Conservatoire national des arts et métiers

ÉCOLE DOCTORALE SCIENCES DES MÉTIERS DE L'INGÉNIEUR

Centre d'études et de recherche en informatique et communications

Thèse de doctorat

Présentée par : **Paul STRANG**
Soutenue le : **28 Mai 2026**
Discipline : **Informatique**

Learning to solve repeated combinatorial optimization problems exactly

THÈSE dirigée par :
Mme Safia KEDAD-SIDHOUM Professeur des universités, Cnam
M. Emmanuel RACHELSON Professeur, ISAE-SUPAERO

et co-encadrée par :
M. Zacharie ALES Professeur associé, ENSTA IP Paris
M. Côme BISSUEL Ingénieur de recherche, EDF R&D
M. Olivier JUAN Ingénieur de recherche, EDF R&D

Jury

M. Riad AKROUR	Chargé de recherche, INRIA	Rapporteur
M. Thibaut VIDAL	Professeur des universités, Polytechnique Montréal	Rapporteur
M. Mathieu BESANCON	Chargé de recherche, INRIA	Examinateur
Mme. Sourour ELLOUMI	Professeur des universités, ENSTA IP Paris	Examinatrice
M. Axel PARMENTIER	Professeur des universités, École Nationale des Ponts et Chaussées	Examinateur

More life

Acknowledgements

Many, many people have had a direct, meaningful impact on the success of this thesis, and have earned my heartfelt gratitude.

First and foremost, I would like to sincerely thank the members of my jury – Dr. Riad Akrour, Prof. Thibaut Vidal, Prof. Sourour Elloumi, Dr. Mathieu Besançon and Prof. Axel Parmentier – for dedicating their time to reviewing my work. As recognized experts in machine learning and optimization, it is a great privilege for me to share with you my work and personal thoughts on how these two fields intersect. I look forward to hearing your comments and objections.

Then, of course, I would like to thank my (numerous) supervisors for their continual support and guidance. Côme, this thesis probably owes more to you than to anyone else. Thank you for being such a skilled and determined scout, as well as for your rare thoughtfulness throughout these (almost four) years. I consider myself lucky to have had the chance to work with you.

Olivier, you are by far the closest collaborator I have had over these years. Thank you for remaining so relentlessly engaged with this research project, and for helping me grow in every way you could. I know I owe you the completion of a fair share of the work presented in this thesis, and I want to let you know that I do intend on paying you back one way or the other.

Zacharie, your (extensive) writing feedback has been invaluable throughout this thesis. Thank you for always looking for a solution to the wildest theoretical, experimental, or rendering issues I encountered, as well as for your exceptional diplomatic skills – which I would never have imagined this thesis would require. I believe that every PhD student would be happy to have you as their supervisor, actually, I know it for a fact.

Safia, your insights and judgment have been essential to the success of this thesis. Thank you for always keeping your door open, and for keeping my best interests at heart in all circumstances. I am deeply impressed by your work ethic and how much you care for your students. I will genuinely miss your guidance and mentoring in the future.

Emmanuel, I have been incredibly lucky to find in you someone as stubborn as I am. Thank you for having the sensibility to always push me to aim higher, while showing your appreciation in the tough (as well as the not so tough) moments. Although our exchanges have certainly been scarcer than either of us would have liked, this manuscript bears your fingerprints throughout its pages. I am genuinely grateful for what you brought to the table, and for how much I learned from you since the first class of SDD.

I also had myself the privilege of supervising the work of two young researchers in machine learning and optimization, Adéchola and Hippolyte. I would like to thank them both for their exceptional dedication throughout our collaboration. Adéchola, I believe that you demonstrated remarkable persistence and determination over the course of our collaboration, and I send you my warmest wishes for your future endeavors. Hippolyte, while our collaboration was not enough to convince you to pursue a research career, I am absolutely confident in your future, regardless of what you choose to set your mind to.

Over the course of the last four years, I have been lucky enough to call many places home. I would like to thank all the people in these places who made me feel like I belonged. First, I would like to thank the members of the Sureli team at ISAE-Suapero for their genuinely warm welcome at the very start of this thesis. The scientific stimulation and dedication there sparked a longing to learn that stayed with me throughout the rest of this thesis.

Then, I would like to thank all the people from EDF who, first at P12 and then at R36, have contributed to making my PhD life happier. In particular, I would like to extend my warmest thanks to Emilie – arguably my sixth supervisor – for investing so much energy in the success of this thesis, and in my travelling of the world. Thanks also to everyone with whom I have had the pleasure to share a lunch, a coffee break or a shuttle trip, including but not limited to Adrien, Alain, Alex, Andréa, Antoine, Bruno, Charlotte, Cécile, Claire, Jean-Armand, Laura, Luc, Orane, Rodolphe and Riad. Despite my notorious dislike of outer Paris commute, I cannot recall of a single bad day spent in your

company, whether in Chatou or Saclay.

I would also like to thank everyone at the Cnam who shared my daily life throughout these years, and in particular everyone with whom I had the pleasure of sharing the 31.1.73 office. I was more than happy to explore with you the many coffee shops and lunch spots Le Marais has to offer.

In addition to these great places, I was also fortunate enough to be invited to the Zuse Institute in Berlin for a four-month visit. I owe to Suresh, Mark, and Mathieu this opportunity, along with many inspiring conversations which, after months of reflection, allowed me to more fully appreciate the depth of my research field. More generally, my thanks go to all the people at ZIB and IOL who made such an insightful experience possible.

The people who share your ups and downs matter at least as much as those who guide your work. In this spirit, I would like to extend my warmest thanks to the other PhD students I met along the way – and in particular to Adil, Mehdi, PA, Hubert, Hector, Luca, Bianca, Anton and Margot, who helped make my PhD journey a less lonely road.

On the same note, and in a more personal vein, I would also like to thank some of my dearest friends, Alexandre, Axel, Jean, Hugo, Liza, Matthias, Samy, Samson, Sébastien and Stanislas, whose appreciation and esteem matter more than they know.

Finally, I would like to thank my parents, my first and most formidable contradictors, who carefully and lovingly shaped the person I am today.

Lucky me, I get to spend what comes after today with Lise; which, by itself, makes me want to live way too long.

Abstract

Mixed-integer linear programming (MILP) plays a central role in combinatorial optimization (CO), a discipline concerned with finding optimal solutions over typically large but finite sets. Specifically, MILPs offer a general modelling framework for NP-hard problems, and have become indispensable in tackling complex decision-making tasks across fields as diverse as operations research, quantitative finance, and computational biology. Modern MILP solvers are built upon the branch-and-bound (B&B) paradigm, which systematically explores the solution space by recursively partitioning the original problem into smaller subproblems, while maintaining provable optimality guarantees. Since the 1980s, considerable research and engineering effort have gone into refining these solvers, resulting in highly optimized systems driven by expert-designed heuristics tuned over large benchmarks.

Nevertheless, in operational settings where structurally similar problems are solved repeatedly, adapting solver heuristics to the distribution of encountered MILPs can lead to substantial gains in efficiency, beyond what static, hand-crafted heuristics can offer. Recent research has thus turned to machine learning to design efficient, data-driven B&B heuristics tailored to specific instance distributions. The variable selection heuristic, or branching heuristic, plays a particularly critical role in B&B computational efficiency, as it governs the selection of variables along which the search space is recursively split. A key milestone was achieved by Gasse et al. [62], who showed that learning to replicate the decisions of a greedy branching expert via imitation learning (IL) could outperform expert-designed branching heuristics. While subsequent works succeeded in learning effective branching strategies by reinforcement, none have yet matched the performance achieved by IL approaches. This trend extends beyond MILPs, as reinforcement learning (RL) baselines consistently underperform both handcrafted heuristics and IL methods across various combinatorial optimization benchmarks.

Yet, if the performance of IL heuristics is capped by that of the experts they learn from, the performance of RL agents is, in theory, only bounded by the maximum score achievable. To cope

with the credit assignment difficulties induced by sparse rewards, prior works have shifted away from the standard Markov decision process (MDP) framework, finding it impractical for learning efficient branching strategies. In this thesis, we show that these alternative formulations, despite improving the empirical performance of RL baselines, introduce approximations of the B&B dynamics that compromise the asymptotic guarantees of generic RL algorithms. To address this issue, we propose reverting to a principled MDP formulation of the branch-and-bound process, which preserves the convergence properties established by previous contributions without sacrificing optimality. Unlike prior formulations, our framework accommodates the full spectrum of modern RL algorithms, including planning-based approaches that leverage look-ahead search for robust policy improvement.

Inspired by the success of AlphaZero [154] in complex combinatorial board games, we explore extending the applicability of planning-based RL from board games to exact combinatorial optimization. To that end, we introduce PlanB&B, a model-based reinforcement learning agent that leverages a learned internal model of the B&B dynamics to discover improved variable selection strategies. Our computational results across both synthetic and real-world industrial benchmarks indicate that the branching dynamics in B&B can be approximated with sufficient fidelity in latent space to enable policy improvement through planning over a learned model, opening the door to broader applications of model-based reinforcement learning to mixed-integer linear programming in the future.

Keywords : Repeated combinatorial optimization problems, Mixed-integer linear programming, Branch-and-bound, Branching heuristics, Graph neural networks, Reinforcement learning, Model-based reinforcement learning, Interior-point methods, Flow matching

Résumé

La programmation linéaire en nombres entiers (PLNE) occupe une place centrale en optimisation combinatoire, une discipline visant à identifier les meilleurs éléments d'un ensemble discret, généralement de grande taille. En particulier, la PLNE offre un cadre général pour la modélisation de problèmes dits NP-difficiles, et s'est imposée comme solution incontournable dans des domaines variés. Les solveurs de PLNE modernes reposent sur l'algorithme du branch-and-bound (B&B), qui explore l'espace des solutions faisables en partitionnant récursivement le problème de départ en sous-problèmes de taille réduite. Depuis les années 1980, d'importants efforts de recherche ont permis de porter ces solveurs à un haut niveau de performance, grâce notamment à la confection d'heuristiques expertes optimisées sur de larges jeux de données.

Cependant, lorsque des problèmes structurellement similaires sont résolus de manière répétée, adapter les heuristiques des solveurs à la distribution des instances rencontrées peut engendrer des gains de performances substantiels, hors de portée des heuristiques statiques. Des travaux récents se sont ainsi tournés vers les méthodes d'apprentissage automatique pour concevoir des heuristiques de B&B adaptées à des distributions d'instances spécifiques. L'heuristique de sélection de variable, ou heuristique de branchement, joue un rôle déterminant dans l'efficacité du B&B : c'est elle qui détermine à chaque itération la variable selon laquelle l'espace de recherche doit être partitionné. Un palier important a été franchi par Gasse et al. [62], qui ont démontré, dans le cadre de problèmes répétés, qu'un agent de branchement entraîné par imitation (IL) à reproduire les décisions d'un expert glouton à moindre coût pouvait surpasser les performances atteintes par des heuristiques expertes. Si des travaux ultérieurs sont parvenus à apprendre des stratégies de branchement par renforcement, aucun n'est encore parvenue à égaler les performances atteintes par les approches d'IL. Ce constat dépasse le simple cadre de la PLNE : sur les principaux benchmarks d'optimisation combinatoire, l'apprentissage par renforcement (RL) accuse un retard systématique tant vis-à-vis des heuristiques

expertes que des méthodes d'IL.

Toutefois, si les performances des heuristiques d'IL sont plafonnées par celles des experts qu'elles imitent, celles des agents de RL ne sont, en théorie, bornées que par la performance maximale atteignable. Pour contourner certaines difficultés survenant dans le cadre de l'exploration d'espaces de grandes dimensions, les travaux antérieurs présents dans la littérature ont choisi de s'écarter du cadre standard des processus de décision Markoviens (MDP) pour la modélisation du processus de sélection de variable. Dans cette thèse, nous montrons que ces formulations alternatives, bien qu'améliorant les performances empiriques des méthodes de RL, introduisent des approximations qui compromettent les garanties théoriques de ces algorithmes. Afin d'y remédier, nous proposons de revenir à une formulation MDP rigoureuse, qui préserve toutefois les propriétés de convergence empiriques établies par les contributions précédentes. Notre cadre se distingue notamment des formulations antérieures en ce qu'il est compatible avec l'ensemble des algorithmes de RL modernes, y compris les approches fondées sur la planification permettant d'exploiter des méthodes de recherche locale pour garantir une amélioration monotone de la politique.

Inspirés par les succès d'AlphaZero [154] et MuZero [152], nous explorons par la suite l'adaptation de ces approches initialement conçues pour les jeux de plateau à l'optimisation combinatoire exacte. Nos résultats indiquent que le processus de branchement peut être approximé avec une précision suffisante pour permettre, par planification sur un modèle appris, une amélioration effective de la politique, ouvrant ainsi la voie à de futures applications mêlant RL et planification à la PLNE.

Mots-clés : Problèmes d'optimisation combinatoire répétés, Programmation linéaire en nombres entiers mixtes, Séparation et évaluation, Heuristiques de branchement, Réseaux de neurones sur graphes, Apprentissage par renforcement, Apprentissage par renforcement basé sur un modèle, Méthode de point intérieur, Flow matching

Contents

Chapter 1

Introduction

Content

This thesis deals with combinatorial optimization problems, and how to solve them efficiently by leveraging data and computation. It was conducted as a part of a collaboration between EDF R&D and the CNAM. It builds upon and extends the groundwork laid by Marc Etheve [55].

1.1 General introduction

1.1.1 AI's bitter lesson

The Bitter Lesson [56] is an original essay published by Richard Sutton in 2019. It argues, from seven decades of evidence across AI subfields, that the most reliable path to progress has been general-purpose methods that scale with data and computation, rather than the injection of human, domain-specific knowledge[1]. In computer vision, learned features displaced hand-engineered edge and shape detectors; in speech, end-to-end neural models supplanted linguistic pipelines; in games, search coupled with learned evaluation functions (e.g., AlphaZero-style self-play [154]) progressively eclipsed expert-crafted heuristics. The common thread is not clever domain theories, but generic compute-amplified search and learning techniques that improve predictably as resources grow. The claim is intentionally "bitter", as it downplays the long-term value of human insight when it cannot be expressed as a scalable inductive bias.

The contemporary landscape tempers this view[2]. In fact, the success of CNNs, Transformers, and GNNs fundamentally hinges on architectural priors (equivariance, attention, message passing) that are themselves deliberate human designs encoding compact statements of structure. Parallel to these developments, hybrid neuro-symbolic approaches argue that deep learning alone is insufficient for systematic generalization, and advocate integrating neural models with rule-based reasoning to recover compositional structure and logical consistency [26]. Finally, sample-efficiency constraints imposed by physical systems in real-world domains such as robotics and materials design, together with the rising environmental cost of large-scale computation, place a natural premium on exploiting structural knowledge more effectively per unit of compute. Still, Sutton's essay has had a profound influence on modern AI research [89, 142, 157], as its central insight is now broadly acknowledged in the literature: sustained progress arises from general-purpose algorithms that scale with computation, while judiciously exploiting domain structure through effective inductive biases.

[1] For the keen reader, we reproduce the original text by R. Sutton in Appendix A.

[2] Various critics have challenged Sutton's claims, chief among them Rodney Brooks in his rebuttal post *A better lesson*.

This thesis deals with combinatorial optimization problems, and how to solve them efficiently by leveraging data and computation. Since we are looking to push the limits of existing combinatorial optimization frameworks, which result from decades of continued "human" research and engineering efforts, we propose to reflect on the historic development and current state of the field through the lens of Sutton's bitter lesson, and, conversely, to reflect on what combinatorial optimization and more broadly operations research (OR) may teach us about the scope and validity of that lesson.

This original dialogue between AI and OR, two fields that have often evolved in parallel, if not in competition, leads us to forge two guiding principles which, within reason, have served as the main philosophical compass for the work presented in this thesis. First, research in combinatorial optimization has produced invaluable, broadly applicable algorithmic frameworks that leverage plethoric knowledge of combinatorial structures to efficiently solve discrete optimization problems. This extensive knowledge, which also represents the legacy of seven decades of research, should not be discarded by future learning approaches, but rather integrated in a principled way. Second, these general frameworks also depend on a broad spectrum of expert-designed, empirically tuned heuristics that we expect, following Sutton's intuition, to be increasingly refined and, eventually, superseded by learning-based counterparts over time. Accordingly, the contributions of this thesis primarily aim to strengthen general combinatorial optimization frameworks by automating the discovery of high-performing, distribution-aware heuristics, replacing static expert rules with learned components that leverage access to historical data and offline computation.

1.1.2 Combinatorial optimization

Combinatorial optimization is a field located at the intersection of mathematics and computer science. It studies the problem of finding an optimal solution within a discrete, typically finite, set of feasible configurations. In its most general form, a combinatorial optimization problem can be formulated as:

$$\min_{x \in \mathcal{X}} f(x), \tag{1.1}$$

where f denotes the objective (or cost) function, and $\mathcal{X}$ represents the feasible set, often defined implicitly by a system of combinatorial or algebraic constraints. Combinatorial optimization problems arise in numerous domains of science and engineering, from routing fleets and scheduling factory production, to managing power grids, designing communication networks, optimizing electronic cir-

cuits, sequencing genomes, and guiding molecular and drug discovery. Landmark problems include the traveling salesman problem[3] (TSP), the boolean satisfiability problem[4] (SAT), and the knapsack problem[5], each of which captures the essence of decision-making under discrete structural constraints.

From a computational perspective, combinatorial optimization problems are notoriously hard to solve. In fact, most combinatorial optimization problems belong to the class of NP-hard problems, for which no polynomial-time algorithms are known. In the worst case, solving an NP-hard problem requires computational effort that grows exponentially with the number of decision variables. As a result, the field has evolved around the development of both exact resolution methods, capable of guaranteeing optimality (e.g., branch-and-bound, dynamic programming), and approximate resolution or heuristic methods, which trade optimality guarantees for scalability (e.g., local search, simulated annealing, genetic algorithms). While the contributions of this thesis primarily target exact frameworks, in practice the boundary between exact and approximate approaches is considerably more porous than their textbook definitions initially suggest. In fact, most exact methods, in practice, rely on heuristics to guide and accelerate the search. Conversely, exact frameworks are frequently applied to broadly intractable problem instances, not in the hope to obtain a certified optimal solution, but to produce high-quality feasible solutions along with an upper bound on the optimality gap. In industrial settings, this gap can often be interpreted as a bound on the profit foregone by not reaching the true optimum. Unsurprisingly, such information holds substantial value for corporate decision-making; for this reason, exact solvers remain a cornerstone of industrial optimization practice, serving both as standalone tools for certification as well as embedded modules within broader approximate resolution frameworks.

Over the past decades, advances in algorithm design, mathematical modeling, and computing power have significantly expanded the tractable frontier of combinatorial optimization. Yet, despite this progress, many large-scale instances remain beyond the reach of classical algorithms. Furthermore, since both exact and approximate methods typically rely on series of heuristics which are known to be suboptimal, there remains vast potential to further accelerate the resolution of currently tractable problems through the discovery of improved heuristic strategies. This motivates a growing line of research at the intersection of optimization and machine learning [21], which seeks to leverage data-

[3] https://en.wikipedia.org/wiki/Travelling_salesman_problem
[4] https://en.wikipedia.org/wiki/Boolean_satisfiability_problem
[5] https://en.wikipedia.org/wiki/Knapsack_problem

driven models to design faster direct-search heuristics, or guide exact solvers more efficiently. Before engaging with the literature associated with this vibrant, emerging field in Section 1.3, we first take a step back to reflect on the forces that have shaped progress in combinatorial optimization over time, and how they align with the principles outlined in Sutton's essay.

1.1.3 Exploiting inductive biases in combinatorial optimization

Combinatorial optimization has long exemplified the success of elaborate human theory in tackling complex real-world problems. For decades, progress in the field has been driven by the design of sophisticated relaxations, heuristics, polyhedral analysis and decomposition schemes, all underpinned by rigorous mathematical principles. Nevertheless, this observation does not necessarily contradict the principles of the bitter lesson. In fact, when reflecting on the historical progress in mixed-integer linear programming [178], the simplex algorithm [43], duality theory, cutting planes [67], column generation [64], branch-and-bound [106], branch-and-cut [131], Dantzig–Wolfe decomposition [44] for large-scale models, and later branch-and-price [19] can hardly be described as narrow "domain tricks", or "custom heuristic methods". Rather, these are general algorithmic frameworks that abstract the search process from problem-specific reasoning, enabling steady improvements in performance as computational power grows. Even cutting planes, often viewed as handcrafted features, became powerful precisely because they evolved into generic, systematic families (e.g. Chvátal–Gomory, MIR, knapsack-cover) that solvers could generate, test, and separate automatically as time and memory permitted. Similar patterns can be observed beyond linear and mixed-integer linear programming. In satisfiability and constraint programming, conflict-driven clause learning (CDCL) solvers decisively surpassed traditional DPLL-style solvers[6], reliant on fixed, hard-coded strategies, by systematically learning from conflicts detected during resolution to guide subsequent search. Likewise, modern metaheuristics have evolved from rigid, hand-tuned strategies into adaptive, general frameworks that deliver robust performance across a vast range of combinatorial problem classes, partly by adjusting their search parameters dynamically during execution.

Hence, in line with Sutton's bitter lesson, many of the methods that form the backbone of modern industrial solvers are not narrow, domain-specific heuristics, but broad, modular algorithmic frameworks that systematically harness available computation. Naturally, this does not imply that human

[6]DPLL solvers are named after Davis, Putnam, Logemann and Loveland, who contributed to their development in the early 1960s.

knowledge is absent from contemporary solvers. Despite not framing it in these terms, modern combinatorial optimization frameworks leverage various inductive biases to structure and accelerate the search process. Some of these biases are elementary, for example, fixing the first city in the TSP to break rotational symmetry, or exploiting the greedy–optimality property of matroids in the minimum spanning tree problem. Some are deeper, and underpin the very organization of the search process, defining how the solution space is decomposed, explored, and eventually pruned.

More generally, the field of discrete optimization is built on the systematic study of combinatorial and polyhedral structures: identifying exploitable regularities, decompositions, or invariants is often the decisive ingredient that renders a CO problem tractable. In this sense, combinatorial optimization may be viewed as one of the richest sources of inductive biases in algorithmic science. Yet, we argue that such biases must not be injected indiscriminately into hybrid combinatorial optimization learning pipelines. Rather, we believe that understanding which biases should be preserved and which should instead be learned from data is critical for designing effective learning-augmented solvers in the future. In the following, we distinguish three broad families of inductive biases in combinatorial optimization: modeling biases, which shape the structure of the problem formulation; framework biases, which define how the search is organized; and heuristic biases, which guide local decisions within these frameworks.

Modeling bias. As previously discussed, clever problem formulation can yield exponential improvements that brute-force scaling can hardly replicate. In this regard, mathematical modeling offers a compelling counterpoint to *The Bitter Lesson*: in combinatorial optimization, structural knowledge does not merely accelerate computation; it can fundamentally alter the complexity landscape of a problem. Entire classes of instances become tractable not because more compute is available, but because their structure has been properly exposed and exploited. In this context, adopting a strict interpretation of Sutton's thesis, one that would dismiss the structural insights accumulated by the combinatorial optimization community over the past decades, appears misguided. Although combinatorial optimization instances arise from real-world industrial problems and can, in principle, be described in pure natural language, representing them as raw textual inputs conceals the algebraic and polyhedral structure that make efficient solution possible. Recent attempts to train or fine-tune large language models to infer optimal routing or scheduling decisions directly from such unstructured descriptions [90, 181] illustrate this point: while these approaches demonstrate broad applicability, they consistently report optimality gaps of several percent even on moderate-scale instances, and remain

far from competitive with commercial solvers on structured problems. In short, because combinatorial problems often admit rich mathematical structure that is simply too valuable to ignore, this thesis does not attempt to rediscover such structure through learning, but rather to enhance existing algorithmic frameworks that explicitly build upon the accumulated body of knowledge on combinatorial structures. Whether a statistical learning process could eventually rediscover such modeling insights from data alone remains an open question. Current models, however, offer little ground for optimism: their statistical nature limits them to capturing distributional regularities, which falls short of the algebraic and combinatorial reasoning required to derive, for instance, a tight extended formulation or an effective decomposition scheme.

Framework bias Modern combinatorial optimization resolution frameworks build upon strong inductive biases to organize the search. Dynamic programming relies on the assumption of optimal substructure: optimal solutions can be constructed from optimal solutions to smaller subproblems. Likewise, Benders and Dantzig–Wolfe decompositions are built on the expectation that large-scale problems possess an exploitable separable structure that can be leveraged algorithmically. Finally, branch-and-bound (B&B) rests on the premise that linear programming relaxations yield bounds tight enough to effectively guide and prune the search tree. This bias is effective when the underlying integer formulation is strong, namely when the linear relaxation provides a good approximation of the convex hull of integer-feasible solutions. Crucially, this last property has motivated decades of research in polyhedral analysis, aimed at deriving tighter formulations for mixed-integer linear programs. Because these assumptions collectively define the logic of the search process, the contributions of this thesis seek to operate within these frameworks, rather than in place of them.

Heuristic bias Despite their structural generality, the frameworks described above also rely extensively on empirical domain knowledge through expert-crafted heuristic components. A classical example is the pivot selection rule in the simplex algorithm: while the simplex itself provides a general optimization framework, its practical efficiency depends on heuristic choices such as determining which variable enters and which leaves the basis at each iteration. Such heuristic strategies typically offer limited theoretical guarantees, and rely instead on empirical insight accumulated over decades of practice to efficiently guide the search process. This reliance is most visible in branch-and-bound solvers, where node, variable, and cut selection strategies tend to be governed by highly intricate heuristic rules,

refined through extensive tuning over large benchmark suites. Unlike modeling and framework biases, which encode structural properties of the problem or the search, heuristic biases typically encode empirical preferences about how to navigate the solution space efficiently. As such, they are natural candidates for replacement by learned components capable of exploiting regularities that static, hand-crafted rules cannot capture.

Accordingly, this thesis seeks to improve the performance of established combinatorial optimization frameworks by replacing fixed expert-crafted heuristics with learned counterparts. With these principles in place, we now formalize the class of problems and the resolution paradigm considered throughout this thesis.

1.2 Problem statement

This thesis considers the setting of repeated combinatorial optimization problems. In such a setting, instances are assumed to be drawn from a common underlying distribution reflecting recurring operational contexts, industrial processes, or structured families of related decision problems. Consistent with the principles articulated in Section 1.1, we do not aim to replace combinatorial optimization solvers with generic learning systems. Instead, we seek to augment existing solvers with machine learning methods that operate in synergy with the expert knowledge already embedded within these frameworks.

The primary avenue explored in this thesis for accelerating the solution of repeated combinatorial optimization problems is to automate the discovery of high-performing, *distribution-aware* solver heuristics by leveraging historical data and large-scale offline computation. Formal support for this view comes from the No Free Lunch theorems for optimization [177], which establish that algorithmic superiority cannot be distribution-independent: performance gains are necessarily tied to the structure of the instance distribution at hand.[7] Modern solvers implicitly acknowledge this distribution dependence through their highly modular design, operating as meta-algorithms that dynamically activate or combine portfolios of heuristic components based on instance features and the current state of the search. Yet the effectiveness of such architectures ultimately depends on the quality of their fixed heuristic modules. In this context, learning improved, distribution-aware heuristics therefore appears

[7] This perspective aligns with the classical algorithm selection framework of Rice [143], which establishes that no single algorithm dominates across all instances, and that performance varies systematically with problem features.

as a natural path toward enhancing solver performance.

Specifically, in this thesis we investigate the learning of distribution-aware heuristics to guide decision-making **within the branch-and-bound algorithm**, a highly general exact resolution framework capable of addressing a broad spectrum of combinatorial optimization problems. While our developments are presented in the context of mixed-integer linear programming (MILP), branch-and-bound itself extends far beyond MILP and underpins solution methods in constraint programming, satisfiability (SAT), and mixed-integer nonlinear programming [71, 165], among others. Consequently, the contributions presented in this thesis are not confined to the already broad class of mixed-integer linear programs, but are designed to extend, in principle, to other settings in which branch-and-bound serves as the underlying resolution framework. With these principles in place, we now proceed to formally define the class of problems under consideration, together with the resolution paradigm adopted throughout this thesis.

1.2.1 Mixed-integer linear programming

Mixed-integer linear programming offers a general modeling framework for NP-hard optimization problems, and has become indispensable in tackling complex decision-making tasks beyond operations research [83], across fields as diverse as quantitative finance [118], computational biology [74] and machine learning [25]. In this thesis, we consider general mixed-integer linear programs defined as:

$$P : \begin{cases} \min c^\top x \\ l \leq x \leq u \\ Ax \leq b \; ; \; x \in \mathbb{Z}^{|\mathcal{I}|} \times \mathbb{R}^{n-|\mathcal{I}|} \end{cases} \tag{1.2}$$

with n the number of variables, m the number of linear constraints, $l, u \in \mathbb{R}^n$ the lower and upper bound vectors, $A \in \mathbb{R}^{m \times n}$ the constraint matrix, $b \in \mathbb{R}^m$ the right-hand side vector, $c \in \mathbb{R}^n$ the objective function, and $\mathcal{I}$ the indices of integer variables. Throughout this document, we are interested in repeated MILPs of fixed dimension $\{P_i = (A_i, b_i, c_i, l_i, u_i)\}_{i \in N}$ sampled according to an unknown distribution $p_0 : \Omega \to \mathbb{R}^{m \times n} \times \mathbb{R}^m \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n$. We stress that no further structural assumptions are imposed on the MILPs beyond the general formulation above. As a result, the methods proposed in this thesis are formulation-agnostic, and remain fully compatible with parallel advances in polyhedral modeling, extended formulations or decomposition techniques, which can be integrated seamlessly as they become available.

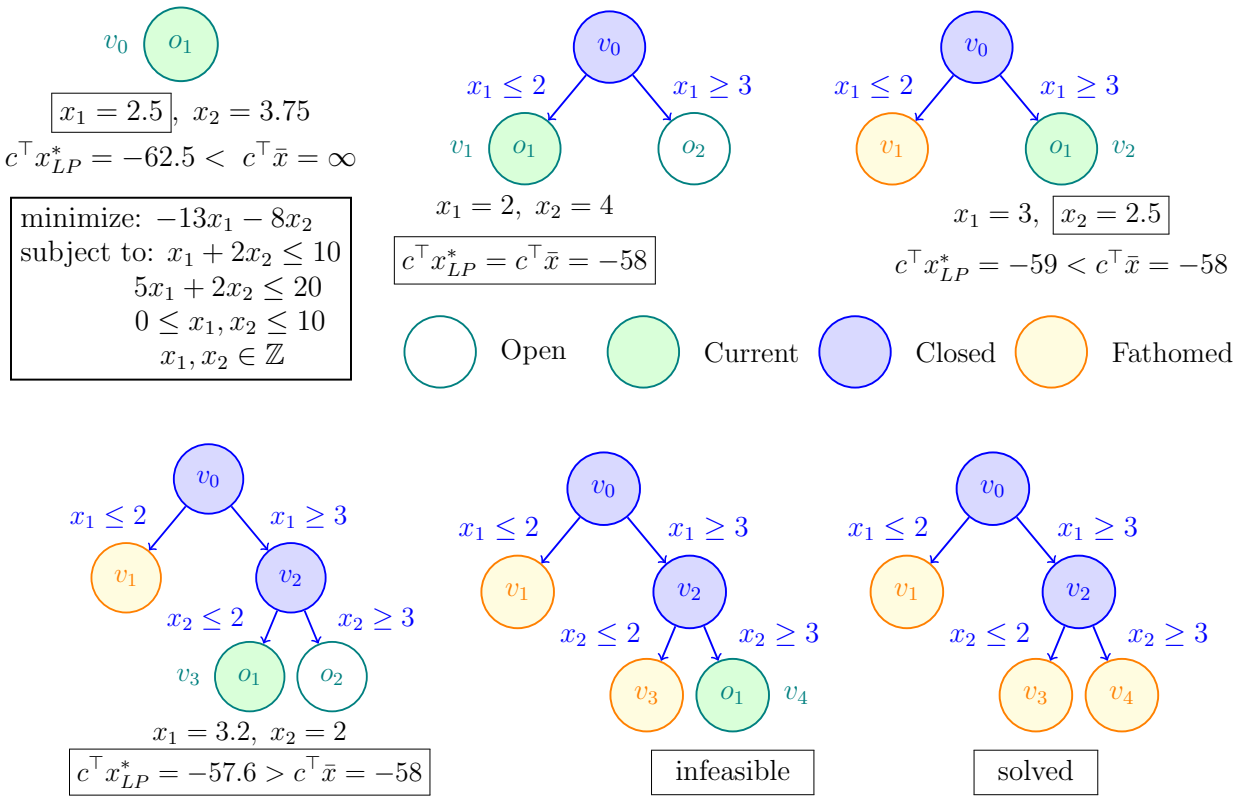


Figure 1.1: Solving a MILP by B&B using variable selection policy π and node selection policy ρ. Each node v_i represents a MILP derived from the original problem, each edge represents the bound adjustment applied to derive child nodes from their parent. Throughout the solving process, v_1 is pruned by integrity, v_3 is pruned by bound, and v_4 by infeasibility.

1.2.2 Branch-and-bound

Modern MILP solvers are built upon the branch-and-bound (B&B) framework [106], which systematically explores the solution space by recursively partitioning the original problem into smaller subproblems, while maintaining provable optimality guarantees. In order to solve MILPs efficiently, the B&B algorithm iteratively builds a binary tree $(\mathcal{V}, \mathcal{E})$ where each node corresponds to a MILP, starting from the root node $v_0 \in \mathcal{V}$ representing the original problem P_0. Throughout the optimization process, B&B nodes are explored sequentially. We denote by $\mathcal{C}$ the set of visited or closed nodes, and $\mathcal{O}$ the set of unvisited or open nodes, such that $\mathcal{V} = \mathcal{C} \cup \mathcal{O}$. Initially, $\mathcal{O} = \{v_0\}$ and $\mathcal{C} = \emptyset$. Additionally, the algorithm maintains an incumbent solution $\bar{x} \in \mathbb{Z}^{|\mathcal{I}|} \times \mathbb{R}^{n-|\mathcal{I}|}$, denoting the best feasible solution found at the current iteration. Its objective value $GUB = c^\top \bar{x}$, called the *global upper bound* on the

optimal value, or *primal bound*, serves as a pruning reference against which open node candidates are compared.[8]

The overall state of the optimization process is thus captured by the triplet $s = (\mathcal{V}, \mathcal{E}, \bar{x})$. We note $\mathcal{S}$ the set of all such triplets. At each iteration, the node selection policy $\rho : \mathcal{S} \to \mathcal{O}$ selects the next node to explore from the set of open nodes. Figure 1.1 illustrates how B&B operates on an example. Let v_t be the node currently selected for expansion. The solver computes $x^*_{LP} \in \mathbb{R}^n$, the optimal solution to the linear programming (LP) relaxation of P_t, the MILP associated with v_t. The objective value $c^\top x^*_{LP}$ of this relaxation, referred to as the *dual bound* at the current node, provides a lower bound on the objective value of any integer-feasible solution in the subtree rooted at v_t. Based on the solution of this relaxation, one of the following cases arises:

- **Pruning by infeasibility.** If P_t admits no fractional solution, no integer-feasible solution exists in the subtree. The node v_t is marked as visited and the branch is pruned by infeasibility.
- **Pruning by bound.** If $x^*_{LP} \in \mathbb{R}^n$ exists, and $GUB < c^\top x^*_{LP}$, no integer solution in the subsequent subtree can improve upon the incumbent. The node v_t is marked as visited and the branch is pruned by bound.
- **Pruning by integrality.** If x^*_{LP} is not dominated by $\bar{x}$, and x^*_{LP} is integer-feasible (all integer variables in $x^*_{LP} \in \mathbb{R}^n$ have integer values), a new incumbent solution $\bar{x} = x^*_{LP}$ has been found. Hence GUB is updated and v_t is marked as visited while the branch is pruned by integrity.
- **Branching.** If none of the above conditions hold, x^*_{LP} admits fractional values for some integer variables. In that case, the branching heuristic $\pi : \mathcal{S} \to \mathcal{I}$ selects a variable x_b with fractional value $\tilde{x}_b$, to partition the solution space. As a result, two child nodes (v_-, v_+) are created, with associated MILPs $P_- = P_t \cup \{x_b \leq \lfloor \tilde{x}_b \rfloor\}$ and $P_+ = P_t \cup \{x_b \geq \lceil \tilde{x}_b \rceil\}$, and added to the current node.[9] Their linear relaxation are solved, before they are added to the set of open nodes $\mathcal{O}$ and v_t is marked as visited.

This process is repeated until $\mathcal{O} = \emptyset$ and $\bar{x}$ is returned. The dynamics of the B&B algorithm between two branching decisions can be described by the function $\kappa_\rho : \mathcal{S} \times \mathcal{I} \to \mathcal{S}$, such that $s' =$

[8] To account for early resolution steps where no incumbent solution has yet been found, we define a special value for $\bar{x}$, whose $GUB = \infty$. For the sake of simplicity, we make this implicit in the remainder of this thesis.

[9] $\tilde{x}_b$ denotes the value of x_b in x^*_{LP}. We use the symbol $\cup$ to denote the refinement of the bound on x_b in P_t.

$\kappa_\rho(s, \pi(s))$. By design, B&B does not terminate before finding an optimal solution and proving its optimality. Consequently, optimizing the performance of B&B on a distribution of MILP instances is equivalent to minimizing the expected solving time of the algorithm.

1.2.3 Learning optimal branching strategies

Since the 1980s, considerable research and engineering effort has gone into refining MILP solvers, resulting in highly optimized systems driven by expert-designed heuristics tuned over large benchmarks [28, 65]. Nevertheless, in operational settings where structurally similar problems are solved repeatedly, adapting solver heuristics to the distribution of encountered MILPs can lead to substantial gains in efficiency, beyond what static, hand-crafted heuristics can offer. Recent research has thus turned to machine learning (ML) to design efficient, data-driven B&B heuristics tailored to specific instance distributions [150].

The variable selection heuristic, or branching heuristic $\pi : \mathcal{S} \to \mathcal{I}$, plays a particularly critical role in B&B overall computational efficiency [4], as it governs the selection of variables along which the search space is recursively split. Empirical evidence corroborates this observation, in particular Etheve [55] shows that optimizing the variable selection policy over a distribution of MILP instances yields substantially greater computational speedups than optimizing the node selection policy.

Additionally, recent theoretical results by Balcan et al. [15] provide formal justification for learning distribution-aware variable selection strategies. Concretely, they show that for any fixed, data-independent branching rule, there exists a distribution of MILP instances on which the expected B&B tree size grows exponentially in the number of variables, even though alternative strategies would yield dramatically smaller trees on the same distribution. These results formalize the intuition that branching performance is inherently distribution-dependent. In a follow-up contribution, Balcan et al. [16] derive sample complexity bounds providing generalization guarantees for learned branching strategies, offering further theoretical support for data-driven branching.

Motivated by these empirical and theoretical insights, the present thesis seeks to leverage machine learning algorithms to automate the discovery of branching strategies which, in the setting of repeated problems, are capable of surpassing the solving time performance achieved by expert-crafted rules implemented in modern MILP solvers. In this context, given a fixed node selection strategy ρ, we

define the target optimal branching strategy π^* associated with a distribution p_0 of MILPs as:

$$\pi^* = \arg\min_{\pi} \mathbb{E}_{P\sim p_0}(|BB_{(\pi,\rho)}(P)|), \tag{1.3}$$

where $|BB_{(\pi,\rho)}(P)|$ denotes the size of the $B\&B$ tree when solving P to optimality following strategies (π, ρ). Following prior works [55, 149], the total number of nodes in the B&B tree is used as an alternative metric to evaluate the performance of learning-based branching agents, as it represents a hardware-independent proxy for computational efficiency.

1.3 Related work

The growing body of work at the intersection of machine learning and combinatorial optimization can be broadly organized along the exact–approximate axis introduced in Section 1.1.2. A first line of work seeks to reinforce exact resolution frameworks by learning improved solver heuristics, while a second proposes to bypass these frameworks entirely, training neural networks to construct feasible solutions directly. We review both directions below, focusing on works orthogonal to the methods later developed in this thesis.

1.3.1 Reinforcing exact solvers: learning in branch-and-bound

A large body of work has proposed to learn better-performing B&B heuristics outside of variable selection. Contributions in cutting plane selection [134, 164, 173], primal heuristics [86, 125, 156] and node selection [79, 103, 186] have all sought to replace expert-designed rules with learned policies, trained either by imitation or by reinforcement.[10]

Cutting planes are generated throughout the B&B search to strengthen LP relaxations by eliminating fractional solutions, thereby raising the dual bound and reducing the number of nodes that need to be explored. Tang et al. [164] first formulated cut selection as a reinforcement learning problem over separation rounds, before Wang et al. [173] introduced a hierarchical sequence model that jointly learns which cuts to select, and in what order to add them to the current MILP formulation, thus addressing cut management challenges that existing heuristics handle independently. On the theoretical side, Balcan et al. [17] provided the first sample complexity bounds for learning cut selection poli-

[10]The reader unfamiliar with the inner workings of B&B solvers may refer to Section 2.1 for a formal introduction to these solver components.

cies in branch-and-cut. Given a training set of integer programs drawn from an application-specific distribution, their guarantees bound the number of samples sufficient to ensure that the empirical B&B tree size under any policy in a given family closely approximates its expected counterpart. In a follow-up contribution, Balcan et al. [18] sharpened these guarantees by conducting a structural analysis of branch-and-cut that characterizes how each algorithmic step responds to perturbations in the parameters defining the added cuts. Together, these results provide a rigorous statistical learning foundation for data-driven cut management, extending to cutting planes the generalization theory previously developed for branching [15, 16] presented in Section 1.2.3.

Primal heuristics seek to discover high-quality feasible solutions early in the search, thereby lowering the primal bound and enabling aggressive pruning of the B&B tree. Looking to enhance this procedure through learning, Nair et al. [125] introduced Neural Diving, a GNN-based framework that generates multiple partial variable assignments and delegates the completion of each associated sub-MILP to the solver, achieving substantial computational speedups on instances with up to 10^5 variables and constraints. Huang et al. [86] followed a complementary approach by learning the destroy heuristic in large neighborhood search: using contrastive learning to distinguish good from bad neighborhood selections, they trained a model to imitate the costly local branching oracle, achieving strong empirical performance on large-scale benchmarks. More broadly, Chmiela et al. [37] showed that the scheduling of primal heuristics itself can be learned, modeling the problem as a multi-armed bandit to dynamically balance execution overhead against primal bound improvement.

On the node selection side, He et al. [79] first proposed to learn an adaptive node searching order via imitation learning (IL), while Labassi et al. [103] proposed training a graph neural network to rank pairs of candidate nodes directly, in order to bypass hand-crafted scoring functions. Finally, at the solver-configuration level, Liberto et al. [108] introduced DASH, a dynamic strategy that switches among a portfolio of branching rules on-the-fly based on features of the evolving search tree. These contributions collectively demonstrate that, while this thesis focuses on learning branching strategies, virtually every heuristic component of a modern B&B solver is amenable to data-driven improvement.

A cross-cutting challenge shared by all learning-based B&B methods is generalization: policies trained on a specific class of instances often degrade on problems of different size or structure. Zarpellon et al. [187] proposed to address this by augmenting variable features with an explicit parameterization of the search tree state, enabling branching policies to generalize across heterogeneous MILP

instances. Lin et al. [109] extended this idea by aggregating search tree features at past branching nodes via a Transformer architecture. However, the performance achieved on heterogeneous families remains substantially below what can be attained on a coherent distribution of instances. Moreover, in the absence of a well-defined instance distribution, the sample complexity guarantees established by Balcan et al. [16] no longer apply, leaving these approaches without formal generalization guarantees. Accordingly, this thesis focuses on learning within a fixed distribution of MILPs rather than across heterogeneous instance families.

1.3.2 Bypassing exact solvers: neural combinatorial optimization

Machine learning applications to CO are not limited to B&B. A parallel and highly active line of research, broadly referred to as neural combinatorial optimization (NCO), proposes to bypass established resolution frameworks entirely and train neural networks to construct feasible solutions directly from problem inputs. In NCO, the combinatorial optimization problem defined in Eq. (1.1) is cast as a finite-horizon Markov decision process, in which a neural policy network builds a solution incrementally through a sequence of discrete decisions. Given a problem instance, an encoder, typically based on attention layers or graph neural networks, produces a latent representation of the input, and an auto-regressive decoder selects one element at a time, for instance the next city in a TSP tour or the next customer in a vehicle routing schedule, until a complete feasible solution is generated.

The modern NCO pipeline originates from Kool et al. [101], who showed that a Transformer-based encoder–decoder architecture, trained end-to-end via reinforcement learning could produce near-optimal solutions for the traveling salesman problem and several vehicle routing variants on instances of up to one hundred nodes. This work established the standard NCO framework: a permutation-equivariant encoder coupled with an auto-regressive pointer-style decoder [170], trained via policy gradient methods using the objective value as reward. Subsequent contributions refined this template along several axes. On the training side, POMO [102] introduced a symmetry-based data augmentation scheme exploiting the invariance of solution quality under cyclic permutations of a tour, yielding lower-variance gradient estimates and faster convergence. Sym-NCO [95] embedded symmetry awareness directly into the training objective, penalizing inconsistencies across equivalent solution representations rather than relying on augmentation at inference alone. On the architectural side, Luo et al. [114] proposed heavier decoder designs to improve generalization to larger instances, while Drakulic et al.

[48] introduced goal-conditioned architectures for multi-objective routing.

Beyond routing, NCO methods have also been applied to scheduling, lot sizing, and various other combinatorial problems amenable to sequential construction. However, a persistent limitation of NCO methods is their difficulty in generalizing beyond the instance sizes and structures encountered during training. Joshi et al. [91] conducted a systematic investigation of this issue, showing that zero-shot generalization to larger instances requires rethinking the entire NCO pipeline, from network layers and learning paradigms to evaluation protocols. To mitigate this challenge, a growing body of work has sought to augment learned policies with search procedures at inference time to compensate for imperfect generalization. Population-based approaches such as Poppy [70] train a diverse set of complementary policies that are rolled out simultaneously at inference, retaining the best solution found across the generated candidate population as the incumbent. Chalumeau et al. [32] extended this paradigm by parameterizing a distribution of diverse, specialized policies conditioned on a continuous latent space, enabling policy adaptation at inference through search in latent space rather than solution space.

Despite these advances, NCO methods currently provide no optimality certificates and remain limited to problem classes for which a natural sequential construction procedure exists. Following the taxonomy introduced in Section 1.1.3, these approaches effectively attempt to learn modeling, framework, and heuristic biases simultaneously from data, without leveraging the structural knowledge brought by decades of research in polyhedral analysis and decomposition theory. In contrast, the present thesis adopts the complementary approach of preserving established resolution frameworks and focusing the learning effort on the heuristic decisions that govern the search process.

1.4 Contributions and overview

The present manuscript is organized in four parts, each composed of a preliminaries chapter introducing the required technical background, followed by a contribution chapter. Since later parts build upon material introduced in earlier ones, the reader is advised to go through chapters sequentially.

Part I: Learning to branch by imitation. The first part explores whether expanding the set of branching candidates available at each B&B node with general branching disjunctions can yield stronger expert demonstrations for learning to branch by imitation.

To that end, Chapter 2 first provides general background on the architecture of modern MILP solvers. Then, it introduces supervised and imitation learning frameworks, along with graph neural networks, the prevailing class of function approximators used for learning in B&B. Brought together, these elements provide the theoretical framework required to present the seminal imitation learning pipeline for learning to replicate expert branching decisions in branch-and-bound [62].

Building on these foundations, Chapter 3 investigates whether expanding the set of branching candidates beyond single variable disjunctions can yield stronger expert demonstrations for imitation learning. Specifically, it introduces *generalized strong branching*, an expert rule that evaluates strong branching scores, the branching oracle from Gasse et al. [62], over Gomory mixed-integer (GMI) disjunctions in addition to standard elementary disjunctions. Our experimental study reveals that, although GMI disjunctions occasionally achieve higher strong branching scores than their single variable counterparts, they do not consistently translate into smaller branch-and-bound trees. We attribute this counterintuitive finding to the interaction between branching and domain propagation modules in modern MILP solvers: single variable branching directly tightens individual variable bounds, which enables cascading bound reductions through propagation modules, an indirect pruning mechanism from which general disjunctions cannot benefit to the same extent. This chapter also highlights a broader phenomenon relevant to IL approaches for learning to branch: the low tree sizes achieved by strong branching are not solely attributable to the quality of its branching decisions, but also to the bound information generated and reused as a byproduct of LP evaluations during strong branching. Crucially, imitation learning policies do not produce this information, resulting in larger B&B trees. These findings nuance the effectiveness of strong branching as a branching oracle in imitation learning, and motivate the transition to reinforcement learning (RL) in the subsequent parts of the thesis.

Part II: Learning to branch by reinforcement. The second part introduces a principled reinforcement learning framework for learning branching policies that directly target B&B tree size minimization. It examines the strengths and limitations of this framework, and investigates the conditions under which RL agents can recover the tree size performance achieved by imitation learning agents trained on strong branching demonstrations.

Chapter 4 introduces necessary background on Markov decision processes (MDP), value-based methods, and policy gradient methods in reinforcement learning. It then discusses the main contemporary challenges facing reinforcement learning, and how these are amplified in combinatorial

optimization settings. Finally, it reviews how existing reinforcement learning approaches for learning to branch, namely TreeMDP [56, 149] and Retro Branching [132], have sought to address these challenges.

Chapter 5 shows that, in order to cope with dire credit assignment issues, prior contributions have shifted away from the traditional Markov decision process framework to model variable selection in B&B, finding it impractical for learning efficient policies. However, by discarding core MDP properties such as temporal sequentiality, the proposed alternative frameworks [56, 132, 149] erode the theoretical guarantees underpinning the asymptotic convergence of generic RL algorithms. Moreover, because these problem formulations rely on approximations of the B&B dynamics, they fail to accommodate the full spectrum of modern RL methods. To address these limitations, Chapter 5 advocates reverting to a principled MDP formulation, termed branch-and-bound Markov decision process (BBMDP). Crucially, BBMDP leverages the results first established by Etheve et al. [56] and Scavuzzo et al. [149] — under a depth-first search (DFS) node selection policy, minimizing the whole B&B tree size is achieved when any subtree is of minimal size — all while preserving MDP properties. Hence, BBMDP allows to harness RL algorithms that are not compatible with the previous frameworks, such as multi-step temporal difference, eligibility traces, or any RL algorithms implementing planning-based policy improvement, for the purpose of learning optimal variable selection strategies.

Additionally, taking inspiration from recent works in RL advocating training value networks via classification instead of regression to foster both scalability and generalization [41, 57], Chapter 5 introduces a histogram classification loss tailored to the branch-and-bound setting. This classification loss is then used for the training of value-based RL agents. The resulting DQN-BBMDP agent achieves state-of-the-art performance among reinforcement learning approaches on the standard Ecole benchmark [139], comprising set covering, combinatorial auction, multiple knapsack, and maximum independent set instance generators, while narrowing the gap with imitation learning baselines.

While these improvements are encouraging, they remain unsatisfying from an RL standpoint: RL agents should in principle be able to match, and even surpass, the tree size performance of imitation learning, which is inherently bounded by that of the suboptimal expert it imitates. In order to investigate this gap, Chapter 5 examines the role of reward sparsity induced by constant rewards in BBMDP. Specifically, we design an alternative reward model directly targeting dual gap reduction, closely mirroring the objective optimized by strong branching, and evaluate whether this denser signal

enables RL agents to close the remaining gap with IL. Contrary to our initial hypothesis, agents trained under these denser, strong branching-aligned rewards also fail to recover the performance of imitation learning baselines, indicating that the performance gap observed between RL and IL stems from other issues inherent to deep reinforcement learning, rather than from the choice of a sparse reward signal. Interestingly, we observe that RL agents trained according to both constant and strong branching-like reward models exhibit a comparable degree of alignment with strong branching behaviour, slightly lagging that of IL baselines. These results prompt us to identify insufficient exploration as the primary factor preventing RL agents from converging towards optimal branching strategies, a limitation that Part III proposes to address explicitly.[11]

Part III: Planning in branch-and-bound. Inspired by the landmark success of AlphaGo [153] and its offspring in complex combinatorial games [152, 154], Part III proposes to mitigate the exploration issues identified in Chapter 5 by leveraging model-based planning to ensure robust policy improvement.

Chapter 6 provides technical background on model-based reinforcement learning, covering standard planning procedures (Monte Carlo tree search, Gumbel search) and advanced MBRL frameworks (AlphaZero, MuZero), before reviewing existing attempts to integrate planning algorithms within learning-based approaches in combinatorial optimization.

Chapter 7 introduces Plan-and-Branch-and-Bound (PlanB&B), a model-based reinforcement learning agent that leverages an internal model of the B&B dynamics to learn improved variable selection strategies. Crucially, our agent operates as a hybrid between AlphaZero and MuZero: the branching component is approximated by a learned dynamics network, while the node selection policy and rewards are simulated exactly. To the best of our knowledge, this is the first MBRL agent specifically designed to solve CO problems without assuming access to a cheap, accurate simulator. PlanB&B achieves state-of-the-art performance across the Ecole benchmark, surpassing the solver's default branching strategy as well as all prior IL and RL baselines on test instances. Notably, alignment analysis reveals that PlanB&B does not achieve these gains by imitating strong branching more closely, but by discovering original branching strategies that actively diverge from the expert while yielding smaller B&B trees.

To assess the practical relevance of PlanB&B beyond synthetic benchmarks, we apply it to real-

[11]The core material presented in this chapter has led to a publication in the *Proceeding of the 36th Conference on Advances in Neural Information Processing Systems* [160], while the reward sparsity study is original to this thesis.

world hydroelectric scheduling instances arising from concrete EDF industrial use cases. On these instances, PlanB&B consistently outperforms all prior learning baselines, with performance gaps widening on out-of-distribution longer-horizon instances. Again, alignment metrics reveal that PlanB&B outperforms IL baselines not through closer strong branching imitation, but by actively departing from strong branching behaviour. Furthermore, progressively fine-tuning PlanB&B on production planning instances of increasing horizon enables accelerating the resolution of week-long scheduling problems, suggesting that our method can translate access to additional data and compute into consistent performance gains on higher-dimensional target instances.

Despite these results, the study of PlanB&B in its current form reveals two main limitations. First, the use of DFS node selection policy imposes a growing computational burden as problem dimensionality increases: on higher-dimensional MILPs, closing the primal gap becomes increasingly difficult, which in turn inflates the number of nodes processed by DFS. Addressing this limitation likely requires moving beyond the MILP bipartite graph representation introduced by Gasse et al. [62] towards more expressive observation functions capable of encoding the full evolving B&B tree, as recently proposed by Zhang et al. [189]. Second, while PlanB&B clearly outperforms all prior learning baselines on EDF hydro instances, its solving time performance only matches that of the solver's default heuristic on test instances, and falls significantly behind it on higher-dimensional out-of-distribution instances. These results may reflect a discrepancy between variable selection as modeled in BBMDP and as implemented in practice by the solvers: while our agent learns only to select a variable for branching at each node, solvers' branching modules also solve additional LP relaxations to derive stronger bounds and prune the B&B search tree more aggressively. This endows solver heuristics with a form of long-term planning ability that the BBMDP formulation, in its current form, does not grant to its branching agents. In other words, while IL and RL agents currently optimize branching alone, solver heuristic modules effectively optimize branching **and** bounding jointly. We argue that, for learned policies to reliably surpass solver heuristics in the future, they must start playing the same game, and be given the same options as their expert counterparts. While such a transition raises practical challenges tied to the deeper integration of learning agents into the core of MILP solver implementation, BBMDP can, in theory, be straightforwardly adapted to level the playing field between learned and hard-coded branching heuristics. This adapted formulation, termed ProbBBMDP, is presented in our interim conclusion on learning to branch in Chapter 8. We highlight the theoretical modifications

brought to the BBMDP framework and discuss the implementation challenges that remain for the training of such "general" branching agents.[12]

Part IV: Flow matching for conic optimization. The interim conclusion presented in Chapter 8 identifies LP resolution as the computational backbone of branch-and-bound. This observation is grounded in a structural property of branch-and-bound, discussed in Section 1.1.3: LP relaxations consistently provide tight dual bounds that enable pruning large portions of the search space, thereby avoiding exhaustive enumeration of the feasible integer solutions. Consequently, nearly every component of a modern MILP solver stands to benefit directly from faster LP resolution. Furthermore, LP relaxations solved throughout the B&B search tree differ only by local bound modifications and formulation strengthening introduced by branching and separation routines, forming a family of closely related problems well suited to the repeated optimization setting considered in this thesis. Accelerating the resolution of LP relaxations, whether exactly through warm-starting or approximately through learned surrogates, therefore represents a high-leverage yet largely unexplored direction for improving MILP solving. Motivated by this observation, Part IV departs from the variable selection problem to explore a complementary research direction: leveraging recent advances in generative modeling to learn continuous-time surrogate models of LP solvers. The resulting framework is designed to serve a dual purpose: producing fast approximate LP solutions when unrolled to completion, and providing high-quality initial points to warm-start exact solvers when unrolled partially.

Chapter 9 introduces the necessary technical background. It first introduces primal–dual interior-point methods (IPMs) for linear programming [129], emphasizing their dynamical structure: IPM iterates trace a smooth trajectory through the primal–dual state space, driven by a barrier parameter that continuously interpolates between a well-conditioned interior point and an optimal solution. This trajectory structure motivates the key idea underlying Part IV: rather than learning a direct map from LP data to an optimal solution, one can learn a continuous-time vector field whose integral curves approximate solver trajectories. Chapter 9 then reviews recent results by Qian et al. [141] showing that message-passing neural networks can be trained to simulate individual IPM iterations, supporting the premise that learning solver dynamics is more natural than learning end-to-end solution maps. Finally, it introduces flow matching [111], a generative modeling framework for learning continuous-time vector

[12]The core material presented in this chapter has been published in the *Proceedings of the 40th Annual AAAI Conference on Artificial Intelligence* [161]. The industrial application, however, is original to this thesis.

fields that transport a source distribution to a target distribution via local velocity regression. Together, these elements provide the theoretical and methodological basis for learning continuous-time LP surrogates: interior-point methods supply the structured trajectories that serve as supervision, while flow matching provides the training objective that turns these trajectories into a learned vector field.

Chapter 10 introduces Flow-IPM, a framework that adapts flow matching to approximate interior-point dynamics for linear programming. For each training LP instance, we generate a bundle of IPM trajectories by perturbing the solver's initialization and re-solving from each perturbed starting point, producing trajectories that all converge to the same optimal solution but traverse different regions of the primal–dual space. A velocity field is then trained to regress against empirical velocities extracted from these paths. At inference time, integrating the learned vector field via an ordinary differential equation (ODE) solver produces approximate primal–dual solutions, usable either as fast LP estimates or as warm-start points for an exact solver. To isolate the benefit of using solver-generated trajectories as supervision, we compare the IPM-based flow against a simpler baseline that trains on straight-line interpolations between random initializations and optimal solutions, without any solver supervision. Our experimental study on combinatorial auction LP relaxations reveals that both models achieve relative objective gaps below 1%, validating the rationale for our approach. Warm-starting experiments confirm measurable iteration savings, though limited by trajectory drift and the geometric difficulty inherent to IPM warm-starting: a good starting point must be simultaneously near-optimal and well-centered with respect to the central path, two requirements that are fundamentally in tension. These findings motivate a shift toward a different class of solvers, namely proximal methods, which solve LPs through iterative projections rather than by tracing a path through the interior of the feasible region. For such methods, warm-starting requires only proximity to the optimal solution, with no additional geometric constraints. Notably, the straight-line flow model already developed constitutes an actionable framework for warm-starting proximal solvers.

We stress that this part of the thesis is exploratory in nature: it opens a novel line of research at the intersection of generative modeling and mathematical programming, and provides initial experimental evidence suggesting that the approach could be viable in specific settings. However, a definitive assessment of its practical relevance requires scaling to industrially relevant LP dimensions and adapting the framework to proximal solver dynamics, both of which are left to future work.

Part I

Learning to branch by imitation

Chapter 2

Preliminaries

Content

The present chapter introduces conceptual material that will recur throughout this manuscript. Before introducing supervised learning and neural networks, along with their recent application to mixed-integer linear programming, we begin by providing contextual background on the algorithmic framework underlying modern MILP solvers.

2.1 Anatomy of modern mixed-integer linear programming solvers

As highlighted in Section 1.2, the overarching goal in branch-and-bound is to close the gap between the primal and dual bounds in as few steps as possible. In a minimization problem, this entails two complementary (oftentimes competing) objectives: on the one hand, raising the dual bound through tighter relaxations, and on the other hand, lowering the primal bound by discovering high-quality feasible solutions. Both aspects must progress in tandem for the solver to achieve strong performance.

The main goal of this thesis is to accelerate the resolution of mixed-integer linear programs by branch-and-bound by designing variable selection strategies yielding search trees of minimal size, see Eq. (1.3). Because this objective involves intricate primal-dual trade-offs, in practice, branching closely works together with primal heuristics, node selection and cutting planes generation modules to produce B&B trees of minimal size. Since variable selection cannot be studied in complete isolation from the rest of the B&B solving process, the following section provides a general overview of how these different modules operate and interact to efficiently guide the search process.

Importantly, throughout this thesis, we mainly rely on the *SCIP Optimization Suite*[1] as our reference framework for mixed-integer linear programming solving. While commercial solvers such as Gurobi or FICO Xpress typically define the current state-of-the-art in terms of raw performance, SCIP stands out as one of the most popular open-source alternatives. Accordingly, while the algorithmic principles presented in this section broadly apply to mixed-integer linear programming solvers, their exposition and emphasis are naturally influenced by the structure and design choices of the SCIP solver.

[1] `https://scipopt.org/`

2.1.1 LP relaxation

Although often discussed only briefly in general introductions to branch-and-bound, linear programming (LP) relaxations play a central role in approximating the integer hull of mixed-integer linear programs and, consequently, in the efficiency of contemporary MILP solvers.

Given a generic MILP as defined in Eq. (1.2), its associated LP relaxation is obtained by replacing the integrality constraints $x_j \in \mathbb{Z}$ with continuous constraints $x_j \in \mathbb{R}$ for all $j \in \mathcal{I}$, and solving the resulting linear program. This relaxation serves two fundamental purposes: first, it provides a valid dual bound $c^\top x^*_{LP}$, second, it yields a fractional solution x^*_{LP}. Both pieces of information are extensively exploited by the other solver components introduced in the following sections.

Crucially, in mixed-integer linear programming, solving LP relaxations remains by far the most computationally intensive task repeatedly performed by the solver, and accounts for the bulk of the overall computational cost. As a result, in modern solvers, LP relaxation resolution relies on carefully engineered strategies, that distinguish between the root node and nodes encountered deeper in the search tree.

At the root node v_0, the solver faces a "cold start" scenario with no prior basis information. Because the root LP is typically the most computationally expensive linear program encountered during the search and determines the initial quality of the global dual bound, solvers often employ concurrent optimization. This involves launching multiple algorithms in parallel threads, typically primal simplex, dual simplex, and barrier (interior point) methods. The process terminates as soon as the first algorithm converges. Notably, if the barrier method wins, a crossover procedure is required to transition from the analytic center solution to an optimal extreme point, thereby establishing a valid basis matrix necessary for initiating the branch-and-bound tree.

Once the search enters the tree, the solving strategy shifts almost exclusively to the dual simplex algorithm. This design choice is driven by the structural nature of branching. When a parent node is branched on, new constraints (branching bounds or cutting planes) are added to the formulation. Geometrically, these additions render the parent's LP optimal solution primal infeasible (as it violates the new restrictions) but leave the dual solution feasible, since no dual constraints were added or removed. The dual simplex algorithm is mathematically designed for this specific state: it iterates from a dual-feasible, primal-infeasible basis toward primal feasibility. Consequently, re-optimizing a child

node typically requires only a small number of pivots making it significantly faster than restarting the optimization from scratch. On top of computational efficiency, the simplex method provides a crucial by-product to accelerate the resolution of MILPs: the *simplex tableau*. Associated with the optimal basis, the tableau expresses each basic variable as an affine combinations of the non-basic variables. As discussed in Section 2.1.5, this algebraic structure is very useful for separation routines, because efficient cutting planes, like Gomory Mixed-Integer cuts, can be derived directly from the rows of the simplex tableau.

Taken together, these considerations highlight that, to first order, the computational burden of branch-and-bound is proportional to the total number of LP relaxations solved, and more specifically to the total number of simplex iterations performed throughout the solving process. Consequently, minimizing this quantity constitutes the primary objective implicitly pursued by essentially every inner branch-and-bound heuristic, including those introduced in the following sections.

2.1.2 Primal heuristics

Primal heuristics are dedicated procedures that search for high-quality feasible solutions throughout the B&B process. Their role is crucial: a good incumbent not only certifies progress toward optimality but also tightens the primal bound, enabling large portions of the tree to be pruned by bound.

Primal heuristics exploit information generated by the LP relaxation, search tree expansion, and domain propagation to construct feasible assignments with minimal associated computational overhead. They are deliberately diverse, reflecting the broad range of structures encountered in MILPs. Broadly, they can be grouped into several distinct families. Relaxation-based heuristics derive feasible solutions by rounding or repairing the fractional LP solution. Propagation and domain-based heuristics leverage variable fixings, reduced-cost information, or local domain consistency to quickly identify feasible points. Local improvement heuristics, such as local branching, search in a neighborhood around the current incumbent, combining intensification and diversification strategies.

Because no single heuristic is effective across all problems, modern solvers deploy a large portfolio of complementary methods, each activated under specific conditions. These heuristics are integrated throughout the search: some run exclusively at the root node, others are triggered periodically or when certain structural patterns arise, and many are opportunistic, reacting to local changes in variable domains or node structure. This opportunistic design ensures that primal improvements can be found

early, enabling aggressive pruning, and that further refinements can be made as the search progresses.

2.1.3 Variable selection

Both in our general introduction, as well as in the introduction of the present section, we have defined "good" variable selection strategies as branching rules yielding small B&B trees. However minimizing B&B tree size is a long, often very long-term objective; it hardly translates into a direct, practical selection criterion for discriminating among branching candidates at a given node. Traditional variable selection strategies therefore rely on tractable, short-term performance indicators to assess the quality of branching decisions. For instance, efficient branching decisions are commonly associated with a significant reduction in the primal-dual gap, making it a natural proxy for assessing the quality of branching decisions. In practice, because branching has little short-term measurable influence on the primal bound, but directly affects the dual bound, and because primal heuristics are highly effective in practice, the current prevailing view in the literature is that branching should primarily focus on tightening the dual bound, while primal bound improvement is best delegated to dedicated (primal) heuristics.

Following this principle, a wide range of branching rules has been proposed. Among them, *strong branching* (SB) [10] stands out as one of the most powerful and effective, albeit computationally expensive, techniques. The core idea of SB is to simulate the effect of branching on several candidate variables before actually committing to a decision. This is achieved by tentatively imposing branching disjunctions associated with each candidate variable (e.g., $x_i \leq \lfloor \hat{x}_i \rfloor$ and $x_i \geq \lceil \hat{x}_i \rceil$ for fractional variable x_i) and solving the corresponding child nodes' LP relaxations to obtain dual bounds. The SB score of a variable is typically based on the estimated dual bound improvement, that is, how much the LP relaxation's objective increases under each branching direction. Intuitively, a variable is promising for branching if both child nodes yield significantly higher associated dual bounds $c^\top x^*_{LP}$ than the current node, indicating that branching on this variable may help prune the search tree more effectively. Mathematically, for a candidate variable x_i, let Δ^i_l and Δ^i_r denote the increases in the LP relaxation objective value under the left and right branch respectively. Typical SB scoring function [2] write

$$\text{score}(x_i) = \sqrt{\Delta^i_l \cdot \Delta^i_r} \quad \text{or} \quad \text{score}(x_i) = \alpha \cdot \max\{\Delta^i_l, \Delta^i_r\} + (1-\alpha) \cdot \min\{\Delta^i_l, \Delta^i_r\}, \tag{2.1}$$

with $\alpha \in [0, 1]$. Such scoring schemes aims to identify variables that exhibit balanced improvement in

both branches, thus maximizing the likelihood of early node fathoming.

Despite its effectiveness in reducing the size of generated B&B trees, the primary drawback of strong branching is its computational overhead. At each node, it requires solving potentially hundreds of LPs, one for each branching direction of every fractional integer variable, which quickly becomes prohibitive even for medium-scale MILPs. This limitation has motivated the development of more computationally efficient branching strategies, designed to retain the effectiveness of strong branching while operating at a fraction of its cost. One of the most influential among these is *pseudo-cost branching* (PC) [30], which replaces explicit LP probing with online estimates of dual bound changes. Instead of solving auxiliary LPs at each node, the solver maintains historical statistics that record how the objective value evolved when a variable was branched on in past nodes. Concretely, for each integer variable x_i, the solver maintains two pseudo-costs psc_l^i and psc_r^i, typically defined as ratios of observed dual bound improvements to branching fractionalities:

$$\mathrm{psc}_l^i \approx \frac{\Delta_l^i}{|x_i - \lfloor x_i \rfloor|}, \qquad \mathrm{psc}_r^i \approx \frac{\Delta_r^i}{|\lceil x_i \rceil - x_i|}.$$

When revisiting the same variable at a future fractional node, the solver predicts the hypothetical bound improvements under each branch by scaling these pseudo-costs by the variable's current fractional distance:

$$\widehat{\Delta}_l^{\,i} = \mathrm{psc}_l^i \cdot |x_i - \lfloor x_i \rfloor|, \qquad \widehat{\Delta}_r^{\,i} = \mathrm{psc}_r^i \cdot |\lceil x_i \rceil - x_i|.$$

These predicted gains serve as surrogates for the dual bound improvements that strong branching would have obtained, and are typically aggregated using the same scoring functions as in SB.

A well-known limitation of pseudo-costs, however, is that their accuracy depends on having collected sufficiently many past observations. Early in the search, pseudo-costs are unreliable, and poor estimates can induce erratic branching decisions[2]. This is especially damaging, as suboptimal branching near the root tends to inflate the search tree dramatically and thus degrade total solving time. This issue has motivated the development of *reliability branching* [5], a hybrid strategy that bridges the gap between the robustness of strong branching and the long-term efficiency of pseudo-costs. In reliability branching, each variable x_i is associated with a reliability threshold k_{rel}. While x_i has been evaluated fewer than k_{rel} times, the solver performs strong branching on x_i to collect high-quality pseudo-cost samples. Once the threshold is met, the solver switches to pseudo-cost predictions for x_i and refrains

[2]In vanilla pseudo-cost branching, all $(\mathrm{psc}_l^i, \mathrm{psc}_l^i)_{i \in \mathcal{I}}$ are initialized to 1 at the root.

from further probing. This approach inherits SB's accuracy during the "exploration" phase, and PC's efficiency thereafter. Additional refinements were brought from hybrid [3] and lookahead-based [5] scoring rules, that integrate structural problem features with pseudo-cost information. These heuristics typically serve as prefilters: they narrow the candidate set to a few promising variables before applying more expensive scoring functions, thereby reducing the cost of branching while still guiding the search toward strong decisions.

Taken together, these methods illustrate a broader evolution of variable selection strategies in branch-and-bound: from exact but expensive probing (strong branching), to online learning-based surrogate approximation (pseudo-cost branching), to adaptive hybrids (reliability branching) that balance information quality with computational efficiency. Nowadays, modern B&B solvers broadly adhere to the reliability branching paradigm, but augment it with numerous solver-specific subcases triggered by carefully calibrated thresholds. While these additions often deliver strong empirical performance, they also make the resulting branching logic increasingly intricate and difficult to interpret[3].

Interestingly, strong branching still represents a gold standard for variable selection in B&B, as it produces the best known expert branching decisions. Its study continues to inform both theoretical analyses of B&B efficiency and the design of advanced branching heuristics, including those based on machine learning, as we will soon discuss. Nevertheless, despite their strong empirical tree size performance, strong branching decisions are known to be suboptimal in the general case. In fact, on small instances for which optimal B&B trees can be computed by exhaustive enumeration, Dey et al. [46] finds that strong branching typically achieves B&B tree size within a factor two of the minimum achievable, across various problem classes. Furthermore, they also exhibit pathological instance families for which strong branching yield trees that are exponentially larger than optimal, echoing earlier findings of Balcan et al. [16]. These results call for deeper theoretical investigation into the structure of optimal branching decisions, as well as further experimental work aimed at identifying branching rules outperforming strong branching on specific classes of instances. The work presented in Part II and Part III directly addresses this goal, as it seeks to automate the discovery of variable selection strategies surpassing the performance achieved by strong branching over a class of MILP instances.

[3] `https://scipopt.org/doc/html/branch__relpscost_8c_source.php`

2.1.4 Node selection

The role of the node selection heuristic is to choose, at each iteration, among all open nodes $o \in \mathcal{O}$, the one whose exploration is expected to most rapidly improve the global dual bound or identify high-quality incumbent solutions. Node selection thus actively participates in controlling the shape of the search tree, balancing diversification and intensification, and ultimately minimizing the number of total processed nodes.

A classical strategy is *best-first search*, also known as *best-bound*. At each iteration, the solver selects the open node with the most promising LP relaxation bound $c^T x^*_{LP}$, typically the smallest objective value in minimization problems. This strategy is theoretically appealing because it maintains the strongest possible global dual bound at all times and aggressively explores regions with high potential for pruning. However, best-first search tends to produce very broad trees, as it postpones committing to any branch until all nodes with comparable bounds have been processed. This behavior results in a large number of open nodes being kept in memory simultaneously, which can be prohibitive for large-scale MILPs.

At the opposite extreme lies *depth-first search*. DFS always expands the most recently created node, diving along a branch until reaching either an integer-feasible solution or a dead end (infeasibility or bound pruning). This strategy offers significant computational advantages, primarily due to its minimal memory footprint and high processing throughput. The latter stems from the strong structural continuity along a DFS path: consecutive nodes differ only by small local modifications, allowing LP reoptimizations to fully exploit dual simplex warm starts and basis reuse. In contrast, strategies that jump between disparate branches incur a significant context switching penalty, often necessitating a costly refactorization of the basis matrix. The depth-first nature of DFS also helps improve the primal bound by quickly uncovering incumbent solutions. However, this behaviour is also associated with slow dual improvement, resulting in limited pruning power. Pure DFS is therefore less efficient on instances where tight dual bounds are essential for reducing the search tree.

Modern B&B solvers blend the advantages of best-bound and DFS to obtain more robust performance. A widely used approach is *best-estimate search*, implemented for example in SCIP. Each B&B

node is assigned an estimated objective value,

$$\widehat{z}(v) = z_{\mathrm{LP}}(v) + \sum_{i \in F(v)} \sqrt{\widehat{\Delta}_l^i \cdot \widehat{\Delta}_r^i},$$

where $F(v)$ is the set of fractional variables at node v, and $\widehat{\Delta}_l^i, \widehat{\Delta}_r^i$ are pseudo-cost-based predictions of the objective degradation under branching. This score effectively interpolates between DFS (favoring nodes with fewer fractional variables) and best-bound (favoring nodes with strong LP relaxations).

Of course, node selection interacts tightly with primal heuristics. When a new incumbent solution is found, all open nodes whose LP bound exceeds the incumbent objective can be immediately pruned, dramatically shrinking the search frontier. Moreover, many solvers also incorporate diving heuristics, which enforce DFS-like dives while a node remains "promising" according to a scoring criterion. These diving subroutines accelerate the discovery of good primal solutions and can reduce B&B tree size significantly.

Overall, node selection must balance several competing objectives: rapid improvement of dual bounds (best-bound behavior), discovery of strong incumbents (DFS-like behavior), control of tree shape and search diversification, while maintaining memory efficiency (limiting the size of the open-node list). Importantly, node selection directly influences variable selection, since the quality of both pseudo-cost estimates and current incumbent depend on the order in which nodes are processed.

2.1.5 Cutting Planes

Cutting plane generation is a central component in modern mixed-integer programming solvers. A cutting plane (or cut) is a valid inequality that is satisfied by all feasible integer points but violated by the current fractional LP solution. Ideal cuts eliminates the fractional incumbent while significantly tightening the relaxation and improving the dual bound. Hence, while branching refines the search space combinatorially, cutting planes strengthen the LP relaxation by removing fractional solutions without excluding any feasible integer point. Because cuts are generated dynamically throughout the search, modern solvers maintain a large arsenal of cut-generating procedures, including Gomory mixed-integer (GMI) cuts, mixed-integer rounding (MIR) cuts, lift-and-project cuts, knapsack cover cuts, clique cuts, cover cuts etc., in order to accommodate the diversity of structures encountered in mixed-integer linear programming.

Since potentially many valid cuts may be generated at each node, the solver must decide which

ones add to the model. This selection step is critical for overall performance and relies on several considerations. First, efficacy. A cut must exhibit significant violation at the current fractional solution (hence significant formulation strengthening) to justify the computational cost of adding it and re-optimizing the LP. Second, orthogonality. Cuts that are too similar provide little additional strengthening and may even degrade numerical stability. Solvers favor cuts that provide complementary information. Finally, numerical robustness is an important governing factor, as cuts with large coefficients or poor scaling can harm LP conditioning. Modern cut management includes filtering steps to avoid numerically unstable inequalities.

In practice, cut selection aims to maximize dual bound improvement per unit of time while avoiding numerical deterioration, much like primal heuristics aim to maximize primal bound improvement while limiting the associated computational overhead. Gomory Mixed-Integer (GMI) cuts [14, 66] are among the most critical and widely used cutting planes in practice, and will be of central importance in Chapter 3. These cuts originate from the simplex tableau associated with the current LP relaxation and exploit the fractional part of an integer basic variable's value. Consider a simplex tableau row

$$x_i = \tilde{x}_i - \sum_{j \notin B} \tilde{a}_{ij} x_j, \qquad i \in B,$$

where x_i is an integer basic variable with fractional value $\tilde{x}_i \notin \mathbb{Z}$, B is the set of basic variables, and x_j are non-basic variables. Since x_i must be integral, any feasible integer point satisfies the disjunction

$$x_i \leq \lfloor \tilde{x}_i \rfloor \quad \vee \quad x_i \geq \lfloor \tilde{x}_i \rfloor + 1.$$

Intersecting this disjunction with the cone defined by the simplex tableau produces the GMI inequality. In normalized form, letting $f_0 = \tilde{x}_i - \lfloor \tilde{x}_i \rfloor$ and $f_j = \tilde{a}_{ij} - \lfloor \tilde{a}_{ij} \rfloor$, the corresponding strengthened GMI cut writes:

$$\sum_{j \in \mathcal{I},\, f_j \leq f_0} \frac{f_j}{f_0} x_j + \sum_{j \in \mathcal{I},\, f_j > f_0} \frac{1 - f_j}{1 - f_0} x_j + \sum_{j \notin \mathcal{I},\, \tilde{a}_{ij} \leq f_0} \frac{\tilde{a}_{ij}}{f_0} x_j + \sum_{j \notin \mathcal{I},\, \tilde{a}_{ij} > f_0} \frac{\tilde{a}_{ij}}{1 - f_0} x_j \;\geq\; 1. \tag{2.2}$$

GMI cuts possess several desirable properties. They dominate all intersection cuts derived from single-row integer disjunctions using the same basis [39]. They apply uniformly to integer and continuous variables. They are computationally inexpensive to compute, relying only on tableau coefficients and exhibit robust numerical behavior when properly normalized. Because of their generality, strength, and low cost, GMI cuts are ubiquitous in both academic and commercial MILP solvers and play a central role in closing a large fraction of the root node relaxation gap.

In modern B&B solvers, cutting planes and branching decisions interact synergistically. Cutting planes strengthen the relaxation and thus improve the dual bound, increasing opportunities for pruning. Branching, in turn, exposes new fractional structures that can be exploited by further cut generation. Chapter 3 investigates in greater depth the connections between branching and cut generation, and how both paradigms can help reinforce one another.

2.2 Supervised Learning

As shown in the previous section, solving mixed-integer linear programs via branch-and-bound involves (repeatedly) taking series of decisions of heterogeneous nature throughout the solving process. A common feature among these decisions is that they must be taken under strict computational budget, since minimizing the total solving time remains the ultimate objective. In order to make the most informed decisions possible, B&B heuristics thus leverage prior knowledge of combinatorial structures together with online statistical estimates of key indicators to guide the decision-making, while selectively allocating additional computational budget to higher-impact decisions. However, in repeated settings such as the one considered in this thesis, it appears natural to seek to achieve further improvements by exploiting patterns present in the underlying distribution of problem instances. This observation naturally motivates the use of data-driven approaches, and in particular supervised learning, to extract such patterns and inform decision-making within the branch-and-bound process. In this section, we introduce the statistical foundations of supervised learning and discuss its extension to sequential decision-making through imitation learning.

2.2.1 Statistical Foundations

Supervised learning is a core paradigm in machine learning concerned with estimating a functional relationship between inputs and outputs from a finite collection of observed samples. Given paired observations of two random variables, the goal is to infer the underlying input–output map, often referred to as the *predictor*. Classical supervised learning problems fall into two broad categories: regression, when the output variable is continuous, and classification, when it takes discrete values. Let X denote the input variable (the *features*) and Y the output variable (the *labels*), with realizations taking values in measurable spaces $\mathcal{X}$ and $\mathcal{Y}$, respectively. Their joint distribution is denoted $\mathcal{D} : \mathcal{X} \times \mathcal{Y} \to [0, 1]$. To evaluate a predictor $f : \mathcal{X} \to \mathcal{Y}$ chosen from a hypothesis class $\mathcal{F}$, we introduce

a *loss function* $l : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ that quantifies the discrepancy between predicted and true labels. In regression, a common choice is the squared Euclidean loss $l(y_1, y_2) = \|y_1 - y_2\|_2^2$. In classification settings, the standard alternative is the cross-entropy loss. For a one-hot label[4] y and a probabilistic prediction $\hat{p} = f(X)$ supported on the same K classes it is defined as $l(y, \hat{p}) = -\sum_{k=1}^{K} y_k \log \hat{p}_k$ and encourages the predictor to assign high probability to the correct class.

Given a loss function and an abstract data-generating distribution $\mathcal{D}$, the general performance of a predictor is measured through its *true risk*,

$$R(f) = \mathbb{E}_{(X,Y)\sim\mathcal{D}} \left[l(Y, f(X)) \right].$$

The optimal predictor within the hypothesis class is therefore

$$f^* \in \arg\min_{f\in\mathcal{F}} R(f).$$

In practice, however, the distribution $\mathcal{D}$ is unknown, and we only have access to a finite dataset of observations $(X_i, Y_i)_{i=1}^n$, which we assume to be drawn independently from $\mathcal{D}$. This naturally motivates the introduction of an empirical counterpart to the above objective. Thus, the *empirical risk* of a predictor f is defined as

$$\hat{R}_n(f) = \frac{1}{n} \sum_{i=1}^{n} l(Y_i, f(X_i)),$$

and the empirical risk minimizer is denoted

$$\hat{f}_n \in \arg\min_{f\in\mathcal{F}} \hat{R}_n(f).$$

The strong law of large numbers ensures that, under mild integrability assumptions, the empirical risk converges almost surely to the true risk as $n \to \infty$,

$$\hat{R}_n(f) \xrightarrow[n\to\infty]{a.s.} R(f),$$

which provides a theoretical justification for empirical risk minimization.

The central challenge in supervised learning is therefore to control the gap between the true risk of the learned predictor and the minimal achievable true risk within a universal, unconstrained function space $\mathcal{G} \supset \mathcal{F}$. This gap is captured by the *excess risk*,

$$\epsilon(\hat{f}_n) = R(\hat{f}_n) - \inf_{f\in\mathcal{G}} R(f).$$

[4]One-hot encoding represents a categorical target as a binary vector where the index corresponding to the true class is 1 and all other indices are 0.

Interestingly, the excess risk decomposes exactly into the approximation error and the estimation error:

$$\epsilon(\hat{f}_n) = \underbrace{\inf_{f\in\mathcal{F}} R(f) - \inf_{f\in\mathcal{G}} R(f)}_{\text{approximation error}} + \underbrace{R(\hat{f}_n) - \inf_{f\in\mathcal{F}} R(f)}_{\text{estimation error}}.$$

While the approximation error is a fixed property of the chosen hypothesis class, the estimation error is driven by the finite training sample. The degree to which any specific predictor overfits this sample is quantified by its associated *generalization error*, $|\hat{R}_n(f) - R(f)|$. A classical uniform convergence argument bounds the estimation error by the worst-case generalization error across the entire hypothesis class, yielding a theoretical surrogate for the bias–variance trade-off:

$$\epsilon(\hat{f}_n) \leq \underbrace{\inf_{f\in\mathcal{F}} R(f) - \inf_{f\in\mathcal{G}} R(f)}_{\text{bias}} + \underbrace{2 \sup_{f\in\mathcal{F}} |\, \hat{R}_n(f) - R(f)\,|}_{\text{variance}},$$

highlighting that enlarging the hypothesis class reduces the approximation bias but typically increases the variance by inflating the worst-case generalization error. This inflation is the mathematical signature of *overfitting*: a richer hypothesis class inherently contains more functions capable of memorizing the finite training data, widening the potential gap between empirical optimism and true performance. Balancing these two effects is a central theme in model design and regularization.

2.2.2 Imitation Learning

This thesis mostly deals with sequential decision-making problems, in which agents must learn how to act over time in dynamic environments. In many of these settings, it can be more practical to learn from expert demonstrations than from explicit labels or reward signals. *Imitation learning* (IL) [130] addresses this situation by seeking a policy that replicates the behaviour of an expert agent, typically assumed to be competent or near-optimal. Formally, let π_E denote the expert policy, generating trajectories $\tau = (s_0, a_0, \ldots, s_T, a_T)$[5] under the underlying system dynamics. The goal of imitation learning is to construct a policy π_θ that matches the expert's behaviour as closely as possible, either in terms of its action distribution or of the induced state distribution.

The most direct instantiation of IL is *behavioral cloning* (BC), which frames the problem as a supervised learning task over state–action pairs sampled from expert trajectories. Given an expert

[5]We refer the reader to Chapter 4 for a detailed introduction to Markov decision processes in sequential decision-making.

dataset

$$\mathcal{D}_E = \{(s_i, a_i)\}_{i=1}^n \sim \pi_E,$$

the learner minimizes an empirical imitation loss of the form

$$\hat{L}(\pi_\theta) = \frac{1}{n} \sum_{i=1}^{n} \ell\big(a_i, \pi_\theta(s_i)\big),$$

where ℓ is typically a cross-entropy loss for discrete actions or a squared error for continuous actions. Behavioral cloning therefore amounts to learning a policy that directly maps observed states to the expert's demonstrated actions.

While conceptually simple and often efficient in practice, BC suffers from a well-known limitation: the learner is trained only on states visited by the expert. At test time, small prediction errors may cause the policy to drift toward states that were never present in the training data, where the supervised predictor may perform arbitrarily poorly. This *distribution shift* or *covariate shift* issue motivates more advanced imitation-learning methods, such as dataset aggregation (DAgger) [146], which iteratively augments the training set with states visited by the learned policy, thereby mitigating compounding errors at test time. As discussed in later sections, imitation learning has proven particularly relevant in combinatorial optimization and mixed-integer linear programming, where neural networks have been successively trained to approximate the behavior of computationally expensive expert decision rules at a significantly lower computational cost. Building on this motivation, we now introduce the class of function approximators that underpins the learning methods developed in this thesis, namely, neural networks.

2.3 Neural Networks

The rise of deep learning over the past two decades is largely due to neural networks' ability to act as highly expressive function approximators capable of handling data with widely varying structures. In fact, neural networks now underpin state-of-the-art methods across computer vision, natural-language processing, time-series forecasting, generative modeling, and reinforcement learning. In this thesis, neural networks serve as primary function approximators, and the hypothesis class $\mathcal{F}$ is therefore generally determined by the choice of neural architecture. This section introduces the fundamental concepts underlying the field of deep learning, along with several neural architectures that will play a central role in the methods developed throughout this work.

2.3.1 Multilayer Perceptron

The most fundamental neural network architecture is the multilayer perceptron (MLP), which acts as the basic building block for modern neural models. An MLP is a feed-forward network represented as a directed acyclic graph. Neurons are organized into layers indexed by depth: an input layer, an output layer, and one or more hidden layers. Each layer computes a nonlinear transformation

$$o_j^{(L)} = \phi_j^{(L)} \left(\sum_{k=1}^{|L-1|} w_{jk}^{(L)} \, o_k^{(L-1)} + b_k^{(L)} \right), \tag{2.3}$$

where $|L|$ denotes the number of active neurons in layer L, $\phi_j^{(L)}$ is a fixed activation function (e.g., sigmoid, ReLU, or tanh), and the parameters $w_{jk}^{(L)}, b_k^{(L)}$ collectively form the parameter vector θ. Composing layers yields the network function

$$f_\theta(X) = f^{(N)} \circ \cdots \circ f^{(0)}(X).$$

A schematic illustration of an MLP architecture is shown in Figure 2.1.

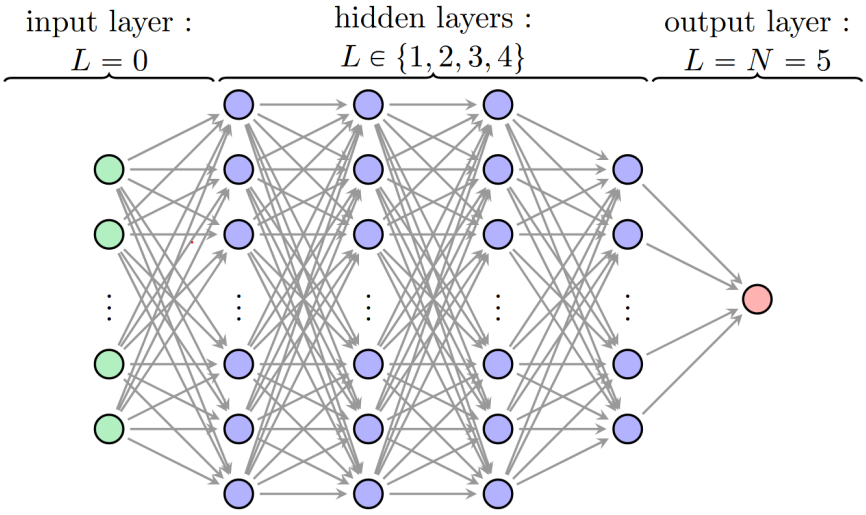


Figure 2.1: Structure of a multilayer perceptron. Excerpt from [55].

2.3.2 Neural network training

Training a neural network amounts to solving an unconstrained optimization problem:

$$\hat{\theta}_n \in \arg\min_\theta \hat{R}_n(\theta) = \frac{1}{n} \sum_{i=1}^{n} l(Y_i, f_\theta(X)).$$

Stochastic gradient descent (SGD) and its variants (most notably Adam [96]) are the workhorses of modern neural network optimization. SGD performs parameter updates of the form

$$\theta \leftarrow \theta - \alpha \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta \, l(Y_i, f(X_i; \theta)), \tag{2.4}$$

with gradients computed efficiently through the **backpropagation** algorithm [147]. Because the loss landscape is highly non-convex, convergence typically reaches a local minimum; however, empirical and theoretical results [38] suggest that in practice, large networks tend to avoid poor local minima.

Although deep learning is not known for abundant theoretical guarantees, one classical result has been particularly influential. The universal approximation theorem [85] states that a one-hidden-layer MLP with a suitable activation function can approximate any continuous function on a compact set to arbitrary precision.

Theorem 2.3.1 (Universal approximator)**.** *Let $f \in \mathcal{C}(K_q)$ be continuous on a compact $K_q \subset \mathbb{R}^q$, and let ϕ be a bounded, continuous, non-decreasing activation function. Then for any $\varepsilon > 0$, there exists an MLP with a single hidden layer of size H such that*

$$\forall x \in K_q, \qquad |\sum_{k=1}^{H} w_{1k}^{(2)} \, \phi(\sum_{i=1}^{q} w_{ki}^{(1)} x_i + b_k^{(1)}) - f(x)| \leq \varepsilon.$$

While the theorem ensures that sufficiently wide networks can achieve arbitrary low training error, it also highlights the importance of regularization to prevent overfitting. Common strategies include dropout [158], early stopping [138], weight regularization, and architectural design choices such as reducing the dimension of hidden layers.

2.3.3 Graph Neural Networks

Many modern learning problems involve data with an inherent relational or combinatorial structure that cannot be represented naturally as fixed-size vectors. Examples include molecular graphs, social networks, routing problems, and, as will be shown soon, mixed-integer linear programs. Neural networks operating on such data must therefore respect the underlying symmetries of graph-structured inputs, in particular permutation invariance of node ordering. *Graph neural networks* (GNNs) address this need by defining computations directly over the nodes and edges of a graph, allowing information to propagate along its topology.

Formally, let $G = (V, E)$ be a graph with node features $\{h_j^{(0)}\}_{j \in V}$ and (optionally) edge features $\{e_{ij}\}_{(i,j) \in E}$. Most GNN architectures rely on the *message-passing* paradigm [63], where node representations are iteratively updated by aggregating information from their neighbors. At layer k, a generic message-passing update takes the form

$$m_j^{(k)} = \bigoplus_{i \in \mathcal{N}(j)} \phi_{\text{msg}}^{(k)}(h_j^{(k)}, h_i^{(k)}, e_{ij}),$$

$$h_j^{(k+1)} = \phi_{\text{upd}}^{(k)}(h_j^{(k)}, m_j^{(k)}),$$

where $\phi_{\text{msg}}^{(k)}$ and $\phi_{\text{upd}}^{(k)}$ are learnable functions typically implemented as MLPs, $\mathcal{N}(j)$ is the set of neighbors of the node j, and $\bigoplus$ is a permutation-invariant aggregation operator such as SUM, MEAN, or MAX. Through repeated iterations, nodes gather information from progressively larger neighborhoods, enabling the model to capture global graph structures.

GNN architectures have become central tools for learning from relational structured data across a broad range of domains. In the context of this thesis, they enable the explicit incorporation of the combinatorial structure of optimization problems into the learning process, allowing neural networks to detect and exploit inherent MILP relational patterns induced by branch-and-bound exploration.

2.4 Imitation learning for variable selection in branch-and-bound

Having described the general algorithmic framework of modern mixed-integer linear programming solvers, and introduced the foundational principles of machine learning theory, we now consider their application within mixed-integer linear programming.

2.4.1 MILP graph bipartite representation

A central challenge in learning-based approaches to mixed-integer linear programming is the design of an appropriate *observation function*, which maps the solver's internal state to a representation amenable to learning. In fact, such a function must expose the relevant combinatorial and algebraic structure of the problem, while remaining invariant to arbitrary modeling choices, such as the ordering of variables and constraints.

As presented in Section 1.2, a MILP is defined by a set of variables and a set of linear constraints, with interactions specified by the constraint matrix: each constraint couples a subset of variables,

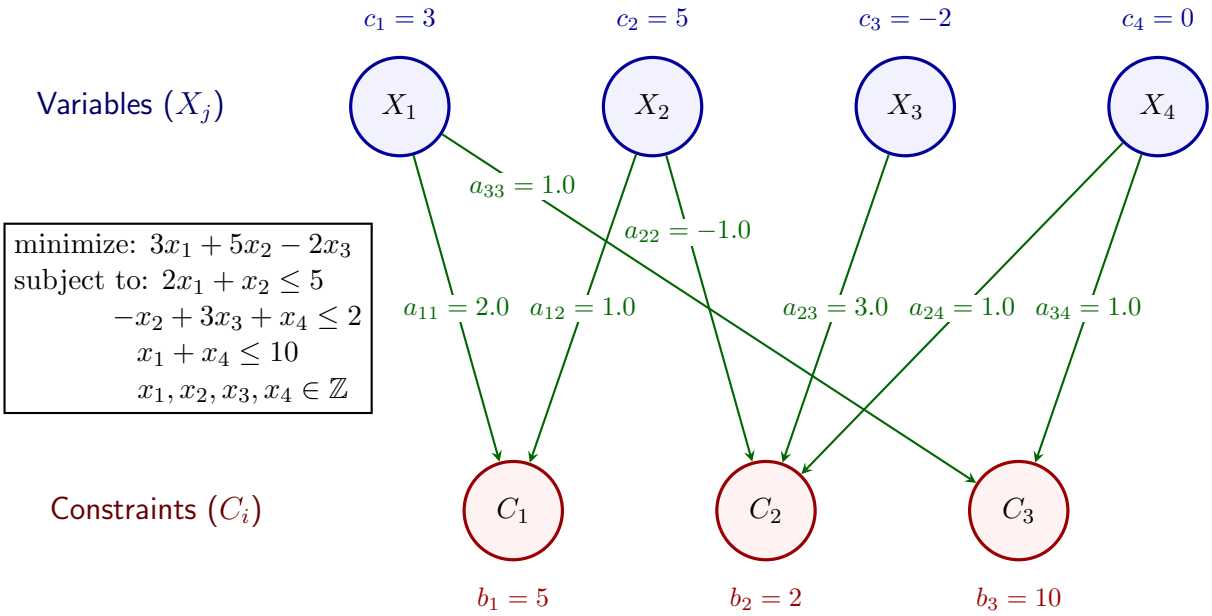


Figure 2.2: Bipartite graph representation of a mixed-integer linear program. Variable nodes X_j correspond to decision variables and constraint nodes C_i to linear constraints; edges indicate nonzero coefficients a_{ij} in the constraint matrix. In particular, bipartite graphs successfully encode (A, b, c).

and each variable typically participates in multiple constraints. Any observation function that fails to reflect this relational structure risks discarding information that is essential for reasoning about downstream solver decisions.

Furthermore, another desirable property for a MILP observation function is to admit function approximators capable of operating across instances of different sizes, *i.e.* across instances with varying numbers of variables and constraints. In fact, in practice modern branch-and-bound solvers dynamically add and remove variables and constraints during presolve to strengthen the problem formulation. Similarly, cutting-plane modules introduce additional constraints throughout the search. As a result, the effective problem dimension evolves in a non-predictable manner throughout the B&B process, and fixing the input dimensionality of the learning model would severely limit its applicability, even in the setting of fixed-size repeated MILPs.

Following the seminal work of Gasse et al. [62], a widely adopted approach consists in representing MILP instances as *bipartite graphs*, connecting variable and constraint nodes. Consider a MILP P as defined in Eq. (1.2), with $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$ and $c, l, u \in \mathbb{R}^n$. The associated bipartite graph is

defined as

$$G = (V, E), \qquad V = X \cup C,$$

where X denotes the set of variable nodes and C the set of constraint nodes. As illustrated in Figure 2.2, an edge $e_{ij} \in E$ connects variable node X_j to constraint node C_i if and only if the coefficient a_{ij} is nonzero. Additionally, node and edge attributes encode local information on the MILP structure. In the context of branch-and-bound, variable nodes are typically endowed with both static and dynamic features such as objective coefficients, variable lower and upper bounds, integrality indicators, reduced costs, fractionality in the current LP relaxation, etc. Likewise, constraint nodes include features such as right-hand-side values, constraint senses, dual variables, or slack information. Edge attributes encode the corresponding nonzero coefficients a_{ij}, often normalized to improve numerical stability. Importantly, this representation is invariant under permutations of variables and constraints, satisfying a key requirement for permutation-equivariant learning architectures.

When coupled with message-passing neural networks, the MILP bipartite graph enables structured information flow between variables and constraints. Variable nodes aggregate information from all constraints in which they appear, while constraint nodes reason jointly over the variables they involve. Through multiple message-passing layers, the network captures higher-order interactions and global structural properties that are difficult to encode using handcrafted features alone. Importantly, the combined use of bipartite MILP graph representation with message passing neural architectures enables training learning models on MILP instances of varying sizes, making it well-suited to the branch-and-bound framework. Beyond this flexibility, it confers a further advantage: by encoding the combinatorial structure of MILPs and enforcing permutation invariance through message passing, this representation introduces strong inductive biases that promote generalization across problem scales. In fact, as will be demonstrated later in this thesis, models trained on smaller, computationally tractable instances can thus be effectively deployed on larger, more challenging instances.

2.4.2 Learning to imitate strong branching

Building on several prior works [6, 7, 79, 94], Gasse et al. [62] proposed to leverage the graph bipartite observation to train graph convolutionnal neural networks to replicate the behaviour of strong branching using behavioral cloning. Their approach mainly relied on SCIP, as it leveraged the solver's full branch-and-bound machinery, simply overriding branching decisions through dedicated callbacks.

Impressively, the resulting policy consistently outperformed reliability branching, the SCIP default branching heuristic, over multiple repeated MILP benchmarks, marking a significant milestone in the literature of learning to branch. In fact, beyond its immediate empirical success, this work also laid the groundwork for a substantial body of research in machine learning for combinatorial optimization. Notably, subsequent contributions in variable selection [73, 125], node selection [103, 186], cut selection [134, 173] and primal heuristics [86, 133] have all adopted the graph bipartite observation function together with the graph neural network architecture first proposed by Gasse et al. [62]. In addition, the authors' follow-up release of the Ecole library [139] provided the community with a standardized interface for generating repeated MILP instances of arbitrary size across four canonical problem classes, which are presented in Appendix B. These four MILP problem classes, hereafter collectively referred to as the Ecole benchmark, have since become widely adopted as the standard testbed for training and evaluating learning methods in mixed-integer linear programming. In turn, the work presented in this thesis builds in large part on the methodological and implementation frameworks introduced by these authors.

In spite of the strong empirical success achieved by GNNs operating on MILP graph representations across a broad range of tasks, recent works [35, 36] have shown that this combination suffers from fundamental expressivity limitations which ultimately restricts the best performance attainable by these models. In particular, message-passing GNNs operating on variable–constraint bipartite graphs are provably unable to distinguish certain MILP instances that differ in key structural properties, such as feasibility or strong branching scores, due to intrinsic limitations in their expressive power. These results can be traced back to the fact that standard message-passing architectures are bounded by the separation power of the Weisfeiler–Lehman test [174], which is insufficient to separate specific classes of MILPs. While these works also show that expressivity can be recovered under additional assumptions, these findings suggest that, although MILP bipartite graph representations coupled with message-passing GNNs provide a powerful and scalable inductive bias, they should not be viewed as universally expressive models of MILP structure, and their limitations must be carefully considered when designing learning-based branching strategies.

In this chapter, we have introduced the theoretical background, from mixed-integer linear programming to graph neural networks, necessary for presenting the work of Gasse et al. [62]. This background

sets the stage for the next chapter, which explores the design of a stronger oracle for learning to branch by imitation. Specifically, we explore evaluating strong branching over a broader class of disjunctions, moving beyond standard elementary splits that involve only a single fractional variable.

Chapter 3

Learning to branch on general disjunctions

Content

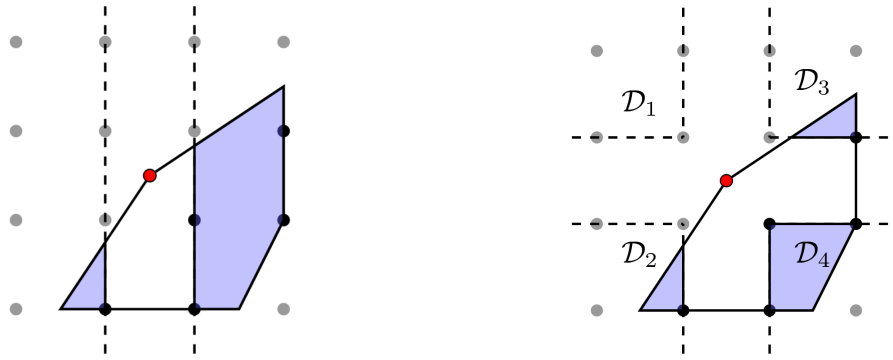
In this chapter, we explore extending the approach first proposed by Gasse et al. [62] by learning to imitate strong branching decisions over a broader set of branching disjunctions. So far, in line with common implementation practice in modern mixed-integer linear programming solvers, we have restricted branching to the operation of partitioning B&B nodes by applying *elementary* (or *single*) *disjunctions*, that is disjunctions involving only one (fractional) integer variable

$$x_i \leq \lfloor \tilde{x}_i \rfloor \quad \vee \quad x_i \geq \lfloor \tilde{x}_i \rfloor + 1,$$

where $\tilde{x}_i$ denotes the value of x_i in x^*_{LP}. Under this restriction, branching effectively reduces to variable selection, that is selecting the variable defining the single disjunction used to partition the current node. However, in practice the branch-and-bound algorithm can leverage richer, *general disjunctions* of the form

$$\bigwedge_{k=1}^{K} \left(\left((\alpha^k)^\top x \leq \alpha_0^k \right) \vee \left((\alpha^k)^\top x \geq \alpha_0^k + 1 \right) \right),$$

to partition the search space, with $(\alpha^k, \alpha_0^k) \in \mathbb{R}^n \times \mathbb{R}$ for all $1 \leq k \leq K$. Such general disjunctions must still be valid, in the sense that they must separate the current LP optimal solution without excluding any feasible integer solution. Naturally, general disjunctions subsume traditional single disjunctions associated with variable x_i, which are recovered by setting $K = 1$, $\alpha^1 = e_i$ and $\alpha_0^1 = \tilde{x}_i$. Figure 3.1 illustrates the partitioning resulting from the application of single and general disjunctions. Note that exploiting general disjunctions yields B&B trees that are no longer binary in the general case.



(a) Single disjunction (b) General disjunction

Figure 3.1: Two valid branching disjunctions for partitioning the LP polyhedron. Both disjunctions effectively separate the LP optimal solution (in red), while preserving all integer points of the original polyhedron (in black). In Figure 3.1a, the single disjunction writes $x_1 \leq \lfloor \tilde{x}_1 \rfloor \ \vee \ x_1 \geq \lfloor \tilde{x}_1 \rfloor + 1$. In Figure 3.1b, the general disjunction writes $(x_1 \leq \lfloor \tilde{x}_1 \rfloor \vee x_1 \geq \lfloor \tilde{x}_1 \rfloor + 1) \wedge (x_2 \leq \lfloor \tilde{x}_2 \rfloor \vee x_2 \geq \lfloor \tilde{x}_2 \rfloor + 1)$. Excerpt from [166].

For each B&B node, there exists an infinite number of valid general disjunction that can be used to partition the LP polyhedron. Mahajan and Ralphs [117] have shown that the problem of finding the general disjunction yielding the highest dual bound improvement, and, consequently, the highest strong branching score, is $\mathcal{NP}$-hard in the general case. Since our goal is to learn to select the valid general disjunction with the highest strong branching score, to keep data generation tractable we need to restrict strong branching evaluation to a finite subset of branching candidates. Building on the original work by Karamanov and Cornuéjols [92] on branching on general disjunctions, as well as subsequent work by Turner et al. [166] investigating the interplay between branching and cutting, we explore learning to replicate strong branching over a broader set of branching disjunctions, including *Gomory mixed-integer (GMI) disjunctions*, defined as the disjunctions underlying the construction of Gomory mixed-integer cuts. The next section formally introduces GMI disjunctions and motivates their use as branching candidates.

3.1 GMI disjunctions

As detailed in Section 2.1.5, Gomory mixed-integer (GMI) cuts as defined in Eq. (2.2) arise from the intersection of a valid disjunction with the rays of the polyhedral cone defined by the simplex optimal basis. Specifically, Andersen et al. [8] showed that original disjunctions underlying GMI intersections cuts are built from single disjunctions, which have been strengthened on nonbasic variable so as to maximize the cutoff distance from the current LP solution. This section provides additional background on GMI disjunctions, their role in the derivation of GMI cuts, along with their practical relevance as branching candidates when learning to imitate strong branching decisions.

3.1.1 Definition

Let us consider once again the rows $i \in B$ of the simplex tableau

$$x_i = \tilde{x}_i - \sum_{j \in J} \tilde{a}_{ij} x_j,$$

where B and J denote the sets of basic and nonbasic indices, respectively. Following these notations, extreme rays r^j for $j \in J$ associated with basis B write

$$r_i^j = \begin{cases} -\tilde{a}_{ij}, & \text{if } i \in B, \\ 1, & \text{if } i = j, \\ 0, & \text{otherwise.} \end{cases}$$

For each basic variable $i \in B$, let us consider the split disjunction

$$\left((\tilde{\alpha}^i)^\top x \leq \tilde{\alpha}_0^i\right) \vee \left((\tilde{\alpha}^i)^\top x \geq \tilde{\alpha}_0^i + 1\right),$$

with $(\tilde{\alpha}^i, \tilde{\alpha}_0^i)$ defined as

$$\tilde{\alpha}_0^i = \lfloor \tilde{x}_i \rfloor, \qquad \tilde{\alpha}_j^i = \begin{cases} \lfloor \tilde{a}_{ij} \rfloor, & j \in J,\ \tilde{a}_{ij} - \lfloor \tilde{a}_{ij} \rfloor \ \leq\ \tilde{x}_i - \lfloor \tilde{x}_i \rfloor, \\ \lceil \tilde{a}_{ij} \rceil, & j \in J,\ \tilde{a}_{ij} - \lfloor \tilde{a}_{ij} \rfloor \ >\ \tilde{x}_i - \lfloor \tilde{x}_i \rfloor, \\ 1, & j = i, \\ 0, & \text{otherwise.} \end{cases}$$

In the following, we write $D(\alpha, \alpha_0)$ the split disjunction associated with hyperplane (α, α_0). As illustrated in Figure 3.2, given a valid disjunction $D(\alpha, \alpha_0)$ and the set of extreme rays $(r^j)_{j \in J}$ defining the LP polyhedral cone, a valid intersection cut can be obtained by intersecting $D(\alpha, \alpha_0)$ with $(r^j)_{j \in J}$.

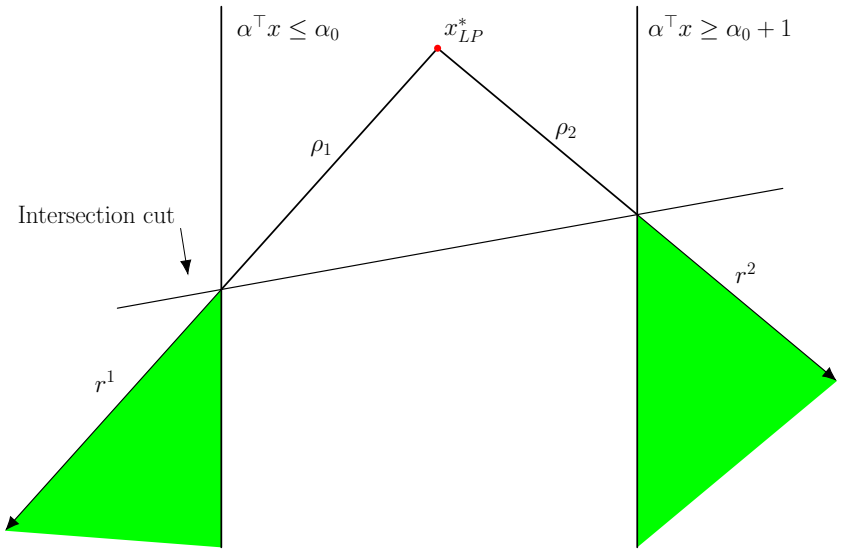


Figure 3.2: Intersection cut derivation from extreme rays $(r^j)_{j \in J}$ associated with basis B and valid split disjunction $D(\alpha, \alpha_0)$.

Following Balas [12], the intersection cut associated with disjunction $D(\alpha, \alpha_0)$ and the extreme rays of the basis B takes the form:

$$\sum_{j \in J} \frac{x_j}{\rho_j(\alpha, \alpha_0)} \geq 1, \tag{3.1}$$

where

$$\rho_j(\alpha, \alpha_0) = \begin{cases} -\dfrac{\varepsilon(\alpha, \alpha_0)}{\alpha^\top r^j} & \text{if } \alpha^\top r^j < 0, \\ \dfrac{1 - \varepsilon(\alpha, \alpha_0)}{\alpha^\top r^j} & \text{if } \alpha^\top r^j > 0, \\ +\infty & \text{otherwise,} \end{cases}$$

taking $\varepsilon(\alpha, \alpha_0) = \alpha^\top \tilde{x} - \alpha_0$.

Importantly, [126, 166] showed that, when intersection cuts are generated from the optimal simplex basis B using split disjunction $D(\tilde{\alpha}^i, \tilde{\alpha}^i_0)$, the resulting cut coincides with the GMI cut associated with row $i \in B$ of the simplex tableau, as defined in Eq. (2.2). Consequently, for each GMI cut, and thus for each fractional integer variable $x_i \in B$, there exists an associated valid general disjunction $D(\tilde{\alpha}^i, \tilde{\alpha}^i_0)$ that can be leveraged for branching. Notably, these disjunctions are inexpensive to compute, as their coefficients are obtained directly from the simplex tableau. Building on this result, and following the work of Karamanov and Cornuéjols [92], in our experiments we propose to augment the set of branching candidates subject to strong branching evaluation with the family of GMI disjunctions $(D(\tilde{\alpha}^i, \tilde{\alpha}^i_0))_{i \in B}$.

3.1.2 GMI disjunctions as branching candidates

Considering the set of GMI disjunctions $(D(\tilde{\alpha}^i, \tilde{\alpha}^i_0))_{i \in B}$ effectively defines a finite set of general disjunction, thereby enabling tractable strong branching evaluation during the data generation phase in IL. Nevertheless, given the arguments presented so far, it remains unclear whether such general disjunctions can provide better branching candidates than the original single branching disjunctions. In other words, given an elementary disjunction, should we expect its associated GMI disjunction to yield stronger branching decisions? To shed light on this question, we provide additional background on the exact derivation of GMI disjunctions $(D(\tilde{\alpha}^i, \tilde{\alpha}^i_0))_{i \in B}$.

Let us consider a valid disjunction $D(\alpha, \alpha_0)$. Intersecting $D(\alpha, \alpha_0)$ with the polyhedral cone defined by the optimal basis B generates an intersection cut parametrized by $(\rho_j(\alpha, \alpha_0))_{j \in J}$ as defined in Eq. (3.1). The cut efficacy, defined as the normalized Euclidean distance between the optimal LP

solution x^*_{LP} and the hyperplane induced by the intersection cut, is given by :

$$d(x^*_{LP}, \alpha, \alpha_0) = \left(\sum_{j \in J} \frac{1}{\rho_j(\alpha, \alpha_0)^2} \right)^{-\frac{1}{2}}. \tag{3.2}$$

In order to generate intersection cuts with maximal efficacy, *i.e.* with the strongest separation power, one should therefore select a split disjunction $D(\alpha, \alpha_0)$ that maximizes $d(x^*_{LP}, \alpha, \alpha_0)$. Equivalently, this amounts to selecting a general disjunction that yields larger values for $\rho_j(\alpha, \alpha_0)$ for $j \in J$. Concretely, $D(\tilde{\alpha}^i, \tilde{\alpha}^i_0)$ is obtained by first considering the elementary disjunction $D(e_i, \lfloor \tilde{x}_i \rfloor)$ associated with the row i of the simplex tableau, before strengthening one-by-one the coefficients associated with non-basic variables so as to maximize $\rho_j(\alpha, \alpha_0)$ for $j \in J$. As shown by Andersen et al. [8], in terms of efficacy, the intersection cuts generated from the disjunctions $D(\tilde{\alpha}^i, \tilde{\alpha}^i_0)$ thus constructed dominate all intersection cuts built from individual simplex tableau rows.

Importantly, dual bound improvement (the ultimate objective of separation) cannot be reduced to cut efficacy alone, as it also depends on the alignment of the cut with the objective, as well as on other algorithmic factors. Nevertheless, higher-efficacy cuts are generally expected to yield stronger dual bound improvements, leading modern solvers to rely in practice on efficacy as a first-order proxy for dual bound improvement.

Building on these insights, several observations follow. First, by construction, GMI disjunctions $D(\tilde{\alpha}^i, \tilde{\alpha}^i_0)$ yield intersection cuts with stronger separation power than those obtained from corresponding single variable disjunctions. Therefore, the intersection cuts produced by GMI disjunctions are generally expected to yield larger dual bound improvements than those generated from corresponding single variable disjunctions. Additionally, as illustrated in Figure 3.2, the strong branching score associated with a valid disjunction $D(\alpha^i, \alpha^i_0)$ is lower bounded by the dual bound improvement obtained by adding the corresponding intersection cut. Consequently, in the general case, strong branching scores achieved by GMI disjunctions $D(\tilde{\alpha}^i, \tilde{\alpha}^i_0)$ are expected to admit a higher lower bound than those associated with elementary disjunctions.

This does not imply that GMI disjunctions systematically induce branching decisions with higher strong branching values than their single disjunction counterparts. Rather, following the intuition that a deep intersection cut is generally expected to provide strong dual bound improvement, it suggests that, since GMI disjunctions produce deeper intersection cuts than elementary disjunctions, they may

likewise constitute stronger branching candidates.

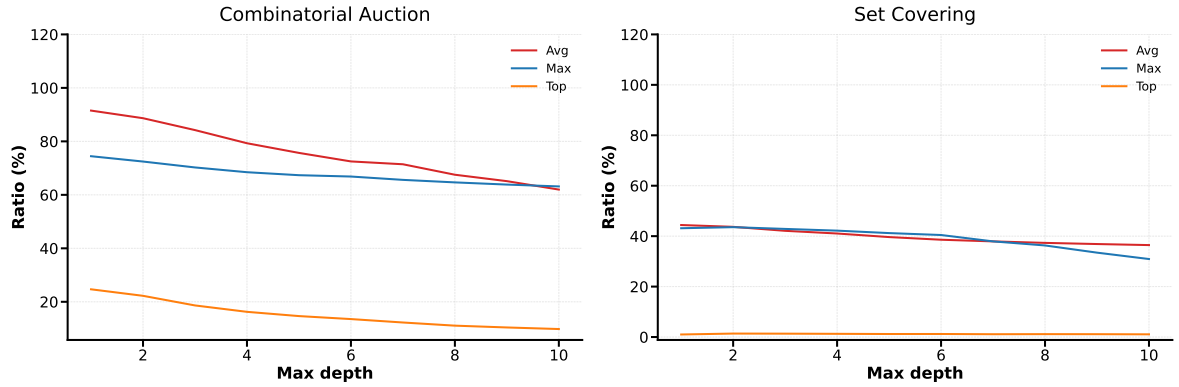
3.1.3 Related work

Following this intuition, a series of works [40, 92, 126] have explored applying strong branching over GMI disjunctions in place of elementary disjunctions, in order to solve MILP while producing smaller B&B trees. Computational experiments on the MIPLIB benchmark [65] provided encouraging evidence regarding the relevance of GMI disjunctions for branching, as GMI-based strong branching consistently closed a larger fraction of the integrality gap within the first hundred B&B nodes. However, these gains were offset by a significant computational overhead, as processing a B&B node under GMI strong branching is reported to take roughly twice as long as with elementary disjunctions, limiting the practical deployment of GMI strong branching in modern MILP solvers.

More recently, building on the same intuition that valid disjunctions generating good intersections cuts must be good branching candidates too, Turner et al. [166] sought to incorporate the information conveyed by the efficacy of the GMI cuts associated with simplex tableau rows into the overall branching score of single disjunctions associated with corresponding basic variables. In practice, GMI cut efficacy proved to be an effective tie-breaking criterion among multiple promising variable candidates, yielding an overall 5% reduction in solution time on MIPLIB instances when implemented on top of hybrid reliability branching. As a result, GMI cut efficacy has been incorporated into the branching candidate scoring function of SCIP's current default branching rule, where it is assigned a small weight (10^{-5}) to serve as tie-breaker.

3.2 Generalized strong branching

In this section, we explore the design of a stronger branching expert for learning to branch by imitation. Specifically, our proposed oracle evaluates strong branching scores over a candidate set comprising both single and GMI disjunctions. While evaluating strong branching over both single and general disjunctions incurs a significant computational overhead, an imitation learning policy trained on such decisions could amortize this cost at inference time, selectively leveraging general disjunctions without triggering repeated expensive evaluations. We investigate whether such a policy can translate the theoretical advantages of GMI disjunctions into practical solving performance gains.



(a) Combinatorial Auction (b) Set Covering

Figure 3.3: GMI disjunction strong branching scores versus single disjunction strong branching scores across instance families. **Avg**: average ratio between the strong branching score of a GMI disjunction and that of its corresponding single disjunction, computed per B&B node. **Max**: ratio between the SB score of the best GMI disjunction and that of the best single disjunction. **Top**: frequency at which the best strong branching action disjunction is a GMI disjunction.

As in Gasse et al. [62], experiments are carried out using the open-source solver SCIP 8.0.3 [27] as backend MILP solver, along with the Ecole library [139] for instance generation. Here, we consider two families of MILP instances: set covering and combinatorial auctions. Detailed background on these problem families, including generation procedures and parameters, is provided in Appendix B.

3.2.1 Strong branching evaluation

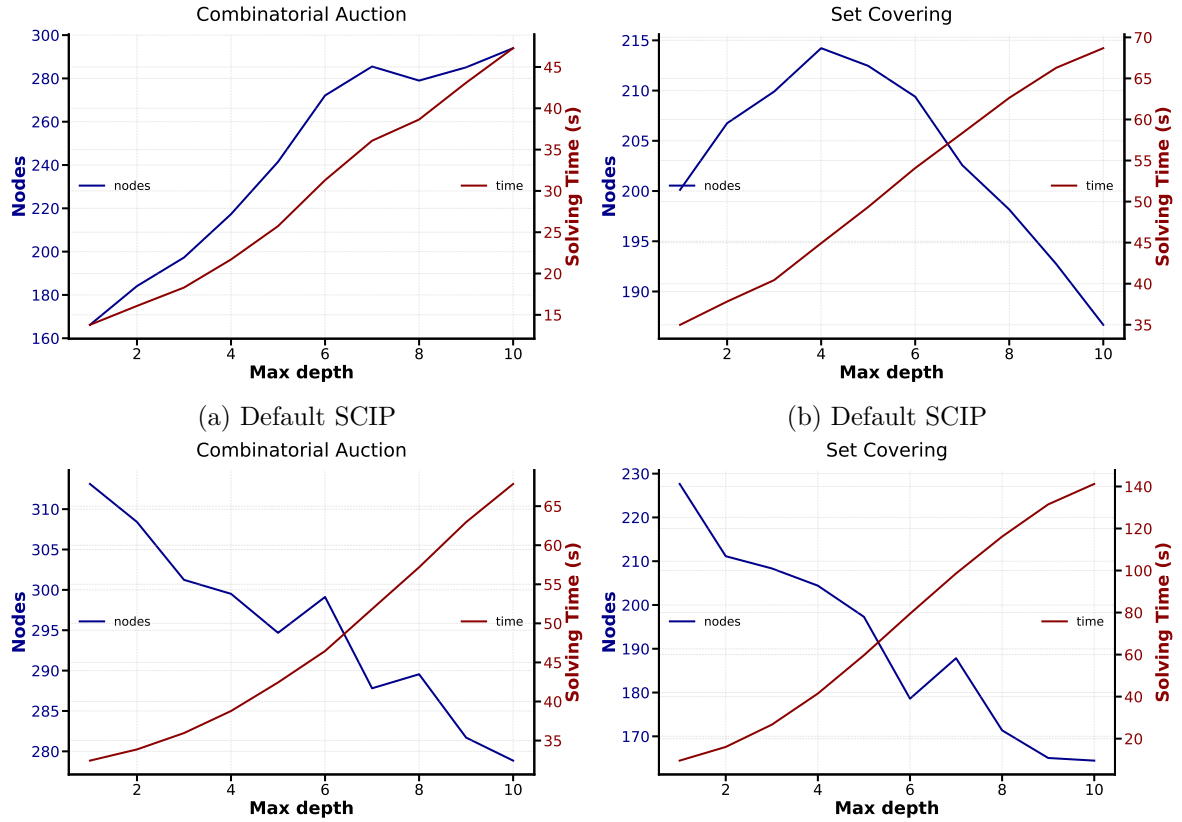
We first start by evaluating our proposed expert rule, which we term *generalized strong branching*, across problem families. Figure 3.3 investigates whether GMI disjunctions empirically constitute viable branching candidates for strong branching imitation on combinatorial auction and set covering instances. Crucially, we report three quantities: (i) the average ratio of the strong branching score achieved by a GMI disjunction divided by that of its corresponding single disjunction, averaged across all B&B nodes; (ii) the average ratio between the best GMI disjunction score and the best single disjunction score at each node; and (iii) the frequency at which the disjunction achieving the highest strong branching score is a GMI disjunction. We study the evolution of these three quantities as a function of the maximum depth at which our generalized strong branching rule is applied, beyond which standard single variable strong branching is used. Across both instance families, GMI disjunctions consistently achieve lower strong branching scores than their single disjunction counterparts. On

combinatorial auction instances, GMI disjunctions still represent the best strong branching action 10 to 30% of the time depending on the depth, whereas on set covering instances this figure never exceeds 5%. These results suggest that, on the benchmarks considered, GMI disjunctions constitute viable alternatives but do not provide intrinsically superior candidates for strong branching evaluation.

3.2.2 B&B tree size

By construction, generalized strong branching selects a disjunction whose strong branching score is at least as high as that of the best single disjunction candidate. Consequently, generalized strong branching is expected to produce B&B trees of smaller size than standard single variable strong branching. Figure 3.4 reports the average B&B tree size and total solving time achieved by generalized strong branching over 100 instances, as a function of the maximum depth at which it is applied. Again, beyond this maximum depth, standard single variable strong branching is used. Surprisingly, as shown in Figures 3.4a and 3.4b, generalized strong branching does not consistently achieve tree size reductions under SCIP with default parameters. On combinatorial auction instances, the opposite is observed: applying general strong branching up to greater depths produces larger B&B trees.

We attribute this counterintuitive behaviour to the interaction between branching and domain propagation in SCIP. Domain propagation techniques originate from constraint programming and are now widely integrated in modern MILP solvers. Their role consist in tightening variable bounds by exploiting the problem's constraint structure, without solving any LP relaxation. When branching on a single disjunction $x_i \leq \lfloor \tilde{x}_i \rfloor$ (resp. $x_i \geq \lfloor \tilde{x}_i \rfloor + 1$), the resulting child node directly tightens the bound of variable x_i. SCIP's propagation modules exploit this bound change to infer tighter bounds on other variables through the constraint matrix, implications, and clique structures. These cascading reductions can fix additional variables and detect infeasibilities before the LP relaxation is re-solved, providing substantial pruning power beyond what is reflected in the LP dual bound alone. By contrast, branching on a GMI disjunction $D(\tilde{\alpha}^i, \tilde{\alpha}^i_0)$ imposes a general linear inequality involving multiple variables on the child node. Such constraints do not directly translate into simple variable bound changes, and therefore provide limited input for SCIP's propagation routines, which are primarily designed to exploit individual bound tightenings. As a result, while GMI disjunctions may yield higher strong branching scores, the child nodes themselves offer less opportunity for domain propagation.



(a) Default SCIP

(b) Default SCIP

(c) Propagation Off

(d) Propagation Off

Figure 3.4: Solving time and B&B tree size performance achieved by the generalized strong branching rule across combinatorial auction and set covering instances. Performance is reported as a function of the maximum depth at which the generalized strong branching rule is applied; beyond this depth, standard single variable strong branching is used.

We validate this hypothesis in Figures 3.4c and 3.4d, which report the average B&B tree size and total solving time achieved by generalized strong branching under SCIP with propagation deactivated. In this setting, generalized strong branching achieves monotone tree size reduction as the maximum depth increases, confirming that propagation is the source of earlier degradation. Unfortunately, this tree size reduction is accompanied by strictly higher solving times, as deactivating propagation removes a mechanism that substantially accelerates resolution. This analysis reveals a limitation of strong branching as an evaluation criterion for general disjunctions: the strong branching score captures the immediate LP dual bound improvement but does not account for the downstream propagation effects that amplify the impact of single-variable branching within the solver's overall algorithmic pipeline. On instances where propagation plays a significant role in tree pruning, this indirect advantage of

single-variable branching can outweigh the direct dual bound gains achieved by GMI disjunctions.[1] Under these circumstances, generalized strong branching thus does not constitute a stronger expert rule for training branching policies by imitation.

3.3 Conclusion

While generalized strong branching over GMI disjunctions did not yield consistent performance improvements on the benchmarks considered, this chapter provides several insights relevant to future work on learning to branch with general disjunctions.

First, our experiments confirm that strong branching scores alone are not a sufficient predictor of branching quality when general disjunctions are involved. The interaction between branching and domain propagation, a mechanism that disproportionately benefits single variable disjunctions, can offset the dual bound gains achieved by general disjunctions, particularly on binary problems where branching amounts to variable fixing. Any future attempt to learn to imitate strong branching over general disjunctions must therefore account for these solver-level interactions, either by incorporating propagation effects into the expert signal or by targeting instance classes where propagation plays a lesser role.

Second, the methodology described in this chapter is not specific to GMI disjunctions and extends naturally to any family of general disjunctions for which strong branching evaluation is tractable. In particular, for specific problem classes where structural domain expert knowledge can be exploited to derive intersection cuts with strong separation properties, augmenting the pool of branching candidates with the associated underlying general disjunctions remains a promising direction. In such settings, the gap between the intersection cuts produced by general and elementary disjunctions may be sufficiently large to overcome the propagation disadvantage observed in our experiments, making the approach viable for deriving stronger experts for imitation learning.

Third and finally, our experiments revealed that our single-variable strong branching implementation produced B&B trees substantially larger than those generated by SCIP's built-in strong branching rule. This discrepancy arises because modern MILP solvers do not treat strong branching evaluations

[1] This is particularly the case for combinatorial auction and set covering instances. Because these problems involve exclusively binary variables, branching on a single disjunction is equivalent to fixing a variable in each child node, which provides especially strong potential for cascading bound reductions through propagation.

as isolated score computations: the bound information obtained from solving child LPs during strong branching is propagated through auxiliary solver modules (including, but not restricted to, domain propagators and conflict analysis) which in turn contribute to pruning and tightening throughout the subsequent search. These side effects are integral to the low tree sizes achieved by strong branching in practice. Crucially, when an imitation learning policy replicates strong branching decisions without actually solving the child LPs, it does not generate this bound information, and therefore cannot benefit from the same downstream effects. As a result, the B&B trees produced by IL policies are substantially larger than those of the expert they imitate. This observation, consistent with findings reported by Gamrath and Schubert [61] and Scavuzzo et al. [149], nuances the effectiveness of strong branching as a demonstration expert for imitation learning: the low tree sizes it achieves are not solely attributable to the quality of the branching decisions themselves, but also to the bound information discovered and reused as a byproduct of LP evaluations. This phenomenon will be revisited over the course of this manuscript.

In light of the findings presented in this chapter; and in line with the broader literature on learning to branch, in the remainder of this thesis we restrict the set of branching candidates to the elementary disjunctions associated with fractional integer variables in the simplex basis. Furthermore, in order to train learning agents that directly target B&B tree size minimization, we turn to the reinforcement learning paradigm for the design of future branching agents. Accordingly, the next chapter introduces the necessary reinforcement learning background underlying the methods later developed in this thesis.

Part II

Learning to branch by reinforcement

Chapter 4

Preliminaries

Content

Reinforcement learning (RL) is concerned with learning efficient control strategies for complex dynamical systems. In this thesis, we propose to view the branch-and-bound process as a dynamical system to control, and the branching heuristic as the control law to optimize. This chapter introduces necessary technical background on reinforcement learning, and reviews prior attempts in the literature to adapt RL algorithms to the branch-and-bound framework.

4.1 Reinforcement Learning

Reinforcement learning aims to train one or several sequential decision-making agents to map state observations to action distributions. Through repeated interaction with its environment, the agent learns which actions lead to favorable outcomes, and which degrade long-term performance via a trial-and-error process. The interaction loop between the agent and its environment in RL is illustrated in Figure 4.1.

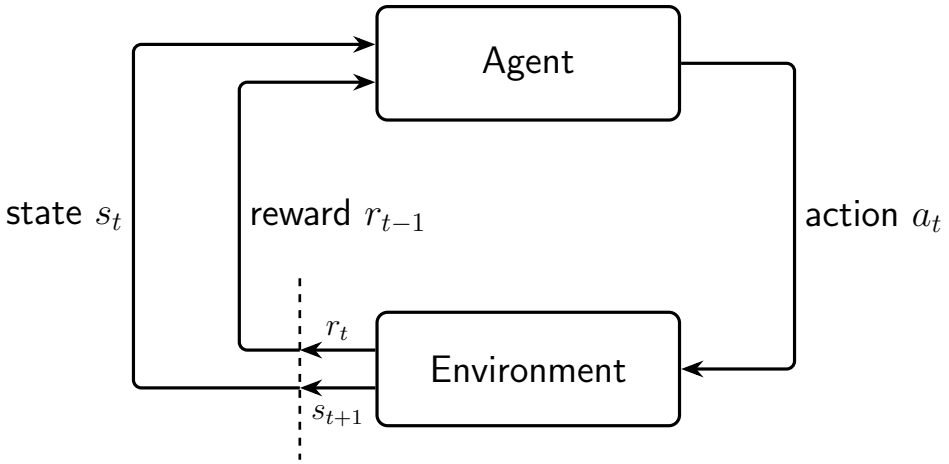


Figure 4.1: Agent-environment interaction loop in RL.

4.1.1 Markov decision processes

In this thesis, we consider the setting of discrete-time, deterministic Markov decision process (MDP) [140] defined by the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, p_0, \mathcal{R})$. The state space $\mathcal{S}$ denotes the set of all states the environment may occupy. The action space $\mathcal{A}$ specifies the set of actions available to the agent, while the (deterministic) Markov transition function $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ encodes the dynamics of the environment. Finally, p_0 defines the initial state distribution, while the reward function $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times$

$\mathcal{S} \to \mathbb{R}$ assigns scalar feedback signals to each transition. At each time step t, the agent observes $s_t \in \mathcal{S}$ the current state of the environment, before executing action $a_t \in \mathcal{A}$ and receiving reward $r_t = \mathcal{R}(s_t, a_t, s_{t+1})$ as the environment transitions to the next state $s_{t+1} = \mathcal{T}(s_t, a_t)$. The resulting sequence $(s_0, a_0, r_0, s_1, a_1, r_1, \dots)$ generated by the agent–environment interaction is called a trajectory (or episode) of the Markov decision process. A trajectory terminates when the environment reaches a terminal state from which no further transitions are possible; in the absence of such a state, the trajectory is infinite.

4.1.2 Value iteration

Given a trajectory starting in state s_0 sampled according to the initial distribution p_0, the total (discounted) gain is defined for all $t \geq 0$ as

$$G_t = \sum_{t'=t}^{\infty} \gamma^{t'-t} \mathcal{R}(s_{t'}, a_{t'}, s_{t'+1}),$$

with $\gamma \in [0, 1]$. The objective of an RL agent is to maximize the expected gain of the trajectories yielded by its action selection policy $\pi : \mathcal{S} \to \mathcal{A}$. This is equivalent to finding the policy maximizing state value $V^\pi : \mathcal{S} \to \mathbb{R}$ and state-action value $Q^\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ functions defined as

$$V^\pi(s_t) = \mathop{\mathbb{E}}_{a_{t'} \sim \pi(s_{t'})} \left[\sum_{t'=t}^{\infty} \gamma^{t'-t} \mathcal{R}(s_{t'}, a_{t'}, s_{t'+1}) \right] \qquad \text{and} \qquad Q^\pi(s_t, a_t) = \mathcal{R}(s_t, a_t, s_{t+1}) + \gamma V^\pi(s_{t+1}).$$

Optimal state and state-action value functions V^* and Q^* indicate the highest achievable cumulated gain in the MDP. They satisfy Bellman optimality equations

$$V(s) = \max_{a \in \mathcal{A}} \mathcal{R}(s, a, s') + \gamma V(s') \qquad \text{and} \qquad Q(s, a) = \mathcal{R}(s, a, s') + \max_{a' \in \mathcal{A}} \gamma Q(s', a'),$$

noting $s' = \mathcal{T}(s, a)$ for $(s, a) \in \mathcal{S} \times \mathcal{A}$. In turn, these equations define the Bellman optimality operator $\mathcal{B}$, such that

$$(\mathcal{B}V)(s) = \max_{a \in \mathcal{A}} \mathcal{R}(s, a, s') + \gamma V(s') \qquad \text{and} \qquad (\mathcal{B}Q)(s, a) = \mathcal{R}(s, a, s') + \max_{a' \in \mathcal{A}} \gamma Q(s', a').$$

For $\gamma \in [0, 1)$, this operator is a contraction in the supremum norm over the function spaces $\mathcal{F}(\mathcal{S}, \mathbb{R})$ and $\mathcal{F}(\mathcal{S} \times \mathcal{A}, \mathbb{R})$ respectively, and admits a unique fixed point. Thus, repeated application of $\mathcal{B}$ to any initial function V_0 or Q_0 converges to respectively V^* or Q^*. This observation provides a principled method for computing optimal value functions, known as value iteration. In the case of

state-action value iteration, the optimal policy can be retrieved by acting greedily according to the optimal value function

$$\pi^*(s) \in \arg\max_{a \in \mathcal{A}} Q^*(s, a).$$

4.1.3 Approximate dynamic programming

In practice, the state space $\mathcal{S}$ is often too large to allow for exact representation of value functions. Approximate dynamic programming therefore replaces exact Bellman updates with a function approximation step, typically implemented using supervised learning techniques introduced in Section 2.2.1. This yields the classical approximate value iteration (AVI) scheme

$$Q_{n+1} \approx \mathcal{A}_{pprox}\, \mathcal{B}Q_n,$$

where $\mathcal{A}_{pprox}$ denotes an approximation operator.

Approximate value iteration alternates between applying the Bellman optimality operator and projecting the result back into a restricted function class. Under standard assumptions, if the approximation error at each iteration is bounded by ε in L^2_μ norm, where μ is a distribution over state–action pairs, approximate value iteration converges to a neighborhood of the optimal state-action value function, with a radius that scales with $\varepsilon/(1-\gamma)$ [24, 122, 123]. Additionally, greedy policies induced by such approximate iterates remain near-optimal.

Each iteration of AVI requires computing the Bellman update $\mathcal{B}Q_n$ over the entire state–action space, which in turn assumes either access to the transition model or exhaustive enumeration over all states. In many practical settings, neither is available: transitions are revealed only sequentially through interaction with the environment, and the agent must learn from individual experiences as they arise. This motivates a shift from global updates to incremental, sample-based learning. To derive such updates, value estimation is cast as a stochastic optimization problem. Concretely, approximating the state-action value function $Q(s, a)$ amounts to minimizing the mean squared error

$$L(Q) = \int_{\mathcal{S}\times\mathcal{A}} \left[Q(s, a) - \mathbb{E}(G^\pi(s, a))\right]^2 \, ds\, da.$$

In practice, this loss is estimated from transitions sampled from the environment. Rather than computing exact expected returns, one introduces the bootstrapped target

$$g(s, a) = \mathcal{R}(s, a, s') + \gamma \max_{a' \in \mathcal{A}} Q(s', a'),$$

to estimate $G^\pi(s,a)$, and applies stochastic gradient descent. The key idea behind bootstrapping is to replace the unknown true return $G^\pi(s,a)$, which depends on the entire future trajectory, with a partial estimate constructed from a single observed reward and the current value estimate at the next state. In doing so, bootstrapping trades variance for bias: Monte Carlo estimates of G^π, obtained by summing rewards along full rollout trajectories, are unbiased but exhibit high variance since they depend on the full sequence of future rewards; the bootstrapped target instead relies on a potentially biased estimate $Q(s',a')$ but requires only a single transition to compute.

In the tabular case[1], the gradient $\nabla_Q Q(s,a)$ reduces to a unit vector selecting the entry (s,a), yielding the classical temporal-difference (TD) update

$$Q(s,a) \leftarrow Q(s,a) + \alpha\,\delta, \qquad \delta = \mathcal{R}(s,a,s') + \gamma \max_{a' \in \mathcal{A}} Q(s',a') - Q(s,a),$$

introduced in [162]. This update rule, known as Q-learning, adjusts the current estimate by a fraction α of the temporal difference error δ, which measures the discrepancy between the one-step bootstrapped target and the current value estimate. Although successive samples collected along online trajectories are correlated, which violates the independence assumptions underlying standard stochastic approximation convergence guarantees, convergence to the optimal value function can still be established provided that all state–action pairs are visited infinitely often and that the learning rates satisfy Robbins–Monro conditions [144].

4.1.4 Deep Q-Networks

The tabular Q-learning update derived above assumes that each state–action pair can be stored and updated independently. This becomes infeasible in large, high-dimensional, or continuous state spaces, where generalization across states is necessary. Deep Q-Networks (DQN) [120] address this by parameterizing the state-action value function with a neural network $Q_\theta : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, trained via temporal-difference regression over observed (s,a,r,s') transitions.

Naively combining Q-learning with deep function approximation, introduces two sources of instability. First, as highlighted in the previous section, consecutive transitions collected by the agent are temporally correlated, violating the independence assumptions underlying stochastic gradient descent. Second, since the bootstrapped target itself depends on the current parameters θ, each gradient step

[1] That is, when Q is represented as a lookup table with one independent parameter per state–action pair.

shifts both the prediction and the target simultaneously, inducing a moving-target issue that can cause oscillation or divergence.

DQN introduces two mechanisms to mitigate these issues. To break temporal correlations, transitions experienced by the agent are stored in a *replay buffer* [110] and sampled uniformly at random for training, a technique known as experience replay. During data collection, exploration is ensured through an ε-greedy policy, which selects a random action with probability ε and otherwise acts greedily with respect to Q_θ. To stabilize regression targets, a separate target network with parameters θ^- is maintained and updated only periodically. For a transition (s, a, r, s'), the learning target becomes

$$r + \gamma \max_{a' \in \mathcal{A}} Q_{\theta^-}(s', a'),$$

and θ is updated by minimizing the squared TD error against this fixed target.

Together, these design choices enable DQN to learn control policies directly from high-dimensional inputs, achieving strong performance on the Atari benchmark suite [119]. Nonetheless, the combination of nonlinear function approximation, bootstrapping, and off-policy learning departs from the assumptions under which Q-learning is known to converge, and DQN remain sensitive in practice to architectural choices, hyperparameters, and random initialization.

Subsequent works have proposed targeted refinements to address each remaining weaknesses. Double DQN mitigates the overestimation bias induced by the max operator, which tends to select actions whose value is overestimated due to approximation error, by decoupling action selection from evaluation in the target computation. Prioritized experience replay [151] replaces uniform sampling in the replay memory with a distribution that favors transitions with large TD errors, focusing updates on more informative experiences at the cost of introducing bias in the gradient estimates [105]. Multi-step targets replace the one-step bootstrap with n-step returns, incorporating longer-horizon reward information and reducing the bias inherent to purely myopic updates [124]. These and other refinements are combined in the Rainbow framework [82], which integrates several orthogonal improvements into a single algorithm.

4.1.5 Policy gradient methods

The value-based methods introduced above derive optimal policies implicitly: once a good approximation of Q^* is available, the policy is obtained by acting greedily, that is, by solving $\arg\max_{a \in \mathcal{A}} Q(s, a)$

at each state. However, this maximization step can itself be challenging, particularly when the action space is large, continuous, or combinatorial. An alternative class of approaches consists in parameterizing the policy directly and optimizing its parameters so as to maximize expected return, thus bypassing the need for explicit action-space optimization. These methods are collectively referred to as *policy gradient* methods.

We consider a family of stochastic policies π_θ parameterized by θ where $\pi_\theta(a|s)$ denotes the probability of selecting action a in state s. The performance objective is defined as the expected return of trajectories generated by π_θ under the initial distribution p_0,

$$J(\theta) = \mathop{\mathbb{E}}_{s_0 \sim p_0} \left[V^{\pi_\theta}(s_0) \right] = \mathop{\mathbb{E}}_{\substack{\tau \sim \pi_\theta \\ s_0 \sim p_0}} \left[\sum_{t=0}^{+\infty} \gamma^t r_t \right],$$

where $\tau = (s_0, a_0, r_0, s_1, \dots)$ denotes a trajectory sampled according to the policy and environment dynamics. The goal is to find parameters θ^* maximizing $J(\theta)$.

A central result underlying policy gradient methods is the policy gradient theorem [163], which provides an explicit expression for the gradient of $J(\theta)$:

$$\nabla_\theta J(\theta) = \mathop{\mathbb{E}}_{\substack{\tau \sim \pi_\theta \\ s_0 \sim p_0}} \left[\sum_{t=0}^{+\infty} \nabla_\theta \log \pi_\theta(a_t|s_t)\, Q^{\pi_\theta}(s_t, a_t) \right].$$

This identity shows that the gradient of the expected return can be estimated from sampled trajectories, without requiring knowledge of the transition dynamics. Intuitively, the gradient weights the log-probability of each action by the quality of the resulting outcome: actions yielding returns above expectation see their probability increased, while those yielding below expectation returns are downgraded.

A direct Monte Carlo instantiation of the policy gradient theorem yields the REINFORCE algorithm [176]. Given a sampled trajectory $\tau = (s_0, a_0, r_0, \dots, s_T, a_T, r_T)$ of length $T+1$, the gradient is approximated by replacing the state-action value function with the empirical return $G_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$:

$$\nabla_\theta J(\theta) \approx \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, G_t.$$

Parameters are then updated by stochastic gradient ascent,

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

with step size $\alpha > 0$. While unbiased, this estimator typically exhibits high variance, which can substantially slow down learning. A classical variance-reduction technique consists in subtracting a baseline $b(s_t)$ that does not depend on the action:

$$\nabla_\theta J(\theta) \approx \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t) \left(G_t - b(s_t)\right).$$

Choosing $b(s_t) = V^{\pi_\theta}(s_t)$ yields an estimator proportional to the advantage function

$$A^{\pi_\theta}(s_t, a_t) = Q^{\pi_\theta}(s_t, a_t) - V^{\pi_\theta}(s_t),$$

and forms the basis of actor-critic methods, in which two function approximators are learned jointly: an *actor* π_θ that selects actions, and a *critic* V_ϕ that estimates the state value function. The critic serves as a learned baseline, replacing Monte Carlo returns with bootstrapped value estimates and thereby reducing variance at the cost of introducing bias, mirroring the same tradeoff encountered in temporal-difference methods.

Policy gradient methods naturally accommodate stochastic policies and continuous action spaces, and directly optimize the expected return without relying on greedy action selection. However, their reliance on sampled returns or learned critics typically results in higher variance and lower sample efficiency compared to value-based methods. In practice, modern reinforcement learning frameworks frequently combine policy gradients with value function approximation and variance-reduction techniques to balance stability, expressiveness, and data efficiency.

4.2 Contemporary challenges in reinforcement learning

Despite substantial empirical progress over the past decade, reinforcement learning continues to face persistent algorithmic and statistical challenges. These challenges are not simply temporary engineering setbacks. They stem from the intrinsic structure of the learning problem, and become particularly acute in domains such as combinatorial optimization, where the state-action space is vast, rewards are sparse, trajectories are long, and the environment appears highly non-stationary from the agent's perspective, as the agent itself continually alters the distribution of visited states. With these domain-specific difficulties in mind, this section reviews the core challenges facing reinforcement learning, and briefly discuss how they hinder the deployment of RL algorithms to real-world complex decision-making problems.

4.2.1 Credit assignment

A central challenge in RL is that the actions taken by the agent may induce outcomes that only materialize after many subsequent steps in the environment. Learning therefore implies assigning delayed credit (or blame) to earlier decisions. In value-based methods this is mediated through bootstrapping and temporal-difference (TD) errors, in policy-gradient methods, it is handled by return-weighted gradients, together with advantage estimation to reduce variance [163]. However, both approaches become brittle when episodes are long and rewards are sparse: return estimates exhibit high variance, while TD propagates reward signals only gradually through the trajectory, slowing the learning process and making optimization sensitive to reward scaling and discounting.

In combinatorial optimization, credit assignment issues arise from the fact that meaningful performance indicators, such as total solving time, B&B tree size or final objective value, typically become available only at the end of an episode, making it difficult to pinpoint which actions were responsible for success or failure. This issue is particularly acute in branch-and-bound, as starting from untrained policy networks induces exceedingly poor branching decisions, which in turn produce exceedingly large B&B trees. Hence, unlike many standard reinforcement learning benchmarks where early failures lead to rapid episode termination, in branch-and-bound poor action policy instead generates extremely long trajectories, often growing exponentially with problem size, thereby rendering learning both computationally costly and statistically challenging.

4.2.2 Function approximation

Modern RL relies on function approximation to generalize across large state-action spaces. When the approximation is poor, the agent may generalize incorrectly, leading to systematic errors that compound over time. This is particularly visible in value-based methods, where approximation errors can be propagated and amplified through Bellman updates. Therefore, to foster generalization, the inductive bias of the function approximator must match the structure of the problem.

In combinatorial optimization, the observation function is tasked with encoding complex combinatorial structures, including graphs, constraint matrices, incumbent solutions, bounds, partial solution assignments, along with solver-specific features describing the global state of the optimization process. In this context, B&B trees represent complex, dynamically evolving objects, for which principled,

tractable observation functions are yet to achieve broad adoption in the literature. Consequently, in order to make predictions, existing IL and RL approaches typically restrict state observations to the MILP associated with the current B&B node, using the bipartite graph representation from [62].

However, as discussed in Section 2.4.2, standard GNNs architectures such as the one used in Gasse et al. [62] exhibit inherent expressivity limitations. In particular, they may fail to distinguish between MILP instances with different feasibility properties or strong branching scores, and, *a fortiori*, between instances with substantially different associated B&B tree sizes. The combination of partial observability and limited representational power therefore restricts both training and generalization performance of GNN-based RL agents trained on bipartite MILP graph observations, imposing a fundamental ceiling on the quality of branching policies that can be learned within such framework.

4.2.3 Exploration–exploitation tradeoff

RL agents must trade off exploiting actions that appear good under their current estimates with exploring alternatives that may yield higher long-term returns. Unlike supervised learning, this tradeoff is intrinsic to RL because the agent's behavior determines the data it observes: exploration is not only a means of improving performance, but also the mechanism by which the agent reduces uncertainty about action values and discovers new regions of the state space. A purely greedy strategy can therefore become self-confirming, over-committing to actions that look good early due to limited data or estimation noise, while failing to gather the evidence needed to identify better choices.

In practice, exploration is often implemented through stochasticity mechanisms that do not explicitly model uncertainty, but ensure sufficient action diversity during learning. The standard example is the ε-greedy strategy used in value-based methods: with probability ε the agent selects an action uniformly at random, and with probability $1-\varepsilon$ it exploits by choosing a greedy action $\arg\max_a Q(s,a)$. Typically, ε is annealed throughout training, to transition from broad exploration ($\varepsilon = 1$) to near-greedy behavior ($\varepsilon \to 0$). A smoother alternative is Boltzmann (softmax) exploration, where actions are sampled according to

$$\pi(a|s) \;\propto\; \exp\!\left(\frac{Q(s,a)}{\tau}\right),$$

for a temperature parameter τ. Related ideas appear in policy-gradient and temporal difference methods via entropy regularization [75, 169], which explicitly encourages stochastic policies by adding an entropy bonus to the objective, thereby fostering exploration by delaying premature convergence

to deterministic behavior.

While simple, broadly applicable, and easy to tune, these exploration schemes are agnostic to which actions are informative, and can therefore be inefficient in large action spaces or sparse-reward settings, where undirected random deviations rarely uncover useful learning signal — a limitation amplified in branch-and-bound by the large number of candidate branching variables and the delayed nature of reward feedback.

4.2.4 Policy improvement

Policy improvement is the central iterative step in reinforcement learning: given a current policy π, the agent updates it to increase expected return under the environment dynamics. In principle, this is straightforward: estimate which actions are better, then shift probability mass toward them. In practice, the objective being optimized is both nonstationary and coupled to the current data distribution. Specifically, the gradient of the expected return, $\nabla J(\pi)$, depends on the state–action distribution induced by the current policy π. Since each policy update modifies both π and its induced state-action distribution, gradient estimates computed from samples collected under a previous policy may be biased with respect to the current state-action distribution, and acting on such biased estimates can degrade rather than improve the policy, especially when value estimates are themselves inaccurate or exhibit high variance.

A second difficulty is that policy updates are typically driven by imperfect value estimates. In value-based methods, policy improvement is performed by extracting a greedy or near-greedy policy from an approximate Q function. Even small approximation errors can thus perturb the ranking of actions among many near-ties, causing the resulting policy to steer the agent into regions of the state space where value estimates are least reliable, thereby amplifying the initial error. This tight coupling between estimation error and distribution shift is a major source of instability and occasional performance collapse. In actor-critic methods, an analogous issue arises because the actor is updated using advantage estimates provided by a learned critic: inaccuracies in the value function translate directly into biased policy gradients, and repeated bootstrapping can create feedback loops in which small systematic errors compound over successive updates.

In combinatorial optimization and mixed-integer linear programming, policy improvement is further complicated by the extreme sensitivity of the underlying search process to early decisions. Small

policy changes, such as selecting a different branching variable near the root, can reshape the entire search tree, inducing abrupt distribution shift and rendering previously accurate value estimates unreliable. This sensitivity is exacerbated by reward sparsity, which provides limited intermediate feedback for diagnosing and correcting such shifts. Finally, these difficulties are further compounded by large state–action spaces, heavy reliance on function approximation, partial observability of solver internals, and expensive, high-variance evaluation.

Taken together, these challenges help explain why reinforcement learning often struggles to translate benchmark successes into real-world reliable gains, especially in the context of combinatorial optimization. In the next section, we review how the literature has attempted to address these obstacles in the context of branch-and-bound, and discuss the main modeling choices and algorithmic compromises proposed to keep learning tractable.

4.3 Reinforcement learning for variable selection in branch-and-bound

Having introduced the fundamentals of reinforcement learning, as well as some of its general limitations, we now turn to describing its recent application to learning to branch.

4.3.1 TreeMDP

Since branching decisions are made sequentially, reinforcement learning appears as a natural candidate to learn good branching policies. Building on the work of He et al. [79], who first introduced a Markov decision process formulation for node selection in branch-and-bound, Etheve et al. [56] and Scavuzzo et al. [149] proposed to model branch-and-bound episodes as TreeMDPs. Under this formulation, B&B states consist in triplets $s_i = (P_i, x^*_{LP,i}, \bar{x}_i)$, where P_i is the MILP associated with current node v_i, $x^*_{LP,i}$ is the solution to its linear relaxation, and $\bar{x}_i$ denotes the incumbent solution at the time when v_i is selected for expansion. The actions available at s_i are the set of fractional variables in $x^*_{LP,i}$. Given (s_i, a_i) the tree Markov transition function produces two child node states (s_i^-, s_i^+) that can be visited in any order. Additionally, aiming to learn branching policies directly minimizing the tree size objective defined in Eq. (1.3), both Etheve et al. [56] and Scavuzzo et al. [149] have opted for a constant negative reward model $r_i = -1$, together with undiscounted episodes, taking $\gamma = 1$.

The formulation outlined above, however, is not sufficient to fully characterize TreeMDPs. In fact, for a branch-and-bound episode to qualify as a TreeMDP, it must satisfy the *tree Markov* property.

Definition 4.3.1. (Tree Markov property) Branch-and-bound episodes $(s_0, a_0, ..., s_T, a_T)$ are said to satisfy the tree Markov property, if and only if, for every non-leaf state node s_i, the global upper bounds GUB_i^- and GUB_i^+ associated with child state nodes s_i^- and s_i^+ respectively can be derived solely from the current state-action pair (s_i, a_i).

Etheve et al. [56] and Scavuzzo et al. [149] identified two distinct sufficient conditions under which a branch-and-bound episode satisfies the tree Markov property.

Proposition 4.3.2. If either of the following condition holds:

(i) $GUB_0 = GUB^*$, *i.e.*, the incumbent solution at the root node is optimal ($\bar{x}_0 = x^*$), or

(ii) the node selection policy ρ implemented throughout the search process is depth-first search,

then the resulting branch-and-bound trajectory $(s_0, a_0, \ldots, s_T, a_T)$ satisfies the tree Markov property, and therefore defines a TreeMDP.

The TreeMDP formulation offers several compelling properties. First, it enables leveraging the bipartite graph representation introduced by Gasse et al. [62] to efficiently encode TreeMDP states s_i, and to use graph neural networks to approximate either or both value functions and policy distributions. Second, Etheve et al. [56] and Scavuzzo et al. [149] have shown that, under the tree Markov property, episode trajectories can effectively be decomposed into independent B&B subtrees. This decomposition allows policies to be learned by minimizing the size of each subtree independently rather than optimizing with respect to the full B&B trajectory, thereby alleviating the credit assignment issues induced by long episode trajectories. Etheve et al. [56] and Scavuzzo et al. [149] have thus proposed ad hoc temporal-difference and policy-gradient updates tailored to TreeMDPs, enabling, under DFS, the adaptation of standard reinforcement learning algorithms to the branch-and-bound framework. Their resulting branching agents, based respectively on DQN and REINFORCE, managed to learn competitive variable selection policies outperforming that of SCIP, though without surpassing the performance achieved by Gasse et al. [62]'s IL baseline.

4.3.2 Retro branching

Subsequently, Parsonson et al. [132] sought to overcome the computational limitations imposed on TreeMDP by the theoretical requirement to restrict the choice of node selection policy to DFS. Assuming that RL branching agents trained following advanced node selection strategies would perform better than their DFS counterparts, they proposed to learn from retrospective trajectories, diving trajectories built from original TreeMDP episodes. Their core intuition was that, by enforcing hierarchical tree consistency over temporal consistency between successive B&B node states, retro branching enables learning from trajectories whose future states are determined by the subtree structure alone, independently of the node selection policy. In fact, Parsonson et al. [132] found retrospective trajectories to alleviate the partial observability induced by the "disordered" exploration of the tree under non-DFS node selection policies, and outperform prior RL agents on set covering instances, though they still fall short of IL approaches.

More generally, a large body of work has proposed to learn, either by imitation or reinforcement, better-performing B&B heuristics outside of variable selection [125, 134]. RL contributions in primal search [156, 179], node selection [55, 79], and cut selection [155, 164, 173] have broadly relied on the TreeMDP framework to train their agents, simply adapting the action set to the task at hand. In line with the trend observed for variable selection, empirical results in these areas consistently show IL approaches outperforming RL baselines. Yet, if the performance of IL heuristics are generally caped by that of the suboptimal experts they learn from, the performance of RL agents are, in theory, only bounded by the maximum score achievable. We note that in order to cope with dire credit assignment problems induced by the use of a constant reward model, prior research efforts have shifted away from the traditional Markov decision process framework, finding it impractical for learning efficient policies in B&B. Instead, these works have prefered opting for unconventional MDP-inspired formulations such as TreeMDP to model sequential decision making in B&B. As a result, these approaches often rely on modified, approximate learning rules adapted from traditional RL algorithms, which erode the theoretical basis for asymptotic optimality.

In the next chapter, we investigate the conditions under which variable selection in branch-and-bound can be effectively cast as a Markov decision process, with the aim of unlocking the application

of the full range of reinforcement learning frameworks for the purpose of learning optimal variable (and other) strategies in mixed-integer linear programming.

Chapter 5

A Markov decision process for variable selection in branch and bound

Content

Building on the works of Gasse et al. [62] and He et al. [79], several previous contributions succeeded in learning efficient branching strategies by reinforcement [56, 132, 149], without surpassing the performance achieved by the IL approach from Gasse et al. [62]. Yet, as discussed in Chapter 4, the performance of IL branching agents is ultimately bounded by that of vanilla strong branching, which is known to be suboptimal in the general case [16, 46]. In principle, RL agents should therefore be able to match and potentially exceed the performance achieved by neural networks trained to mimick strong branching behaviour.

In this chapter, we show that despite improving the empirical performance of RL baselines, both TreeMDP and retrospective trajectory formulations introduce approximations of the B&B dynamics that undermine the asymptotic performance achievable by RL branching agents in the general case. In order to remedy this issue, we propose reverting to a principled vanilla MDP formulation of the branch-and-bound process, which preserves convergence properties brought by previous contributions without sacrificing optimality. Our formulation, which we refer to as branch-and-bound Markov decision process (BBMDP), see Section 5.1, allows to define a proper Bellman optimality operator, which, in turn, enables to unlock the full potential of state-of-the-art approximate dynamic programming algorithms [81] for the purpose of learning optimal B&B variable selection strategies.

Furthermore, inspired by recent works in RL advocating training value networks via classification instead of regression in order to foster both scalability and generalization [41, 57], we introduce in Section 5.2 a histogram classification loss tailored to the branch-and-bound setting. We evaluate the resulting DQN-BBMDP agent on the Ecole benchmark in Section 5.4, achieving state-of-the art performance among RL agents while narrowing the gap with the IL approach from Gasse et al. [62].

While our agent strongly improves over prior RL-based approaches, it still falls short of the performance achieved by IL methods. A natural hypothesis is that IL benefits from a strong inductive bias, effectively inheriting from strong branching demonstration, whereas RL must recover this structure from sparse reward signals. Building on this insight, in Section 5.5, we investigate training DQN-BBMDP following an alternative reward model designed to closely align with the dual gap reduction objective explicitly optimized in strong branching. In contradiction with the original hypothesis, our study shows that dual gap reduction based RL branching agent underperfom the original ones, indicating that failure to converge to optimality stems from intrinsic reinforcement learning challenges described in Section 4.2, rather than from the choice of a sparse reward model in BBMDP.

We mention that the work presented in Sections 5.1, 5.2, 5.3 and 5.4 has led to a publication in the *Proceeding of the 38th Conference on Advances in Neural Information Processing Systems* [160], while the results presented in Section 5.5 are original to this thesis.

5.1 Markov decision process formulation

By using the current B&B node as the observable state, prior attempts to learn optimal branching strategies have relied on the TreeMDP formalism to train RL agents. However, TreeMDPs are not MDPs, as they do not define a Markov process on the state random variable (for instance, a transition yields two states and is hence not a Markov process on the state variables). As a result, this forces Etheve et al. [56] and Scavuzzo et al. [149] to redefine Bellman updates and derive *ad hoc* convergence theorems for temporal difference, value iteration, and policy gradient algorithms. In order to leverage broader theoretical results from the reinforcement learning literature, we propose describing variable selection in B&B as a proper Markov decision process.

5.1.1 Definition

The problem of finding an optimal branching strategy according to Eq. (1.3) can be described as a regular deterministic Markov decision process. To this end, we introduce branch-and-bound Markov decision processes (BBMDP), taking $\gamma = 1$ since episodes horizons are bounded by the (finite) largest possible number of nodes.

State-action space. $\mathcal{S}$ is the set of all B&B trees $s_t = (\mathcal{V}_t, \mathcal{E}_t, \bar{x}_t)$ that can be built following an arbitrary node selection policy ρ. We stress that this set includes intermediate B&B trees, whose incumbent solutions $\bar{x}_t$ are yet to be proven optimal. $\mathcal{A}$ is the set of all integer variables indices $\mathcal{I}$.

Transition function. The Markov transition function is defined as $\mathcal{T} = \kappa_\rho$ with κ_ρ the branching operation described in Section 1.2.2. Note that if the variable associated with a_t is not fractional in x^*_{LP}, then $s_{t+1} = \mathcal{T}(s_t, a_t) = s_t$ as relaxing a variable that is not fractional has no impact on the LP relaxation. Importantly, all states for which $\mathcal{O} = \emptyset$ are terminal states.

Starting states. Initial states are single node trees, where the root node is associated to a MILP P_0 drawn according to the distribution p_0 defined in Section 1.2.2 (hence the use of p_0 for both the initial

problem P_0 and the MDP's initial state s_0).

Reward model. We define $\mathcal{R}(s, a) = -2$ for all transitions until episode termination. Since each transition adds two B&B nodes, the overall value of a trajectory is the opposite of the number of nodes added to the B&B tree from the root node, which aligns with the definition of Eq. (1.3).

Unlike in TreeMDP, in BBMDP the current state is defined as the state of the entire B&B tree, rather than merely the current B&B node. The transition function returns a B&B tree whose open nodes are sorted according to the node selection policy ρ, thus reflecting the true dynamics of the B&B algorithm, instead of a couple of pseudo-states associated with the child nodes of the last node expansion. Note that the definition above sets BBMDPs among the specific class of MDPs called stochastic shortest path problems [140].

Although BBMDP provides a principled MDP formulation for variable selection in B&B, it imposes at first severe practical constraints: since optimal policies appear to be functions of the entire B&B state, learning them seemingly requires (i) an observation function capable of encoding the full B&B search tree, along with the solver associated internal memory; and (ii) a function approximator able to operate on such complex, structured evolving objects. While the derivation of such observation function and function approximator is conceptually feasible, for example by representing the search state as a B&B hypertree in which each B&B nodes is itself encoded as a MILP bipartite graph, this approach quickly becomes computationally prohibitive in practice owing to the substantial redundancy inherent to such representations. Therefore, in the next section, we propose to exploit the tree structure of BBMDP to rewrite Bellman recursive updates in terms of subtree value functions. Crucially, under depth-first search node selection policy, these subtree value functions can be defined on inputs of comparable complexity to those used in TreeMDP (i.e., MILP bipartite graphs), thereby circumventing the need to explicitly encode the full search tree.

5.1.2 Learning optimal branching strategies in BBMDP

Like in TreeMDP, B&B episodes can be decomposed in independent subtree trajectories to facilitate RL agents training. Consider a deterministic branching policy π, and let us rewrite V^π to exhibit its tree structure. Given an open node $o_i \in \mathcal{O}_t$, we note $T(o_i)$ the subtree rooted at o_i, and define $\bar{V}^\pi(s_t, o_i)$

the function that returns the opposite of the size of the subtree rooted at o_i, when branching according to π starting from s_t until episode termination. Then V^π can be expressed as:

$$V^\pi(s_t) = \sum_{o_i \in \mathcal{O}_t} \bar{V}^\pi(s_t, o_i). \tag{5.1}$$

In plain words, the total number of nodes that will be added to the B&B tree past s_t is equal to the sum of the sizes of all the subtrees $T(o_i)$ rooted in the open nodes of s_t. Because the full B&B tree is a complex object to manipulate, it is tempting to discard the tree structure in s_t and define $\bar{V}$-value functions merely as functions of $(o_i, \bar{x}_{o_i})$, rather than functions of (s_t, o_i), where $\bar{x}_{o_i} \in \mathbb{R}^n$ is the incumbent solution when o_i is processed by the B&B algorithm. The rationale for such value functions is that the size of the subtree rooted at $o_i \in \mathcal{O}_t$, for a given incumbent solution $\bar{x}_{o_i}$, should be the same, regardless of the parents of o_i, its position in the tree, or the branching decisions taken in the other open subtrees $T(o_j)$ for $j \neq i$. It turns out, this last statement does not always hold, quite counter-intuitively. Let us write τ_i the time steps at which the nodes $o_i \in \mathcal{O}_t$ are selected by the node selection strategy ρ.[1] Now, consider for instance a node selection procedure ρ that performs a breadth-first search through the tree. The number of nodes in $T(o_i)$ will depend strongly on whether an improved incumbent solution $\bar{x}_{o_i}$ was found in the subtrees explored between s_t and s_{τ_i}, and, in turn, on the branching decisions taken in these subtrees. This example highlights the major importance of the node selection strategy ρ, when one wishes to define subtree sizes based solely on $(o_i, \bar{x}_{o_i})$.

Consider now two open nodes o_i and o_j in $\mathcal{O}_t$. Conversely to the previous example, if one can guarantee that the subtree rooted at o_j will be solved to optimality before o_i is considered for expansion in the B&B process, then the number of nodes in $T(o_i)$ will not be affected by the branching decisions taken at any node under o_j. In fact, if o_j is solved to optimality, $\bar{x}_{o_i}$ will either not change ($\bar{x}_{o_i} = \bar{x}_t$) if no feasible solution in $T(o_j)$ improves GUB, or either be the best feasible solution of the MILP associated with o_j, which strictly does not depend on the series of actions taken in $T(o_j)$. In other words, to make sure that the size of $T(o_i)$ does only depend on the branching decisions taken in $T(o_i)$, all nodes $o_j \in \mathcal{O}_t$ must have been either fully explored or strictly unexplored at τ_i. Applying this argument recursively induces that the only node selection strategy which enables predicting a subtree size only based on $(o_i, \bar{x}_{o_i})$, is a depth-first search (DFS) exploration of the B&B tree. The same observation was made by Etheve et al. [56] and Scavuzzo et al. [149] previously. Therefore, in the

[1] Following our indexation of $o_i \in \mathcal{O}_t$, we have $t = \tau_1 < ... < \tau_i < ... < \tau_{|\mathcal{O}_t|}$.

following, we consider $\rho = DFS$ and write $\bar{V}^\pi(o_i, \bar{x}_{o_i})$ the opposite of the size of $T(o_i)$ in this context. We can now derive a refined Bellman update to train branching agents in BBMDP.

Proposition 5.1.1. In DFS-BBMDP, the Bellman equation $V^\pi(s) = \mathcal{R}(s, \pi(s), s') + V^\pi(s')$ yields:

$$\bar{V}^\pi(o_1, \bar{x}_{o_1}) = -2 + \bar{V}^\pi(o'_1, \bar{x}_{o'_1}) + \bar{V}^\pi(o'_2, \bar{x}_{o'_2}), \tag{5.2}$$

with $o_1 = \rho(s)$, $o'_1 = \rho(\mathcal{T}(s, \pi(s))$, and o'_2 is the sibling of o'_1 in the B&B tree.

Proof. Eq. (5.2) follows directly from injecting Eq. (5.1) in the Bellman equation, and observing that most terms in the sums simplify as $\bar{V}^\pi(o_i, \bar{x}_{o_i}) = \bar{V}^\pi(o'_{i+1}, \bar{x}_{o'_{i+1}})$ for $i \geq 2$. □

Keeping the same notation convention for o_1, o'_1 and o'_2, we define

$$\bar{Q}^\pi(o_1, \bar{x}_{o_1}, a) = -2 + \bar{V}^\pi(o'_1, \bar{x}_{o'_1}) + \bar{V}^\pi(o'_2, \bar{x}_{o'_2}). \tag{5.3}$$

Analogous to $\bar{V}^\pi$, the $\bar{Q}^\pi$ function returns the opposite of the size of the subtree rooted at $o_i \in \mathcal{O}_t$ when branching on action $a \in \mathcal{A}$ at o_i and following policy π until $T(o_i)$ is fathomed. Note that if $\bar{V}^\pi$ and $\bar{Q}^\pi$ are not strictly value functions[2], they naturally emerge when applying Bellman equations to BBMDP value functions under $\rho = DFS$. We stress that, in depth-first search BBMDPs, it is not necessary to learn Q^* to derive π^*, since acting according to a policy minimizing the size of the subtree rooted in the current B&B node is equivalent to acting according to a global optimal policy:

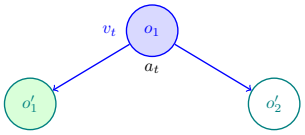
$$\pi^*(s) = \arg\max_{a \in \mathcal{A}} Q^*(s,\, a) = \arg\max_{a \in \mathcal{A}} \bar{Q}^*(o_1, \bar{x}_{o_1}, a). \tag{5.4}$$

5.1.3 Approximate dynamic programming

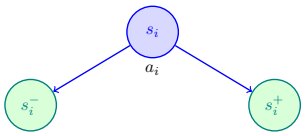
The previous properties enable learning π^* by training a neural network to approximate $\bar{Q}^*$ using traditional temporal difference algorithms. Notably, $\bar{Q}^*$ is easier to learn than Q^*, as it relies solely on quantities observable at time t, whereas the previous decomposition of Q^* depends on all $\bar{x}_{o_i}$ for $o_i \in \mathcal{O}_t$, which are not known at time t. Moreover, $\bar{Q}^*$ trains on much shorter trajectories than Q^*, which helps mitigate credit assignement issues.[3]

[2] In particular, they are not define on $\mathcal{S}$ and $\mathcal{S} \times \mathcal{A}$ respectively.

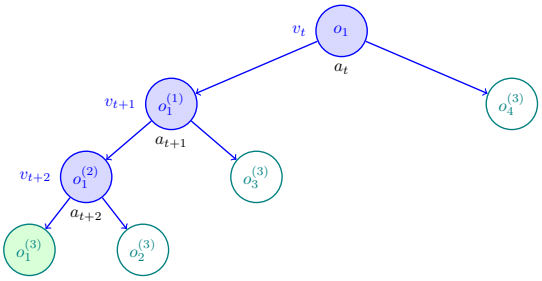
[3] On average, the length of subtree trajectories is logarithmically shorter than the length of BBMDP episodes.



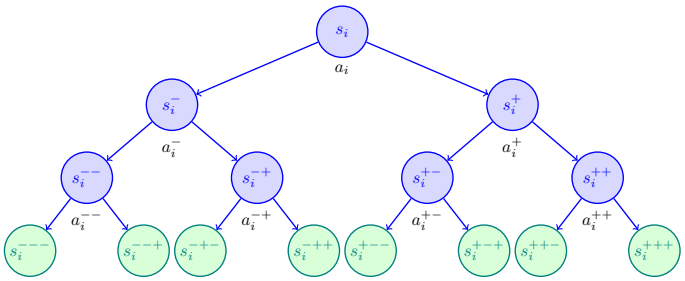
(a) 1-step BBMDP Bellman equation:
$\bar{V}^\pi(o_1, \bar{x}_{o_1}) = -2 + \bar{V}^\pi(o'_1, \bar{x}_{o'_1}) + \bar{V}^\pi(o'_2, \bar{x}_{o'_2})$



(b) 1-step TreeMDP Bellman equation:
$V^\pi(s_i) = -2 + V^\pi(s_i^-) + V^\pi(s_i^+)$



(c) k-step BBMDP Bellman equation:
$\bar{V}^\pi(o_1, \bar{x}_{o_1}) = -2k + \sum_{i=1}^{k+1} \bar{V}^\pi(o_i^{(k)}, \bar{x}_{o_i^{(k)}})$



(d) k-step TreeMDP Bellman equation:
$V^\pi(s_0) = -2^1 - 2^2 - ... - 2^k + \sum_{i=1}^{2^k} V^\pi(s_i^j)$

Figure 5.1: When minimizing 1-step temporal difference, TreeMDP and BBMDP yield equivalent learning updates, as shown in 5.1a, 5.1b. However, over k-step trajectories, Bellman updates associated with the two methods diverge, as shown in 5.1c, 5.1d.

Consider a transition (s, a, r, s'). Following Eq. (5.2), the Bellman optimality equation yields a sequence of $\bar{Q}_n$ functions via dynamic programming updates of the form:

$$\bar{Q}_{n+1}(o_1, \bar{x}_{o_1}, a) = -2 + \max_{a' \in \mathcal{A}} \bar{Q}_n(o'_1, \bar{x}_{o'_1}, a') + \max_{a'' \in \mathcal{A}} \bar{Q}_n(o'_2, \bar{x}_{o'_2}, a''). \tag{5.5}$$

One can also consider a k-step Bellman operator ($k \geq 1$), generalizing Eq. (5.5). Let π be a policy, and $s^{(k)}$ the state reached after a first application of a from s, and $k-1$ subsequent applications of π. Provided the subtree rooted at o_1 has not been fathomed within these k time steps, $s^{(k)}$ has $k+1$ new open nodes which we label $o_i^{(k)}$. Then, the Bellman update becomes:

$$\bar{Q}_{n+1}(o_1, \bar{x}_{o_1}, a) = -2k + \sum_{i=1}^{k+1} \max_{a' \in \mathcal{A}} \bar{Q}_n(o_i^{(k)}, \bar{x}_{o_i^{(k)}}, a'). \tag{5.6}$$

This Bellman update can be approximated using a graph neural network $\bar{Q}_\theta$, whose weights are trained to minimize the k-step temporal difference loss

$$\mathcal{L}_{reg}(\theta) = \mathbb{E}\left[\left(\bar{Q}_\theta(o_1, \bar{x}_{o_1}, a) - \bar{Q}_{n+1}(o_1, \bar{x}_{o_1}, a)\right)^2\right], \tag{5.7}$$

using a target network $\bar{Q}_{\theta^-}$ to bootstrap $\bar{Q}_n$-values in Eq (5.6). Importantly, both $\bar{Q}_\theta$ and $\bar{Q}_{\theta^-}$ functions accommodate standard MILP graph bipartite inputs.

5.1.4 BBMDP vs TreeMDP

Table 5.1: Side by side comparison of BBMDP and TreeMDP frameworks, complementing Figure 5.1.

	BBMDP	TreeMDP
MDP	Yes	No
State s	$s_t = (\mathcal{V}_t, \mathcal{E}_t, \bar{x}_t)$	$s_i = (P_i, x^*_{LP,i}, \bar{x}_i)$
Action a	$a_t \in \mathcal{I}$	$a_i \in \mathcal{I}$
Reward r	-2	-1
Next state s'	$s_{t+1} = (\mathcal{V}_{t+1}, \mathcal{E}_{t+1}, \bar{x}_{t+1})$	s_i^-, s_i^+
k-step next state $s^{(k)}$	$s_{t+k} = (\mathcal{V}_{t+k}, \mathcal{E}_{t+k}, \bar{x}_{t+k})$	s_i^1, s_i^2, ..., $s_i^{2^k}$
$\mathcal{B}(V^\pi) = V^\pi$	$\bar{V}^\pi(o_1, \bar{x}_{o_1}) = -2 + \bar{V}^\pi(o'_1, \bar{x}_{o'_1}) + \bar{V}^\pi(o'_2, \bar{x}_{o'_2})$	$V^\pi(s_i) = -2 + V^\pi(s_i^-) + V^\pi(s_i^+)$
$\mathcal{B}^k(V^\pi) = V^\pi$	$\bar{V}^\pi(o_1, \bar{x}_{o_1}) = -2k + \sum_{i=1}^{k+1} \bar{V}^\pi(o_i^{(k)}, \bar{x}_{o_i^{(k)}})$	$V^\pi(s_i) = -\sum_{j=1}^{k} 2^j + \sum_{j=1}^{2^k} V^\pi(s_i^j)$

TreeMDP provides an intuitive and computationally efficient framework for training RL agents to learn improved variable selection strategies in branch-and-bound. However, by relaxing core MDP assumptions, most notably the existence of a single successor state and a well-defined Markov transition kernel, TreeMDP fails to faithfully capture the dynamics of variable selection in B&B in the general case. This limitation is illustrated in Figure 5.1 and further highlighted in Table 5.1. While TreeMDP constitutes a valid approximation of BBMDP when learning from one-step temporal-difference targets, it induces TD targets that are inconsistent with the true B&B dynamics when extended to multi-step returns. In particular, defining a k-step TD target in TreeMDP is ill-defined with respect to the actual execution of the B&B algorithm. Since branching on an action a_i at state s_i produces two child states s_i^- and s_i^+, recursively applying the TreeMDP Bellman operator from Etheve [55] to V^π yields

$$V^\pi(s_i) = -\sum_{j=1}^{k} 2^j + \sum_{j=1}^{2^k} V^\pi(s_i^j),$$

which implicitly assumes a full B&B tree expansion of depth k, as depicted in Figure 5.1d for $k = 3$. Crucially, such trajectories cannot arise when solving a MILP with B&B under a DFS node selection

policy, since the subtree rooted at s_i^- must be completely fathomed before its sibling node s_i^+ can be considered for expansion. Therefore, the k-step TD target defined above relies on an approximation of the B&B dynamics that violates the solver's sequential execution. This mismatch fundamentally limits the performance attainable by multi-step TD methods within the TreeMDP framework, as verified empirically in the next section.

Another setting in which TreeMDP yields inconsistencies is that of planning in model-based reinforcement learning. In fact, in many modern RL frameworks, policy improvement is achieved by running a local search procedure within a simulator of the MDP, and ultimately implementing the action that maximizes estimated returns.[4] Throughout the simulation phase, actions are typically selected using upper bound confidence criteria like PUCT:

$$a^k = \arg\max_{a \in \mathcal{A}} \left[Q(s,a) + \pi(s,a) \cdot \frac{\sqrt{\sum_{b \in \mathcal{A}} N(s,b)}}{1 + N(s,a)} \left(c_1 + \log\left(\frac{\sum_{b \in \mathcal{A}} N(s,b)}{c_2}\right)\right)\right], \tag{5.8}$$

where $N(s,a)$ corresponds to the number of visits of the (s,a) pair, and c_1, c_2 are search hyperparameters. This formulation balances exploration and exploitation, guiding the search toward promising states while ensuring sufficient exploration of the local state-action space. However in TreeMDP, such exploration criteria lead to inconsistencies, as the exploration between node s_i^- and s_i^+ is balanced based on their estimated value. However, since both s_i^- and s_i^+ originate from the same branching decision a_i, differentiating between them for expansion is irrelevant. In B&B, branching on a_i implies that s_i^- and s_i^+ are both necessarily added to the B&B tree. Consequently, in TreeMDP, MCTS could guide the search towards a suboptimal action a_i based on the prediction that it produces a state s_i^- of small subtree size, while failing to account for the potentially enormous subtree size associated with s_i^+. This fundamental inconsistency hinders the effective application of model-based reinforcement learning algorithms within the TreeMDP framework.

In summary, by discarding core MDP concepts of temporality and sequentiality, TreeMDP fails to accommodate the full range of modern RL algorithms. In contrast, BBMDP leverages the results first established by Etheve et al. [56] and Scavuzzo et al. [149] — in DFS, minimizing the whole B&B tree size is achieved when any subtree is of minimal size (Eq. (5.4)) — all while preserving MDP properties. Hence, BBMDP allows to harness RL algorithms that are not compatible with the TreeMDP

[4]For a comprehensive introduction to MCTS and the model-based reinforcement learning paradigm, see Section 6.1.

framework, such as multi-step temporal difference, eligibility traces, or RL algorithms implementing planning-based policy improvement, for the purpose of learning optimal variable selection strategies. In the same fashion, BBMDP can be applied to augment the pool of RL algorithms available for learning improved cut selection and primal search heuristics, simply by adapting the action set and the reward model to the task at hand. Complete exploration and comparison of such algorithms is beyond the scope of this thesis, and left to future work, as our primary objective here is to study the conditions under which B&B search can be effectively cast as an MDP, and provide the community with a solid basis for principled algorithms.

5.2 A histogram classification loss for BBMDP

Having introduced a principled Markov decision process formulation for variable selection in branch-and-bound, and highlighted its theoretical benefits, we now propose additional refinements aimed at improving the empirical performance of DQN agents trained within this framework. This section introduces a classification-based loss for learning value functions in BBMDP, replacing standard mean squared error regression with a cross-entropy objective inspired by recent advances in value-based reinforcement learning to improve learning stability and generalization.

5.2.1 Background

As they investigated the uneven success met by advanced neural network architectures in supervised learning versus reinforcement learning, Farebrother et al. [57] found that training agents using a cross-entropy classification objective significantly improved the performance and scalability of value-based RL methods. However, adopting cross-entropy classification in place of mean squared error regression necessitates mappings transforming scalars into distributions and distributions back to scalars. Farebrother et al. [57] found the Histogram Gaussian loss (HL-Gauss) [87], which exploits the ordinal structure of the regression task by distributing probability mass on multiple neighboring histogram bins, to be a reliable solution across multiple RL benchmarks. Concretely, in HL-Gauss, the support of the value function $\mathcal{Z}_v \subset \mathbb{R}$ is divided in m_b bins of equal width forming a partition of $\mathcal{Z}_v$. Bins are centered at $\zeta_i \in \mathcal{Z}_v$ for $1 \leq i \leq m_b$, we use $\eta = (\zeta_{max} - \zeta_{min})/m_b$ to denote their width. Given a scalar $z \in \mathcal{Z}_v$, we define the random variable $Y_z \sim \mathcal{N}(\mu = z, \sigma^2)$ and note respectively ϕ_{Y_z} and Φ_{Y_z} its associate probability density and cumulative distribution function. The scalar z can then be

encoded into a histogram distribution on $\mathcal{Z}_v$ using the function $p_{hist} : \mathbb{R} \to \Delta^{m_b}$. Concretely, $p_{hist}(z)$ converts scalars to distributions by computing the aggregated mass of ϕ_{Y_z} on each bin of $\mathcal{Z}_v$:

$$p_{hist}(z) = (p_i(z))_{1\leq i\leq m_b} \text{ with } p_i(z) = \int_{\zeta_i-\frac{\eta}{2}}^{\zeta_i+\frac{\eta}{2}} \phi_{Y_z}(y)dy = \Phi_{Y_z}(\zeta_i + \frac{\eta}{2}) - \Phi_{Y_z}(\zeta_i - \frac{\eta}{2}). \tag{5.9}$$

Conversely, histogram distributions $(p_i)_{1\leq i\leq m_b}$ outputted by value networks can be converted to scalars by computing the expectation: $z = \sum_{i=1}^{m_b} p_i \cdot \zeta_i$.

5.2.2 Derivation

Adapting HL-Gauss to BBMDP is challenging because the value support spans several orders of magnitude: on the Ecole benchmark, $\mathcal{Z}_v$ typically spans over $[-10^6, -2]$. With equal-width binning, HL-Gauss forces a stark trade-off between resolution and coverage: choosing a small bin width to resolve typical values requires an exceedingly large number of bins to cover the support[5], whereas choosing a moderate number of bins makes the discretization too coarse over the region where learning is most sensitive. Since value functions predict the number of node of binary trees built with B&B, it seems natural to choose bins centered at $\zeta_i = -2^i$ to partition $\mathcal{Z}_v$. In order to preserve bins of equal size, we consider distributions on the support $\psi(\mathcal{Z}_v)$ with $\psi(z) = \log_2(-z)$ for $z \in \mathcal{Z}_v$, such that $\psi(\mathcal{Z}_v)$ is efficiently partitioned by bins centered at $\zeta_i = i$ for $1 \leq i \leq m_b$. Thus, in BBMDP, histograms distributions are given by $p_{hist}(z) = (p_i \circ \psi(z))_{1\leq i\leq m_b}$ for $z \in \mathcal{Z}_v$, and can be converted back to $\mathcal{Z}_v$ through $z = \sum_{i=1}^{m_b} p_i \cdot \psi^{-1}(\zeta_i)$ with $\psi^{-1}(z) = -2^z$.

Taking $\bar{Q}_\theta$ and $\bar{Q}_{\theta^-}$ to output vectors in $\mathbb{R}^{m_b}$ rather than scalars, we recover scalar training targets from Eq. (5.6) by mapping the bootstrapped histogram distribution predictions from $\bar{Q}_{\theta^-}$ back to $\mathcal{Z}_v$, and summing with the cumulative reward collected along the k-step trajectory:

$$\bar{Q}_{n+1}(o_1, \bar{x}_{o_1}, a) = -2k + \sum_{i=1}^{k+1} \max_{a'\in\mathcal{A}} \sum_{i=1}^{m_b} \text{softmax}\left(\bar{Q}_{\theta^-}(o_i^{(k)}, \bar{x}_{o_i^{(k)}}, a')\right) \cdot \psi^{-1}(\zeta_i). \tag{5.10}$$

Finally, we obtain a HL-Gauss cross-entropy loss tailored to the B&B setting

$$\mathcal{L}_{CE}(\theta) = \mathbb{E}\left[\bar{Q}_\theta(o_1, \bar{x}_{o_1}, a) \cdot \log p_{hist}\left(\bar{Q}_{n+1}(o_1, \bar{x}_{o_1}, a)\right)\right], \tag{5.11}$$

by training the histogram distribution predicted by Q_θ to match the target distribution $p_{hist}\left(\bar{Q}_{n+1}\right)$.

[5] and hence an exceedingly large number of neurons in the network output layer

5.3 Experimental study

We now compare our branching agent against prior IL and RL approaches. For our experiments, we use the open-source solver SCIP 8.0.3 [27] as backend MILP solver, along with the Ecole library [139] both for instance generation and environment simulation. As to SCIP configuration, as in previous work, we set the time limit to one hour, disable restart, and deactivate cut generation beyond root node. Unless explicitly stated otherwise, all other parameters are set to their default value.

5.3.1 Benchmarks

We consider the usual standard MILP benchmarks for learning branching strategies: set covering, combinatorial auctions, maximum independent set and multiple knapsack problems, collectively referred to as the Ecole benchmark. Thorough introduction to these problem families is provided in Appendix B. We train and test on instances of the same dimension as Scavuzzo et al. [149] and Parsonson et al. [132], as described in Table B.1. As a reminder, the size of action set $\mathcal{A}$ is equal to the number of integer variables in P. Consequently, action set sizes in the Ecole benchmark range from 30 to 480 available actions for train/test instances, and from 50 to 980 for transfer instances.

5.3.2 Baselines

We compare our DQN-BBMDP agent against prior RL baselines, namely DQN-TreeMDP (DQN-tMDP) [56], REINFORCE-TreeMDP (PG-tMDP) [149] and the current state-of-the-art DQN-Retro [132] agent. Since there is no publicly available implementation of Etheve et al. [56], we re-implemented DQN-TreeMDP and trained it on the four Ecole benchmarks, using when applicable the same network architectures and training parameters as in DQN-BBMDP and DQN-Retro. For PG-TreeMDP, we used the official implementation[6] from Scavuzzo et al. [149], using for each benchmark the tMDP+DFS network weights. Finally, because Parsonson et al. [132] only trained its agent on set covering instances, we took inspiration from their official implementation[7] to train and evaluate DQN-Retro agents on the four Ecole benchmarks. Importantly, contrary to the other RL baselines, we trained and tested DQN-Retro following SCIP's default node selection strategy.

We also compare our branching agent against the reference IL expert from Gasse et al. [62], as well

[6] `https://github.com/lascavana/rl2branch`
[7] `https://github.com/cwfparsonson/retro_branching`

Algorithm 1 DQN-BBMDP

1: **for** $t = 0...N-1$ **do**
2: Draw randomly an instance $P \sim p_0$.
3: Solve P by acting following a combined ϵ-greedy and Boltzman exploration according to $\bar{Q}_\theta$.
4: Collect transitions along the generated tree $(s_i, a_i, \sum_{j=1}^{k} r_{i+j}, s_{i+k})$ and store them into a replay buffer $\mathcal{B}_{uffer}$.
5: Update θ using the loss described in Eq (5.11) on transition batches drawn from $\mathcal{B}_{uffer}$ following prioritized experience replay.
6: **end for**

as against IL-DFS, which is the same expert, only trained and evaluated following a depth-first search node selection policy. We trained and tested both agents using the official Ecole re-implementation[8].

Finally, we report the performance of reliability pseudo cost branching (SCIP), the default branching heuristic used in SCIP, strong branching (SB), the greedy expert from which the IL agent learns from, and random branching (Random), which randomly selects a fractional variable at each step.

5.3.3 Implementation details

We use MILP bipartite graphs to encode the information contained in $(o_i, \bar{x}_{o_i})$. Thus, following previous approaches, we adopt the graph convolutional architecture of Gasse et al. [62] to parameterize our $\bar{Q}$ networks.

In Algorithm 1, we provide a high-level description of the DQN-BBMDP training pipeline. Our DQN implementation includes several Rainbow-DQN features [81]: double DQN [167], n-step learning and prioritized experience replay (PER) [151]. Moreover, as DQN-BBMDP learns distributions representing Q-values, it integrates elements of Bellemare et al. [20]. To ensure sufficient exploration of the state-action space, DQN agents are trained following Boltzmann and ϵ-greedy exploration combined. Concretely, agents select actions uniformly from $\mathcal{A}$ with probability ϵ, while following a Boltzmann exploration strategy with temperature τ for the remaining probability $1-\epsilon$. Decay rates for ϵ and τ along with all other hyperparameters used to train DQN agents on the Ecole benchmarks are listed in Table 5.2. To allow fair comparisons, when applicable, we keep SCIP parameters, training parameters and network architectures fixed for all DQN agents.

In Section 5.1.1, we defined $\mathcal{R}(s,a) = -2$ for all transition, so that the overall value of a trajectory matched the size of the B&B tree. In practice, all negative constant reward model yield equivalent

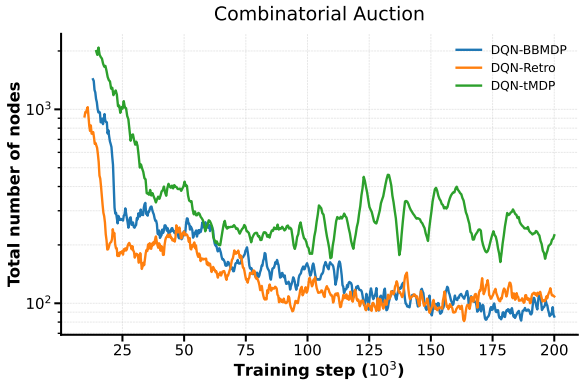
[8] `https://github.com/ds4dm/learn2branch-ecole/tree/main`

optimal policies in BBMDP, therefore, we chose to implement $\mathcal{R}(s, a) = -1$ for all RL baselines in order to allow clearer comparison between BBMDP and TreeMDP agents.

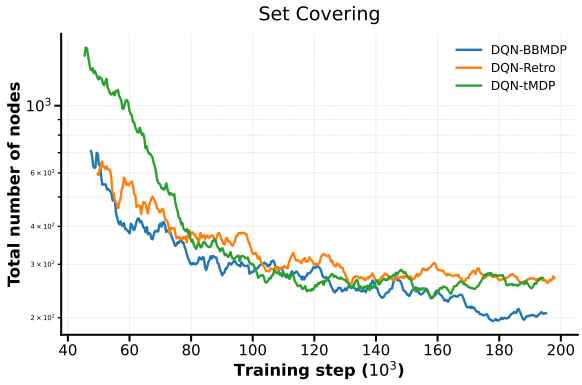
5.3.4 Training & evaluation

All experiments were conducted on an NVIDIA DGX A100 system equipped with 8× A100 40GB GPUs, 2× AMD EPYC 7742 64-core CPUs (128 threads total), and 1 TB of DDR4 RAM. Models are trained on instances of each benchmark separately, and evaluated on test instances and transfer instances. For evaluation, we report the node and time performance over 100 test instances unseen during training, as well as on 100 transfer instances of higher dimensions, as shown in Table B.1. At evaluation, performance scores are averaged over 5 seeds. Importantly, when comparing a machine learning (IL or RL) branching strategy with a standard SCIP heuristic, time performance is the only relevant criterion. In fact, when implementing one of its own branching rules, SCIP triggers a series of techniques strengthening the current MILP formulation. If these techniques effectively reduce the number of nodes to visit, they incur computational overhead which ultimately increases SCIP overall solving time. This renders node comparisons between ML and non-ML branching strategies negligible relative to solving time evaluations, as previously observed by [61, 149].

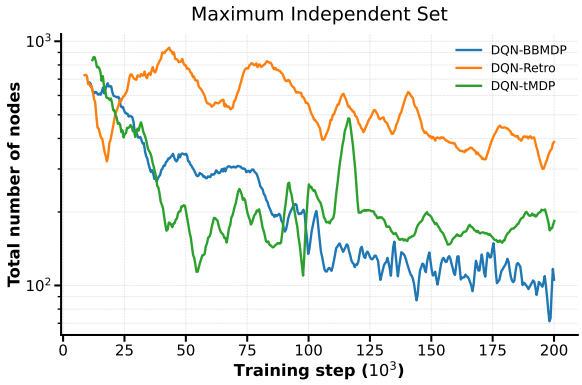
We present validation curves for DQN-BBMDP, DQN-TreeMDP and DQN-Retro in Figure 5.2. For each benchmark we trained for 200k gradient steps, which took approximately 2 days for combinatorial auction instances, 3 days for set covering instances, 5 days for multiple knapsack instances and 7 days for maximum independent set instances. As shown in Figure 5.2, DQN-BBMDP training was interrupted near full convergence on 3 out of 4 benchmarks. Crucially, based on training performance DQN-BBMDP outperforms all DQN baselines across every benchmark.



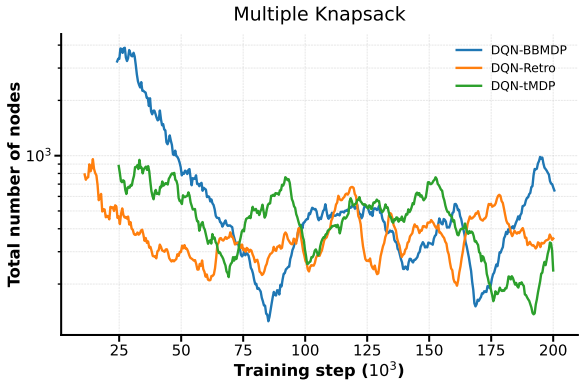
(a) Combinatorial Auction



(b) Set Covering



(c) Maximum Independent Set



(d) Multiple Knapsack

Figure 5.2: Validation curves for DQN-BBMDP, DQN-Retro and DQN-tMDP agents, in log scale. Throughout training, agents are evaluated on 20 validation instances after each batch of 100 training instances solved. Note that on the multiple knapsack benchmark, none of the agents reach convergence.

Table 5.2: Training parameters for all DQN branching agents. For DQN-Retro, we take $\gamma = 0.99$ as in Parsonson et al. [132].

Module	Training parameter	Value
Q-learning	Batch size	128
	Optimizer	Adam
	k-step return	3
	Learning rate l_r	5×10^{-5}
	Discount factor γ	1.0
	Agent steps per network update	10
	Soft target network update τ_{net}	10^{-4}
	Total number of gradient step	200×10^3
Replay buffer	Buffer minimum size $\lvert\mathcal{B}_{replay}\rvert_{init}$	20×10^3
	Buffer maximum capacity $\lvert\mathcal{B}_{replay}\rvert_{max}$	100×10^3
Prioritized experience replay	PER α	0.6
	PER β_{init}	0.4
	PER β_{final}	1.0
	$\beta_{init} \rightarrow \beta_{final}$ learner steps	100×10^3
	Minimum experience priority	10^{-3}
Exploration	Start exploration probability ϵ_{init}	1.0
	Minimum exploration probability ϵ_{min}	2.5×10^{-2}
	ϵ-decay	10^{-4}
	Start temperature τ_{init}	1.0
	Minimum temperature τ_{min}	10^{-3}
	τ-decay	10^{-5}
HL-Gauss (only for DQN-BBMDP)	z_{min}	-1
	z_{max}	16
	m_b	18
	σ	0.75

5.4 Computational results

5.4.1 Main results

Table 5.3: Performance comparison of branching agents on four standard MILP benchmarks. For each method, we report total number of B&B nodes, presolve time and total solving time outside of presolve. Lower is better, **red** indicates best agent overall, **blue** indicates best among RL agents. Presolve is common to all methods. Following prior works, we report geometrical mean over 100 test instances unseen during training and over 100 higher-dimensional transfer instances. Norm. Score denotes the aggregate average performance obtained by each agent across the four MILP benchmarks, normalized by the score of DQN-BBMDP.

Test instances

	Set Covering		**Comb. Auction**		**Max. Ind. Set**		**Mult. Knapsack**		**Norm. Score**	
Method	Node	Time	Node	Time	Node	Time	Node	Time	Node	Time
Presolve	–	4.74	–	0.90	–	1.78	–	0.20	–	–
Random	3289	5.94	1111	2.16	386.8	2.01	733.5	0.55	995	374
SB	35.8	12.93	28.2	6.21	24.9	45.87	161.7	0.69	36	2358
SCIP	62.0	2.27	20.2	1.77	19.5	2.44	289.5	**0.53**	51	253
IL	**133.8**	0.90	**83.6**	0.65	**40.1**	**0.36**	272.0	0.69	**82**	**95**
IL-DFS	136.4	**0.74**	95.5	**0.56**	69.4	0.44	472.8	1.07	114	129
PG-tMDP	649.4	2.32	168.0	0.94	153.6	0.92	436.9	1.57	233	206
DQN-tMDP	175.8	0.83	203.3	1.11	168.0	1.00	266.4	0.73	151	136
DQN-Retro	183.0	1.14	103.2	0.78	223.0	1.81	250.3	0.67	137	160
DQN-BBMDP	**152.3**	**0.77**	**97.9**	**0.62**	**103.2**	**0.69**	**236.6**	**0.66**	**100**	**100**

Transfer instances

	Set Covering		**Comb. Auction**		**Max. Ind. Set**		**Mult. Knapsack**		**Norm. Score**	
Method	Node	Time	Node	Time	Node	Time	Node	Time	Node	Time
Presolve	-	12.3	-	2.67	-	5.16	-	0.46	–	–
Random	271632	842	317235	749	215879	2102	93452	70.6	5555	2737
SB	672.1	398	389.6	255	–	–	**1709**	**12.5**	9	1425
SCIP	3309	48.4	1376	14.77	3368	90.0	30620	22.1	62	90
IL	**2610**	23.1	**1309**	**9.4**	**1882.0**	**38.6**	9747	43.5	**39**	**54**
IL-DFS	3103	**22.5**	1802	10.2	3501	51.9	43224	131	75	80
PG-tMDP	44649	221	6001	30.7	**3133**	**39.5**	35614	123	298	223
DQN-tMDP	8632	71.3	20553	116	45634	477	**22631**	**65.1**	439	445
DQN-Retro	6100	59.4	2908	18.4	119478	1863	27077	79.5	494	662
DQN-BBMDP	**5651**	**46.4**	**2273**	**11.8**	7168	81.3	37098	109	**100**	**100**

Computational results obtained across the Ecole benchmark are presented in Table 5.3. Importantly, on multiple knapsack instances, given the high computational cost associated with opting for a depth-first search node selection policy, all baselines excepted IL-DFS are evaluated following SCIP

default node selection policy.

On test instances, DQN-BBMDP consistently obtains best performance among RL agents. When compared against prior state-of-the-art DQN-Retro, DQN-BBMDP achieves an aggregate average 27% reduction of total number of node and 38% reduction of solving time outside presolve across the four Ecole benchmarks. Contrary to Parsonson et al. [132], we find DQN-Retro to yield performance comparable to DQN-tMDP. Remarkably, all RL agents outperform the SCIP solver on 3 out of 4 benchmarks in terms of solving time. Although the IL agent remains the most efficient branching agent, DQN-BBMDP enabled reducing the gap in number of explored nodes between RL agents and IL across all four benchmarks by more than half.

On transfer instances, DQN-BBMDP also dominates among RL agents, although it is outperformed by PG-tMDP on maximum independent set instances and by DQN-Retro on multiple knapsack instances. The aggregate performance gap between DQN-BBMDP and other RL baselines is notably wider on transfer instances, which aligns with the advantages of using a principled MDP formulation over TreeMDP combined with a HL-Gauss cross-entropy loss. In fact, DQN-BBMDP is the first RL agent to demonstrate robust generalization capabilities on higher dimensional instances, outperforming SCIP on 3 out of 4 benchmarks.

Interestingly, on set covering, combinatorial auction and multiple knapsack benchmarks, DQN-BBMDP outperforms IL on test instances both in average solving time and in total number of wins, that is the number of test instances solved faster than any other baseline, see Table 5.5. Surprisingly, this improvement is driven by the use of depth-first search. Because DFS is more memory-efficient when processing successive B&B nodes, see Section 2.1.4, DQN-BBMDP solves instances faster on average than IL, despite producing larger search trees. This interpretation is further supported by the fact that IL-DFS outperforms both the IL and DQN-BBMDP baselines on this set of benchmarks, while generating trees that are larger than those of IL but smaller than those of DQN-BBMDP.

Although DQN-BBMDP nearly closes the performance gap on test instances, both IL agents clearly dominate on transfer instances, suggesting that the generalization capacity of RL agents still lags behind that of IL methods. We believe this to be partly due to out of distribution effects. Since DQN agents are tested on transfer instances that are intrinsically more complex than training instances, their Q-networks are lead to evaluate B&B nodes with subtree sizes far exceeding those encountered in the later stages of training. In contrast, IL networks learn to predict the strong branching action

via behavioral cloning, making them robust to such out-of-distribution scaling effects.

5.4.2 Ablation study

Table 5.4: Ablation impact of BBMDP, HL-Gauss loss and DFS. We remove one component one at the time, and evaluate corresponding versions on 100 set covering test instances after training for 200k gradient steps as described in section 5.3.

k-step return	DQN-BBMDP	w.o. DFS	w.o. HL-Gauss	w.o. BBMDP	DQN-TreeMDP
$k = 1$	158.9	156.2 (−2%)	175.8(+10%)	158.9(+0%)	175.8(+10%)
$k = 3$	**152.3**(−4%)	150.1 (−5%)	172.3 (+8%)	162.1 (+2%)	178.9 (+13%)

We perform an ablation study on set covering test instances to separate the performance gain associated with BBMDP and the HL-Gauss classification loss. Since BBMDP and TreeMDP are strictly equivalent when minimizing one-step temporal difference, we evaluate the performance gap between one-step and k-step TD learning for both DQN-BBMDP and DQN-TreeMDP.

As shown in Table 5.4, we find that the bulk of the performance gain is brought by the use of a cross-entropy loss. Nonetheless, we observe that the use of a multi-step TD loss improves the performance of DQN-BBMDP, but degrades the performance of DQN-TreeMDP. This supports our initial claim that, in the general case, despite improving the empirical properties of RL algorithms, TreeMDP introduces approximations which hinder the asymptotic performance achievable by RL agents. Following Parsonson et al. [132], we also evaluate the cost of opting for depth-first search instead of best estimate search, SCIP's default node selection policy, when learning branching strategies. Contrary to their work, we find DFS not to be restrictive in practice in terms of node performance on set covering instances.[9]

For the second time, our empirical results contradict those reported by Parsonson et al. [132], prompting us to further investigate the source of these discrepancies. After thorough examination of both Parsonson et al. [132]'s article and official implementation, we found that the baseline labeled as DQN-tMDP (FMSTS-DFS in their article) was quite distant from the branching agent originally described by Etheve et al. [56]. In fact, in Parsonson et al. [132], the Etheve et al. [56] branching agent is not trained on TreeMDP trajectories, but on retrospective trajectories built from TreeMDP episodes, using a DFS construction heuristic. The resulting baseline is thus best interpreted as a

[9]Further analysis of the computational burden associated with DFS across the Ecole benchmark is provided in Section 7.2.6.

particular instantiation of retro-branching, as it does not correspond to the original DQN-TreeMDP agent proposed by Etheve et al. [56]. Therefore, Parsonson et al. [132] could not conclude on the superiority of retrospective trajectories over TreeMDP, nor could they assess the limitations of DFS-based RL agents. In contrast, our contribution provides evidence that, while DFS is generally expected to hinder the training performance of RL agents due to its reputation as a suboptimal node selection policy, the theoretical guarantees brought by DFS in BBMDP enable to surpass prior state-of-the-art non-DFS agents. We believe this to be due to the fact that optimizing the node selection policy has less influence on tree size performance compared to optimizing the variable selection policy, as evidenced by Etheve [55].

5.4.3 Additional performance metrics

We include further computational results on instances of the Ecole benchmark.

Table 5.5 provides additional performance metrics to compare the different baselines across test and transfer benchmarks. For each benchmark, we report the number of wins and the average rank of each baseline across 100 evaluation instances. The number of wins is defined as the number of instances where a baseline solves a MILP problem faster than any other baseline. When multiple baselines fail to solve an instance to optimality within the time limit, their performance is ranked based on final dual gap. The results in this table presented are broadly consistent with those provided in 5.3.

Finally, Table 5.6 recapitulates the computational results presented in Table 5.3, and provides for each baseline the per-benchmark standard deviation over five seeds, as well as the fraction of test instances solved to optimality within the time limit.

Table 5.5: Additional performance metrics for each baseline on train / test and transfer instance benchmarks. For each benchmark, we report the number of wins, and the average rank of each baseline across the 100 evaluation instances. We also report for each baseline the fraction of test instances solved to optimality within time limit. The number of wins is defined as the number of instances where a baseline solves a MILP problem faster than all other baselines. When multiple baselines fail to solve an instance to optimality within time limit, their performance is ranked based on final dual gap.

	Train / Test			Transfer		
Method	Solved	Wins	Rank	Solved	Wins	Rank
SCIP	100/100	8/100	5.7	100/100	10/100	3.3
IL	100/100	1/00	3.8	100/100	29/100	1.8
IL-DFS	100/100	35/100	2.1	**100/100**	**58/100**	**1.7**
PG-tMDP	100/100	0/100	6.6	78/100	0/100	6.8
DQN-tMDP	100/100	11/100	2.9	96/100	0/100	5.0
DQN-Retro	100/100	1/100	4.9	98/100	0/100	5.1
DQN-BBMDP	100/100	**44/100**	**1.9**	**100/100**	**3/100**	**4.2**

Set covering

	Train / Test			Transfer		
Method	Solved	Wins	Rank	Solved	Wins	Rank
SCIP	100/100	14/100	4.1	100/100	14/100	3.51
IL	100/100	16/100	3.5	**100/100**	**47/100**	**1.7**
IL-DFS	100/100	31/100	2.6	100/100	20/100	2.5
PG-tMDP	100/100	0/100	5.7	100/100	0/100	5.8
DQN-tMDP	100/100	0/100	6.2	100/100	0/100	6.5
DQN-Retro	100/100	10/100	3.6	100/100	1/100	4.6
DQN-BBMDP	100/100	**34/100**	**2.3**	**100/100**	**18/100**	**2.9**

Combinatorial Auction

	Train / Test			Transfer		
Method	Solved	Wins	Rank	Solved	Wins	Rank
SCIP	100/100	9/100	5.7	100/100	7/100	4.5
IL	100/100	**72/100**	**1.6**	100/100	**57/100**	**1.7**
IL-DFS	100/100	10/100	2.4	100/100	0/100	3.2
PG-tMDP	100/100	0/100	4.9	**100/100**	**36/100**	**1.8**
DQN-tMDP	100/100	1/100	4.8	85/100	0/100	6.1
DQN-Retro	100/100	**6/100**	5.4	22/100	0/100	6.7
DQN-BBMDP	100/100	2/100	**3.3**	95/100	0/100	4.2

Maximum Independent Set

	Train / Test			Transfer		
Method	Solved	Wins	Rank	Solved	Wins	Rank
SCIP	100/100	**88/100**	**1.4**	**100/100**	**60/100**	**1.9**
IL	100/100	1/100	4.5	100/100	6/100	3.4
IL-DFS	100/100	1/100	5.8	98/100	0/100	6.0
PG-tMDP	100/100	0/100	6.0	98/100	5/100	5.0
DQN-tMDP	100/100	1/100	3.5	99/100	**14/100**	**3.5**
DQN-Retro	100/100	3/100	3.5	98/100	9/100	3.8
DQN-BBMDP	100/100	**6/100**	**3.3**	**100/100**	6/100	4.3

Multiple Knapsack

Table 5.6: Computational performance comparison on four MILP benchmarks. Following prior works, we report geometrical mean over 100 instances, averaged over 5 seeds, as well as per-benchmark standard deviations.

	Train / Test			Transfer		
Method	Nodes	Time	Solved	Nodes	Time	Solved
Random	3289 ± 4.2%	5.9 ± 4.3%	100/100	270365 ± 9.5%	811 ± 7.9%	60/100
SB	35.8 ± 0.0%	12.93 ± 0.0%	100/100	672.1 ± 0.0%	398 ± 0.2%	82/100
SCIP	62.0 ± 0.0%	2.27 ± 0.0%	100/100	3309 ± 0.0%	48.4 ± 0.1%	100/100
IL	**133.8 ± 1.0%**	0.90 ± 4.8%	100/100	**2610 ± 0.7%**	23.1 ± 1.5%	**100/100**
IL-DFS	136.4 ± 1.8%	**0.74 ± 5.3%**	100/100	3103 ± 2.0%	**22.5 ± 3.1%**	**100/100**
PG-tMDP	649.4 ± 0.7%	2.32 ± 2.4%	100/100	44649 ± 3.7%	221 ± 4.1%	78/100
DQN-tMDP	175.8 ± 1.1%	0.83 ± 4.5%	100/100	8632 ± 4.9%	71.3 ± 5.8%	96/100
DQN-Retro	183.0 ± 1.2%	1.14 ± 4.1%	100/100	6100 ± 4.2%	59.4 ± 4.2%	98/100
DQN-BBMDP	**152.3 ± 0.6%**	**0.77 ± 5.6%**	100/100	**5651 ± 2.2%**	**46.4 ± 3.3%**	**100/100**

Set covering

	Train / Test			Transfer		
Method	Nodes	Time	Solved	Nodes	Time	Solved
Random	1111 ± 4.3%	2.16 ± 6.6%	100/100	354650 ± 6.7%	814 ± 7.1%	64/100
SB	28.2 ± 0.0%	6.21 ± 0.1%	100/100	389.6 ± 0.0%	255 ± 0.2%	88/100
SCIP	20.2 ± 0.0%	1.77 ± 0.1%	100/100	1376 ± 0.0%	14.77 ± 0.1%	100/100
IL	**83.6 ± 0.8%**	0.65 ± 7.3%	100/100	**1309 ± 1.6%**	**9.4 ± 2.2%**	**100/100**
IL-DFS	95.5 ± 0.9%	**0.56 ± 7.1%**	100/100	1802 ± 2.0%	10.2 ± 1.8%	100/100
PG-tMDP	168.0 ± 2.8%	0.94 ± 6.0%	100/100	6001 ± 2.7%	30.7 ± 2.4%	100/100
DQN-tMDP	203.3 ± 4.2%	1.11 ± 4.0%	100/100	20553 ± 3.8%	116 ± 3.9%	100/100
DQN-Retro	103.2 ± 1.2%	0.78 ± 7.5%	100/100	2908 ± 1.7%	18.4 ± 2.7%	100/100
DQN-BBMDP	**97.9 ± 1.2%**	**0.62 ± 8.5%**	100/100	**2273 ± 1.9%**	**11.8 ± 2.0%**	**100/100**

Combinatorial auction

	Train / Test			Transfer		
Method	Nodes	Time	Solved	Nodes	Time	Solved
Random	386.8 ± 5.4%	2.01 ± 4.8%	100/100	215879 ± 6.7%	2102 ± 6.2%	25/100
SB	24.9 ± 0.0%	45.87 ± 0.4%	100/100	169.9 ± 0.2%	2172 ± 0.9%	15/100
SCIP	19.5 ± 0.0%	2.44 ± 0.4%	100/100	3368 ± 0.0%	90.0 ± 0.2%	100/100
IL	**40.1 ± 3.45%**	**0.36 ± 3.1%**	100/100	**1882 ± 4.0%**	**38.6 ± 3.2%**	**100/100**
IL-DFS	69.4 ± 6.5%	0.44 ± 4.8%	100/100	3501 ± 2.7%	51.9 ± 2.6%	100/100
PG-tMDP	153.6 ± 5.0%	0.92 ± 2.6%	100/100	**3133 ± 4.6%**	**39.5 ± 3.8%**	**100/100**
DQN-tMDP	168.0 ± 5.6%	1.00 ± 3.4%	100/100	45634 ± 7.4%	477 ± 5.1%	85/100
DQN-Retro	223.0 ± 4.1%	1.81 ± 3.6%	100/100	119478 ± 6.1%	1863 ± 4.8%	22/100
DQN-BBMDP	**103.2 ± 9.3%**	**0.62 ± 6.8%**	100/100	7168 ± 5.3%	81.3 ± 4.2%	95/100

Maximum independent set

	Train / Test			Transfer		
Method	Nodes	Time	Solved	Nodes	Time	Solved
Random	733.5 ± 13.0%	0.55 ± 6.9%	100/100	93452 ± 14.3%	70.6 ± 9.2%	99/100
SB	161.7 ± 0.0%	0.69 ± 0.1%	100/100	**1709 ± 0.5%**	**12.5 ± 0.9%**	**100/100**
SCIP	289.5 ± 0.0%	0.53 ± 0.2%	100/100	30260 ± 0.0%	22.14 ± 0.2%	100/100
IL	272.0 ± 12.9%	0.69 ± 8.5%	100/100	9747 ± 7.5%	46.5 ± 6.6%	100/100
IL-DFS	472.8 ± 13.0%	1.07 ± 9.0%	100/100	43224 ± 9.0%	131 ± 8.6%	98/100
PG-tMDP	436.9 ± 21.2%	1.57 ± 16.9%	100/100	35614 ± 14.3%	123 ± 15.4%	98/100
DQN-tMDP	266.4 ± 7.2%	0.73 ± 4.6%	100/100	**22631 ± 8.6%**	**65.1 ± 5.5%**	99/100
DQN-Retro	250.3 ± 9.5%	0.67 ± 5.0%	100/100	27077 ± 8.8%	79.5 ± 6.2%	98/100
DQN-BBMDP	**236.6 ± 6.4%**	**0.66 ± 2.7%**	100/100	37098 ± 7.0%	109 ± 4.9%	**100/100**

Multiple knapsack

5.5 Assessing the impact of reward sparsity in BBMDP

Although our DQN-BBMDP agent substantially improves over prior RL-based approaches on the Ecole benchmark suite, it still falls short of the performance achieved by IL methods, notably IL-DFS. This observation appears to contradict theoretical intuition, since, in principle, when nearing convergence, reinforcement learning agents should be able to at least recover the performance of imitation learning agents trained to replicate strong branching. A natural explanation is that IL benefits from a substantial inductive advantage: by construction, it inherits the structural prior of strong branching through supervision, whereas RL must reconstruct this prior solely from weak reward feedback. Building on this insight, we investigate training DQN-BBMDP following an alternative reward model, designed to closely align with the dual gap reduction objective explicitly optimized in strong branching. Contrary to our initial expectation, we find that RL branching agents trained following SB-inspired reward models underperfom the original ones. This suggests that the remaining performance gap between DQN-BBMDP and IL-DFS is better explained by intrinsic challenges of reinforcement learning such as the one presented in Section 4.2, beyond reward sparsity.

5.5.1 Strong branching reward models

We first introduce a naive strong branching inspired reward model, denoted $\mathcal{R}_0^{SB}$, intended to provide a denser, well-motivated reward signal along BBMDP trajectories:

$$\mathcal{R}_0^{SB}(s, a, s') = \text{score}(\Delta_l^a, \Delta_r^a) \tag{5.12}$$

where Δ_l^a and Δ_r^a denote the dual bound improvements achieved by the LP relaxations of the left and right child nodes obtained after branching on variable a at state s, as defined in Section 2.1.3. Importantly, when branching on a leads to the direct pruning of one or both child nodes, the associate dual bound improvement Δ_i^a is set to the remaining gap between the original *GUB* and the current LP relaxation objective:

$$\Delta_i^a = c^\top(\bar{x}_0 - x_{LP}^*). \tag{5.13}$$

To preserve the Markov property of the reward model, the state definition must be augmented with $\bar{x}_0$, the incumbent solution at the root node. Accordingly, in the following we define $s_t = (\mathcal{V}_t, \mathcal{E}_t, \bar{x}_t, \bar{x}_0)$. Similarly, subtree value functions now operate on triplet inputs $(o_i, \bar{x}_{o_i}, \bar{x}_0)$, still encoded using MILP bipartite graphs.

Although applying $\mathcal{R}_0^{SB}$ to BBMDP is appealing, it requires careful consideration. In fact, directly adapting the strong branching scoring function yields a reward model that, under undiscounted returns, incentivizes arbitrarily long trajectories.

Proposition 5.5.1. Under undiscounted returns, $\mathcal{R}_0^{SB}$ fails to recover strong branching behavior in BBMDP. Instead, it systematically favors decisions that run counter to it.

Proof. Formal proof is deferred to Proposition 5.5.2. □

Strong branching yields small B&B trees in practice by optimizing over a short-term objective. Counterintuitively, maximizing the cumulative sum of strong branching scores over entire B&B episodes produces terribly poor variable selection strategies, because agents are incentivized to prolong trajectories as much as possible in order to keep accumulating rewards. This partly stems from the fact that, for all transitions, $\mathcal{R}_0^{SB}(s, a, s') > 0$. A seemingly straightforward fix is to redefine the reward as the difference between the LP relaxation improvement achieved by the agent's action and that achieved by the strong branching action:

$$\mathcal{R}_1^{SB}(s, a, s') = \text{score}(\Delta_l^a, \Delta_r^a) - \max_{a' \in \mathcal{A}} \text{score}(\Delta_l^{a'}, \Delta_r^{a'}) \tag{5.14}$$

Since $\mathcal{R}_1$ is negative by construction, the agent is incentivized not only to match strong branching decisions, but also to terminate episodes as quickly as possible, thereby implicitly minimizing the size of the B&B tree. Unfortunately, such reward model induce prohibitive evaluation costs: it requires solving all branching candidate LP relaxations at every step of the B&B process, which is intractable in an RL training loop.

Hence, training RL agents following SB-inspired reward models necessarily involves optimizing γ-discounted returns, with $\gamma < 1$. A natural extreme is to take $\gamma = 0$, in which case the agent focuses on maximizing immediate dual bound improvement, effectively mirroring strong branching. However, discounting with $\gamma > 0$ opens the door to learning policies that trade-off short-term gains for larger long-term dual bound improvements, effectively implementing a deeper version of strong branching which, while theoretically attractive, is typically far too computationally expensive in practice to be deployed within modern solvers. Therefore, in this section we propose to investigate the conditions under which γ-discounted returns yield practically relevant optimal policies in BBMDP.

Proposition 5.5.2. Under $\mathcal{R}_0^{SB}$, setting $\gamma = \frac{1}{2}$ renders all BBMDP trajectories equivalent: every policy yields the same discounted return, equal to the initial primal–dual gap at the root. Moreover, for $\gamma < \frac{1}{2}$ (resp. $\gamma > \frac{1}{2}$), BBMDP trajectories yield discounted returns strictly inferior (resp. strictly superior) to the initial gap $c^\top(\bar{x}_0 - x^*_{LP})$.

Proof. We proceed by induction on the depth of the B&B search tree. Let P be a MILP and let k denote the depth of its associated B&B tree. Let $\bar{x}_0$ be the incumbent solution at the root node, and x^*_{LP} the optimal solution of the root LP relaxation. The primal-dual gap at the root node thus writes $c^\top(\bar{x}_0 - x^*_{LP})$.

Initialization: In the case where the B&B tree associated to P has depth $k = 1$, exactly one branching decision is made at the root and the episode terminates immediately. The discounted return of the trajectory thus reduces to the one-step reward:

$$\begin{aligned} V^\pi(s_0) &= \mathcal{R}_0^{SB}(s_0, a_0, s_1) \\ &= \frac{1}{2}(c^\top(\bar{x}_0 - x^*_{LP}) + c^\top(\bar{x}_0 - x^*_{LP})) \quad \text{(by Equation (5.13))} \\ &= c^\top(\bar{x}_0 - x^*_{LP}). \end{aligned}$$

Note that initialization is verified for any $\gamma \in [0, 1]$. In particular, if $\gamma < \frac{1}{2}$ (resp. $\gamma > \frac{1}{2}$), then $V^\pi(s_0) \leq c^\top(\bar{x}_0 - x^*_{LP})$, (resp. $V^\pi(s_0) \geq c^\top(\bar{x}_0 - x^*_{LP})$).

Heredity: We assume that, for any MILP instance whose B&B tree has depth at most $k \geq 1$, the corresponding discounted sum of rewards under both $\mathcal{R}_0^{SB}$ and $\gamma = \frac{1}{2}$ in BBMDP equals $c^\top(\bar{x}_0 - x^*_{LP})$. Consider now an instance P' whose B&B tree has depth $k + 1$. Let o_l and o_r be the two children of the root node after branching on action a. Crucially, the subtrees rooted in o_l and o_r have depth at most k, hence the induction hypothesis applies to both. Thus, the Bellman equation yields:

$$\begin{aligned} V^\pi(s_0) &= \mathcal{R}_0^{SB}(s_0, a_0, s_1) + \gamma V^\pi(s_1) \\ &= \frac{1}{2}(\Delta_l^a + \Delta_r^a) + \frac{1}{2}(\bar{V}^\pi(o_l, \bar{x}_{o_l}, \bar{x}_0) + \bar{V}^\pi(o_r, \bar{x}_{o_r}, \bar{x}_0)) \\ &= \frac{1}{2}(\Delta_l^a + \Delta_r^a) + \frac{1}{2}\left((c^\top\bar{x}_0 - (c^\top x^*_{LP} + \Delta_l^a)\right) + \frac{1}{2}\left((c^\top\bar{x}_0 - (c^\top x^*_{LP} + \Delta_r^a)\right) \quad \text{(by induction)} \\ &= c^\top(\bar{x}_0 - x^*_{LP}) \end{aligned}$$

Note that for $\gamma < \frac{1}{2}$ (respectively $\gamma > \frac{1}{2}$), then by the induction hypothesis,

$$\bar{V}^\pi(o_i, \bar{x}_{o_i}, \bar{x}_0)) \leq c^\top\bar{x}_0 - (c^\top x^*_{LP} + \Delta_i^a), \quad \text{(respectively } \bar{V}^\pi(o_i, \bar{x}_{o_i}, \bar{x}_0)) \geq c^\top\bar{x}_0 - \left(c^\top x^*_{LP} + \Delta_i^a\right)$$

for $i \in \{l, r\}$, yielding

$$V^{\pi}(s_0) < c^{\top}(\bar{x}_0 - x^*_{LP}) \quad (\text{respectively } V^{\pi}(s_0) > c^{\top}(\bar{x}_0 - x^*_{LP})).$$

Initialization and heredity are satisfied, therefore, by induction, under $\mathcal{R}_0^{SB}$ and $\gamma = \frac{1}{2}$, all BBMDP trajectories yield the same discounted cumulative return, equal to $c^{\top}(\bar{x}_0 - x^*_{LP})$. Moreover, for $\gamma < \frac{1}{2}$ (resp. $\gamma > \frac{1}{2}$), BBMDP trajectories yield discounted returns strictly inferior (resp. strictly superior) to $c^{\top}(\bar{x}_0 - x^*_{LP})$. □

Proof. We now provide a formal proof for Proposition 5.5.1. From the Bellman equation, we have:

$$V^{\pi}(s_0) = \frac{1}{2}(\Delta_l^a + \Delta_r^a) + \gamma \cdot (\bar{V}^{\pi}(o_l, \bar{x}_{o_l}, \bar{x}_0) + \bar{V}^{\pi}(o_r, \bar{x}_{o_r}, \bar{x}_0)). \tag{5.15}$$

Let's suppose that there exists a variable $x_j \in \mathcal{A}$ such that branching on $x_j \in \mathcal{A}$ at s_0 terminates the episode. Now, let's consider the class of policies that (i) select an action in $\mathcal{A}$ at s_0, and then (ii) branch deterministically on x_j in the two successor states s_1 and s_2, hence effectively solving the original MILP. For any such policy, the maximum value of a trajectory verifies:

$$V^{\pi}(s_0) = \max_{a \in \mathcal{A}}(\tfrac{1}{2} - \gamma)(\Delta_l^a + \Delta_r^a) + \gamma \left(c^{\top}\bar{x}_0 - c^{\top}x^*_{LP}\right). \tag{5.16}$$

If $\gamma < \frac{1}{2}$, the coefficient $\frac{1}{2} - \gamma$ is positive, and the maximizing action coincides with the strong branching choice (i.e., the action that maximizes $\Delta_l^a + \Delta_r^a$), yielding a value of $c^{\top}(\bar{x}_0 - x^*_{LP})$. In contrast, if $\gamma > \frac{1}{2}$, the coefficient becomes negative and the maximizing action is the *opposite* of strong branching, namely the action that *minimizes* the immediate dual bound improvement. Applying the same argument recursively implies that, for $\gamma > \frac{1}{2}$, the reward model $\mathcal{R}_0^{SB}(s, a, s')$ induces optimal policies that not only deviate from strong branching, but in fact systematically extend the length of B&B trajectories in order to increase total cumulative return. Conversely, when $\gamma < \frac{1}{2}$, strong branching behavior is favored, as it generally aligns with the objective of minimizing the size of the B&B tree.

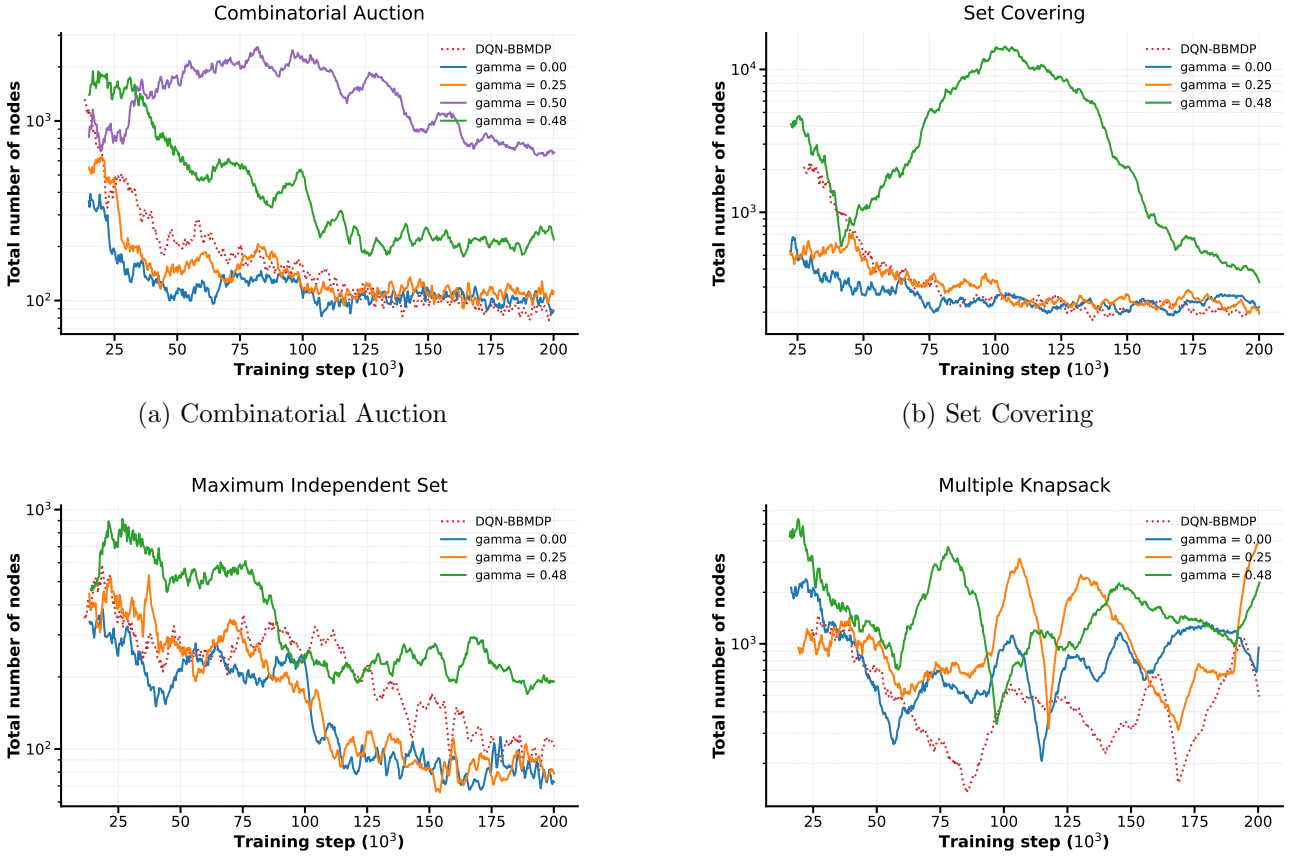
□

To ensure trajectories whose values are bounded by 1, in the following, we propose training RL agents following a refined, normalized version of $\mathcal{R}_0^{SB}$, defined as :

$$\mathcal{R}^{SB}(s, a, s') = \frac{\text{score}(\Delta_l^a, \Delta_r^a)}{c^{\top}(\bar{x}_0 - x^*_{LP}).} \tag{5.17}$$

5.5.2 Computational results

We train DQN-BBMDP agents under $\mathcal{R}^{SB}$ using three distinct discount rates $\gamma \in \{0.0, 0.25, 0.48\}$. Aside from the reward model and discounting factor, all other experimental settings are kept identical to those used in the previous sections. Figure 5.3 shows validation curves obtained across the Ecole benchmark.



(a) Combinatorial Auction

(b) Set Covering

(c) Maximum Independent Set

(d) Multiple Knapsack

Figure 5.3: Comparison of validation curves across four different scenarios. Top row shows baselines, bottom row shows our method and ablations. On combinatorial auction, we add the baseline corresponding to $\gamma = 0.5$.

Two key observations arise from these curves. First, on combinatorial auction, set covering and maximum independent set instances, we observe that $\mathcal{R}^{SB}$ branching agents reach convergence faster than the original DQN-BBMDP agent. This aligns with the perspective of the bitter lesson: under restricted data and compute, injecting domain expertise (here, strong branching rewards) enables to improve the performance achievable by learning agents. Second, however, as training scales, we

observe that agents agnostic to such priors tend to catch up over longer horizons, and eventually surpass expert-guided agents.

We now turn to analyzing how the choice of γ affects the performance of the learned branching policy under $\mathcal{R}^{SB}$. Judging from validation training curves, no single $\mathcal{R}^{SB}$ baseline appears to consistently dominate the others. Table 5.7 therefore provides a clearer comparison by reporting the agents' computational performance on both test and transfer instances from the Ecole benchmark.

Table 5.7: Performance comparison of $\mathcal{R}^{SB}$ branching agents across the Ecole benchmark. For each method, we report total number of B&B nodes, presolve time and total solving time outside of presolve.

Test instances

	Set Covering		**Comb. Auction**		**Max. Ind. Set**		**Mult. Knapsack**		**Norm. Score**	
Method	Node	Time	Node	Time	Node	Time	Node	Time	Node	Time
Presolve	–	4.74	–	0.90	–	1.78	–	0.20	–	–
IL	**133.8**	0.90	**83.6**	0.65	**40.1**	**0.36**	272.0	0.69	**82**	**95**
IL-DFS	136.4	**0.74**	95.5	**0.56**	69.4	0.44	472.8	1.07	114	129
$\mathcal{R}$ (original)	**152.3**	0.77	**97.9**	0.62	**103.2**	**0.69**	**236.6**	0.66	**100**	**100**
$\mathcal{R}^{SB}(\gamma = 0.00)$	154.7	**0.79**	101.1	**0.64**	82.3	0.59	281.8	0.76	101	102
$\mathcal{R}^{SB}$ $(\gamma = 0.25)$	160.5	0.81	101.9	0.64	74.5	0.54	415.9	0.99	114	110
$\mathcal{R}^{SB}$ $(\gamma = 0.48)$	261.7	1.12	168.5	0.82	135.1	0.85	766.7	2.02	202	229

Transfer instances

	Set Covering		**Comb. Auction**		**Max. Ind. Set**		**Mult. Knapsack**		**Norm. Score**	
Method	Node	Time	Node	Time	Node	Time	Node	Time	Node	Time
Presolve	-	12.3	-	2.67	-	5.16	-	0.46	–	–
IL	**2610**	23.1	**1309**	**9.4**	**1882.0**	**38.6**	9747	43.5	**39**	**54**
IL-DFS	3103	**22.5**	1802	10.2	3501	51.9	43224	131	75	80
$\mathcal{R}$ (original)	5651	46.4	**2273**	**11.8**	**7168**	**81.3**	37098	109	**100**	**100**
$\mathcal{R}^{SB}$ $(\gamma = 0.00)$	**4212**	**29.4**	2991	15.8	12925	198	**16430**	**53.4**	108	109
$\mathcal{R}^{SB}$ $(\gamma = 0.25)$	4290	29.7	2530	12.7	38362	427	58180	175	220	215
$\mathcal{R}^{SB}$ $(\gamma = 0.48)$	14415	98.1	6538	30.0	51324	582	69739	219	362	346

Computational results analysis suggest three main takeaways. First, across the Ecole benchmark, under $\mathcal{R}^{SB}$, discounting with $\gamma = 0$ yields the strongest variable selection strategies overall, in particular those generalizing best to higher-dimensional instances. Importantly, this does not imply that one-step strong branching necessarily induces better branching strategies than deeper, less myopic variants; rather, it indicates that taking $\gamma = 0$ provides the most effective learning signal when the goal is to learn to replicate strong branching behaviour via reinforcement learning.

Second, contrary to our initial expectation, we find that, in BBMDP, adopting a denser reward model closely mirroring strong branching does not allow to recover the performance of IL baselines on either test or transfer instances, falling well short in both settings. In fact, we even find that the best $\mathcal{R}^{SB}$ configuration fails to consistently outperform the original DQN-BBMDP agent on the Ecole benchmark test instances. This suggests that DQN-BBMDP effectively suffers from issues inherent to deep reinforcement learning such as the ones described in Section 4.2, beyond reward sparsity.

Third and last, our computational results are also consistent with a "bitter lesson" interpretation: ultimately, inductive biases remain biases, hence injecting priors that are misaligned with the learning objective can harm generalization. This is particularly exemplified here: although strong branching behaviour is generally correlated with smaller B&B trees, learning to imitate strong branching is not equivalent to directly minimizing B&B tree size. Accordingly, while $\mathcal{R}^{SB}$ achieves computational performance comparable to that of the original DQN-BBMDP agent on test instances, its overall performance drops by about $\approx 10\%$ on transfer instances relative to the original DQN-BBMDP agent.

Overall, these results argue for retaining the constant negative reward model when the goal is to learn branching policies that minimize B&B tree size. That said, this conclusion does not apply uniformly across problem classes: on set covering, for instance, $\mathcal{R}^{SB}$ baselines tend to demonstrate better generalization performance than the original DQN-BBMDP agent, suggesting that strong branching reward shaping can remain beneficial in specific regimes.

5.5.3 Alignment with strong branching behaviour

As shown in Table 5.7, the original DQN-BBMDP agent performs competitively with the $\mathcal{R}^{SB}$ baselines, despite being trained without explicit strong branching signal. This naturally raises the question of whether DQN-BBMDP surpasses $\mathcal{R}^{SB}$ agents by aligning more closely with strong branching, or instead by learning genuinely distinct branching strategies. Conversely, because $\mathcal{R}^{SB}$ baselines still fail to recover the performance level of IL despite optimizing a strong branching inspired objective, one may ask whether this gap can be explained by a weaker alignment with strong branching behaviour.

To answer these questions, Table 5.8 provides several alignment metrics designed to quantify the extent to which IL, DQN-BBMDP and $\mathcal{R}^{SB}$ policies resemble strong branching. First, we report the average cross-entropy between the policy predicted by each learning baseline and the strong branching policy, normalized by the cross-entropy between the strong branching policy and a uniform random

Table 5.8: Performance comparison of $\mathcal{R}^{SB}$ branching agents across the Ecole benchmark. Alignment metrics between ML baselines and SB on MIS test and transfer instances. Obviously, SB is perfectly aligned with SB.

Test instances

	Set Covering			**Comb. Auction**			**Max. Ind. Set**			**Mult. Knapsack**		
Method	CE	Score	Freq	CE	Score	Freq	CE	Score	Freq	CE	Score	Freq
SB	0.00	1.00	1.00	0.00	1.00	1.00	0.00	1.00	1.00	0.00	1.00	1.00
IL	**0.59**	**0.84**	**0.57**	**0.70**	**0.77**	**0.49**	**0.84**	**0.69**	**0.45**	**0.86**	**0.99**	**0.18**
$\mathcal{R}$	0.89	0.80	0.50	0.88	0.73	0.44	0.97	0.55	0.37	0.88	0.99	0.16
$\mathcal{R}^{SB}_{(0.00)}$	**0.87**	**0.82**	**0.53**	**0.87**	**0.75**	**0.46**	**0.95**	**0.66**	**0.44**	**0.87**	**0.99**	**0.17**
$\mathcal{R}^{SB}_{(0.25)}$	0.88	0.81	0.52	0.88	0.74	0.44	0.96	0.59	0.38	0.87	0.99	0.16
$\mathcal{R}^{SB}_{(0.48)}$	0.92	0.72	0.42	0.91	0.65	0.36	0.99	0.51	0.30	0.92	0.99	0.15

Transfer instances

	Set Covering			**Comb. Auction**			**Max. Ind. Set**			**Mult. Knapsack**		
Method	CE	Score	Freq	CE	Score	Freq	CE	Score	Freq	CE	Score	Freq
SB	0.00	1.00	1.00	0.00	1.00	1.00	0.00	1.00	1.00	0.00	1.00	1.00
IL	**0.60**	**0.85**	**0.57**	**0.70**	**0.77**	**0.49**	**0.86**	**0.76**	**0.40**	**0.88**	**0.99**	**0.17**
$\mathcal{R}$	0.92	0.78	0.51	0.89	0.71	0.42	0.97	0.58	0.37	0.90	0.99	**0.16**
$\mathcal{R}^{SB}_{(0.00)}$	**0.91**	**0.81**	**0.52**	**0.88**	**0.74**	**0.45**	**0.96**	**0.66**	**0.39**	**0.92**	**0.99**	0.15
$\mathcal{R}^{SB}_{(0.25)}$	0.91	0.81	0.52	0.90	0.72	0.43	0.97	0.65	0.36	0.91	0.99	0.16
$\mathcal{R}^{SB}_{(0.48)}$	0.94	0.71	0.41	0.95	0.62	0.34	0.99	0.50	0.28	0.94	0.98	0.14

policy (CE). Second, we report the average strong branching score of the action selected by the learning baseline, normalized by the score of the action selected by the strong branching expert (Score). Third, we report the frequency with which the learning baseline selects the same action as the strong branching expert (Freq). All metrics are averaged over 100 test instances and 20 higher-dimensional transfer instances.

Three main takeaways emerge from the computational results reported in Table 5.8. First, the original $\mathcal{R}$ baseline and modified $\mathcal{R}^{SB}$ baselines exhibit remarkably similar alignment scores with respect to strong branching across all benchmarks. This observation suggests that, even when dual bound improvement is not explicitly optimized through the reward signal, DQN-BBMDP agent still tends to learn policies that maximize dual bound improvements at each branching step. A plausible explanation is that, given the intrinsic difficulty of exploring combinatorial state–action spaces, policies

that approximate strong branching decisions provide a natural and effective direction in the parameter space for reducing the size of the B&B tree.

Second, all RL baselines consistently display weaker alignment with strong branching than the IL baseline. Taken together, these results suggest that the performance gap observed between RL and IL methods can broadly be attributed to differences in their ability to reproduce strong branching behavior. Nevertheless, it should be noted that in settings where DQN-BBMDP achieves higher performance than $\mathcal{R}^{SB}$ baselines – most notably on combinatorial auction and maximum independent set instances – it does so while exhibiting slightly lower strong branching alignment scores. This suggests that although DQN-BBMDP policies broadly resemble those induced by $\mathcal{R}^{SB}$, optimizing directly for B&B tree size allows the agent to locally depart from strong branching when doing so leads to better decisions. In other words, when the optimal strong branching action cannot be predicted reliably, the original $\mathcal{R}$ baseline appears able to identify alternative branching decisions that more effectively reduce the final B&B tree size.

Finally, the multiple knapsack benchmark provides a particularly instructive case study. Unlike the other benchmarks, it is characterized by a very flat strong-branching landscape: most candidate branching variables receive highly similar strong branching scores. This has an important consequence: as many actions are nearly equivalent in terms of immediate dual bound improvement, identifying the exact strong branching action becomes substantially harder. Indeed, while all learning baselines select actions whose associated dual bound improvements exceed 99% of that achieved by the strong branching action, they reproduce the exact strong branching decision much less frequently than on the other benchmarks, doing so only about $\approx 16\%$ of the time – far less frequently than on the other benchmarks. This observation also helps explain why DQN-BBMDP struggles to converge on the multiple knapsack benchmark. When the supervision or reward signal induced by strong branching is nearly flat across actions, the learning problem becomes poorly conditioned, making it difficult for the agent to infer a stable preference ordering over candidate variables and, consequently, to achieve robust policy improvement in DQN. In principle, DQN-BBMDP agents trained with a constant reward model should be less exposed to this issue, since their objective does not explicitly depend on dual bound improvement. However, the fact that they nonetheless exhibit similar unstable node validation curves, as shown in Figure 5.3d, supports the hypothesis that, under limited exploration, even RL agents trained without explicit dual bound improvement supervision tend in practice to gravitate

toward strong branching-like decisions.

5.6 Conclusion and perspectives

Guiding combinatorial optimization (CO) algorithms with reinforcement learning (RL) has proven challenging, including beyond the field of mixed-integer linear programming (MILP) [23]. Not only are RL agents consistently outperformed by human-expert CO heuristics or IL agents trained to mimic these experts, but their application has also been limited so far to fairly easy problem instances. This begs for a thorough study of these problems' properties, for well-posed formulations and principled methods. In this chapter, we showed the theoretical and practical limits of the concept of TreeMDP for learning optimal branching strategies in MILP, highlighting in which context TreeMDPs were a valid formulation. Introducing BBMDP, we proposed a rigorous description of variable selection in B&B, unlocking the use of vanilla dynamic programming methods and their refinements. In turn, the resulting DQN-BBMDP method outperformed prior RL-inspired agents on the Ecole benchmark.

Nonetheless, there remains significant room for improvement for RL approaches, as the generalization capacity of RL agents still lags behind that of IL on both test and transfer instances. Interestingly, such performance gap cannot be attributed to the constant negative reward formulation adopted in BBMDP, as our computational study shows that DQN agents trained following a strong branching inspired objective perform slightly worse than the original DQN-BBMDP agents on the Ecole benchmark. Instead, we postulate that RL approaches face two major challenges relative to IL methods. First, out-of-distribution effects: transfer instances are intrinsically more complex than training instances, leading Q-networks to evaluate B&B nodes whose associated subtree sizes lie far outside the range observed during training. This extrapolation regime can significantly degrade value estimation on transfer instances. In contrast, IL agents trained via behavioral cloning learn to imitate strong branching decisions directly, which makes them less sensitive to scaling effects in subtree size and therefore more robust under distribution shift. Second, exploration in high-dimensional combinatorial spaces poses a fundamental difficulty. As highlighted in Section 4.2.3, unlike supervised learning, RL must discover high-quality decisions through interaction, balancing exploration and exploitation. However, standard exploration mechanisms such as ε-greedy or Boltzmann sampling are poorly suited to large combinatorial state–action spaces: random deviations from a reasonably strong baseline are unlikely to uncover informative improvements, making efficient exploration particularly challenging in

the B&B setting.

Despite these hurdles, we believe that building on a principled MDP formulation of variable selection in B&B is an essential stepping stone to achieve substantial acceleration of solving time for higher-dimensional MILPs in the future. Inspired by the success of AlphaGo [153] and its offspring [152, 154] in complex combinatorial games, in Part III we propose to mitigate both exploration and out-of-distribution issues by adapting planning-based search procedures to the B&B setting.

Part III

Planning in branch-and-bound

Chapter 6

Preliminaries

Content

As highlighted in Section 5.6, even when relying on a principled MDP formulation of variable selection in B&B, traditional RL agents fail to surpass the performance achieved by imitation learning agents trained to replicate the behaviour of strong branching. Importantly, this limitation is not specific to mixed-integer linear programming: across various CO benchmarks [23, 58], traditional RL approaches consistently underperfom either handcrafted heuristics or IL methods trained to mimick these experts. One of the main challenges lies in the high dimensionality and combinatorial complexity of CO problems, which exacerbates exploration and credit assignment issues in sequential decision making and prevent (deep) RL agents from reliably converging to high-performing regions of the parameter space. While supervised learning can partially alleviate these issues by scaling up neural architectures and exploiting abundant labeled data, such approaches are impractical in RL due to sample inefficiency and unstable learning dynamics [112].

Despite these challenges, RL has managed to achieve remarkable success in a restricted subset of combinatorial problems: board games. Notably, by leveraging environment simulators to perform model-based planning, AlphaZero [154] established a new performance frontier in the game of Go, surpassing human world champions for the first time. Within this approach, the integration of learned policy and value functions with look-ahead search procedures like Monte Carlo Tree Search was found to steer action selection towards high-value regions of the state space, effectively mitigating traditional exploration and credit assignment issues arising in sparse reward environment settings. Taking inspiration from these advances, we propose to build upon the theoretical framework introduced in Chapter 5 to adapt AlphaZero-style agents [152] to the branch-and-bound setting. To this end, the present chapter introduces the necessary background on planning and model-based reinforcement learning (MBRL) that underpins the algorithms developed in Chapter 7.

6.1 Model-based reinforcement learning

6.1.1 Definition

Model-based reinforcement learning (MBRL) denotes the class of approaches that exploit an explicit *model* of the environment to improve decision making. In contrast to model-free RL, which learns primarily from irreversible interaction with the MDP, MBRL leverages the ability to simulate

alternative[1] futures, thereby implementing policy improvement through planning. In our deterministic setting, a model may be identified with the pair of transition and reward functions $(\mathcal{T}, \mathcal{R})$, or any equivalent parameterization enabling the agent to predict, at least approximately, the downstream consequences of candidate actions.

This viewpoint highlights the fundamental distinction between model-based and model-free methods: while the two paradigms share the same objective, learning high-return policies, they differ in how computation is ultimately allocated. Model-free methods seek to amortize decision making into parametric approximators (policy and value functions), whereas MBRL explicitly spends computation at decision time to evaluate multiple future scenarios under its environment model. As a result, MBRL naturally introduces a trade-off between data and compute: when interaction is expensive but simulation is cheap (or can be made cheap through learned surrogates), planning can extract substantially more learning signal per real environment transition.

6.1.2 Theoretical motivation

A convenient formalization of model-based reinforcement learning is through approximate policy iteration [24, 104].[2] Let π denote the current (stochastic) policy, and let Q^π be its associated state-action value function. In principle, policy improvement reduces to identifying

$$\pi^+(s) \in \arg\max_{a \in \mathcal{A}} \; Q^\pi(s, a),$$

but evaluating Q^π accurately is generally intractable in large state spaces. MBRL circumvents this by constructing a computable surrogate of Q^π using a dynamics model and a value estimate. Concretely, given (possibly learned) transition and reward models $(\hat{g}_\phi, \hat{r}_\psi)$ and a terminal value estimate $\hat{v}_\omega$ (possibly a learned value function), one may define the finite-horizon look-ahead state-action value

$$\hat{Q}^{(H)}_{\phi,\omega,\psi}(s, a) \;=\; \mathbb{E}_{\hat{g}_\phi}\left[\sum_{t=0}^{H-1} \gamma^t \hat{r}_\psi(s_t, a_t) \;+\; \gamma^H \hat{v}_\omega(s_H) \;\middle|\; s_0 = s,\; a_0 = a\right], \tag{6.1}$$

[1] In the MBRL literature, simulated trajectories are also referred to as *imagined* rollouts or, more informally, *dreamed* trajectories.

[2] Approximate policty iteration (API) generalizes the approximate dynamic programming scheme of Section 4.1.3 to the policy iteration setting, where an agent alternates between evaluating the current policy and improving it greedily with respect to the estimated value function.

where actions $(a_t)_{t\geq 1}$ are selected by a proposal policy (e.g., the current policy network) and the trajectory $(s_t)_{t\geq 1}$ is generated by $\hat{g}_\phi$. Planning can then be viewed as approximately solving

$$\arg\max_{a\in\mathcal{A}} \hat{Q}^{(H)}_{\phi,\omega,\psi}(s,a),$$

or a softened variant, yielding a policy improvement operator whose accuracy is controlled by the planning budget, the horizon H, and the fidelity of the learned components.

In combinatorial settings such as the one considered in this thesis, the central difficulty lies in deriving policies that consistently yield high-quality decisions, while learning from sparse, delayed feedback over long episodes and under an enormous branching factor. Typically, model-free methods must discover good trajectories through trial-and-error interaction with the environment, which becomes increasingly sample-inefficient as the branching factor grows. Planning alleviates this difficulty by enabling the agent to evaluate multiple candidate action sequences before committing to a real decision. By simulating future trajectories under its model, the agent can compare the anticipated consequences of different choices at each state, producing more informed action selections than a policy network alone would provide. From this perspective, MBRL trades additional computation at decision time, through explicit search within a model, for more reliable policy improvement in large discrete spaces.

Importantly, MBRL spans a spectrum. At one extreme, the dynamics are known explicitly and cheap to simulate (e.g., board games), and planning is performed directly in the true environment model. At the other extreme, the dynamics are unknown or expensive to query, and the agent learns a surrogate model from experience, planning in an approximate world that may be accurate only in limited regions of the state space. Interestingly, branch-and-bound sits in-between these regimes. While in principle, the transitions induced by a branching decision are deterministic and fully specified by the solver's rules, in practice, simulating branching steps entails substantial solver work (LP solves, separation rounds, propagation, and other solver side-routines), making look-ahead planning with the true solver prohibitively expensive beyond very shallow depth, and therefore impractical for integration into a MBRL training pipeline. This motivates learning a cheaper model of the B&B dynamics, as done for instance in MuZero [152], in order to enable broader and deeper search under fixed computational budget.

In the remainder of this chapter, we use *planning* to denote any procedure that, given access to a

model and a current state, returns a distribution over available actions by evaluating multiple simulated trajectories. This includes Monte-Carlo Tree Search (MCTS) as used in AlphaZero and MuZero, as well as receding-horizon planning schemes common in continuous control [175]. The next section introduces the planning search procedures most relevant to the algorithms introduced in Chapter 7.

6.2 Planning in model-based reinforcement learning

Planning algorithms can be organized by how they approximate the Bellman optimality equations. Classical dynamic programming applies Bellman backups over the full state space, which is infeasible for the problems considered in this thesis. Modern planning in large spaces instead constructs a partial search structure (often a tree rooted at the current state), using sampling and learned heuristics to allocate computation adaptively.

At a high level, such a forward-search planning routine can be decomposed into three components [29]: (i) a proposal mechanism deciding which actions, and, recursively, which trajectories to evaluate, (ii) an evaluation mechanism providing value estimates for partially explored trajectories, and (iii) a policy extraction rule that aggregates the statistics resulting from local search into an improved action distribution at the root.

6.2.1 Monte-Carlo Tree Search

Monte-Carlo Tree Search is a family of sampling-based planning algorithms that scale to large branching factors by selectively expanding promising regions of the search space. Starting from a root node representing the current state, MCTS incrementally builds a search tree in which nodes correspond to states and edges to actions. At any point during the search, this tree covers only a subset of the reachable state space. The tree is grown sequentially by iterating four phases: **selection** (traverse the current tree using an exploration rule), **expansion** (add new node to the tree), **evaluation** (estimate the value of the newly added node, via rollouts or a value network), and **backup** (propagate the evaluation back to update the statistics along the path). Each such iteration of selection, expansion, evaluation and backup is referred to as an MCTS simulation.

Selection. For each node s and outgoing edge (s, a) in the tree, MCTS maintains summary statistics collected from previous simulations. Let $N(s)$ be the visit count of state s, $N(s, a)$ the number of times

action a was selected at s, and $Q(s,a)$ the mean return observed across simulations in which action a was selected at s. A canonical selection rule is the upper-confidence bound (UCB) [11]. Applying UCB to tree search yields the UCT algorithm [99], which balances exploitation through empirical value estimation with exploration through a confidence bonus. UCB selects edge (s,a) based on:

$$a \in \arg\max_a \; Q(s,a) \; + \; c\sqrt{\frac{\log N(s)}{1+N(s,a)}}. \tag{6.2}$$

When a policy network provides a prior $P(s,a)$, the search procedure commonly replaces Eq. (6.2) by a prior-weighted exploration term, thus implementing a PUCT [145] variant:

$$a \in \arg\max_a \; Q(s,a) \; + \; P(s,a) \cdot \frac{\sqrt{N(s)}}{1+N(s,a)} \cdot \left(c_1 + \log\left(\frac{N(s)}{c_2}\right)\right). \tag{6.3}$$

Expansion and evaluation. An MCTS simulation starts at the root s^0 and repeatedly applies the selection rule until it reaches a leaf state, i.e., a state that is either terminal or not yet expanded. Let $(s^0, a^1, r^1, s^1, \ldots, s^{l-1}, a^l, r^l, s^l)$ denote a simulated path, where s^l is the leaf. Typically, (s^l, r^l) is obtained by feeding (s^{l-1}, a^l) to the model. At that point, the tree is expanded by initializing the edges (s^l, a) exiting the selected leaf node, and adding to the tree the (new) leaf nodes associated with these edges. The newly expanded node s^l is evaluated by an estimator that returns a scalar value prediction v^l together with a policy prior p^l over its available actions. Each edge (s^l, a) exiting the newly expanded node is then initialized to

$$N(s^l,a) = 0, Q(s^l,a) = 0, P(s^l,a) = \left[\mathrm{p}^l\right]_a.$$

Each MCTS simulation requires at most one model transition (to obtain (s^l, r^l) from (s^{l-1}, a^l)), and one evaluation round (to compute $(\mathrm{p}^l, \mathrm{v}^l)$ estimates from s^l), keeping the per-simulation cost controlled.

Backup. After leaf evaluation, MCTS updates the statistics along the simulated trajectory. A standard backup forms, for each depth $k \in \{1, \ldots, l\}$, a bootstrapped return

$$G_k \; = \; \sum_{\tau=0}^{l-k} \gamma^\tau r^{k+\tau} \; + \; \gamma^{\,l-k} \mathrm{v}^l, \tag{6.4}$$

and then updates each traversed edge (s^{k-1}, a^k) by incrementing its visit count and updating its empirical value estimate via a running average:

$$Q(s^{k-1},a^k) \leftarrow Q(s^{k-1},a^k) + \frac{G^k - Q(s^{k-1},a^k)}{N(s^{k-1},a^k)}, \qquad N(s^{k-1},a^k) \leftarrow N(s^{k-1},a^k) + 1. \tag{6.5}$$

Root policy extraction. Finally, after running B simulation rounds, corresponding to the available simulation budget, MCTS returns an improved root policy, typically extracted from visit counts at the root:

$$\pi_{\text{MCTS}}(a \mid s^0) \propto N(s^0, a)^{1/\tau}, \tag{6.6}$$

where $\tau > 0$ controls exploration during data generation. This visit-count policy can then be used both for action selection and as a supervised target for the policy network.

6.2.2 Gumbel Search

Vanilla MCTS exhibits several limitations when used as a planning procedure in a model-based reinforcement learning pipeline. In fact, in traditional MBRL applications, the planning budget is concentrated at the root of the search tree, prior to committing to a single real action. As emphasized by Danihelka et al. [42], the appropriate objective in such regimes is that of simple regret: the planning objective consists in identifying a high-value root action given a limited number of simulations, rather than optimizing the sequence of actions sampled within the search itself, as done in MCTS. Moreover, Danihelka et al. [42] show that, when $|\mathcal{A}(s^0)| \gg B$[3], classical MCTS implementations may fail to act as reliable policy improvement operators. Because the search does not visit a sufficiently diverse set of root actions, the induced policy target becomes highly sensitive to heuristic design choices. Unfortunately, such small budget regime is precisely the one encountered in B&B, as high-quality decisions must be produced while devoting as little additional computation as possible to online planning.

Motivated by this observation, Danihelka et al. [42] propose a root planning mechanism explicitly tailored to small simulation budgets, replacing several ad hoc exploration heuristics with two principled components: (i) sampling a diverse set of candidate actions without replacement from the root policy prior, and (ii) selectively allocating the available planning budget across these candidates via successive elimination.

Gumbel Top-m sampling. Concretely, let $\text{logits}(a) = \log \pi(a \mid s^0)$ denote the log-probabilities of the root prior policy, commonly referred to as *logits*. Their approach consists in sampling without replacement a subset of m candidate actions, which is achieved by drawing, for each action a, an

[3] i.e. when the simulation budget is small in front of the number of candidate actions available at the root,

independent sample $g(a) \sim \text{Gumbel}(0)$ from the standard Gumbel distribution.[4] Candidate actions are then ranked by $\text{logits}(a) + g(a)$ and the m-best are retained. The resulting Gumbel-Top-m set

$$A_{\text{top-}m} = \arg_{a \in \mathcal{A}} \text{top}(\text{logits}(a) + g(a),\, m),$$

forms a principled candidate subset: it remains biased toward high-probability actions under the policy prior while inducing diversity across episodes.

Sequential Halving. Once the candidate set is fixed, Danihelka et al. [42] allocate simulations using sequential halving [93]: candidates are evaluated in successive rounds, and after each round the worst-performing half is eliminated, concentrating the remaining budget on the most promising candidates. Specifically, at the end of each round, remaining candidates are sorted using a scoring function of the form

$$\text{logits}(a) + g(a) + \sigma(Q(s^0, a)), \tag{6.7}$$

where $Q(s^0, a)$ denotes the empirical state-action value estimate accumulated throughout the search and σ controls the influence of value estimates relative to the prior.[5]

Embedding this root selection mechanism within an otherwise standard Monte Carlo Tree Search procedure yields Gumbel Search, a planning algorithm designed to provide robust policy improvement in MBRL even when only few simulations can be afforded per decision. This perspective will be reused in Chapter 7 to design planning procedures compatible with tight computational budgets in B&B.

6.3 Model-based reinforcement learning for board games

Although model-based reinforcement learning has become a central paradigm in continuous-control settings [76], where learned dynamics models are routinely combined with short-horizon planning or model predictive control optimization [77, 78], its most visible breakthroughs in the deep learning era were first achieved in discrete combinatorial domains, and in particular in board games. In the present section, we introduce both AlphaZero and MuZero frameworks, which will serve as the main conceptual and algorithmic building blocks for the methods developed in Chapter 7.

[4] The standard Gumbel distribution has cumulative distribution function $F(x) = e^{-e^{-x}}$. Adding independent Gumbel noise to log probabilities and taking the argmax is equivalent to sampling from the corresponding categorical distribution [116], a property known as the Gumbel-max trick.

[5] Typically, σ implements $\sigma(Q(s,a)) = (c_{visit} + \max_{a \in \mathcal{A}} N(s,a)) \cdot c_{scale} \cdot Q(s,a)$ where c_{visit} and c_{scale} are search hyperparameters.

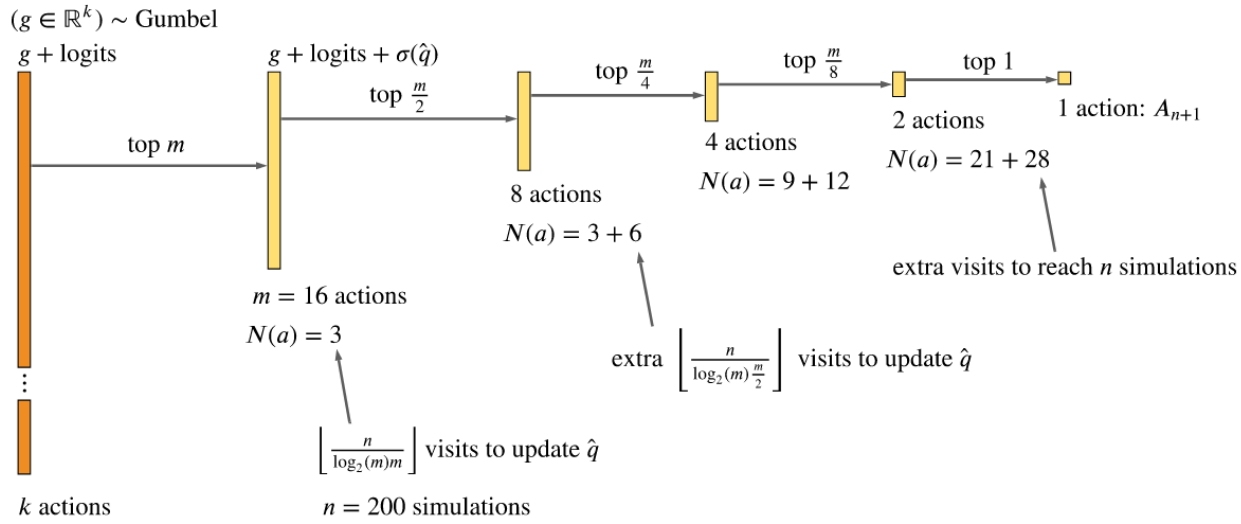


Figure 6.1: The number of considered actions and their visit counts $N(a)$ at the end of each round, when using sequential halving on a k-armed stochastic bandit. In this example search uses $n = 200$ simulations and starts by sampling $m = 16$ actions without replacement. Sequential Halving divides the budget of n simulations equally to $\log_2(m)$ phases. In each round, all considered actions are visited equally often. After each round, one half of the actions is rejected based on the score defined in Eq. (6.7). Excerpt from Danihelka et al. [42].

6.3.1 AlphaZero

AlphaZero [154] is a model-based reinforcement learning algorithm that combines MCTS planning with neural policy and value function approximation. It was introduced in board games such as chess and Go, where the dynamics are known and cheap to simulate. In this setting, access to an accurate simulator supports iterative self-play training: MCTS produces improved decisions that serve as stable policy targets, enabling steady performance gains over successive updates.

Function approximation. For a given visited state s_t, AlphaZero's prediction network f_θ computes a policy prior and a value estimate $(\hat{\mathrm{p}}_t, \hat{\mathrm{v}}_t) = f_\theta(s_t)$, where $\hat{\mathrm{p}}_t$ is a categorical distribution over legal actions and $\hat{\mathrm{v}}_t$ estimates the expected outcome of the game (or, more generally, the expected return) from s_t under the agent's current decision rule. The policy prior is used to steer exploration during the search, while the value prediction replaces costly rollouts providing the bootstrap evaluation used in both planning and training.

Self-play data generation. AlphaZero leverages f_θ to generate training data by self-play. At each encountered state s_t, it runs MCTS to obtain a target policy π_t, and selects an action by sampling from this distribution.[6] The resulting episode yields a terminal outcome $z \in \{-1, +1\}$ (or a terminal return), which serves as the supervised target for the value head of the prediction network. Each timestep thus produces a training tuple (s_t, π_t, z) that is stored for subsequent training. To encourage exploration at the start of training, AlphaZero injects Dirichlet noise into the root prior during self-play, thereby diversifying the trajectories collected in the replay buffer.

Learning objective. The prediction network is trained on episodes generated by self-play. On each training batch, the network is updated to match the search-improved policy returned by MCTS and to predict the game's final outcome:

$$\mathcal{L}(\theta) = \underbrace{(\hat{\mathrm{v}}_t - z)^2}_{\text{value regression}} + \underbrace{(-\pi_t^\top \log \hat{\mathrm{p}}_t)}_{\text{policy loss}}. \tag{6.8}$$

The self-play training framework outlined above allowed AlphaZero to exploit access to large-scale computational resources to continually generate fresh training data while optimizing its prediction network. This closed loop of planning and learning proved able to steadily strengthen the performance of self-play agents, ultimately reaching unprecedented levels of performance in Chess, Go, and Shogi. In principle, one could envision training an AlphaZero-style agent for learning optimal branching policies by using a B&B solver as a simulator. In practice, however, running online search within the solver at the scale required by self-play would be impractical under the computational resources available in the context of this thesis. This motivates turning to the MuZero framework, which learns an internal dynamics model alongside a prediction network, enabling look-ahead search to be carried out in a cheaper latent space.

6.3.2 MuZero

In order to extend the applicability of MCTS-based RL algorithms to broader control tasks where efficient simulators are not available, Schrittwieser et al. [152] introduced MuZero as a successor to AlphaZero. A defining design choice in MuZero is to forgo learning a model that faithfully simulates

[6] Crucially, MCTS relies on the game simulator to expand the search tree and on the prediction network to supply policy priors and leaf value estimates. After B simulations, the visit counts root policy defined in Eq. (6.6) yields policy target π_t.

the environment dynamics by accurately predicting next visited state observations, and instead learn a model tailored for planning. Specifically, MuZero learns a latent dynamics model that can be unrolled for a few steps to predict quantities relevant to MCTS, thereby enabling planning directly in latent space. In this sense, the model is best understood as an abstract, value-equivalent surrogate of the real environment: it need not reconstruct future observations, but only preserve the return-relevant structure that determines action preferences.

Concretely, MuZero learns a model consisting of three interconnected networks. First, the representation network h_θ maps raw state observations s_t to a latent state $\hat{s}_t = h_\theta(s_t)$. This internal state can then be passed to the prediction network f_θ to obtain state policy and value estimates $(\hat{\mathrm{p}}_t, \hat{\mathrm{v}}_t)$. Alternatively, latent states can be passed along with an action a_t to the dynamics network g_θ, to produce predictions for both the true reward r_t and the internal representation $h_\theta(s_{t+1})$ of the next visited state when taking action a_t in s_t: $g_\theta(\hat{s}_t, a_t) = (\hat{r}_t, \hat{s}_{t+1})$. The overall architecture of MuZero is summarized in Figure 6.2. At each decision step, self-play agents use h, f and g to perform MCTS, thereby generating improved policy targets π_t from which actions are sampled.[7] During training, the model is unrolled for K hypothetical steps and its predictions are aligned with sequences sampled from real trajectories collected by MCTS actors. The overall training loss used in Schrittwieser et al. [152] writes

$$\mathcal{L}_t(\theta) = \sum_{k=0}^{K} \underbrace{\left(-\, z_{t+k}{}^\top \log \hat{\mathrm{v}}_t^k\right)}_{\text{value loss}} + \underbrace{\left(-\, r_{t+k}{}^\top \log \hat{r}_t^k\right)}_{\text{reward loss}} + \underbrace{\left(-\, \pi_{t+k}{}^\top \log \hat{\mathrm{p}}_t^k\right)}_{\text{policy loss}} \tag{6.9}$$

where z_{t+k} correspond to bootstrapped return targets, and r_{t+k} is observed from past episodes.[8] By enforcing consistency between the predictions observed along simulated rollouts and those observed on real trajectories, MuZero constrains its model to capture only the information most relevant for accurately estimating future states' policy, value and step-wise rewards.

This approach was shown to significantly reduce the burden of MDP dynamics approximation, allowing to deploy MBRL in visually complex domains where model-free RL methods had previously dominated, while retaining, and even surpassing, AlphaZero-level performance on board games. Subsequent work introduced several enhancements to the MuZero framework, including an auxiliary

[7] To lighten notations, going forward, we omit the parameter index θ for neural networks.

[8] Scalar rewards and value estimates are represented as categorical distributions over a fixed support, allowing the reward and value heads to be trained via cross-entropy classification, as has become common practice following the work of Silver et al. [154].

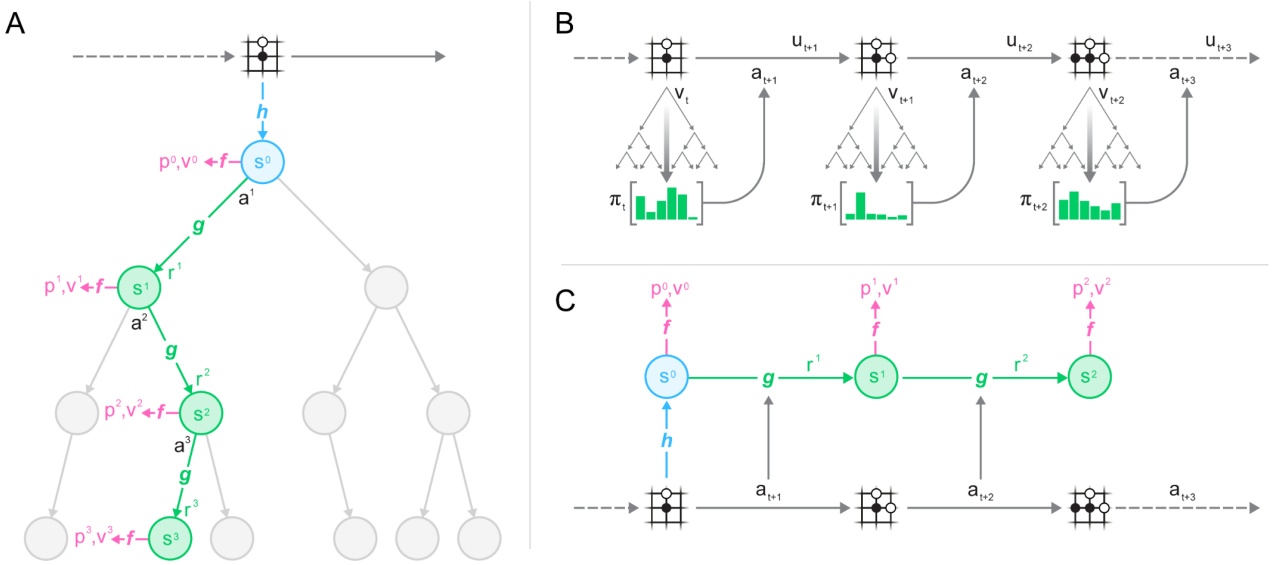


Figure 6.2: **(A) Search.** At each decision step, MuZero integrates h, f and g to perform MCTS as described in Section 6.2, thereby generating improved policy targets π_t. **(B) Act.** Actions are sampled from the search policy π_t, which is proportional to the visit count for each available action at the root node. **(C) Learn.** During training, the model is unrolled for K hypothetical steps (here, $K = 2$) and its predictions are aligned with sequences sampled from real trajectories. The model is then updated following the loss described in Eq. (6.9). Excerpt from Schrittwieser et al. [152].

temporal-consistency loss to accelerate the training of the dynamics network [184] under limited computational budget, and the use of Gumbel search [42], which proved particularly useful in environments with large action space.

6.3.3 Existing adaptations to combinatorial optimization

While many contributions have sought to leverage traditional model-free reinforcement learning to learn improved CO heuristics, both within exact [56, 156, 173], and approximate [32, 70, 101] resolution frameworks, attempts to develop RL methods that explicitly integrate model-based planning remain limited in comparison. In the context of MILP and B&B, we were simply unable to identify any prior reference matching this description. By contrast, in the context of approximate resolution, a small line of work has explored learning planning-augmented heuristics. In fact, several contributions have investigated leveraging Monte Carlo Tree Search, either coupled with function approximation [171, 182] or not [31], to guide the discovery of high-quality feasible solutions in routing problems. More recently, also drawing inspiration from model-based planning, Pirnay and Grimm [137] introduced

a self-improvement operator that refines a learned policy by generating alternative trajectories via stochastic beam search[9], and treating improved trajectories as targets for imitation learning. Applied to both routing and scheduling problems, their approach matched the performance of imitation learning agents trained on expert demonstrations, highlighting the potential of planning-based methods as a data-efficient alternative to expert supervision in combinatorial optimization.

The next chapter builds on these observations and introduces PlanB&B, a model-based reinforcement learning framework adapting MuZero to the B&B setting.

[9]Beam search [148] is a population-based look-ahead procedure that incrementally constructs solutions while maintaining a beam of the m most promising partial trajectories at each depth, pruning the rest. Stochastic beam search introduces randomized, diversity-promoting selection to sample candidate continuations without replacement while remaining biased toward high-probability trajectories [100].

Chapter 7

Model-based reinforcement learning for exact combinatorial optimization

Content

Since the beginning of the deep learning era, model-based reinforcement learning has first achieved its most striking successes in complex combinatorial board games, where access to cheap and accurate simulators enable tight integration between learning and planning. Several of the core ideas underlying these results have since been adapted to continuous control settings, leading to strong MBRL performance across a broad range of both synthetic and real-world continuous benchmarks [76, 78]. By contrast, comparatively few contributions have investigated how AlphaZero-style frameworks might be adapted to improve the resolution of combinatorial optimization problems, in spite of the fact that these methods were originally specifically developed to tackle discrete, combinatorial problems.

The present chapter aims to help bridge this gap. Inspired by the work of Schrittwieser et al. [152], we seek to extend the applicability of MCTS-based RL algorithms from board games to exact combinatorial optimization. To that end, we introduce Plan-and-Branch-and-Bound (PlanB&B), a model-based reinforcement learning agent that leverages an internal model of the B&B dynamics to learn improved variable selection strategies (Section 7.1). To the best of our knowledge, this is the first MBRL agent specifically designed to solve CO problems without assuming access to a cheap, accurate simulator. Our PlanB&B agent achieves state-of-the-art performance on test sets across the Ecole benchmark [139], surpassing prior IL and RL baselines (Section 7.2). Furthermore, to assess the potential of PlanB&B beyond synthetic benchmarks, Section 7.3 considers real-world MILP instances arising from an EDF industrial use case. In this setting, PlanB&B matches IL baselines on smaller training instances and delivers clear performance gains on higher-dimensional, industrially relevant instances. These results suggest that the branching dynamics in B&B can be approximated with sufficient fidelity in latent space to enable policy improvement through planning over a learned model, opening the door to broader applications of MBRL to mixed-integer linear programming.

We mention that the work presented in Sections 7.1 and 7.2 has been published in the *Proceedings of the 40th Annual AAAI Conference on Artificial Intelligence* [161], while the results presented in Section 7.3 are original to this thesis.

7.1 Planning in Branch-and-Bound

In BBMDP (see Section 5.1), rewards are constant $\mathcal{R} = -1$, and transitions are deterministic. In particular, the transition function $\mathcal{T} = \kappa_\rho$ is defined as the composition of the branching operator κ, and the node selection policy ρ. In this context, taking a step in the environment entails solving the two linear programs associated with the nodes generated by the branching decision. Unfortunately, this procedure is expensive to simulate, and remains difficult to approximate accurately [141]. In order to overcome these limitations, we introduce Plan-and-branch-and-bound (PlanB&B), a model-based reinforcement learning agent adapting the MuZero framework to the B&B setting. Crucially, our learned model is not explicitly trained to solve linear programs. Instead, it learns the dynamics of an abstract, value-equivalent MDP that retains only the aspects of B&B essential for enabling policy improvement via look-ahead search. In practice, PlanB&B approximates only the branching component κ, while the node selection policy ρ and rewards are simulated. As a result, the agent plans using a transition model that is partially learned and partially exact, making PlanB&B a hybrid between AlphaZero (fully simulated dynamics) and MuZero (fully learned dynamics).

7.1.1 General description

Adopting the BBMDP formulation, let $s_t = (\mathcal{V}_t, \mathcal{E}_t, \bar{x}_t) \in \mathcal{S}$ be a B&B tree where $\mathcal{V}_t = \mathcal{O}_t \cup \mathcal{C}_t$, and let $o = \rho(s_t) \in \mathcal{O}_t$ be the node currently selected for expansion under DFS. MuZero is originally designed to operate on full state observations s_t as input to its internal model. In contrast, prior RL and IL branching agents have been designed to rely solely on information from the current B&B node, represented by the pair $(o, \bar{x}_t)$, and encoded using the MILP bipartite graph representation function from Gasse et al. [62]. While, under DFS, this local information is sufficient to recover the optimal policy at state s_t, it is generally insufficient to infer the subsequent state s_{t+1}. In fact, whenever taking action a_t at s_t leads to fathoming the subtree under o, the next visited node will be a leaf node that cannot be deduced directly from the triplet $(o, \bar{x}_t, a_t)$. To address this, our model learns to predict recursively, **along subtree trajectories**, the internal representations $(\hat{o}_l, \hat{o}_r)$ associated with the left and right child nodes (o_l, o_r) generated by the agent's branching decisions.[1] Doing so, PlanB&B enables simulating imagined subtree trajectories while relying exclusively on the information available at the

[1] Following our convention, in DFS, o_l designates the node selected immediately next for expansion, while o_r designates the node selected once the subtree rooted in o_l has been fathomed.

current node.

The representation network h first maps the pair $(o, \bar{x}_t)$ to an internal representation $\hat{o}$ which serves as the root node for the imagined B&B tree $\hat{\mathrm{T}} = (\hat{\mathcal{O}}, \hat{\mathcal{C}})$. Initially, $\hat{\mathcal{O}} = \{\hat{o}\}$ and $\hat{\mathcal{C}} = \emptyset$. The current imagined node $\hat{o}$ can then be passed to the dynamics network g along with any action in $\mathcal{A}$ to generate $(\hat{o}_l, \hat{o}_r)$. Since $r_t = -1$ is constant in BBMDP, g is not tasked with predicting the future reward. However, note that if either of the real child nodes is pruned, either by bound, integrity or infeasibility, its associated subtree value is null, and, consequently, its node internal representation should not be considered for expansion by the model. To distinguish nodes leading to branching decisions from nodes destined to be pruned, we task the prediction network f with estimating future nodes' *branchability* $\mathrm{b} \in \{0, 1\}$, in addition to policy and subtree value estimates[2]. Based on the predicted branchability values, PlanB&B updates the sets $(\hat{\mathcal{O}}, \hat{\mathcal{C}})$ by discarding unbranchable nodes. If $\hat{\mathcal{O}} = \emptyset$, the imagined subtree has been fully explored, and all subsequent recursive calls to g will, by convention, receive a null reward. In-depth model description and network architectures for h, f and g are presented in Section 7.1.3.

7.1.2 Simulating subtree trajectories

The overall interaction between the networks h, f, and g during the simulation of a k-step subtree trajectory is illustrated in Figure 7.1. We now describe this procedure formally. Let $\hat{o} \in \mathcal{H}$ be the internal representation of the B&B current node $o = \rho(s_t)$ and let $a \in \mathcal{A}$ be an action to perform. The dynamics network g generates $\hat{o}_l, \hat{o}_r$ the internal representations of the two child nodes created when branching on variable a at s_t: $\hat{o}_l, \hat{o}_r = g(\hat{o}, a)$. The policy, subtree value and branchability associated with $\hat{o}_l, \hat{o}_r$ are computed using the prediction network: $\mathrm{p}_{\hat{o}_l}, \bar{\mathrm{v}}_{\hat{o}_l}, \mathrm{b}_{\hat{o}_l} = f(\hat{o}_l)$; $\mathrm{p}_{\hat{o}_r}, \bar{\mathrm{v}}_{\hat{o}_r}, \mathrm{b}_{\hat{o}_r} = f(\hat{o}_r)$. Then, the two child nodes are added to the imagined tree $\hat{\mathrm{T}} = \{\hat{\mathcal{O}}, \hat{\mathcal{C}}\}$ based on their predicted branchability. If $\hat{o}_i \in \{\hat{o}_l, \hat{o}_r\}$ is predicted to be branchable, it is added to the set of imagined open nodes $\hat{\mathcal{O}}$, otherwise, it is added to the set of imagined closed nodes $\hat{\mathcal{C}}$. Crucially, the predictions $\mathrm{p}_{\hat{o}_i}, \bar{\mathrm{v}}_{\hat{o}_i}, \mathrm{b}_{\hat{o}_i}$ are stored in $\hat{\mathrm{T}}$ along with $\hat{o}_i$, allowing the look-ahead search to retrieve the policy and value estimates associated with the state corresponding to the imagined B&B tree expanded. For example, let $\hat{\mathrm{T}}^k = (\hat{\mathcal{O}}^k, \hat{\mathcal{C}}^k)$ be the imagined subtree obtained after unrolling the PlanB&B model for k steps from s_t. The imagined current node $\tilde{o} = \rho(\hat{\mathcal{O}}^k)$ is returned by the node selection policy simulator as the node in $\hat{\mathcal{O}}^k$ with the

[2]By construction, κ_ρ ensures that the current node at s_t is always branchable.

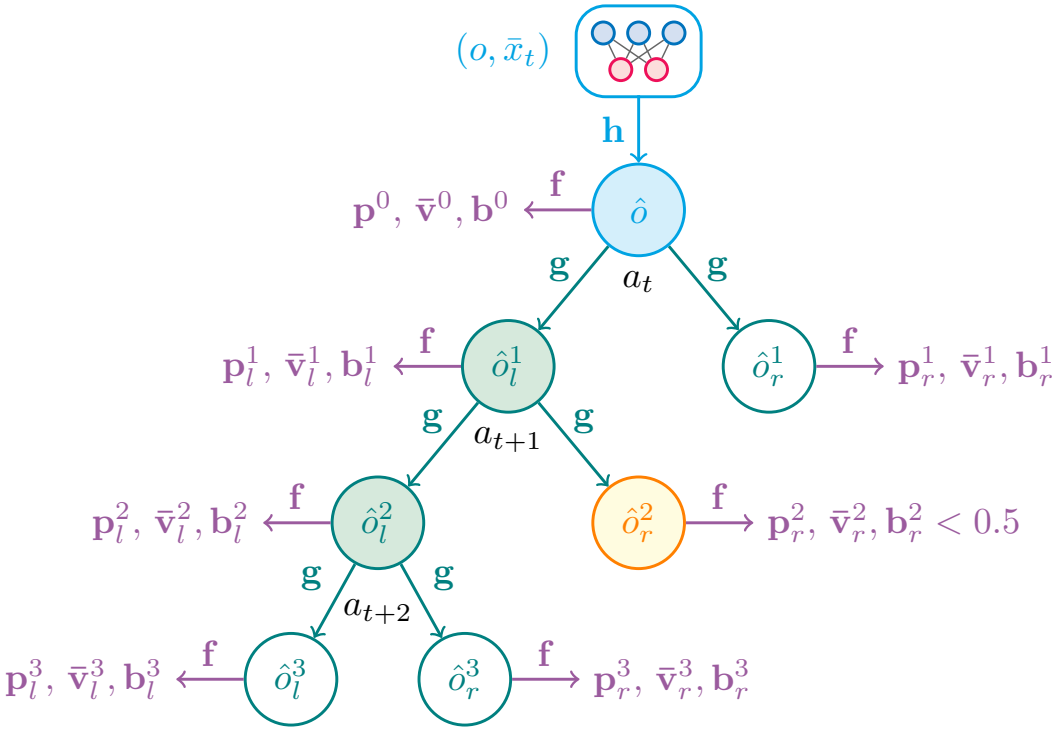


Figure 7.1: Planning in B&B over a learned model. The combined use of h, f, and g allows simulating subtree rollouts starting from the current B&B node. Here, $\hat{\mathrm{T}}^3 = (\hat{\mathcal{O}}^3, \hat{\mathcal{C}}^3)$ with $\hat{\mathcal{O}}^3 = \{\hat{o}_l^3, \hat{o}_r^3, \hat{o}_r^1\}$ and $\hat{\mathcal{C}}^3 = \{\hat{o}_r^2\}$. To simplify notations, we write z_j^i in place of $\mathrm{z}_{\hat{o}_j^i}$ for $\mathrm{z} \in \{\mathrm{p}, \bar{\mathrm{v}}, \mathrm{b}\}$.

greatest depth. In case of equality, the $\tilde{o}$ is chosen as the leftmost node in $\hat{\mathcal{O}}$ among those with maximal depth. This ensures compatibility with arbitrary DFS tie-breaking criteria, since the visitation order among sibling nodes of equal depth is implicitly determined by the dynamics network g. Finally, the state policy $\hat{\mathrm{p}}_t^k$ and value estimates $\hat{\mathrm{v}}_t^k$ associated with $\hat{\mathrm{T}}^k$ can be retrieved from $\hat{\mathcal{O}}^k$ through:

$$\hat{\mathrm{p}}_t^k = \mathrm{p}_{\tilde{o}}, \qquad \hat{\mathrm{v}}_t^k = \sum_{\hat{o} \in \hat{\mathcal{O}}^k} \bar{\mathrm{v}}_{\hat{o}}\,.$$

To simplify notations, in the following sections we write $\hat{\mathrm{p}}_t^k, \hat{\mathrm{v}}_t^k = f(\hat{\mathrm{T}}^k)$ and $\hat{\mathrm{T}}^{k+1} = g(\hat{\mathrm{T}}^k, a)$ to summarize the global interaction between f and g when simulating subtree trajectories.

7.1.3 Model architecture

Consistent with previous chapters, MILP inputs $(o, \bar{x}_t)$ are encoded as bipartite graphs $\mathcal{G} = (\mathcal{V}_\mathcal{G}, \mathcal{C}_\mathcal{G}, \mathcal{E}_\mathcal{G})$, where $\mathcal{V}_\mathcal{G}$ denotes the set of variable nodes, $\mathcal{C}_\mathcal{G}$ denotes the set of constraint nodes, and $\mathcal{E}_\mathcal{G}$ denotes the set of edges linking variable and constraint nodes. We note respectively d_v, d_c,

d_e the input dimension of variable nodes, constraint nodes and edges. Bipartite graphs thus can be interpreted as vectors from the observation space $\mathcal{M} = \mathbb{R}^{n \times d_v} \times \mathbb{R}^{m \times d_c} \times \mathbb{R}^{n \times m \times d_e}$.

Representation network The representation network $h : \mathcal{M} \to \mathbb{R}^{n \times d_h} \times \mathbb{R}^{m \times d_h} \times \mathbb{R}^{n \times m \times d_h}$, maps graph bipartite observations to a triplet of embeddings vectors of higher dimension d_h. It is composed of three fully-connected modules: the variable encoder $\mathbf{e}_v$, the constraint encoder $\mathbf{e}_c$ and the edge encoder $\mathbf{e}_e$, which independently compute initial embeddings for the corresponding graph input components. In the following, we note $\mathcal{H} = \mathbb{R}^{n \times d_h} \times \mathbb{R}^{m \times d_h} \times \mathbb{R}^{n \times m \times d_h}$ the latent space associated with internal representations.

Prediction network The prediction network f consists of a shared convolutional core followed by three output heads $\mathbf{p}, \bar{\mathbf{v}}, \mathbf{b}$, which maps internal representations $\hat{o} \in \mathcal{H}$ to their estimated policy, subtree value and branchability score respectively. In order to enhance its generalization capacity to higher-dimensionnal MILP instances with higher associated tree size, the value network represents subtree values as histogram distributions over the support of the value function. Crucially, histogram distributions outputted by $\bar{\mathrm{v}}$ can be transformed back and forth into scalar values using the HL-Gauss transform operators introduced in Section 5.2.

We now describe the overall architecture of the prediction network. The shared core comprises two graph convolutionnal layers $\mathcal{G}^f_{v \to c} : \mathcal{H} \to \mathcal{H}$ and $\mathcal{G}^f_{c \to v} : \mathcal{H} \to \mathcal{H}$ followed by an output module $\mathbf{o_m} : \mathcal{H} \to \mathbb{R}^{n \times d_h}$. This module consists in two fully connected layers followed by non-linear activations. Finally, the three output heads are computed by dedicated linear networks: $\mathbf{o_v} : \mathbb{R}^{n \times d_h} \to \mathbb{R}^{m_b}$, $\mathbf{o_p} : \mathbb{R}^{n \times d_h} \to \mathbb{R}^n$, $\mathbf{o_b} : \mathbb{R}^{n \times d_h} \to \mathbb{R}^2$ which produce the final predictions from the shared feature representation:

$$
\begin{aligned}
\mathbf{p} &: \begin{cases} \mathcal{H} \to \mathbb{R}^{m_b} \\ \mathbf{p}(\mathbf{x}) = \mathbf{o_p} \circ \mathbf{o_m} \circ \mathcal{G}^f_{c \to v} \circ \mathcal{G}^f_{v \to c}(\mathbf{x}) \end{cases} \\
\bar{\mathbf{v}} &: \begin{cases} \mathcal{H} \to \mathbb{R} \\ \bar{\mathbf{v}}(\mathbf{x}) = \mathrm{pool} \circ \mathbf{o_v} \circ \mathbf{o_m} \circ \mathcal{G}^f_{c \to v} \circ \mathcal{G}^f_{v \to c}(\mathbf{x}) \end{cases} \\
\mathbf{b} &: \begin{cases} \mathcal{H} \to \mathbb{R}^2 \\ \mathbf{b}(\mathbf{x}) = \mathrm{pool} \circ \mathbf{o_b} \circ \mathbf{o_m} \circ \mathcal{G}^f_{c \to v} \circ \mathcal{G}^f_{v \to c}(\mathbf{x}) \end{cases}
\end{aligned}
$$

where $\mathrm{pool}(\cdot)$ designates the mean pooling operation and m_b the number of bins used to partition the support of the value function, as defined in Section 5.2.

Dynamics network Our dynamic network g comprises two independent heads $\mathbf{g}^l$ and $\mathbf{g}^r$ dedicated respectively to the left and right child nodes. In our convention, the left node is the node first visited by the DFS node selection policy, while the right node is the node visited once its sibling node's subtree has been fathomed. We describe the architecture of $\mathbf{g}^l$, which is identical to the one of $\mathbf{g}^r$. The $\mathbf{g}^l$ module is composed of an action embedding module $\mathbf{e}^l_a$, two graph convolutionnal layers $\mathcal{G}^l_{v\to c}$, $\mathcal{G}^l_{c\to v}$ and an output module $\mathbf{o}^l_{\mathbf{g}}$. The action embedding module $\mathbf{e}^l_a : \mathcal{H} \times \mathcal{A} \to \mathcal{H}$ only transforms the representation of the variable node in the bipartite graph corresponding to the input variable $a \in \mathcal{A}$, by feeding it to a 2-layer fully connected neural network. The output of $\mathbf{e}^l_a$ is fed to the convolutionnal block $\mathcal{G}^l_{v\to c}$, $\mathcal{G}^l_{c\to v}$, and in turn to the output module $\mathbf{o}^l_{\mathbf{g}} : \mathcal{H}^2 \to \mathcal{H}$, which is also connected to the output of $\mathbf{e}^l_a$ through a residual link. While the dynamic network does not predict any reward, this could be incorporated to explore the use of alternative reward models. Finally, g can be summarized as:

$$\mathbf{g}^l : \begin{cases} \mathcal{H} \times \mathcal{A} \to \mathcal{H} \\ \mathbf{g}^l(\mathbf{x}) = \mathbf{o}^l_{\mathbf{g}} \circ (\mathbf{Id} + \mathcal{G}^l_{c\to v} \circ \mathcal{G}^l_{v\to c}) \circ \mathbf{e}^l_a(\mathbf{x}) \end{cases}$$
$$\mathbf{g}^r : \begin{cases} \mathcal{H} \times \mathcal{A} \to \mathcal{H} \\ \mathbf{g}^r(\mathbf{x}) = \mathbf{o}^r_{\mathbf{g}} \circ (\mathbf{Id} + \mathcal{G}^r_{c\to v} \circ \mathcal{G}^r_{v\to c}) \circ \mathbf{e}^r_a(\mathbf{x}). \end{cases}$$

7.1.4 Data generation

Since $\rho = DFS$, learning a policy minimizing the local subtree size is equivalent to learning a globally optimal policy. Therefore, we can use the model introduced in the previous sections to derive improved branching policy targets π_t through planning. In the following, we describe the planning algorithm used in PlanB&B.

In PlanB&B, every node of the look-ahead search tree is associated with an imagined B&B subtree $\hat{\mathrm{T}}$ rooted in the current B&B node $o = \rho(s_t)$. For each action $a \in \mathcal{A}$ available at node $\hat{\mathrm{T}}$, there is a corresponding edge storing the statistics $N(\hat{\mathrm{T}}, a), Q(\hat{\mathrm{T}}, a), P(\hat{\mathrm{T}}, a)$ respectively representing the visit count, normalized Q-value and policy prior associated with the pair $(\hat{\mathrm{T}}, a)$. Due to the large branching action space typically encountered in B&B, we adopt Gumbel search [42] to build improved policy targets, as detailed in Section 6.2.2. The resulting planning procedure follows the usual MCTS template: it iteratively performs path selection, node expansion/evaluation, and value backpropagation for a total of B simulations. Relative to vanilla MCTS, Gumbel search modifies only the selection

rule, while leaving the expansion and backup steps mostly unchanged.

Selection Each simulation consists in a path $(a^1, \dots, a^l)$ starting from the root of the search tree, represented by the initial subtree $\hat{\mathrm{T}}^0 = (\mathcal{O}^0, \mathcal{C}^0)$ where $\mathcal{O}^0 = \{\hat{o}\}$ and $\mathcal{C}^0 = \emptyset$, and diving towards a leaf node $\hat{\mathrm{T}}^l$ that is yet to be expanded. To generate such paths, Gumbel Search uses two different types of selection criterion for choosing a^k for $k = 1 \dots l$, depending on whether $\hat{\mathrm{T}}^{k-1}$ corresponds to the root or to a deeper node. Accordingly, at the root node, our search algorithm implements Sequential Halving [93] in combination with the Gumbel-Top-k trick [100] to efficiently sample a promising subset of $m < |\mathcal{A}|$ actions for exploration. In contrast, beyond root node, a^k is simply selected as :

$$a^k = \arg\max_{a \in \mathcal{A}} \left(\pi'(\hat{\mathrm{T}}^{k-1}, a) - \frac{N(\hat{\mathrm{T}}^{k-1}, a)}{1 + \sum_{b \in \mathcal{A}} N(\hat{\mathrm{T}}^{k-1}, b)} \right)$$

where $\pi'(\hat{\mathrm{T}}^{k-1}, a)$ is a function of the observed statistics $Q(\hat{\mathrm{T}}^{k-1}, a)$ and $P(\hat{\mathrm{T}}^{k-1}, a)$ introduced in Eq. (7.1) in the next paragraphs.

Expansion / evaluation At the final step of the simulation, upon reaching the unvisited edge $(\hat{\mathrm{T}}^{l-1}, a^l)$, the transition $\hat{\mathrm{T}}^l = g(\hat{\mathrm{T}}^{l-1}, a^l)$ is computed by applying the dynamics network. The policy and value estimates for $\hat{\mathrm{T}}^l$ are then obtained via the prediction network such that $\hat{\mathrm{p}}^l, \hat{\mathrm{v}}^l = f(\hat{\mathrm{T}}^l)$. The search tree is subsequently expanded by initializing each outgoing edge $(\hat{\mathrm{T}}^l, a)$ with the statistics

$$N(\hat{\mathrm{T}}^l, a) = 0, \quad Q(\hat{\mathrm{T}}^l, a) = 0, \quad P(\hat{\mathrm{T}}^l, a) = \left[\hat{\mathrm{p}}^l\right]_a .$$

Importantly, each simulation requires a single call to the dynamics function, and two calls to the prediction network.

Backpropagation Once the search tree is expanded, the statistics N and Q are updated along the simulation path, starting from $\hat{\mathrm{T}}^l$. For $k = l \dots 1$, noting $G^k = -(l-k) + \hat{\mathrm{v}}^l$, each edge $(\hat{\mathrm{T}}^{k-1}, a^k)$ is updated following:

$$Q(\hat{\mathrm{T}}^{k-1}, a^k) := \frac{N(\hat{\mathrm{T}}^{k-1}, a^k) \cdot Q(\hat{\mathrm{T}}^{k-1}, a^k) + G^k}{N(\hat{\mathrm{T}}^{k-1}, a^k) + 1}, \qquad N(\hat{\mathrm{T}}^{k-1}, a^k) := N(\hat{\mathrm{T}}^{k-1}, a^k) + 1.$$

Root improved policy target After completing B simulation steps, an improved policy target π_t is computed from the statistics accumulated at the root of the search tree, such that $\pi_t(a) = \pi'(\hat{\mathrm{T}}^0, a)$ for $a \in \mathcal{A}$, with π' defined as:

$$\pi'(\hat{\mathrm{T}}^k, a) = \mathrm{softmax}(P(\hat{\mathrm{T}}^k, a) + \sigma(\tilde{Q}(\hat{\mathrm{T}}^k, a))). \tag{7.1}$$

where $\sigma(\cdot)$ denotes the same transformation as in Section 6.2.2. In turn, $\tilde{Q}(\hat{\mathrm{T}}^k, a))$ is defined as :

$$\tilde{Q}(\hat{\mathrm{T}}^k, a)) = \begin{cases} Q(\hat{\mathrm{T}}^k, a) & \text{if } N(\hat{\mathrm{T}}^k, a) > 0 \\ \sum P(\hat{\mathrm{T}}^k, a) \cdot Q(\hat{\mathrm{T}}^k, a) & \text{else.} \end{cases}$$

Importantly, as done in MuZero, the Q-value statistics used to balance exploration and exploitation in both Eq. (6.7) and Eq. (7.1) are normalized $Q^\dagger \in [0, 1]$ values, computed using a simple normalization scheme applied across the search tree:

$$Q^\dagger(\hat{\mathrm{T}}^{k-1}, a^k) = \frac{Q(\hat{\mathrm{T}}^{k-1}, a^k) - \min_{(\hat{\mathrm{T}}, a) \in Tree} Q(\hat{\mathrm{T}}, a)}{\max_{(\hat{\mathrm{T}}, a) \in Tree} Q(\hat{\mathrm{T}}, a) - \min_{(\hat{\mathrm{T}}, a) \in Tree} Q(\hat{\mathrm{T}}, a)}.$$

Finally, by combining networks h, f and g to simulate subtree rollouts, Gumbel search yields an improved target policy π_t from which the selected action a_t can be sampled. Running this procedure in parallel over multiple self-play workers generates B&B episodes which are stored in a replay buffer, and subsequently used to train the model.

7.1.5 Learning

PlanB&B is trained over K-step subtree trajectories $(s_t, a_t, \ldots, s_{t+K})$ sampled from memory. As in MuZero, the model is unrolled from s_t over K steps along the same action sequence as the observed trajectory, thereby generating a sequence of imagined subtrees $(\hat{\mathrm{T}}^0, \ldots, \hat{\mathrm{T}}^K)$. As illustrated in Figure 7.2, for each step $k = 0, \ldots, K$, the model recursively predicts $\hat{\mathrm{p}}_t^k$ and $\hat{\mathrm{v}}_t^k$ along the imagined trajectory to approximate the search policy π_{t+k} and n-step return value target $z_{t+k} = -n + \hat{\mathrm{v}}_t^{k+n}$. As in MuZero, the value network is trained via classification, using the HL-Gauss loss described in Section 5.2. Additionally, the predicted branchability scores of imagined nodes are trained to match the true branchability labels of real nodes. The overall loss minimized in PlanB&B is provided in Eq. (7.2).

Tree consistency loss Taking inspiration from Ye et al. [184], we introduce a new self-supervised loss $\mathcal{L}_{\text{ssl}}$ designed to enforce temporal consistency between consecutive internal B&B node representations, with the goal of accelerating convergence of the learned dynamics under restricted computational budget. Let $\hat{o} \in \mathcal{H}$ be the internal representation associated with $o = \rho(s_t)$ the current B&B node. By definition, $\hat{o} = h(o, \bar{x}_t)$. When feeding $\hat{o}$ to the dynamics network g along with a_t, we obtain $\hat{o}_l, \hat{o}_r$,

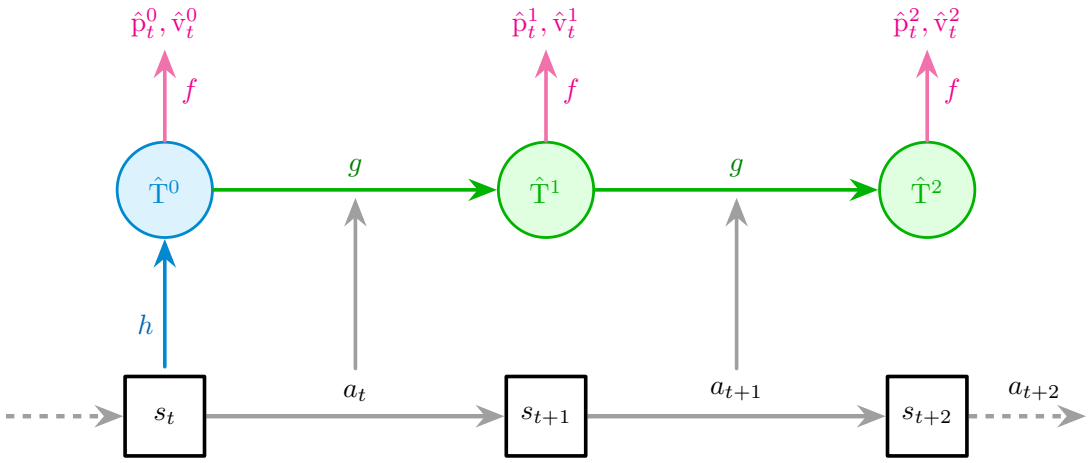


Figure 7.2: Learning in PlanB&B. The PlanB&B model is unrolled over K-step to produce K-step imagined trajectory. Along this trajectory, the policy network p is trained to replicate π_{t+k} the policy outputted by the search algorithm, while the subtree value network $\bar{v}$ is trained to match z_{t+k}, the n-step return target.

the internal representation of the child nodes associated with branching on a_t at s_t. Let us write τ_l, τ_r the time steps at which the real nodes o_l, o_r associated with $\hat{o}_l$, $\hat{o}_r$ are visited. Provided that these time steps are finite, we can compute the latent representation targets $h(o_l, \bar{x}_{\tau_l}), h(o_r, \bar{x}_{\tau_r}) \in \mathcal{H}$ for $\hat{o}_l$ and $\hat{o}_r$. Otherwise, by definition o_i is unbranchable ($\mathrm{b}_{o_i} = 0$), in that case we refrain from enforcing temporal consistency.

Our self-supervised loss $\mathcal{L}_{ssl}$ builds upon the SimSiam architecture from Chen and He [34]. Let $\hat{o} \in \mathcal{H}$ be the internal representation of node $o \in \mathcal{O}_{t+k}$ obtained after unrolling the PlanB&B model for k steps, and let $\check{o} \in \mathcal{H}$ be its associated target representation as defined in the previous paragraph. As illustrated in Figure 7.3, given $\mathbf{r}_{proj}$ and $\mathbf{r}_{pred}$ two fully connected block modules with associated hidden dimension d_{proj}, the self-supervised consistency loss writes:

$$\mathcal{L}_{\text{ssl}}(\check{o}, \hat{o}) = \text{sim}(\text{flat} \circ \text{sg} \circ \mathbf{r}_{proj}(\check{o}), \text{flat} \circ \mathbf{r}_{pred} \circ \mathbf{r}_{proj}(\hat{o}))$$

with flat$(\cdot)$ the flattening operation, sg$(\cdot)$ the stop-gradient operation and sim$(\cdot)$ the cosine similarity. Thus, the overall training objective associated with a trajectory $(s_t, a_t, ..., s_{t+K})$ is given by:

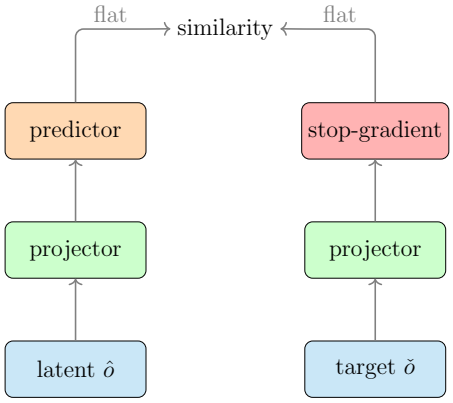


Figure 7.3: PlanB&B tree-consistency loss modeled after the SimSiam architecture from [34].

$$\mathcal{L}_t(\theta) = \frac{1}{K+1}\sum_{k=0}^{K} \lambda_{\mathrm{p}} \underbrace{\left(-\pi_{t+k}{}^{\top} \log \hat{\mathrm{p}}_t^k\right)}_{\text{policy loss}} + \lambda_{\mathrm{v}} \underbrace{\left(-z_{t+k}{}^{\top} \log \hat{\mathrm{v}}_t^k\right)}_{\text{value loss}} \tag{7.2}$$
$$+ \frac{1}{|\hat{\mathrm{T}}^K|} \sum_{\hat{o}\in\hat{\mathrm{T}}^K} \lambda_{\mathrm{b}} \underbrace{\left(-\mathrm{b}_o{}^{\top} \log \hat{\mathrm{b}}_{\hat{o}}\right)}_{\text{branchability loss}} + \lambda_{\mathrm{ssl}} \underbrace{\mathcal{L}_{\mathrm{ssl}}(\check{o}, \hat{o})}_{\text{consistency loss}}$$

where θ is the global parameter for f, g, h, and $\lambda_{\mathrm{p}}, \lambda_{\mathrm{v}}, \lambda_{\mathrm{b}}, \lambda_{\mathrm{ssl}}$ are hyper-parameter loss weights.

7.1.6 Training pipeline

Before presenting the computational performance achieved by PlanB&B, we first provide a high-level overview of the PlanB&B training pipeline. Our implementation builds on the official codebases of Ye et al. [184] and Wang et al. [172]. Table 5.2 lists the hyperparameters we adjusted from their default settings to match our training setup. Our training pipeline follows the parallelized architecture introduced in EfficientZero, implementing a double-buffering mechanism, as illustrated in Figure 7.4.

The training process operates synchronously across multiple components. MILP actors run B&B episodes using the current model, updated every 100 training steps, to produce policy targets π_t via Gumbel Search and push generated trajectories into a shared replay buffer. CPU rollout workers sample these trajectories from replay buffer, and preprocess them by extracting B&B subtree trajectories from full B&B episodes. GPU batch workers unroll the PlanB&B model for K step using the target model, executing the most compute-intensive steps on the GPU. Finally, the learner worker receives both historic and imagined subtree trajectories and performs gradient updates according to Eq. (7.2).

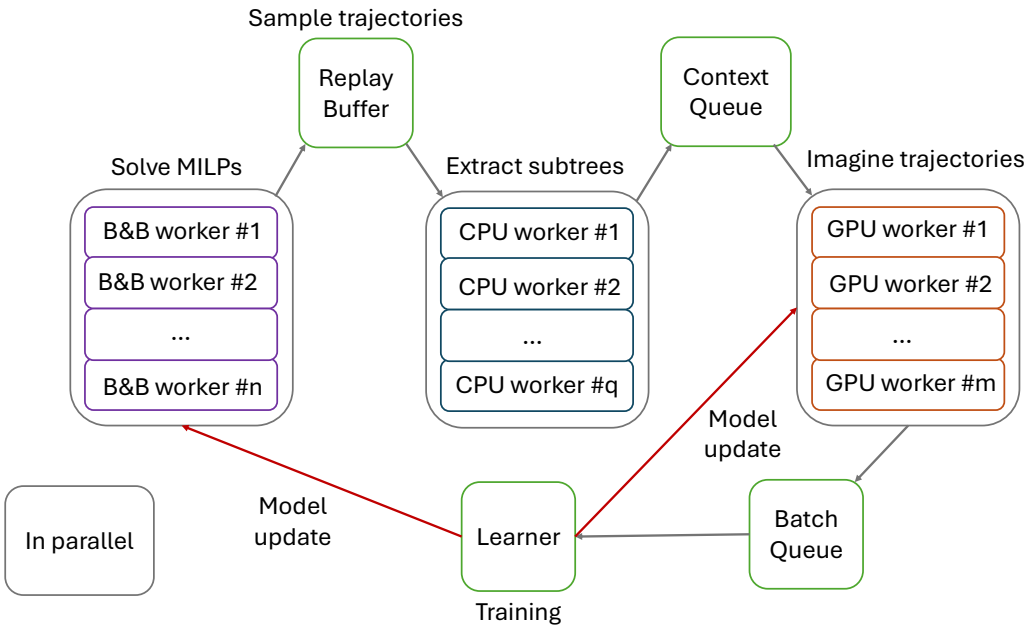


Figure 7.4: PlanB&B training pipeline, build upon the framework from Ye et al. [184].

All components run in parallel. The replay buffer is shared between MILP actors and CPU workers, while a context queue is used for communication between CPU and GPU workers. A separate batch queue connects GPU workers and the learner. This design ensures efficient utilization of both CPU and GPU resources throughout training. Importantly, all workers except for CPU workers leverage GPU acceleration to evaluate the PlanB&B model. All experiments were conducted on a server equipped with an Intel(R) Xeon(R) Platinum 8480CL 128-core processor, 1024 GB of RAM and 4×NVIDIA A100 GPUs (40GB VRAM each). Resources were efficiently allocated across all workers using the Ray library [121], ensuring optimal workload distribution.

Table 7.1: Training parameters for PlanB&B. All unspecified settings were left unchanged and follow the default configuration of Wang et al. [172].

Parameter	Setting
Training steps	10^5
Batch size	128
Optimizer	Adam
Learning rate l_r	$10^{-3} \rightarrow 10^{-5}$
Model unroll step (K)	3
TD steps (n)	3
Discount factor γ	1.0
Policy loss coefficient λ_{p}	1
Value loss coefficient λ_{v}	1
Branchability loss coefficient λ_{b}	1
Tree consistency loss coefficient λ_{ssl}	1
HL-Gauss min log value z_{min}	-1
HL-Gauss max log z_{max}	16
HL-Gauss number of bins m_b	18
HL-Gauss σ_G	0.75
Number of simulations N	50
Gumbel search root node action number M	10
Gumbel search shift factor c_{visit}	50
Gumbel search scaling factor c_{scale}	0.1
Variable node feature dimension d_v	43
Constraint node feature dimension d_c	5
Edge feature dimension d_c	1
Latent space embedding dimension d_h	64
Projection space dimension d_{proj}	16
Replay buffer capacity	10^5
MILP solving time limit (s)	3600

7.2 Experimental study

We now seek to assess the computational performance of our model-based branching agent by answering the following questions:

(Q1) Can PlanB&B learn an efficient policy network to guide MILP solving? In particular, how does it compare against solver heuristics, as well as prior RL and IL approaches?

(Q2) When provided with additional search budget, can our agent further improve the quality of its decisions by leveraging its internal model of B&B?

(Q3) To what extent does the branching behavior of PlanB&B align with that of the expert strong branching (SB) strategy?

(Q4) To what extent do the branchability and temporal consistency losses contribute to PlanB&B's convergence and overall performance?

(Q5) Is the use of a DFS node selection policy inherently detrimental to the performance of branching agents?

7.2.1 Experimental setup

In our experiments, SCIP 8.0.3 [27] is used as backend MILP solver, along with the Ecole library [139] for instance generation, see Appendix B for further benchmark details.

Benchmarks. We consider four standard MILP benchmarks: set covering (SC), combinatorial auctions (CA), maximum independent set (MIS) and multiple knapsack (MK) problems.

Baselines. We compare our PlanB&B agent against prior RL agents, namely DQN-tMDP [56], PG-tMDP [149], DQN-Retro [132], and DQN-BBMDP. We also compare against the IL expert from Gasse et al. [62], trained and evaluated under both SCIP's default node selection policy (IL*), and DFS (IL-DFS). Finally, we report the performance of reliability pseudo cost branching (SCIP), the default branching heuristic used in SCIP, strong branching (SB) [10], and random branching (Random). SCIP configuration is common to all baselines. As in prior works, we set the time limit to one hour, disable restart, and deactivate cut generation beyond root node. All the other parameters are left at their default value.

Training & evaluation. Branching agents are trained on instances of each benchmark separately. For evaluation, we report performance in terms of both node count and solving time on 100 test instances unseen during training, as well as on 100 transfer instances of higher dimensions. Evaluation metrics are averaged over 5 random seeds.

7.2.2 Baselines comparison (Q1)

Unlike in conventional MBRL settings, PlanB&B operates under strict time and computational constraints. Crucially, every second spent on planning directly increases the overall solving time, thereby impacting final performance. Therefore, to allow systematic comparison, the results reported for PlanB&B reflect the performance of its policy network alone, without any computational bud-

Table 7.2: Performance comparison of branching agents on four standard MILP benchmarks. For each method, we report the total number of B&B nodes, presolve time, and total solving time excluding presolve. The presolve phase is identical across all methods. Lower values indicate better performance. **Red** highlights the overall best agent, while **blue** marks the best-performing RL-based agent. Following prior work, results are reported as the geometric mean over 100 unseen test instances and an additional 100 higher-dimensional transfer instances, averaged across 5 random seeds. Norm. Score represents the aggregate average performance of each agent across the four MILP benchmarks, normalized by the score of PlanB&B.

	Set Covering		**Comb. Auction**		**Max. Ind. Set**		**Mult. Knapsack**		**Norm. Score**	
Method	Node	Time	Node	Time	Node	Time	Node	Time	Node	Time
Presolve	–	4.74	–	0.90	–	1.78	–	0.20	–	–
Random	3289	5.94	1111	2.16	386.8	2.01	733.5	0.55	1068	454
SB	35.8	12.93	28.2	6.21	24.9	45.87	161.7	0.69	46	4279
SCIP	62.0	2.27	20.2	1.77	19.5	2.44	289.5	0.53	58	363
IL*	**133.8**	0.90	**83.6**	0.65	**40.1**	0.36	272.0	0.69	**96**	116
IL-DFS	136.4	**0.74**	92.1	0.56	68.5	0.44	411.5	1.07	130	131
PG-tMDP	649.4	2.32	168.0	0.94	153.6	0.92	436.9	1.57	272	254
DQN-tMDP	175.8	0.83	203.3	1.11	168.0	1.00	266.4	0.73	207	188
DQN-Retro	183.0	1.14	103.2	0.78	223.0	1.81	250.3	0.67	208	241
DQN-BBMDP	**152.3**	**0.77**	97.9	0.62	103.2	0.69	236.6	0.66	134	135
PlanB&B	186.2	0.87	**84.7**	**0.54**	**44.8**	**0.32**	**220.0**	**0.55**	**100**	**100**

Test instances

	Set Covering		**Comb. Auction**		**Max. Ind. Set**		**Mult. Knapsack**		**Norm. Score**	
Method	Node	Time	Node	Time	Node	Time	Node	Time	Node	Time
Presolve	–	12.3	–	2.67	–	5.16	–	0.46	–	–
Random	271632	842	317235	749	215879	2102	93452	70.6	8050	3870
SB	672.1	398	389.6	255	169.9	2172	**1709**	**12.5**	13	2243
SCIP	3309	48.4	1376	14.77	3368	90.0	30620	22.1	121	132
IL*	**2610**	23.1	**1282**	9.4	**1993.0**	**38.6**	11730	43.5	**70**	**83**
IL-DFS	3103	**22.5**	1828	10.2	3348	51.9	43705	130.8	151	136
PG-tMDP	44649	221	6001	30.7	3133	43.6	35614	123	373	290
DQN-tMDP	8632	71.3	20553	116	45634	477	22631	65.1	787	679
DQN-Retro	6100	59.4	2908	18.4	119478	1863	27077	79.5	1166	1254
DQN-BBMDP	**5651**	46.4	2273	11.8	7168	81.3	37098	109	189	160
PlanB&B	5869	**46.2**	**1665**	**9.1**	**2853**	**41.1**	**13574**	**51.2**	**100**	**100**

Transfer instances

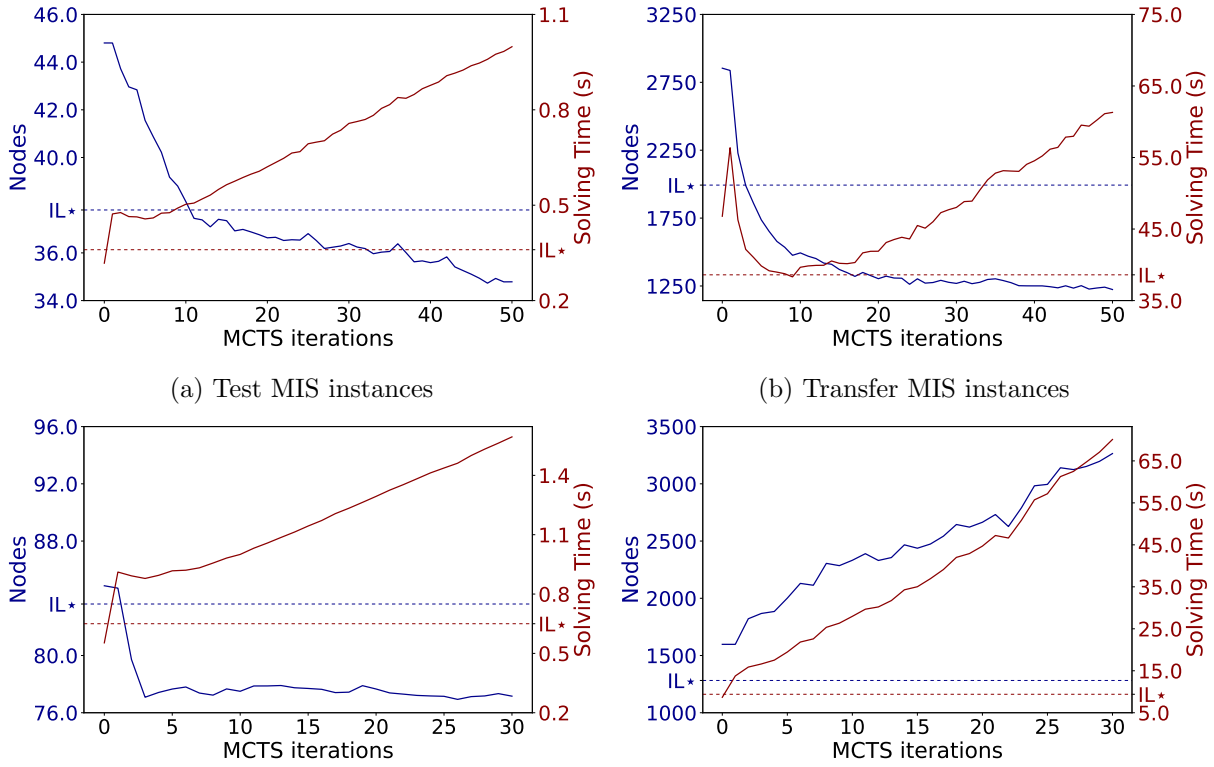
get allocated to MCTS simulations at evaluation time. Computational results obtained on the four benchmarks are presented in Table 7.2. On aggregate test instances, compared to prior RL baselines, PlanB&B's policy network achieves $\approx 1.35\times$ reductions both in tree size and solving time. This performance gap broadens further to $\approx 1.9\times - \approx 1.6\times$ on aggregate transfer instances, underscoring

the superior generalization capabilities of PlanB&B relative to prior RL agents. PlanB&B also outperforms the IL-DFS agent on both test and transfer instances, providing, to our knowledge, the first evidence of an RL-based branching strategy surpassing an IL agent trained to mimic strong branching. Specifically, our experiments demonstrate that, under a DFS node selection policy, PlanB&B learns branching strategies superior to that of the IL agent from Gasse et al. [62]. When compared against the standard IL* baseline, PlanB&B manages to achieve $\approx 10\%$ lower solving time on test instances, despite producing $\approx 5\%$ larger trees in average. However, on transfer instances, PlanB&B is globally unable to overcome the performance limitations imposed by DFS, although it still manages to outperform the IL* baseline on combinatorial auction instances. The influence of DFS on the performance of branching agents is further analyzed in Section 7.2.6. Finally, PlanB&B clearly outperforms the default SCIP baseline on both test and transfer instances, despite operating under DFS, a performance achieved by no other learning-based baseline.

Table 7.8 provides additional performance metrics to compare the different baselines across test and transfer instances. For each benchmark, we report the number of wins and the average rank of each baseline across 100 evaluation instances. These complimentary metrics corroborate the results reported in Table 7.2. Finally, Table 7.9 recapitulates the computational results presented in Table 7.2, and provides for each baseline the per-benchmark standard deviation over five seeds, as well as the fraction of test instances solved to optimality within the time limit.

7.2.3 Planning in B&B (Q2)

Although PlanB&B demonstrates strong performance while relying solely on its policy network, we further investigate its capacity to derive stronger policies by leveraging its learned model to plan at evaluation time. Figures 7.5a-7.5b show the effect of increasing PlanB&B's simulation budget B on both node and time performance over maximum independent set test and transfer instances. On test instances, PlanB&B achieves lower average tree size than the IL* baseline as soon as $B > 12$, reaching up to $\approx 20\%$ tree size reduction for $B = 50$. On transfer instances, not only increased simulation budget enables to reduce average tree size up to $\approx 50\%$, but this reduction also translates into improved solving time performance, with best solving time (achieved at $B = 9$) matching the solving time performance of the IL* agent. Remarkably, when planning over its internal model, PlanB&B produces branching strategy yielding smaller trees than that produced by the IL* agent,



(a) Test MIS instances

(b) Transfer MIS instances

(c) Test CA instances

(d) Transfer CA instances

Figure 7.5: Policy improvement associated with increased simulation budget over both maximum independent set and combinatorial auction benchmarks.

despite operating under DFS.

Unfortunately, this planning gain is not uniform across problem classes. In our experiments, outside of the MIS benchmark, the value head $\bar{\mathbf{v}}$ of the prediction network f does not generalize reliably to higher-dimensional MILPs. As discussed in Section 5.6, we again attribute this to out-of-distribution effects: $\bar{\mathbf{v}}$ is trained on instances inducing structurally smaller subtrees than those encountered at transfer time, so its predictions become increasingly biased as tree sizes grow. Importantly, the degradation of $\bar{\mathbf{v}}$ does not prevent PlanB&B's policy head $\mathbf{p}$ from producing strong branching decisions on transfer instances. However, it does reduce the effectiveness of planning at evaluation time: inaccurate value estimates undermine the backups used by search and can cancel the benefits of increased simulation budgets. This effect is illustrated on instances in Figures 7.5c–7.5d. While model-based planning improves node performance on test combinatorial auction instances, the same procedure yields little to no improvement on transfer instances, where value generalization is weakest. This limitation could

Table 7.3: Alignment metrics between ML baselines and SB on maximum independent set test and transfer instances.

Instances	Policy	Iter.	(↓) SB C-Entropy	(↑) SB Score	(↑) SB Freq.
	SB	–	0.00	1.00	1.00
Test	IL*	–	0.84	0.69	0.45
	PlanB&B	0	1.97	0.63	0.39
	PlanB&B	50	2.08	0.65	0.40
Transfer	IL*	–	0.86	0.76	0.40
	PlanB&B	0	1.46	0.72	0.39
	PlanB&B	50	1.34	0.71	0.38

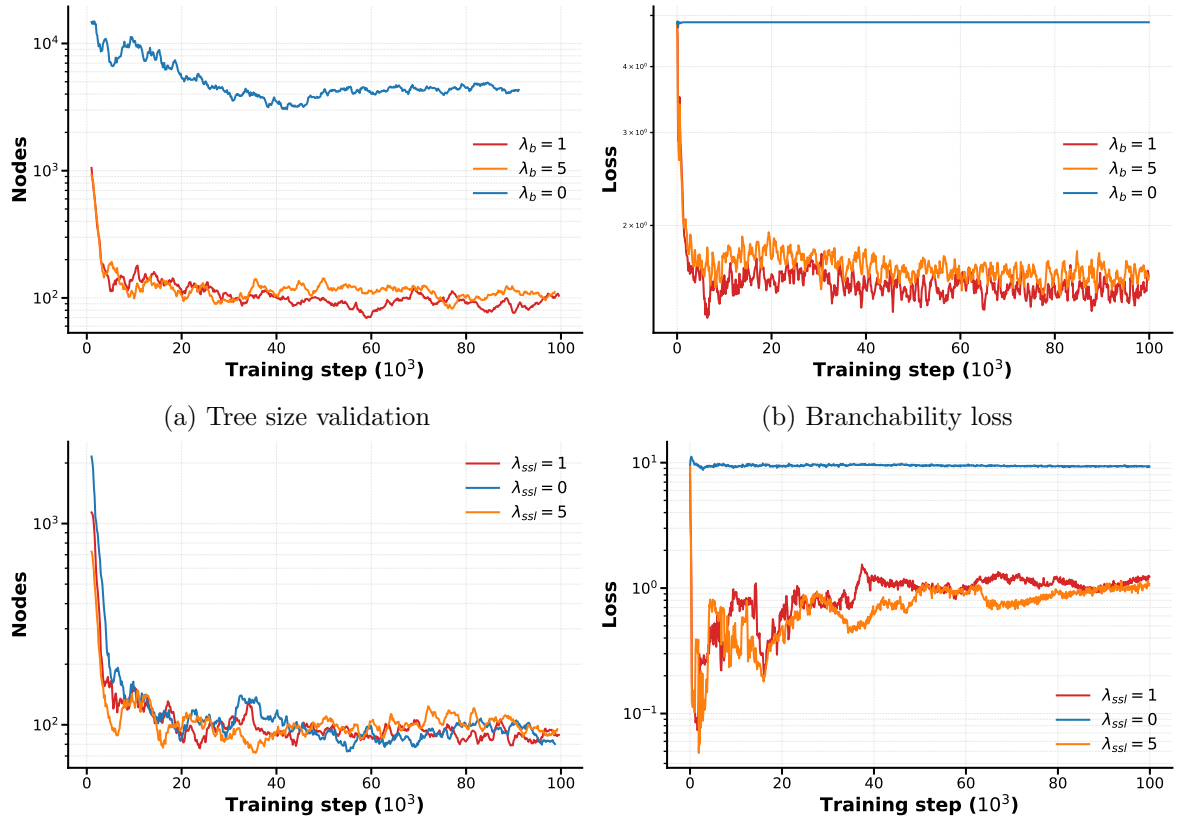
potentially be addressed in future works by gradually increasing the dimensionality of the instances solved during the training of the MILP actors, thereby allowing the value network to progressively adapt to larger B&B trees such as the one encountered in transfer benchmarks.

7.2.4 Is PlanB&B learning to strong branch? (Q3)

Given the performance trends observed in Figures 7.5a-7.5b, a natural question arises: does PlanB&B manage to derive better branching decisions than IL baselines by following SB more closely, or does it discover genuinely novel strategies? After all, strong branching can be interpreted as a one-step, full-width MCTS planning procedure aimed at maximizing immediate dual gap reduction: each candidate branching action is evaluated by solving the two child LP relaxations, and the action with the best aggregated dual-bound score is selected.[3] Given the relatively high alignment observed between SB and DQN-BBMDP in Section 5.5.3, the performance gains of PlanB&B may stem from a closer approximation of strong branching than that achieved by the IL baselines.

To answer this question, Table 7.3 reports several alignment metrics designed to assess which of the IL* and PlanB&B policies more closely resembles strong branching. All metrics are averaged over 100 test instances and 20 higher-dimensional transfer instances. Crucially, these alignments metrics are the same as the ones introduced in Section 5.5.3. Across all metrics, PlanB&B policies exhibit lower alignment with SB than IL* policies. Remarkably, the refined branching policy returned by the MCTS is roughly as close to SB as the policy yielded by the policy network. This suggests that

[3]Interestingly, the computational trade-off associated with LP evaluation in strong branching is reminiscent of model-based planning in MBRL, where additional look-ahead computation is exchanged for better-informed decisions. We further reflect on the connections between planning and mixed-integer linear programming in Chapter 8.



(a) Tree size validation

(b) Branchability loss

(c) Tree size validation

(d) Temporal consistency loss

Figure 7.6: PlanB&B ablation study for branchability and temporal consistency losses on combinatorial auction (CA) instances.

PlanB&B can surpass IL baselines not through closer imitation of strong branching, but by discovering and exploiting original strategies.

7.2.5 Targeted ablations (Q4)

In this section, we investigate how the branchability and temporal consistency losses introduced in PlanB&B affect the agent's convergence behavior and final performance. Ablation results, including training curves and validation metrics on combinatorial auction instances, are reported in Figure 7.6. First, as shown in Figures 7.6a-7.6b, learning to distinguish between branchable and unbranchable nodes is indispensable for the agent to be able to improve its current policy through planning. When setting $\lambda_{\mathrm{b}} = 0$, the agent fails to derive any benefit from search, because it no longer reliably detects episode termination, and therefore cannot assign meaningful values to simulated rollouts. As a re-

Table 7.4: Average transition time (ms) associated with the execution of κ_ρ across the Ecole benchmark. We evalute both SCIP's default node selection policy and depth-first search.

	Set Covering		Comb. Auction		Max. Ind. Set		Mult. Knapsack	
	Test	Transfer	Test	Transfer	Test	Transfer	Test	Transfer
SCIP	10.9	17.1	8.25	14.3	14.7	28.3	5.46	8.28
DFS	8.42	14.8	6.39	12.0	11.9	24.80	5.27	8.24
Gain	−22%	−13%	−23%	−16%	−19%	−12%	−3%	−0.5%

sult, the branchability loss emerges as a key ingredient for PlanB&B to translate search into steady performance gains.

Second, our results suggest that the temporal consistency loss provides, at best, a modest acceleration of training, and has little measurable impact on final performance under our data regime. As shown in Figure 7.6c, any advantage from setting $\lambda_{\text{ssl}} > 0$ is short-lived, largely vanishing after the first 5–10k training steps. This is consistent with the training loss reported in Figure 7.6d: after a rapid initial decrease over the first few thousand steps, the temporal consistency loss gradually rises again before settling at a relatively high plateau. More generally, this behavior matches the intended role of temporal-consistency objectives in MBRL: they primarily act as an auxiliary shaping signal that improves representation learning and stabilizes optimization early on, while long-run performance is ultimately governed by the accuracy of the policy, value, and branchability predictions. In sum, while our experiments suggest that the self-supervised consistency loss is not a decisive ingredient for PlanB&B on the Ecole benchmark, it may prove more valuable in more challenging regimes (e.g., larger instances), where stabilizing training and speeding up early learning become critical.

7.2.6 Influence of DFS (Q5)

We now examine how adopting depth-first search for node selection affects the computational performance of our branching agents. From Table 7.2, two key observations emerge. First, the node and time gaps between DFS and non-DFS variants are highly benchmark-dependent: MK exhibits a large tree-size penalty under DFS, whereas SC exhibits a much smaller one. Second, these gaps systematically increase when moving from test to higher-dimensional transfer instances. In this section, we show that the overall runtime impact of DFS can be largely explained by two coarse but informative quantities: (i) the average per-transition speed-up in LP solves induced by DFS warm-starting, and

Table 7.5: IL* agent associated discovery times on the Ecole benchmark, with t_r defined as the ratio between t_d and T, the overall length of the trajectory. Computational results align with previous observations, with set covering instances displaying low t_r, and multiple knapsack very high t_r.

	Set Covering		Comb. Auction		Max. Ind. Set		Mult. Knapsack	
	Test	Transfer	Test	Transfer	Test	Transfer	Test	Transfer
t_d	8.52	102.1	20.7	257	14.1	264	275	5865
t_r	0.13	0.093	0.51	0.37	0.74	0.29	1.00	1.00

(ii) the average time to discover an optimal incumbent.

As discussed in Section 2.1.4, DFS exerts two competing effects on B&B efficiency. On the one hand, DFS tends to expand larger trees compared to best-bound inspired node selection heuristics, increasing the number of processed nodes and, therefore, the total computational workload. On the other hand, DFS follows deep diving trajectories, which makes successive LP relaxations highly correlated: solvers can then exploit warm starts and basis reuse to reduce LP solving time. As shown in Table 7.4, the associated transition speed-up can be quite substantial in BBMDP. The net performance of DFS therefore results from the interplay between two opposing forces: larger trees that increase the number of explored nodes, and faster LP solves that offset this overhead. The latter helps explain some of the solving time performance gaps observed in Table 7.2 between DFS and non-DFS baselines. In particular, it clarifies how PlanB&B can solve test instances faster than the IL* baseline despite producing, on average, larger trees.

While the per-transition speed-ups reported in Table 7.4 help rationalize some of the gaps observed between DFS and non-DFS baselines, they do not account for the sometimes dramatic tree-size differences typically observed between IL* and IL-DFS. To account for these, it is helpful to consider the notion of *discovery step* t_d, defined as the time at which the B&B algorithm first identifies the global optimal solution x^*. In fact, as soon as $\bar{x} = x^*$, the primal gap is closed, and, consequently, all node selection policies are equivalent for $t \geq t_d$. Therefore, in theory, the smaller the value of t_d, the smaller the node performance gap, and ultimately the smaller the solving time gap. Table 7.5 reports both absolute and relative average discovery times obtained by IL across the Ecole benchmarks. Put together with the DFS speed-ups reported in Table 7.4, these results are consistent with our initial observation. In fact, on test instances, multiple knapsack combines a negligible per-transition DFS gain with the latest possible incumbent discovery time, yielding both a large tree-size penalty and

a runtime disadvantage for the IL-DFS baseline against IL*. At the other end of the spectrum, set covering exhibits early incumbent discovery together with substantial DFS speed-ups, resulting in only a small tree-size disadvantage but a pronounced runtime advantage for IL-DFS against IL*. Interestingly, transfer instances reveal two systematic shifts. First, the relative per-transition speed-up provided by DFS consistently shrinks compared to test instances. Second, across all problem classes, the discovery time t_d increases by at least an order of magnitude. Together, these effects align with our earlier observation: across the Ecole benchmarks, both node and time gaps widen on transfer instances, to the detriment of DFS-based variants.

These results highlight a structural limitation of adopting DFS when solving higher-dimensional MILPs: as problem size grows, closing the primal gap becomes increasingly difficult, which in turn exacerbates the computational burden associated with DFS. Addressing this limitation likely requires moving beyond the traditional MILP bipartite graph representation, and adopting more expressive representations encoding the full B&B tree as recently proposed by Zhang et al. [189].

7.3 Application to real-world industrial instances

A large fraction of recent learning-based contributions in MILP solving have been evaluated primarily on the Ecole benchmark [62, 73, 86, 132, 149, 173]. While this benchmark has played an important role in standardizing experimental protocols, there is growing evidence that performance gains observed on these synthetic families do not always translate to broader, real-world settings [58]. This limitation is particularly salient for learning branching policies in B&B.

We posit that one of the structural reasons for this limitation lies in the nature of the instance families featured in the Ecole benchmark. In fact, these problems typically admit comparatively strong formulations [46], meaning that their LP relaxations provide tight bounds. In this regime, strong branching (SB) is an exceptionally powerful expert: by explicitly evaluating candidate branches through temporary LP solves, it obtains highly informative local bound estimates that lead to aggressive pruning. Consequently, imitation learning approaches that approximate strong branching decisions inherit much of this advantage and naturally emerge as strong baselines.

However, many real-world industrial MILPs differ substantially from these benchmark settings. Industrial formulations are often the result of layered modeling compromises, legacy constraints, and

evolving operational requirements. Their LP relaxations can be weak, producing loose bounds and slower gap closure. In such regimes, the advantage of strong branching can be attenuated: local bound improvements do not necessarily translate into effective global pruning, and the computational overhead of strong branching becomes more difficult to amortize. Imitation-based policies that closely track strong branching therefore likely inherit the limitations of a weaker expert in this regime, and exhibit diminished practical performance.

Such discrepancies highlights an important point: benchmarks built on well-structured instances may implicitly favor expert-driven or imitation-based methods, whereas weaker or noisier industrial formulations can expose their limitations. Evaluating learning-based branching approaches on realistic industrial instances is therefore essential to assess their robustness beyond curated benchmarks. In the following, we investigate how our PlanB&B agent performs on real-world hydroelectric scheduling instances, which present markedly different structural properties from the canonical Ecole families. In particular, these instances are truly mixed-integer, combining both binary and continuous decision variables, rather than simply binary as in the Ecole benchmark.

7.3.1 Hydraulic valley optimization

Hydro valley scheduling problems model the short-term operation of cascaded hydroelectric systems composed of reservoirs and turbine plants connected along a river network. Over a finite horizon, the operator must decide how much water to turbine, spill, and store in each reservoir so as to maximize electricity production revenue under time-varying prices, while satisfying hydraulic balance equations and a range of operational constraints (e.g., reservoir bounds, turbine activation constraints, ramping limits, and spill capacities). These problems are inherently dynamic and strongly coupled across space (through upstream/downstream flows) and time (through storage), and they naturally give rise to mixed-integer linear programs due to discrete turbine commitment decisions.

In addition to their industrial relevance, hydro valley MILPs provide a challenging testbed for learning-based solver components: instances share a common physical structure yet exhibit meaningful variability through inflows, prices, and target volume requirements, making them representative of repeated optimization settings. To the interested reader, Appendix C provides additional background on the industrial context in which these problems arise at EDF. Additionally, it presents the Hydro-MILP formulation chosen to model the problem, along with the instance generation procedure used

to construct problem instances from historical data.

7.3.2 Experimental setup

As described in Appendix C, EDF's target use case is week-ahead production planning for hydroelectric valleys. In the Hydro-MILP formulation, the number of variables and constraints (and thus the problem dimension) scales linearly with the number of time steps in the planning horizon. We therefore train our agents on smaller instances corresponding to a few days of operation, and progressively extend the horizon to obtain increasingly large and complex problems. Using historical hydraulic valley data, we construct six benchmarks of increasing dimension, covering horizons of 2, 3, 4, 5, 6, and 7 days of production. Problem dimensions for each benchmark are specified in Table C.2. In our experiments, all learning agents are first trained on the TwoHydro benchmark (a 2-day planning horizon). After convergence, to validate our approach, we transfer the resulting networks to initialize training on ThreeHydro (3 days) problems. Final performance is then evaluated across all six Hydro-MILP benchmarks. Note that, in principle, given sufficient computational resources, this warm-start procedure could be extended iteratively up to the SevenHydro benchmark.

7.3.3 Computational results

Table 7.6 reports computational results obtained by learning agents trained on both TwoHydro and ThreeHydro benchmarks. Again, the performance reported for PlanB&B corresponds to that achieved by its sole policy network. We also report the performance of SCIP and vanilla strong branching (VSB). VSB differs from SCIP's built-in strong branching rule (typically reported in Tables 5.3 and 7.2), which incorporates numerous subroutines to accelerate the search by exploiting LP solution information. Instead, VSB corresponds to the vanilla strong branching oracle used to generate IL demonstrations. The node count achieved by VSB therefore represents a practical lower bound on the B&B tree size achievable by a perfect strong branching imitator. Three main observations follow.

First, PlanB&B clearly outperforms all previous learning baselines, including IL* and IL-DFS, across the Hydro-MILP benchmarks. Notably, the node and runtime gaps considerably widen as the problem horizon increases, indicating that PlanB&B's advantage becomes more pronounced on larger, more challenging instances. In principle, this would be consistent with our broader hypothesis that, on realistic industrial instances, strong branching provides a less effective demonstration expert than on

Table 7.6: Performance comparison of learning agents on Hydro-MILP instances of increasing dimensions. For each horizon benchmark, results are reported as the geometric mean over 100 instances, averaged across 5 random seeds. Here, DQN denotes DQN-BBMDP.

TwoHydro training

	TwoHydro		**ThreeHydro**		**FourHydro**		**FiveHydro**		**SixHydro**		**SevenHydro**	
Method	Node	Time	Node	Time	Node	Time	Node	Time	Node	Time	Node	Time
Presolve	–	0.44	–	1.11	–	1.96	–	2.75	–	5.35	–	5.39
VSB	34.7	1.39	79.0	3.74	120.7	6.21	194.4	12.39	305.3	26.2	308.4	33.0
SCIP	**8.61**	0.41	**23.4**	**1.07**	**34.3**	**1.63**	**67.3**	**2.97**	**79.0**	**4.34**	**79.8**	**4.43**
IL*	**50.2**	0.46	**188.9**	2.27	463.9	3.34	969	8.9	2528	26.4	2569	26.8
IL-DFS	57.5	0.44	221.9	2.38	560.1	4.18	1141	9.5	2724	28.4	2748	31.3
DQN	79.4	0.43	714	4.31	1082.5	9.56	5027	32.0	14179	119	14852	123
PlanB&B	54.3	**0.26**	191.5	**1.16**	**403.8**	**2.59**	**773.1**	**5.31**	**1554**	**13.0**	**1578**	**13.2**

ThreeHydro training

	TwoHydro		**ThreeHydro**		**FourHydro**		**FiveHydro**		**SixHydro**		**SevenHydro**	
Method	Node	Time	Node	Time	Node	Time	Node	Time	Node	Time	Node	Time
Presolve	–	0.44	–	1.11	–	1.96	–	2.75	–	5.35	–	5.39
VSB	34.7	1.39	79.0	3.74	120.7	6.21	194.4	12.39	305.3	26.2	308.4	33.0
SCIP	**8.61**	0.41	**23.4**	**1.07**	**34.3**	**1.63**	**67.3**	**2.97**	**79.0**	**4.34**	**79.8**	**4.43**
IL*	–	–	225.4	2.44	460.1	3.32	1048	8.23	2701	27.5	2718	27.8
IL-DFS	–	–	250.1	2.68	534.0	4.07	1124	9.4	2949	30.4	2956	32.7
DQN	–	–	530.8	3.61	1537	13.1	5083	31.5	18450	149	18349	147
PlanB&B	–	–	**179.2**	**1.08**	**368.1**	**2.41**	**631**	**4.41**	**1197**	**10.2**	**1264**	**10.9**

the Ecole benchmark. However, we observe that vanilla strong branching, which again can be viewed as an ideal strong branching IL agent, consistently achieves B&B tree size far lower than any other learning baseline, including PlanB&B, across every production horizon. This discrepancy prompts us to explore an alternative explanation for the degraded performance of IL on real-world instance, which we discuss in Chapter 8.

Second, despite improving over prior learning baselines, PlanB&B remains clearly behind SCIP on higher-dimensional transfer instances. While it matches SCIP performance on the training horizon (e.g., TwoHydro when trained on TwoHydro, and ThreeHydro when trained on ThreeHydro), it incurs systematically higher solving times on longer horizon instances. These results underscore the inherent difficulty of matching a solver's highly modular, built-in adaptivity during zero-shot transfer. We propose delving into a deeper explanation for SCIP's superiority over learning baselines on higher-dimensional, industrially relevant instances in Chapter 8.

Third, Table 7.6 reveals that the performance achieved by the PlanB&B agent trained on Three-

Table 7.7: Alignment metrics between learning baselines and strong branching on Hydro-MILP instances.

	Train on ThreeHydro											
	TwoHydro		**ThreeHydro**		**FourHydro**		**FiveHydro**		**SixHydro**		**SevenHydro**	
Method	Score	Freq.	Node	Score	Freq.	Score	Freq.	Score	Freq.	Score	Freq.	Score
SB	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
IL*	**0.62**	**0.48**	**0.58**	**0.45**	**0.58**	**0.45**	**0.56**	**0.43**	**0.55**	**0.42**	**0.54**	**0.42**
DQN	0.49	0.39	0.48	0.38	0.48	0.38	0.46	0.36	0.46	0.36	0.46	0.36
PlanB&B	**0.62**	**0.48**	**0.58**	**0.45**	**0.57**	**0.44**	**0.54**	**0.41**	**0.53**	**0.40**	**0.52**	**0.40**

Hydro consistently improves the performance of the TwoHydro counterpart across all benchmarks, thus widening the gap with other learning baselines and narrowing that with SCIP. This validates our training strategy: PlanB&B can be initialized on lower-dimensional instances and progressively fine-tuned on problems of increasing dimension as computational resources become available, each stage strengthening transfer performance on the target higher-dimensional instances.

Table 7.10 provides additional performance metrics to compare different baselines across Hydro-MILP benchmarks. For each benchmark, we report the number of wins and the average rank of each baseline across 100 evaluation instances. These complimentary metrics broadly align with the results reported in Table 7.11. Notably, on higher-dimensional transfer instances, despite trailing by a substantial gap in average solving time, PlanB&B still achieves approximately 15% wins, indicating that its branching policy remains competitive on a non-negligible fraction of instances even where SCIP dominates on aggregate. Finally, Table 7.11 recapitulates the computational results presented in Table 7.2, and provides for each baseline the per-benchmark standard deviation over five seeds, as well as the fraction of test instances solved to optimality within the time limit.

7.3.4 Behaviour analysis

Following the analysis carried out in Sections 5.5.3 and 7.2.4, we assess the behavior of each learning baseline through its alignment with the strong branching expert. While PlanB&B achieves strong branching alignment comparable to IL on instances matching the training horizon, on higher-dimensional instances, it manages to outperform IL baselines by roughly a factor $\times 2$ in both tree size and solving time, all while displaying slightly lower alignment with strong branching. These results reinforce the findings of Section 7.2.4: unlike traditional value-based method that rely on undirected

exploration, PlanB&B manages to discover original, high-performing variable selection strategies that produce smaller B&B trees while actively diverging from strong branching patterns.

7.4 Conclusion and perspectives

Combinatorial optimization has proven to be a challenging setting for traditional model-free RL approaches. In this chapter, we introduced PlanB&B, a novel model-based reinforcement learning framework leveraging a learned internal model of B&B to discover improved variable selection strategies. Our experimental study leads to four main findings.

First, PlanB&B's dynamics network approximates LP resolution in the latent space with sufficient fidelity to enable policy improvement through model-based planning, thereby extending the applicability of MBRL algorithms originally developed for combinatorial board games to mixed-integer linear programming.

Second, in the context of synthetic repeated MILP benchmarks, our MBRL agent learns branching strategies that clearly outperform both SCIP and former RL baselines. Moreover, further analysis shows instances where PlanB&B surpasses IL, not by more closely replicating expert behavior, but by actively diverging from strong branching patterns, highlighting the potential of RL to uncover branching strategies going beyond existing expert heuristics.

Third, experiments on real-world hydroelectric scheduling instances drawn from EDF operational use cases demonstrate that PlanB&B's advantages extend beyond synthetic benchmarks. On industrial instances, PlanB&B consistently outperforms all prior learning baselines, with performance gaps widening on longer-horizon instances. This behavior indicates stronger generalization to out-of-distribution regimes and reinforces the practical viability of model-based planning beyond curated benchmarks. Additionally, progressive fine-tuning on problems of increasing dimension was found to narrow the gap with SCIP at each stage, offering a practical path toward competitive performance on target industrial instances.

Fourth, however, our computational study also highlights the burden imposed by the DFS node selection policy when seeking to solve higher-dimensional MILPs. In order to fully unlock the potential of MBRL in exact combinatorial optimization, we expect future research to explore the design of scalable observation functions capable of efficiently encoding evolving B&B trees.

Table 7.8: Additional performance metrics for each baseline on train / test and transfer instance benchmarks. For each benchmark, we report the number of wins, and the average rank of each baseline across the 100 evaluation instances. We also report for each baseline the fraction of test instances solved to optimality within time limit.

Set Covering

	Train / Test			Transfer		
Method	Solved	Wins	Rank	Solved	Wins	Rank
SCIP	100/100	3/100	6.61	100/100	2/100	4.75
IL*	100/100	0/100	4.38	100/100	29/100	1.85
IL-DFS	100/100	18/100	**2.65**	100/100	**58/100**	**1.71**
PG-tMDP	100/100	0/100	6.6	78/100	0/100	7.95
DQN-tMDP	100/100	4/100	3.54	96/100	1/100	5.64
DQN-Retro	100/100	0/100	5.65	99/100	0/100	5.85
DQN-BBMDP	100/100	24/100	2.49	100/100	0/100	4.83
PlanB&B	100/100	**51/100**	**2.76**	100/100	**10/100**	**3.42**

Combinatorial Auction

	Train / Test			Transfer		
Method	Solved	Wins	Rank	Solved	Wins	Rank
SCIP	100/100	4/100	7.08	100/100	5/100	4.47
IL*	100/100	10/100	3.9	100/100	27/100	**2.34**
IL-DFS	100/100	33/100	3.17	100/100	26/100	2.68
PG-tMDP	100/100	0/100	5.66	100/100	0/100	6.82
DQN-tMDP	100/100	0/100	5.08	100/100	0/100	7.97
DQN-Retro	100/100	1/100	3.47	100/100	0/100	5.52
DQN-BBMDP	100/100	14/100	3.01	100/100	3/100	3.9
PlanB&B	100/100	**38/100**	**2.74**	100/100	**39/100**	**2.30**

Maximum Independent Set

	Train / Test			Transfer		
Method	Solved	Wins	Rank	Solved	Wins	Rank
SCIP	100/100	2/100	6.59	100/100	5/100	4.98
IL*	100/100	37/100	**2.42**	100/100	**33/100**	**2.13**
IL-DFS	100/100	14/100	3.22	100/100	7/100	3.36
PG-tMDP	100/100	0/100	5.43	100/100	23/100	3.39
DQN-tMDP	100/100	0/100	5.47	85/100	0/100	7.05
DQN-Retro	100/100	2/100	6.31	22/100	0/100	7.82
DQN-BBMDP	100/100	2/100	3.93	98/100	0/100	4.99
PlanB&B	100/100	**43/100**	**2.63**	100/100	**32/100**	**2.28**

Multiple Knapsack

	TwoHydro			ThreeHydro		
Method	Solved	Wins	Rank	Solved	Wins	Rank
SCIP	**100/100**	**72/100**	**1.69**	**100/100**	**53/100**	**2.24**
IL*	100/100	0/100	5.35	100/100	7/100	3.62
IL-DFS	100/100	1/100	6.74	100/100	0/100	6.86
PG-tMDP	100/100	0/100	6.92	98/100	4/100	5.85
DQN-tMDP	100/100	1/100	4.26	99/100	10/100	4.1
DQN-Retro	100/100	2/100	4.12	100/100	8/100	4.43
DQN-BBMDP	100/100	5/100	3.99	100/100	4/100	5.09
PlanB&B	**100/100**	**19/100**	**2.92**	**100/100**	**14/100**	**3.81**

Table 7.9: Computational performance comparison on four MILP benchmarks. Following prior works, we report geometrical mean over 100 instances, averaged over 5 seeds, as well as the per-benchmark standard deviations.

	Train / Test			Transfer		
Method	Nodes	Time	Solved	Nodes	Time	Solved
Random	3289 ± 4.2%	5.94 ± 4.3%	100/100	271632 ± 12.7%	842 ± 9.8%	60/100
SB	35.8 ± 0.0%	12.93 ± 0.0%	100/100	672.1 ± 0.0%	398 ± 0.2%	82/100
SCIP	62.0 ± 0.0%	2.27 ± 0.0%	100/100	3309 ± 0.0%	48.4 ± 0.1%	100/100
IL⋆	**133.8 ± 1.0%**	0.90 ± 4.8%	100/100	**2610 ± 0.7%**	23.1 ± 1.5%	**100/100**
IL-DFS	136.4 ± 1.8%	**0.74 ± 5.3%**	100/100	3103 ± 2.0%	**22.5 ± 3.1%**	**100/100**
PG-tMDP	649.4 ± 0.7%	2.32 ± 2.4%	100/100	44649 ± 3.7%	221 ± 4.1%	78/100
DQN-tMDP	175.8 ± 1.1%	0.83 ± 4.5%	100/100	8632 ± 4.9%	71.3 ± 5.8%	96/100
DQN-Retro	183.0 ± 1.2%	1.14 ± 4.1%	100/100	6100 ± 4.2%	59.4 ± 4.2%	98/100
DQN-BBMDP	**152.3 ± 0.6%**	**0.77 ± 5.6%**	100/100	**5651 ± 2.2%**	46.4 ± 3.3%	**100/100**
PlanB&B	186.2 ± 0.4%	0.87 ± 6.0%	100/100	5869 ± 2.5%	**46.2 ± 3.1%**	**100/100**

Set covering

	Train / Test			Transfer		
Method	Nodes	Time	Solved	Nodes	Time	Solved
Random	1111 ± 4.3%	2.16 ± 6.6%	100/100	3172355 ± 7.5%	749 ± 9.1%	64/100
SB	28.2 ± 0.0%	6.21 ± 0.1%	100/100	389.6 ± 0.0%	255 ± 0.2%	88/100
SCIP	20.2 ± 0.0%	1.77 ± 0.1%	100/100	1376 ± 0.0%	14.77 ± 0.1%	100/100
IL⋆	**83.6 ± 0.8%**	0.65 ± 8.5%	100/100	**1282 ± 1.6%**	9.4 ± 2.0%	**100/100**
IL-DFS	95.5 ± 0.9%	0.56 ± 7.2%	100/100	1828 ± 2.0%	10.2 ± 1.6%	100/100
PG-tMDP	168.0 ± 2.8%	0.94 ± 6.0%	100/100	6001 ± 2.7%	30.7 ± 2.4%	100/100
DQN-tMDP	203.3 ± 4.2%	1.11 ± 4.0%	100/100	20553 ± 3.8%	116 ± 3.9%	100/100
DQN-Retro	103.2 ± 1.2%	0.78 ± 7.5%	100/100	2908 ± 1.7%	18.4 ± 2.7%	100/100
DQN-BBMDP	97.9 ± 1.2%	0.62 ± 8.5%	100/100	2273 ± 1.9%	11.8 ± 2.0%	100/100
PlanB&B	**84.7 ± 1.4%**	**0.54 ± 7.9%**	100/100	**1665 ± 2.3%**	**9.1 ± 2.4%**	**100/100**

Combinatorial auction

	Train / Test			Transfer		
Method	Nodes	Time	Solved	Nodes	Time	Solved
Random	386.8 ± 5.4%	2.01 ± 4.8%	100/100	215879 ± 6.7%	2102 ± 6.2%	25/100
SB	24.9 ± 0.0%	45.87 ± 0.4%	100/100	169.9 ± 0.2%	2172 ± 0.9%	15/100
SCIP	19.5 ± 0.0%	2.44 ± 0.4%	100/100	3368 ± 0.0%	90.0 ± 0.2%	100/100
IL⋆	**40.1 ± 3.45%**	0.36 ± 3.4%	100/100	**1882 ± 4.0%**	**38.6 ± 3.3%**	**100/100**
IL-DFS	68.5 ± 6.5%	0.44 ± 4.1%	100/100	3501 ± 2.7%	51.9 ± 2.1%	100/100
PG-tMDP	153.6 ± 5.0%	0.92 ± 2.6%	100/100	3133 ± 4.6%	43.6 ± 2.9%	100/100
DQN-tMDP	168.0 ± 5.6%	1.00 ± 3.4%	100/100	45634 ± 7.4%	477 ± 5.1%	85/100
DQN-Retro	223.0 ± 4.1%	1.81 ± 3.6%	100/100	119478 ± 6.1%	1863 ± 4.8%	22/100
DQN-BBMDP	103.2 ± 9.3%	0.62 ± 6.8%	100/100	7168 ± 5.3%	81.3 ± 4.2%	95/100
PlanB&B	**44.8 ± 7.6%**	**0.32 ± 6.4%**	100/100	**2853 ± 4.9%**	**41.1 ± 5.4%**	**100/100**

Maximum independent set

	Train / Test			Transfer		
Method	Nodes	Time	Solved	Nodes	Time	Solved
Random	733.5 ± 13.0%	0.55 ± 6.9%	100/100	93452 ± 14.3%	70.6 ± 9.2%	99/100
SB	161.7 ± 0.0%	0.69 ± 0.1%	100/100	**1709 ± 0.5%**	**12.5 ± 0.9%**	**100/100**
SCIP	289.5 ± 0.0%	0.53 ± 0.2%	100/100	30260 ± 0.0%	22.1 ± 0.2%	100/100
IL⋆	272.0 ± 12.9%	0.69 ± 8.1%	100/100	11730 ± 7.1%	43.5 ± 6.4%	100/100
IL-DFS	411.5 ± 13.0%	1.07 ± 8.8%	100/100	43705 ± 9.2%	130.8 ± 8.3%	98/100
PG-tMDP	436.9 ± 21.2%	1.57 ± 16.9%	100/100	35614 ± 14.3%	123 ± 15.4%	98/100
DQN-tMDP	266.4 ± 7.2%	0.73 ± 4.6%	100/100	22631 ± 8.6%	65.1 ± 5.5%	99/100
DQN-Retro	250.3 ± 9.5%	0.67 ± 5.0%	100/100	27077 ± 8.8%	79.5 ± 6.2%	100/100
DQN-BBMDP	236.6 ± 6.4%	0.66 ± 2.7%	100/100	37098 ± 7.0%	109 ± 4.9%	100/100
PlanB&B	**220.0 ± 5.9%**	**0.55 ± 4.9%**	100/100	**13574 ± 6.6%**	**51.2 ± 4.8%**	**100/100**

Multiple knapsack

Table 7.10: Additional performance metrics for each baseline on both test and higher-dimensional transfer Hydro-MILP instances. For each benchmark, we report the number of wins, and the average rank of each baseline across the 100 evaluation instances. We also report for each baseline the fraction of test instances solved to optimality within time limit.

TwoHydro training

	TwoHydro			ThreeHydro			FourHydro		
Method	Solved	Wins	Rank	Solved	Wins	Rank	Solved	Wins	Rank
SCIP	100/100	36/100	2.95	100/100	**48/100**	**1.93**	99/100	**59/100**	**1.66**
IL*	100/100	0/100	3.87	99/100	1/100	3.44	95/100	2/100	3.50
IL-DFS	100/100	7/100	3.12	98/100	11/100	2.96	94/100	6/100	3.18
DQN-BBMDP	100/100	12/100	2.88	90/100	2/100	4.64	90/100	3/100	4.54
PlanB&B	100/100	**45/100**	**2.18**	100/100	**38/100**	**2.03**	99/100	**30/100**	**2.12**

	FiveHydro			SixHydro			SevenHydro		
Method	Solved	Wins	Rank	Solved	Wins	Rank	Solved	Wins	Rank
SCIP	100/100	**71/100**	**1.49**	100/100	**82/100**	**1.29**	100/100	**83/100**	**1.25**
IL*	92/100	3/100	3.28	88/100	1/100	3.25	88/100	0/100	3.28
IL-DFS	92/100	8/100	3.27	88/100	1/100	3.71	88/100	3/100	3.55
DQN-BBMDP	90/100	0/100	4.71	70/100	0/100	4.71	75/100	0/100	4.72
PlanB&B	99/100	**18/100**	**2.25**	95/100	**16/100**	**2.04**	97/100	**14/100**	**2.20**

ThreeHydro training

	TwoHydro			ThreeHydro			FourHydro		
Method	Solved	Wins	Rank	Solved	Wins	Rank	Solved	Wins	Rank
SCIP	–	–	–	100/100	**48/100**	**1.96**	99/100	**52/100**	**1.86**
IL*	–	–	–	98/100	10/100	3.02	97/100	18/100	2.35
IL-DFS	–	–	–	98/100	2/100	3.74	96/100	0/100	3.85
DQN-BBMDP	–	–	–	92/100	1/100	4.2	90/100	2/100	4.68
PlanB&B	–	–	–	100/100	**39/100**	2.08	98/100	28/100	2.24

	FiveHydro			SixHydro			SevenHydro		
Method	Solved	Wins	Rank	Solved	Wins	Rank	Solved	Wins	Rank
SCIP	100/100	**65/100**	**1.48**	100/100	**80/100**	**1.33**	100/100	**80/100**	**1.31**
IL*	96/100	15/100	2.59	91/100	1/100	2.84	91/100	6/100	2.72
IL-DFS	93/100	0/100	3.88	88/100	0/100	3.94	87/100	0/100	3.85
DQN-BBMDP	85/100	0/100	4.78	70/100	0/100	4.85	65/100	0/100	4.89
PlanB&B	99/100	**30/100**	**2.27**	98/100	**19/100**	**2.04**	99/100	**14/100**	**2.23**

Table 7.11: Computational performance comparison on the Hydro-MILP benchmarks. Following prior works, we report geometrical mean over 100 instances, averaged over 5 seeds, as well as the per-benchmark standard deviations.

TwoHydro training

	TwoHydro		ThreeHydro		FourHydro	
Method	Nodes	Time	Nodes	Time	Nodes	Time
SCIP	**8.61 ± 0.0%**	0.41 ± 0.0%	**23.4 ± 0.0%**	**1.07 ± 0.0%**	**34.3 ± 0.0%**	**1.63 ± 0.0%**
IL*	**50.2 ± 4.3%**	0.46 ± 8.2%	**188.9 ± 12.7%**	2.27 ± 16.4%	463.9 ± 4.22%	3.34 ± 3.53%
IL-DFS	57.5 ± 4.5%	0.44 ± 7.9%	221.9 ± 13.1%	2.38 ± 16.2%	560.1 ± 4.4%	4.18 ± 3.9%
DQN-BBMDP	79.4 ± 5.68%	0.43 ± 5.49%	714 ± 2.77%	4.31 ± 3.74%	1082.5 ± 2.64%	9.56 ± 4.03%
PlanB&B	54.3 ± 7.9%	**0.26 ± 11.2%**	191.5 ± 6.45%	**1.16 ± 3.81%**	**403.8 ± 6.74%**	**2.59 ± 8.62%**

	FiveHydro		SixHydro		SevenHydro	
Method	Nodes	Time	Nodes	Time	Nodes	Time
SCIP	**67.3 ± 0.0%**	**2.97 ± 0.0%**	**79.0 ± 0.0%**	**4.34 ± 0.0%**	**79.8 ± 0.0%**	**4.43 ± 0.0%**
IL*	969 ± 3.4%	8.91 ± 3.5%	2528 ± 5.71%	26.4 ± 5.57%	2569 ± 5.44%	26.8 ± 5.52%
IL-DFS	1141 ± 3.2%	9.48 ± 3.4%	2724 ± 5.82%	28.4 ± 5.77%	2748 ± 5.91%	31.3 ± 5.84%
DQN-BBMDP	5027 ± 3.86%	32.0 ± 4.36%	14719 ± 7.32%	119 ± 8.02%	14852 ± 8.47%	123 ± 7.51%
PlanB&B	**773.1 ± 7.21%**	**5.31 ± 9.13%**	**1554 ± 3.37%**	**13.0 ± 5.72%**	**1578 ± 4.78%**	**13.2 ± 5.98%**

ThreeHydro training

	TwoHydro		ThreeHydro		FourHydro	
Method	Nodes	Time	Nodes	Time	Nodes	Time
SCIP	**8.61 ± 0.0%**	0.41 ± 0.0%	**23.4 ± 0.0%**	**1.07 ± 0.0%**	**34.3 ± 0.0%**	**1.63 ± 0.0%**
IL*	–	–	225.4 ± 10.8%	2.44 ± 11.9%	460.1 ± 4.0%	3.32 ± 4.1%
IL-DFS	–	–	250.1 ± 10.3%	2.68 ± 13.5%	534.0 ± 4.52%	4.07 ± 4.03%
DQN-BBMDP	–	–	530.8 ± 5.4%	3.61 ± 4.9%	1537 ± 2.43%	13.1 ± 3.99%
PlanB&B	–	–	**179.2 ± 6.83%**	**1.08 ± 4.37%**	**368.1 ± 7.0%**	**2.41 ± 8.44%**

	FiveHydro		SixHydro		SevenHydro	
Method	Nodes	Time	Nodes	Time	Nodes	Time
SCIP	**67.3 ± 0.0%**	**2.97 ± 0.0%**	**79.0 ± 0.0%**	**4.34 ± 0.0%**	**79.8 ± 0.0%**	**4.43 ± 0.0%**
IL*	1048 ± 3.18%	8.23 ± 3.14%	2701 ± 5.6%	27.5 ± 5.8%	2718 ± 5.7%	27.8 ± 5.7%
IL-DFS	1124 ± 3.21%	9.44 ± 3.27%	2949 ± 5.8%	30.4 ± 6.0%	2956 ± 5.9%	32.7 ± 5.8%
DQN-BBMDP	5083 ± 4.0%	31.5 ± 4.2%	18450 ± 5.18%	149 ± 6.58%	18349 ± 7.64%	147 ± 8.02%
PlanB&B	**631 ± 6.29%**	**4.41 ± 5.72%**	**1197 ± 4.38%**	**10.2 ± 5.34%**	**1264 ± 5.26%**	**10.9 ± 6.57%**

Chapter 8

Interim conclusion

Parts I, II and III addressed the problem of learning improved variable selection strategies to accelerate the solution of repeated mixed-integer linear programs. Before turning to a complementary research direction in Part IV, we first take a step back to examine the broader picture emerging from the contributions and findings presented in these parts, and outline promising directions for future work on learning to branch. To the best of our knowledge, the research directions proposed below are yet to be discussed, let alone explored, in the literature.

8.1 On preserving the strong branching signal

So far, this thesis has sought to learn stronger variable selection policies by leveraging reinforcement learning to train agents that directly target tree size minimization, instead of cloning the behaviour of an expert known to be suboptimal in the general case [46]. To that end, we first proposed reverting to a principled Markov decision process formulation that accurately models variable selection in B&B, and introduced a histogram classification loss tailored to the training of value-based RL branching agents. Building on these foundations, we then adapted model-based reinforcement learning techniques originally developed for combinatorial board games to the problem of learning to branch, leveraging look-ahead search to mitigate insufficient exploration issues and provide stronger policy improvement guarantees. Interestingly, despite these improvements, imitation learning remains a rather competitive baseline on synthetic benchmarks with provably good formulations, while requiring considerably less implementation complexity and training infrastructure than its reinforcement learning and planning-based counterparts. Moreover, while imitation learning proves less effective on EDF hydro instances,

strong branching itself remains a powerful oracle on both synthetic and industrial problems: the vanilla strong branching baseline achieves substantially smaller tree sizes than all other baselines, as reported in Tables 5.3 and 7.6.

Together, these observations suggest that the supervisory signal provided by strong branching demonstrations retains considerable value, even against more sophisticated reinforcement learning and planning-based approaches. Explicitly preserving this signal within the PlanB&B framework therefore appears as a promising direction for future extensions. A natural approach would consist in augmenting the current model with an auxiliary prediction head tasked with estimating the dual bound improvement at each branching step. The principle behind this design is well established in deep reinforcement learning: Jaderberg et al. [88] demonstrated that auxiliary tasks trained jointly with the primary policy through a shared representation can yield substantial gains in both sample efficiency and final performance, by encouraging the encoder to capture features that are relevant beyond the immediate reward signal. More recently, Guo et al. [72] showed that a similar mechanism, whereby a self-supervised latent prediction objective shares its encoder with the policy network, improves the quality of learned representations in hard-exploration settings. Analogously, training a dual bound prediction head alongside the policy, value and branchability heads of PlanB&B would inject a dense, expert-derived learning signal into the shared graph neural network encoder, encouraging it to learn features predictive of LP bound tightening, and potentially improving both sample efficiency and final policy quality while retaining the flexibility of the reinforcement learning formulation.

8.2 Potential architectural bottlenecks in learning to branch

However promising, the success of such an approach hinges on the ability of the GNN encoder to faithfully represent strong branching information in the first place. Yet, the alignment analyses reported in Tables 5.8 and 7.7 suggest that message-passing GNNs face a fundamental expressivity barrier in this regard. As discussed in Section 2.4.2, message-passing GNNs on MILP bipartite graphs are bounded by the separation power of the Weisfeiler–Lehman test, which is provably insufficient to distinguish certain MILP instances that differ in their associated strong branching scores [35]. Chen et al. [36] sharpen this result by defining a class of MILPs for which message-passing GNNs can approximate strong branching with arbitrary precision, and proving that outside this class, no message-passing architecture can do so, regardless of its parameterization. They further show that

second-order folklore GNNs (2-FGNNs), whose expressivity exceeds the Weisfeiler–Lehman barrier, recover universal approximation over the entire MILP space. It is plausible that PlanB&B encounters this expressivity ceiling during training: once the encoder can no longer reliably reproduce strong branching decisions, the agent departs from expert behaviour and optimizes for what it can represent. This interpretation would account for a striking pattern observed on maximum independent set and EDF hydro instances: both PlanB&B and IL baselines exhibit comparable alignment with strong branching, yet PlanB&B achieves roughly twice the performance in practice, suggesting that its advantage stems not from better imitation, but from a more effective response to the encoder's representational constraints. A natural way to test this hypothesis would be to replace the GNN backbone with a more expressive architecture, such as the 2-FGNNs proposed by Chen et al. [36], or to augment the existing message-passing architecture with random node features to break symmetry [35], though the practical implications of the latter for training stability in RL remain to be investigated. If such a change were to improve strong branching alignment for both IL and PlanB&B baselines, it would confirm that the current performance ceiling is at least partly architectural rather than algorithmic. We stress, however, that this experiment would serve more as a diagnostic than a practical solution: while standard message-passing GNNs scale linearly in the number of edges of the bipartite graph, 2-FGNNs operate on node pairs and incur cubic cost in the number of graph nodes, making them prohibitively expensive for online use within B&B, where the network is queried at every branching step.

8.3 From variable selection to joint branching and bounding strategies

Beyond expressivity issues, we observe that a significant gap persists between learned and solver branching heuristics on real-world problems. While PlanB&B clearly outperforms all prior learning baselines on EDF hydro instances, its solving time only matches that of the solver's default heuristic on test instances, and falls significantly behind it on higher-dimensional out-of-distribution instances. These results may reflect a discrepancy between variable selection as modeled in BBMDP and as implemented in practice by modern solvers. In BBMDP, the agent learns only to select a branching variable at each node. By contrast, solver branching modules do considerably more: in reliability branching, the solver performs strong branching evaluations on under-explored variables, solving child LP relaxations that yield dual bound improvements, updated pseudocosts, and domain propagation

information, all of which contribute to pruning and tightening throughout the subsequent search. This endows solver branching heuristics with a form of long-term planning: by investing heavily in LP evaluations early in the search to build reliable pseudocosts, they trade short-term computational cost for stronger pruning downstream, an ability that the BBMDP formulation, in its current form, does not grant to learned agents. In other words, while IL and RL agents currently optimize branching alone, solver heuristic modules effectively optimize branching **and** bounding jointly.

We believe that, for learned policies to reliably surpass solver heuristics in the future, they will need to play the same game, and be given the same options as their CO expert counterparts. Concretely, this would require extending the BBMDP action space beyond pure variable selection to encompass the auxiliary LP evaluations that solvers perform during branching. Under a first model, the agent would not choose which variable to branch on directly, but instead decide which candidates to probe via LP solves, thus mirroring the selective probing logic of reliability branching with a learned policy in place of hard-coded thresholds. The branching decision would then follow from the resulting pseudocosts, and the bound information generated by these probes would flow into the solver's general machinery, just as under standard reliability branching. A second, more ambitious model would additionally give the agent control over the final branching decision itself, rather than deferring it to pseudocost rankings. In this setting, the agent would jointly optimize both the probing strategy and the branching decision, effectively subsuming the full logic of reliability branching within a single learned policy.

Importantly, adopting either of these models entails further modifications to the BBMDP formulation. In particular, the reward signal needs to shift from tree size minimization to a metric that faithfully captures the true computational cost of solving MILP instances, such as the total number of LP solves or LP iterations, since an agent that controls probing decisions can no longer be evaluated solely on the number of B&B nodes it produces. Furthermore, and perhaps more fundamentally, the subtree decomposition underlying our DFS-BBMDP agents needs to be revisited. Recall that, under $\rho = \text{DFS}$, the cost of processing a subtree $T(o_i)$ rooted at node o_i only depends on the local MILP and associated incumbent $\bar{x}_{o_i}$. This property is what enables defining subtree value functions on bipartite graph observations alone, and learning optimal policies by minimizing each subtree independently. Once probing decisions enter the picture, however, this subtree independence breaks down: LP evaluations performed in one subtree generate pseudocost information that persists across the entire search, influencing branching quality (and therefore subtree size) at nodes explored later on. The observable

state would thus need to be augmented with per-variable pseudocost statistics and probing counts, making it dependent on the history of probing decisions taken throughout the tree.

Therefore, under such extended BBMDP formulations, the DFS node selection policy would no longer confer any subtree decomposition advantage. This is not entirely a loss, however, since Section 7.2.6 already identified DFS as a growing liability when scaling to higher-dimensional instances. Moreover, even without the subtree independence guarantee, the inductive bias associated with B&B subtree decomposition need not be abandoned entirely. Zhang et al. [189] recently proposed a tripartite graph representation of the B&B search tree that extends the standard variable-constraint bipartite graph with a third class of vertices representing the open (leaf) nodes of the current search tree. Each leaf node vertex carries features describing its local branching constraints, LP bound, depth, and estimate value, and is connected to the variable vertices through edges encoding the branching constraints accumulated along its path from the root. This representation is shown to be theoretically sufficient for capturing the state of the search tree. In our setting, it would provide a natural basis for recovering subtree value decomposition: rather than relying on a single global value prediction, the GNN could produce a per-leaf-node value estimate, each trained to predict the cost of resolving the corresponding subtree, and the global value function would be recovered by summing over all leaf node outputs. This would preserve the structural decomposition of BBMDP while operating on an observation that encodes the full search state, including the pseudocost and probing history that the extended formulations require. In the following, we collectively refer to these extended formulations as ProbBBMDP, for Probing Branch-and-Bound Markov Decision Process.

While theoretically consistent, ProbBBMDP formulations raise serious algorithmic and practical implementation challenges. First, the action space grows exponentially: rather than selecting a single variable for branching, the agent must now choose a subset of variables to probe (on top, under the second model, of a variable to branch on) resulting in an action space whose cardinality scales as $\mathcal{O}(2^{|\mathcal{I}|})$ rather than $\mathcal{O}(|\mathcal{I}|)$. Second, implementing these extended formulations require substantially tighter integration with the solver's internal machinery. Modern MILP solvers such as SCIP are large, highly optimized C codebases in which branching, propagation, separation, conflict analysis, and pseudocost book-keeping are deeply intertwined. Wielding the level of fine-grained control needed to selectively trigger LP evaluations, update pseudocosts, and retrieve propagation outcomes at each branching step goes well beyond what current Python interfaces like Ecole or even PySCIPOpt can offer, both of which

being primarily designed for high-level callback access and observation extraction rather than low-level algorithmic control over the solver's internal decision pipeline. Moreover, frequent round-trips between a Python-based learning agent and the solver's C core could introduce non-negligible communication overhead, potentially offsetting the computational gains that learned probing strategies aim to achieve.

Despite these challenges, we believe that this direction warrants further exploration: as long as learned policies operate on a strictly narrower action space than solver heuristics, they remain structurally disadvantaged regardless of the quality of their branching decisions. Additionally, beyond its conceptual appeal, ProbBBMDP offers a practical advantage for training stability. Under the first model, the agent controls only which variables to probe, while the branching decision itself is deferred to pseudocost rankings. As a result, even an untrained agent produces branching behaviour that falls between uninitialized pseudocost branching and full strong branching, rather than the catastrophic random branching that an uninitialized BBMDP policy would induce. This substantially bounds the worst-case B&B tree size early in training, yielding structurally shorter episodes and reducing the sample complexity of the initial learning phase. In turn, this property could enable training directly on higher-dimensional, industrially relevant instances, a regime where current BBMDP agents must first be trained on smaller problems before progressive fine-tuning, precisely because random branching on large instances produces intractably long episodes.

Interestingly, ProbBBMDP also admits a natural multi-agent decomposition. The probing and branching decisions, while coupled through their shared effect on the search tree, optimize for partially distinct objectives: the probing agent seeks to allocate LP evaluations where they yield the most informative bound change, while the branching agent seeks to select variables that minimize the remaining search effort given the information currently available. Decomposing the problem into two cooperating agents would allow each to operate on a simpler action space while sharing a common state representation and a shared reward signal reflecting overall solving cost. This decomposition also suggests a natural staged training curriculum. In a first phase, only the probing agent is trained with branching decisions deferred to pseudocost rankings, thus directly benefiting from the training stability guarantees described above. Once the probing policy has converged, a second phase introduces the branching agent, which now operates on a pseudocost landscape already shaped by a learned probing strategy rather than by uninitialized or hard-coded heuristics. A final joint training phase would then allow both agents to co-adapt: the probing agent can learn to allocate LP evaluations that specifically

complement the branching agent's emerging strategy, while the branching agent adjusts to the evolving information landscape. This staged approach progressively increases the complexity of the learning problem while maintaining stable training conditions at each phase, and could prove critical for scaling to industrially relevant instance sizes.

Finally, we stress that BBMDP, rather than TreeMDP, is the only viable foundation for ProbBB-MDP extensions. As established in Chapter 4, TreeMDP relaxes core MDP properties, which prevents it from accommodating multi-step temporal difference learning and planning-based policy improvement, and renders it ill-defined outside the DFS regime where the tree Markov property 4.3.1 does not hold. Since the extended formulations proposed above explicitly break subtree independence **under any node selection policy**, TreeMDP cannot serve as their basis. BBMDP, by contrast, preserves full MDP structure irrespective of the node selection policy, and therefore provides the necessary starting point for any formulation that seeks to integrate branching, probing, and bounding within a single learning agent.

8.4 From learned to exact simulation

A further benefit of deeper solver integration concerns the planning procedure itself. As described in Chapter 7, PlanB&B currently operates as a hybrid between AlphaZero and MuZero: while node selection and rewards are simulated exactly, the branching operation is approximated by a learned dynamics network g. This design choice was motivated not only by the cost of performing actual LP solves within MCTS simulations, but also by the difficulty of implementing reversible local search within MILP solvers: simulating candidate branching decisions without irreversibly altering the solver's internal state turns out to be impractical, if not impossible, through the Python interfaces currently available. Learning a model of the environment, however, comes at a well-documented price. The dynamics network's approximation errors compound over multi-step rollouts, degrading the quality of the imagined subtree trajectories on which policy improvement targets are built. In the MuZero paper itself, Schrittwieser et al. [152] observe that while the learned model matches the performance of a perfect simulator in board games, planning gains plateau much earlier in Atari, a discrepancy the authors attribute to greater model inaccuracy. More recently, He et al. [80] provided a stronger diagnosis, showing that MuZero's learned model struggles to accurately evaluate policies that deviate from its data collection distribution. They highlight that the model policy head serves a dual pur-

pose: beyond guiding the search toward promising actions, it implicitly constrains planning to regions where the model remains accurate, thereby limiting the exploration of novel strategies that could yield further improvement. These findings suggest that much of MuZero's empirical success may stem from learning a powerful state representation rather than from genuine model-based exploration and policy improvement. The implication for PlanB&B is direct: as the branching policy improves and departs from the trajectories on which the dynamics network was trained, the model becomes less reliable precisely where it matters most. The policy improvement ceiling imposed by model inaccuracy may therefore be a fundamental limitation of the current MuZero-style PlanB&B design. This is consistent with the findings of Section 7.2.3, where PlanB&B's value network fails to consistently achieve zero-shot generalization to higher-dimensional transfer instances, precisely the regime in which the dynamics model is furthest from its training distribution.

If the solver integration described in Section 8.3 were achieved, the learned dynamics network could be replaced with exact LP solves within MCTS, effectively turning PlanB&B into a full AlphaZero-style agent. Every simulated transition would then be exact, eliminating compounding model error and allowing deeper search to translate directly into better branching decisions. The dynamics network and its associated training losses would become unnecessary, simplifying the architecture and removing a source of instability during training. The policy improvement targets generated by MCTS would in turn be grounded in exact transitions, producing a cleaner supervisory signal for PlanB&B policy and value networks.

This shift involves a fundamental tradeoff. Under the current design, each MCTS simulation requires only a forward pass through the dynamics network (and two forward pass through the prediction network), enabling scaling the look-ahead search procedure to hundreds of simulations per branching step during training. Under an AlphaZero-style design, in contrast, each simulation would require one or more LP solves, which are orders of magnitude more expensive. Crucially, however, the computational results of Chapter 7 suggest that this tradeoff may be less prohibitive than it first appears. While leveraging PlanB&B's learned model for planning at test time can reduce B&B tree size, the computational overhead of running MCTS simulations rarely translates into solving time gains in practice. In fact, PlanB&B often achieves its strongest results when relying solely on its policy network at evaluation time, without any test-time planning. This indicates that the primary value of the learned model lies in generating better training signal through policy improvement targets

derived from search, rather than in being deployed as a planning engine at inference. If this is the case, then not having a lightweight surrogate model available at test time is less of a liability: what matters is the quality of the search performed during training, not its cost at deployment. Scaling computational resources to perform exact look-ahead search with the solver during training, even at the expense of fewer simulations, could therefore yield cleaner policy improvement targets and ultimately stronger policies. The prospect of planning with a perfect model of the B&B dynamics thus represents, in our view, an appealing target for the research programme outlined in this section: a fully solver-integrated learning agent that jointly optimizes branching, probing, and bounding through exact look-ahead search.

Collectively, our findings reinforce a central observation from Chapter 2: LP resolution is not merely one component among many in branch-and-bound, but the computational backbone on which many if not all others depend. Strong branching, pseudocost initialization, propagation, and, as we now argue, planning-based policy improvement all rely, directly or indirectly, on the ability to solve LP relaxations efficiently. This should come as no surprise: the very premise on which branch-and-bound rests is that LP relaxations yield bounds tight enough to guide and prune the search. A single LP solve simultaneously produces a dual bound for pruning, a fractional solution for branching heuristics, a simplex tableau for cut generation, and basis information for warm-starting subsequent solves. Accelerating LP resolution therefore constitutes an orthogonal research direction to learning better branching strategies, one that could arguably represent a high leverage target for accelerating MILP solving as a whole. Importantly, the structure of B&B makes this direction particularly amenable to learning: the LP relaxations solved throughout an episode are highly correlated, as successive nodes differ only by local bound modifications introduced by branching and propagation. In the repeated MILP setting considered in this thesis, this correlation extends across instances as well, since problems drawn from the same distribution share common constraint structure and similar feasible regions. This dual regularity, both within episodes and across instances, provides a strong inductive basis for learning fast LP surrogates from data. Furthermore, a fast approximate surrogate for LP solutions would also retain considerable value within an AlphaZero-style framework. In fact, MCTS must evaluate many candidate branching actions to identify the most promising ones, and performing a full LP solve for each candidate at every simulation step may remain prohibitively expensive. A learned model

capable of cheaply approximating dual bound improvements could serve as a lightweight filter within the search, guiding simulations toward promising branches before committing exact LP solves to a selected few. In such a hybrid design, the agent would have access to two modes of LP evaluation: one exact, producing full solver side effects, and one approximate, serving only to steer the search, effectively combining the reliability of exact transitions with the scalability of learned surrogates. In Part IV, we take a first step toward building such a surrogate model, as we explore leveraging recent advances in generative modeling to provide a learning framework capable of both producing fast LP estimates and warm-start exact LP solvers.

Part IV

Flow matching for conic optimization

Chapter 9

Preliminaries

Content

As discussed in Chapter 8, leveraging machine learning techniques to accelerate the resolution, whether exact or approximate, of repeated linear relaxations represents a high-leverage yet largely unexplored direction for improving MILP solving. The proposed approach aims to address both aspects: our learned model can serve both as a fast approximate LP solver, as well as a warm-starting procedure to accelerate the convergence of exact solvers.

The approach developed in Chapter 10 rests on a simple observation: primal-dual interior-point methods (IPMs), introduced in Section 9.1, do not jump directly to an optimal solution, but trace a structured trajectory through a primal–dual state space, driven by a barrier parameter that continuously interpolates between a well-conditioned interior point and an optimal KKT solution. Each step along this trajectory is a local, well-conditioned correction rather than a global prediction, and the trajectory itself carries far more supervisory signal than the endpoint alone. This viewpoint suggests that the natural object to learn is not a map from LP data to an optimal solution, but a continuous-time dynamical system whose integral curves approximate solver trajectories. The present chapter introduces the ingredients needed to make this idea concrete.

We first introduce primal–dual IPMs for linear programming, emphasizing their geometric and dynamical structure rather than specific implementation details. Our presentation follows that of Nocedal and Wright [129], to which we refer the reader for a thorough treatment (Chapter 14). We then review recent results by Qian et al. [141] showing that message-passing neural networks can emulate IPM iterations, which supports the premise that learning solver dynamics is more natural than learning end-to-end solution maps, while also exposing the limitations of discrete-time, architecture-dependent approaches. Finally, we introduce flow matching as a principled framework for learning continuous-time vector fields from trajectory data, and discuss its relevance to approximate interior-point dynamics.

9.1 Interior-Point Methods

Interior-point methods [129] solve linear programs by approaching optimal solutions from the interior of the feasible region. In contrast with simplex-type algorithms, which move along the boundary of the feasible polytope by pivoting between adjacent vertices, IPMs maintain strict positivity and progressively enforce primal feasibility, dual feasibility, and complementarity through smooth updates.

The distinction is thus both geometric and computational: simplex methods perform many inexpensive iterations, each moving along an edge of the polytope, while IPMs perform fewer, more expensive iterations that cut through the interior, each requiring the solution of a structured linear system. Beyond their algorithmic significance, IPMs are appealing for our purposes because their iterates admit a natural dynamical interpretation. Concretely, at each step, the solver maintains a primal–dual state (x, λ, s) satisfying strict positivity, and updates it by solving a Newton system. The resulting sequence of states evolves along a smooth curve, called the central path, parametrized by a barrier parameter that controls the trade-off between centrality and optimality. The existence of this structure motivates our approach: since the dynamics are smooth and governed by a known geometry, it is natural to ask whether they can be approximated by a learned vector field.

9.1.1 Primal–dual optimality conditions

Consider the standard-form linear program

$$\min_{x \in \mathbb{R}^n} \; c^\top x \quad \text{s.t.} \quad Ax = b, \quad x \geq 0, \tag{9.1}$$

with $A \in \mathbb{R}^{m \times n}$ of full row rank. Its dual reads

$$\max_{\lambda \in \mathbb{R}^m,\, s \in \mathbb{R}^n} \; b^\top \lambda \quad \text{s.t.} \quad A^\top \lambda + s = c, \quad s \geq 0, \tag{9.2}$$

where λ are Lagrange multipliers for the equality constraints $Ax = b$, and s are the dual slacks associated with the nonnegativity constraints $x \geq 0$. Primal–dual optimal solutions $(x^\star, \lambda^\star, s^\star)$ satisfy the Karush-Kuhn-Tucker (KKT) conditions

$$Ax = b, \tag{9.3}$$

$$A^\top \lambda + s = c, \tag{9.4}$$

$$x_i s_i = 0, \quad i = 1, \ldots, n, \tag{9.5}$$

$$(x, s) \geq 0. \tag{9.6}$$

The complementarity relations (9.5) are the main source of non-smoothness. They encode a combinatorial active-set structure: for each coordinate i, either the primal variable fades ($x_i = 0$), or the dual slack does ($s_i = 0$). Geometrically, complementarity forces optimal solutions onto the boundary of the feasible polytope. Algorithmically, it prevents a direct application of Newton's method to the

full KKT system, since the complementarity surface is non-smooth. Interior-point methods circumvent this obstacle by maintaining strict positivity ($x > 0$, $s > 0$) and approaching complementarity only in the limit.

9.1.2 Central path and barrier viewpoint

The key idea is to replace the non-smooth complementarity condition $x_i s_i = 0$ with a smooth, perturbed relaxation

$$x_i s_i = \tau, \quad i = 1, \dots, n, \tag{9.7}$$

for a barrier parameter $\tau > 0$, while enforcing primal and dual feasibility. Writing $X = \operatorname{diag}(x)$ and $S = \operatorname{diag}(s)$, the perturbed KKT system becomes

$$Ax = b, \tag{9.8}$$

$$A^\top \lambda + s = c, \tag{9.9}$$

$$XSe = \tau e, \tag{9.10}$$

$$(x, s) > 0.$$

Under standard regularity assumptions[1], this system admits a unique solution $(x(\tau), \lambda(\tau), s(\tau))$ for each $\tau > 0$. The parametric curve $\{(x(\tau), \lambda(\tau), s(\tau)) : \tau > 0\}$ is called the *central path*. As $\tau \to 0$, central-path points converge to the primal–dual optimal set.

The central path admits a variational characterization: $x(\tau)$ is the unique minimizer of the logarithmic barrier problem

$$\min_{Ax=b,\, x>0} \; c^\top x - \tau \sum_{i=1}^{n} \log x_i,$$

with corresponding dual slacks $s_i(\tau) = \tau / x_i(\tau)$. The barrier parameter τ thus controls the trade-off between optimizing the linear objective (small τ) and staying away from the boundary of the feasible region (large τ).

A convenient scalar summary of the proximity of an IPM solution to complementarity is the *duality measure*

$$\mu := \frac{x^\top s}{n}.$$

[1] Specifically, the existence of a strictly feasible primal-dual pair.

Along the central path, μ coincides with τ, while more generally, it equals the average primal–dual complementarity gap. Driving $\mu \to 0$ is therefore a natural proxy for approaching a KKT (optimal) point.

9.1.3 Path-following via Newton steps

Primal–dual IPMs approximately track the central path by applying damped Newton iterations to the perturbed KKT system (9.8)–(9.10). At an iterate (x, λ, s) with $(x, s) > 0$, we define the feasibility residuals

$$r_b := Ax - b, \qquad r_c := A^\top \lambda + s - c.$$

A path-following search direction is obtained by solving the Newton system

$$\begin{bmatrix} 0 & A^\top & I \\ A & 0 & 0 \\ S & 0 & X \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta \lambda \\ \Delta s \end{bmatrix} = - \begin{bmatrix} r_c \\ r_b \\ XSe - \sigma \mu e \end{bmatrix}, \tag{9.11}$$

where $\sigma \in [0, 1]$ is a centering parameter. The system (9.11) admits a natural block interpretation. The first two block rows reduce dual and primal infeasibility (r_c and r_b, respectively), while the third row drives the complementarity products toward the target value $\sigma\mu$. Setting $\sigma = 0$ yields the pure Newton (affine-scaling) direction, which aggressively targets complementarity but often permits only short steps before violating positivity. Conversely, setting σ close to 1 produces a centering direction that improves proximity to the central path at the expense of slow reduction in μ. Practical methods use intermediate values. Typically, predictor-corrector variants combine an affine predictor step (targeting optimality) with a centering corrector to achieve high accuracy in a limited number of iterations.

The next iterate is obtained via a damped update

$$(x, \lambda, s) \leftarrow (x, \lambda, s) + \alpha(\Delta x, \Delta \lambda, \Delta s), \qquad x > 0,\ s > 0,$$

where the step length $\alpha \in (0, 1]$ is chosen to preserve strict positivity. This constraint is what makes the method an *interior*-point method: the iterates remain strictly inside the positive orthant throughout the solving process, and the boundary is approached only asymptotically.

9.1.4 A continuous-time interpretation

The discrete Newton iteration admits a continuous-time interpretation. Let $(x(\tau), \lambda(\tau), s(\tau))$ denote the central-path solution. Differentiating the central-path equations (9.8)–(9.10) with respect to τ yields the ODE

$$A\dot{x}(\tau) = 0, \tag{9.12}$$

$$A^\top \dot{\lambda}(\tau) + \dot{s}(\tau) = 0, \tag{9.13}$$

$$S(\tau)\dot{x}(\tau) + X(\tau)\dot{s}(\tau) = e, \tag{9.14}$$

or equivalently,

$$\begin{bmatrix} 0 & A^\top & I \\ A & 0 & 0 \\ S & 0 & X \end{bmatrix} \begin{bmatrix} \dot{x} \\ \dot{\lambda} \\ \dot{s} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ e \end{bmatrix}, \tag{9.15}$$

where dots denote derivatives with respect to τ. This ODE makes explicit the intuition that IPMs induce an implicit vector field defined by the KKT geometry, with τ acting as a continuation parameter. The same structured KKT matrix appears in both the Newton system (9.11) and the central-path ODE (9.15). Importantly, the correspondence is exact only when the iterates are feasible ($r_b = 0$, $r_c = 0$) and the centering parameter is small ($\sigma \approx 0$): under these conditions, the Newton step reduces to an Euler step along the vector field (9.15). In general, the Newton system (9.11) simultaneously corrects feasibility violations and controls proximity to the central path via σ, effects that are not accounted for in the idealized central path ODE (9.15). Nevertheless, the analogy remains informative: IPM dynamics can be viewed as approximately tracing the integral curves of a smooth vector field governed by the KKT geometry, a perspective that motivates learning such a field from data.

This continuous-time viewpoint motivates the approach developed in Chapter 10. Rather than learning the idealized central-path ODE directly, we treat the actual solver trajectories, which incorporate feasibility corrections, centering, and step-length damping, as samples from a richer, empirical vector field, and ask whether this field can be learned from collections of trajectories sampled on repeated problem instances. The next two sections build directly toward this idea.

9.2 Neural Emulation of LP Solvers with Message Passing

A natural learning-based alternative to classical LP solvers is to predict an optimal solution $x^\star_{LP}$ directly from the problem data (A, b, c). However, the solution map $(A, b, c) \mapsto x^\star_{LP}$ is piecewise linear, and governed by an active-set structure that can change abruptly under small perturbations of the data. Moreover, any one-shot predictor must implicitly learn to satisfy simultaneously global coupled sets of constraints, from primal-dual feasibility to complementary slackness, which makes robust generalization difficult. Crucially, Qian et al. [141] proposed a different strategy: rather than learning the solution map directly, they asked whether message-passing graph neural networks can emulate the iterative dynamics of an interior-point solver.

9.2.1 Tripartite graph encoding

LP instances are represented as *tripartite* graphs extending the bipartite graph encoding from Gasse et al. [62] by adding a single objective node connected to every variable node and every constraint node, with edge weights carrying the cost coefficients c_j and right-hand side values b_i respectively. This encoding is designed so that linear-algebraic quantities appearing in primal–dual updates (e.g., matrix-vector products Ax, $A^\top \lambda$, diagonal scalings, and inner products) can be expressed through local message passing and aggregation.

9.2.2 Message-passing GNNs for IPM emulation

The main theoretical result of Qian et al. [141] shows that, with an appropriate architecture and parameter choices, a finite-depth message-passing neural network (MPNN) can reproduce one iteration of standard primal–dual IPM variants. Building on this result, the authors propose IPM-MPNN: a GNN trained in a supervised fashion to predict the intermediate iterates produced by a reference IPM solver. Concretely, the network is trained not only to match the final solution, but to track a sequence of solver iterates, with additional regularization terms encouraging primal and dual feasibility as well as near-optimal objective values.

This work supports a central premise of the present thesis: learning solver dynamics is more natural and more robust than learning a direct end-to-end map from problem data to optimal solutions. By tracking intermediate solver states, the learning task is decomposed into local, well-conditioned

regression problems rather than a single global prediction.

At the same time, the IPM-MPNN framework exhibits two structural limitations that motivate the approach developed in Chapter 10. First, trajectory emulation is inherently discrete-time: a network with L layers imitates L solver iterations, and there is no mechanism to interpolate between or extrapolate beyond the training regime. Second, the architecture is rigidly coupled to the solver: each network layer emulates one IPM iteration, so that imitating longer trajectories requires proportionally deeper networks, incurring higher memory cost and increased training difficulty. In practical terms, since production IPM solvers typically converge in 30–40 iterations regardless of problem size [68], an IPM-MPNN targeting full convergence would require a network of corresponding depth.

In the next chapter, we pursue a complementary approach that seeks to address these limitations. Leveraging the continuous-time interpretation of IPMs in (9.15), we seek to learn a continuous vector field whose integral curves emulate interior-point trajectories. Importantly, a single network of fixed depth is queried repeatedly by an ODE integrator, decoupling the model architecture from the length of the emulated trajectory.

9.3 Flow Matching

The previous section introduces a clear learning objective: rather than predicting $x^\star_{LP}$ directly, we seek to learn a continuous-time dynamical system that reproduces the trajectories generated by interior-point solvers. Flow matching provides a principled framework for this task. Originally introduced in the context of generative modeling [111], it reduces the problem of learning a continuous-time vector field to supervised regression on local velocities, without requiring backpropagation through ODE solutions during training.

9.3.1 Learning vector fields via transport

Flow matching formulates the problem of learning a transport map between two distributions on $\mathbb{R}^d$. Let p_0 denote a simple reference distribution (e.g., a standard Gaussian) and p_1 a target distribution. Rather than learning a direct map T such that $T(z_0) \sim p_1$ for $z_0 \sim p_0$, flow matching learns a time-dependent vector field $v_\theta : \mathbb{R}^d \times [0,1] \to \mathbb{R}^d$ whose ODE flow transports p_0 to p_1:

$$\frac{dz_t}{dt} = v_\theta(z_t, t), \qquad z_0 \sim p_0. \tag{9.16}$$

Generating a sample from p_1 then reduces to numerically integrating (9.16) from $t = 0$ to $t = 1$, making the computational accuracy trade-off explicit and tunable.

9.3.2 The flow matching objective

The key idea is to train v_θ through local velocity regression rather than explicit distribution matching. In fact, training continuous normalizing flows via maximum likelihood [33] requires integrating the learned ODE during training and backpropagating through the numerical solver, which is computationally expensive and can be numerically unstable for long integration horizons. Flow matching avoids this entirely. Let $(z_0, z_1) \sim \pi$ be a coupling between samples from p_0 and p_1. Assume we can define an interpolation process $(z_t)_{t\in[0,1]}$ such that: (i) the conditional distribution $p_t(\cdot \mid z_0, z_1)$ is easy to sample, and (ii) the process admits a known conditional velocity field $u_t(\cdot \mid z_0, z_1)$. Flow matching then minimizes

$$\mathcal{L}(\theta) = \mathbb{E}_{(z_0,z_1)\sim\pi,\, t\sim\mathcal{U}[0,1],\, z_t\sim p_t(\cdot|z_0,z_1)} \left[\|v_\theta(z_t, t) - u_t(z_t \mid z_0, z_1)\|^2 \right]. \tag{9.17}$$

Intuitively, the model is trained to match the local motion prescribed by the chosen bridge between z_0 and z_1. Crucially, training does not require any ODE integration, only sampling intermediate states z_t and evaluating the target velocity u_t.

9.3.3 Straight-line interpolation

A particularly simple and widely used instantiation for the conditional path $p_t(\cdot \mid z_0, z_1)$ chooses the deterministic linear interpolation

$$z_t = (1 - t)z_0 + tz_1, \tag{9.18}$$

for which the target velocity is constant:

$$u_t(z_t \mid z_0, z_1) = z_1 - z_0.$$

The objective (9.17) then reduces to

$$\mathcal{L}(\theta) = \mathbb{E}_{(z_0,z_1)\sim\pi}\, \mathbb{E}_{t\sim\mathcal{U}[0,1]} \left[\|v_\theta((1 - t)z_0 + tz_1, t) - (z_1 - z_0)\|^2 \right]. \tag{9.19}$$

This can be interpreted as an optimal-transport path between paired samples: the straight line from z_0 to z_1 is the displacement mapping that minimizes kinetic energy [111].

9.3.4 Connection to density evolution

The ODE (9.16) governs individual trajectories, but flow matching is ultimately concerned with transporting distributions. When particles distributed according to p_0 evolve under (9.16), the density p_t satisfies the continuity equation

$$\partial_t p_t(z) + \nabla \cdot \big(p_t(z)\, v_\theta(z,t)\big) = 0, \tag{9.20}$$

encoding conservation of probability mass: the velocity field v_θ fully determines how mass redistributes over time, with no particles created or destroyed. Flow matching thus learns a vector field controlling the collective transport of p_0 toward p_1, with the regression objective of Eq. (9.17) providing a simulation-free means of fitting this field from paired samples. This perspective is conceptually aligned with our objective: interior-point methods trace trajectories through a primal–dual state space governed by the KKT geometry, and the velocity field we seek to learn is precisely the continuous-time dynamics driving these trajectories.

9.3.5 Relevance to interior-point dynamics

In the context of generative modeling for which flow matching was originally developed, p_0 is typically a standard Gaussian and p_1 a target data distribution (e.g., images) from which one seeks to sample. In our setting, both distributions carry different meaning. The source distribution p_0 will encode initial primal–dual state, for instance, default solver initializations or points in a neighborhood of the analytic center, while the target distribution p_1 will encode near-optimal primal–dual solutions produced by an IPM solver. Under this interpretation, the flow matching objective (9.17) trains a vector field whose integral curves transport initial states to near-optimal solutions along learned paths. The key structural advantage over one-shot prediction is that the learned dynamics decompose the prediction into a sequence of local, incremental corrections analogous to the Newton steps of an IPM, each of which is expected to be a better-conditioned regression target, at least away from the terminal regime where trajectories collapse onto a point mass and the velocity field concentrates. The next chapter instantiates this framework for linear programming.

Chapter 10

Flow matching for linear programming

Content

Chapter 9 established two key ingredients: primal-dual interior-point methods, which trace smooth trajectories approaching LP optimality from the interior of the polytope; and flow matching, which provides a simulation-free regression objective for learning continuous-time vector fields from paired trajectory data. The present chapter explores bringing these two ingredients together, developing a concrete framework for learning to solve repeated linear programs.

Importantly, our setting differs from that of traditional generative flow matching, where the learned vector field defines an artificial transport between noise and target data distributions. Here, the object of interest is the trajectory structure of primal–dual interior-point solvers, whose iterates approximately follow the central-path ODE while additionally incorporating feasibility corrections, centering heuristics, and step-length damping. The neural network serves as a cheap surrogate trained on these solver trajectories to emulate their dynamics at a fraction of the computational cost. This places our approach in the tradition of learned numerical emulators [98, 136], where the governing equations are known but their repeated evaluation is the computational bottleneck. The flow matching objective [111] is well suited to this task, as it reduces to local velocity regression without requiring backpropagation through the ODE integration.

Our framework is intended to serve a dual purpose within the broader context of this thesis. First, during the resolution of MILP instances, integrating the learned vector field to $t = 1$ provides a fast approximate solution to the LP relaxation associated with a candidate branching decision, yielding an estimate of the corresponding dual bound improvement without incurring the full cost of an exact LP solve. This equips branching agents with a lightweight predictive model that can be leveraged to inform both probing and branching decisions in ProbBBMDP. Second, the framework provides a standalone mechanism for accelerating the resolution of repeated LPs: by integrating the learned vector field to an intermediate time $t < 1$, one obtains an approximate primal-dual iterate situated near the central path, which can then be leveraged to warm-start an exact interior-point solver. This procedure is expected to significantly reduce the number of iterations needed to reach optimality, providing a substantial speedup as each IPM iteration requires solving a structured linear system at a cost of $O(n^2m + nm^2)$.

We stress that, while our framework is initially designed for accelerating linear programming, the construction generalizes directly to conic optimization. In fact, interior-point methods naturally extend to general conic programs (including second-order cone and semidefinite programs) by replacing

the logarithmic barrier $-\sum_i \log x_i$ with the corresponding self-concordant barrier for the cone, which preserves the block structure of the Newton dynamic system. The flow matching framework developed here is therefore applicable to any problem class for which IPMs can generate target demonstration trajectories.

10.1 From the central path to supervision target flows

A natural starting point for connecting interior-point methods to flow matching is the observation, developed in Section 9.1.4, that the central path admits a continuous-time interpretation as an ODE. This section examines carefully this connection. We first show that the central-path ODE defines a smooth vector field on the strictly feasible interior, satisfying the standard regularity properties required of a flow. We then explain why, despite this appealing structure, the central-path vector field is not itself the object we seek to learn: its integral curves transport only points that already lie on the central path to the optimal solution, whereas the flow we require must transport arbitrary points sampled from the strictly feasible interior. This observation motivates the approach developed in the remainder of the chapter: generating trajectory bundles by perturbing the solver's initialization, thereby constructing regression targets that approximate the empirical vector field induced by the full IPM dynamics, rather than by the central-path ODE alone.

10.1.1 The central-path vector field

Consider a standard-form linear program as defined in (9.1), with $A \in \mathbb{R}^{m\times n}$ of full row rank. IPM solvers maintain a primal–dual iterates of the form $y = (x, \lambda, s) \in \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^n$, where x is the primal variable, λ the dual variable, and s the dual slack. The state y lies in $\mathbb{R}^d$ with $d = 2n + m$. Under the non-degeneracy assumption adopted in this work, the LP admits a unique primal–dual optimal solution $y^\star = (x^\star, \lambda^\star, s^\star)$, and the central path converges to $y^\star$ as $\tau \to 0$.

Recall from Section 9.1.4 that differentiating the perturbed KKT system with respect to the barrier parameter τ yields the central-path ODE

$$\begin{bmatrix} 0 & A^\top & I \\ A & 0 & 0 \\ S & 0 & X \end{bmatrix} \begin{bmatrix} \dot{x} \\ \dot{\lambda} \\ \dot{s} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ e \end{bmatrix}, \tag{10.1}$$

where dots denote differentiation with respect to τ. In the following, we define $M(y) \in \mathbb{R}^{(2n+m)\times(2n+m)}$

such that Eq. (10.1) can be written

$$M(y)\,\dot{y} = \begin{bmatrix} 0 \\ 0 \\ e \end{bmatrix}. \tag{10.2}$$

A key observation is that the vector field $f(y) = M(y)^{-1}[0\ 0\ e]^T$ is well defined not only on the central path itself, but on the entire open set

$$\mathcal{D}^\circ = \{(x, \lambda, s) \in \mathbb{R}^d \mid x > 0,\ s > 0\}, \tag{10.3}$$

since the KKT matrix $M(y)$ is non-singular whenever A has full row rank and $(x, s) > 0$ [129]. One can therefore consider the autonomous ODE

$$\dot{y} = f(y) = M(y)^{-1} \begin{bmatrix} 0 \\ 0 \\ e \end{bmatrix}, \qquad y \in \mathcal{D}^\circ, \tag{10.4}$$

as a dynamical system on $\mathcal{D}^\circ$.

10.1.2 Flow properties of the extended vector field

The entries of $M(y)^{-1}$ are rationale functions of (x, s) with no poles in $\mathcal{D}^\circ$, rendering f smooth on $\mathcal{D}^\circ$. In particular, f is locally Lipschitz continuous, the Cauchy–Lipschitz theorem thus guarantees existence and uniqueness of solutions to (10.4) for any initial condition $y_0 \in \mathcal{D}^\circ$. The smooth dependence theorem for ODEs further ensures that the flow map $\phi_t : \mathcal{D}^\circ \to \mathcal{D}^\circ$, defined by $\phi_t(y_0) = y_t$, is smooth in (y_0, t) and satisfies standard flow properties: (i) identity $\phi_0 = \mathrm{id}$, (ii) group law $\phi_{t+s} = \phi_t \circ \phi_s$, (iii) and ϕ_t is a C^∞ diffeomorphism for each t, with inverse ϕ_{-t}.

Importantly, the first two block rows of the ODE enforce $A\dot{x} = 0$ and $A^\top\dot{\lambda} + \dot{s} = 0$, so that primal and dual feasibility are preserved along trajectories: if y_0 satisfies $Ax_0 = b$ and $A^\top\lambda_0 + s_0 = c$, then these relations hold for all t. In other words, the strictly feasible interior set

$$\mathcal{F}^\circ = \{(x, \lambda, s) \in \mathbb{R}^d \mid Ax = b,\ A^\top\lambda + s = c,\ x > 0,\ s > 0\}, \tag{10.5}$$

is an invariant submanifold of the flow. The central path represents the unique integral curve of (10.4) that lies on $\mathcal{F}^\circ$ and satisfies the complementarity condition $XSe = \tau e$ for some $\tau > 0$.

At first glance, this setting appears ideally suited for flow matching: the central-path ODE defines a smooth, well-posed flow on a natural state space, with all the regularity that the framework requires. However, as we now argue, the central path alone does not provide the trajectory diversity that flow matching needs as training data.

10.1.3 A single curve is not a flow matching dataset

As described in Section 9.3, flow matching learns a vector field by regressing against conditional velocities along a family of trajectories connecting samples from a source distribution p_0 to samples from a target distribution p_1. The diversity of these trajectories is essential: it is what allows the learned vector field to generalize beyond any single path and to define a meaningful transport map between distributions.

The central path, by contrast, is a single parametric curve $\tau \mapsto y(\tau)$ explicitly defined for each LP instance. While the extended vector field (10.4) does define integral curves from arbitrary initial conditions in $\mathcal{D}^\circ$, integrating this field from a point sampled outside the central path does not produce a trajectory converging to the LP optimal solution $y^\star$: the central-path ODE encodes the geometry of the barrier trajectory, not a global transport toward optimality. Consequently, integral curves originating off the central path may reach the boundary of $\mathcal{D}^\circ$ (where x_i or s_i hits zero) in finite time without ever approaching $y^\star$. The central-path ODE is therefore not the vector field we seek to learn: while it suggests that the underlying IPM dynamics are smooth, structured, and ultimately governed by the KKT geometry, it does not, by itself, provide the global transport supervision that flow matching requires.

10.1.4 From one path to many: perturbed solver trajectories

The preceding analysis suggests a natural resolution: rather than relying on the single canonical trajectory provided by the central path, we generate a family of IPM trajectories converging to a KKT points. Concretely, for each LP instance, we perturb the solver's canonical initialization with random noise (while ensuring that the perturbed point remains in the interior of the positive orthant), and run the IPM solver from this perturbed starting point. Under the non-degeneracy assumption, all perturbed trajectories converge to the same optimal solution $y^\star$, but they traverse different regions of the primal–dual state space on their way there.

This construction yields exactly the structure that flow matching requires: a bundle of conditional paths, indexed by noise realizations, all sharing the same terminal point but differing in their initial conditions and intermediate states. The learned vector field is then trained to match the empirical velocities along these paths, effectively learning a feedback law that steers any starting point $y_0 \in \mathcal{D}^\circ$

toward $y^\star$ following IPM dynamics.

10.2 Learning interior-point dynamics

10.2.1 Perturbation of the canonical initialization

Let $y_0^{\mathrm{c}} = (x_0^{\mathrm{c}}, \lambda_0^{\mathrm{c}}, s_0^{\mathrm{c}})$ denote the solver's default (canonical) initialization, typically computed via the Mehrotra heuristic [129]. The canonical initialization satisfies $(x_0^{\mathrm{c}}, s_0^{\mathrm{c}}) > 0$ and approximately satisfies primal and dual feasibility, i.e., $Ax_0^{\mathrm{c}} \approx b$ and $A^\top \lambda_0^{\mathrm{c}} + s_0^{\mathrm{c}} \approx c$. Our goal is to generate a *family* of starting points whose trajectories under the IPM solver all converge to the same optimal solution $y^\star$, thereby creating a bundle of paths suitable for flow matching supervision.

We construct perturbed starting points as follows. Given a noise scale $\sigma_\varepsilon > 0$, we sample a perturbation $\varepsilon \sim \mathcal{N}(0, \sigma_\varepsilon^2 I_d)$ and form the raw perturbed state $\tilde{y} = y_0^{\mathrm{c}} + \varepsilon$. This perturbation generically violates the positivity requirements, and must therefore be corrected before it can serve as a valid initialization. Following the initialization procedure of the Clarabel interior-point solver [69], we recover strict positivity by applying a uniform additive shift to $\tilde{x}$ and $\tilde{s}$ independently. Let $m = \min_i \tilde{x}_i$ denote the minimum component and $p = \sum_i \max(\tilde{x}_i, 0)$ the sum of positive components. The target margin is

$$\delta = \max\left(1,\ \frac{\alpha\, p}{n}\right), \tag{10.6}$$

where $\alpha = 0.1$ and n is the dimension. The shifted variable is then

$$x_0 = \tilde{x} + \max(0,\ \delta - m) \cdot e, \tag{10.7}$$

and analogously for s_0. This ensures $[x_0]_i \geq \delta > 0$ for all i, with the shift magnitude determined globally rather than coordinate-wise[1]. The dual variable λ_0 requires no correction as it is unconstrained. The resulting point $y_0 = (x_0, \lambda_0, s_0)$ thus strictly belongs to the interior of the positive orthant, and can be used to initialize the IPM solver. The noise scale σ_ε controls the diversity of the generated trajectory bundle: if too small, all perturbed trajectories cluster near the canonical path and the learned vector field sees limited variability. In contrast, if σ_ε is too large, perturbed starting points may lie far from the central path, leading to trajectories with long transient phases dominated by feasibility restoration rather than central-path following. In practice, we calibrate $\sigma_\varepsilon = 0.1$ so that perturbed starting points remain within a neighborhood of the central path.

[1] We use $[\cdot]_i$ to denote the i-th component of a vector.

10.2.2 Flow-IPM training

A distinguishing feature of our training procedure is that each LP instance is used exactly once. A fresh instance is sampled from a parametric instance generator such as those presented in Appendix B, solved from a single perturbed initialization, and never visited again. This design mirrors the standard setting in generative modeling [84, 111], where each training sample is drawn independently from the data distribution and contributes a single gradient update. In our context, the "data distribution" is the distribution over LP instances induced by the generator, and each "sample" is a solver trajectory on a freshly generated instance. Training on unique instances encourages the learned vector field to capture the structural regularities of IPM dynamics across the instance family, rather than memorizing instance-specific trajectories. In the following, we refer to the resulting framework as *Flow-IPM*. The velocity model is a parametric vector field $v_\theta : \mathbb{R}^d \times [0,1] \times \mathcal{P} \to \mathbb{R}^d$, where $\mathcal{P}$ denotes the space of LP instances, trained by minimizing

$$\mathcal{L}(\theta) = \mathbb{E}_{P\sim\mathcal{D}}\, \mathbb{E}_{y_0\sim p_0}\, \mathbb{E}_{t\sim\mathcal{U}[0,1]} \left[\left\| v_\theta(y_t, t \mid P) - u_t \right\|^2 \right], \tag{10.8}$$

where $\mathcal{D}$ is the training distribution over LP instances, $p_0 \approx \mathcal{N}(y_0^c, \sigma_\varepsilon^2 I_d)$ denotes the distribution of perturbed initializations[2], and the pair (y_t, u_t) is obtained from the solver trajectory generated from y_0 as described below.

Generating perturbed IPM trajectories. Concretely, the training loop proceeds as follows. At each iteration, an LP instance $P = (A, b, c)$ is sampled from the instance generator. A perturbed starting point y_0 is constructed using the procedure of Section 10.2.1, and the reference IPM solver is run from y_0 to convergence, producing a trajectory of $N + 1$ iterates $\{y_k\}_{k=0}^N$. To generate denser trajectories providing finer-grained velocity supervision, the solver's step-length damping parameter is set to $\alpha = 0.05$, yielding trajectories of approximately $N \approx 10^3$ iterates. The terminal iterate y_N is a near-optimal primal–dual solution satisfying $\mu_N \le \mu_{\min}$ for a prescribed tolerance $\mu_{\min}$.

Time re-indexation. Flow matching operates on a continuous time interval $[0, 1]$, whereas the solver produces a discrete sequence of iterates indexed by iteration count k. To bridge this gap, we re-parametrize trajectories by arc length. Given the raw solver trajectory $\{y_k\}_{k=0}^N$, we compute the

[2]This is a notational convenience: the actual initial point y_0 is not drawn directly from $\mathcal{N}(y_0^c, \sigma_\varepsilon^2 I_d)$, but undergoes the shifting procedure described in Section 10.2.1 to enforce feasibility.

cumulative Euclidean distances

$$\ell_k = \sum_{j=0}^{k-1} \|y_{j+1} - y_j\|, \qquad \ell_0 = 0, \tag{10.9}$$

and resample the trajectory at $K = 500$ equally spaced arc-length values by linear interpolation, yielding points $\{\bar{y}_i\}_{i=0}^{K-1}$ with uniform time stamps $t_i = i/(K-1)$. This construction ensures that equal increments in t correspond to equal displacements in the primal–dual state space, producing velocity targets of uniform magnitude along the trajectory.

Velocity targets. Velocity targets are obtained by finite differences on the resampled trajectory. The velocity target at the i-th resampled point is

$$u_i = \frac{\bar{y}_{i+1} - \bar{y}_i}{t_{i+1} - t_i} \tag{10.10}$$

For a sampled time $t \in [t_i, t_{i+1}]$, the state and velocity are obtained by linear interpolation:

$$y_t = (1-\alpha)\,\bar{y}_i + \alpha\,\bar{y}_{i+1}, \qquad u_t = u_i, \qquad \alpha = \frac{t - t_i}{t_{i+1} - t_i}. \tag{10.11}$$

We sample $S = 16$ time points per instance using a logit-normal distribution centered at $t = 0.5$. This sampling strategy, introduced by Esser et al. [54] and now standard practice in flow matching, concentrates training signal in the mid-trajectory region where the velocity field typically exhibits the greatest spatial variation. Each sampled time yields a training tuple (t, y_t, u_t, P). The resulting S tuples are used to evaluate (10.16) and update the parameters of v_θ via a single gradient step, after which the instance P is discarded.

The main computational cost of this online protocol is that each training step requires solving an LP to generate the trajectory, which dominates the cost of the gradient update itself. In practice, this cost is amortized by batching multiple instances in parallel.

10.2.3 Relationship with generative flow matching

The loss (10.16) takes the same functional form as the conditional flow matching (CFM) objective of Lipman et al. [111] recalled in Section 9.3: both regress a learned vector field against conditional velocities along interpolating paths. However, our setting differs from generative CFM in three aspects.

First, for a fixed LP instance P, all perturbed trajectories converge to the same optimal solution $y^\star(P)$. The target distribution is therefore a point mass rather than a diffuse distribution, making

the transport contractive. The velocity field necessarily diverges as $t \to 1$, which is why we integrate only to $t < 1$ at inference time. Second, the vector field is conditioned on the LP instance P, and each instance induces its own dynamics. The model must therefore generalize across the instance distribution $\mathcal{D}$, learning to produce instance-specific velocity fields for unseen problem data, rather than learning a single fixed transport map as in standard generative modeling. Third, the conditional paths are genuine solver trajectories rather than synthetic interpolations (e.g., straight lines), and the conditional velocity field is approximated by finite differences rather than available in closed form.

These departures mean that the density-transport guarantees of CFM do not directly apply. Nevertheless, the regression objective remains well-posed: each training tuple provides an unbiased estimate (up to discretization error) of the local velocity along a deterministic trajectory. The learned vector field v_θ can thus be understood as a neural approximation of the instance-dependent central-path dynamics, one that additionally accounts for the deviations introduced by the solver's centering corrections, step-length damping, and other practical heuristics.

10.2.4 Optimal-transport baseline

To assess the value of using solver-generated trajectories as conditional paths, we compare our IPM-based flow against a vanilla flow matching baseline that uses straight-line interpolation, as introduced in Section 9.3.3. In this baseline, the conditional path between a source sample $y_0 \sim \mathcal{N}(0, I_d)$ and the optimal solution $y^\star(P)$ obtained from the solver is the deterministic linear interpolation

$$y_t = (1 - t)\, y_0 + t\, y^\star, \qquad t \in [0, 1], \tag{10.12}$$

for which the conditional velocity is constant along the path:

$$u_t = y^\star - y_0. \tag{10.13}$$

The training objective is identical to (10.16), with the pair (y_t, u_t) now given by (10.12)–(10.13) instead of being extracted from a solver trajectory. The architecture, hyperparameters, and training protocol (online instance generation, logit-normal time sampling, single gradient step per instance) are kept the same, so that any performance difference can be attributed to the choice of conditional path.

This baseline represents the most direct application of flow matching to LP solving: the network must learn to transport unstructured Gaussian noise to the optimal solution along straight lines,

conditioned solely on the instance data (A, b, c), without access to any kind of solver supervision. Comparing the two approaches isolates the benefit of incorporating IPM dynamics into the conditional paths. Where the OT baseline must learn the entire transport from an unstructured source distribution to the optimal solution, the IPM-based flow decomposes the prediction into local velocity targets along a geometrically structure trajectory, yielding what we expect to be better-conditioned supervision targets.

10.3 Model architecture

The learned vector field $v_\theta(y, t \mid P)$ must accept both primal–dual state $y \in \mathbb{R}^d$ and time $t \in [0, 1]$ as inputs, condition on the LP instance $P = (A, b, c)$, and output a velocity vector in $\mathbb{R}^d$. Unlike Qian et al. [141], who adopt a tripartite graph encoding with a dedicated objective node, we retain the bipartite variable–constraint representation proposed by Gasse et al. [62]. The key difference between these representation models lies in how the objective function c participates in message passing. In the tripartite encoding, a single objective node is connected to every variable and every constraint node, enabling global quantities such as $c^\top x$ to be computed via message aggregation. However, this fully-connected macro node incurs a computational overcost in GNN architectures: message passing must be carried out in three stages per layer instead of two, substantially increasing both training and inference time.

In the bipartite encoding, the objective vector enters the network through the variable node features: each variable node j receives the cost coefficient c_j as an input feature, and the constraint matrix coefficients A_{ij} appear as edge weights. We note that the same representational effect as the tripartite objective node can be achieved within the bipartite framework through a simple reformulation of the LP, by augmenting the formulation with a single auxiliary variable z and a single linking constraint $z = c^\top x$:

$$\tilde{A} = \begin{bmatrix} c^\top & -1 \\ A & 0 \end{bmatrix}, \qquad \tilde{b} = \begin{bmatrix} 0 \\ b \end{bmatrix}, \qquad \tilde{c} = \begin{bmatrix} c \\ 0 \end{bmatrix}. \tag{10.14}$$

In the bipartite graph of the augmented $\tilde{P} = (\tilde{A}, \tilde{b}, \tilde{c})$, the objective coefficients c_j appear as edge weights connecting all original variable nodes to the new constraint node enforcing $z = c^\top x$, so that information about the objective function propagates to every node within two message-passing rounds. In practice, encoding c_j directly as a node feature achieves comparable performance and is the approach

retained for the experiments reported here.

The architecture comprises four stages: input encoding, time-conditioned message passing, time-conditioned decoding, and velocity prediction. The message-passing backbone is based on the GCN+ convolution [115], a graph convolutional operator that incorporates edge features into the message function via an additive interaction followed by a nonlinearity. We describe each stage below. The following subsections detail the architecture retained for our model; readers less concerned with architectural specifics may skip directly to Section 10.3.5.

10.3.1 Input encoding

Each node receives a feature vector obtained by concatenating static instance features, the dynamic primal–dual state at time t, and a random walk positional encoding (RWPE) [49]. For variable nodes, the static features consist of the objective coefficient c_j, the variable bounds l_j and u_j, and summary statistics (mean and standard deviation) of the corresponding column of A; for constraint nodes, the right-hand side b_i and row-wise summary statistics of A. The dynamic state appended to each node is the interpolated primal value $x_j(t)$ for variable nodes and the interpolated dual multiplier $\lambda_i(t)$ for constraint nodes. The RWPE provides each node with a d_{pe}-dimensional signature encoding its local graph topology, computed on a homogeneous view of the bipartite graph.

The concatenated feature vector is projected to a hidden dimension D by a two-layer MLP with intermediate layer normalization and SiLU activation [51], separately for variable and constraint nodes. Edge features are likewise projected to $\mathbb{R}^D$ by a shared two-layer MLP.

10.3.2 Time-conditioned message passing

The time $t \in [0, 1]$ is encoded via a sinusoidal positional embedding [168] followed by a two-layer MLP, producing a per-node time embedding $\mathbf{t} \in \mathbb{R}^D$.

GCN+ convolution. The core message-passing operator is the GCN+ convolution [115], which extends the standard GCN formulation by incorporating edge features directly into the message function. For an edge from source node i to destination node j with edge embedding $\mathbf{e}_{ij} \in \mathbb{R}^D$, the message is obtained by applying a SiLU activation to the sum of linearly projected source features and edge features. Messages are then aggregated over the source neighborhood and combined with a linear

self-connection that preserves the destination node's own representation. At each layer, two separate GCN+ convolutions handle the two edge directions of the bipartite graph: constraint-to-variable and variable-to-constraint.

Block structure. Each message-passing layer wraps the GCN+ convolution in a residual block with the following structure. The convolution output is first layer-normalized, passed through a SiLU activation and dropout, then added to the input via a scaled residual connection with factor $1/\sqrt{L}$, where L denotes the total number of layers. A feed-forward network follows, consisting of a pre-norm layer normalization, a two-layer MLP with $2D$ hidden units and SiLU activation, and a second scaled residual addition. The time embedding is then injected via Feature-wise Linear Modulation (FiLM) [135], which applies a learned element-wise affine transformation of the time embedding to the node representations. A final layer normalization completes the block.

Stacking and aggregation. The network stacks $L = 8$ such blocks, with hidden dimension $D = 256$. The scaled residual factor $1/\sqrt{L}$ prevents representation norm explosion in deep networks by ensuring that residual contributions decay with depth. Intermediate representations from all layers are aggregated via Jumping Knowledge (JK) with element-wise max pooling [183], which allows the decoder to access features from any depth and mitigates over-smoothing.

10.3.3 Time-conditioned decoding

After message passing, three separate decoder heads produce representations for the primal, dual, and slack velocity outputs. Each head consists of a stack of three residual blocks with Adaptive Layer Normalization (AdaLN) for time conditioning. In each block, the layer-normalized hidden representation is modulated by a learned affine transformation of the time embedding — analogous to FiLM but applied after normalization — before being processed by a feed-forward network with $4D$ hidden units and a residual connection. The last decoder block is zero-initialized, so that the full decoder initially acts as an identity map.

The primal and dual decoder heads each take as input the corresponding node representations from the message-passing backbone. The slack decoder head takes the constraint node representations augmented with a projection of the current slack state $s_i(t)$. The slack state is not available during message passing and is injected only at decode time via a two-layer MLP: $\mathbf{h}_i^{\text{slack}} = \mathbf{h}_i^{\text{con}} + \text{MLP}(s_i(t))$.

This design allows the slack decoder to benefit from the shared constraint representations built during message passing while learning features specific to slack dynamics.

10.3.4 Velocity prediction

Each decoder head terminates with a graph normalization layer followed by a zero-initialized linear projection to a single scalar per node. The primal head predicts $\dot{x}_j \in \mathbb{R}$ at each variable node, the dual head predicts $\dot{\lambda}_i \in \mathbb{R}$ at each constraint node, and the slack head predicts $\dot{s}_i \in \mathbb{R}$ at each constraint node. The full velocity vector $v_\theta(y, t \mid P) = (\dot{x}_1, \ldots, \dot{x}_n, \dot{\lambda}_1, \ldots, \dot{\lambda}_m, \dot{s}_1, \ldots, \dot{s}_n) \in \mathbb{R}^d$ is assembled by concatenation. This decomposition respects the structure of the primal–dual state: the network predicts local updates for each variable and constraint, and the global coherence of the velocity (e.g., the relation $A\dot{x} = 0$ along feasible trajectories) can be enforced implicitly through message passing rather than by explicit projection.

Zero initialization of the output heads ensures that the model predicts near-zero velocities at the start of training, providing a safe initialization since the true velocity field has bounded magnitude along solver trajectories.

10.3.5 Forward integration

At inference time, given a new LP instance $P_{\text{new}} = (A, b, c)$, we generate an approximate primal–dual solution by numerically integrating the learned vector field:

$$\frac{dy}{dt} = v_\theta(y, t \mid P_{\text{new}}), \qquad y_0 = y_{\text{init}}, \tag{10.15}$$

from $t = 0$ to $t = T \leq 1$, using an adaptive ODE solver [47].

The initial state y_{init} is the canonical IPM initialization y_0^{c}, computed via the Mehrotra heuristic as described in Section 10.2.1. Note that while the model is trained on perturbed initializations, the canonical initialization lies within the support of the perturbation distribution.

The terminal time T controls the trade-off between the quality of the neural approximation and the remaining work left for the exact solver. Integrating to $T = 1$ produces the model's best estimate of the optimal solution, suitable for fast approximate LP solving. Integrating to $T < 1$ produces an intermediate iterate that can be used to warm-start an exact solver, as described below.

Interestingly, FlowIPM naturally accommodates the computational trade-offs inherent to model-based planning. The number of integration steps can be adjusted at inference time without retraining: a coarse discretization of $[0, 1]$ yields a fast but rough estimate of the LP solution, while a finer discretization improves accuracy at additional cost. This tunability is particularly relevant in the context of ProbBBMDP described in Chapter 8, where the agent must evaluate many candidate LP relaxations under a limited computational budget. By controlling the integration resolution, the agent can allocate more computation to promising candidates and less to those that are quickly ruled out.

10.3.6 Warm-starting the exact solver

An approximate solution y_T ($T < 1$) can be passed to an exact IPM solver to serve as an initial starting point. Because y_T is assumed to be situated in a neighborhood of the central path with small duality measure, in principle the exact solver requires fewer iterations to converge compared to a cold start. Before initialization, the feasibility residuals $r_b = Ax_T - b$ and $r_c = A^\top \lambda_T + s_T - c$ are computed to quantify constraint violations introduced by the learned dynamics. If the violations exceed a prescribed threshold, y_T is projected onto the feasible set via a single Newton correction step. The exact solver is then initialized with the corrected state and run to full precision. This two-phase approach ensures that the final solution inherits the correctness guarantees of the classical IPM while benefiting from the speed of the learned warm start. Importantly, unlike in the data generation phase where α is reduced to produce denser trajectories (cf. Section 10.2.2), the exact solver used for warm-starting retains its default step-length damping, ensuring that iteration savings are measured under realistic operating conditions.

10.4 Experimental study

The experiments presented below serve as an initial empirical investigation of the FlowIPM framework, with the dual aim of validating the theoretical construction developed in this chapter and identifying its practical limitations, while also surfacing empirical insights that may inform future refinements or extensions of the approach. The primary objective is to assess whether a GNN-based velocity field, trained via flow matching on IPM trajectories, can learn to produce primal–dual iterates of sufficient quality to (i) approximate optimal LP solutions and (ii) warm-start an exact solver with measurable iteration savings. The experimental setting is deliberately kept to a single problem family

in order to isolate the respective benefits associated with the learning dynamics of both IPM and OT flow models.

10.4.1 Experimental setting

Problem instances. We consider combinatorial auction LP relaxations generated using the Ecole library. Each instance is drawn with a randomized number of items $n_{\text{items}} \sim \mathcal{U}\{50, 150\}$ and bids $n_{\text{bids}} \sim \mathcal{U}\{150, 300\}$, yielding LPs with variable and constraint counts that vary across instances. Prior to graph construction, each LP is converted to standard inequality form $Ax \leq b$ by absorbing variable bounds into the constraint matrix, and then row-normalized: each row of A is divided by its Euclidean norm, and the objective vector c is divided by $\|c\|$. Training instances are generated on-the-fly (online generation), so that the model never encounters the same LP twice.

Training data generation. For each LP instance, we obtain a full trajectory $\{\bar{y}_i\}_{i=0}^{K-1}$ by running the Clarabel interior-point solver [69] with access to all intermediate iterates. To generate the trajectory diversity required by the flow matching objective (cf. Section 10.2.1), the solver is configured with a perturbation noise parameter $\sigma_\varepsilon = 0.1$, which introduces stochastic perturbations in the IPM iterations. Each solver call also receives a unique random seed, ensuring that repeated solves of structurally similar instances produce distinct trajectories.

Loss function. The training loss is a weighted sum of mean squared errors between predicted and target velocities for the three state components:

$$\mathcal{L} = 350 \cdot \mathcal{L}_{\text{primal}} + \mathcal{L}_{\text{dual}} + \mathcal{L}_{\text{slack}}, \tag{10.16}$$

where $\mathcal{L}_{\text{primal}} = \text{MSE}(\dot{x}_\theta, (K-1)(\bar{x}_{i+1} - \bar{x}_i))$, and similarly for the dual and slack components. The large weight on the primal loss reflects the fact that primal accuracy directly governs the quality of the predicted objective value $c^\top x$, which is the quantity of primary interest for approximate LP solving.

Training protocol. The model is trained for 500,000 gradient steps using AdamW [113] with $\beta_1 = 0.9$, $\beta_2 = 0.95$, a peak learning rate of 3×10^{-4}, and weight decay of 0.01 applied to weight matrices only (biases and normalization parameters are excluded). The learning rate follows a linear warmup over 5,000 steps followed by cosine decay to 1% of the peak value. Gradients are clipped to a maximum norm of 1.0. An exponential moving average (EMA) of the model parameters with decay 0.999 is

Table 10.1: Training and data generation hyperparameters for FlowIPM.

Module	Parameter	Value
Data generation	Instance family	Combinatorial Auction
	IPM solver	Clarabel
	Solver perturbation σ_ε	0.1
	Trajectory points K	500
Neural architecture	Hidden dimension D	256
	Message-passing blocks L	8
	RWPE dimension d_{pe}	16
	Decoder blocks (per head)	3
	JK aggregation	Element-wise max
Flow matching	Time sampling	Logit-normal (μ=0, σ_t=0.7)
	Samples per instance S	16
Optimization	Optimizer	AdamW (β_1=0.9, β_2=0.95)
	Peak learning rate	3×10^{-4}
	LR schedule	Warmup (5k steps) + cosine decay
	Min LR ratio	1%
	Weight decay	0.01
	Batch size	32
	Gradient clipping	Max norm 1.0
	EMA decay	0.999
	Total gradient steps	200×10^3
Loss weights	Primal / dual / slack	350 / 1 / 1
Evaluation	ODE solver	Dormand–Prince 5(4)
	ODE tolerances (atol, rtol)	10^{-5}

maintained throughout training and used for evaluation.

Time steps are sampled from a logit-normal distribution with location $\mu = 0$ and scale $\sigma_t = 0.7$, which concentrates samples around $t = 0.5$ where the velocity field exhibits the largest variation. For each LP instance, 16 time samples are drawn, yielding $S = 16$ training pairs per instance per gradient step. The full training hyperparameters are summarized in Table 10.1.

Baselines. We compare the IPM-based conditional flow (IPM) against the optimal-transport baseline (OT) described in Section 10.2.4, in which the conditional paths are straight-line interpolations from Gaussian noise to the optimal solution. The architecture, hyperparameters, and training protocol are identical between the two settings, so that any performance difference can be attributed to the choice of conditional path rather than to architectural or optimization confounds.

Evaluation protocol. At evaluation time, the learned vector field is integrated from $t = 0$ to $t = 1$ using the Dormand–Prince 5(4) adaptive solver [47] with absolute and relative tolerances of 10^{-5}. We report the following metrics, each averaged over held-out 128 LP instances unseen during training. The primary evaluation metric is relative objective gap,

$$\mathrm{ObjGap}(\hat{x}, x^\star) = \left| \frac{c^\top \hat{x} - c^\top x^\star}{c^\top x^\star} \right|,$$

assessing the quality of the predicted objective value, which is the quantity of direct practical relevance for approximate LP solving. To evaluate quality of the predicted solution, we also report root mean squared error,

$$\mathrm{RMSE}_x(\hat{x}, x^\star) = \|\hat{x} - x^\star\|_2, \qquad \mathrm{RMSE}_\lambda(\hat{\lambda}, \lambda^\star) = \|\hat{\lambda} - \lambda^\star\|_2,$$

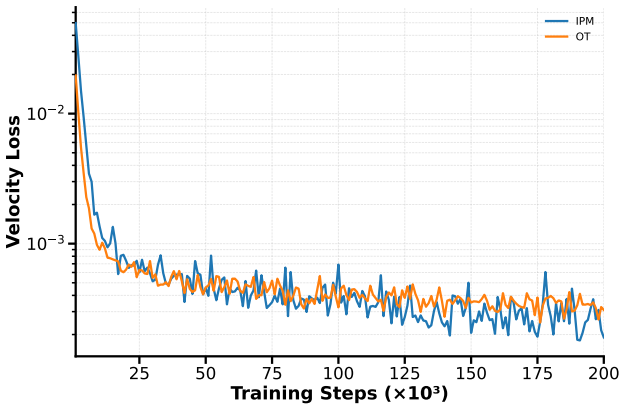
which measures how close the integrated solution is to the reference optimum. Beyond endpoint accuracy, the trajectory drift is evaluated by computing both primal and dual RMSE at equally-spaced checkpoints $t \in \{0.2, 0.6, 1.0\}$, which helps to determine where along the path the dynamics learned by each model begin to diverge from its reference trajectory. Finally, to quantify the warm-starting potential of our proposed approach, we measure the reduction in the number of IPM iterations required by an exact solver when initialized from $\hat{y}_T$ compared to a cold start, reported as a function of the integration horizon T. Evaluation is performed every 1000 training steps.

10.4.2 Computational results

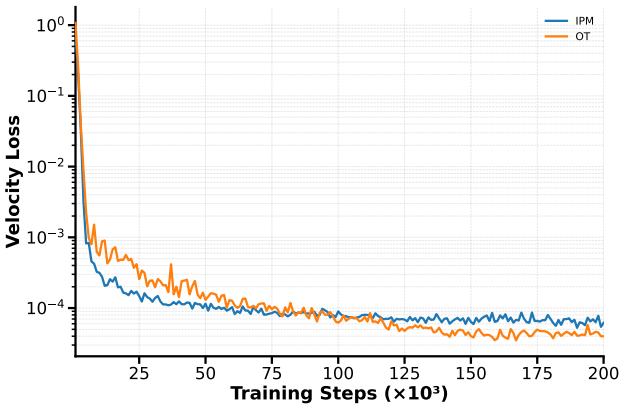
Computational results are summarized in Figures 10.1 and 10.2. All figures systematically compare the IPM and OT flow models. We organize the discussion around four aspects: velocity regression quality, objective gap accuracy, trajectory drift, and warm-starting performance.

Velocity regression. Figures 10.1a–10.1b report the validation loss for the primal and dual components of the learned velocity field v_θ. Both flow models exhibit comparable validation loss throughout training, indicating that the regression problem is globally of similar difficulty regardless of whether the conditional paths follow IPM trajectories or straight-line interpolations.

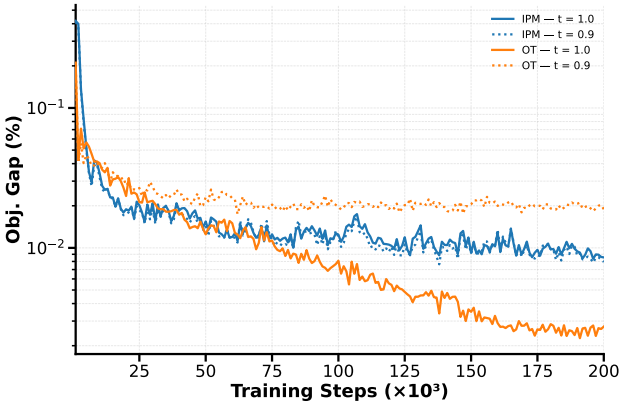
Objective gap. Figure 10.1c shows the evolution of the relative objective gap obtained by integrating v_θ from $t = 0$ to T, for $T \in \{0.9, 1.0\}$, as a function of the training step. Both models reach satisfactory error levels: the IPM model achieves a relative objective gap of approximately 0.8%, while the OT



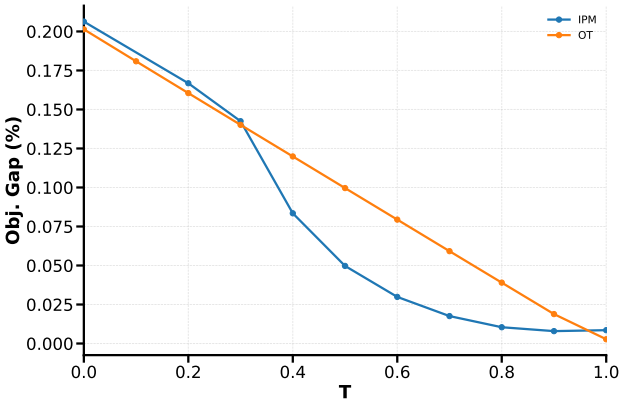
(a) Primal velocity validation loss.



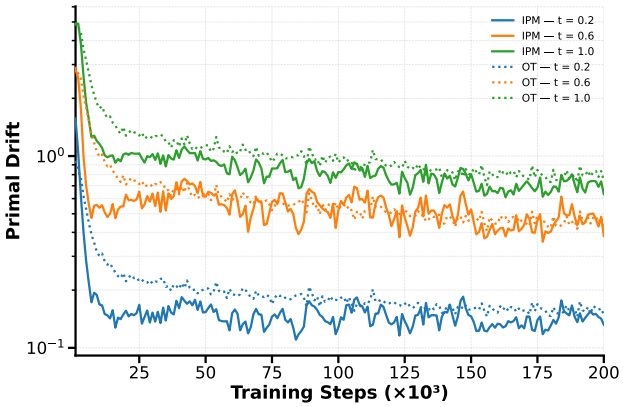
(b) Dual velocity validation loss.



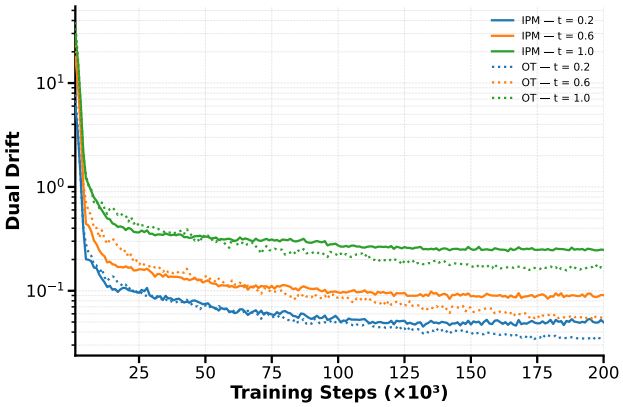
(c) Relative objective gap at $T = 0.9$ and $T = 1.0$.



(d) Relative objective gap as a function of T.



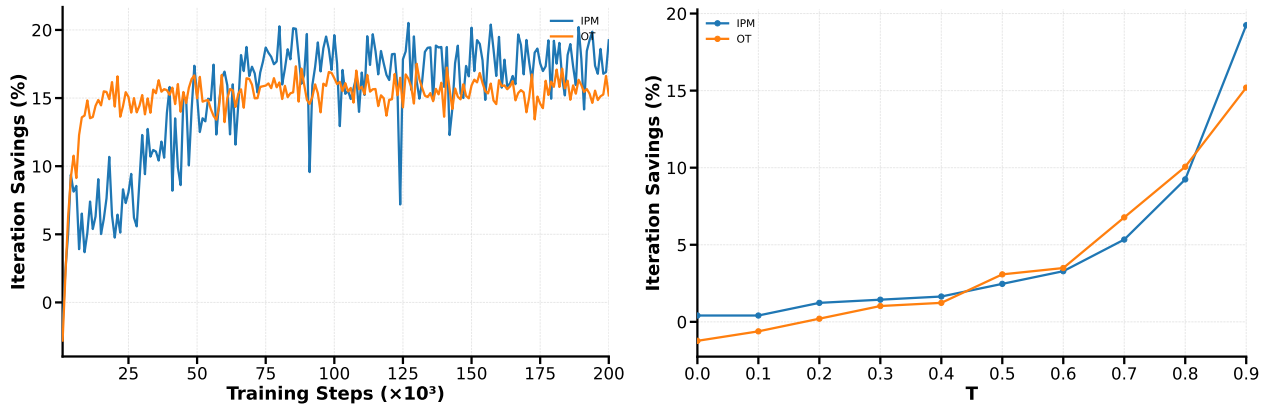
(e) Primal RMSE drift at $T \in \{0.2, 0.6, 1.0\}$.



(f) Dual RMSE drift at $T \in \{0.2, 0.6, 1.0\}$.

Figure 10.1: Training and evaluation curves for IPM and OT flow models on combinatorial auction LP relaxations. All metrics are computed on 512 held-out instances.

model reaches an impressive 0.02%. The gap between $T = 0.9$ and $T = 1.0$ reveals a notable difference between the two models: the OT model improves substantially over this last interval, whereas the



(a) IPM iterations saved by warm-starting at $T = 0.9$ throughout training.

(b) IPM iterations saved (%) as a function of T (final trained models).

Figure 10.2: Warm-starting performance of IPM and OT flow models. Iteration savings are measured relative to a cold-start baseline on 512 held-out combinatorial auction LP relaxations.

IPM model shows little change. This phenomenon is further illustrated in Figure 10.1d, which plots the relative objective gap of the final trained models as a function of $T \in \{0.0, 0.1, \ldots, 1.0\}$. The OT model reduces the objective gap approximately linearly in T, which is expected since its conditional paths are straight-line interpolations and the objective $c^\top x$ is linear in x. The IPM model, by contrast, reduces the gap more aggressively during the early and middle portions of the trajectory, consistent with the structure of IPM dynamics: the bulk of the objective improvement occurs during the first IPM iterations, when the barrier parameter τ is large, step sizes are generous, and the iterates move rapidly through the interior of the feasible region.

However, this early advantage does not translate into better terminal accuracy. We believe that a fundamental limitation lies in the arc-length reparametrisation introduced in Section 10.2.2. In the raw IPM trajectory, the early iterations produce large displacements in primal–dual space (aggressive Newton steps with generous step lengths), while the late iterations produce increasingly small displacements as the iterates approach the boundary of the positive orthant and step sizes must be reduced to maintain strict positivity. Arc-length reparametrisation assigns time proportionally to displacement, so the early, large-displacement phase occupies most of $[0, 1]$, while the terminal phase, where the objective gap drops from 10^{-3} to 10^{-8} through delicate corrections near the boundary, is compressed into a narrower interval near $t = 1$. Although the velocity targets in this terminal region have the same magnitude as everywhere else (by construction), they encode geometrically subtle adjustments: the

iterates must navigate close to the boundary of the positive orthant where the complementarity products $x_i s_i$ approach zero, and even small errors in the predicted velocity can push the trajectory out of the strictly feasible interior. Because this critical regime occupies so little of $[0, 1]$, it is undersampled during training (the logit-normal distribution concentrates samples around $t = 0.5$), and the network does not receive sufficient supervision to learn these fine corrections accurately. As a result, the IPM objective gap stagnates at a higher plateau than the OT model, which does not face this issue: its straight-line conditional paths distribute displacement uniformly across $[0, 1]$ by construction, and the terminal regime does not involve boundary-approach dynamics.

Addressing this limitation requires resolving a fundamental tension in the time reparametrisation. Replacing arc-length with iteration-count indexation would allocate more of $[0, 1]$ to the terminal regime, but at the cost of producing velocity targets with magnitudes spanning several orders of magnitude, which degrades the conditioning of the regression problem. Conversely, concentrating the time sampling distribution near $t = 1$ (e.g., via a beta distribution skewed toward 1) would increase terminal coverage at the expense of the early-trajectory region. More generally, arc-length reparametrisation favours regression stability (uniform-magnitude targets) but sacrifices terminal accuracy, while iteration-count indexation favours terminal coverage but destabilises the regression. Designing a reparametrisation that balances both requirements, or alternatively, adopting a curriculum that progressively shifts sampling density toward the terminal regime as training progresses, remains an open problem.

Trajectory drift. Figures 10.1e–10.1f report the primal and dual RMSE at integration checkpoints $T \in \{0.2, 0.6, 1.0\}$ throughout training. Despite following fundamentally different conditional paths, both the IPM and OT models exhibit comparable drift profiles. Crucially, both primal and dual drift increases by roughly an order of magnitude between $T = 0.2$ and $T = 1.0$, illustrating the effect of compounding errors: small velocity prediction errors accumulate through the ODE integration, progressively displacing the integrated trajectory from the reference path.

Warm-starting. Figure 10.2a reports the average percentage of IPM iterations saved by warm-starting the exact solver with y_T at $T = 0.9$, as a function of the training step. The IPM model provides a slight advantage over the OT model during the later stages of training. This is expected: since the IPM model is trained to reproduce IPM trajectories, its integrated iterates inherit the geometric properties

of central-path neighborhoods, making them more readily exploitable by an interior-point solver than the unstructured points produced by the OT flow. The warm-starting benefit is examined in greater detail in Figure 10.2b, which reports the fraction of iterations saved as a function of T for the final trained models. In both cases, the warm-starting gains are less pronounced than initially anticipated. This outcome is consistent with the trajectory drift documented above: since the integrated solutions y_T drift away from the reference IPM trajectory, the warm-start points supplied to the exact solver lie far from the central path. Before being passed to the solver, these points undergo the positivity restoration procedure described in Section 10.2.1, which ensures strict positivity but does not correct feasibility violations ($r_b = Ax_T - b$ and $r_c = A^\top \lambda_T + s_T - c$). Consequently, the exact solver must spend its initial iterations restoring feasibility rather than exploiting proximity to the optimal solution, largely negating the intended benefit of warm-starting. Reducing trajectory drift, whether through improved time reparameterization, feasibility-enforcing architectural constraints, or explicit projection layers, appears necessary to unlock the full warm-starting potential of our proposed approach.

10.5 Towards learning proximal point dynamics

The computational results of the previous section suggest that, while the Flow-IPM framework produces approximate LP solutions of good quality, its warm-starting performance is hampered by trajectory drift and the geometric difficulty of producing iterates well-centered near the central path. Although improved time reparameterization and feasibility enforcement could mitigate these issues, a more fundamental remedy is to target a solver class for which warm-starting does not require centrality in the first place.

Indeed, a well-known limitation of interior-point methods is that they are notoriously difficult to warm-start [59, 185]. The fundamental obstruction is geometric: IPM iterates must remain in the strict interior of the feasible region, and a good warm start must simultaneously be close to the optimal solution and well-centered with respect to the central path. These two requirements are in tension, since near-optimal points necessarily approach the boundary of the positive orthant (by the complementarity equation (9.5)), which degrades centrality and leads to tiny step sizes. This tension is precisely what makes the terminal regime $t \to 1$ problematic in our framework: the learned flow produces a point that is near-optimal but potentially near the boundary, which is the regime where IPM warm-starting is most fragile.

An alternative class of solvers avoids this problem entirely. Proximal point methods solve optimization problems through iterative application of proximal operators, rather than by tracing a path through the interior of the feasible region. The alternating direction method of multipliers (ADMM) [128], one of the most widely used practical instantiations of proximal points, has become a competitive solver for large-scale LPs. The core idea is to decompose the feasible set into two simpler pieces, the affine subspace $\{x : Ax = b\}$ on the one hand, and the nonnegative orthant $\{x \geq 0\}$ on the other, and maintain a separate copy of the current solution iterate for each. At every iteration, the first copy is projected onto the affine subspace (a linear solve), the second is projected onto the nonnegative orthant (a component-wise clipping), and a dual variable accumulates the disagreement between the two copies, steering future iterates toward consensus. The optimal solution emerges as the fixed point where both copies agree and the accumulated disagreement stabilizes.

Crucially, the method operates without barrier functions or centrality requirements: its iterates $y = (x, z, u)$ live in an unconstrained space, with feasibility enforced through projections within each iteration rather than through a barrier that restricts the domain.[3] Convergence to a fixed point is monotone, and warm-starting is straightforward: initializing closer to the fixed point directly reduces the number of iterations required, with no geometric obstruction analogous to the IPM centrality trap.

Adapting the flow matching framework of this chapter to learn ADMM dynamics requires only minor modifications. The perturbation procedure simplifies, since there is no strict feasibility requirement on the starting point: Gaussian perturbations can be applied directly without projection or clamping. The velocity field is better behaved, as ADMM iterates converge smoothly to a fixed point rather than approaching a boundary singularity, removing the need for terminal-time truncation. The model architecture requires only minimal changes, the main adaptation being the state features provided to each node, which reflect ADMM's internal variables rather than the IPM-specific quantities. The potential payoff is appealing: ADMM typically requires hundreds or thousands of inexpensive iterations where an IPM would require tens of costly ones, so that skipping a large fraction of the trajectory through a learned warm start could translate into substantial wall-clock savings.

In summary, the proximal setting removes the main practical limitation of the IPM-based approach, namely the geometric fragility of warm-starting near the boundary, while preserving the core

[3]Importantly, x denotes the primal variable, z a consensus variable enforcing nonnegativity, and u the scaled dual encoding the running disagreement.

methodological contribution: learning solver dynamics via flow matching regression. We note that several proximal point instantiations beyond ADMM could serve as target dynamical system; chief among them are primal-dual hybrid gradient methods (PDHG), whose matrix-free iterations and lower-dimensional state space may further simplify the learning problem. We leave the identification of which variant provides the most suitable framework for flow matching regression to future work.

10.6 Conclusion

This chapter introduced Flow-IPM, a framework leveraging flow matching techniques to learn the continuous-time dynamics of interior-point solvers for linear programming. A graph neural network regresses a time-dependent velocity field against empirical velocities extracted from perturbed solver trajectories, and integrates this field at inference time to produce approximate primal–dual solutions that can serve as fast LP estimates or warm-start an exact solver. To our knowledge, this is the first application of flow matching to approximate optimization solver dynamics. Our experimental study partly validates the rationale for this approach: both IPM and optimal transport flow models achieve relative objective gaps under 1% on small combinatorial auction LP relaxations, suggesting that GNN-based velocity fields can learn to approximate LP solutions with reasonable accuracy. However, at the same time, the results expose a clear limitation specific to the IPM-based flow: the arc-length reparameterization compresses the terminal regime of IPM trajectories, where iterates cautiously approach the boundary of the positive orthant, into a narrow interval near $t = 1$ that is undersampled during training. The OT baseline consequently achieves better terminal accuracy despite lacking any solver supervision. This outcome highlights a structural mismatch between IPM convergence geometry and the uniform-magnitude velocity targets produced by arc-length indexation.

These findings must, however, be interpreted within the limits of our experimental setting: the LP instances considered here remain small (50–150 variables), and production solvers resolve them in milliseconds at most. At this scale, the neural network inference cost alone exceeds the solver time it aims to save, and warm-starting iteration savings, while measurable, do not translate into wall-clock gains. The practical value of learned LP surrogates, whether for warm-starting or for approximate dual bound estimation within branch-and-bound solvers, can only be assessed on larger instances where LP resolution constitutes a genuine computational bottleneck.

The question of whether these iteration savings would translate into wall-clock gains at larger scale is not merely hypothetical: a growing body of work in computational physics suggests that neural warm-starting can deliver substantial gains precisely when applied to large-scale iterative solvers. Eshaghi et al. [53] report iteration count reductions of up to tenfold and runtime savings of 25–90% by using neural operators to provide high-quality initial guesses for iterative partial derivative equations (PDE) solvers, with the largest gains observed on well-conditioned problems with physics-informed initialization. Zhou et al. [190] achieve at least twofold wall-clock acceleration of steady-state CFD simulations through neural operator-based initialization. In a complementary direction, Zhang et al. [188] show that interleaving neural operator predictions with classical relaxation methods yields a hybrid solver whose convergence rate is uniform across the frequency spectrum, substantially outperforming either component alone. These results share a common principle: a learned model that reduces the initial residual presented to an iterative solver translates directly into fewer iterations and, at sufficient problem scale, into wall-clock savings. Two caveats temper the analogy with our setting. First, the benchmarks in these works remain predominantly academic, and the transition to industrially realistic problem scales is itself an open challenge in the neural PDE community [1]. Second, warm-starting interior-point methods is structurally harder than warm-starting linear system solvers: as discussed in Section 10.5, near-optimal primal–dual iterates lie close to the boundary of the positive orthant, degrading centrality and limiting step sizes, a geometric fragility that has no counterpart in the PDE setting. Nevertheless, these results suggest that scaling Flow-IPM to industrially relevant LP dimensions, where each IPM iteration involves solving a linear system at cost $O(n^2m + nm^2)$, is where the approach is most likely to demonstrate its value.

More broadly, this chapter represents an exploratory contribution in a largely uncharted research direction: learning continuous-time surrogates of optimization solvers via flow matching. The most actionable path forward is the combination of flow matching with proximal solver dynamics, which sidesteps the geometric fragility of IPM warm-starting while preserving the core methodological contribution of learning solver dynamics from trajectory data. The OT flow model developed and validated in this chapter directly applies to this setting, since ADMM iterates live in an unconstrained space where a warm-start point needs only to be close to the fixed point to reduce the iteration count, with no additional feasibility or centrality conditions. The superior terminal accuracy achieved by the OT baseline, which was a limitation in the IPM context, becomes a pure advantage in the proximal setting.

General conclusion

This thesis has sought to accelerate the solution of repeated combinatorial optimization problems by replacing expert-crafted heuristics with learned counterparts that integrate, rather than bypass, into the algorithmic framework upon which modern solvers are built. Two guiding principles, articulated in our general introduction, have underpinned this effort. The first is that existing frameworks produced by decades of research in combinatorial optimization incorporate rich inductive biases which, we argue, are worth preserving rather than discarding. The second is that the empirically tuned heuristics governing local decisions within these frameworks are natural candidates for replacement by learned components, which can exploit distributional regularities that static rules, by definition, cannot capture. In closing this manuscript, we reflect on the broader conceptual trajectory that emerges from the contributions presented in this thesis, and situate it within the ongoing debate, crystallized in Sutton's bitter lesson, between the integration of domain theory and the exploitation of computational scale as competing drivers of progress.

A parallel with branching rule design. In hindsight, the path followed in this thesis mirrors, in condensed form, a fundamental tension that has shaped the design of variable selection strategies in branch-and-bound solvers. As discussed in Section 2.1.1, LP evaluations produce information that is essential for guiding the search, yet they constitute the dominant source of computational cost within the solver. The branching rules presented in Section 2.1.3 can be read as successive resolutions of this tension. Strong branching occupies one extreme, solving auxiliary LP relaxations for every branching candidate to obtain exact dual bound improvements, many of which provide little additional value to the search. Pseudo-cost branching occupies the other, replacing explicit LP evaluations with online surrogate estimates built from historical observations. However, these estimates are unreliable near the root, where poor branching decisions are most damaging as they inflate the B&B tree considerably. Reliability branching reconciles the two, selectively applying strong branching evaluation on under-

explored variables before switching to pseudo-cost predictions once sufficient observations have been collected. Each successive strategy seeks to extract more value from the informative signal produced by LP evaluations, while reducing the computational cost of obtaining it.

The contributions of this thesis follow a closely parallel trajectory. Part I explores the design of a generalized strong branching oracle that trades additional LP solves for a stronger supervisory signal. This logic mirrors strong branching itself, which trades additional computational cost for better branching decisions. The difference is that in our proposed framework, the additional solves occur offline, during data generation, rather than online, during the search. Part II departs from this oracle by training model-free RL agents that directly target B&B tree size minimization, learning value estimates from direct interaction with the environment rather than from expert demonstrations. The analogy with pseudo-cost branching is direct: both replace exact evaluations with surrogate scores refined through accumulated experience. However, where pseudo-costs are updated online during the search, our value-based agents are trained offline on collected trajectories. Part III introduces model-based planning to mitigate the insufficient exploration issues identified in Part II, leveraging learned dynamics and look-ahead search to derive stronger policy targets where the agent's own estimates are least reliable. This echoes the logic of reliability branching, which invests in exact LP solves precisely where its own surrogate estimates are least trustworthy. The interim conclusion then identifies the next logical step: extending the agent's action space beyond variable selection to encompass the selective LP evaluations that solver heuristics perform alongside branching. The motivation stems from symmetric limitations: reliability branching exploits information gathered online during the search but fails to transfer collected experience across instances, while PlanB&B leverages offline training but discards the online information generated by MCTS simulations after each branching decision. ProbBBMDP bridges this gap by subsuming the full logic of reliability branching within a single learned policy: the agent decides both which candidates to probe and which variable to branch on, thus endowing learned agents with the online long-term planning capacity that was previously exclusive to hard-coded expert heuristics. Training model-based reinforcement learning agents in this extended action space could yield significant performance gains, as the agent would learn to allocate computational effort where it matters most. Since online computational efficiency is critical, it may further prove natural to equip such agents with two distinct evaluation regimes: exact LP solves to generate reliable information for long-term planning, and learned approximate evaluations to guide immediate branching decisions at a

fraction of the cost. Part IV takes this reasoning to its natural conclusion by targeting LP resolution itself as the object of learning, asking whether the very solver dynamics that underlie LP evaluation can be approximated by a learned model.

The literature on learning to branch, despite operating within the branch-and-bound framework, has to some extent overlooked the design principles embedded in classical branching rules, as it ruled out auxiliary LP solves from the action space of learned policies. The implicit view seems to be that dedicating additional computational resources to online LP evaluation becomes unnecessary (if not archaic) once learned policies can be leveraged to exploit the distributional regularities that arise from solving repeated instances. At least, this was certainly the author's own conviction at the outset of this thesis. Yet, the persistent difficulty of translating learned branching heuristics into reduced solving times on industrially relevant instances suggests that this assumption deserves to be revisited. In fact, we argue that this view overlooks a crucial asymmetry: over decades, MILP solvers have developed sophisticated machinery that systematically exploits LP relaxation solutions to inform the search **well beyond branching decisions**. By excluding auxiliary LP solves from the action space, learned policies forfeit access to this machinery entirely. While this trade-off can prove effective on smaller instances, our experiments suggest that exploiting recurring problem structure alone can not suffice to produce superior branching strategies on higher-dimensional real-world problems. We therefore believe that frameworks such as ProbBBMDP, which unlock access to the solver's full capacity while exploiting distributional regularities, represent the most promising path forward for accelerating the solution of repeated mixed-integer linear programs through access to large-scale offline computation.

What about the bitter lesson? The progression just described reveals a pattern that runs counter to our initial intuition. Having pursued reinforcement learning precisely to overcome the limitations of expert imitation, we arrive at the conclusion that the path forward may not lie in replacing expert knowledge, but in engaging with it more deeply. This trajectory finds a remarkable echo in the recent history of competitive game engines. In 2017, AlphaZero [154] demonstrated that a pure self-play RL agent, trained without human knowledge beyond the rules, could defeat the strongest classical chess engine of the time. Its successor MuZero [152] went further, dispensing with knowledge of the rules themselves by learning an internal dynamics model from interaction alone. Together, these results were widely interpreted as a confirmation of the bitter lesson in its purest form: general-purpose search and learning, given sufficient computation, could surpass decades of accumulated expert knowl-

edge. Yet, a different paradigm ultimately prevailed in practice. In 2020, Stockfish[159], the leading open-source chess engine, integrated NNUE [127], a neural network evaluation function originally developed for shogi. Stockfish is built on alpha-beta search [97], a tree search algorithm that, much like branch-and-bound, systematically explores a combinatorial space while pruning subtrees that provably cannot improve the current best solution. Crucially, Stockfish preserved this framework, with all its expert-designed pruning, reduction, and extension heuristics, intact. Only the evaluation function, the heuristic that scores board positions to guide the search, was replaced by a learned counterpart, trained not by self-play but by supervised distillation: the network learned to replicate evaluations produced by Stockfish's own deep search, compressing expert-level knowledge into a fast neural surrogate. Since this transition, Stockfish has decisively surpassed all pure RL engines, winning every major computer chess championship since 2020. Notably, the situation in Go is different: no classical framework ever achieved the level of competitiveness that alpha-beta search reached in chess, and RL-based engines such as KataGo [180] remain dominant. This contrast suggests that the hybrid paradigm tends to prevail precisely when a strong expert-designed framework exists to serve as its scaffold.

Because industrial mixed-integer linear programs routinely involve hundreds of thousands of variables and produce search trees whose scales far exceed those encountered in board games, we initially expected the lessons of Go, where RL agents remain unmatched, to carry over to mixed-integer linear programming. Yet branch-and-bound solvers, unlike Go engines before AlphaZero, already encode decades of domain knowledge into highly effective algorithmic machinery. In this regard, MILP may have more in common with chess than with Go, and the trajectory of Stockfish, from handcrafted heuristics to learned evaluation within a preserved expert framework, may well foreshadow the future of mixed-integer linear programming solvers. The resulting architecture could mirror Stockfish's design: reliability branching as the preserved expert framework, a learned ProbBBMDP controlling variable selection and LP evaluation allocation, and a surrogate model playing the role of NNUE, providing fast approximate LP scores where exact solves are not warranted. The branch-and-bound machinery, including pruning, propagation, and cut generation, would not only remain intact but be fully leveraged through the bound information generated by the agent's LP evaluation decisions; only the heuristic choices that govern the search would be learned.

Returning to the broader debate announced in our opening paragraph, the contributions of this thesis, together with evidence from game engines, suggest that the tension crystallized in the bitter

lesson offers the right lens through which to examine the future of machine learning for combinatorial optimization. Where exactly to draw the line between the structural knowledge that should be preserved and the heuristics that should be learned from data is not a question that this thesis, or any single body of work, can settle definitively. But the progression from Part I to Part III suggests that previous contributions on learning to branch may have drawn this line at the wrong level of abstraction: the natural target for learned policies is not the branching decision alone, but the joint control of LP evaluation and variable selection that modern solvers already perform through expert handcrafted rules. By excluding auxiliary LP solves from the learned agent's action space, prior work on learning to branch effectively forfeited the very inductive bias upon which efficient branch-and-bound rests: that solving LP relaxations yields the bound information needed to prune the search tree. Restoring this access is, we believe, a prerequisite for learned policies to compete with solver heuristics on industrially relevant instances.

If nothing else, we hope to have shown that this question is worth asking precisely, for the answer wherever it ultimately falls, will shape the design of the next generation of exact solvers.

From discrete to continuous optimization. Finally, the progression of this thesis also entails a less obvious but consequential transition: from discrete to continuous optimization, and accordingly from classification to regression as the dominant learning paradigm. In Parts I–III, the learning problem is naturally discrete: the agent selects a branching variable from a finite candidate set, a decision that reduces to classification over structured inputs and that neural networks handle effectively. In Part IV, the learning problem becomes continuous: the model must regress a time-dependent velocity field against empirical vectors extracted from solver trajectories, producing primal–dual iterates in $\mathbb{R}^{2n+m}$ that must satisfy demanding numerical tolerances to be useful for warm-starting. Regression problems are generally harder for deep learning than their classification counterparts, a phenomenon that has motivated a growing body of work advocating distributional or classification-based value prediction even in reinforcement learning [57, 87]. Beyond the classification–regression distinction, continuous optimization solvers impose numerical precision requirements that are fundamentally in tension with the approximate nature of neural computation: as interior-point iterates converge toward the boundary of the positive orthant, step sizes must be precisely controlled to maintain feasibility, a regime where

the precision achievable with standard floating-point networks falls short of what standard LP solvers deliver. While proximal methods sidestep this fragility by requiring only proximity to the fixed point with no additional geometric constraints, the broader challenge of integrating approximate neural models into precision-critical numerical pipelines remains an open problem that extends well beyond the scope of this thesis.

Bibliography

[1] Machine learning solutions looking for PDE problems. *Nature Machine Intelligence*, 7:1, 2025. doi:doi.org/10.1038/s42256-025-00989-w.

[2] T. Achterberg. *Constraint integer programming.* PhD Thesis, 2007.

[3] T. Achterberg and T. Berthold. Hybrid Branching. In W.-J. van Hoeve and J. N. Hooker, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, Lecture Notes in Computer Science, pages 309–311, Berlin, Heidelberg, 2009. Springer. ISBN 978-3-642-01929-6. doi:10.1007/978-3-642-01929-6_23.

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

[5] T. Achterberg, T. Koch, and A. Martin. Branching rules revisited. *Operations Research Letters*, 33(1):42–54, Jan. 2005. ISSN 0167-6377. doi:10.1016/j.orl.2004.04.002.

[6] A. M. Alvarez, Q. Louveaux, and L. Wehenkel. A supervised machine learning approach to variable branching in branch-and-bound. Technical report, Université de Liège, Liège, Belgium, 2014.

[7] A. M. Alvarez, Q. Louveaux, and L. Wehenkel. A Machine Learning-Based Approximation of Strong Branching. *INFORMS Journal on Computing*, 29(1):185–195, Jan. 2017. ISSN 1091-9856. doi:10.1287/ijoc.2016.0723. Publisher: INFORMS.

[8] K. Andersen, G. Cornuéjols, and Y. Li. Reduce-and-Split Cuts: Improving the Performance of Mixed-Integer Gomory Cuts. *Management Science*, 51(11):1720–1732, Nov. 2005. ISSN 0025-

1909, 1526-5501. doi:10.1287/mnsc.1050.0382. Publisher: Institute for Operations Research and the Management Sciences (INFORMS).

[9] V. Andréassian, O. Delaigue, C. Perrin, B. Janet, and N. Addor. CAMELS-FR: A large sample, hydroclimatic dataset for france, to support model testing and evaluation, 2021.

[10] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. *Finding cuts in the TSP (A preliminary report)*, volume 95. Citeseer, 1995.

[11] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, 47(2):235–256, May 2002. ISSN 1573-0565. doi:10.1023/A:1013689704352.

[12] E. Balas. Intersection Cuts—A New Type of Cutting Planes for Integer Programming. *Operations Research*, 19(1):19–39, Feb. 1971. ISSN 0030-364X, 1526-5463. doi:10.1287/opre.19.1.19. URL .

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

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

[15] M.-F. Balcan, T. Dick, T. Sandholm, and E. Vitercik. Learning to branch. In J. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 344–353. PMLR, 10–15 Jul 2018.

[16] M.-F. Balcan, T. Dick, T. Sandholm, and E. Vitercik. Learning to Branch: Generalization Guarantees and Limits of Data-Independent Discretization. *Journal of the ACM*, 71(2):1–73, Apr. 2024. ISSN 0004-5411, 1557-735X. doi:10.1145/3637840. URL .

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

[18] M.-F. F. Balcan, S. Prasad, T. Sandholm, and E. Vitercik. Structural analysis of branch-and-cut and the learnability of gomory mixed integer cuts. *Advances in Neural Information Processing Systems*, 35:33890–33903, 2022.

[19] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. Savelsbergh, and P. H. Vance. Branch-and-price: Column generation for solving huge integer programs. *Operations research*, 46(3): 316–329, 1998.

[20] M. G. Bellemare, W. Dabney, and R. Munos. A distributional perspective on reinforcement learning. In *International conference on machine learning*, pages 449–458. PMLR, 2017.

[21] Y. Bengio, A. Lodi, and A. Prouvost. Machine learning for combinatorial optimization: A methodological tour d'horizon. *European Journal of Operational Research*, 290(2):405–421, Apr. 2021. ISSN 0377-2217. doi:10.1016/j.ejor.2020.07.063.

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

[23] F. Berto, C. Hua, J. Park, M. Kim, H. Kim, J. Son, H. Kim, J. Kim, and J. Park. RL4CO: an Extensive Reinforcement Learning for Combinatorial Optimization Benchmark, June 2023. arXiv:2306.17100 [cs].

[24] D. P. Bertsekas and J. N. Tsitsiklis. *Neuro-dynamic programming*. Athena Scientific, 1996.

[25] D. Bertsimas and G. Margaritis. Global Optimization: A Machine Learning Approach, Nov. 2023. URL . arXiv:2311.01742 [cs, math].

[26] T. R. Besold, S. Bader, H. Bowman, P. Domingos, P. Hitzler, K.-U. Kühnberger, L. C. Lamb, P. M. V. Lima, L. de Penning, G. Pinkas, et al. Neural-symbolic learning and reasoning: A survey and interpretation 1. In *Neuro-symbolic artificial intelligence: The state of the art*, pages 1–51. IOS press, 2021.

[27] K. Bestuzheva, M. Besançon, W.-K. Chen, A. Chmiela, T. Donkiewicz, J. v. Doornmalen, L. Eifler, O. Gaul, G. Gamrath, A. Gleixner, L. Gottwald, C. Graczyk, K. Halbig, A. Hoen, C. Hojny, R. v. d. Hulst, T. Koch, M. Lübbecke, S. J. Maher, F. Matter, E. Mühmer, B. Müller, M. E. Pfetsch, D. Rehfeldt, S. Schlein, F. Schlösser, F. Serrano, Y. Shinano, B. Sofranac, M. Turner,

S. Vigerske, F. Wegscheider, P. Wellner, D. Weninger, and J. Witzig. The SCIP Optimization Suite 8.0. Technical Report, Optimization Online, Dec. 2021.

[28] R. E. Bixby. A brief history of linear and mixed-integer programming computation. *Documenta Mathematica*, 2012:107–121, 2012.

[29] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[30] M. Bénichou, J.-M. Gauthier, P. Girodet, G. Hentges, G. Ribière, and O. Vincent. Experiments in mixed-integer linear programming. *Mathematical Programming*, 1:76–94, 1971. Publisher: Springer.

[31] T. Cazenave, J.-Y. Lucas, H. Kim, and T. Triboulet. Monte carlo vehicle routing. In *ATT at ECAI 2020*, 2020.

[32] F. Chalumeau, S. Surana, C. Bonnet, N. Grinsztajn, A. Pretorius, A. Laterre, and T. Barrett. Combinatorial optimization with policy adaptation using latent space search. *Advances in Neural Information Processing Systems*, 36:7947–7959, 2023.

[33] R. T. Chen, Y. Rubanova, J. Bettencourt, and D. K. Duvenaud. Neural ordinary differential equations. *Advances in neural information processing systems*, 31, 2018.

[34] X. Chen and K. He. Exploring Simple Siamese Representation Learning, Nov. 2020. arXiv:2011.10566 [cs].

[35] Z. Chen, J. Liu, X. Wang, J. Lu, and W. Yin. On Representing Mixed-Integer Linear Programs by Graph Neural Networks, Oct. 2022. arXiv:2210.10759 [cs, math].

[36] Z. Chen, J. Liu, X. Chen, X. Wang, and W. Yin. Rethinking the Capacity of Graph Neural Networks for Branching Strategy, Feb. 2024. arXiv:2402.07099 [cs, math].

[37] A. Chmiela, A. Gleixner, P. Lichocki, and S. Pokutta. Online learning for scheduling mip heuristics. In A. A. Cire, editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 114–123, Cham, 2023. Springer Nature Switzerland. ISBN 978-3-031-33271-5.

[38] A. Choromanska, M. Henaff, M. Mathieu, G. B. Arous, and Y. LeCun. The loss surfaces of multilayer networks. In *Artificial intelligence and statistics*, pages 192–204. PMLR, 2015.

[39] G. Cornuéjols. Valid inequalities for mixed integer linear programs. *Mathematical programming*, 112(1):3–44, 2008.

[40] G. Cornuéjols, L. Liberti, and G. Nannicini. Improved strategies for branching on general disjunctions. *Mathematical Programming*, 130(2):225–247, Dec. 2011. ISSN 0025-5610, 1436-4646. doi:10.1007/s10107-009-0333-2.

[41] W. Dabney, G. Ostrovski, D. Silver, and R. Munos. Implicit quantile networks for distributional reinforcement learning. In *International conference on machine learning*, pages 1096–1105. PMLR, 2018.

[42] I. Danihelka, A. Guez, J. Schrittwieser, and D. Silver. Policy improvement by planning with gumbel. In *International Conference on Learning Representations*, 2022.

[43] G. B. Dantzig. Maximization of a linear function of variables subject to linear inequalities. *Activity analysis of production and allocation*, 13(1):339–347, 1951.

[44] G. B. Dantzig and P. Wolfe. Decomposition principle for linear programs. *Operations research*, 8(1):101–111, 1960.

[45] P.-E. Delon. *Perflex II, Placement de l'énergie : modélisation générique d'une vallée (version 1.2)*, 2013.

[46] S. S. Dey, Y. Dubey, M. Molinaro, and P. Shah. A Theoretical and Computational Analysis of Full Strong-Branching, Nov. 2021. arXiv:2110.10754 [math].

[47] J. R. Dormand and P. J. Prince. A family of embedded runge-kutta formulae. *Journal of computational and applied mathematics*, 6(1):19–26, 1980.

[48] D. Drakulic, S. Michel, and J.-M. Andreoli. GOAL: A Generalist Combinatorial Optimization Agent Learner, June 2024. arXiv:2406.15079 [cs] version: 1.

[49] V. P. Dwivedi, A. T. Luu, T. Laurent, Y. Bengio, and X. Bresson. Graph neural networks with learnable structural and positional representations. *arXiv:2110.07875*, 2021.

[50] EDF. L'hydraulique en chiffres, 2022.

[51] S. Elfwing, E. Uchibe, and K. Doya. Sigmoid-weighted linear units for neural network function approximation in reinforcement learning. *Neural networks*, 107:3–11, 2018.

[52] G. M. Energetici. Historical data day ahead market, 2022.

[53] M. S. Eshaghi, C. Anitescu, N. Valizadeh, Y. Wang, X. Zhuang, and T. Rabczuk. Nows: Neural operator warm starts for accelerating iterative solvers. *arXiv:2511.02481*, 2025.

[54] P. Esser, S. Kulal, A. Blattmann, R. Entezari, J. Müller, H. Saini, Y. Levi, D. Lorenz, A. Sauer, F. Boesel, et al. Scaling rectified flow transformers for high-resolution image synthesis. In *Forty-first international conference on machine learning*, 2024.

[55] M. Etheve. *Solving repeated optimization problems by Machine Learning.* PhD thesis, Paris, HESAM, 2021.

[56] M. Etheve, Z. Alès, C. Bissuel, O. Juan, and S. Kedad-Sidhoum. Reinforcement Learning for Variable Selection in a Branch and Bound Algorithm. In E. Hebrard and N. Musliu, editors, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, Lecture Notes in Computer Science, pages 176–185, Cham, 2020. Springer International Publishing. ISBN 978-3-030-58942-4. doi:10.1007/978-3-030-58942-4_12.

[57] J. Farebrother, J. Orbay, Q. Vuong, A. A. Taïga, Y. Chebotar, T. Xiao, A. Irpan, S. Levine, P. S. Castro, A. Faust, A. Kumar, and R. Agarwal. Stop Regressing: Training Value Functions via Classification for Scalable Deep RL, Mar. 2024. arXiv:2403.03950 [cs, stat].

[58] S. Feng, W. Sun, S. Li, A. Talwalkar, and Y. Yang. A Comprehensive Evaluation of Contemporary ML-Based Solvers for Combinatorial Optimization, May 2025.

[59] A. Forsgren. On warm starts for interior methods. In *IFIP Conference on System Modeling and Optimization*, pages 51–66. Springer, 2005.

[60] A. S. Fukunaga. A branch-and-bound algorithm for hard multiple knapsack problems. *Annals of Operations Research*, 184(1):97–119, 2011.

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

[62] M. Gasse, D. Chetelat, N. Ferroni, L. Charlin, and A. Lodi. Exact Combinatorial Optimization with Graph Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[63] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural message passing for quantum chemistry. In *International conference on machine learning*, pages 1263–1272. Pmlr, 2017.

[64] P. C. Gilmore and R. E. Gomory. A linear programming approach to the cutting-stock problem. *Operations research*, 9(6):849–859, 1961.

[65] A. Gleixner, G. Hendel, G. Gamrath, T. Achterberg, M. Bastubbe, T. Berthold, P. Christophel, K. Jarck, T. Koch, and J. Linderoth. MIPLIB 2017: data-driven compilation of the 6th mixed-integer programming library. *Mathematical Programming Computation*, 13(3):443–490, 2021. Publisher: Springer.

[66] R. Gomory. An algorithm for the mixed integer problem. Technical report, 1960.

[67] R. E. Gomory. Outline of an algorithm for integer solutions to linear programs. 1958.

[68] J. Gondzio. Interior point methods 25 years later. *European Journal of Operational Research*, 218(3):587–601, 2012.

[69] P. J. Goulart and Y. Chen. Clarabel: An interior-point solver for conic programs with quadratic objectives, May 2024. arXiv:2405.12762 [math].

[70] N. Grinsztajn, D. Furelos-Blanco, and T. D. Barrett. Population-Based Reinforcement Learning for Combinatorial Optimization. Oct. 2022. doi:10.48550/arXiv.2210.03475.

[71] I. E. Grossmann. Review of nonlinear mixed-integer and disjunctive programming techniques. *Optimization and engineering*, 3(3):227–252, 2002.

[72] Z. D. Guo, S. Thakoor, M. Pîslar, B. A. Pires, F. Altché, C. Tallec, A. Saade, D. Calandriello, J.-B. Grill, Y. Tang, M. Valko, R. Munos, M. G. Azar, and B. Piot. BYOL-Explore: Exploration by Bootstrapped Prediction, June 2022. arXiv:2206.08332 [cs, stat].

[73] P. Gupta, E. B. Khalil, D. Chetélat, M. Gasse, Y. Bengio, A. Lodi, and M. P. Kumar. Lookback for Learning to Branch, June 2022. arXiv:2206.14987 [cs, math, stat].

[74] D. Gusfield. *Integer linear programming in computational and systems biology: an entry-level text and course*. Cambridge University Press, 2019.

[75] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. In *Proceedings of the 35th International Conference on Machine Learning*, pages 1861–1870. PMLR, July 2018. ISSN: 2640-3498.

[76] D. Hafner, J. Pasukonis, J. Ba, and T. Lillicrap. Mastering diverse control tasks through world models. *Nature*, 640(8059):647–653, 2025.

[77] N. Hansen, X. Wang, and H. Su. Temporal Difference Learning for Model Predictive Control, July 2022. arXiv:2203.04955 [cs].

[78] N. Hansen, H. Su, and X. Wang. TD-MPC2: Scalable, Robust World Models for Continuous Control, Mar. 2024. arXiv:2310.16828 [cs].

[79] H. He, H. Daume III, and J. M. Eisner. Learning to Search in Branch and Bound Algorithms. In *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014.

[80] J. He, T. M. Moerland, J. A. d. Vries, and F. A. Oliehoek. What model does MuZero learn?, Oct. 2024. arXiv:2306.00840 [cs].

[81] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver. Rainbow: Combining Improvements in Deep Reinforcement Learning, Oct. 2017. arXiv:1710.02298 [cs].

[82] M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-second AAAI conference on artificial intelligence*, 2018.

[83] F. S. Hillier and G. J. Lieberman. *Introduction to operations research*. McGraw-Hill, 2015.

[84] J. Ho, A. Jain, and P. Abbeel. Denoising diffusion probabilistic models. *Advances in neural information processing systems*, 33:6840–6851, 2020.

[85] K. Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4 (2):251–257, 1991.

[86] T. Huang, A. Ferber, Y. Tian, B. Dilkina, and B. Steiner. Searching Large Neighborhoods for Integer Linear Programs with Contrastive Learning, Feb. 2023. arXiv:2302.01578 [cs].

[87] E. Imani and M. White. Improving Regression Performance with Distributional Losses. In *Proceedings of the 35th International Conference on Machine Learning*, pages 2157–2166. PMLR, July 2018. ISSN: 2640-3498.

[88] M. Jaderberg, V. Mnih, W. M. Czarnecki, T. Schaul, J. Z. Leibo, D. Silver, and K. Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks. *arXiv:1611.05397*, 2016.

[89] D. Jayalath, G. Landau, B. Shillingford, M. Woolrich, and O. P. Jones. The brain's bitter lesson: Scaling speech decoding with self-supervised learning. *arXiv:2406.04328*, 2024.

[90] X. Jiang, Y. Wu, M. Li, Z. Cao, and Y. Zhang. Large language models as end-to-end combinatorial optimization solvers. *arXiv:2509.16865*, 2025.

[91] C. K. Joshi, Q. Cappart, L.-M. Rousseau, and T. Laurent. Learning the Travelling Salesperson Problem Requires Rethinking Generalization, May 2022. arXiv:2006.07054 [cs, stat].

[92] M. Karamanov and G. Cornuéjols. Branching on general disjunctions. *Mathematical Programming*, 128(1-2):403–436, June 2011. ISSN 0025-5610, 1436-4646. doi:10.1007/s10107-009-0332-3.

[93] Z. Karnin, T. Koren, and O. Somekh. Almost optimal exploration in multi-armed bandits. In *International conference on machine learning*, pages 1238–1246. PMLR, 2013.

[94] E. Khalil, P. L. Bodic, L. Song, G. Nemhauser, and B. Dilkina. Learning to Branch in Mixed Integer Programming. *Proceedings of the AAAI Conference on Artificial Intelligence*, 30(1), Feb. 2016. ISSN 2374-3468. doi:10.1609/aaai.v30i1.10080. Number: 1.

[95] M. Kim, J. Park, and J. Park. Sym-NCO: Leveraging Symmetricity for Neural Combinatorial Optimization. 2022.

[96] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv:1412.6980*, 2014.

[97] D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. *Artificial intelligence*, 6(4): 293–326, 1975.

[98] D. Kochkov, J. A. Smith, A. Alieva, Q. Wang, M. P. Brenner, and S. Hoyer. Machine learning–accelerated computational fluid dynamics. *Proceedings of the National Academy of Sciences*, 118 (21):e2101784118, 2021.

[99] L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[100] W. Kool, H. Van Hoof, and M. Welling. Stochastic beams and where to find them: The gumbel-top-k trick for sampling sequences without replacement. In *International Conference on Machine Learning*, pages 3499–3508. PMLR, 2019.

[101] W. Kool, H. van Hoof, and M. Welling. Attention, Learn to Solve Routing Problems!, Feb. 2019. arXiv:1803.08475 [cs, stat].

[102] Y.-D. Kwon, J. Choo, B. Kim, I. Yoon, Y. Gwon, and S. Min. POMO: Policy Optimization with Multiple Optima for Reinforcement Learning. In *Advances in Neural Information Processing Systems*, volume 33, pages 21188–21198. Curran Associates, Inc., 2020.

[103] A. G. Labassi, D. Chételat, and A. Lodi. Learning to Compare Nodes in Branch and Bound with Graph Neural Networks. *arXiv:2210.16934*, 2022.

[104] M. G. Lagoudakis and R. Parr. Least-squares policy iteration. *Journal of machine learning research*, 4(Dec):1107–1149, 2003.

[105] T. Lahire, M. Geist, and E. Rachelson. Large batch experience replay. *arXiv:2110.01528*, 2021.

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

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

[108] G. D. Liberto, S. Kadioglu, K. Leo, and Y. Malitsky. DASH: Dynamic Approach for Switching Heuristics. *European Journal of Operational Research*, 248(3):943–953, Feb. 2016. ISSN 0377-2217. doi:10.1016/j.ejor.2015.08.018.

[109] J. Lin, J. Zhu, H. Wang, and T. Zhang. Learning to branch with Tree-aware Branching Transformers. *Knowledge-Based Systems*, 252:109455, Sept. 2022. ISSN 0950-7051. doi:10.1016/j.knosys.2022.109455.

[110] L.-J. Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3):293–321, 1992.

[111] Y. Lipman, R. T. Q. Chen, H. Ben-Hamu, M. Nickel, and M. Le. Flow Matching for Generative Modeling, Feb. 2023. arXiv:2210.02747 [cs].

[112] H. Liu, Y. Kuang, J. Wang, X. Li, Y. Zhang, and F. Wu. Promoting Generalization for Exact Solvers via Adversarial Instance Augmentation, Oct. 2023. arXiv:2310.14161 [cs].

[113] I. Loshchilov and F. Hutter. Decoupled weight decay regularization. *arXiv:1711.05101*, 2017.

[114] F. Luo, X. Lin, F. Liu, Q. Zhang, and Z. Wang. Neural Combinatorial Optimization with Heavy Decoder: Toward Large Scale Generalization. *Advances in Neural Information Processing Systems*, 36:8845–8864, Dec. 2023.

[115] Y. Luo, L. Shi, and X.-M. Wu. Can classic gnns be strong baselines for graph-level tasks? simple architectures meet excellence. *arXiv:2502.09263*, 2025.

[116] C. J. Maddison, D. Tarlow, and T. Minka. A* sampling. *Advances in neural information processing systems*, 27, 2014.

[117] A. Mahajan and T. Ralphs. On the complexity of selecting disjunctions in integer programming. *SIAM Journal on Optimization*, 20(5):2181–2198, 2010.

[118] R. Mansini, W. Ogryczak, M. G. Speranza, and E. T. A. of European Operational Research Societies. *Linear and mixed integer programming for portfolio optimization*, volume 21. Springer, 2015.

[119] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv:1312.5602*, 2013.

[120] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing Atari with Deep Reinforcement Learning, Dec. 2013. arXiv:1312.5602 [cs].

[121] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, pages 561–577, 2018.

[122] R. Munos. Error bounds for approximate policy iteration. In *Proceedings of the Twentieth International Conference on International Conference on Machine Learning*, pages 560–567, 2003.

[123] R. Munos. Performance bounds in l_p-norm for approximate value iteration. *SIAM journal on control and optimization*, 46(2):541–561, 2007.

[124] R. Munos, T. Stepleton, A. Harutyunyan, and M. Bellemare. Safe and efficient off-policy reinforcement learning. *Advances in neural information processing systems*, 29, 2016.

[125] V. Nair, S. Bartunov, F. Gimeno, I. von Glehn, P. Lichocki, I. Lobov, B. O'Donoghue, N. Sonnerat, C. Tjandraatmadja, P. Wang, R. Addanki, T. Hapuarachchi, T. Keck, J. Keeling, P. Kohli, I. Ktena, Y. Li, O. Vinyals, and Y. Zwols. Solving Mixed Integer Programs Using Neural Networks, July 2021. arXiv:2012.13349 [cs, math].

[126] G. Nannicini, G. Cornuéjols, M. Karamanov, and L. Liberti. Branching on Split Disjunctions.

[127] Y. Nasu. Efficiently updatable neural-network-based evaluation functions for computer shogi. *The 28th World Computer Shogi Championship Appeal Document*, 185, 2018.

[128] P. Neal, C. Eric, P. Borja, and E. Jonathan. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine learning*, 3(1):1–122, 2011.

[129] J. Nocedal and S. J. Wright. *Numerical optimization*. Springer series in operations research. Springer, New York, 2nd ed edition, 2006. ISBN 978-0-387-30303-1.

[130] T. Osa, J. Pajarinen, G. Neumann, J. A. Bagnell, P. Abbeel, and J. Peters. An algorithmic perspective on imitation learning. *Foundations and Trends® in Robotics*, 7(1-2):1–179, 2018.

[131] M. Padberg and G. Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM review*, 33(1):60–100, 1991.

[132] C. W. F. Parsonson, A. Laterre, and T. D. Barrett. Reinforcement Learning for Branch-and-Bound Optimisation using Retrospective Trajectories, Dec. 2022. arXiv:2205.14345 [cs].

[133] M. B. Paulus and A. Krause. Learning To Dive In Branch And Bound, Jan. 2023. arXiv:2301.09943 [cs, math].

[134] M. B. Paulus, G. Zarpellon, A. Krause, L. Charlin, and C. Maddison. Learning to Cut by Looking Ahead: Cutting Plane Selection via Imitation Learning. In *Proceedings of the 39th International Conference on Machine Learning*, pages 17584–17600. PMLR, June 2022. ISSN: 2640-3498.

[135] E. Perez, F. Strub, H. De Vries, V. Dumoulin, and A. Courville. Film: Visual reasoning with a general conditioning layer. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

[136] T. Pfaff, M. Fortunato, A. Sanchez-Gonzalez, and P. Battaglia. Learning mesh-based simulation with graph networks. In *International conference on learning representations*, 2020.

[137] J. Pirnay and D. G. Grimm. Self-improvement for neural combinatorial optimization: Sample without replacement, but improvement. *arXiv:2403.15180*, 2024.

[138] L. Prechelt. Early stopping-but when? In *Neural Networks: Tricks of the trade*, pages 55–69. Springer, 1998.

[139] A. Prouvost, J. Dumouchelle, L. Scavuzzo, M. Gasse, D. Chételat, and A. Lodi. Ecole: A Gym-like Library for Machine Learning in Combinatorial Optimization Solvers, Nov. 2020. arXiv:2011.06069 [cs, math].

[140] M. L. Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[141] C. Qian, D. Chételat, and C. Morris. Exploring the power of graph neural networks in solving linear optimization problems. In *International conference on artificial intelligence and statistics*, pages 1432–1440. PMLR, 2024.

[142] S. Reed, K. Zolna, E. Parisotto, S. G. Colmenarejo, A. Novikov, G. Barth-Maron, M. Gimenez, Y. Sulsky, J. Kay, J. T. Springenberg, et al. A generalist agent. *arXiv:2205.06175*, 2022.

[143] J. R. Rice. The algorithm selection problem. In *Advances in computers*, volume 15, pages 65–118. Elsevier, 1976.

[144] H. Robbins and S. Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[145] C. D. Rosin. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence*, 61(3):203–230, 2011.

[146] S. Ross, G. Gordon, and D. Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 627–635. JMLR Workshop and Conference Proceedings, 2011.

[147] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[148] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, Harlow, England, 4 edition, 2020. ISBN 978-0134610993.

[149] L. Scavuzzo, F. Y. Chen, D. Chételat, M. Gasse, A. Lodi, N. Yorke-Smith, and K. Aardal. Learning to branch with Tree MDPs, Oct. 2022. arXiv:2205.11107 [cs, math].

[150] L. Scavuzzo, K. Aardal, A. Lodi, and N. Yorke-Smith. Machine Learning Augmented Branch and Bound for Mixed Integer Linear Programming, Feb. 2024. arXiv:2402.05501 [cs, math].

[151] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. *arXiv:1511.05952*, 2015.

[152] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, T. Lillicrap, and D. Silver. Mastering Atari, Go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, Dec. 2020. ISSN 1476-4687. doi:10.1038/s41586-020-03051-4. Number: 7839 Publisher: Nature Publishing Group.

[153] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, Oct. 2017. ISSN 0028-0836, 1476-4687. doi:10.1038/nature24270. URL .

[154] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419): 1140–1144, Dec. 2018. ISSN 0036-8075, 1095-9203. doi:10.1126/science.aar6404.

[155] J. Song, R. Lanka, Y. Yue, and B. Dilkina. A General Large Neighborhood Search Framework for Solving Integer Linear Programs, Dec. 2020. arXiv:2004.00422 [cs, math, stat].

[156] N. Sonnerat, P. Wang, I. Ktena, S. Bartunov, and V. Nair. Learning a Large Neighborhood Search Algorithm for Mixed Integer Programs, May 2022. arXiv:2107.10201 [cs, math].

[157] A. Srivastava, A. Rastogi, A. Rao, A. A. M. Shoeb, A. Abid, A. Fisch, A. R. Brown, A. Santoro, A. Gupta, A. Garriga-Alonso, et al. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *Transactions on machine learning research*, 2023.

[158] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15 (1):1929–1958, 2014.

[159] Stockfish Developers. Stockfish: Strong open source chess engine, 2024. Open-source chess engine (https://stockfishchess.org/).

[160] P. Strang, Z. Alès, C. Bissuel, O. Juan, S. Kedad-Sidhoum, and E. Rachelson. A markov decision process for variable selection in branch & bound. *Advances in Neural Information Processing Systems*, 38:100886–100912, 2026.

[161] P. Strang, Z. Alès, C. Bissuel, O. Juan, S. Kedad-Sidhoum, and E. Rachelson. Planning in branch-and-bound: Model-based reinforcement learning for exact combinatorial optimization. *Proceedings of the AAAI Conference on Artificial Intelligence*, 40(30):25627–25635, Mar. 2026. doi:10.1609/aaai.v40i30.39759.

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

[163] R. S. Sutton and A. G. Barto. *Reinforcement learning: an introduction*. Adaptive computation and machine learning series. The MIT Press, Cambridge, Massachusetts, second edition edition, 2018. ISBN 978-0-262-03924-6.

[164] Y. Tang, S. Agrawal, and Y. Faenza. Reinforcement Learning for Integer Programming: Learning to Cut. In *Proceedings of the 37th International Conference on Machine Learning*, pages 9367–9376. PMLR, Nov. 2020. ISSN: 2640-3498.

[165] M. Tawarmalani and N. V. Sahinidis. *Convexification and global optimization in continuous and mixed-integer nonlinear programming: theory, algorithms, software, and applications*, volume 65. Springer Science & Business Media, 2013.

[166] M. Turner, T. Berthold, M. Besançon, and T. Koch. Branching via Cutting Plane Selection: Improving Hybrid Branching, June 2023. arXiv:2306.06050 [cs, math].

[167] H. Van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[168] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention Is All You Need, Dec. 2017. arXiv:1706.03762 [cs].

[169] N. Vieillard, O. Pietquin, and M. Geist. Munchausen Reinforcement Learning, Nov. 2020. arXiv:2007.14430 [cs, stat].

[170] O. Vinyals, M. Fortunato, and N. Jaitly. Pointer networks. *Advances in neural information processing systems*, 28, 2015.

[171] Q. Wang and Y. Hao. Routing optimization with monte carlo tree search-based multi-agent reinforcement learning: Q. wang and y. hao. *Applied Intelligence*, 53(21):25881–25896, 2023.

[172] S. Wang, S. Liu, W. Ye, J. You, and Y. Gao. Efficientzero v2: Mastering discrete and continuous control with limited data. In *International Conference on Machine Learning*, pages 51041–51062. PMLR, 2024.

[173] Z. Wang, X. Li, J. Wang, Y. Kuang, M. Yuan, J. Zeng, Y. Zhang, and F. Wu. Learning Cut Selection for Mixed-Integer Linear Programming via Hierarchical Sequence Model. Feb. 2023.

[174] B. Weisfeiler and A. Leman. The reduction of a graph to canonical form and the algebra which appears therein. *nti, Series*, 2(9):12–16, 1968.

[175] G. Williams, A. Aldrich, and E. A. Theodorou. Model predictive path integral control: From theory to parallel computation. *Journal of Guidance, Control, and Dynamics*, 40(2):344–357, 2017.

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

[177] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 2002.

[178] L. A. Wolsey. *Integer programming*. Wiley, Hoboken, NJ, second edition edition, 2021. ISBN 978-1-119-60653-6.

[179] D. Wu and A. Lisser. A deep learning approach for solving linear programming problems. *Neurocomputing*, 520:15–24, Feb. 2023. ISSN 09252312. doi:10.1016/j.neucom.2022.11.053.

[180] D. J. Wu. Accelerating self-play learning in go. *arXiv:1902.10565*, 2020.

[181] Z. Xiao, D. Zhang, Y. Wu, L. Xu, Y. J. Wang, X. Han, X. Fu, T. Zhong, J. Zeng, M. Song, et al. Chain-of-experts: When llms meet complex operations research problems. In *The twelfth international conference on learning representations*, 2023.

[182] Z. Xing and S. Tu. A graph neural network assisted monte carlo tree search approach to traveling salesman problem. *Ieee Access*, 8:108418–108428, 2020.

[183] K. Xu, C. Li, Y. Tian, T. Sonobe, K.-i. Kawarabayashi, and S. Jegelka. Representation learning on graphs with jumping knowledge networks. In *International conference on machine learning*, pages 5453–5462. pmlr, 2018.

[184] W. Ye, S. Liu, T. Kurutach, P. Abbeel, and Y. Gao. Mastering Atari Games with Limited Data, Dec. 2021. arXiv:2111.00210 [cs].

[185] E. A. Yildirim and S. J. Wright. Warm-start strategies in interior-point methods for linear programming. *SIAM Journal on Optimization*, 12(3):782–810, 2002.

[186] K. Yilmaz and N. Yorke-Smith. A Study of Learning Search Approximation in Mixed Integer Branch and Bound: Node Selection in SCIP. *AI*, 2(2):150–178, June 2021. ISSN 2673-2688. doi:10.3390/ai2020010. Number: 2 Publisher: Multidisciplinary Digital Publishing Institute.

[187] G. Zarpellon, J. Jo, A. Lodi, and Y. Bengio. Parameterizing Branch-and-Bound Search Trees to Learn Branching Policies, June 2021. arXiv:2002.05120 [cs, stat].

[188] E. Zhang, A. Kahana, A. Kopaničáková, E. Turkel, R. Ranade, J. Pathak, and G. E. Karniadakis. Blending neural operators and relaxation methods in pde numerical solvers. *Nature Machine Intelligence*, 6(11):1303–1313, 2024.

[189] S. Zhang, S. Zeng, S. Li, F. Wu, and X. Li. Learning to Select Nodes in Branch and Bound with Sufficient Tree Representation. Oct. 2024.

[190] X.-H. Zhou, J. Han, M. I. Zafar, E. M. Wolf, C. R. Schrock, C. J. Roy, and H. Xiao. Neural operator-based super-fidelity: A warm-start approach for accelerating steady-state simulations. *Journal of Computational Physics*, 529:113871, 2025.

Appendix A

The Bitter Lesson

For the keen reader, we reproduce Richard Sutton's original essay *The Bitter Lesson*.

"The biggest lesson that can be read from 70 years of AI research is that general methods that leverage computation are ultimately the most effective, and by a large margin. The ultimate reason for this is Moore's law, or rather its generalization of continued exponentially falling cost per unit of computation. Most AI research has been conducted as if the computation available to the agent were constant (in which case leveraging human knowledge would be one of the only ways to improve performance) but, over a slightly longer time than a typical research project, massively more computation inevitably becomes available. Seeking an improvement that makes a difference in the shorter term, researchers seek to leverage their human knowledge of the domain, but the only thing that matters in the long run is the leveraging of computation. These two need not run counter to each other, but in practice they tend to. Time spent on one is time not spent on the other. There are psychological commitments to investment in one approach or the other. And the human-knowledge approach tends to complicate methods in ways that make them less suited to taking advantage of general methods leveraging computation. There were many examples of AI researchers' belated learning of this bitter lesson, and it is instructive to review some of the most prominent.

In computer chess, the methods that defeated the world champion, Kasparov, in 1997, were based on massive, deep search. At the time, this was looked upon with dismay by the majority of computer-chess researchers who had pursued methods that leveraged human understanding of the special structure of chess. When a simpler, search-based approach with special hardware and software proved vastly

more effective, these human-knowledge-based chess researchers were not good losers. They said that "brute force" search may have won this time, but it was not a general strategy, and anyway it was not how people played chess. These researchers wanted methods based on human input to win and were disappointed when they did not.

A similar pattern of research progress was seen in computer Go, only delayed by a further 20 years. Enormous initial efforts went into avoiding search by taking advantage of human knowledge, or of the special features of the game, but all those efforts proved irrelevant, or worse, once search was applied effectively at scale. Also important was the use of learning by self play to learn a value function (as it was in many other games and even in chess, although learning did not play a big role in the 1997 program that first beat a world champion). Learning by self play, and learning in general, is like search in that it enables massive computation to be brought to bear. Search and learning are the two most important classes of techniques for utilizing massive amounts of computation in AI research. In computer Go, as in computer chess, researchers' initial effort was directed towards utilizing human understanding (so that less search was needed) and only much later was much greater success had by embracing search and learning.

In speech recognition, there was an early competition, sponsored by DARPA, in the 1970s. Entrants included a host of special methods that took advantage of human knowledge—knowledge of words, of phonemes, of the human vocal tract, etc. On the other side were newer methods that were more statistical in nature and did much more computation, based on hidden Markov models (HMMs). Again, the statistical methods won out over the human-knowledge-based methods. This led to a major change in all of natural language processing, gradually over decades, where statistics and computation came to dominate the field. The recent rise of deep learning in speech recognition is the most recent step in this consistent direction. Deep learning methods rely even less on human knowledge, and use even more computation, together with learning on huge training sets, to produce dramatically better speech recognition systems. As in the games, researchers always tried to make systems that worked the way the researchers thought their own minds worked—they tried to put that knowledge in their systems—but it proved ultimately counterproductive, and a colossal waste of researcher's time, when, through Moore's law, massive computation became available and a means was found to put it to good use.

In computer vision, there has been a similar pattern. Early methods conceived of vision as searching for edges, or generalized cylinders, or in terms of SIFT features. But today all this is discarded. Modern

deep-learning neural networks use only the notions of convolution and certain kinds of invariances, and perform much better.

This is a big lesson. As a field, we still have not thoroughly learned it, as we are continuing to make the same kind of mistakes. To see this, and to effectively resist it, we have to understand the appeal of these mistakes. We have to learn the bitter lesson that building in how we think we think does not work in the long run. The bitter lesson is based on the historical observations that 1) AI researchers have often tried to build knowledge into their agents, 2) this always helps in the short term, and is personally satisfying to the researcher, but 3) in the long run it plateaus and even inhibits further progress, and 4) breakthrough progress eventually arrives by an opposing approach based on scaling computation by search and learning. The eventual success is tinged with bitterness, and often incompletely digested, because it is success over a favored, human-centric approach.

One thing that should be learned from the bitter lesson is the great power of general purpose methods, of methods that continue to scale with increased computation even as the available computation becomes very great. The two methods that seem to scale arbitrarily in this way are search and learning.

The second general point to be learned from the bitter lesson is that the actual contents of minds are tremendously, irredeemably complex; we should stop trying to find simple ways to think about the contents of minds, such as simple ways to think about space, objects, multiple agents, or symmetries. All these are part of the arbitrary, intrinsically-complex, outside world. They are not what should be built in, as their complexity is endless; instead we should build in only the meta-methods that can find and capture this arbitrary complexity. Essential to these methods is that they can find good approximations, but the search for them should be by our methods, not by us. We want AI agents that can discover like we can, not which contain what we have discovered. Building in our discoveries only makes it harder to see how the discovering process can be done."

Richard Sutton, 2019

Appendix B

Ecole benchmark

The Extensible Combinatorial Optimization Learning Environments [139], or Ecole, is a benchmarking library built around the SCIP solver and a collection of parametric MILP instance generators. Instead of relying on a fixed dataset of instances, Ecole defines distributions over problem instances via procedural generators for canonical NP-hard families, namely combinatorial auctions, set covering, maximum independent set, multiple knapsack. Each generator is controlled by a small set of parameters, such as problem size (number of items/elements/vertices), structural densities, capacity ratios, or bundle/set statistics. Sampling an instance thus amounts to drawing random data from the prescribed distribution and constructing the corresponding MILP model. This design yields effectively unbounded training data and makes it possible to precisely specify the training distribution through a compact parameterization.

From the perspective of learning to solve repeated MILPs, parametric generators are particularly well suited: they mirror industrial settings in which a solver is deployed on a recurring stream of instances that share modeling assumptions and structural regularities, while varying in their concrete data (realizations of costs, demands, capacities, graphs, etc.). By sampling fresh instances on the fly, one can train policies (e.g., branching or node selection heuristics) that optimize expected solver performance under the target distribution, without overfitting to a finite benchmark set. Moreover, because the underlying distribution is explicit, one can perform controlled generalization studies by perturbing generator parameters at evaluation time. This enables systematic tests of robustness across scales (e.g., increasing the number of variables/constraints) and across distribution shifts (e.g., changing densities, weight/value statistics, or capacity regimes) while keeping the high-level modeling

Table B.1: Instance size for each benchmark. Performance is evaluated on test instances that match the size of the training instances, as well as on larger instances, to further assess the generalization capacity of our agents. Last two columns indicate the approximate number of integer variables after presolve, both for train/test and transfer instances.

			Parameter value		# Int. variables	
Benchmark	Generation method	Parameters	Train / Test	Transfer	Train / Test	Transfer
Combinatorial auction	Leyton-Brown et al. [107]	Items Bids	100 500	200 1000	100	200
Set covering	Balas and Ho [13]	Items Sets	500 1000	1000 1000	100	130
Maximum independent set	Bergman et al. [22]	Nodes	500	1000	480	980
Multiple knapsack	Fukunaga [60]	Items Knapsacks	100 6	100 12	30	50

assumptions fixed.

In our experiments, we adopt Ecole's canonical generators for the four MILP families considered in this thesis, using fixed parameter ranges to define the training distribution and modifying these parameters to create out-of-distribution test and transfer sets. Table B.1 summarizes the generation procedure and the associated parameters used for each class of MILP instances.

We emphasize that Ecole has become a standard experimental backbone for learning-assisted MILP solving: beyond the library paper itself [139], it underpinned the NeurIPS 2021 ML4CO competition, which established standardized tasks, datasets, and evaluation protocols for ML-guided solver components. Moreover, a growing number of subsequent works on learning branching strategies in branch-and-bound adopt Ecole/SCIP-based pipelines and benchmark instance families to ensure reproducibility and controlled generalization studies.

B.1 Combinatorial auctions

The combinatorial auction winner determination problem models the allocation of indivisible items to bidders who may place bids on arbitrary bundles of items. This setting captures complementarities between items and arises in applications such as spectrum auctions and logistics. The resulting optimization problem is NP-hard and is commonly formulated as a set packing problem.

For m items, we are given n bids $\{\mathcal{B}_j\}_{j=1}^n$. Each bid $\mathcal{B}_j \subseteq \{1,\dots,m\}$ is a subset of items with an associated price p_j. The objective is to select a subset of non-overlapping bids of maximum total value:

$$\max \quad \sum_{j=1}^{n} p_j x_j \tag{B.1}$$

$$\text{s.t.} \quad \sum_{j:\, i\in\mathcal{B}_j} x_j \le 1, \qquad \forall i \in \{1,\dots,m\}, \tag{B.2}$$

$$x \in \{0,1\}^n. \tag{B.3}$$

Here, $x_j = 1$ indicates that bid $\mathcal{B}_j$ is accepted. The constraint matrix is binary and typically sparse, with structure induced by the overlap pattern between bids.

B.2 Set covering

The set covering problem is a classical covering problem arising in facility location, crew scheduling, and network design. It consists in selecting a minimum number of sets whose union covers all required elements. The problem is NP-hard and serves as a canonical benchmark for covering-type MILPs.

Given elements $e \in \{1,2,\dots,m\}$ and a collection $\mathcal{S}$ of n sets whose union covers all elements, the formulation is:

$$\min \quad \sum_{s\in\mathcal{S}} x_s \tag{B.4}$$

$$\text{s.t.} \quad \sum_{s:\, e\in s} x_s \ge 1, \qquad \forall e \in \{1,\dots,m\}, \tag{B.5}$$

$$x \in \{0,1\}^n. \tag{B.6}$$

Here, $x_s = 1$ indicates that set s is selected. The constraint matrix is binary and often exhibits moderate to high density depending on the instance generator.

B.3 Maximum independent set

Given a graph $G = (V, E)$, the maximum independent set (MIS) problem seeks a largest subset of vertices such that no two selected vertices are adjacent. MIS is a fundamental graph optimization problem and is NP-hard. In MILP form, it can be expressed either via edge constraints or via clique

inequalities. We adopt a clique-cover formulation, which typically yields a stronger linear relaxation when maximal cliques are used.

Let $\mathcal{C} \subseteq 2^V$ be a collection of cliques whose union covers all edges of G. The formulation writes:

$$\max \quad \sum_{v \in V} x_v \tag{B.7}$$

$$\text{s.t.} \quad \sum_{v \in C} x_v \leq 1, \qquad \forall C \in \mathcal{C}, \tag{B.8}$$

$$x \in \{0,1\}^{|V|}. \tag{B.9}$$

Each constraint enforces that at most one vertex is chosen within each clique. When $\mathcal{C}$ consists of maximal cliques, this formulation dominates the simple edge-based encoding.

B.4 Multiple knapsack

The multiple knapsack problem (MK) generalizes the classical 0–1 knapsack problem to multiple bins. It models assignment and resource allocation problems in which items must be packed into capacitated containers to maximize total profit. MKP is NP-hard and exhibits a characteristic block structure in its constraint matrix.

Given n items with prices $\{p_j\}_{j=1}^n$ and weights $\{w_j\}_{j=1}^n$, and m knapsacks with capacities $\{c_i\}_{i=1}^m$, the formulation is:

$$\max \quad \sum_{i=1}^{m} \sum_{j=1}^{n} p_j x_{ij} \tag{B.10}$$

$$\text{s.t.} \quad \sum_{j=1}^{n} w_j x_{ij} \leq c_i, \qquad \forall i \in \{1, \ldots, m\}, \tag{B.11}$$

$$\sum_{i=1}^{m} x_{ij} \leq 1, \qquad \forall j \in \{1, \ldots, n\}, \tag{B.12}$$

$$x \in \{0,1\}^{m \times n}. \tag{B.13}$$

Here, $x_{ij} = 1$ indicates that item j is assigned to knapsack i. The first set of constraints enforces capacity feasibility, while the second ensures that each item is assigned to at most one knapsack.

Appendix C

Hydro valley modeling

C.1 Industrial Context

The French hydroelectric fleet has an installed capacity of approximately 26 GW, making hydropower the second largest source of electricity generation after nuclear energy. Electricité de France (EDF), as the leading electricity producer in France, operates 433 hydroelectric plants out of the 2300 installed nationwide [50]. The optimal planning of these assets is therefore of major economic and operational importance.

A hydraulic valley consists of a sequence of reservoirs and turbine plants, collectively referred to as dams, located along a main river possibly fed by tributaries. Such a system fulfills a dual role: electricity production through turbine operation and energy storage through water retention in reservoirs. At each time step, the operator must decide whether to release water to generate electricity or to store it for future production. This decision must balance two primary objectives: (i) maximizing revenue from electricity generation in the presence of time-varying market prices, and (ii) reaching prescribed reservoir volume targets at the end of the planning horizon. In addition to these economic objectives, operational, structural, and environmental constraints must be satisfied.

Production planning at EDF is structured in two hierarchical stages. First, a long-term optimization determines the seasonal trajectory of water volumes, accounting for hydrological forecasts, environmental regulations, demand expectations, and financial considerations. This stage fixes, for each day, the initial reservoir volumes and the target volumes to be reached at the end of the period. Second, a short-term optimization determines the detailed hourly production schedule that maximizes financial return while respecting the volume constraints imposed by the long-term trajectory.

Currently, stochastic dynamic programming is used for long-term planning, while daily production schedules are computed by solving mixed-integer linear programs (MILPs). Since dynamic programming requires the repeated solution of many unit transition MILPs, computational efficiency is critical. Until recently, EDF relied on a commercial MILP solver. However, this solution is costly and remains a general-purpose tool not specifically tailored to hydroelectric scheduling. EDF therefore aims to develop a dedicated solver to reduce both financial and technological dependence on external companies and to exploit structural properties specific to these problems.

In particular, EDF engineers have developed a high-performance linear programming component based on a minimum-cost flow algorithm to replace the classical simplex method for solving LP relaxations. However, hydroelectric optimization problems are intrinsically mixed-integer: turbine activation and deactivation introduce discrete decisions modeled through binary variables. Consequently, a branch-and-bound (B&B) framework is required, and in this context, the design of efficient branching strategies becomes a central challenge.

Due to industrial confidentiality, the real-world MILP instances cannot be released. EDF engineers therefore developed a realistic synthetic instance generator designed to capture the main structural features of industrial hydro-scheduling problems.

C.2 Hydro-MILP formulation

Based on internal documentation [45] and technical discussions with EDF engineers, we selected a simplified yet structurally faithful model of hydraulic valley optimization, referred to as Hydro-MILP. The objective is to maximize electricity production revenue over a finite time horizon. While many operational constraints could be incorporated, we retain the following: (i) only the first reservoir is required to meet a prescribed target volume at the end of the planning horizon; (ii) water may be spilled (i.e., discharged without passing through turbines) subject to maximum spill flow limits ; (iii) f or selected dams, variations in turbined flow between consecutive time steps are bounded through constraints on power gradients ; (iv) within each plant, turbines must be activated sequentially in ascending order. Several constraints present in industrial models are not included here. In particular, we do not penalize the number of turbine on/off switches, nor do we impose minimum up/down times. Furthermore, certain real systems require mandatory environmental discharges (e.g., for agricultural

or ecological purposes), which are omitted for simplicity.

Hydro-MILP

Parameters

- Number of timestamps : $T = 24 \in \mathbb{N}$
- Valley architecture and rules :
  - Number of dams : $B \in \mathbb{N}$
  - Number of turbines per plant : $G_b \in \mathbb{N}, \quad \forall b \in [\![1, B]\!]$
  - Minimum and maximum capacities of the reservoirs : $V_b^{\min}, V_b^{\max}, \quad \forall b \in [\![1, B]\!]$
  - Turbines' efficiency constant : $\{C_b\}_{b \in [\![1,B]\!]}$
  - Maximum power of the turbines : $\{P_{b,g}^{\max}(t)\}_{b \in [\![1,B]\!],\, g \in [\![1,G_b]\!],\, t \in [\![1,T]\!]}$
  - Maximum spilled debit : $\{Q_b^{s,\max}\}_{b \in [\![1,B]\!]}$
  - Maximum power gradient : $\{G_b^{\max}\}_{b \in [\![1,B]\!]}$
  - Structure of the hydro valley : $\text{prec}(b) = \{p \in [\![1, B]\!] \text{ s.t. } p \text{ discharges water into } b\}$, $\forall b \in [\![1, B]\!]$
- Electricity prices : $\{\pi(t)\}_{t \in [\![1,T]\!]}$
- Water inflow : $\{Q_b^{\text{in}}(t)\}_{t \in [\![1,T]\!]}, \quad \forall b \in [\![1, B]\!]$
- Extreme values :
  - Targeted volume for the first reservoir : V^{target}
  - Initial dams' volumes : $V_b^{\text{init}}, \quad \forall b \in [\![1, B]\!]$

Variables

- Volume of the dams : $\{V_b(t)\}_{t \in [\![1,T]\!],\, b \in [\![1,B]\!]}$
- Turbined debit : $\{D_b(t)\}_{t \in [\![1,T]\!],\, b \in [\![1,B]\!]}$
- Final volume of the first dam : $V_1(T+1)$
- Turbines' power : $\{P_{b,g}(t)\}_{b \in [\![1,B]\!], g \in [\![1,G_b]\!],\, t \in [\![1,T]\!]}$
- Turbines' activation status (binary variables) : $\{O_{b,g}(t)\}_{b \in [\![1,B]\!], g \in [\![1,G_b]\!],\, t \in [\![1,T]\!]}$
- Spilled flow : $\{Q_b^s(t)\}_{b \in [\![1,B]\!],\, t \in [\![1,T]\!]}$

Constraints

Volume initialization

$V_b(1) = V_b^{\text{init}}, \quad \forall b \in [\![1, B]\!]$

Turbined debit consistency

$D_b(t) = C_b \times \sum_{g=1}^{G_b} P_{b,g}(t), \quad \forall t \in [\![1, T]\!],\ \forall b \in [\![1, B]\!]$

Kirchoff dynamics

$V_b(t) = V_b(t-1) + dt \times (Q_b^{\text{in}}(t-1) + \sum_{p \in \text{prec}(b)} (D_p(t-1) + Q_p^s(t-1)) - D_b(t-1) - Q_b^s(t-1)),$

$\forall t \in [\![2, T+1]\!],\ \forall b \in [\![1, B]\!]$

Maximum spilled debit

$Q_b^s(t) \leq Q_b^{s,\max}, \quad \forall b \in [\![1, B]\!],\ \forall t \in [\![1, T]\!]$

Turn-off & turn-on turbines, above and below power limits

$P_{b,g}(t) \leq O_{b,g}(t) \times P_{b,g}^{\max}(t) \quad \forall b \in [\![1, B]\!],\ \forall g \in [\![1, G_b]\!],\ \forall t \in [\![1, T]\!]$

$O_{b,g}(t) \times P_{b,g}^{\min}(t) \leq P_{b,g}(t) \quad \forall b \in [\![1, B]\!],\ \forall g \in [\![1, G_b]\!],\ \forall t \in [\![1, T]\!]$

Minimum and maximum volumes

$V^b(t) \leq V_b^{\max} \quad \forall b \in [\![1, B]\!],\ \forall t \in [\![1, T]\!]$

$V^b(t) \geq V_b^{\min} \quad \forall b \in [\![1, B]\!],\ \forall t \in [\![1, T]\!]$

Gradient flow constraints

$\sum_{g=1}^{G_b} (P_{b,g}(t-1) - P_{b,g}(t)) \leq G_b^{\max}, \quad \forall b \in [\![1, B]\!],\ \forall t \in [\![2, T]\!]$

$\sum_{g=1}^{G_b} (P_{b,g}(t) - P_{b,g}(t-1)) \leq G_b^{\max}, \quad \forall b \in [\![1, B]\!],\ \forall t \in [\![2, T]\!]$

Domino EDP constraint

$O_{b,g-1}(t) \leq O_{b,g}(t), \quad \forall b \in [\![1, B]\!],\ \forall t \in [\![1, T]\!],\ \forall g \in [\![2, G_b]\!]$

Targeted volume

$V_1(T+1) = V^{\text{target}}$

Objective

$$\max_P \sum_{t \in [\![1,T]\!]} \pi(t) \times \sum_{\substack{g \in [\![1,G]\!] \\ b \in [\![1,B]\!]}} P_{b,g}(t) \tag{C.1}$$

C.3 Hydro-MILP Datasets

Generating concrete instances on the aforementioned MILP model, all parameters and data listed in C.2 are required. While synthetic data could have been used, we opted for real-world values to ensure realism. Due to industrial confidentiality constraints, we restricted ourselves to publicly available datasets. For the valley architecture, operational rules, and natural inflows, we used the case of the Ain river [9], with inferred values where data was unavailable. The resulting topology of our test valley is represented in Figure C.1. For electricity prices, we used hourly spot prices from the Italian electricity market [52], which are freely accessible. Table C.1 summarizes the input datasets.

Dataset	Description	Source	Reference
Ain and its tributaries' inflow	Hourly flow rates (m^3/s) of the Ain hydraulic valley from 1980 to 2018.	HYCAR, INRAE	[9]
Electricity prices	Hourly electricity prices of the Italian market from 2004 to 2021.	GME Italy	[52]

Table C.1: Input datasets for the generation of Hydrolib.

The initial and target volumes are variables of the long-term optimization. In the dynamic programming context, the Hydro-MILP subproblems are solved for various values of V^{init} and V^{target}. Randomly sampling these values is very likely to produce infeasible instances. To avoid this, we sample the initial volumes as

$$V_b^{\text{init}} = x_b^{\text{init}} \times V_b^{\max}, \quad x_b^{\text{init}} \sim \mathcal{U}(0.1, 0.9), \quad \forall\, b \in [\![1, B]\!].$$

We then infer feasible bounds on the target volume of the first reservoir by solving the LP relaxation of a MILP that shares the same constraints as the Hydro-MILP, uses the sampled initial volumes, and takes V^{target} as its objective. Maximizing and minimizing this relaxation yields bounds V_-^{target} and V_+^{target}, from which we sample

$$V^{\text{target}} \sim \mathcal{U}\left(V_-^{\text{target}}, V_+^{\text{target}}\right).$$

We generated six Hydro-MILP benchmarks of increasing dimensions, corresponding to production planning problems over horizons of 2, 3, 4, 5, 6, and 7 days, respectively. Each benchmark consists in 10,000 feasible instances. Table C.2 summarizes the dimensions and characteristics of each benchmark.

	TwoHydro	ThreeHydro	FourHydro	FiveHydro	SixHydro	SevenHydro
Binary vars	720	1080	1440	1800	2160	2520
Continuous vars	1602	2394	3186	3978	4770	5562
Constraints	3883	5827	7771	9715	11659	13603

Table C.2: Hydrolib dataset characteristics (solved with the full strong branching rule).

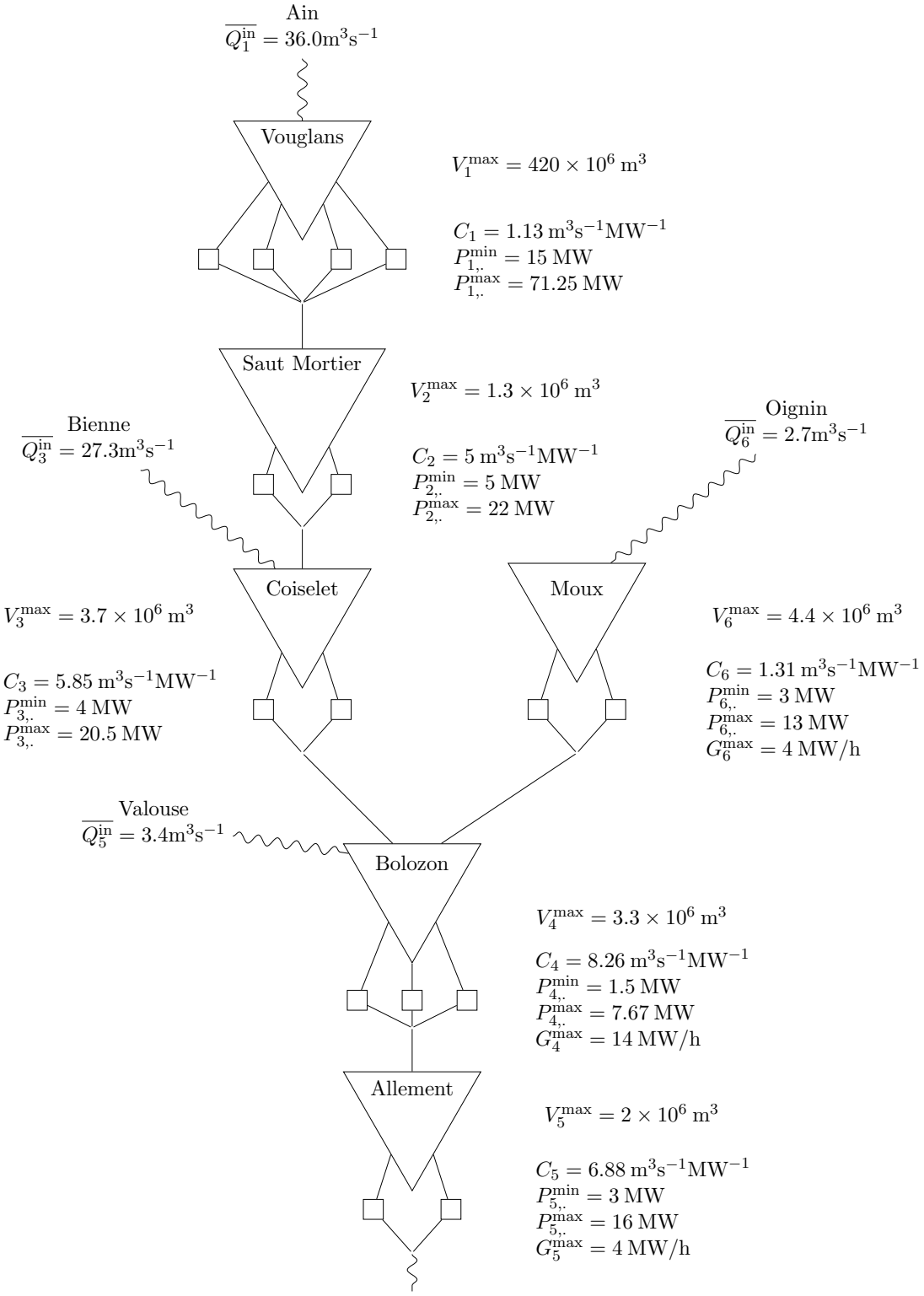


Figure C.1: Topology of the Ain hydraulic valley, with associated industrial parameters.