysis of dual degeneracy in mixed-integer programming. *EURO Journal on Computational Optimization*, 8(3-4):241–261, 2020.

[66] M. Gasse, D. Chételat, N. Ferroni, L. Charlin, and A. Lodi. Exact combinatorial optimization with graph convolutional neural networks. *NeurIPS*, 32, 2019.

[67] M. Gasse, S. Bowly, Q. Cappart, J. Charfreitag, L. Charlin, D. Chételat, A. Chmiela, J. Dumouchelle, A. Gleixner, A. M. Kazachkov, et al. The machine learning for combinatorial optimization competition (ml4co): Results and insights. In *NeurIPS 2021 Competitions and Demonstrations Track*, pages 220–231. PMLR, 2022.

[68] A. Gleixner, G. Hendel, G. Gamrath, T. Achterberg, M. Bastubbe, T. Berthold, P. Christophel, K. Jarck, T. Koch, J. Linderoth, et al. MIPLIB 2017: data-driven compilation of the 6th mixed-integer programming library. *Mathematical Programming Computation*, 13(3):443–490, 2021.

[69] R. E. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical Society*, 64(5):275–278, 1958.

[70] J. Gondzio. Interior point methods 25 years later. *European Journal of Operational Research*, 218(3):587–601, 2012.

[71] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT press, 2016.

[72] M. Greenacre, P. J. Groenen, T. Hastie, A. I. d'Enza, A. Markos, and E. Tuzhilina. Principal component analysis. *Nature Reviews Methods Primers*, 2(1):100, 2022.

[73] P. Gupta, M. Gasse, E. Khalil, P. Mudigonda, A. Lodi, and Y. Bengio. Hybrid models for learning to branch. *Advances in neural information processing systems*, 33: 18087–18097, 2020.

[74] P. Gupta, E. B. Khalil, D. Chetélat, M. Gasse, Y. Bengio, A. Lodi, and M. P. Kumar. Lookback for learning to branch. *arXiv preprint arXiv:2206.14987*, 2022.

[75] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023. URL https://www.gurobi.com.

[76] C. Hansknecht, I. Joormann, and S. Stiller. Cuts, primal heuristics, and learning to branch for the time-dependent traveling salesman problem. *arXiv preprint arXiv:1805.01415*, 2018.

[77] H. He, H. Daume III, and J. M. Eisner. Learning to search in branch and bound algorithms. *Advances in Neural Information Processing Systems*, 27, 2014.

[78] G. Hendel. Adaptive large neighborhood search for mixed integer programming. *Mathematical Programming Computation*, 14(2):185–221, 2022.

[79] G. Hendel, M. Miltenberger, and J. Witzig. Adaptive algorithmic behavior for solving mixed integer programs using bandit algorithms. In *Operations Research Proceedings 2018*, pages 513–519. Springer, 2019.

[80] G. Hendel, D. Anderson, P. Le Bodic, and M. E. Pfetsch. Estimating the size of branch-and-bound trees. *INFORMS Journal on Computing*, 34(2):934–952, 2022.

[81] M. Hewitt, G. L. Nemhauser, and M. W. Savelsbergh. Combining exact and heuristic approaches for the capacitated fixed-charge network flow problem. *INFORMS Journal on Computing*, 22(2):314–325, 2010.

[82] T. Huang, A. M. Ferber, Y. Tian, B. Dilkina, and B. Steiner. Searching large neighborhoods for integer linear programs with contrastive learning. In *International Conference on Machine Learning*, pages 13869–13890. PMLR, 2023.

[83] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle. Paramils: an automatic algorithm configuration framework. *Journal of artificial intelligence research*, 36: 267–306, 2009.

[84] IBM. IBM ILOG CPLEX Optimizer, 2023. URL https://www.ibm.com/products/ilog-cplex-optimization-studio/cplex-optimizer.

[85] M. Karamanov and G. Cornuéjols. Branching on general disjunctions. *Mathematical Programming*, 128(1):403–436, 2011.

[86] E. Khalil. *Towards tighter integration of machine learning and discrete optimization*. PhD thesis, Georgia Institute of Technology, 2019.

[87] E. Khalil, P. Le Bodic, L. Song, G. Nemhauser, and B. Dilkina. Learning to branch in mixed integer programming. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

[88] E. B. Khalil, B. Dilkina, G. L. Nemhauser, S. Ahmed, and Y. Shao. Learning to run heuristics in tree search. In *International Joint Conference on Artificial Intelligence*, pages 659–666, 2017.

[89] E. B. Khalil, C. Morris, and A. Lodi. MIP-GNN: A data-driven framework for guiding combinatorial solvers. *AAAI*, 2022.

[90] E. B. Khalil, P. Vaezipoor, and B. Dilkina. Finding backdoors to integer programs: A monte carlo tree search framework. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 3786–3795, 2022.

[91] P. Kilby, J. Slaney, S. Thiébaux, T. Walsh, et al. Estimating search tree size. In *Proc. of the 21st National Conf. of Artificial Intelligence, AAAI, Menlo Park*, 2006.

[92] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[93] R. Kirk, A. Zhang, E. Grefenstette, and T. Rocktäschel. A survey of zero-shot generalisation in deep reinforcement learning. *Journal of Artificial Intelligence Research*, 76:201–264, 2023.

[94] B. Krishnamoorthy and G. Pataki. Column basis reduction and decomposable knapsack problems. *Discrete Optim.*, 6(3):242–270, 2009.

[95] M. Kruber, M. E. Lübbecke, and A. Parmentier. Learning when to use a decomposition. In *International conference on AI and OR techniques in constraint programming for combinatorial optimization problems*, pages 202–210. Springer, 2017.

[96] A. G. Labassi, D. Chételat, and A. Lodi. Learning to compare nodes in branch and bound with graph neural networks. *Advances in Neural Information Processing Systems*, 2022.

[97] A. Land and A. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.

[98] P. Le Bodic and G. Nemhauser. An abstract model for branching and its application to mixed integer programming. *Mathematical Programming*, 166(1-2):369–405, 2017.

[99] A. Lenstra, H. Lenstra Jr., and L. Lovász. Factoring polynomials with rational coefficients. *Math. Ann.*, 261(4):515–534, 1982. ISSN 0025-5831. doi: 10.1007/BF01457454. URL http://dx.doi.org/10.1007/BF01457454.

[100] H. Lenstra, Jr. Lattices. In *Algorithmic number theory: lattices, number fields, curves and cryptography*, volume 44 of *Math. Sci. Res. Inst. Publ.*, pages 127–181. Cambridge Univ. Press, Cambridge, 2008.

[101] H. Lenstra Jr. Integer programming with a fixed number of variables. *Math. Oper. Res.*, 8(4):538–548, 1983.

[102] K. Leyton-Brown, M. Pearson, and Y. Shoham. Towards a universal test suite for combinatorial auction algorithms. In *Proceedings of the 2nd ACM conference on Electronic commerce*, pages 66–76, 2000.

[103] S. Li, W. Ouyang, M. Paulus, and C. Wu. Learning to configure separators in branch-and-cut. *Advances in Neural Information Processing Systems*, 36, 2024.

[104] J. Lin, J. Zhu, H. Wang, and T. Zhang. Learning to branch with tree-aware branching transformers. *Knowledge-Based Systems*, 252:109455, 2022.

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

[106] D. Liu, M. Fischetti, and A. Lodi. Learning to search in local branching. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 3796–3803, 2022.

[107] A. Lodi and A. Tramontani. Performance variability in mixed-integer programming. In *Theory driven by influential applications*, pages 1–12. INFORMS, 2013.

[108] L. Lovász and H. Scarf. The generalized basis reduction algorithm. *Mathematics of Operations Research*, 17(3):751–764, 1992.

[109] A. Mahajan and T. Ralphs. On the complexity of selecting disjunctions in integer programming. *SIAM Journal on Optimization*, 20(5):2181–2198, 2010.

[110] A. Mahajan and T. K. Ralphs. Experiments with branching using general disjunctions. In *Operations Research and Cyber-Infrastructure*, pages 101–118. Springer, 2009.

[111] H. Mahmoud and J. W. Chinneck. Achieving MILP feasibility quickly using general disjunctions. *Computers & operations research*, 40(8):2094–2102, 2013.

[112] A. Marcos Alvarez, L. Wehenkel, and Q. Louveaux. Online learning for strong branching approximation in branch-and-bound. 2016.

[113] S. Mehrotra and Z. Li. On generalized branching methods for mixed integer programming. *Techical Report 2004*, 15, 2004.

[114] M. Minsky. Steps toward artificial intelligence. *Proceedings of the IRE*, 49(1):8–30, 1961.

[115] T. M. Mitchell. *Machine Learning*. McGraw Hill, 2017.

[116] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[117] V. Nair, S. Bartunov, F. Gimeno, I. von Glehn, P. Lichocki, I. Lobov, B. O'Donoghue, N. Sonnerat, C. Tjandraatmadja, P. Wang, et al. Solving mixed integer programs using neural networks. *arXiv preprint arXiv:2012.13349*, 2020.

[118] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. Wiley Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, 1999.

[119] J. Nocedal and S. J. Wright. Linear programming: Interior-point methods. In *Numerical Optimization*, chapter 14. Springer, 2006.

[120] OpenAI. Introducing ChatGPT. https://openai.com/blog/chatgpt, 2022. [Online; accessed 04-April-2023].

[121] J. H. Owen and S. Mehrotra. Experimental results on using general disjunctions in branch-and-bound for general-integer linear programs. *Computational optimization and applications*, 20(2):159–170, 2001.

[122] M. Padberg and G. Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review*, 33(1):60–100, 1991.

[123] V. T. Paschos. *Applications of combinatorial optimization*, volume 3. John Wiley & Sons, 2014.

[124] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[125] K. K. Patel. Progressively strengthening and tuning MIP solvers for reoptimization. *arXiv preprint arXiv:2308.08986*, 2023.

[126] M. B. Paulus, G. Zarpellon, A. Krause, L. Charlin, and C. Maddison. Learning to cut by looking ahead: Cutting plane selection via imitation learning. In *International Conference on Machine Learning*, pages 17584–17600. PMLR, 2022.

[127] D. A. Pomerleau. Efficient training of artificial neural networks for autonomous navigation. *Neural Computation*, 3(1):88–97, 1991.

[128] A. Prouvost, J. Dumouchelle, L. Scavuzzo, M. Gasse, D. Chételat, and A. Lodi. Ecole: A gym-like library for machine learning in combinatorial optimization solvers. *arXiv preprint arXiv:2011.06069*, 2020.

[129] C. Qian, D. Chételat, and C. Morris. Exploring the power of graph neural networks in solving linear optimization problems. In *International Conference on Artificial Intelligence and Statistics*, pages 1432–1440. PMLR, 2024.

[130] A. Santos Xavier, F. Qiu, X. Gu, B. Becu, and S. S. Dey. MIPLearn: An extensible framework for learning-enhanced optimization, June 2023.

[131] L. Scavuzzo. Code for the paper “Learning to branch with Tree MDPs", 2022. https://github.com/lascavana/rl2branch.

[132] L. Scavuzzo. Implementation of the generalized basis reduction algorithm, 2023. https://github.com/lascavana/GeneralizedBasisReduction.

[133] L. Scavuzzo. Code for the paper “A study of lattice reformulations for integer programming", 2023. https://github.com/lascavana/lattice_reformulation.

[134] L. Scavuzzo. Code for chapter 5, 2024. https://github.com/lascavana/ObjValPrediction.

[135] L. Scavuzzo, F. Y. Chen, D. Chételat, M. Gasse, A. Lodi, N. Yorke-Smith, and K. Aardal. Learning to branch with tree MDPs. *Advances in Neural Information Processing Systems*, 2022.

[136] L. Scavuzzo, K. Aardal, A. Lodi, and N. Yorke-Smith. Machine learning augmented branch and bound for mixed integer linear programming. *Mathematical Programming*, pages 1–44, 2024.

[137] M. Sewak. *Deep Reinforcement Learning*. Springer, 2019.

[138] M. Seyfi, A. Banitalebi-Dehkordi, Z. Zhou, and Y. Zhang. Exact combinatorial optimization with temporo-attentional graph neural networks. In *Proceedings of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases (ECML PKDD)*, pages 268–283. Springer, 2023.

[139] P. Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *International conference on principles and practice of constraint programming*, pages 417–431. Springer, 1998.

[140] V. Shoup. NTL – A library for doing number theory. https://libntl.org/.

[141] J. Song, Y. Yue, B. Dilkina, et al. A general large neighborhood search framework for solving integer linear programs. *Advances in Neural Information Processing Systems*, 33:20012–20023, 2020.

[142] N. Sonnerat, P. Wang, I. Ktena, S. Bartunov, and V. Nair. Learning a large neighborhood search algorithm for mixed integer programs. *arXiv preprint arXiv:2107.10201*, 2021.

[143] R. S. Sutton, D. A. McAllester, S. P. Singh, Y. Mansour, et al. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems*, volume 99, pages 1057–1063. Citeseer, 1999.

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

[145] M. Turner, T. Koch, F. Serrano, and M. Winkler. Adaptive Cut Selection in Mixed-Integer Linear Programming. *Open Journal of Mathematical Optimization*, 4:5, 2023.

[146] R. J. Vanderbei. Linear programming: foundations and extensions. *Journal of the Operational Research Society*, 49(1):94–94, 1998.

[147] O. Vinyals, I. Babuschkin, J. Chung, M. Mathieu, M. Jaderberg, W. Czarnecki, A. Dudzik, A. Huang, P. Georgiev, R. Powell, T. Ewalds, D. Horgan, M. Kroiss, I. Danihelka, J. Agapiou, J. Oh, V. Dalibard, D. Choi, L. Sifre, Y. Sulsky, S. Vezhnevets, J. Molloy, T. Cai, D. Budden, T. Paine, C. Gulcehre, Z. Wang, T. Pfaff,

T. Pohlen, D. Yogatama, J. Cohen, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, C. Apps, K. Kavukcuoglu, D. Hassabis, and D. Silver. AlphaStar: Mastering the Real-Time Strategy Game StarCraft II. https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/, 2019.

[148] A. Votyakov and M. Frumkin. An algorithm for finding the general integer solution of a system of linear equations. *Studies in discrete optimization (Russian)*, pages 128–140, 1976.

[149] Z. Wang, X. Li, J. Wang, Y. Kuang, M. Yuan, J. Zeng, Y. Zhang, and F. Wu. Learning cut selection for mixed-integer linear programming via hierarchical sequence model. In *International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=Zob4P9bRNcK.

[150] F. Wesselmann and U. Stuhl. Implementing cutting plane management and selection techniques. In *Technical Report*. University of Paderborn, 2012.

[151] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3):229–256, 1992.

[152] L. A. Wolsey. *Integer Programming*. John Wiley & Sons, 2020.

[153] S. J. Wright. *Primal-dual interior-point methods*. SIAM, 1997.

[154] Y. Wu, W. Song, Z. Cao, and J. Zhang. Learning large neighborhood search policy for integer programming. *Advances in Neural Information Processing Systems*, 34: 30075–30087, 2021.

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

[156] G. Zarpellon, J. Jo, A. Lodi, and Y. Bengio. Parameterizing branch-and-bound search trees to learn branching policies. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 3931–3939, 2021.

ACKNOWLEDGEMENTS

It was the 1st of May of the year 2020 when, while locked in my childhood home due to the Covid-19 pandemic, I started working on this thesis. One of the main lessons that the pandemic thought me was just how important human connection and support is. I am unbelievably lucky for the people who were by my side during the development of this thesis. This acknowledgements section is therefore an understatement and constitutes only a brief account of them. I shall express my gratitude more properly in person.

Firstly, I am deeply indebted to my supervisors, Professor Karen Aardal and Dr. Neil Yorke-Smith for their infinite support, understanding and flexibility, especially in the more difficult times. It is rare to find such a thing and I count myself incredibly lucky to have had the pleasure of working with you. I also had the pleasure to work with outstanding collaborators, such as Professor Andrea Lodi, Professor Wolsey, Dr. Maxime Gasse and Dr. Didier Chételat, who have inspired me and marked my academic journey.

After the confusing period of the pandemic and the following unfortunate period during which we had no office, I was extremely excited to come together with my colleagues. In our new offices, I found all the warmth and support I was lacking in the lonely days of working from home. I am extremely grateful for my colleagues in the Discrete Mathematics and Optimization group for making my PhD experience infinitely better, for everything I learned through all of you, and of course, for your friendship, which will live on beyond this PhD programme. Outside the office, I was also lucky to count on many friends who have supported me, inspired me and warmed my heart, some even from the distance. Finally, I would like to thank my loving parents who have given everything to set me on the path to success. This is all thanks to you.

Curriculum Vitæ

Lara Victoria Scavuzzo Montaña

11-08-1995	Born in Buenos Aires, Argentina.

Education

2013–2016	BSc Engineering Physics Universitat Politècnica de Catalunya, Spain
2017–2020	MSc Applied Mathematics Technische Universiteit Delft, The Netherlands
2020–2024	PhD. Applied Mathematics Technische Universiteit Delft, The Netherlands *Thesis:* – *Promotor:* Prof. dr. ir. K.I. Aardal

Awards

2023	OR Letters best paper award.

List of Publications

1. A. Prouvost, J.Dumouchelle, **L. Scavuzzo**, M.Gasse,D. Chételat, and A. Lodi. *Ecole: A gym-like library for machine learning in combinatorial optimization solvers.* arXiv preprint arXiv:2011.06069, 2020.

2. M. Gasse et al. *The machine learning for combinatorial optimization competition (ML4CO): Results and insights.* In NeurIPS 2021 Competitions and Demonstrations Track, pages 220–231. PMLR, 2022.

3. **L. Scavuzzo**, F. Y. Chen, D. Chételat, M. Gasse, A. Lodi, N. Yorke-Smith, and K. Aardal. *Learning to branch with tree MDPs.* Advances in Neural Information Processing Systems, 2022.

4. A. Iñesta, G. Vardoyan, **L Scavuzzo**, S. Wehner. *Optimal entanglement distribution policies in homogeneous repeater chains with cutoffs.* In NPJ Quantum Information 9(1)46, 2023.

5. K. Aardal, **L. Scavuzzo**, and L. A.Wolsey. *A study of lattice reformulations for integer programming.* Operations Research Letters, 51(4):401–407, 2023.

6. **L. Scavuzzo**, K. Aardal, A. Lodi, and N. Yorke-Smith. *Machine learning augmented branch-and-bound for mixed integer linear programming.* Mathematical Programming, 2024.