Learning variable selection rules for the branch-and-bound algorithm using reinforcement learning

Lara Scavuzzo Montaña

TU Delft

Learning variable selection rules for the branch-and-bound algorithm using reinforcement learning

by

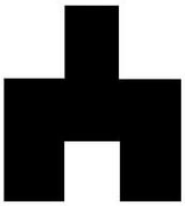
Lara Scavuzzo Montaña

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Monday January 20, 2020 at 10:00 AM.

Student number:	4723635	
Project duration:	March 11, 2019 – January 3, 2020	
Thesis committee:	Prof. dr. ir. K. I. Aardal,	TU Delft, supervisor
	Dr. ir. N. Yorke-Smith,	TU Delft, supervisor
	Dr. ir. S. E. Verwer,	TU Delft

This thesis is confidential and cannot be made public until January 20, 2021.

TU Delft



Sorry, mate. Wrong path.

- Black mirror: Bandersnatch

Abstract

Mixed Integer Linear Programming (MILP) is a generalization of classical linear programming where we restrict some (or all) variables to take integer values. Numerous real-world problems can be modeled as MILPs, such as production planning, scheduling, network design optimization and many more.

MILPs are, in fact, $\mathcal{NP}$-hard. State-of-the-art solvers use the branch-and-bound algorithm, an exact method that, in combination with a diverse mixture of heuristics, can tackle a fair range of practical problems. This algorithm sequentially partitions the search space using linear relaxations, thus creating a search tree. The exploration ends only when a solution, together with its proof of optimality, is found. The tree's size can vary dramatically depending on the approach that is used to create it and explore it.

One of the most influential decision-making strategies within the branch-and-bound algorithm is the branching rule, i.e., the criterion that is used to subdivide the search space. Currently, there is no mathematical understanding of this complex process. For this reason, all widely accepted branching rules are based on hand-crafted strategies which have been shown to perform well in practice.

The work presented in this report is part of a blossoming line of research in the intersection of Combinatorial Optimization and Machine Learning. Specifically, we take further steps in the direction of branching rule discovery through machine learning techniques. In contrast to previously proposed methods which relied on supervised learning, we take the novel approach of leveraging a Reinforcement Learning (RL) algorithm. Our goal is to achieve a data-driven acceleration of the tree search. In this thesis, we lay the fundamental groundwork for the integration of RL into the branch-and-bound process. Through the proposed model, we gain insights on the benefits and limitations of RL, while improving on the state-of-the-art branching rules for a particular class of instances.

Acknowledgements

The work presented in this thesis concludes my journey as a MSc student at TU Delft. Hereby, I would like to express my deepest gratitude to all the people that directly or indirectly contributed to this exceptional experience.

First and foremost, I would like to thank my supervisors, Professor Karen Aardal and Professor Neil Yorke-Smith. Thank you in the first place for agreeing to supervise this naïve MSc student that knocked on your doors with such an ambitious project. As time proved the challenge to be much bigger than anticipated, I was able to persevere thanks to your constant guidance and excellent feedback. I would also like to express my gratitude to the third member of my thesis committee, Professor Sicco Verwer for kindly taking the time to read this report.

I am truly honoured to have had the opportunity to collaborate with the Canada Excellence Research Chair in Data Science for Real-Time Decision-Making at Polytechnique Montréal. For this, I am extremely grateful to Professor Andrea Lodi, who made this possible in the first place. This collaboration allowed me to spend one month among outstanding researchers who at the same time are pioneers in the field concerning this thesis. I would like to thank them for the amazing warmth with which I was welcomed. It goes without saying that I am specially thankful to Didier Chételat and Maxime Gasse, for their patience towards me and constant support. I am beyond grateful for the opportunity to work side by side two exceptional researchers such as yourselves. I look forward to continuing working with you.

Taking now my thanking spree back to Europe, I would like to express my gratitude towards all the people that have contributed to make my experience in Delft so incredibly enriching, in truly every aspect of my life. A special thanks goes to my family, for supporting me and in this way allowing me to pursue my passion. Finally, thanks to Álvaro, the person who has heroically put up with my rollercoaster of emotions and my recurrent insecurities during the past nine months. Thank you for standing by my side during this process and for bringing joy to my life day after day. ¿Tú supiste?

Lara Scavuzzo
Ribadeo, January 2020

Contents

List of Acronyms

CO	Combinatorial Optimization
DQN	Deep Q-Network
FSB	Full Strong Branching
GCNN	Graph Convolutional Neural Network
GPU	Graphics Processing Unit
IL	Imitation Learning
LP	Linear Program(ming)
MDP	Markov Decision Process
MI(L)P	Mixed Integer (Linear) Program
ML	Machine Learning
PPO	Proximal Policy Optimization
RL	Reinforcement Learning
TD	Temporal Difference
TRPO	Trust Region Policy Optimization

1

Introduction

In chess, there are more than four hundred board setups after both players have made one move. After six turns, the number of possible games is over one hundred million. The number of ways in which a complete chess game can transpire is greater than the estimated number of atoms in the universe [1].

One could then be inclined to think that, each time we play chess, the resulting succession of moves has never been played before. This is not completely true, since chess games are not sampled uniformly from this immense pool of possibilities. Why is that? In reality, some actions are more probable than others. Players will build a decision-making strategy that, to a greater or lesser extent, depending on their skills and experience, will maximize their probability of winning. Fascinatingly, humans are capable of learning these tactics by experiencing a remarkably small subset of the possible game developments.

This thesis is developed in the intersection of two fields, both of which deal with the exploration of enormous sets of alternatives in search of optimality.

Combinatorial Optimization (CO) is the field of mathematics that deals with the problem of finding an optimal object within a finite set of objects. Often, these discrete domains are large configuration spaces, such that exhaustive enumeration is infeasible. Research in combinatorial optimization aims at finding efficient solution strategies that scale favourably with the problem size.

Reinforcement Learning (RL) is a subfield of machine learning (ML). In particular, it comprises algorithms that learn decision-making strategies through trial and error. During the past decade, we have witnessed extraordinary advances in machine learning, particularly in the domain of Deep Learning [2]. This revolution continues to unravel in fields as diverse as image recognition or natural language processing. More recently, deep neural network architectures have been successfully

applied in combination with reinforcement learning algorithms to learn complex behavioral strategies [3, 4, 5].

Given the enormous capabilities of ML, one may wonder if the application of ML techniques to the domain of CO could enable a prosperous collaboration. In fact, there has been a recent surge in work that aims to use machine learning algorithms to enhance or augment combinatorial optimization methodologies [6]. The role of ML is to provide assistance, specifically for tasks that are not well understood mathematically. In this way, we preserve all the theoretical guarantees while accelerating some internal processes.

This body of research can be classified into two categories [6]. On the one hand, we may assume expert knowledge about the optimization methodology. At the same time we may wish to alleviate heavy computations by using a function approximator. In this context, supervised learning algorithms can provide fast and generic approximations. On the other hand, there might be cases for which no effective strategy is known. When no expert demonstration is available, reinforcement learning can be used to discover such strategies.

Throughout this thesis, we will work under the second framework. We will address the problem of variable selection, which arises within the process of the branch-and-bound algorithm [7]. Just like an inexperienced chess player, we will attempt to learn robust strategies that lead us to our goal as fast as possible. We will do so by playing, rather than observing chess champions, although, as we will later discover, we will need some prior knowledge, acquired by demonstration.

1.1. Problem statement

The branch-and-bound algorithm is a general exact method for solving discrete optimization problems and one of the most fundamental tools in CO. This algorithm guarantees optimality through an enumerative search of the candidate space, combined with a mechanism that discards provably sub-optimal feasible regions. Notably, it is used to solve Mixed Integer Programs (MIP). Numerous real life problems can be modeled as MIPs, including some truly life-saving applications, such as ambulance coverage optimization [8] or the kidney exchange problem [9].

MIPs are $\mathcal{NP}$-hard [10]. This means that we do not know of any algorithm that is able to solve these type of problems in a time that scales like a polynomial over the problem size. Over time, branch-and-bound based solvers have incorporated a variety of auxiliary mechanisms (be heuristic rules or ancillary routines) that speed-up the solving process. It is through these algorithmic improvements that the space of tractable problems has increased dramatically [11].

Out of these secondary processes, the choice of *variable selection* rule plays a fun-

damental role in the algorithmic development, having a considerable impact in the solving time [11]. These rules formalize the procedure through which the search space is split, one of the elementary mechanisms of the branch-and-bound algorithm, known as *branching*. In spite of the substantial influence of variable selection rules in the overall efficiency, they are not well understood mathematically [12]. For this reason, branching strategies are based on domain expert knowledge and are validated through computational studies [11].

The work at hand explores the idea of using machine learning techniques to obtain new, hopefully better, branching rules. We focus on the solution of Mixed Integer **Linear** Programs (MILP). While this idea is not new, previously proposed methods leverage supervised learning algorithms [12]. Instead, we consider the novel approach of branching-rule discovery through reinforcement learning.

1.2. Research questions

Through the research presented in this thesis, we attempt to answer the question:

Can effective variable selection rules be learnt for the branch-and-bound algorithm by means of reinforcement learning techniques?

To do that, we answer the following subquestions:

1. What are the current branching rules used for branch-and-bound?
    - What can we learn from these rules, in terms of relevant decision-quality proxies? Can we use this information to engineer the appropriate features?
    - What are the shortcomings of these rules?
2. What learning technique is more appropriate for this task?
    - What are the settings under which training is feasible?
3. How can we define a suitable evaluation protocol to perform a fair comparison of the results?

1.3. Contribution

The work hereby presented constitutes one step forward in the relatively recent line of research that aims towards a better integration of ML techniques into CO methodologies. Specifically, we address the novel approach of leveraging an RL algorithm to learn a sub-task within the branch-and-bound algorithm.

The author would like to bring attention to the particular circumstances under which this work was developed. At the that time this thesis was started, there was no precedent for formulating the variable selection problem as a Markov Decision Process. Independently, the author developed a series of ideas that were remarkably similar to those later published by Gasse et al. [13] and being published by Zarpellon et al. [14]. Towards the end of this thesis, a collaboration started between the authors of the aforementioned papers and the author of this thesis. We will refer to the group of researchers that participated in this external collaboration as the DS4DM team, as they are part of the Canada Excellence Research Chair in Data Science for Real-Time Decision-Making [15]. Due to the fact that this work was part of a collective effort, we make a distinction between the overall contribution of this thesis, and the particular contributions of the author.

The novelty of this project resides in:

1. For the first time, we propose a reinforcement learning methodology which is able to improve on the initial policy and, furthermore, on the state-of-the-art branching policies.

2. In order to do so, a novel critic architecture is presented, which effectively guides the learning process by making predictions on the stage of the search.

In particular, the author's contributions (previously and with posteriority to the aforementioned publications) can be summarized in:

- The novel approach of representing the variable selection process within the framework of a Markov Decision Process.
- The selection and implementation of a suitable RL algorithm to learn a branching policy.
- The analysis and selection of appropriate features to represent the problem at hand.
- The establishment of a set of guidelines for an efficient reward mechanism and the formal definition of a set of possible functions that comply with these requirements.
- A contribution to the training procedure of the critic, in the form of a new normalization factor.
- The characterization of the proposed model through the definition of thorough experimental evaluations.

1.4. Thesis outline

This report is organized as follows. First, Chapter 2 provides an introduction to the topic of machine learning and, in particular, reinforcement learning, as well as an

overview of the literature in this field. Chapter 3 presents all necessary definitions and terminology on MILPs. Moreover, both classical and more modern variable selection rules are discussed in the context of learning to branch. The chapter concludes with an overview of benchmarking strategies and considerations for the problem at hand. The topics covered in both of these chapters follow a progression from common knowledge to state-of-the-art.

Chapter 4 comprises a description of all the methodological choices for the proposed approach. We start by first motivating the fundamental design choices (Section 4.1). This leads us to the problem formulation detailed in Section 4.2. Section 4.3 characterizes the model we adopt: the feature selection, architecture and reward mechanism. Finally, in Section 4.4 we discuss the characteristics and selection criteria for the benchmarking set that will later be used to evaluate the proposed method.

Chapter 5 is comprised of three sections, which correspond to the three phases of development of this project. The content follows a sequential progression that serves as further motivation for the model presented in Chapter 4. In particular, the first two sections discuss the experiments that were carried out before the start of the collaboration. It is in Section 5.3 that we finally address in detail the approach introduced in the preceding chapter. This proposed method is tested in a variety of settings and compared against other branching rules.

Finally, Chapter 6 concludes this thesis with a discussion of the results and a contextualization of the presented work with respect to a future outlook.

2

Fundamentals of Reinforcement Learning

This chapter presents an introduction to the field of Machine Learning and, in particular, all the fundamental concepts in Reinforcement Learning. Throughout the chapter we progress from results that are considered common knowledge in the reinforcement learning community, towards state-of-the-art methods. Along the way, we discuss the different types of algorithms that have been proposed, to then center our attention into two specific algorithmic categories, namely temporal difference learning and policy gradient algorithms.

2.1. Introduction to Machine Learning

As humans, we are capable of leveraging past experiences into valuable knowledge that helps us navigate the complexities of our world. The quest for artificially creating such an intelligent system first started as far back as the subsequent years to World War II [16]. During decades, scientists have been working to recreate the intricate mechanisms that we consider to be key in achieving intelligence. Most importantly, we have created systems capable of *learning*. In this context, we can understand learning as "the process of converting experience into expertise or knowledge" [17].

Machine Learning (ML) is the field that comprises all algorithms and statistical models that are able to infer information from data by learning to identify patterns in it. These inference mechanisms are acquired during an initial training phase. The goal of the learning process is therefore to be able to generalize, to extrapolate the obtained knowledge to new, unseen data, avoiding memorization. For this reason, machine learning algorithms require substantially large datasets. In particular, to avoid overfitting, the data is divided into three sets: the *training set* (used

for learning), the *validation set* (used to evaluate the performance during learning and consequently tune the algorithm's parameters) and the *test set* (used as a final evaluation).

Ever since the turn of the millennium, the field of Machine Learning has witnessed a series of breakthroughs that have enabled the deployment of many such algorithms in real-life applications, such as skin cancer classification [18] or autonomous robots [19]. For an overview of these advancements and its applications see, e.g., [20] or [21].

Machine learning algorithms can be classified according to the mechanism they use to find patterns in data. Primarily, one may provide the model with labeled or unlabeled data. In *supervised learning*, every data point is paired with a corresponding target, i.e., the desired solution to the task, which may be, for example, a classification or a prediction. The goal then is to tune an approximator function such that, for every input, the output is as close as possible to the target. On the contrary, in *unsupervised learning* tasks, no labeling is provided. This scheme is typically used when one wishes to find correlations in the underlying distribution of the data.

There is a third and completely different paradigm, known as Reinforcement Learning (RL). Within this framework, a learning agent must learn through interaction with its environment. Rather than providing the correct answer to compare to, we set a more general or abstract objective, which the agent must fulfill by exploring the environment and learning how to react under every given circumstance. The fulfilment of such goal is encouraged by defining a reward function that must be maximized. The ultimate objective is not finding the hidden patterns in the data, but rather exploiting them to achieve optimal behavior.

2.2. Reinforcement Learning preliminaries

In the context of reinforcement learning there are two basic interacting entities: the *agent* and the *environment*. The agent is the subject that takes actions and progressively learns to do so in a more effective way. The environment, on the other hand, is everything else that is external to the agent. When the agent chooses to perform an action over the environment, the latter reacts by changing its state and returning a reward signal, which provides the agent with some feedback about the actions it has previously taken. These interactions happen sequentially at discrete time steps.

Markov Decision Processes

Reinforcement learning is usually defined within the framework of a Markov Decision Process (MDP). A finite MDP is characterized by:

- A finite set of states $\mathcal{S}$.

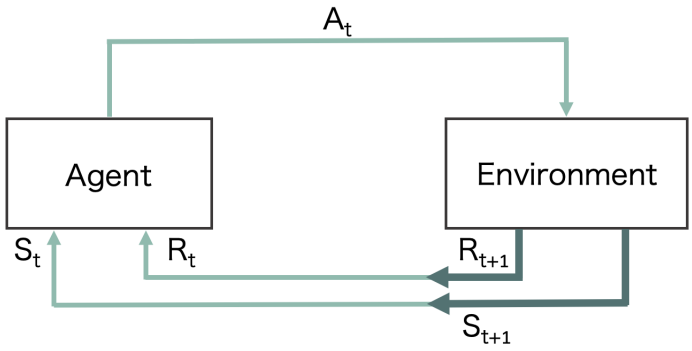


Figure 2.1: The reiforcement learning paradigm: an agent interacts with its environment by exerting an action and observing a new state and a reward signal. Figure adapted from [22].

- A finite set of actions $\mathcal{A}$.
- A finite set of rewards $\mathcal{R} \subset \mathbb{R}$
- A transition probability function

$$P(s', r|s, a) := \mathbb{P}[S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a]$$

  where, for a given time step $t \in \mathbb{N}$, $S_t \in \mathcal{S}$ is a representation of the environment's state at that time, $A_t \in \mathcal{A}$ is the action taken by the agent and $R_{t+1} \in \mathcal{R}$ represents the subsequently obtained reward.

Furthermore, the process satisfies

$$\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_t, S_{t-1}, ..., S_0]\,,$$

i.e., past state descriptions do not provide any new information. This is known as the *Markov property*. We treat the case of finite MDPs in order to simplify notation but in fact, these concepts can be easily generalized to the case where state and action spaces can be infinite.

Figure 2.1 shows a schematic representation of the interaction between the agent and the environment. The result is a sequence of states, actions and rewards known as *trajectory*:

$$S_0, A_0, R_1, S_1, A_1, ..., R_t, S_t, A_t, ...$$

In many cases, this interaction can be broken down into finite sequences, or *episodes*. For example, an agent that learns to play a game: whenever the game is over (because the agent either won or lost) a new and independent game is started. We define the *terminal state* as the state that is reached when an episode finishes. The length of an episode need not be fixed, and is represented by a random variable T. Without loss of generality, we will treat only the episodic case.

Policies and rewards

Reinforcement learning is a unique paradigm within machine learning because of the way in which it formalizes the idea of goal through a reward system. This concept is summarized in the *reward hypothesis*:

> **Reward hypothesis:** *All of the agent's goals can be described as (and therefore are equivalent to) the maximization of the cumulative sum of the received rewards.*

At each time step, the agent chooses an action based on a policy $\pi(s)$. This policy can be deterministic, in which case it becomes a mere map between states and actions. However, in many cases the policy is stochastic, meaning that each state is assigned a probability distribution function over the action space. This is,

$$\pi(a|s) := \mathbb{P}[A_t = a|S_t = s]\,.$$

The objective of the agent is then to modify its policy in order to maximize the cumulative reward, or *return*, defined as

$$G_t := \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$$

where $\gamma \in [0, 1]$ is the discount factor. In this way, γ governs the trade-off between immediate and future reward. This formalizes the idea that actions affect subsequent states and therefore have consequences that go beyond instantaneous reward.

One of the unique challenges of reinforcement learning RL is the *exploration versus exploitation dilemma*. The problem arises from the fact that we want the agent to act optimally, but it can only learn to do so by acting sub-optimally. This is, the agent must explore the environment seeking information about how to make good decisions, but at the same time it should exploit the best observed actions, given the current information. The goal of any RL algorithm is to optimally balance these two activities.

The value function

The *value function* of state s under policy π is the expected return when starting from state s and then sequentially following policy $\pi(a|s)$. Formally speaking,

$$V_\pi(s) := \mathbb{E}_{\tau\sim\pi}[G_t|S_t = s]$$

where we use τ to denote a trajectory, and we use the notation $\tau \sim \pi$ to signify that the subsequent states and actions are drawn from their respective probability distributions, i.e.,

$$S_k \sim P(S_k|S_{k+1}, A_{k+1}) \qquad A_k \sim \pi(A_k|S_k)$$

for $k \geq t$.

Similarly, the *action-value function* of policy π (also known as Q-value or Q-function) is defined as

$$Q_\pi(s,a) := \mathbb{E}_{\tau\sim\pi}[G_t|S_t = s, A_t = a]\,.$$

Notice that these two magnitudes are related through the following expression

$$V_\pi(s) = \sum_{a\in\mathcal{A}} \pi(a|s)Q_\pi(s,a)\,.$$

The optimal policy

The goal of an RL algorithm is to find the *optimal policy*. Formally, we can say that policy π is better than or equal to policy π' if $V_\pi(s) \geq V_{\pi'}(s)$ for all $s \in \mathcal{S}$. Therefore, we define the optimal policy to be the one that is better than or equal to all other policies. This policy may not be unique. However, all optimal policies have the same value-function

$$V^*(s) := \max_\pi V_\pi(s)$$

and Q-function

$$Q^*(s,a) := \max_\pi Q_\pi(s,a)\,.$$

A note on Imitation Learning

Imitation Learning (IL) algorithms are closely related to RL. Both paradigms are designed to learn an optimal decision-making strategy for an MDP. Under exceptionally challenging circumstances, such as processes with very sparse rewards or without a clear reward mechanism, RL algorithms may fail to find a good policy. Instead of extrapolating optimal behavior from a reward signal, one can learn by imitating an expert. In that case, the expert is regarded as an oracle which, for each given state, provides the best possible action. Imitation learning algorithms build a policy that tries to emulate the expert's decision-making. Behavioral cloning [23] is an example of such an algorithm, where the goal is to optimize a parametrized policy π_θ by minimizing a loss function $L(a, \pi_\theta(s))$ based on expert-acting samples $\{s_i, a_i\}_{i=1}^n$.

2.3. Algorithms for reinforcement learning

An RL algorithm consists of a set of instructions that specify how to update the agent's policy given some experience (i.e., interactions with the environment) in order to maximize the return.

Many algorithms have been proposed throughout the relatively long history of RL. For a detailed review of classical approaches we refer to [22]. In this chapter, only a brief discussion of some of these methods will be presented, together with their

connection to state-of-the-art algorithms.

First and foremost, we will define four classification criteria for RL algorithms. These four distinctive characteristics are key to understand the properties of each algorithm and also have a connection with how research in this field has evolved.

The first and second criterion are related to the type of problem we deal with, ergo they are beyond the algorithm designer's control. On the one hand, we may or may not have complete knowledge of the environment's dynamics. This is, whether or not we have access to the transition probability function $P(s', r|s, a)$. In the event that this information is available and the algorithm takes advantage of it, we say the method is *model-based*. Otherwise, it is *model-free*. On the other hand, a distinction must be made regarding the size of the state space. The first tasks that were solved using RL had a relatively small number of possible states. In order to solve these instances, agents could learn a map from every system state to an action. These are known as *tabular* methods, because the policy can be represented as a table assigning a value to each state. Certainly, this setting is very restrictive. When dealing with larger state spaces, the great memory requirements to maintain a table are not the only problem. The state space has to be exhaustively explored in order to completely fill in the table. For this reason, methods that employ function approximators are the ones typically used in practice. The strength of these algorithms relies in their ability to generalize a good decision-making strategy from experience with a small subset of the state space. Furthermore, these methods can be applied in the case of continuous state spaces.

RL algorithms can also be classified on the basis of the strategy they use to generate experience. As we previously discussed, there is a trade-off between exploration and exploitation. We want the agent to act optimally, but in order to find the optimal policy it must act sub-optimally. To tackle this problem, some implementations make use of a secondary policy that is used solely with the purpose of generating experience for the main policy to train on. This is the *on-policy* versus *off-policy* dichotomy.

Finally, there is a fourth way to classify RL algorithms. *Value-based* methods rely on obtaining an estimate of the action-value function. This estimate is then required for decision-making. The alternative is using *policy-based* methods. In that setting, no estimate of the action-value function is necessary for action selection. Instead, a parametrized policy is updated using the gradient of some performance measure.

These ideas are summarized in Table 2.1. In the reminder of this chapter, we will explore two prevalent classes of RL algorithms. Both of them work within the less restrictive setting of model-free learning.

Classification criterion	True	False
Do we have perfect knowledge of the environment's dynamics?	Model-based	Model-free
Is the state space finite and of small size?	Tabular	Approximation
Is the policy used for experience generation the same as the one being optimized?	On-policy	Off-policy
Is the policy based on an estimate of the value function?	Value-based	Policy-based

Table 2.1: Types of reinforcement learning algorithms.

2.3.1. ε-greedy policies

Before proceeding to analyze some of the most important RL algorithms, let us clarify an important concept: that of ε-greedy policies.

Within value-based methods, there is a variety of approaches to defining a policy based on the action-value estimates. A simple idea could be to choose a greedy policy: take the action that maximizes the Q-value at each step. By doing this, policies tend to shift towards a deterministic behavior. This could lead to a lack of exploration of the search space, specifically in the case of on-policy methods. A common strategy to solve this problem is to choose the Q-value maximizing action only with probability $1 - \varepsilon$, where ε is a very small number. The rest of the times, an action is taken at random. This is known as ε-greedy policy.

2.4. Temporal Difference learning

Temporal Difference learning (TD-learning) [24] is a class of model-free value-based RL algorithms. This means that, under this setting, the agent learns an estimate of the action-value function, without assuming any knowledge about the environment's dynamics. In contrast to other methods, complete episodic roll-outs are not needed. Instead, updates are based on estimated returns. In particular, updates are of the form

$$\hat{V}_\pi(S_t) \leftarrow (1-\alpha)\hat{V}_\pi(S_t) + \alpha\hat{G}_t$$
$$\hat{V}_\pi(S_t) \leftarrow \hat{V}_\pi(S_t) + \alpha(\hat{G}_t - \hat{V}_\pi(S_t))$$
$$\hat{V}_\pi(S_t) \leftarrow \hat{V}_\pi(S_t) + \alpha\left(R_{t+1} + \gamma\hat{V}_\pi(S_{t+1}) - \hat{V}_\pi(S_t)\right)$$

where α is a learning parameter and we use a circumflex to emphasize that the magnitudes are estimates. The rationale behind this update is to use the return G_t as a target for the estimate $\hat{V}_\pi$, given that the latter is the expected value of the former. As we have previously remarked, we avoid having to calculate full trajectories by using an estimate of the return. This new target is based on the function we are trying to approximate.

The strategy of updating estimates on the basis of the estimates themselves is called *bootstrapping*. The quantity $R_{t+1} + \gamma \hat{V}_\pi(S_{t+1}) - \hat{V}_\pi(S_t)$ is a sort of error measure of the estimate. The same reasoning can be applied to Q-values:

$$Q_\pi(S_t, A_t) \leftarrow Q_\pi(S_t, A_t) + \alpha \left(R_{t+1} + \gamma Q_\pi(S_{t+1}, A_{t+1}) - Q_\pi(S_t, A_t) \right) .$$

Notice that these are still estimated magnitudes, but we have omitted the circumflex to simplify notation.

TD-learning can be generalized by using an n-step bootstrapping, for which the estimated return would read

$$\hat{G}_t = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n \hat{V}_\pi(S_{t+n}) .$$

In the reminder of this section, we will discuss one particular form of TD-learning with 1-step bootstrapping: Q-learning. For an overview of other related methods, we refer to Chapter 6 of [22].

2.4.1. Q-learning

The Q-learning algorithm [25] is considered "one of the early breakthroughs in reinforcement learning" [22]. It uses the TD-learning update in an off-policy fashion: the target estimate is governed by a greedy policy. In mathematical terms, this translates as

$$Q_\pi(S_t, A_t) \leftarrow Q_\pi(S_t, A_t) + \alpha \left(R_{t+1} + \gamma \max_{a \in \mathcal{A}} Q_\pi(S_{t+1}, a) - Q_\pi(S_t, A_t) \right) .$$

This update directly approximates the optimal action-value function Q^*, even if the experience samples $(S_t, A_t, R_{t+1}, S_{t+1})$ are generated using a different policy, e.g., ε-greedy. For a proof of convergence, see [25].

Q-learning with neural networks

Classical Q-learning is restricted to the tabular case, but the update principles it is based on can be easily extended to a domain with function approximators. Tsitsiklis et al. [26] proved the convergence of this method under linear parameterizations of the Q-function. However, they also show that non-linear approximation functions are prone to instabilities and even divergence from optimal behavior.

In spite of this, several approaches have been successful in using non-linear parameterizations for Q-value estimation. Tesauro et al. [27] were the first to show the potential of using neural networks for this purpose. In this context, the Q-learning update,

$$R_{t+1} + \gamma \max_{a \in \mathcal{A}} Q_\pi(S_{t+1}, a) - Q_\pi(S_t, A_t)$$

represents an error measure that can be used to apply gradient decent on the network's parameters. Tesauro et al. use this scheme to train an algorithm capable of playing backgammon using a feature representation of the board state. The agent is trained by playing against itself. This strategy proves to be superior with respect to networks trained on data generated by human-players.

Riedmiller et al. [28, 29] introduced the idea of *batch reinforcement learning*. Instead of updating the Q-function estimate after each transition step, they first generate experience by gathering several of these steps, to then use them together for a batch update. This allows the use of more advanced algorithms for continuous optimization such as Rprop [30].

In spite of these advances, RL was, for a long time, limited to low-dimensional state spaces, where features had to be hand-crafted by domain experts. One of the biggest breakthroughs in modern reinforcement learning came with the work of Mnih et al. [31]. This seminal work bridged the gap between Q-learning and deep learning. The authors identify three sources of instabilities for Q-learning with neural networks that together prevent convergence to the optimal policy. They are:

- Highly correlated observations in the trajectories.
- Correlation between the Q-values and the target values for the update.
- The fact that small changes in the Q-function can dramatically change the data distribution the agent trains on.

To tackle these problems, the authors propose two novel strategies:

- **Experience replay:** as the agent explores the environment, observations are gathered and stored in memory. During learning, experience batches are sampled at random from the memory's dataset of observations.
- **Target network freezing:** they use a twin network, the target network, to produce target values for the update. This target network is frozen for several training steps, after which it is updated with the policy network's weights.

With this methodology, the loss function becomes

$$L(\theta) = \mathbb{E}_{(s,a,r,s') \sim U(D)} \left[\left(r + \gamma \max_{a'} Q(s', a'; \theta^T) - Q(s, a; \theta) \right)^2 \right]$$

where θ and θ^T are the policy and target network parameters, respectively, and D is the experience dataset.

Thanks to these advances, the authors were able to successfully train an agent that learns from raw sensory data (frames from classic Atari games), outperforming all other attempts of learning the same task at the time of the publication. In short, this means that, through this training scheme, the network derived an informative representation of the state upon which it was able to generalize past experiences to a successful decision-making strategy. This approach to learning came to be commonly known as deep Q-network (DQN). Algorithm 1 describes the whole training process on a high level.

The work in [31] set a precedent for modern research in Deep Reinforcement Learning. More recent research on value-based methods expands on the ideas proposed by the authors (see, e.g., [32] and [33]).

Algorithm 1 DQN

Input: A parametrized Q-function $Q(s, a; \theta)$, a maximum number of episodes N, a maximum number of steps T, a target update period P and a memory size M.

Initialize replay memory D to capacity M
Initialize network parameters θ randomly
Initialize target network parameters $\theta^T = \theta$
for episode=1,...,N **do**
    Observe the initial state S_0
    for t=0,...,T **do**
        Sample X from a Bernouilli distribution with success probability ε
        if $X = 1$ **then**
            Select an action A_t at random
        else
            $A_t = \arg\max_a Q(S_t, a; \theta)$
        Execute action A_t and observe R_{t+1} and S_{t+1}
        Store transition $(S_t, A_t, R_{t+1}, S_{t+1})$ in the replay memory D
        Randomly sample a transition minibatch $(S_j, A_j, R_{j+1}, S_{j+1})$ from D.
        Set $y_j = \begin{cases} R_{j+1} & \text{if } S_{j+1} \text{ is a terminal state} \\ R_{j+1} + \gamma \max_a Q(S_{j+1}, a; \theta^T) & \text{otherwise} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(S_j, A_j, \theta))^2$ with respect to the parameters θ
        Every P steps, $\theta^T \leftarrow \theta$
    end for
end for

2.5. Policy gradient algorithms

We have seen that value-based methods are characterized by policies that rely on an estimate of the action-value function $Q_\pi(s, a)$. In this section we will consider a new paradigm: methods that learn a parametrized policy $\pi_\theta(a|s; \theta)$ which does not

rely on action-value estimates for action selection. This does not mean that they cannot exploit state value estimates during learning, but these are not used in the decision-making process.

The algorithms we will explore in this section are based on the gradient of some performance measure $J(\theta)$ of the policy parameters θ. The objective is to maximize the performance, hence the updates are of the form

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta).$$

Algorithms that follow this procedure are known as *policy gradient methods*.

2.5.1. The REINFORCE algorithm

An obvious choice for the performance measure is

$$J(\theta) = \sum_{s \in S} d_{\pi_\theta}(s) V_{\pi_\theta}(s)$$

where $d_{\pi_\theta}(s)$ is the stationary distribution of Markov chain for π_θ. This poses a challenge. The performance depends on the policy and the distribution of states given that policy, both of which depend on the policy parameters θ. The policy parametrization is chosen such that the dependence on θ can easily be computed. However, the relation between the state distribution and the policy parameters depends on the environment and is therefore unknown. Fortunately, there is a way to circumvent this problem using the result of the policy gradient theorem.

Theorem 1 (Policy gradient theorem)**.**

$$\nabla_\theta J(\theta) \propto \mathbb{E}_{\substack{a \sim \pi_\theta \\ s \sim d_{\pi_\theta}}} \left[Q_{\pi_\theta}(s,a) \nabla_\theta \ln \pi_\theta(a|s) \right].$$

We refer to Chapter 13 of [22] for a proof of this theorem.

The classical REINFORCE algorithm [34] uses the result of the policy gradient theorem, and performs a Monte-Carlo estimation of the right-hand-side of the expression. In particular, the expectation is approximated by sampling trajectories. Furthermore, the Q-function is substituted by the return G_t (recall that $Q_{\pi_\theta}(S_t, A_t) = \mathbb{E}_{\tau \sim \pi_\theta}[G_t | S_t, A_t]$). In summary, the performance gradient is approximated with

$$\hat{\mathbb{E}}[G_t \nabla_\theta \ln \pi_\theta(A_t|S_t)]$$

where we use $\hat{\mathbb{E}}$ to denote the Monte-Carlo estimation of the expected value. The REINFORCE method is outlined in Algorithm 2.

A common variation of the REINFORCE algorithm is to subtract a baseline $b(s)$, to instead use updates of the form

$$\hat{\mathbb{E}}[(G_t - b(s)) \nabla_\theta \ln \pi_\theta(a|s)] \tag{2.1}$$

Algorithm 2 REINFORCE

Input: A differentiable parametrized policy $\pi_\theta(a|s;\theta)$, a maximum number of episodes N and a step size α.

```
i = 0
while i < N do
    Generate trajectory S_0, A_0, R_1, S_1, A_1, ..., S_T
    for t=1,...,T do
        G_t ← Σ_{k=t+1}^{T} γ^{k-t-1} R_k
        θ ← θ + α γ^t G_t ∇_θ π_θ(a|s;θ)
    end for
    i ← i + 1
```

Notice that

$$
\begin{aligned}
&\mathbb{E}_{\substack{a\sim\pi_\theta \\ s\sim d_{\pi_\theta}}} \left[\left(Q_{\pi_\theta}(s,a) - b(s) \right) \nabla_\theta \ln \pi_\theta(a|s) \right] \\
&= \mathbb{E}_{s\sim d_{\pi_\theta}} \left[\sum_{a\in\mathcal{A}} \pi_\theta(a|s) \left(Q_{\pi_\theta}(s,a) - b(s) \right) \nabla_\theta \ln \pi_\theta(a|s) \right] \\
&= \mathbb{E}_{s\sim d_{\pi_\theta}} \left[\sum_{a\in\mathcal{A}} \pi_\theta(a|s) \left(Q_{\pi_\theta}(s,a) - b(s) \right) \frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} \right] \\
&= \mathbb{E}_{s\sim d_{\pi_\theta}} \left[\sum_{a\in\mathcal{A}} \left(Q_{\pi_\theta}(s,a) - b(s) \right) \nabla_\theta \pi_\theta(a|s) \right] \\
&= \mathbb{E}_{s\sim d_{\pi_\theta}} \left[\sum_{a\in\mathcal{A}} Q_{\pi_\theta}(s,a) \nabla_\theta \pi_\theta(a|s) \right] - \mathbb{E}_{s\sim d_{\pi_\theta}} \left[b(s) \nabla_\theta \sum_{a\in\mathcal{A}} \pi_\theta(a|s) \right] \\
&= \mathbb{E}_{\substack{a\sim\pi_\theta \\ s\sim d_{\pi_\theta}}} \left[Q_{\pi_\theta}(s,a) \nabla_\theta \ln \pi_\theta(a|s) \right] - \mathbb{E}_{s\sim d_{\pi_\theta}} \left[b(s) \nabla_\theta 1 \right] \\
&= \mathbb{E}_{\substack{a\sim\pi_\theta \\ s\sim d_{\pi_\theta}}} \left[Q_{\pi_\theta}(s,a) \nabla_\theta \ln \pi_\theta(a|s) \right]
\end{aligned}
$$

Therefore, we can say that, for any $b(s)$,

$$
\nabla_\theta J(\theta) \propto \mathbb{E}_{\substack{a\sim\pi_\theta \\ s\sim d_{\pi_\theta}}} \left[\left(Q_{\pi_\theta}(s,a) - b(s) \right) \nabla_\theta \ln \pi_\theta(a|s) \right]
$$

which assures that the estimator in Eq 2.1 is unbiased.

The only requirement for the baseline is that it must not depend on the action. Given that the baseline can be zero for all states, this constitutes a generalization of the REINFORCE algorithm. This simple trick can help reduce the variance introduced by the estimated returns [22]. A common choice for the baseline is an estimate of the value function $V_{\pi_\theta}(s)$.

2.5.2. Actor-critic algorithms: A2C

The REINFORCE algorithm (both with and without baseline) suffers from slow convergence due to the high variance of its estimates. Furthermore, we must generate full trajectories in order to perform a parameter update, which can be inconvenient in practice.

In contrast, actor-critic methods address these problems by combining the advantages of policy gradient methods and temporal difference methods. Just like in REINFORCE, there is a parameterized policy: the actor. However, in this case, there is a secondary function approximator which estimates the value function: the critic. We will denote this second parameterization as $V_{\pi_\theta}(s;\phi)$, where ϕ are the critic parameters.

Notice that if we use a REINFORCE method with baseline and we choose $b(s) = V_{\pi_\theta}(s)$, we would also need to estimate the value function. However, we do not consider REINFORCE with baseline to be an actor-critic method. The key difference between these is that actor-critic algorithms use the estimated state value for bootstrapping. In particular, while for REINFORCE we estimated $Q_{\pi_\theta}(s,a)$ with G_t, actor-critic methods use $R_{t+1} + \gamma V_{\pi_\theta}(S_{t+1};\phi)$ instead, so that full trajectories are no longer needed. In detail, the update becomes

$$\theta \leftarrow \theta + \alpha \left(R_{t+1} + \gamma V_{\pi_\theta}(S_{t+1};\phi) - V_{\pi_\theta}(S_t;\phi)\right) \nabla_\theta \ln \pi_\theta(A_t|S_t;\theta)\,. \tag{2.2}$$

In the RL literature, it is common to work with the *advantage*, defined as

$$\mathcal{A}_{\pi_\theta}(S_t, A_t) := Q_{\pi_\theta}(S_t, A_t) - V_{\pi_\theta}(S_t)\,.$$

Intuitively, this magnitude represents the "advantage", whether positive or negative, of taking action A_t instead of the expected action under policy π_θ. Likewise, we can define an estimated advantage

$$\hat{\mathcal{A}}_t = R_{t+1} + \gamma V_{\pi_\theta}(S_{t+1};\phi) - V_{\pi_\theta}(S_t;\phi)\,.$$

We can therefore rewrite 2.2 to read

$$\theta \leftarrow \theta + \alpha\, \hat{\mathcal{A}}_t \nabla_\theta \ln \pi_\theta(A_t|S_t;\theta)\,.$$

This gives rise to the name of the algorithm that uses these updates: advantage actor critic (A2C) [3].

2.5.3. Actor-critic algorithms: PPO

In an effort to improve the robustness and scalability of vanilla policy gradient algorithms (such as A2C), Schulman et al. developed the Trust Region Policy Optimization (TRPO) algorithm [35]. In contrast with the REINFORCE paradigm,

the authors choose a different performance measure $J(\theta)$ and prove that its maximization guarantees policy improvement (we refer to the original paper for details). Specifically, they propose to solve

$$\begin{aligned} \underset{\theta}{\text{maximize}} \quad & \hat{\mathbb{E}}\left[\hat{\mathcal{A}}_t \frac{\pi_\theta(A_t|S_t)}{\pi_{old}(A_t|S_t)}\right] \\ \text{subject to} \quad & \hat{\mathbb{E}}\left[\mathrm{KL}\left(\pi_\theta(\cdot|S_t), \pi_{old}(\cdot|S_t)\right)\right] \leq \delta \end{aligned} \tag{2.3}$$

for some δ, where we have used π_{old} to denote the policy before the update, and $\mathrm{KL}(\cdot)$ is the Kullback-Leibler divergence between the two probability distributions. Recall that $\hat{\mathbb{E}}$ represents an estimated expectation, calculated through averaging on samples of the type $(S_t, A_t, R_{t+1}, S_{t+1})$. The enforcement of the constraint in Eq 2.3 ensures that the policy update is small enough, so that the assumptions that this algorithm relies on remain true. TRPO was shown to provide a scalable and robust strategy to learning challenging tasks, such as continuous control problems. However, as the authors themselves point out in a posterior paper, TRPO "is relatively complicated and is not compatible with architectures that include noise (such as dropout) or parameter sharing (between the policy and value function, or with auxiliary tasks)" [36].

In order to alleviate these shortcomings, they propose the Proximal Policy Optimization (PPO) algorithm [36]. Instead of imposing a constraint, the magnitude of the update is limited. In particular, a PPO update works in the following way. Define

$$r_t(\theta) = \frac{\pi_\theta(A_t|S_t)}{\pi_{old}(A_t|S_t)}$$

and

$$r_t^{clip}(\theta) = \begin{cases} 1-\varepsilon & \text{if } r_t(\theta) < 1-\varepsilon \\ 1+\varepsilon & \text{if } r_t(\theta) > 1+\varepsilon \\ r_t(\theta) & \text{otherwise} \end{cases}$$

where ε is a parameter. This is, r_t^{clip} is a clipped version of r_t in the interval $[1-\varepsilon, 1+\varepsilon]$. Finally, the function to maximize becomes

$$\hat{\mathbb{E}}\left[\min\left(r_t(\theta)\hat{\mathcal{A}}_t\,,\, r_t^{clip}(\theta)\hat{\mathcal{A}}_t\right)\right].$$

The function clipping prevents very large policy updates, achieving the same effect as the TRPO constraint enforcement. In this way, PPO attains similar performance while being much simpler to implement, tune and deploy. In practice, the objective to be minimized becomes

$$L(\theta) = L^{\mathrm{A}}(\theta) + c_1 L^{\mathrm{C}}(\theta) - c_2 S(\theta)$$

where c_1 and c_2 are parameters,

$$L^{\mathrm{A}}(\theta) = -\hat{\mathbb{E}}\left[\min\left(r_t(\theta)\hat{\mathcal{A}}_t\,,\, r_t^{clip}(\theta)\hat{\mathcal{A}}_t\right)\right]$$

is the actor's loss, $L^{\mathrm{C}}(\theta)$ is the critic's loss and $S(\theta)$ is an entropy bonus that prevents the policy from becoming deterministic and therefore encourages exploration. The critic loss is usually chosen to be the square-error between the current estimate $V(S_t;\theta)$ and an estimate of the return G_t. Notice that the actor and the critic may or may not share parameters, hence considering a common parameter vector θ is a generalization that encompasses both cases.

Algorithm 5 shows the PPO procedure schematically. Schulman et al. adopt a similar strategy to the DQN algorithm, where fixed-length trajectories are used to generate samples (see Algorithm 1). However, in this case, samples are not stored in a replay memory but rather discarded after use.

Algorithm 3 PPO

Input: A differentiable parametrized policy $\pi_\theta(a|s;\theta)$, a differentiable parameterized value function $V_\pi(s;\theta)$, a maximum number of epochs N and a maximum number of time steps T.

for epoch=1,...,N **do**
  for $t = 1, ..., T$ **do**
    Draw $A_t \sim \pi_{old}(\cdot|S_t)$
    Obtain sample $(S_t, A_t, R_{t+1}, S_{t+1})$
  end for
  Calculate advantages $\hat{\mathcal{A}}_1, ..., \hat{\mathcal{A}}_T$
  Perform stochastic gradient decent over θ with loss $L(\theta)$ over sample minibatches.
  $\pi_{old} \leftarrow \pi_\theta$
end for

3

Mixed Integer Linear Programs

In this chapter we present all the necessary definitions and terminology regarding Mixed Integer Linear Programs. After discussing all fundamental concepts, an overview both classical and more modern variable selection rules will be provided, always in the context of learning to branch. The chapter concludes with an overview of benchmarking strategies and considerations for the problem at hand.

3.1. Fundamentals

3.1.1. Definitions

A mixed integer linear program (MILP) is a programming problem in which both constraints and objective function are linear, and furthermore some of the variables are restricted to take integer values. Formally, it is defined as follows:
Given a matrix $A \in \mathbb{R}^{m \times n}$, vectors $b \in \mathbb{R}^m$ and $c \in \mathbb{R}^n$, and a subset $I \subseteq \{1, 2, ..., n\}$, the associated MILP is the problem of finding z^* such that

$$z^* = \min\{c^T x \mid x \in \mathbb{R}^n,\, Ax \leq b,\, x_i \in \mathbb{Z} \text{ for all } i \in \mathcal{I}\}\,. \tag{3.1}$$

The entries A, b and c are usually assumed to be rational. The set $S = \{x \mid x \in \mathbb{R}^n, Ax \leq b, x_i \in \mathbb{Z} \text{ for all } i \in \mathcal{I}\}$ is known as the feasible set. A feasible solution $x^* \in S$ is called *optimal* if $c^T x^* = z^*$.

Given S, a *linear relaxation* of S is a set $S' = \{x \mid x \in \mathbb{R}^n, A'x \leq b'\}$ for some A' and b' such that $S \subseteq S'$. Therefore, a *linear programming relaxation* (LP relaxation) of a MILP with feasible set S is defined as

$$z' = \min\{c^T x | x \in S'\}\,. \tag{3.2}$$

Out of all the linear relaxations of S, the natural linear relaxation is defined as $S_0 = \{x|Ax \leq b, x \in \mathbb{R}^n\}$. This is, in order to obtain S_0 we merely drop the integrality constraint from S. With a slight abuse of notation, we will refer to the natural linear relaxation as the linear relaxation of S.

In this chapter we will be discussing linear programs, both with and without integrality constraints. For this matter, let us define some important terminology. For a given MILP with feasible set S and linear relaxation S', x is an LP *solution* if $x = \text{argmin}\{c^T x | x \in S'\}$. Furthermore, x is *integer feasible* if $x \in S$. The *objective value* z is the value that the objective function takes at such solution, i.e., $z = c^T x$.

3.1.2. Solution methods

The discrete nature of MILPs may seem advantageous to the inexperienced mind, as one may use exhaustive enumeration to find the optimal solution. Nonetheless, this is in fact the recurrent challenge faced in combinatorial optimization problems. The obstacle lies in the rapid growth of the number of solutions with the problem size, to such an extent that one could be dealing with an amount of possibilities of the order of magnitude of the number of atoms in the universe. From an algorithmic perspective, the question then becomes: is there a more efficient way of solving MILPs than complete enumeration?

MILPs are, in fact, $\mathcal{NP}$-hard [10]. In practice, this means that there is no (known) polynomial time algorithm for finding an optimal solution. Fortunately, modern solvers are capable of solving reasonably sized MILPs within an acceptable time frame. State-of-the-art software relies on two basic concepts, that will be explored in this section.

Cutting-plane methods

Cutting planes were one of the first proposed methods to tackle MILPs [37, 38]. The key idea is to start from the natural linear relaxation of the feasible set S and progressively tighten this relaxation by discarding the regions that are not part of the convex hull of S. This approach reduces the MILP into a successive resolution of linear programs.

The cuts are generated in the following way. Consider a feasible set S and its natural linear relaxation S_0. Let x^0 be the optimal solution to the linear program defined by S_0. Given that $S \subseteq S_0$, we know that $c^T x^0$ sets an upper bound for the optimal value of the original MILP. In fact, if $x^0 \in S$ then x^0 is also an optimal solution to the original MILP.

Instead, consider the non-trivial case where $x^0 \notin S$. Then, we define a *cutting plane* to be an inequality of the type $\alpha^T x \leq \beta$, for some $\alpha \in \mathbb{R}^n$ and $\beta \in \mathbb{R}$, such that $\alpha^T x \leq \beta$ for all $x \in S$ (i.e. it is *valid* for S) and $\alpha^T x^0 > \beta$.

Define

$$S_1 = S_0 \cap \{x | \alpha^T x \leq \beta\}\,.$$

Clearly, S_1 is also a linear relaxation of S, and furthermore $S \subseteq S_1 \subseteq S_0$. The recursive application of the cutting-plane paradigm reduces the search space until the optimal solution is found. This is guaranteed to happen because the successively smaller LP relaxations converge into the convex hull of S.

Though mathematically elegant, this method is known to be impractical [39]. In particular, one may require exponentially many cuts to reach the convex hull of S. In addition, numerical errors can slow down convergence or even lead to cutting off the optimal solution.

Branch and bound

The branch-and-bound (B&B) algorithm was first developed in the 1960s as a general purpose algorithm for tackling discrete optimization problems [7]. The core idea behind it is the successive partition of the feasible set into smaller problems. What at first may seem to be an unnecessarily elaborate enumerative scheme, is actually a clever strategy to navigate the feasible set, exploiting mechanisms that attempt to control the exponential nature of the search.

The branching mechanism

Let S_0 be the natural linear relaxation of the feasible set S. Let x^0 be an optimal solution. Once more, we consider the non-trivial case in which $x^0 \notin S$. This implies that at least one of the components in x^0 violates the integrality constraints. Let $\mathcal{J} \subseteq \mathcal{I}$ be the subset of integral variables such that for all $j \in \mathcal{J}$ the j-th component of x^0 (denoted as x_j^0 from now on) is not integer. We will refer to $\mathcal{J}$ as the set of candidate variables. Choose arbitrary $j \in \mathcal{J}$. Notice that any $x \in S$ satisfies

$$x_j \leq \lfloor x_j^0 \rfloor \qquad \text{or} \qquad x_j \geq \lceil x_j^0 \rceil\,. \tag{3.3}$$

These inequalities allow us to partition the feasible set into two subproblems (or branches), excluding x^0.

$$S_1 = S_0 \cap \{x \,|\, x_j \leq \lfloor x_j^0 \rfloor\} \qquad \text{and} \qquad S_2 = S_0 \cap \{x \,|\, x_j \geq \lceil x_j^0 \rceil\}\,.$$

Figure 3.1 shows a two-dimensional example of this mechanism. The optimal solution must live in one of the two subproblems. This process is then repeated by solving the LP relaxations of the subproblems. It is also possible to split the problem using other types of inequalities, or even subdivide it into more than two problems (see, e.g., [40], [41] or [42]), although these approaches are less common.

These successive divisions create a tree-like structure, where the original problem is represented by a root node N_0, and the hierarchical nature of the splitting is captured by the parent-child relationships between nodes. The complexity of the

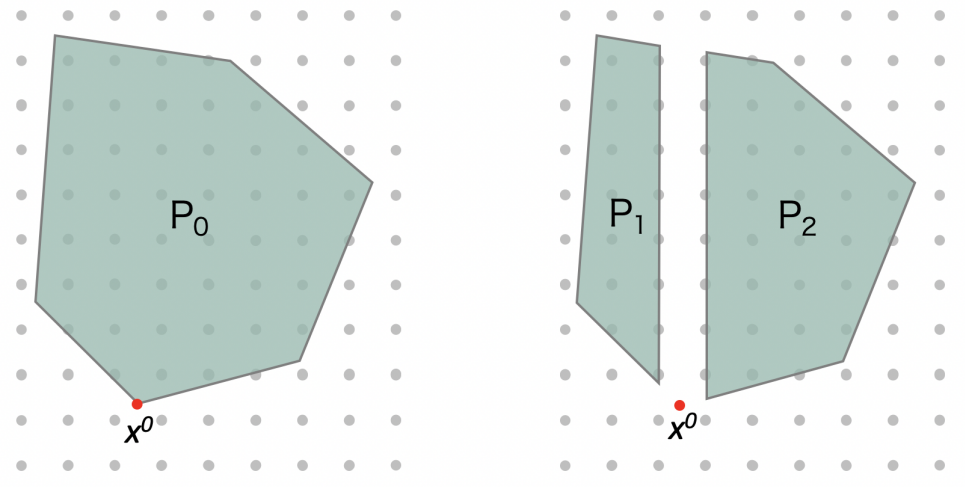


Figure 3.1: Branching: the original LP relaxation is split into two subproblems to exclude the solution that violates integrality constraints.

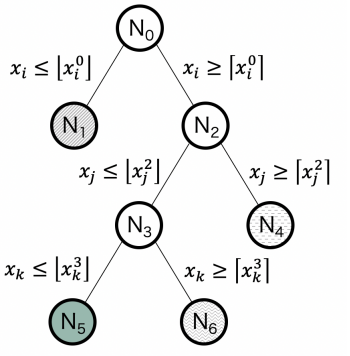
children nodes is the same as the parent node's, but the subproblems become successively smaller.

The pruning mechanism

Simply exploiting the subdivision process would yield a basic enumerative scheme. Instead, we must incorporate a mechanism that allows us to rule out unpromising branches. Clearly, whenever a node's relaxation is infeasible, it is not further subdivided. The same happens with nodes whose optimal solution to the relaxed problem satisfies the integrality constraints, i.e., when the optimum is integer feasible.

However, there is a third mechanism through which a branch can be excluded from the search, or *pruned*. Along the search, whenever an integer feasible solution is found, the cor-

Figure 3.2: Example of a branch-and-bound tree. The original problem is represented by the root node N_0. Each branching process produces children nodes.



responding objective value is stored. These solutions set an upper bound to the optimum. The best known upper bound, i.e., the lowest, is referred to as the *incumbent*. On the other hand, notice that, for any given node, a solution of the LP relaxation sets a lower bound to integer feasible solutions in that node or its descendants. Therefore, if the solution to an LP relaxation is greater than the incumbent, such node can be immediately discarded, since we have certified that further subdivision will not yield a better solution.

Figure 3.2 shows an example of a branch and bound tree. The original problem (N_0) is split into two subproblems by branching on variable x_i. It could happen that the solution to N_1 is integer feasible. If this is the case, we store the solution and prune the node. The search continues by branching on N_2. One of the children, N_4 is pruned by bound, i.e., $c^T x^4 \geq c^T x^1$. When branching on node N_3, we may find N_5 to be integer feasible and N_6 to be infeasible. If $c^T x^1 \geq c^T x^5$ then x^5 is the optimal solution.

A concise description of this process is shown in Algorithm 4.

Algorithm 4 Branch and Bound

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

Initialization: $\mathcal{L} = \{N_0\}$, $z^{inc} = +\infty$, $x^{inc} = \emptyset$
if $\mathcal{L} = \emptyset$ **then return** z^{inc} and x^{inc}
Choose $N_k \in \mathcal{L}$
$\mathcal{L} \leftarrow \mathcal{L} \setminus \{N_k\}$
Solve the LP relaxation of N_k
if infeasible **then goto** 2
Let x^k be the optimal solution and z^k the objective value
if $z^k \geq z^{inc}$ **then goto** 2
if x^k is integer feasible **then**
    $z^{inc} \leftarrow z^k$
    $x^{inc} \leftarrow x^k$
else branch:
    Choose $i \in I$ such that $x_i^k \notin \mathbb{Z}$.
    Split N_k into N^+ and N^- using the inequalities in Eq 3.3 on variable i
    $\mathcal{L} \leftarrow \mathcal{L} \cup \{N^+, N^-\}$
goto 2

Solution methods in practice

We have discussed two strategies that can be used to solve MILPs. However, using one of these methods alone can still be highly inefficient, leaving large problems out of reach. Modern solvers usually combine a branch-and-bound baseline with cutting planes that tighten the linear relaxations at the nodes. In fact, today's leading solvers (such as CPLEX [43], SCIP [44] or GUROBI [45]) make use of a large collection of rules and heuristics that accelerate the search, such as:

- **Presolving**: presolving techniques transform the original problem (the root node) into an equivalent one that is expected to be easier to solve. They achieve this mainly through three lines of attack. First, they attempt to find ways to reduce the size of the problem, by discarding irrelevant information. Second, they exploit implicit data to tighten the variable bounds. Third, they extract useful information that can be of service during the search, such as implications or sub-structures. A detailed overview of the fundamental concepts of presolvers can be found in [46].
- **Primal heuristics**: primal heuristics are incomplete methods. That means that, in contrast to the branch-and-bound algorithm, there is no guarantee that they will find a solution, let alone an optimal one. They trade this guarantee for a faster execution. Primal heuristics can be used independently or as an auxiliary tool to the branch-and-bound algorithm, given that finding feasible solutions early on in the search helps to prune the tree and therefore speed-up the process. Furthermore, under many circumstances, the user may be willing to compromise optimality and may be satisfied with a lower quality feasible solution. There is an extensive collection of primal heuristic rules, such as the feasibility pump [47]. For an overview of some of the most popular primal heuristics see, e.g., [48].

These are some of the common techniques used in practice to accelerate the branch-and-bound search. Years of computational experimentation have lead to an extensive body of knowledge on how to effectively combine them [11]. In addition to these ancillary methods, there are two decision-making processes that arise within the basic branch-and-bound routine. On the one hand we have the problem of **node selection** (see step 3 of Algorithm 4), which consists on choosing the next subproblem (node) to process. On the other hand, at each branching step, one must decide on which criterion to use to further subdivide a problem. The latter is known as **variable selection** (step 13 of Algorithm 4), as the decision usually comes down to selecting one fractional variable.

There is a lack of mathematical understanding of the underlying processes governing node and variable selection [12]. There has only been one recent attempt at theoretically analyzing the branching mechanism [49]. This causes the quest for efficient strategies to remain an open question. Once again, modern solvers rely on heuristic decision-making rules that have been shown to work well in practice. In this thesis, we focus on the variable selection problem. The reminder of this chapter presents an overview of the literature regarding variable selection rules. For a survey on node selection techniques we refer to [50] and [12].

3.2. Branching on variables

Branching, i.e., subdividing the feasible set, is the fundamental mechanism of the branch-and-bound algorithm. In order to do this, we need to find a linear inequality

$\alpha^T x \leq \beta$ that defines two subsets S_- and S_+ of the parent node's feasible region S. This is,

$$S_- = S \cap \{x : \alpha^T x \leq \beta\}$$

$$S_+ = S \cap \{x : \alpha^T x \geq \beta + 1\}$$

with α_i and β_i being realatively primes for all $i \in \{1, ..., n\}$. We require the inequality to be linear so that the children nodes continue to be MILPs. The vast majority of solvers use *trivial inequalities*, i.e., those that involve one variable x_i ($i \in \mathcal{I}$) and use the optimal solution to the node's LP relaxation $\hat{x}$ to define

$$S_- = S \cap \{x : x_i \leq \lfloor \hat{x}_i \rfloor\}$$

$$S_+ = S \cap \{x : x_i \geq \lceil \hat{x}_i \rceil\}$$

This procedure is known as branching on variables. Other types of inequalities are rare, but have been used successfully in some cases, see, e.g., [40, 51, 52].

A good branching selection rule must serve two goals. They are both related to efficiency, but are usually at odds with each other. On the one hand, a branching rule must **create nodes efficiently**. This is, it must make branching decisions that lead to small search trees. On the other hand, it is important to **process each node efficiently**. Each branching decision cannot come at the cost of high computational overhead, or else the overall solving time would exceed practical requirements.

In summary, a good branching rule must wisely evaluate the potential of each of the branching decisions. This raises a new question: what is an effective way of assessing the quality of a branching candidate? Ever since the branch-and-bound algorithm was first proposed, researchers have tried to answer this question in diverse ways. In the following section we will explore some of the most relevant and popular branching rules.

3.3. Classical branching strategies: LP-bound degradation

During the search process, there are two meaningful magnitudes that should be tracked. First of all, as integer feasible solutions are found, we must keep track of the *incumbent*, i.e., the lowest objective value achieved by these solutions. The incumbent constitutes an upper bound to the optimal solution. On the other hand, each open node has an associated lower bound, defined as the objective value of its parent's LP relaxation. The lowest of these bounds, known as *best bound*, is also a magnitude of great importance.

As the solving process advances, the upper and lower bound converge. Accelerating this convergence is what leads to smaller search trees. Early on, researchers realized that branching on variables that produced a large LP-bound degradation in the children nodes helped to close the gap between upper and lower bound. This idea

lead to a fruitful line of research.

The first to propose LP-bound degradation as a mean to quantify branching variable quality were Benichou et al. [53]. They use a scoring method, which they named *pseudocosts*, to rank variables. These pseudocosts are defined in the following way. Consider a node with LP solution $\tilde{x}$ and objective value z. Let x_j be the candidate variable used for branching. Let z^+ and z^- be the objective values of the children's LP relaxations. Then, we formally define the LP-bound degradations to be $\Delta^{\pm} = z^{\pm} - z$ (for the up and down branches, respectively). The pseudocosts of variable j are defined as

$$P_j^- = \frac{\Delta^-}{f_j} \qquad P_j^+ = \frac{\Delta^+}{1 - f_j}$$

where $f_j = \tilde{x}_j - \lfloor \tilde{x}_j \rfloor$. These two pseudocosts are then combined together to yield a unique score upon which variables are ranked.

The pseudocosts of each variable are not known at the beginning of the search. They can be collected every time a variable is used for branching. Therefore, four questions must be answered in order to deploy a pseudocost-based branching rule:

1. How to combine the pseudocost information from several different nodes?
2. How to combine the up and down pseudocosts?
3. What if one (or both) of the children nodes is infeasible?
4. How to initialize the search?

Let us address the first question. The original authors rely on the experimental observation that, for a given variable, the pseudocosts collected at each node are of the same order of magnitude. Therefore, they propose to only use the first pseudocost that becomes available for each variable. Nowadays, a much more accepted approach [54, 55] is to average over all available data, obtaining estimates $\hat{P}_j^-$ and $\hat{P}_j^+$.

With respect to the second question, several functions have been proposed, although computational studies [56] favor a weighting mechanism of the form

$$score(j) = (1 - \mu) \min(\hat{P}_j^- f_j, \hat{P}_j^+ (1 - f_j)) + \mu \max(\hat{P}_j^- f_j, \hat{P}_j^+ (1 - f_j))$$

where $\mu \in [0, 1]$ is a parameter. In contrast, solvers like SCIP use instead

$$score(j) = \max(\hat{P}_j^- f_j, \epsilon) \max(\hat{P}_j^+ (1 - f_j), \epsilon)$$

with $\epsilon = 10^{-6}$ [57].

There is no standard answer to the third question. Several approaches have been proposed in order to account for node infeasibility information, see, e.g., [50] and

[58].

The question remains on how to initialize the search when no pseudocosts are available, as they can only be obtained in hindsight, after making a branching decision. The authors that propose this method give the unsatisfactory answer of using information acquired from previous runs. The choice of initialization gives rise to several branching rules. We will review three of the most important ones.

Strong branching
One way to obtain information about the pseudocosts of the branching candidates is to explicitly calculate them by tentatively branching on each candidate. In this way, initialization is solved and, furthermore, there is no need to estimate the pseudocosts by averaging: they can be unambiguously determined. Consequently, the scoring function is of the form

$$score(j) = (1-\mu)\min(\Delta_j^-, \Delta_j^+) + \mu \max(\Delta_j^-, \Delta_j^+)$$

Strong branching has been proven to be the best known branching rule in terms of node creation efficiency [11]. Indeed, strong branching is able to find an optimality certificate in the least number of nodes, compared to other rules. However, it incurs in an excessive computational cost, which makes it completely impractical.

One alternative that reduces the enormous computational cost of strong branching is to only consider a reduced set of candidates. This can be done in several ways. The developers of SCIP report several approaches [55] [50], like prioritizing candidates with high pseudocost estimates and stopping the exploration after a fixed amount of rounds without improvement, or limiting the number of LP iterations on each tentative branching. This effectively speeds up the search with respect to the strategy of considering all candidates (which is usually referred to as *full strong branching*). However, the computational overhead is still a burden that prevents strong branching from becoming the standard rule in practice.

Pseudocost branching with strong branching initialization
The branching decisions made at the top of the tree are the ones that have the largest impact in the final tree size. For this reason, allocating more computational resources to the first few branching steps seems justified.

As was already hinted in [53], one can explicitly compute the pseudocosts for all uninitialized variables. This idea was suggested in [53] and later explored with computational experiments in [59] and [56]. A variation of this scheme is to use strong branching up to a certain node depth and adopt pseudocost estimates thereafter [50].

Reliability branching
Reliability branching [55] builds upon the idea of strong branching initialization. Instead of restricting the explicit pseudocost computation to the first time a vari-

able is considered, or to a certain node depth, it is used until a variable's estimate is deemed reliable. The concept of reliability is formalized by counting the number of times a variable's pseudocost was obtained and setting a threshold for switching to pseudocost estimates. Currently, most solvers rely on slightly more sophisticated versions of this branching rule [11].

3.4. Modern branching strategies

In the previous section we examined classical branching rules, which are implemented and available in most state-of-the-art solvers. This chapter does not aim at presenting an exhaustive overview of all proposed branching strategies, which would be out of the scope of this thesis. Instead, in this section, we review some less standard strategies that are relevant for the context of the work at hand. Henceforth, we present a series of work that served as a precursor to ML-based branching rules, as well as the first attempts at learning to branch.

3.4.1. Predecessors

The ability to assess the impact of a decision one step ahead is arguably what makes strong branching such a powerful branching rule. Indeed, in [60] authors show that exploiting information from two steps ahead helps to improve the node creation efficiency (with the significant associated computational overhead). Instead of using online look-aheads or exploiting data from past branchings, one could run an exploration phase to gather information upfront, rather than during the search. Karzan et al. [61], followed by Fischetti and Monaci [62], were the first to propose such a scheme, where the search is restarted after running a shallow tree exploration.

In contrast to strong branching, these two rules do not use information about LP-bound degradation. In [61], variables are ranked based on their estimated effectiveness in producing fathomed nodes. This estimate is obtained based on fathoming information from the collection phase. On the other hand, in [62], authors build a prioritized branching variable list by trying to find a backdoor, i.e., a minimum cardinality set of variables whose integrality requirements are necessary to find the optimal solution.

Another key contribution to the line of research that lead to ML-based branching rules, was the work of Di Liberto et al. [63]. This is an extension of their previous paper [64], in which they propose a dynamic mechanism for branching. They were inspired by the work on portfolio algorithms, which, given the absence of a rule that performs well for any instance, try to adaptively assign a favorable rule to each problem. The authors take this idea one step further, by choosing a branching rule that is tailored to each sub-problem encountered down the search tree. This is done by defining a feature representation of each sub-problem, which is used to cluster similar samples together. Branching rules are assigned to each of the clusters by a

genetic algorithm.

There are three main takeaways from these precursors. First and foremost, the LP-bound degradation is not the only good indicator for branching strategies. Extracting more diverse data can be well-justified. Second, one can benefit from collecting this data upfront, instead of only exploiting online or historical search information. Finally, the work in [63] shows that it is crucial to consider the evolution of the problem as we subdivide the search space. Adaptiveness is therefore a desirable feature for a branching rule.

3.4.2. Early attempts at learning to branch

The idea of using supervised learning techniques to find efficient branching rules is not new. The first to propose this was Marcos Alvarez et al. [65], who extended their work in [66]. This approach is also explored in [67] and [68].

All of the above proposed methods share a common concept: much like in reliability branching, the goal is to find a fast approximation of strong branching. In order to do this, they create a representation of the problem (a feature space) and try to learn a scoring function based on empirical experience of strong branching score observations. This adds much more flexibility to the method, given that one can consider many more problem features and not be limited to past branching statistics, like in reliability branching.

The feature spaces defined in the aforementioned work have some common design traits. In particular, [66] and [67] make use of the same features. Authors distinguish between static and dynamic features. While static features describe the original problem (focusing mostly on the interaction between each variable and the constraints), dynamic features are node dependent and portray the current state of the solving process. All approaches make use of historical data related to pseudocosts, but usually in the form of more diverse statistical measures. As Marcos Alvarez et al. point out in [66], feature selection is a crutial step. Features must be cheap to compute and independent of the problem's size and scale. The matter of feature selection will be addressed in more detail in Section .

While the aforementioned work has a common goal and high-level approach, these methods differ in various aspects. In particular, we can distinguish two distinct research directions.

On the one hand, in [66], Marcos Alvarez et al. continue on the line of Di Liberto et al. [63] by running an up-front data collection phase once and for all. Specifically, they solve a set of randomly generated instances and record feature-score pairs for each candidate variable, which are later used for training. The scoring is based on strong branching. Instances are chosen to be small to ensure that the sample collection phase reaches the deeper nodes within the time limit that they impose.

On the other hand, [67] and [68] take an approach that is closer to reliability branching. Instead of using a unique data generation phase, they train theirs models online. Notice that [67] is by the same authors as [66]. In this second paper, Marcos Alvarez et al. reuse the framework of their previous work, but learn an instance-specific linear regression instead. In particular, they use the concept of reliability from [57]: at each branching step, feature representations are extracted for each candidate variable; if the candidate is unreliable, strong branching is used to calculate its score, while also saving the feature-score pair for training; if the candidate is reliable, the trained linear regression is used instead. In [68], Khalil et al. take a very similar approach. Their main contribution is that they frame the problem as a learn-to-rank problem, rather than a regression. They argue that no computational resources should be consumed in learning to predict the actual strong branching score, when the real goal is to learn a ranking of the variables. This contrasts with the proposal of Marcos Alvarez et al. in both [66] and [67].

Overall, in the case of learning to branch, the online versus offline learning dichotomy can be summarized in an adaptiveness trade-off. It is unrealistic to think that we can find one rule that works well on any instance. On this matter, online learning offers a policy that is tailored to each individual problem. However, one can argue that training on samples extracted on a shallow tree exploration lacks adaptiveness to the tree evolution. In contrast, offline learning algorithms can exploit the initial data collection phase to gather a more diverse set of observations, at different solving stages. Marcos Alvarez et al. offer an extension to their work on online learning [67] to tackle this particular issue. They propose to continuously train the machine learning model as variable pseudocosts naturally become available during the search, i.e., not through look-aheads but by observing the aftereffects of branching.

Comparing the performance of the aforementioned methods is challenging due to the diverse benchmarking methodologies chosen by the authors. In general, these resulting branching rules are proven successful at providing a fast approximator of strong branching. This is, they achieve a considerable speed-up with respect to strong branching, coupled to a small degradation in node creation efficiency. However, the overall performance is below that of classical rules such as reliability branching. Khalil et al. [68] show the most competitive results, with a superior node creation efficiency. In spite of this, CPLEX's default branching rule (which is probably a version of reliability branching) wins in terms of total solving time. The authors note that their algorithm could benefit from a better integration with the solver, in order to make execution time based comparisons more fair.

3.4.3. State-of-the-art ML-based branching rules

Lastly, we shall discuss two very recent publications on the topic of variable selection through computational intelligence.[1]

In [13], Gasse et al. propose the first deep learning approach to variable selection. Their model leverages the bipartite graph representation of MILPs (see Chapter 4 for a more detailed description). Instead of relying on heavily hand-crafted features, the authors encode the complex variable interdependence introduced by the constraints by using a Graph Convolutional Neural Network (GCNN) [69]. This has the advantage of automatically propagating variable information based on the problem's graph structure. With this representation, they mimic strong branching decisions through an imitation learning algorithm, known as behavioral cloning [23]. This model is trained and tested on four sets of randomly generated instances, each one representing a different problem class. Authors show that this approach greatly outperforms all previously proposed ML-based methods, and furthermore compares favourably with classic branching strategies. They extend these results to tests sets containing larger sized instances, evidencing that their method is capable of generalizing outside of its training set (in terms of size, but not instance class).

Finally, let us mention the work of Zarpellon et al. [14]. They present the first framework for learning a branching rule through RL. In spite of not achieving any performance gains, their main contribution is the design of a carefully crafted set of features that describe the branch-and-bound tree. These features will be discussed in more detail in Section 4.3.1. Furthermore they propose a set of possible reward functions for the RL algorithm.

3.5. Benchmarking MIP solvers

When conducting a computational analysis of the performance of different MIP solvers, the first step is to define a benchmarking methodology. This task is particularly challenging as there are many pitfalls to avoid. In this section, we address the issue of defining a fair benchmarking protocol to compare different solvers. For this, we first present methods to study and handle performance variability. Then, we provide a set of guidelines to select a benchmarking dataset. Finally, we describe a series of performance measures or tests that yield a comprehensive comparison among solvers.

3.5.1. Performance variability

When evaluating a particular solver on a particular instance, *performance variability* is the name we give to the observed differences in performance which are due to changes in the environment that a priori may seem performance-neutral. The

[1] Both these papers were made available to the author on July 14th 2019, after several months of independent work. The methodology described in Chapter 4 was developed independently of this work unless otherwise stated.

branch-and-bound algorithm is particularly prone to performance variability. Indeed, one small alteration can trigger a cascade of events that lead to a deviation in the original tree structure, causing the solution process to change dramatically. These seemingly performance-neutral changes include the order in which variables and constraints are presented, the randomization seed or the platform on which the solver is run.

One of the main causes of performance variability in known to be imperfect tie-breaking for heuristic decisions [70]. When the solver does not have a robust tie-breaking strategy, decisions may be influenced by arbitrary factors such as the order of the candidates or accumulated numerical error. Rounding errors are platform-dependent and are subject to the order in which arithmetic operations are made. Another cause for performance variability can be the existence of more than one optimal LP basis at the root node. The lack of uniqueness affects cut generation and some primal heuristics.

As a consequence, performance variability can affect solvers and instances to a very different extent. This rises the question of how to study this phenomenon in a controlled way [71] and how to build more robust strategies capable of tackling it (or even benefiting from it [72]). We distance ourselves from this line of work and try to answer the following question instead:

> "*[When comparing the performance of several solvers] how likely is it that the observed performance difference is created by variability rather than algorithmic change?.*"[70]

The approach proposed in [70] is to artificially generate variability to later quantify it, as a measure of the robustness of the solver, as well as a sort of confidence measure of the performance indicators. In [71], three variability generators are considered:

- Random row and column permutations.
- Initialization of the random number generator (seed).
- Degenerate pivots in the root node.

Once variability is generated, the extent of its effects must be measured. In [70] this is done through the *variability score*. Let ρ be a performance measure (e.g., solver time, number of simplex iterations or number of visited nodes). Let $\{\rho\}_{i=1}^{n}$ be the different performance measures obtained as a result of variation. Then, the variability score is defined as

$$\nu := \frac{\sqrt{n \sum_{i=1}^{n} (\rho_i - \bar{\rho})^2}}{\bar{\rho}}$$

where $\bar{\rho} = \sum_{i=1}^{n} \rho_i$ is the sample average. In other words, the variability score is the ratio between the sample variance and the sample average. This normalization allows us to make comparisons independently of the difficulty of the model at hand.

3.5.2. Benchmarking set

Several aspects must be taken into account when selecting the set of instances for benchmarking. First of all, the size of the set must be chosen in accordance with the magnitude of the differences that one wishes to characterize. The smaller the variation, the larger the dataset must be. This is also particularly important in order to control the impact of performance variability.

Second of all, one must rigorously report the composition of the set and, in particular, the selection criteria. One cannot expect the results to generalize to different instance types. In an effort to avoid biased performance assessments, the MIP community has developed a standard benchmarking set comprising real-world problems from academia and industry [73]. This dataset was carefully selected and is periodically updated to be representative of the variety of problems that a MIP solver encounters. For certain applications, one may restrict the test set to a particular class of instances. However, in this case, one holds the responsibility to inform the reader of the specific limitations of such approach.

It is common practice to subdivide the test set into different categories that account for the level of difficulty of the instances therein. As pointed out in [11], this can easily lead to a biased evaluation. The potential pitfall is to base the classification criterion on data originating from one solver exclusively. Generating the subsets in this way yields results that can easily be misinterpreted. We refer to [11] for an illustrative example of this phenomenon. Instead, one must choose a criterion that either considers all solvers equally or one that exclusively exploits data that is inherent to the instances.

3.5.3. Benchmarking methodology

There is no standard procedure to evaluate MIP solvers. In this section, however, we will discuss common practices for solver comparison. Specifically, we will focus on the case of benchmarking for branching rule analysis. Typically, after the benchmarking set is chosen, instances are solved while performance metrics are recorded. We can distinguish two types of tests:

- **Tests with a node limit:** when running experiments with a node limit, we can analyze the trade-off between the two desired properties of a branching rule: node creation efficiency and node processing efficiency. In particular, reporting the closed gap provides a metric for the quality of the branching decisions, while reporting the elapsed time serves as a measure of the speed at which decisions are made. This test is particularly interesting when instances cannot be solved to completion due to computational constraints.
- **Tests with a time limit:** experiments with a time limit test solvers on a

more practical setting. Important metrics to report are the (mean) closed gap and the number of instances solved by each solver.

Notice that reporting the total number of visited nodes when imposing a time limit can be hard to interpret in a setting where neither all not none of the instances are solved. Many authors introduce an ad hoc penalty system that can uncontrollably introduce biases. However, if the time limit is large enough so that all instances are solved by all solvers, we can analyze the same trade-off as with a node limit. In this case, we need to report the total node count and the elapsed time.

Once the data is generated, there are different options to present it, mainly:

- **Tabular display:** the chosen performance metric is aggregated over instances, usually through the geometric mean, in order to limit the effect of large numbers [11].
- **Performance profiles:** Dolan and Moré [74] propose a visual representation of each solver's performance, as opposed to large numeric tables. Their method, known as performance profiles, provides information about the ratio of a solver's performance with respect to the best observed performance, calculated by instance. These profiles can be presented for different performance metrics. In spite of their popularity as a data analysis tool, performance profiles can be misleading when comparing more than two solvers. An example of this can be found in [75], where authors show that they can only draw correct conclusions about the best performing solver, whereas the remaining ones cannot be properly compared.

4

Methodology

In this chapter we will describe all the methodological choices for the proposed approach. We start by first motivating the fundamental design choices (Section 4.1). This leads us to the problem formulation detailed in Section 4.2. Section 4.3 characterizes the model we adopt: the feature selection, architecture and reward mechanism. Finally, in Section 4.4 we discuss the characteristics and selection criteria for the benchmarking set that will later be used to evaluate the proposed method.

4.1. Motivation

Before proceeding to describe the methodological approach, we must first justify the rationale behind it. This section establishes the motivation for choosing reinforcement learning as a tool to design and discover new branching strategies. The question of "*why reinforcement learning?*" will be answered by breaking it down into two subquestions, namely "*why learning?*" and "*why reinforcement?*".

Why learning?
In the literature study in Chapter 3 we already established a tentative answer to this question. Given the lack of fundamental understanding of the processes that govern and affect branching, state-of-the-art solvers rely on hand-crafted heuristics. In Chapter 3 we reviewed studies that lead to believe that (i) exploiting a diverse set of data describing the problem can be beneficial, in contrast to relying only on (estimates of) LP-bound degradation, and (ii) we could shift part of the computational burden from the actual tree search into a preliminary (or "training") phase.

To emphasize the first point, Figures 4.1 and 4.2 show the results of an experiment on the strong branching scores (LP-bound degradation). The objective is to test the assumption on which most pseudocost-based rules rely: that past values of the LP-bound degradation observed on each variable are a good basis to estimate future

values of said magnitude. For this, instance `gen-ip054` (from MIPLIB2017 [73]) was solved using two different node selection rules (depth and breadth first search), random variable selection for branching and a custom branch-and-bound solver that provided a perfect test environment, without any other interfering processes. At each branching step, strong branching scores were calculated and stored (notice that this information was not used for branching purposes). Figure 4.1 shows the progression of these scores, per variable, for the two different node selection rules. These results indicate that, in general, the assumption is not even partially true. If it were, we would expect to see horizontal lines. Figure 4.1(b) shows some correlation between consecutive SB-scores for certain variables. Figure 4.2 shows the same experiment but with vertical lines that indicate the end of a diving, i.e., when the exploration changes from a deep node into a shallow one. This reveals that said correlations are due to the diving nature of depth first search. In other words, they are "in-branch" correlations. Indeed, the breadth first search experiment shows no such correlations. In practice, node selection rules tend to alternate these two behaviors, so we cannot rely on in-branch correlations.

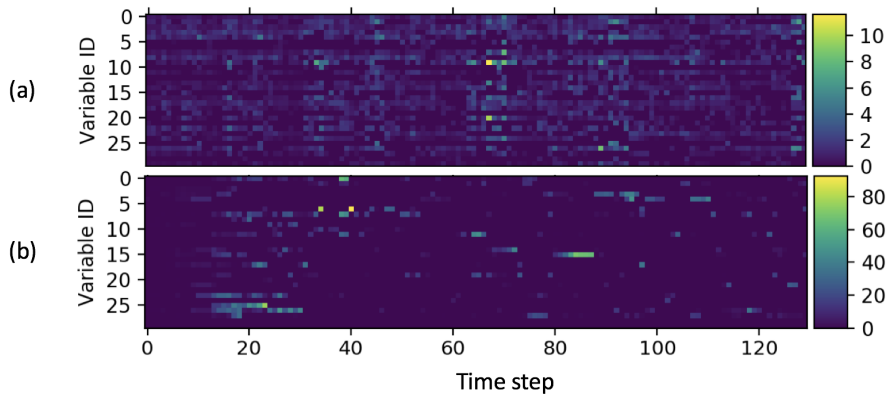


Figure 4.1: Evolution of the strong branching scores per variable. The node selection rule was set to (a) breadth first search and (b) depth first search.

We conclude that estimating pseudocosts on the basis of past observations is not well justified. These experiments reinforce the argument that it is worthwhile to diversify the type of data on which branching rules base their decision-making. However, this poses another challenge: how should the data be combined in order to effectively discern good branching decisions? The premise of this thesis is to abstract this process away, by learning to branch through machine learning, instead of relying on manually-engineered expert knowledge.

Generally, machine learning algorithms are able to exploit the structure of the problem to gain insights on how to tackle them. This becomes both a benefit and an

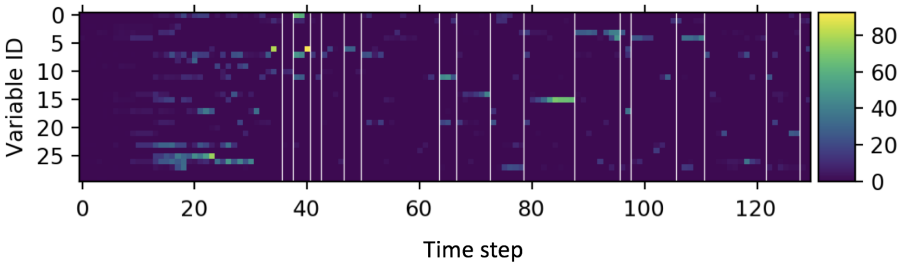


Figure 4.2: Evolution of the strong branching scores per variable using the depth first search rule. Vertical lines indicate the end of a diving, i.e., an abrupt change in the depth of the node under processing.

obstacle when faced with the problem of learning to branch. On the one hand, we will see that the algorithms are able to leverage the common structure of MILPs with similar constraint types. On the other hand, these methods tend to fail to generalize when faced with a diverse set of MILP instances, with very different combinatorial structures. In other words, machine learning is, as of today, limited to the resolution of intrinsically similar instances. This can still find many applications in practice (e.g. companies that solve the same type of bussiness-based model), and is an initial stepping stone towards more general strategies.

Why reinforcement?
Unlike other machine learning tasks, the problem at hand presents the challenge of learning without having information about the true label, which, in this case would represent the best possible action. Generating such labels is infeasible even for small-sized instances, since it would require exploring all possible trajectories. Nonetheless, as we discussed in Chapter 2, some authors do consider supervised learning methods (e.g. imitation learning) with strong branching as an expert. This approach is flawed for two reasons. First, performance can be significantly affected when the apprentice encounters states that deviate from those encountered during training. This could happen if the training samples are not representative enough of the tasks that the agent will be tested on (e.g., when trying to generalize to larger instances), or simply because the error in the learnt policy leads the agent to observe a very different state distribution. Second, strong branching can be little understandable as an expert, given that it leverages information "from the future", such as child-node infeasibilities. When trying to train an apprentice that has restricted information, the expert's actions may lack explainability, resulting in a learnt policy that is far from the target.

In contrast, a reinforcement learning approach seems more suitable for the task of learning to branch. In spite of the lack of true labels, we can construct a notion of what our objective should be: produce trees of small size. In this context, reinforcement learning can formalize this abstract goal into something tangible through

a reward system.

4.2. MDP formulation of the branch-and-bound search

The variable selection problem can be modeled as a MDP. The branch-and-bound process represents an environment which consults an agent every time a branching decision in needed. Each encountered subproblem corresponds to an environment state. Under this setting, the process would have the Markov property, as each subproblem is a MILP of its own and can be treated independently. However, in a practical setting, the agent will not have access to a full representation of the subproblem. This converts the process into a partially observable MDP. To compensate for this lack of information, we can introduce supplementary data into the state representation (see Section 4.3.1).

In the reminder of this thesis, we will consider the following terminology, which unifies concepts from RL and MILP solvers:

- **The environment** is the branch-and-bound process. The state of the search is represented by a set of features that will be discussed in the next section.
- An interaction **time step** is a call to the brancher from the solver.
- The **action space** is the set of variables.
- The **agent** makes branching decisions only. All other actions, e.g., node selection, pruning and cut generation, are modelled as being part of the environment.
- An **episode** concludes when the optimal solution is found.

4.2.1. A note on solver settings

One of the primary difficulties of training and benchmarking branching rules is the irremediable interaction with other solver features, such as node selection rules or primal heuristics. These rules are clearly not independent and the consequences of this interdependence must be taken into account.

One may consider the option of turning off any heuristic rule that may interfere in the benchmarking process. Though this could be scientifically interesting, it is ultimately unpractical if one hopes to build algorithms that are functional in real situations.

For this reason, we build a model that operates under almost default solver settings. Only two modifications were made. In particular, cut generation and primal heuristics are only enabled in the root node. The reason for the first restriction is solely due to software development constraints and can be avoided with a more seamless

integration with the solver's source code. With regard to primal heuristics, preliminary experimentation showed that they could hinder the learning process. We argue that the reason is the decreased correlation between the reward function and the agent's actions when primal heuristics are enabled. Indeed, if an integer feasible solution is found due to the latter, the agent will perceive a higher reward without having actively participated in this discovery. In Chapter 5 we present experiments that show that this choice is not restrictive.

4.2.2. Node taxonomy

For the purpose of this thesis, we establish a node taxonomy in order to unify the terminology across different sources. In particular, we define:

- **Idle node:** node that was processed but not pruned. This means that it was found to be feasible but not integer feasible and, furthermore, its bound did not exceed the incumbent. As a consequence, a variable was chosen for branching and two children nodes were created. This node no longer participates in the search.
- **Pruned node:** node that was excluded and did not produce any children. The reasons for exclusion can be infeasibility, integer feasibility or a bound that exceeded the incumbent.
- **Open node:** node that was created but is yet to be processed.
- **The focus node:** node under consideration at the current time step.

4.3. Characterization of the model

In this section, we address the proposed model in more detail. Firstly, we review the feature selection process that lead to the definition of the environment's state-representation. These features are the input to the agent's model. Secondly, we describe the neural network architecture used for the agent, together with other design aspects. Finally, the last part of this section presents a study of the reward function design process, which concludes with the definition of three different possible reward mechanisms.

4.3.1. Feature selection

A key initial step is to define a representation for the environment's states. This topic has been addressed in a variety of ways in the literature. By first analyzing in detail the design choices in previous work, we can identify their strengths and shortcomings, in order to compose a balanced and informative state representation.

Let us first define a set of classification criteria for the data that can be extracted along the search tree. We can categorize features according to three aspects, namely:

1. Type: features can represent a general property of the solving process or can be linked to a particular variable or constraint. We therefore classify them in:

- **Process features**
- **Variable features**
- **Constraint features**

2. Scope:
   - **Local features** are extracted on a node level, i.e., are a characteristic of the focus node.
   - **Global features** have a wider scope, taking into consideration data from several nodes. In this way, historical data can be incorporated into the feature description.

3. Evolution: features can be extracted once and for all at the root node, or they can evolve together with the tree search. Hence we distinguish between:
   - **Static features**
   - **Dynamic features**

Figure 4.3 summarizes this classification method.

From the work discussed in Chapter 3, we will examine three different proposals for feature selection, namely the work by Khalil et al. [68], Gasse et al. [13] and Zarpellon et al. [14]. In [67] and [66], authors use a feature space that resembles that of [68], so we will only consider the latter.

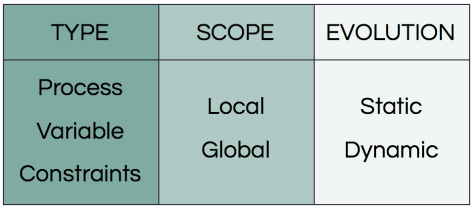


Figure 4.3: Feature classification diagram.

Tables 4.1, 4.2 and 4.3 summarize these proposals. The color coding is meant to ease the reading process, by grouping together sets of features that have a common characteristic. The criterion for grouping them is highlighted in yellow.

Lastly, before proceeding with the analysis, the reader should be warned that some of the tables are less detailed, because of the different levels of information conveyed in the papers under examination. However, this missing information does not affect the conclusions drawn in this section.

Table 4.1 shows the features selected by Khalil et al. [68]. Their feature design stresses the difference between static and dynamic features. The most remarkable characteristic is that only variable features are considered. They do not extract any information about the overall search process. Constraint features are taken into

account but in an aggregated way, i.e., only in relation to the variables that participate in them.

As mentioned in Chapter 3, Gasse et al. [13] proposed a novel way of propagating variable features through the interrelations imposed by the constraints. For this reason, they are able to directly use raw variable and constraint features, without the need to manually engineer an aggregation method (see Table 4.2). They stress the distinction between variable and constraint features. In contrast, the data is primarily local. This means that they reduced the scope of the information, to look almost exclusively at the focus node, therefore disregarding a lot of historical tree information.

In [14], Zarpellon et al. do not describe features in great detail. All the available information is shown in Table 4.3. They put the focus on the tree evolution by using exclusively (to the best of our knowledge) dynamic features. In this work, three groups of features are considered, each of which can be described using the classification method previously defined. In particular, they consider global process features , local process features and variable (both local and global) features.

We conclude this analysis by comparing these approaches, with the goal of applying the learnt lessons towards a more complete set of features. The excellent performance of the model in [13] shows that not only is it possible, but also beneficial to let the learning mechanism discover effective ways to combine the data, in a manner that is representative of the complex interactions that it describes. This idea is also in line with the premise of this thesis. However, the study presented in this section seems to indicate the set of features used in [13] could be too limited. For this reason, we propose to extend them with local and global process features, in a manner that resembles the work of [14]. This provides the agent with a more general overview of the state of the search.

Name	Description	T	S	E
coef	Objective function coefficient (raw, positive only, negative only).	V	Local	S
num_const	Number of constraints a variable participates in.	V	Local	S
const_degree_stats	Statistics of the degree of all constraints a variable participates in.	V	Local	S
const_coef_stats	Statistics of the coefficients in the constraints a variable participates in (pos./neg.).	V	Local	S
up_frac	Up fractionality $\lceil \hat{x}_j \rceil - \hat{x}_j$	V	Local	D
slack	$\min(\lceil \hat{x}_j \rceil - \hat{x}_j, \hat{x}_j - \lfloor \hat{x}_j \rfloor)$	V	Local	D
pcost	Pseudocosts (up and down) and its ratio, sum and product.	V	Global	D
infeas	Number and fraction of nodes in which a variable lead to an infeasible children.	V	Global	D
const_degree_local	Dynamic version of the degree statistics, plus ratios to the static counterpart.	V	Global	D
coef_to_rhs	Min/Max over the ratios between a variable's coefficient and the RHS (pos./neg.).	V	Local	D
coef_ratios	Min/Max over ratios between a variable's coefficient to the sum over all other variables' coefficients (pos./neg.).	V	Local	D
active_const_coef	Statistics over the active constraints a variable participates in.	V	Local	D

Table 4.1: Feature selection used in [68]. The authors stress their choice of including static and dynamic features. All the extracted data is variable dependent.

Name	Description	T	S	E
obj_cos_sim	Cosine similarity with the objective function.	C	Local	S
bias	RHS, normalized by constraint coefficients.	C	Local	S
is_tight	Tightness indicator in LP solution.	C	Local	D
dual	Dual solution value, normalized.	C	Local	D
cons_age	LP age, normalized by number of LPs.	C	Local	D
cons_coef	Constraint coefficient, normalized	C/V	Local	S
var_type	Type (B/I/II/C) as one-hot vector.	V	Local	S
coef	Objective coefficient, normalized.	V	Local	S
has_lb	Lower bound indicator.	V	Local	D
has_up	Upper bound indicator.	V	Local	D
sol_is_at_lb	Solution value equals lower bound.	V	Local	D
sol_is_at_ub	Solution value equals upper bound.	V	Local	D
frac	Solution fractionality.	V	Local	D
basis_status	Simplex basis status as one-hot encoding.	V	Local	D
red_cost	Reduced cost, normalized.	V	Local	D
var_age	LP age, normalized.	V	Local	D
sol_val	Solution value.	V	Local	D
inc_val	Value at incumbent	V	Global	D
avg_val	Average value in optimal feasible solutions.	V	Global	D

Table 4.2: Feature selection used in [13]. Features are divided in the variable and constraint type. Constraint coefficients are also considered, which link variables and constraints to each other.

Name	Description	T	S	E
growth_rate	Tree growth rate.	P	Global	D
tree_comp	Composition of the tree (processed, open and leaf nodes).	P	Global	D
g_bounds	Evolution of the global bounds.	P	Global	D
feas_stats	Statistics of the feasible solutions.	P	Global	D
bound_stats	Statistics of the open node bounds.	P	Global	D
depth	Depth of the focus node.	P	Local	D
bound	Bound of the focus node.	P	Local	D
var_meas	Aggregated variables' measures in solution.	P	Local	D
var_bound	Variable bounds.	V	Local	D
sol_val	Solution value.	V	Local	D
score	Score.	V	Global	D
branch_stats	Statistics of past branchings.	V	Global	D
other	Participation in other search components.	V	Global	D

Table 4.3: Feature selection used in [14]. Only dynamic features are considered. The selection is well balanced between global and local data.

4.3.2. The agent

The agent plays a pivotal role in a MDP. Let us address the two elements that conform it. On the one hand, we have an actor, which is a function approximator (a neural network in this case) that computes the agent's policy. On the other hand, as discussed in Chapter 2 some reinforcement learning algorithms make use of an auxiliary component: the critic. Its role is to guide the optimization process, in search of the optimal policy. In our model, the state representation of the branch-and-bound process is split into two, one for the critic and one for the actor. We shall refer to them as S_t^a and S_t^c, respectively. In this section, the actor and critic models will be discussed in more detail.

Actor model

We adopt the actor model from [13]. The reason for this choice will become apparent in Chapter 5. The most remarkable property of this architecture is the way that variable and constraint information are combined. Undeniably, the complex variable interdependence, which is introduced by the constraints, must be taken into account for decision-making. Instead of manually engineering features that describe these intricate interactions, this model abstracts this process away with the use of graph convolutional neural networks.

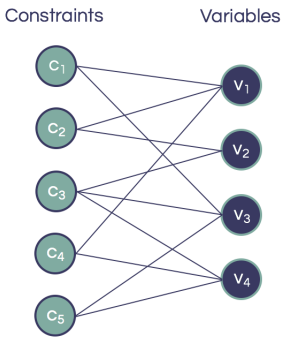


Figure 4.4: Bipartite graph representation of the MILP

The authors in [13] propose a neural network architecture that exploits the bipartite formulation of a given (MI)LP. This is only a different description of the same problem, in which variables and constraints are represented by nodes, forming two disjoint sets. Edges between a variable node and a constraint node represent the involvement of the variable in the constraint. Figure 4.4 shows an example of such graph with $m = 5$ constraints and $n = 4$ variables. In this example, the first constraint (equivalently, the first row of the constraint matrix) would have the form

$$a_{11}x_1 + a_{13}x_3 \leq b_1 \,.$$

The branching policy is a probability distribution over the action space. In order to calculate it, the network uses a feature vector for each of the variables. The process for obtaining these features is illustrated in Figure 4.5. In the first place, raw sets of features are extracted for constraints ($\mathrm{C} \in \mathbb{R}^{m \times d_c}$), variables ($\mathrm{V} \in \mathbb{R}^{n \times d_v}$) and edges ($\mathrm{E} \in \mathbb{R}^{e \times d_e}$). These tensors go through their independent embedding layers. Two graph convolution passes follow. These passes propagate information, firstly towards the constraint features, and secondly towards the variable features.

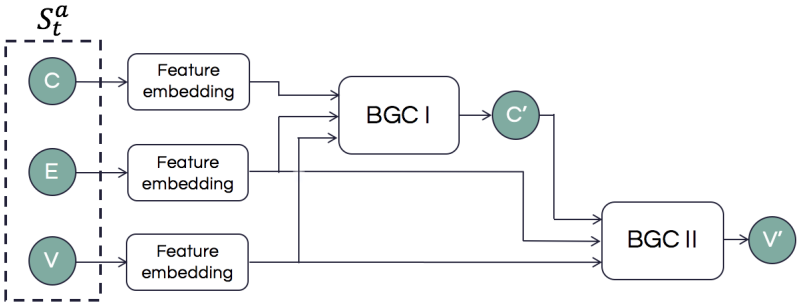


Figure 4.5: Schematic representation of the embedding process. The first step consists of fully connected embedding layers. Feature information is combined following two passes of a bipartite graph convolution (BGC). Finally, the output is a tensor describing the variables, which incorporates interaction data.

In particular, we learn layers $f^{(k)}$ and $g^{(k)}$ $(k \in \{\mathrm{I}, \mathrm{II}\})$ such that, for $i \in \{1, 2, ..., m\}$ and $j \in \{1, 2, ..., n\}$,

$$c_i' = f^{(\mathrm{I})}\left(c_i, \sum_{j:(ij)\in\mathcal{E}} g^{(\mathrm{I})}(c_i, v_j, e_{i,j})\right)$$

$$v_j' = f^{(\mathrm{II})}\left(v_j, \sum_{i:(ij)\in\mathcal{E}} g^{(\mathrm{II})}(c_i, v_j, e_{i,j})\right)$$

where we have used $\mathcal{E}$ to denote the edge set.

After this extended feature embedding process, the tensor V' is used as input to a simple two-layer feed-forward neural network. This converts the feature vector of each variable into a single scalar. After a softmax normalization step, these values can be used as a probability distribution over the action space. Non-candidate variables are masked out previous to this normalization.

Critic model

As discussed in Chapter 2, some RL algorithms make use of both an actor and a critic. The critic plays the role of guiding the learning process by influencing the updates to the actor's policy. It is therefore only used during the training phase. This is, once the agent is trained, the actor model is self-sufficient to produce a policy.

The critic must estimate the state-value function, defined as

$$V_\pi(S_t^c) = \mathbb{E}_{\tau\sim\pi}[G_t|S_t^c]$$

In the context of the branch-and-bound search, this can be interpreted as a prediction of the progress of the search, i.e., how far along the solving process is. For this

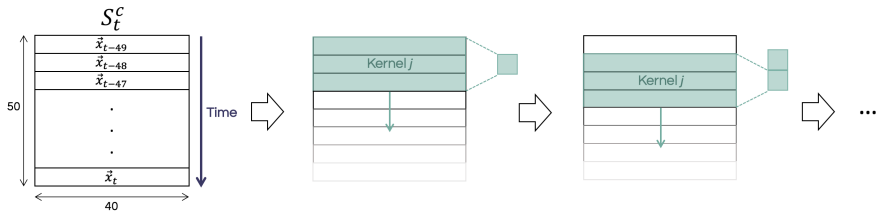


Figure 4.6: Critic input data going through a 1D convolution. Kernel characteristics such as size and stride are only illustrative and not representative of the actual parameters used.

reason, it seems optimal to use process features (see Section 4.3.1) as an input to the critic network.

In the reminder of this subsection we will describe the critic in more detail[1]. The input is of the form of a multivariate time series. Specifically, at each branching step a vector $\vec{x}_t$ of 40 features is recorded and the critic receives a 50-step time window of such data (see Appendix A for more details). In other words, the critic state is defined as

$$S_t^c = (\vec{x}_{t-49}, ..., \vec{x}_{t-1}, \vec{x}_t)$$

The raw input goes through 3 layers of 1D convolutions. This type of model is commonly used when dealing with time series. The process is illustrated in Figure 4.6. Finally, two fully connected layers transform the output of the last convolution into a scalar that estimates the state-value.

4.3.3. Reward function

Designing a good reward function is a key step in the process of deploying a reinforcement learning scheme. Devising a reward function for the task of learning to branch is particularly challenging. The reason is that, unlike many popular RL-testing environments, in the case of MILPs allowing complete roll-outs is prohibitive in terms of computational cost. Consequently, we must design a reward system that can provide the agent with useful feedback during the search.

A first step is to identify a list of properties that are desirable for a reward function. In particular, such a function should be:

- Bounded: we need to have control over the magnitude of the rewards. This is

[1] **Note from the author**: the critic architecture used for the experiments is due to the team at DS4DM [15]. The author of this thesis had developed independently another critic architecture. The feature analysis was conducted before the author gained knowledge of the approach used by DS4DM. Coincidentally, the independently drawn conclusions were very similar. However, the critic architecture designed by DS4DM was superior in performance and was therefore adopted when the collaboration began halfway through this thesis work.

important, for example, for choosing hyperparameters that are effective across different instances.

- Negative: negative rewards encourage the agent to terminate the episode as fast as possible. Positive rewards must always be treated with care, as the agent may find a strategy with which it drags the episode for longer, receiving smaller instant rewards but greater returns.
- Optimum-agnostic: the agent should be oblivious of the actual value of the optimal objective value (its magnitude, sign, etc). The only valuable information relies on how close the agent is expected to be from the optimum.
- Cheap: the computation of rewards should not add an excessive overhead.

Likewise, it is helpful to pinpoint which are the desirable behaviors that the reward should encourage. Taking inspiration from the literature [76, 77, 14], we strive to emphasize the following three:

1. **Encouraging gap closure**: The gap, in its various definitions, is a very common metric for the solving state of a MILP run. It accounts for the distance between the best known upper bound (incumbent) and the best known lower bound (best bound). As the solution process advances, the gap tends to zero. The faster the convergence, the quicker the process finishes.
2. **Encouraging node pruning**: Node pruning is the process through which the tree growth is controlled. A high pruning rate can only be achieved by finding good upper bounds (i.e. integer feasible solutions) and quickly discarding hopeless branches. Therefore the agent should attempt to prune a high percentage of the nodes it processes, minimizing the number of open nodes.
3. **Encouraging solution integrality**: Following [78] one can interpret the tree search as a quest to reduce the uncertainty on the value of each of the variables: at the top of the tree there is very high uncertainty of the variable's values, while at the leaves there is none. There is a clear link between encouraging leaf creation and encouraging pruning. Hence, we can design a system that rewards finding solutions that are (or are close to being) integer feasible.

Reward functions that encourage solution integrality proved to be very inefficient to implement, given the current software architecture of SCIP. For this reason, they are not discussed in this thesis.

Using these guidelines, we propose three reward functions. The first one is based on the gap, defined as

$$\Delta_t = |I_t - B_t|$$

where I_t and B_t represent the incumbent and the best bound at time-step t, respectively. Then, the reward is defined as

$$R_t^{gap} = \begin{cases} -1 & \text{if } \Delta_t = \Delta_{t-1} \\ \frac{\Delta_0 - \Delta_t}{\Delta_0} - 1 & \text{otherwise} \end{cases} .$$

Notice that $R_t^{gap} \in [-1, 0]$. In other words, this reward takes Δ_0 (the gap at the root node, before any actions are taken), and uses it as a measure of the difficulty of the problem given to the agent. This magnitude serves as a scaling factor to quantify the improvements that the agent achieves.

The second reward discourages the increase in number of open nodes. In particular,

$$R_t^{open} = -\frac{\#\text{ of open nodes currently}}{\max(\#\text{ of open nodes observed})}\,.$$

With this function, the agent is punished with $R_t = -1$ unless the number of open nodes decreases, which can only happen through pruning.

Finally, the last proposed reward considers the difference in number of idle nodes after taking an action. Specifically, taking $N(t)$ to be the number of idle nodes at time step t, we define

$$R_t^{idle} = N(t-1) - N(t)\,.$$

Though different, these three reward functions are correlated in practice. Their performance will be evaluated and compared in Chapter 5.

4.4. Instances

In this section we describe the instance set used for experimentation. In particular, we use a group of randomly generated combinatorial auction instances, which are comprised exclusively of set packing constraints (see the constraint classification defined for MIPLIB [79]). As was pointed out at the beginning of this chapter, learning a branching rule across a very diverse set of instances is very challenging. This is due to the nature of machine learning methods, which exploit problem structure. Learning within a particular instance class is an important first step towards a more general rule.

The instance set we consider is one of the collections used in Gasse et al. [13]. Only one of these collections is considered due to time limitations. The choice was motivated by the larger performance gap between the learnt policy and the expert in this particular class, which indicates a larger potential for improvement.

Let us formally define the type of instance we are regarding. Let M be a set of items (with $|M| = m$) and N a set of bids (with $|N| = n$) of the type $\{S_j, p_j\}$ with $S_j \subseteq M$ and $p_j \in \mathbb{R}$, for $j = 1, ..., n$. This is, p_j is the bid amount for bundle S_j. The variable $x_j \in \{0, 1\}$ represents whether S_j is sold or not. A combinatorial

auction problem is an instance of the type

$$\begin{aligned} \text{maximize} \quad & \sum_{j \in N} p_j x_j \\ \text{subject to} \quad & \sum_{j \in N : i \in S_j} x_j \leq 1, \quad \forall i \in M \\ & x_j \in \{0,1\}, \quad \forall j \in N\,. \end{aligned}$$

Instances were generated using the method described in Section 4.3 of [80]. We split them into the four categories shown in Table 4.4, according to size and purpose.

We must address one important issue. As mentioned earlier in this chapter, most solver settings are left on the default mode. This means that presolving will be performed for all instances. The presolving process can change the structure of the problem, in which case the assumption of a homogeneous set no longer holds. For this reason, we analyze the result of preprocessing instances. In particular, we found:

1. **Variables:**
   - All variables remain binary.
   - Presolving reduced the number of variables in all but one instance.
   - The median reduction was 12, with a maxmimum of 36 and a minimum of 0.
2. **Constraints:**
   - The number of constraints was reduced in 96% of the instances.
   - The median reduction was 3, with a maximum of 11 and a minimum of 0.
   - The change in constraint type proportions is displayed in Table 4.5.

With this we conclude that we can still rely on the hypothesis of a common instance structure even when applying a presolving step.

Group	Number	n	m
Train	10,000	500	100
Validation	30	500	100
Test (Easy)	50	500	100
Test (Hard)	50	1000	200

Table 4.4: Detailed composition of the different instance groups.

Constraint type	Before	After
Single variable	0.37%	0%
Variable bound	0.62%	0.2%
Set packing	98.9%	99.8%

Table 4.5: Variation in the constraint type proportion when applying SCIP's default presolving procedure.

5

Experimental results

This chapter presents the experimental results of the various considered approaches. In particular, the chapter is divided into three sections which correspond to the three phases of development of this project.

The first section of this chapter does not present any experimental results per se. Instead, it serves as a justification to the design choices of the training procedure in the posterior sections. In these subsequent sections (5.2 and 5.3), the presented work is posterior to the author gaining access to Gasse et al. [13] and Zarpellon et al. [14]. Given the success of the approach in [13] and the points exposed in 5.1, we shift our goal towards the extension and improvement of the results presented in that paper. For this reason, the policy obtained in [13] is taken as a baseline throughout the chapter. Towards the end of this chapter, both the proposed and the baseline policy are compared with classical branching rules.

The author would like to note that, while Sections 5.1 and 5.2 are the result of independent work, the experiments in Section 5.3 were developed as part of a collaboration of the author with DS4DM [15].

5.1. First experiments

The first models used to tackle the problem at hand were notably unsuccessful. They were developed before the publication of [13], which set a precedent for discovering branching rules using machine learning. In spite of this, it is interesting to briefly mention them to point out the main takeaways from their failure. Specifically, these first prototypes did not succeed primarily for the following reasons:

- **Pretraining:** initially the agent started from randomly initialized weights. This is common practice in the machine learning community, particularly for

supervised algorithms. However, in the case of reinforcement learning algorithms, the opposite is true (see [5] for an example and [4] for a notable counterexample). The vast action spaces that are usually encountered in practical applications make for a challenging framework for learning. For this reason, and after the publication of [13], actor weights were instead initialized to a policy learnt via imitation learning.

- **Policy optimization:** the task of selecting a policy optimization algorithm is not straightforward, given the vast literature in deep reinforcement learning. As an initial step, a simple REINFORCE (with baseline) model was selected. Together with the lack of pretraining, this simple algorithm was unable to explore the policy space effectively. As a consequence, more sophisticated methods were employed in the subsequent attempts.

- **Solver:** at first, a simple custom branch-and-bound solver was implemented and used as a bare experimental environment. The main advantage of this approach was the ability to isolate the effect of the branching rule. However, the limitations were numerous. In particular, the impact of other solver components could not be tested. Furthermore, the final policy could not be compared with a practical baseline. For these reasons, the model was adapted to work with SCIP [44], one of the state-of-the-art solvers. The motivation for choosing this particular solver was threefold: it is free for academic use, it is one of the few solvers that provide access to the source code and it has a reasonable Python interface [81].

During this stage of development, abundant knowledge about the reward mechanism and feature selection was acquired, which later was successfully put to use through more advanced models.

5.2. A value-based approach

After the publication of [13], all the previously gathered knowledge was applied to the extension of this approach using RL. The transition was straightforward, since the underlying framework, though independently developed, was almost identical.

In this section we will explore a value-based approach. In particular, the actor architecture is recycled, so that the same network parametrization can be used as a starting point, i.e., used as pretraining. We obtain an initial policy by using SCIP's internal full strong branching (FSB) as an expert. The core idea is the following: the imitation learning scheme learns a probability distribution over action space which assigns greater probabilities to desirable actions. In this way, the output is positively correlated with the "value" of each action, i.e., Q-values. For this reason, we propose to use a value-based approach, specifically a DQN [31], to improve over the initial policy.

5.2.1. Experimental setup

Recall from Chapter 2 that the DQN update is based on the following loss function

$$L(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)}[r + \gamma \max_{a'} Q(s', a'; \theta_i^T) - Q(s, a; \theta_i)]$$

where θ_i and θ_i^T are the policy and target network parameters at iteration i, respectively, D is a dataset of transitions (the replay memory) and γ is the discount factor. This update uses bootstrapping, i.e., we update the network's estimation with the target $r + \gamma \max_{a'} Q(s', a'; \theta_i^T)$, which depends on the network's estimation itself. This is a key component of the DQN training scheme, which allows us to only use transitions of the form $(S_t, A_t, R_{t+1}, S_{t+1})$ instead of full trajectories. However, it poses a problem under this particular setting: initially, the network is not pre-trained to estimate the Q-values and, even though we hypothesize that the output is directly proportional to them, the scale might be very different. We address this problem in two ways:

- **Two-phase training**: we split training into two. On the first run, we freeze all layers but the last one. The objective is to train only the actor layer, while leaving the feature embedding untouched. During this initial phase, the focus is to obtain accurate Q-value estimates, rather than making substantial changes to the policy.
- **Increasing discount factor**: the discount factor γ has the role of controlling "how greedy" the agent is. However, in this case, γ also governs the degree of bootstrapping. A completely greedy agent ($\gamma = 0$) uses no bootstrapping in the updates. In order to take advantage of this, we set $\gamma = 0$ at the beginning of the training session and we gradually increase it to reach $\gamma = 0.99$ by the end of phase 1.

Using these two tricks, training proceeds in the following way. The agent is trained on an episode basis: on each episode one instance is solved, from which experience is extracted and stored in the replay memory. At the end of every episode a parameter update takes place, for which experience mini-batches are sampled from the replay memory using a uniform distribution. In total, 1000 episodes are run with no instance repetition. The evaluation of the results is done by solving 30 new (unseen) instances with 5 different random seeds. Performance variability was implicitly induced through two methods: (i) column and row permutations, and (ii) varying the random number generator's initialization. See Section 3.5 for an overview of the performance variability generation and analysis methodology.

5.2.2. Results and discussion

After thorough hyperparameter tuning, results showed at most a 2% improvement over the initial policy on the easy test set. This improvement is not only small but also hard to reproduce with other instance types or with a scaled-up algorithm. The problem seems to lie in the combination of bootstrapping and the initialized policy.

When we randomly initialize the neural network parameters, it is common practice to take values close to zero. This prevents the output from being excessively biased towards a particular point in parameter space. In our case, even if the policy is initially good, the Q-value estimation is very inaccurate and furthermore hard to correct. The increasing discount can be an effective solution, but it is difficult to coordinate the pace of this evolution with other learning hyperparameters. The sensitivity of this method becomes more apparent when trying scale this up into a parallel architecture, which would be desirable to increase performance.

In summary, this approach can effectively be used to improve the initial policy. However, it is highly unstable, in the sense that it must be carefully tuned for each task. In an attempt to tackle this problem, a different setup was tested, in which the policy optimization was decoupled from the Q-value estimate improvement. This was done by first generating experience with the fixed initial policy and then using this experience as training samples for an additional linear transform, which was appended at the end of the baseline architecture (see Figure 4.5). Notice that in this way the policy is fixed during the initial training phase. This slightly different approach showed the same instabilities as the previous one.

5.3. An actor-critic approach

There does not seem to be an easy way to learn Q-values, but we could learn state values instead. By adopting an actor-critic architecture, the critic can be pretrained separately, starting from small, randomly initialized weights. In this section, we address this approach.

Here, the model characterization presented in Section 4.3 finally applies. We build an agent that consists of an actor and a critic. Each of these two components obtains an independent state representation, namely S_t^a and S_t^c, respectively. In the reminder of this chapter, we experiment with this configuration.

5.3.1. Experimental setup

The agent was trained under various settings, in order to determine the best training conditions. The complete training pipeline consists of three steps:

1. **Actor pretraining**: the actor is warm-started to the best performing set of parameters achieved through imitation learning. In particular, two different initial policies were tested, which differ in the type of expert used.

2. **Critic pretraining**: we generate samples of the type (S_t^c, G_t) by running an agent with the initial set of actor parameters. The agent only acts according to this initial policy and stores return information, without making any policy updates.

3. **Training**: Finally, the policy is optimzied using PPO updates (see Section

2.5.3 for an overview of this method). For this, 100 train instances are used over 20 epochs with 8 parallel agents that generate experience independently. During each epoch, two validation runs are executed on 30 validation instances. Algorithm 5 describes the process on a high level. Hyperparameters such as the number of epochs or the number of agents were chosen on the basis of preliminary experimental results and computational resources.

Algorithm 5 PPO training

Input: 100 train instances and 30 validation instances

Create parallel agents that share a common actor and critic.
for epoch= $1, 2, ..., 20$ **do**
    Shuffle order of training instances
    for j=0,...,99 **do**
        Take instance number j.
        Send instance to all agents.
        while not all agents are finished **do**
            Let agents take 32 steps.
            Save this trajectory information in the form of transitions $(S_t, A_t, R_{t+1}, S_{t+1})$
            Perform global update of the agent's parameters using PPO.
        if j mod 50 == 0 **then**
            Run validation.
    end for
end for

Parallelization

The parallelization of the training process helps not only by optimizing the usage of computational resources (hence decreasing the training time) but can also stabilize learning by decorrelating the data [3]. All the experiments in this section were run on a parallelized architecture due to DS4DM [15]. In particular, experience is generated by several agents running in parallel with different seeds and with implicitly induced performance variability on the instances.

The initial policies

As was previously mentioned, the actor is pretrained using the imitation learning scheme in [13]. Two different initial policies were obtained by running this training scheme with different experts. In particular, the experts were two versions of full strong branching: SCIP's internal full strong branching (sFSB) and a vanilla version of full strong branching (vFSB) coded by DS4DM [15] for the purpose of their work in [13]. The key difference is that vFSB gathers information by tentatively branching on all candidates and uses it exclusively for calculating a branching score. Contrarily, sFSB exploits this information for various purposes, including tightening the focus node's LP relaxation or updating solver parameters, among others.

Both initial policies were tested to evaluate the ability of reinforcement learning to achieve improvement under both settings.

Critic normalization

The critic plays the role of evaluating the progress of the actor. Specifically, it must make a prediction on the return that the agent will receive by following the current policy, given the current state. This poses a problem: while the rewards are bounded, the return is not. The number of steps until termination, T, can vary significantly from one instance to the other, making the return take values on completely different scales, even in the discounted case ($\gamma < 1$). This decreases the precision of the predictions.

In order to tackle this problem and taking inspiration on the work in [82], we implement[1] an instance-dependent linear transformation. The dependency on the instance is not a problem, since the critic is only used for training the policy and is discarded at the time of deployment.

Let us describe the normalization process in more detail. During pretraining, the critic receives input-target pairs $\{(X_j, Y_j)\}_{j=1}^{n}$. In particular, $X_j := S^c_{t_j}$ and $Y_j = G_{t_j}$, where t_j is the time of sample j. These sample pairs are extracted during a data generation phase in which an agent, using the initial policy, solves the training instances and saves samples with a probability $p = 0.05$. For the purpose of normalization, we will label targets with the instance whose solution process produced them. This is, we will talk about Y^i_j, if sample j was generated while solving instance i. The normalization is a linear transform of the form

$$Y^i_j \leftarrow \frac{Y^i_j - \mu^i}{\sigma^i}.$$

Several options were tested for defining μ^i and σ^i. The best performing normalization was

$$\mu_i = 0 \quad \text{and} \quad \sigma_i = \hat{N}(T_i)$$

for the undiscounted case ($\gamma = 1$) and

$$\mu_i = 0 \quad \text{and} \quad \sigma_i = \sum_{t=1}^{\hat{N}(T_i)} \gamma^{t-1}$$

for $\gamma < 1$. Here we define $\hat{N}(T_i)$ to be the estimated final node count of instance i (following the initial policy). This estimate is built by solving each instance several times during the sampling process.

There is an intuition behind this choice. The number $\hat{N}(T_i)$ estimates both the difficulty of the instance and the number of steps that will be needed to solve it. The

[1] The idea of a normalized critic is due to DS4DM [15] but the final form of this normalization is due to the author.

number of steps is equivalent to the number of actions taken by the agent and, most importantly, to the number of rewards the agent will receive. All defined rewards tend to penalize the agent with $R_t = -1$ at most steps. Indeed, rewards closer to zero are only given if significant progress is made as a consequence of the last action. In the undiscounted case, this translates in a maximum (in absolute value) possible return of $-\sum_{t=1}^{\hat{N}(T_i)} 1 = -\hat{N}(T_i)$. Using this quantity as a normalization transforms the task of predicting the return left into a sort of "percentage of return left" metric, which is a much easier endeavor. In the discounted case, we must only correct for the effect of the discount factor, but the intuitive explanation remains the same.

Using this normalization, the critic's outputs lie in the interval $[-1, 0]$. During actor-critic training time, these predictions are transformed back into state-values using the same instance-dependent normalizing factors.

It is interesting to note that computational experiments showed that updating the normalization factors as the policy changes did not yield better results, even causing a degraded performance in some cases. Therefore, in the experiments that follow, the instance-specific normalization was fixed for good during the critic pretraining stage.

Validation
During training the policy is evaluated at regular intervals. This evaluation is done on a different set of instances (validation instances) and with agents that act greedily. The experiments shown in the coming sections are the result of combining validation data from solving 30 instances with 5 different seeds. This amounts to a total of 150 samples.

Two metrics are used in order to determine the quality of the policy during training: the total node count and the total number of LP iterations. In Section 3.5 we discussed that the most common evaluation metrics are the number of nodes and the solving time. However, in a highly parallel architecture such as the one at hand, obtaining a reliable measurement of elapsed time is not trivial. Instead, the total number of LP iterations is easily available and highly correlated with total solution time, given that all of the branching rules to be compared in these experiments are based on the same neural network architecture, hence taking approximately the same time per decision.

5.3.2. Environment settings

As a first step, we experiment with different environmental settings in order to determine the best configuration for learning. In particular, we answer the following questions:

- Which discount factor γ to choose?
- Out of the proposed rewards, which one yields the best performance?

- Which are the differences between the two initial policies?

Figure 5.1 shows a series of experiments to determine the best discount factor configuration. In particular, we train agents using the methodology described in the previous section with all types of reward and two discount factors: $\gamma = 1$ (no discounting) and $\gamma = 0.99$ (discounting). The initial policy was obtained with sFSB as an expert. Results indicate that greedier agents perform worse for all types of reward mechanism.

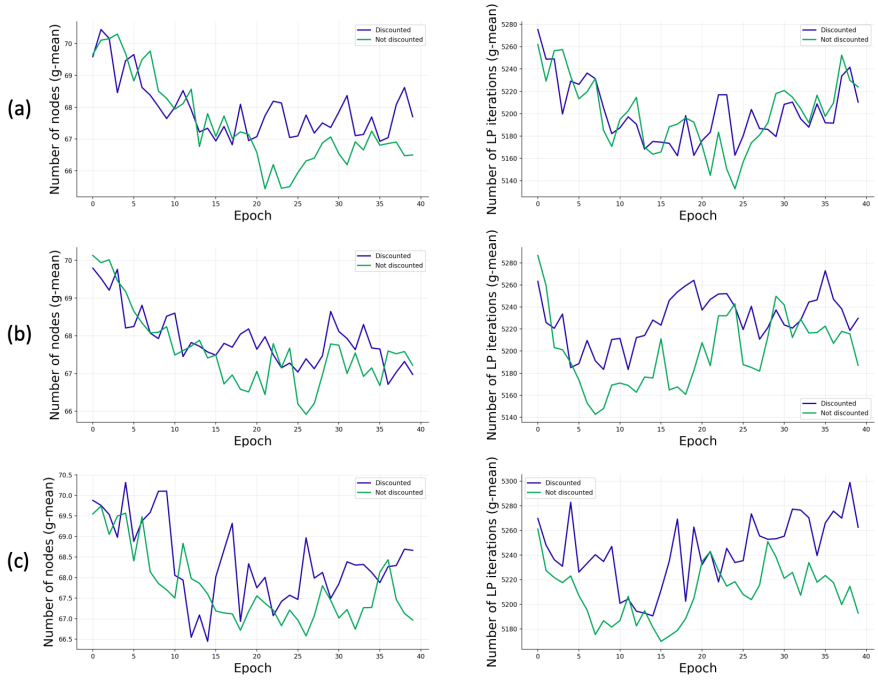


Figure 5.1: Validation results for $\gamma = 1$ (green) and $\gamma = 0.99$ (blue). From top to bottom, figures show results for the different rewards: (a) open, (b) idle and (c) gap. The metrics used are the geometric mean of two magnitudes: total node count (left) and number of LP iterations (right).

Having settled for no discount, we compare the performance of the different rewards in Figure 5.2. While there are no significant differences among them, the gap reward seems to perform systematically worse. Even though the discrepancy between the remaining two is small, the open reward shows slightly superior results and will therefore be used in the upcoming experiments.

Finally, we study the effect of the initial policy. Recall that the agent's parametrization is not initialized at random. Instead, the agent is pretrained using a supervised learning approach. We consider two alternatives to carry out this procedure. In particular, we vary the expert used for supervision. Both experts are based on the

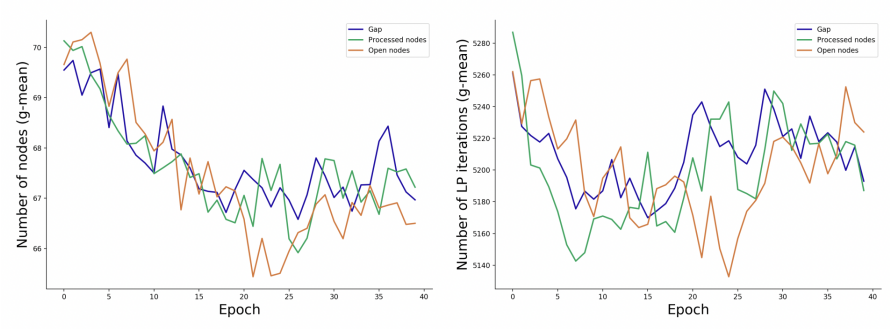


Figure 5.2: Validation results for the three reward types with $\gamma = 1$. The metrics are the geometric mean of the number of processed nodes (left) and the number of LP iterations (right).

full strong branching (FSB) rule. We refer to Section 3.3 for an overview of this rule.

The first expert is SCIP's implementation of FSB (sFSB), while the second is a custom-made vanilla version due to DS4DM [15] (vFSB). The main difference lies in the branching data usage (or lack thereof). In sFSB, information collected from the candidate children nodes is not only leveraged for scoring. For example, if one node is found to be infeasible during a tentative branching process, we can extract a valid inequality for the node under consideration. In this way, the LP relaxation is tightened and the scoring process is restarted. Internal solver statistics are also updated on the basis of this information. On the contrary, vFSB exploits the acquired knowledge for scoring purposes exclusively.

In terms of node creation efficiency, sFSB is a better rule. This claim will be corroborated through a more thorough evaluation in Section 5.3.3. However, it is an inferior expert, as can be seen in Figure 5.3 (notice the first data point). We can justify this behavior based on the explainability of each expert's demonstrations. If we understand "taking an action" as a synonym of branching on a variable, the sFSB agent is allowed to transition through a series of environment states without taking an action. These internal states are unavailable to the apprentice agent, which may find no correlation between the initial state and the subsequent expert action. Conversely, the vFSB expert acts sub-optimally, but its behavior is closer to the one the agent is allowed to have and, consequently, it is easier to imitate.

Let us denote the resulting imitation learning policies as sIL and vIL, as to stress which expert was used for training. We have determined that vFSB is a superior expert. However, we have yet to analyze the differences between using sIL and vIL as initial policies. Going back to Figure 5.3, let us now focus on the whole training curves. Starting from the sIL policy, the RL training is able to improve the performance significantly. Yet, it can at best reach the level of the initial vIL policy. Conversely, in the other setting we achieve smaller gains, specially in terms of number of nodes. Notice that equal performance does not mean equal policy. We are

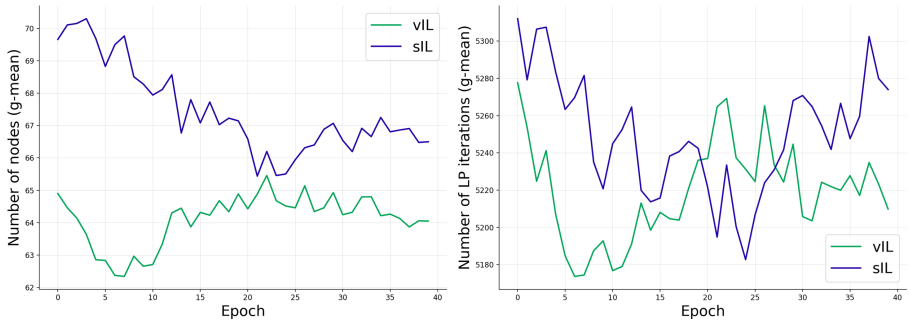


Figure 5.3: Validation results with the two initial policies: vFSB (green) and sFSB (blue). The metrics are the geometric mean of the number of processed nodes (left) and the number of LP iterations (right).

starting from two distinct points in policy space, and then tracking the evolution through a one dimensional metric.

We observe that one initial policy leads to better gains, but the other results in an overall superior performance. This gives rise to a question: is it possible to achieve such gains while starting from the better performing policy? To answer this question, we propose two hypothesis that explain the underwhelming results:

- We could be stuck in a **local minimum**. By starting in such a good policy we may be hindering our chances for improvement if said policy is in or close to a local minimum.

- We could have achieved **optimal behavior**, conditional to the information given. Recall that the experts (both sFSB and vFSB) have access to a different state representation, namely, the LP bound degradations. In consequence, they operate in a different MDP, hence the expert's policy may be unachievable under the agent's settings.

Let us assume the first hypothesis is true, i.e., there is a better policy within the defined MDP (a global minimum) but the initial parameterization lies close to a local a minimum, towards which the agent converges. We try to overcome this obstacle by considering yet a third initial policy. Specifically, we obtain an undertrained policy by prematurely stopping the imitation learning training process. We will refer to it as the uIL policy.

Figure 5.4 shows the loss curve resulting from the full training procedure with vFSB. The red point indicates the stage at which the process was terminated to obtain uIL. The RL training curve is presented in Figure 5.5. We can observe a more irregular evolution, in addition to a poorer performance. Moreover, other undertrained policies were generated (by stopping the pretraining process at different stages) and

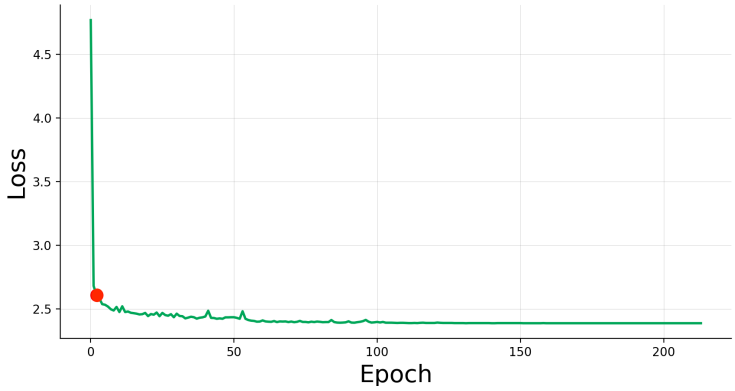


Figure 5.4: Evolution of the validation loss during training under the imitation learning scheme with vFSB as an expert.

tested with analogous outcomes.

Unfortunately, we cannot fully determine the reasons why we cannot draw our performance metrics further down. In the reminder of this chapter, a more thorough evaluation will be presented. Later in Chapter 6, these results will be used to better assess the limitations of the approach and propose possible solutions.

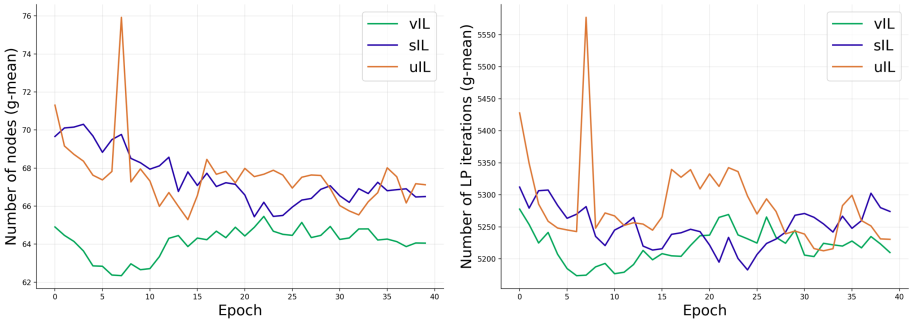


Figure 5.5: Validation results for three initial policies: fully trained sFSB (blue), fully trained vFSB (green) and undertrained vFSB (orange). The metrics are the geometric mean of the number of processed nodes (left) and the number of LP iterations (right).

5.3.3. Evaluation

In this section we present a more detailed experimental comparison of the different branching rules. In particular, we follow a benchmarking methodology similar to the one in Gasse et al. [13], but increasing the number of test instances to 50, in

order to refine our results. Once again, performance variability is purposely generated, using 5 seeds. Therefore, the number of samples is 250. The RL-based policies that will be assessed are defined using the weights that produced best validation performance during training, in terms of number of explored nodes (see Figure 5.3). All the experiments were run using a GPU accelerated machine.

Results are shown in Table 5.1. All instances are solved within the time limit, hence we only report the total node count (in geometric mean), the average performance variability score and the elapsed time (in 1-shifted geometric mean). See Section 3.5 for a justification of this choice. The same testing procedure was repeated for easy and hard instances (see Section 4.4 for an explanation of such distinction).

Let us first analyze the results of the FSB rules. As we anticipated at the beginning of the chapter, sFSB performs better than vFSB, both in terms of node creation and solution time. This result is expected, as the sFSB is allowed to leverage the information it gains in a variety of ways, as opposed to only for branching. However, vFSB exhibits a smaller performance variability. The added variability of sFSB could be a consequence of the node tightening, which highly depends on the order in which variables are considered.

In spite of its superior performance, we can again corroborate that sFSB is a worse expert. This becomes apparent when comparing the sIL and vIL policies. In this sense, vIL does better on all metrics, even in the task of generalization to larger instances. Notice that both policies outperform their respective expert in terms of time, which is, in practice, the most important metric.

Let us now address the RL policies. On the easy set, both attain gains: a 7% improvement in node count in the case of sRL and a 4% for vRL. Furthermore, the RL training procedure is able to take the sIL policy to the performance level of vRL. However, the results on the hard set show a very different behavior. While sRL ob-

Policy	Easy			Hard		
	Nodes	ν	Time	Nodes	ν	Time
sFSB	3	1.10	4.01	76	0.87	109.0
sIL	55	0.65	1.87	920	0.64	11.6
sRL	51	0.66	1.85	909.25	0.6	11.6
vFSB	24	1.05	6.40	378	0.55	278.6
vIL	51	0.62	1.84	842	0.53	11.3
vRL	49	0.62	1.84	1563	2.29	20.0
Reliability	7	1.49	2.66	769	0.93	20.3

Table 5.1: Experimental comparison of all the ML-based policies considered, the experts and reliability branching. Experiments are run over 50 instances and 5 seeds. All instances are solved to optimality. The metrics are the geometric mean of the node count (Nodes) the average performance variability (ν) and the 1-shifted geometric mean of elapsed time (Time).

Policy	Heuristics off			Heuristics on		
	Nodes	ν	Time	Nodes	ν	Time
sFSB	3	1.10	4.01	3	1.01	3.89
sIL	55	0.65	1.87	53	0.64	1.84
sRL	51	0.66	1.85	51	0.68	1.82
vFSB	24	1.05	6.40	24	1.05	6.25
vIL	51	0.62	1.84	50	0.63	1.82
vRL	49	0.62	1.84	48	0.62	1.81
Reliability	7	1.49	2.66	7	1.49	2.59

Table 5.2: Experimental comparison of all the ML-based policies considered, the experts and reliability branching. Experiments are run over 50 easy test instances and 5 seeds. All instances are solved to optimality. The metrics are the geometric mean of the node count (Nodes) the average performance variability (ν) and the 1-shifted geometric mean of elapsed time (Time). Primal heuristics were enabled for the experiments shown in the right column.

tains a 1% gain in node count, this does not translate into faster solving processes. In the case of vRL, we see that the policy is not able to generalize outside its train set, yielding an increase in both number of visited nodes and elapsed time. This could be a sign of overfitting.

Finally, we can put these results into perspective by comparing them to a widely accepted classical branching rule: reliability branching. As pointed out in Gasse et al. [13], the ML-based rules outperform this standard heuristic in terms of solution time. This can be due to the fact that, in the case of reliability branching, computations cannot be accelerated by a GPU.

Let us lastly address the matter of primal heuristics. As we discussed in Section 4.2, primal heuristics are disabled during training. Table 5.2 presents a comparison of the effect of allowing primal heuristics at test time, with respect to maintaining the previous settings. The experimental procedure is identical to the one in Table 5.1, although in this case only easy instances are considered. These results demonstrate that all policies are affected similarly and therefore: (i) primal heuristics can be considered as independent processes that can be enabled after the training stage, and (ii) all of the conclusions we had drawn still hold under the new solver setting.

5.3.4. A specialized policy

In the previous sections, we have presented a series of experiments that provide insights on both the proposed method and the problem we are addressing. In this section, we will abandon the practical setting momentarily to ask a more fundamental question.

In Table 5.1 we observed that the agent is unable to generalize to larger instances under some settings. In fact, generalization is a known issue in deep reinforcement learning, which the RL community has recently started to attempt to solve (see,

Policy	Nodes	ν	Time
vFSB	32	1.01	7.68
vIL	73	0.57	1.84
tRL	68	0.59	1.82

Table 5.3: Experimental comparison of the vFSB rule, the vIL policy and the instance-specific policy tRL. Experiments are run over 20 easy test instances and 5 seeds. All instances are solved to optimality. The metrics are the geometric mean of the node count (Nodes) the average performance variability (ν) and the 1-shifted geometric mean of elapsed time (Time).

e.g., [83]). On the other hand, we would like to test to what extent we can expect to improve the initial policy using reinforcement learning. For this, we propose another experiment, in which we drop the generalization constraint. We train a tailored RL policy (tRL) by reproducing the training procedure in Algorithm 5 with only one instance at a time (for a total of 20 instances). In this way we obtain 20 different policies that specialize on each particular instance. Note that the reduction of the number of instances (from 50 in the previous section to 20) was solely due to time limitations. Notice also that in this case we train on instances from the (easy) test set. The initial policy used for this experiment was vIL.

Results are shown in Table 5.3. The evaluation procedure is identical to the one described in the previous section, except for the reduction in the number of samples. We compare the tailored policy to two other: vIL (so that we can perceive the improvement) and vFSB (so that we can assess how close we are to "ideal" behavior). Notice that we choose vFSB as a comparison because, contrary to sFSB, it shares the same action space as the ML-based policies.

Interestingly, there is a great gap between the performance of vFSB and that of the specialized tRL. A proportion of this gap can be of course attributed to a possibly imperfect learning procedure. However, this discrepancy can also be due to the different state representation. In other words, this calls into question whether or not we can expect close-to-expert performance without expert privileges.

6

Conclusions and future work

6.1. Discussion of the presented work

In this thesis, we have laid the groundwork for the integration of reinforcement learning methodologies into the branch-and-bound search process. To do this, we defined an appropriate learning framework, as well as a series of progressively more suitable models. The experiments that were carried out reveal the great complexity of the task we are addressing. In particular, we deal with partial observability and a large state space, as well as a non-stationary state distribution, which is due to the differences among instances. This hinders our capabilities for generalization.

We propose a model that effectively improves the performance of other methods, such as the imitation learning scheme of Gasse et al. [13] and the widely accepted reliability branching [57]. However, these performance gains are small and one may argue that the extra computational cost is not justified.

In spite of this, the experiments that were carried out have provided important knowledge about the limitations of the ML-based approaches, which is a first step to overcoming them. First of all, we should stress once again that we are working on the restricted setting of learning one policy per instance class. Even though this is still a very interesting setting for practical applications, ML-models are still far from having the generalization capabilities to learn universal policies. Second, we should make a distinction within the ML algorithms if we wish to study the specific limitations of each approach. On the one hand, we can learn by imitation. In this case, the choice of expert poses a problem. Indeed, as we saw in Chapter 5 an expert that uses a different state representation (vFSB) is not ideal, and if furthermore the expert does not share an action space with the apprentice (sFSB), performance can be greatly affected. In general, we cannot expect the agent to converge to the expert's policy if the information we provide is different. On the other hand, we can consider the expert-free approach of reinforcement learning. In this case, the main

weakness could be the generalization ability, even within the same instance class. This is due to the ill-posed problem of learning on non-stationary state distributions. Nonetheless, thanks to the constant advancements in the field of RL, this seems like a less restrictive and therefore more promising research direction.

6.2. Future work directions

Considering the advances presented in this report, we conclude with the identification of a series of interesting research directions, namely:

1. Testing the proposed method in other instance classes. In particular, on non-binary problems.

2. With regard to improving the performance of RL-based policies we can distinguish two possibilities:

   - As we discussed in Chapter 5, we may be close to reaching the optimal policy within the MDP we are considering. This would explain the reduced gains when starting from a better initial policy. If this is the case, we could change the MDP by, e.g., modifying the actor's state representation.
   - If, on the contrary, the problem is a suboptimal training procedure, whereas there is room for improvement, a sensible adjustment would be to upgrade the critic architecture and/or state representation. Indeed, the critic is, to a great extent, responsible for guiding the search towards better parametrizations of the policy. It would be interesting to consider an extension of the state representation, at the expense of more costly training sessions. In particular, we propose to include, e.g., strong branching scores. This idea is motivated by the recently published work of Vinyals et al. [5]. The authors learn to play the game of StarCraft, where the actor has only partial observability of the game development. However, the critic is given a full representation of the state, which is a key factor to achieve the performance levels they report.

3. Using a learning agent opens up the possibility of coordinating the variable selection rule with the node selection rule. There is a precedent for ML-based node selection methods (see e.g. [84]). It would be interesting to integrate the two decision-making strategies, in a way that they jointly optimize the total node count.

References

[1] Claude E Shannon. Programming a computer for playing chess. In *Computer chess compendium*, pages 2–13. Springer, 1988.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[3] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937, 2016.

[4] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[5] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

[6] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine Learning for Combinatorial Optimization: a Methodological Tour d'Horizon. *arXiv preprint arXiv:1811.06128*, 2018.

[7] Ailsa Land and Alison Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.

[8] Pieter L van den Berg, Peter Fiskerstrand, Karen Aardal, Jørgen Einerkjær, Trond Thoresen, and Jo Røislien. Improving ambulance coverage in a mixed urban-rural region in norway using mathematical modeling. *PloS one*, 14(4): e0215385, 2019.

[9] Miguel Constantino, Xenia Klimentova, Ana Viana, and Abdur Rais. New insights on integer-programming models for the kidney exchange problem. *European Journal of Operational Research*, 231(1):57–68, 2013.

[10] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990. ISBN 0716710455.

[11] Tobias Achterberg and Roland Wunderling. Mixed integer programming: Analyzing 12 years of progress. In *Facets of Combinatorial Optimization*, pages 449–481. Springer, 2013.

[12] Andrea Lodi and Giulia Zarpellon. On learning and branching: a survey. *Top*, 25(2):207–236, 2017.

[13] Maxime Gasse, Didier Chetelat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks. In H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 15554–15566. Curran Associates, Inc., 2019.

[14] Giulia Zarpellon, Jason Jo, Andrea Lodi, and Yoshua Bengio. Representing branch-and-bound search trees to learn branching. Unpublished, 2019.

[15] DS4DM. Canada Excellence Research Chair in Data Science for Real-Time Decision-Making at Polytechnique Montréal. https://cerc-datascience.polymtl.ca.

[16] Stuart J Russell and Peter Norvig. *Artificial Intelligence: a modern approach*. Malaysia; Pearson Education Limited, 2016.

[17] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[18] Andre Esteva, Brett Kuprel, Roberto A Novoa, Justin Ko, Susan M Swetter, Helen M Blau, and Sebastian Thrun. Dermatologist-level classification of skin cancer with deep neural networks. *Nature*, 542(7639):115, 2017.

[19] Athanasios S Polydoros and Lazaros Nalpantidis. Survey of model-based reinforcement learning: Applications on robotics. *Journal of Intelligent & Robotic Systems*, 86(2):153–173, 2017.

[20] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521 (7553):436–444, 2015.

[21] Li Deng, Dong Yu, et al. Deep learning: methods and applications. *Foundations and Trends in Signal Processing*, 7(3–4):197–387, 2014.

[22] Richard S Sutton and Andrew G Barto. *Introduction to reinforcement learning*. MIT press Cambridge, 1998.

[23] Dean A Pomerleau. Efficient training of artificial neural networks for autonomous navigation. *Neural Computation*, 3(1):88–97, 1991.

[24] Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

[25] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8 (3-4):279–292, 1992.

[26] John N Tsitsiklis and Benjamin Van Roy. Analysis of temporal-diffference learning with function approximation. In *Advances in Neural Information Processing Systems*, pages 1075–1081, 1997.

[27] Gerald Tesauro. Practical issues in temporal difference learning. In *Advances in Neural Information Processing Systems*, pages 259–266, 1992.

[28] Martin Riedmiller. Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In *European Conference on Machine Learning*, pages 317–328. Springer, 2005.

[29] Martin Riedmiller, Thomas Gabel, Roland Hafner, and Sascha Lange. Reinforcement learning for robot soccer. *Autonomous Robots*, 27(1):55–73, 2009.

[30] Martin Riedmiller and Heinrich Braun. A direct adaptive method for faster backpropagation learning: The rprop algorithm. In *Proceedings of the IEEE international conference on neural networks*, volume 1993, pages 586–591. San Francisco, 1993.

[31] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[32] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[33] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, et al. Massively parallel methods for deep reinforcement learning. *arXiv preprint arXiv:1507.04296*, 2015.

[34] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[35] John Schulman, Sergey Levine, Philipp Moritz, Michael Jordan, and Pieter Abbeel. Trust region policy optimization. In *Proceedings of the 32nd International Conference on Machine Learning-Volume 37*, pages 1889–1897. JMLR. org, 2015.

[36] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[37] George Dantzig, Ray Fulkerson, and Selmer Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the Operations Research Society of America*, 2(4):393–410, 1954.

[38] Ralph E Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical society*, 64(5):275–278, 1958.

[39] Egon Balas, Sebastian Ceria, Gérard Cornuéjols, and N Natraj. Gomory cuts revisited. *Operations Research Letters*, 19(1):1–9, 1996.

[40] Ralf Borndörfer, Carlos E Ferreira, and Alexander Martin. Decomposing matrices into blocks. *SIAM Journal on Optimization*, 9(1):236–269, 1998.

[41] Jean-Maurice Clochard and Denis Naddef. Using path inequalities in a branch and cut code for the symmetric traveling salesman problem. In *IPCO*, pages 291–311, 1993.

[42] Denis Naddef. Polyhedral theory and branch-and-cut algorithms for the symmetric TSP. In *The traveling salesman problem and its variations*, pages 29–116. Springer, 2007.

[43] IBM ILOG CPLEX Optimization Studio. URL https://www.ibm.com/products/ilog-cplex-optimization-studio.

[44] Ambros Gleixner, Michael Bastubbe, Leon Eifler, Tristan Gally, Gerald Gamrath, Robert Lion Gottwald, Gregor Hendel, Christopher Hojny, Thorsten Koch, Marco E. Lübbecke, Stephen J. Maher, Matthias Miltenberger, Benjamin Müller, Marc E. Pfetsch, Christian Puchert, Daniel Rehfeldt, Franziska Schlösser, Christoph Schubert, Felipe Serrano, Yuji Shinano, Jan Merlin Viernickel, Matthias Walter, Fabian Wegscheider, Jonas T. Witt, and Jakob Witzig. The SCIP Optimization Suite 6.0. Technical report, Optimization Online, July 2018. URL http://www.optimization-online.org/DB_HTML/2018/07/6692.html.

[45] LLC Gurobi Optimization. Gurobi optimizer reference manual, 2019. URL http://www.gurobi.com.

[46] Martin WP Savelsbergh. Preprocessing and probing techniques for mixed integer programming problems. *ORSA Journal on Computing*, 6(4):445–454, 1994.

[47] Matteo Fischetti, Fred Glover, and Andrea Lodi. The feasibility pump. *Mathematical Programming*, 104(1):91–104, 2005.

[48] Timo Berthold. Primal heuristics for mixed integer programs. Master's thesis, Technischen Universität Berlin, 2006.

[49] Pierre Le Bodic and George Nemhauser. An abstract model for branching and its application to mixed integer programming. *Mathematical Programming*, 166 (1-2):369–405, 2017.

[50] Tobias Achterberg. *Constraint Integer Programming.* PhD thesis, 01 2007.

[51] Hendrik W Lenstra Jr. Integer programming with a fixed number of variables. *Mathematics of Operations Research*, 8(4):538–548, 1983.

[52] Karen Aardal, Cor AJ Hurkens, and Arjen K Lenstra. Solving a system of linear diophantine equations with lower and upper bounds on the variables. *Mathematics of Operations Research*, 25(3):427–442, 2000.

[53] Michel Bénichou, Jean-Michel Gauthier, Paul Girodet, Gerard Hentges, Gerard Ribière, and O Vincent. Experiments in mixed-integer linear programming. *Mathematical Programming*, 1(1):76–94, 1971.

[54] JJH Forrest, JPH Hirst, and JOHN A Tomlin. Practical solution of large mixed integer programming problems with umpire. *Management Science*, 20 (5):736–773, 1974.

[55] Tobias Achterberg, Thorsten Koch, and Alexander Martin. Branching rules revisited. *Operations Research Letters*, 33(1):42–54, 2005.

[56] Jeff T Linderoth and Martin WP Savelsbergh. A computational study of search strategies for mixed integer programming. *INFORMS Journal on Computing*, 11(2):173–187, 1999.

[57] Tobias Achterberg and Timo Berthold. Hybrid branching. In *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*, pages 309–311. Springer, 2009.

[58] Emilie Danna and Andrea Lodi. Using infeasible nodes to select branching variables, 2009. URL http://www.or.deis.unibo.it/andrea/pscost-ISMP2009.pdf.

[59] J-M Gauthier and Gerard Ribiere. Experiments in mixed-integer linear programming using pseudo-costs. *Mathematical Programming*, 12(1):26–47, 1977.

[60] Wasu Glankwamdee and Jeff Linderoth. Lookahead branching for mixed integer programming. Technical report, Technical Report 06T-004, Lehigh University, 2006.

[61] Fatma Kılınç Karzan, George L Nemhauser, and Martin WP Savelsbergh. Information-based branching schemes for binary linear mixed integer problems. *Mathematical Programming Computation*, 1(4):249–293, 2009.

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

[63] Giovanni Di Liberto, Serdar Kadioglu, Kevin Leo, and Yuri Malitsky. Dash: Dynamic approach for switching heuristics. *European Journal of Operational Research*, 248(3):943–953, 2016.

[64] Serdar Kadioglu, Yuri Malitsky, and Meinolf Sellmann. Non-model-based search guidance for set partitioning problems. In *Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.

[65] Alejandro Marcos Alvarez, Quentin Louveaux, and Louis Wehenkel. A supervised machine learning approach to variable branching in branch-and-bound. 2014.

[66] Alejandro Marcos Alvarez, Quentin Louveaux, and Louis Wehenkel. A machine learning-based approximation of strong branching. *INFORMS Journal on Computing*, 29(1):185–195, 2017.

[67] Alejandro Marcos Alvarez, Louis Wehenkel, and Quentin Louveaux. Online learning for strong branching approximation in branch-and-bound. 2016.

[68] Elias Boutros Khalil, Pierre Le Bodic, Le Song, George Nemhauser, and Bistra Dilkina. Learning to branch in mixed integer programming. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[69] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral networks and locally connected networks on graphs. *arXiv preprint arXiv:1312.6203*, 2013.

[70] Thorsten Koch, Tobias Achterberg, Erling Andersen, Oliver Bastert, Timo Berthold, Robert E Bixby, Emilie Danna, Gerald Gamrath, Ambros M Gleixner, Stefan Heinz, et al. Miplib 2010. *Mathematical Programming Computation*, 3(2):103, 2011.

[71] Emilie Danna. Performance variability in mixed integer programming. In *Workshop on Mixed Integer Programming, Columbia University, New York*, volume 20, 2008.

[72] Yuji Shinano, Tobias Achterberg, Timo Berthold, Stefan Heinz, and Thorsten Koch. Parascip: a parallel extension of scip. In *Competence in High Performance Computing 2010*, pages 135–148. Springer, 2011.

[73] MIPLIB 2017, 2018. http://miplib.zib.de.

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

[75] Nicholas Gould and Jennifer Scott. A note on performance profiles for benchmarking software. *ACM Transactions on Mathematical Software (TOMS)*, 43 (2), 2016.

[76] Martina Fischetti, Andrea Lodi, and Giulia Zarpellon. Learning MILP resolution outcomes before reaching time-limit. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 275–291. Springer, 2019.

[77] Matteo Fischetti and Michele Monaci. Exploiting erraticism in search. *Operations Research*, 62(1):114–122, 2014.

[78] Andrew Gilpin and Tuomas Sandholm. Information-theoretic approaches to branching in search. *Discrete Optimization*, 8(2):147–159, 2011.

[79] Ambros Gleixner, Gregor Hendel, Gerald Gamrath, Tobias Achterberg, Michael Bastubbe, Timo Berthold, Philipp Christophel, Kati Jarck, Thorsten Koch, Jeff Linderoth, et al. Miplib 2017: Data-driven compilation of the 6th mixed-integer programming library. Technical report, Technical report, Optimization Online, 2019.

[80] Kevin Leyton-Brown, Mark Pearson, and Yoav Shoham. Towards a universal test suite for combinatorial auction algorithms. In *Proceedings of the 2nd ACM conference on Electronic commerce*, pages 66–76. Citeseer, 2000.

[81] PySCIPOpt: Python interface for the SCIP Optimization Suite. URL `https://github.com/SCIP-Interfaces/PySCIPOpt`.

[82] Hado P van Hasselt, Arthur Guez, Matteo Hessel, Volodymyr Mnih, and David Silver. Learning values across many orders of magnitude. In *Advances in Neural Information Processing Systems*, pages 4287–4295, 2016.

[83] Karl Cobbe, Oleg Klimov, Chris Hesse, Taehoon Kim, and John Schulman. Quantifying generalization in reinforcement learning. *arXiv preprint arXiv:1812.02341*, 2018.

[84] Ashish Sabharwal, Horst Samulowitz, and Chandra Reddy. Guiding combinatorial optimization with UCT. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, pages 356–361. Springer, 2012.

A

Appendix: Critic state representation

Feature	**T**	**S**	**E**
Statistics on the number of LP iterations with distinctions on node type and reason.	P	Global	D
Solving time.	P	Global	D
Quantile statistics on the number of open nodes.	P	Global	D
Averaged statistics of pseudocosts.	P	Global	D
Primal and dual bounds.	P	Global	D
Node counts by type.	P	Global	D
Gap.	P	Global	D
Number of backtracks.	P	Global	D

Table A.1: High-level description of the features extracted to build the critic's state representation.