| | |
|---|---|
| **Titre:** Title: | New Foundations of Machine Learning for Combinatorial Optimization |
| **Auteur:** Author: | Antoine Prouvost |
| **Date:** | 2021 |
| **Type:** | Mémoire ou thèse / Dissertation or Thesis |
| **Référence:** Citation: | Prouvost, A. (2021). New Foundations of Machine Learning for Combinatorial Optimization [Thèse de doctorat, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/9050/ |

**POLYTECHNIQUE MONTRÉAL**
affiliée à l'Université de Montréal

# New Foundations of Machine Learning for Combinatorial Optimization

**ANTOINE PROUVOST**
Département de mathématiques et de génie industriel

Thèse présentée en vue de l'obtention du diplôme de *Philosophiæ Doctor*
Mathématiques

Juin 2021

**POLYTECHNIQUE MONTRÉAL**
affiliée à l'Université de Montréal

Cette thèse intitulée :

**New Foundations of Machine Learning for Combinatorial Optimization**

présentée par **Antoine PROUVOST**
en vue de l'obtention du diplôme de *Philosophiæ Doctor*
a été dûment acceptée par le jury d'examen constitué de :

**Guy DESAULNIERS**, président
**Andrea LODI**, membre et directeur de recherche
**Yoshua BENGIO**, membre et codirecteur de recherche
**Quentin CAPPART**, membre
**Axel PARMENTIER**, membre externe

# DEDICATION

*To the compassionate and open minded people I met in Montreal...*

# ACKNOWLEDGEMENTS

This Ph.D. would not have been possible without the trust that my advisors, Andrea Lodi and Yoshua Bengio, put in me. Thank you for giving me a great freedom of research, especially for the work on Ecole, while still providing me with support and valuable feedback. I was lucky to have advisors that could provide opportunities to showcase my work and network with the research community. I am additionally grateful to IVADO, for sharing the trust of my advisors, and granting me financial support throughout my research.

The daily struggles of administration, project management, and computer networks would have been much harder without the amazing help that the CERC DS4DM team provided. Thank you Koladé, Mariia, Mehdi, and Khalid, for going out of your way to support us.

I am grateful to have found in the DS4DM chair and in Mila a considerate and stimulating work environment. Thank you to the whole community for countless discussions, and in particular to Didier, Maxime, Giulia, and Mathieu for always taking the time to answer so many questions.

Thank you Jonathan, Louis-Martin, and AlayaCare for giving me the trust and opportunity to develop my knowledge on impactful, practical data. It was important for me to realize how different problem solving can be between companies and academia.

Aleksandr, I am sorry that our cutting planes project did not work out as expected, despite our best efforts. All is not lost, we both learned a lot, and many aspects inspired the core design Ecole. Thank you for the guidance you provided, and for understanding that I needed to move on.

The development of Ecole was an important moment for me to express my passion for software engineering. Thank you to all the Ecole and SCIP developers for the times we spent building this project together. I hope it will carry on to help many researchers. None of Ecole would have been possible without the free tools, services, and libraries, provided by the online open source community. I wish to be able one day to give back as much as it has been offering me.

To my close friends in Montreal, whose path I am overjoyed I crossed, and to my family in France, thank you for supporting me with your love through the best and the worst of times.

# RÉSUMÉ

De nombreux problèmes de décision à travers la société peuvent se formuler sous la forme de problèmes d'optimisation à variables discrètes. Ces problèmes, comme ceux de la programmation linéaire en nombres entiers, sont généralement $\mathcal{NP}$-dur à résoudre. Dans les dernières décennies, de nombreux travaux de recherche ont été menés pour tenter de résoudre le plus efficacement des problèmes de grande envergure. De nouvelles améliorations restent nécessaires, cependant, à mesure que le domaine progresse, celles-ci deviennent de plus en plus marginales. Il devient également plus difficile de suivre la façon dont les différentes techniques d'optimisation interfèrent les unes avec les autres, comme c'est le cas par exemple dans les solveurs d'optimisation modernes. Dans cette thèse, nous soutenons que l'apprentissage automatique est un candidat prometteur pour remplacer les heuristiques utilisées pas les algorithmes d'optimisation. Les modèles statistiques ont l'avantage de pouvoir s'adapter automatiquement à des problèmes distribués selon une loi de probabilité inconnue (empirique). Plutôt que de remplacer entièrement les algorithmes d'optimisation par des techniques d'apprentissage automatique, nous défendons qu'exploiter les algorithmes d'optimisation existants fournit une structure adaptée à l'apprentissage, ainsi que la possibilité d'exploiter de fortes garanties d'optimalité (losrqu'elles existent).

Tout d'abord, nous donnons un exemple d'une manière dont l'apprentissage automatique peut typiquement être utilisé pour résoudre des tâches prédictives. Nous démontrons comment l'apprentissage peut être employé pour mieux modéliser les problèmes d'optimisation sans retravailler l'algorithme d'optimisation. Nous illustrons l'efficacité de la méthodologie en l'appliquant à un problème de tournées de travailleurs de la santé dans le cadre de patients recevant des soins à domicile. Nous entraînons un réseau de neurones récurrent avec les relevés médicaux quotidiens des patients afin d'estimer leur risque d'incident. Ces prédictions fournissent des informations tactiques permettant de hiérarchiser les visites lors du calcul des itinéraires des soignants.

Ensuite, nous développons un cadre méthodologique permettant de mieux comprendre les possibilités d'application de l'apprentissage automatique aux problèmes d'optimisation combinatoire. Nous passons en revue la littérature récente et la classons en fonction du degré d'intégration des techniques d'apprentissage et d'optimisation. Nous examinons les différentes méthodes d'apprentissage utilisées, et nous transposons les fondations de la théorie de l'apprentissage statistique à celle de l'apprentissage d'algorithmes d'optimisation.

Enfin, en observant les défis d'ingénierie logicielle existants pour mener des recherches sur

l'apprentissage automatique à l'intérieur de solveurs d'optimisation combinatoire, nous présentons le développement d'Ecole, une bibliothèque logicielle permettant de surmonter ces obstacles. Notre bibliothèque s'appuie sur les processus de décision de Markov, ainsi que sur la bibliothèque OpenAI Gym, pour fournir des abstractions intuitives et hautement personnalisables. Elle permet de reproduire des travaux de recherche existants avec une accélération significative et une forte réduction de la complexité du code source des utilisateurs.

# ABSTRACT

Several decision problems arising in the word involve computing optimal outcomes with regards to discrete choices. These problems, such as mixed integer programming ones, are in general $\mathcal{NP}$-hard to solve. In the last decades, a large body of research has emerged to successfully tackle large-scale optimization problems. New improvements are still needed, but as the field matures, they become harder to establish. It also becomes harder to track how different optimization techniques interfere with one another, as it happens, for instance, in modern optimization solvers. In the present thesis, we argue that machine learning is a promising candidate to complement heuristics in optimization algorithms. Statistical models have the advantage that they can be adapted to a given probability distribution of problem instances in an automated and scalable way. Rather than fully replacing optimization algorithms with machine learning, we argue that leveraging existing optimization algorithms provides useful structure to facilitate learning, as well as the possibility to leverage strong optimality guarantees (when they exist).

First, we illustrate one way that machine learning can typically be used for predictive tasks. We show how learning can be practically used to better model optimization problems without reworking the optimization algorithm. We demonstrate the effectiveness of the methodology through an application to an healthcare workers routing problem in the context of home healthcare patients. We train a recurrent neural network on daily medical records of patients to estimate their risk of adverse events. These predictions provide tactical information to prioritize visits when computing care workers routes.

Second, we develop a methodological framework to better understand the possible ways of applying machine learning to combinatorial optimization problems. We survey the recent literature and classify it by the level of integration of learning and optimization techniques. We look at the different learning methods used, and transpose key elements of the learning theory to learned optimization algorithms.

Finally, motivated by the software engineering challenges to conduct machine learning research inside combinatorial optimization solvers, we present the development of Ecole, a library to overcome these hurdles. Our library builds on Markov decision processes and the OpenAI Gym library to provide intuitive and highly customizable abstractions. It can already be used to reproduce existing research with significant speedup and with reduced complexity in user source code.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

## LIST OF SYMBOLS AND ACRONYMS

**API** application programming interface. xi, 56, 60, 61, 63, 64

**B&B** branch-and-bound. 1, 2, 4, 5, 7, 22, 23, 32, 35, 38, 39, 41–43, 46, 50, 53, 56–58, 61, 64, 65, 70, 73

**C-Index** concordance index. 11–14, 16

**CNN** convolutional neural network. 33, 51

**CO** combinatorial optimization. xiii, 1–7, 18–23, 25, 29, 30, 34, 35, 37–40, 44–46, 48, 50–53, 55–57, 70–73

**CP** constraint programming. 2

**DL** deep learning. 1, 4, 6, 7, 55, 70–73

**GAN** generative adversarial network. 3, 38

**GIL** global interpreter lock. 56, 65

**GNN** graph neural network. 4, 5, 28, 32, 33, 36, 51, 61, 70, 71, 73

**GRU** gated recurrent unit. 11, 13, 14

**HHC** home healthcare. 6, 8, 9, 13

**HT** home telemonitoring. 8, 9, 13, 14

**LP** linear programming. xii, 22, 23, 32, 38, 40, 50, 58, 65

**LSTM** long short-term memory. 11, 13

**MDP** Markov decision process. xiii, 3–5, 7, 25, 26, 30, 34, 42, 49, 55, 56, 58–60, 63, 70–72

**MILP** mixed-integer linear programming. xii, 1, 2, 4, 5, 22, 23, 31, 32, 34, 36–39, 44, 45, 47, 50–52, 55–58, 60–62, 64, 71, 73, 74

**MIQP** mixed-integer quadratic programming. 2, 37

**ML** machine learning. xiii, 1, 3–8, 19–21, 24, 25, 27, 29, 30, 32–46, 48–53, 55–57, 59, 63, 68, 70–74

**MLP** multilayer perceptron. 26, 36

**NN** neural network. 3, 4, 74

**OR** operations research. xii, 1, 8, 37, 57

**QP** quadratic programming. 37, 38, 40

**ReLU** rectified linear unit. 10

**RL** reinforcement learning. xiii, 3–5, 25, 26, 30, 31, 33, 34, 43–45, 48, 50, 56–60, 70–73

**RNN** recurrent neural network. xii, 6, 27, 28, 35, 36, 50, 51, 72

**SAT** boolean satisfiability problem. 2

**SDP** semidefinite programming. 31, 32

**SGD** stochastic gradient descent. 10, 12, 14, 27, 51

**TSP** traveling salesman problem. 4, 18–20, 33, 35, 36, 44–46, 52

# CHAPTER 1 INTRODUCTION

## 1.1 Motivation

Many decision problems that arise in the planification of complex systems (*e.g.* cities, companies, or organizations) require a deep mathematical modeling of the problem to find an efficient operations plan, let alone a working one. Such problems can be viewed as constrained optimization problems, where the objective represents a measure of cost (*e.g.* financial, social, or well-being) and constraints model the process and its physical and societal restrictions. The present thesis gives particular attention to mixed-integer linear programming (MILP), a paradigm where variables can be integer or continuous, but constraints are restricted to be linear. It has been the workhorse of combinatorial optimization (CO) for the past decades. Despite belonging to the class of $\mathcal{NP}$-complete problems, MILP has been successfully applied to a wide variety of large-scale problems, including energy distribution, supply-chain, scheduling, transportation, and finance.

Such successes can, in part, be attributed to the advent of modern solvers – complex softwares designed to solve diverse optimization problems by using an arsenal of state-of-the-art algorithms. Solvers such as SCIP [1], Cplex [2], and Gurobi [3] build on the branch-and-bound (B&B) algorithm [4] and enhance it with a large number of heuristics [5–7]. Everything is finely tuned on a collection of "*real-world*" problem instances such as MipLib [8].

At the same time, machine learning (ML) algorithms, *i.e.* algorithms learned from examples, are becoming fruitful to solve other types of applications, like computer vision, language translation, speech recognition, and recommender systems, by providing both *predictive* and *generative* capabilities. In recent years, deep learning (DL) has been particularly powerful at solving the aforementioned tasks [9]. Moreover, its dramatic achievements seem to inspire new applications in fields left untouched by ML, as it is the case with drug discovery, and, the topic of this thesis, CO.

The overlap between ML and optimization in general is quite broad. ML practitioners know that *learning* is defined as optimizing a (stochastic) loss function over a dataset of training examples. The present thesis focuses on the opposite interaction, that is, how to leverage ML to solve operations research (OR) problems, specifically in CO. CO algorithms have slowly evolved to perform well on problems from a "*real-world*" distribution. However, doing so requires extensive trials and errors through expert research. Incorporating ML inside CO algorithms would make is possible to automatically tune an algorithm to a problem distri-

bution of interest (such as an application-specific distribution), by searching the algorithms space. The set of CO algorithms that can be searched is also much larger, possibly offering important performance improvements over manually written algorithms.

## 1.2 Background

### 1.2.1 Combinatorial Optimization

CO is a sub-field of mathematical optimization (the minimization of a function over a constrained domain) involving discrete variables whose Cartesian product spans a domain too large to be enumerated. There exist different classes of problems, such as boolean satisfiability problem (SAT), constraint programming (CP), MILP, mixed-integer quadratic programming (MIQP), that assume prerequisites on the variables (binary, integer,...) and constraints (linear, quadratic,...). The present thesis gives particular attention to MILP. Its achievements and generality make it a candidate of choice for many practical applications. Furthermore, focusing on MILP does not diminish the generality of the methods presented in this thesis.

An MILP problem can be formulated as

$$\begin{aligned} \min\ & c^\intercal x \\ \text{s.t. } & Ax \geq b \\ & x \geq 0 \\ & \forall i \in \mathcal{I},\ x_i \in \mathbb{Z} \end{aligned} \tag{1.1}$$

where $x \in \mathbb{R}^n$ are the problem *variables*, $c \in \mathbb{R}^n$ is the *objective* vector, $A \in \mathbb{R}^{m \times n}$ are the *constraints coefficients*, and $b \in \mathbb{R}^m$ are the *constraints right-hand sides*. A subset $\mathcal{I}$ of the variables are restricted to take integer value. These integrality constraints are what makes solving the optimization problem *combinatorial*.

Directly finding solutions to the problem can be achieved with a variety of so-called *primal heuristics*. *Exact algorithms*, like the ones implemented in solvers, aim to prove that a solution is *optimal*, by providing a *lower bound* matching the objective value of a solution. Solvers implement an enhanced version of the B&B algorithm [4]. *Presolving* techniques [10], *cutting planes* methods [11, 12], and *primal heuristics* [13] are critical aspects that make solvers so powerful [5–7]. The reader is referred to *Integer Programming* [14] for a textbook on MILP.

Overall, solvers must not be seen as a single consistent algorithm, but rather as a set of techniques whose performance may vary [15]. The orchestration of all these techniques is manually tuned by researchers on "*real-world*" problem instances such as MipLib [8].

### 1.2.2 Machine Learning

The field of ML is trying to make sense of data generated in the world. In order to make predictions or decisions, ML algorithms are *trained* to perform well on given data, rather than being explicitly conceived to solve that application. Training a ML model is defined as optimizing a loss function (the performance of the model) over some inner parameters. The success of a model is defined by its ability to *generalize* to unseen examples. ML therefore has strong connections to statistics.

*Supervised learning* is the most reliable way to train ML models as it aims to directly minimize the prediction error of the model compared to a ground truth. The downside is that it requires a labeled dataset. Collecting such a dataset often involves a lot of tedious human work. On the contrary, the goal of *unsupervised learning* is to learn meaningful representations of the data without needing explicit labels. It can be used for tasks like probability density estimation, dimensionality reduction, and clustering. It encompasses many different techniques, such as generative adversarial networks (GANs) [16]. Recently, *self-supervised learning* has emerged to rip the benefits of supervised learning without the need for human labeling. The idea is to create substitute tasks whose labels can be computed automatically. It has been driving the latest advances in language processing, from Word2Vec [17] to BERT [18]. The reader is referred to *Pattern Recognition and Machine Learning* [19] and *Machine Learning: A Probabilistic Perspective* [20] for textbooks on supervised and unsupervised learning.

When applied to control problems, ML is best described by the Markov decision process (MDP) framework used in reinforcement learning (RL). In this framework, an agent interacts with an environment receiving rewards. The optimization problem for the agent is to maximize its expected sum of future rewards. The problem can also be tackled through imitation learning, that is, imitation learning applied to the actions of an expert. The reader is referred to *Reinforcement Learning: An Introduction* [21] for a textbook on RL.

Deep neural networks (NNs) – high dimensional composable functions – have shown tremendous performances compared to rule-based systems popular in the $20^{\text{th}}$ century. They have sparked a new ML revolution, propagating to other fields, including CO.

# CHAPTER 2 CRITICAL LITERATURE REVIEW

The application of ML to solve CO is not a new topic. In fact, in the 1990s, numerous approaches in the ML community tried to solve CO problems with NNs [22]. In 2015, DL has already successfully been applied to a wide variety of learning tasks, including outside academic research. New architectures, such as those based on attention [23], make it possible to apply NNs to unexplored data types. Vinyals *et al.* [24] refined attention mechanisms to build the Pointer Network, a NN able to operate on complete graphs, and applied it to find solutions to the traveling salesman problem (TSP). At that time, the motivation in the ML community was not as much about building practical solutions to CO problems, but rather, in the line of Neural Turing Machines [25], to demonstrate that NNs can solve algebraic tasks. With the development of graph neural networks (GNNs) [26], and the implication of the CO community, NNs have become much more efficient at finding good solutions to CO problems [27].

Yet, finding good solutions (primal heuristics) is only part of the challenge in CO. Proving that the solution is optimal (or providing an optimality gap) with an exact algorithm is as equally important. In CO, exact algorithms are often implemented in solvers. Researchers have long tried to find the best parameters of a solver for specific probability distributions of problem instances [28, 29]. To achieve this goal, ML quickly became part of the equation [30]. Similarly to solver configuration, the less analytical training procedures of NNs led DL researchers to look for good *hyper-parameters tuning* algorithms (or *auto-ml*).

As more advanced ML techniques reached researchers working with solvers, new approaches emerged to learn entire solver heuristics, instead of simply configuring parameters. For instance, in MILP, Khalil *et al.* [31] learned when to run primal heuristics, and Gasse *et al.* [32] learned a B&B variable selection policy. In DL as well, researchers went beyond hyper-parameters tuning to learn a gradient descent update rule [33–35].

Building on the work of Lodi and Zarpellon [36] that explores the specific case of B&B variable selection, our methodology [37] (Chapter 5) puts all this development into context and offers a unified framework based on MDPs to understand "*learning to optimize*" approaches with either imitation or reinforcement learning. Since then, ML strategies for CO kept growing richer. For B&B variable selection, new research shows the ability to learn a policy that is lighter [38], that can scale to larger problem instances [39], that can generalize to diverse instances [40], and that can be improved with RL [41], including on real-world instances [42]. The solver heuristics considered for learning are also widening their scope, with researchers

focusing on cutting plane selection [43] and B&B node selection [44]. GNNs, due to their unprecedented flexibility, seem to be an indisputable tool for building ML models for CO [45].

Alternatively, ML can be used in solvers without explicitly controlling them. For instance, ML has been applied to learn whether an MILP problem will be solved before it reaches a time limit [46]. While this information could be seen as a building block for automated decision making, it also favors a collaborative approach with human experts, where ML performs analysis but decisions are left to humans.

Working on these advanced topics requires a deep understanding of ML tools and CO solvers. Even then, the amount of software engineering required to get to a first working experiment is dissuasive for many researchers. Software libraries are emerging to accelerate research in the field by providing intelligible and reusable abstractions that are well tested and easy to install. OR-Gym [47] and OpenGraphGym [48] let users develop primal heuristics for CO problems using the OpenAI Gym [49] interface that is familiar to RL practitioners. MIPLearn [50], on the other hand, enables users to learn to configure solvers and works with both Gurobi [3] and CPLEX [2]. Our library Ecole [51] (Chapter 6), also based on the OpenAI Gym interface, fills the empty space of learning heuristics as MDPs in general purpose solvers such as SCIP [1]. The tools used to build GNNs have also improved and matured, with libraries such as Pytorch Geometric [52] and Deep Graph Library [53].

ML has also been used to properly define the optimization problems of interest. Modeling a practical problem mathematically usually involves back and forth between optimization and domain experts. ML can be used to automate modeling [54]. Furthermore, mathematical constraints are usually rigid, while expert domain knowledge is more intuitive. ML can be applied to reproduce such expertise before feeding it to an optimization pipeline [55]. Some constraints cannot even be formulated explicitly, but some novel frameworks make it possible to embed learned domain restrictions inside optimization algorithms [56].

It is in this large body of research that the contributions presented in this thesis are set. The ML task detailed in Chapter 4 is actually part (although it is not highlighted in the publication) of an optimization problem and can be framed as a way to estimate the parameters of that problem. Chapter 5 extends the present literature review, and offers a methodological framework to build CO algorithms that embed ML. Chapter 6 introduces a new software in the ecosystem of ML and CO that gives a practical implementation to the aforementioned framework.

# CHAPTER 3 DISCUSSION OF THE RESEARCH PROJECT AS A WHOLE AND GENERAL ORGANIZATION OF THE THESIS INDICATING THE COHERENCE OF THE ARTICLES IN RELATION TO THE RESEARCH GOALS

The present thesis studies ways in which ML can be leveraged for CO. We show, through an ML application, how ML can be used before of optimization to estimate the parameters of an optimization problem of interest. Furthermore, we define a framework for integrating ML as a component of optimization algorithms. The applications of mathematical optimization, and CO in particular, to train and verify ML models is out of the scope of this thesis.

## 3.1 Contributions

**Adverse Event Prediction by Telemonitoring and Deep Learning** (Chapter 4, [57])
Our first contribution presents a recurrent neural networks (RNNs) used to predict home healthcare (HHC) patients risk of relapsing on a daily basis. Home recovery is beneficial for both patients and hospitals, but close monitoring – through sensors and care worker visits – is difficult to set up. The approach presented builds on a ranking objective from survival analysis to assess relative patient risks, while accounting for censored data. We show that our DL model is able to outperform manual alert thresholds proposed by care workers. This publication focuses solely on risk estimation through DL. However, this work fits more generally in the interplay of ML and CO. Risk predictions are used by the partner company to help decide which patients to visit on a given day. A vehicle routing problem [58], not presented in our contribution, is solved by the company to find care workers routes of minimal length. Our application therefore demonstrates how ML can be successfully applied ahead of discrete optimization to define the parameters of the problem to solve.

**Machine Learning for Combinatorial Optimization:**
**a Methodological Tour d'Horizon** (Chapter 5, [37])
Our second contribution establishes a framework for using ML in CO and has already served hundreds of researchers worldwide. In this framework, we propose to view the evolution of handcrafted heuristics of CO solvers as a weak form of learning, where training is achieved through researcher trial and error. We advocate using ML for solving CO problems as a way to automatically tune algorithms to the statistical distribution of problems of interest (defined according to the application). We offer a methodology, an extensive survey of the

recent accomplishments, and a background on both ML and CO. We examine how ML can be used as a way to build primal heuristics, as a way to configure optimization algorithms, or more generally as a control policy inside an existing algorithm seen as a MDP. We survey using either reinforcement, or imitation learning from readily available expert policies, to deliver optimization improvements. We transcribe fundamental concepts of learning theory, including the notion of statistical learning, generalization, and meta-learning for their application in CO. Finally, we discuss the future and challenges of scaling ML to larger CO problems.

**Ecole: A Library for Learning Inside MILP Solvers** (Chapter 6, [51])
Our third contribution mirrors the aforementioned article. It provides a software interface, Ecole, to the methodology previously presented. Ecole is a software built to facilitate the application of ML tasks inside the SCIP [1] solver. For instance, it can be used for learning to select B&B variables to branch on. Such control problems are framed as MDPs, as suggested by Bengio *et al.* [37]. In Ecole, they are presented according to the familiar OpenAI Gym interface [49]. Ecole offers a dual interface in Python and C++ that is modular. A number of observation and reward functions are provided to reproduce previous research, such as that of Gasse *et al.* [32] and Khalil *et al.* [59]. We show that Ecole is significantly faster than the interface of Gasse *et al.* [32] to extract data from the solver. Ecole is well tested, well documented, and easy to install. We aim for Ecole to help researchers building better ML models for CO, but also to define what heuristics in SCIP are worth learning. This is why we built Ecole to be deeply customizable, so that it enables cutting-edge research.

## 3.2 Thesis Outline

The remainder of this thesis is organized as follows. In Chapter 2, we review critical literature on the integration of ML in CO algorithms. Chapters 4, 5, and 6 contain the core contribution of the thesis, as outlined in the previous section. Specifically, Chapter 4 details a DL model to estimate health care patient risk of adverse event, Chapter 5 presents in extensive details a literature review, methodological framework, and theoretical background that complete those of the present thesis, and Chapter 6 introduces a software library to facilitate the application of ML inside a CO solver. In Chapter 7, we discuss the three contributions as a united ensemble. Finally, in Chapter 8, we present a summary of our work, examine it's limitations, and debate new research directions that build on this thesis.

# CHAPTER 4 ARTICLE 1 - ADVERSE EVENT PREDICTION BY TELEMONITORING AND DEEP LEARNING

Authors: Antoine Prouvost, Andrea Lodi, Louis-Martin Rousseau, and Jonathan Vallee
Published in *International Conference on Human-Centred Software Engineering* [57].

**Abstract** Home health care comes as a potential solution to increasing stress on health-care systems, as well as concerns for medical patients comfort. However, additional distance from the care workers to the patients lead to more challenges, some of which can be addressed with machine learning and operations research algorithms. In this paper, we focus on automating a risk assessment of remote patients. Namely, we describe a risk prediction framework for home telemonitoring patients and show that learning a risk from weak signals in the patient's data outperforms simple risk threshold proposed by care workers to automate the task. We combine recurrent neural networks with a ranking objective from survival analysis to evaluate the risk of patient's adverse events. Training and testing of our methodology is achieved on a retrospective dataset gathered by an Ontario home health care agency during the course of a multi-year pilot home telemonitoring program. Results are benchmarked against alerts that were manually engineered by registered nurses, and against a simple linear baseline. This is an additional step in the application of machine learning in health care for patient-centered personalized treatments.

## 4.1 Introduction

Population ageing comes with increased care needs since 85% of elderly will develop chronic conditions [60]. From 6.9% in 2000 to an estimated proportion of 19.3% of the global population in 2050 [61], the elderly account for a growing proportion of the health care costs.

Keeping elderly healthy and at home longer is thus a critical endeavour. Home healthcare (HHC) is starting to be widely adopted since it is seen as a cost effective alternative to traditional care and because patients often prefer it.

Home telemonitoring (HT), a specialized form of HHC is a potential alternative that empowers patients to take charge of their health, generates reliable data that can be leveraged to better assess the patients' states and that may improve the patient's medical condition [62].

Given the growing demand for HHC and HT, data is accumulating at an extreme velocity, in a great volume and in a variety of forms. The advancements in monitoring devices is

also contributing to the velocity and volume of data generated by HT programs. Valuable information lies in this data. There is thus a pressing need for improved decision systems that can use the information.

When a patient is admitted to a home care agency, she generally gets visited by a registered nurse who will perform an initial needs assessment [63]. If the agency offers a HT program, patients can be admitted to it. While on a HT program, the patient answers a periodic questionnaire during which she will be asked to take some vital signs readings. This information is then transmitted to the HHC agency where a nurse monitors a HT case load. Based on the patient diagnosed conditions and initial assessment, the care workers create alerts based on acceptable range of each of the measured vital signs as shown in the work of Suh *et al.* [64]. Sometimes, with more advanced systems, complex rules can be developed to get alerted based on combinations of suspect readings.

In all cases, care workers bear the weight of setting up patients with the right set of alerts based on their conditions. The manually engineered rules then need to evolve with the patient's condition in order to remain reliable. When a vital sign reading is out of the acceptable range, the monitoring nurse can perform one or two of the following actions: (1) call the patient to determine next steps, and/or (2) schedule an in-person visit. The challenge is to prevent costly hospital readmissions and emergency room visits, but there is also a cost to each intervention. To add complexity, most of the alarms are false positives, not leading to adverse events.

Early detection of these events serves the purpose of the triple aim of improving outcomes: (1) quality of health services, (2) improving health of populations and (3) reducing costs [65].

The contribution of this paper is to propose a patient ranking approach to adverse events prediction and to show that it performs well on a retrospective patient cohort.

The remainder of the paper is organized as follows: In Section 4.2, we review the body of work related to adverse events predictions. In Section 4.3, we review the technical background required for our proposed solution to adverse events prediction. Section 4.4 details our proposed framework and Section 4.5 reports our results. We conclude the paper with a discussion and perspectives in Section 4.6.

## 4.2 Adverse Event Prediction Related Work

Since about 20% of Medicare patients are readmitted within 30 days of discharge [66] and since Huntington *et al.* [67] established financial penalties to hospital with the highest readmission rates 30 days after discharge, prediction of adverse events such as hospital readmis-

sions has been extensively done in health care research.

Linear models such as multivariate logistic regression and Cox Proportional Hazard [68–71] are often used because of their understandable nature. Indeed, most of the work so far has been interested in understanding the significant factors that lead to adverse events. Conversely, neural networks have not been used as much because they are seen as hard to interpret black-boxes [72] despite their success in many industries, from computer vision to market finance.

In health care, some examples of neural network use are as diagnostic tool [73], as prediction tools [74, 75], as emergency states detectors [76], and in psychology [77]. In particular, Baxt *et al.* [73] hypothesizes that neural networks could outperform linear models because of their capacity to capture relationships between input variables that are not seen by simpler models. More recently, additional work has been done using neural networks to anticipate patient outcome (mortality, readmission, extended stay, *etc.*) from their electronic health records [78, 79], including using recurrent neural networks [80].

### 4.3 Technical background

Neural networks are parametric functions approximators built by composition. The network is built by alternatively composing matrix multiplications and a non linear (element-wise) function, called activation function (such as the rectified linear unit (ReLU), $x \mapsto \max(x, 0)$). This type of architecture is that of a multi-layer perceptron. Finding the coefficients of the matrices building the network is an optimization problem, also called "*learning*" or "*fitting*" the model. The loss function of this problem depends on the input data. In supervised learning, neural networks are optimized on a training set to minimize a loss function between their prediction and an observed target. The nature of the loss function depends on the task. Usually, mean square error is used for regression and cross-entropy for classification. To minimize the training error, neural networks are designed differentiable, and optimized using stochastic gradient descent (SGD), a gradient descent approach where the loss is approximated over a subset of the training examples (called a "*mini-batch*"). Optimizing on a training set eventually leads to degrading performances on unseen examples (this is known as *over-fitting*). To curtail this, the performance of the neural network is monitored on a dataset of unseen examples, known as the *validation* set, and optimization is stopped once the validation metrics worsen. This is done in combination with regularization techniques: any method that empirically reduces the test error, at the expense of the training error.

Recurrent neural networks, reuse their internal parameters sequentially to be able to process

and learn over time series data of arbitrary length. At every step, they combine information coming from the current time step with past information condensed by the neural network into a hidden state vector. Popular models include the long short-term memory (LSTM) [81] and gated recurrent units (GRUs) [82]. When time series are long, recurrent neural networks can be trained with truncated back propagation through time [83]. Namely, the gradients are approximated, and optimization steps are taken, over consecutive subsets of data along the time dimension. Data global to a time series can be combined with the neural network, for instance through the initial hidden vector. The reader is referred to *Deep learning* [9] for an extensive textbook on deep learning.

### 4.4 Adverse Events Prediction

Because predicting adverse events, either as a regression (for the time to the next adverse event), or as a very in-balanced classification (classifying if an adverse event will occur in the next $k$ days), is hard, we chose to model the problem as a survival task. Namely, we predict a *latent* patient risk of experiencing an adverse event. It is important to understand that this risk is interpreted only relative to other patient risks, that is a patient risk is higher than another if the former is more likely to experience an adverse event than the latter. In other words, this risk is only introduced to output a ranking of patients. Given a predicted ranking, and the true ranking (computed from the times to the next adverse event), we can compute a score metric called concordance index (C-Index) [84]. This measure is not differentiable and therefore cannot be optimized through gradient based methods (as it is done for neural networks). Hence, we use a surrogate loss derived from the maximum likelihood of the Cox proportional hazard model for survival analysis [85]. This likelihood loss function is used to compute gradients for the neural network, but model selection and final scores are expressed in terms of C-Index. The details of both the C-Index and the Cox likelihood functions can be burdensome and we deliberately omit it here. In short, the C-Index is a measure counting the percentage of pairs (of patients here) properly ranked. The Cox model makes more assumptions on the mathematical form of the risk function in order to derive a likelihood. Both are able to deal with censored data, *i.e.* patients exiting the program (or the program ending). The interested reader is referred to the aforementioned literature, as well as the adaptation for neural networks introduced by Katzman *et al.* [86]. Unlike in Cox proportional hazard model, the problem contains a strong time component as we wish to predict a risk for every patient *on every day* (with new information coming in). The metrics are therefore evaluated on a daily basis across all patients. Modeling the problem as a ranking problem is in line with the application pursued here. Indeed, on every day, it

is sufficient for the care giver to be able to rank the patients by risk, in order to provide an intuition on where to prioritize, as care workers cannot visit all patients on a daily basis.

As depicted in Figure 4.1, we use a recurrent neural network to process patient data, both static (patient information and diagnoses) and time distributed (vital signs measured on a daily basis). The network outputs a risk for every patient, on every day. We use the Cox log-likelihood as a loss function to train the network, as was previously done by Katzman *et al.* [86], and report the C-Index as well.

Static patient data contain medical diagnoses codes from the International Statistical Classification of Diseases and Related Health Problems (ICD9, ICD10) [87], which are not very informative on their own. To tackle this issue, Choi *et al.* [88] uses an unsupervised deep learning approach to embed these codes into a more meaningful vector space, using additional information from the disease, as well as other types of codes. The embedded codes show desirable properties such as diseases with similar symptoms or prescriptions are close together in Euclidian distance. We used their pre-trained embeddings to represent this part of our data. Because patients have a variable number of diseases, we need to use another (small) recurrent neural network (hidden inside the orange parallelogram in Figure 4.1) to process these diseases before passing them on to the main (larger) recurrent neural network. Alternatively, we also try not to include that information, and simply pass a null vector (as usually done), with the intuition that learning should be easier.

We face more challenges as we have some missing data in the vital signs observed on a daily basis. Although more complex solutions exist to model this (*e.g.*, Che *et al.* [89]), we chose the simple approach of modelling missing data with additional binary variables representing if the data is missing.

The data is split into training (60%), validation (20%) and test (20%) sets. We made sure that no information from the future could be used to make prediction and hence computed the splits as dates: from the beginning to the first split date would constitute the training set *etc.* Past events (previous targets) however can be used as input after their occurring date. Some patients appear in multiple sets, while other are new in every one. This is because of the nature of the application as we want to keep assessing the risk of patients throughout their participation in the program and not just once.

Neural networks and their optimization algorithms come with a number of so called "*hyper-parameters*": parameters that cannot be optimized directly. In our case, these hyper-parameters divide into two groups. The first group controls the optimization algorithm itself: SGD, or the Adam variant [90], the gradient step size, the $L_2$ regularization multiplier, the number of examples in a SGD mini-batch, the number of time steps considered in the

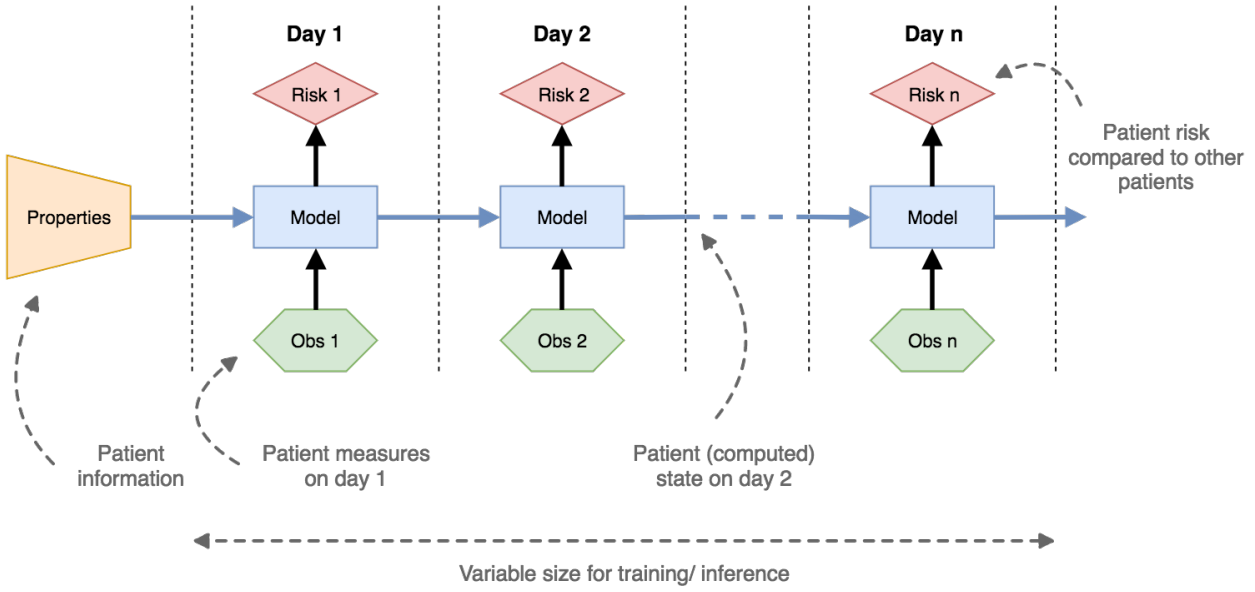



Figure 4.1 A recurrent neural network is used to combine the patients *a priori* information with their daily vital signs and outputs a risk.

truncated version of back-propagation through time, and the use of dropout (a regularization technique) [90]. The second group controls architectural decisions: number and size of hidden layers (the matrices involved in the neural network), the type of recurrent layers (LSTM or GRUs), and whether to use the patient static data as the initial hidden vector or to simply omit it. A pragmatic way to select these hyper-parameters is to generate randomly some configurations, train the networks for each of them and finally keep the one performing best (in C-Index) on the validation set.

Expanding on the evaluation of machine learning performance, we compared it to manually engineered alerts (four levels of severity). We also compare to a simple linear survival baseline using the time distributed readings independently (time dependencies are not taken into account) without using static patient data.

## 4.5 Results

The dataset we use has been gathered by an Ontario private HHC agency during the course of a multi-year pilot HT program and is fully anonymized. The input data include the patient static information (sex, age, and medical records through ICD codes), and daily vital sign readings (blood glucose, systolic, diastolic, heart beat, $SpO_2$ oximeter, and weight).

The dataset also contains observed adverse events experienced by the patients and used to compute the losses (either Cox log-likelihood or C-Index). On any given day, past events are also added as input.[1] The 320 patients in our study were aged from 31 to 101 with an average age of 79. Women represented 56.25% of the patient group. Moreover, 36.25% of patients experienced at least one hospital readmission or emergency room visit while on the HT program. The average number of events per patient was 0.72 with a standard deviation of 1.30. The average number of events for the 36.25% of patients that experienced at least one was 1.98 with a standard deviation of 1.47. Finally, 91.56% of patients had comorbidities, hypertension being the more frequent at 55.31% followed by chronic heath failure with 46.25%, diabetes with 39.69% and chronic obstructive pulmonary disease at 38.13%. In total, we have access to 76,359 daily patient observations.

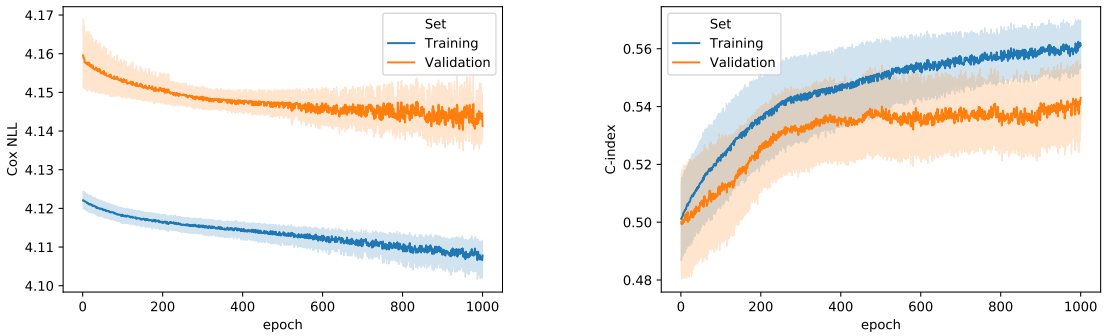
The training process of neural networks was highly stochastic, with results often close to random, as there is a strong noise to signal ratio in the data. This made it difficult to differentiate promising models from random luck. Therefore, hyper-parameters configurations were manually selected from across all runs (over a hundred), based not only on validation performance, but as well on stability (reduced stochasticity during training) in addition to small gap between the training and validation scores.

These configurations were then retrained over twenty random seeds to average the results. The training for the best performing set of hyper-parameters is depicted in Figure 4.2. Important details of this configuration are: three GRU layers with eight units each and dropout for the network architecture, an SGD optimizer with a batch size of 64, and a truncation length for back-propagation through time of 15 days for the training procedure. It is worth noting that the neural network presented in Figure 4.2 does *not* use static information about the patients (this was a configurable hyper-parameter choice). That is among all the configurations trained, the best performing model was one that did not make use of the static information. This is, further discussed in Section 4.6.

We removed from these twenty models a few that did not perform well on the training or validation set (three of them). Due to the stochasticity in the training process, some trained networks can under-perform. We can legitimately filter them out, as long as we do so on validation or training sets.[2] Then, we computed their final score on the test set. The results are given in Figure 4.3. The box plot reports a test concordance index of $58.8 \pm 4.6\%$. We performed a Student T-test against the value of 50% and rejected with p-value $3.7 \times 10^{-7}$ that our model is equivalent to a random one.

[1] This is correct as no information from the future is added to the input.

[2] The only difference is that the training procedure now includes a filtering phase.




(a) Cox negative log-likelihood.

(b) Concordance index.

Figure 4.2 Training of the most promising set of hyper-parameters, averaged over twenty runs.

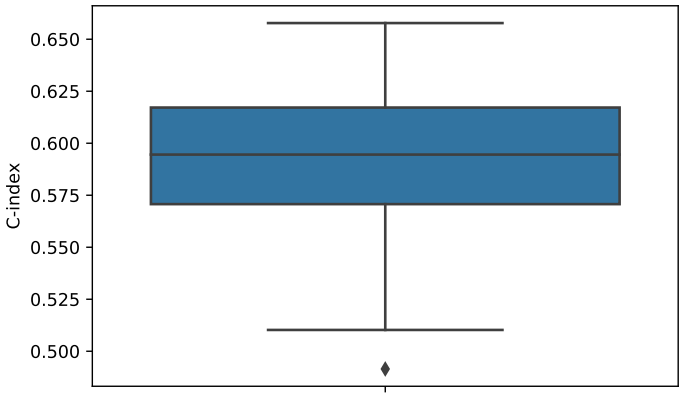



Figure 4.3 A box plot of the test error of the most promising set of hyper-parameters, over seventeen runs (three dropped due to under-performance on training or validation set).

We compare against two baselines. The first ones are the manual alerts set by the care workers. We have a history of four levels of urgency (none, low, medium, high) for every patient and every day. As done with the latent risk modeled by the neural network, these alerts yield a natural ranking that can be used to compute a concordance index. These alerts achieve concordance indices of 48.7%, 50.7%, and 51.1% on respectively the training, the validation, and the test set. Note that these alerts were not set based on training data, so these results could be aggregated. However, we provide them separated so that the test error could be compared to the neural network on the same data points. The second baseline is a linear survival regression model, where data points are the vital signs for every patient and every day, as if they were independent (no time dependencies are taken into account). This model achieves a training error (on training and validation sets combined) of 49.7% and a test error of 48.0%. Even if these two baselines are simplistic, the fact that they do not achieve better than a random draw shows the difficulty of the problem.

We implemented the neural network in PyTorch [91], used Lifelines [92] to compute the linear baseline and the C-Index, made use of Numpy [93], Scipy [94], Pandas [95], Scikit-Learn [96], IPython [97], and Jupyter [98] for pre-processing of the data and post-processing of the results, and rendered the figures with Matplotlib [99] and Seaborn [100].

### 4.6 Discussion

The high stochasticity of the problem makes training hard and long, therefore making comparison between different neural network architectures and training procedures either expensive (through averaging) or unfair (some configuration randomly performing abnormally well or poorly). As stated in the previous section, passing the patient static information as the first hidden vector of the recurrent neural network (as opposed to just passing a null vector), as proposed in Section 4.4, did not improve the performances and was therefore not selected through hyper-parameter search. Further inquiring should be done to find out if a better model could be obtained using the static patient information. Our hypothesis is that this data does have predictive power for this task, but that the specific part of the neural network responsible for it failed to learn, due to optimization hardships. Indeed, not only this part of the model adds more parameters (layers) to train, these parameters are also updated less frequently in the truncated variant of back-propagation through time. Potential directions could be to focus more on the training of this part of the model (for instance through pertaining), or to include this data at every time-step (explicitly or through an attention mechanism).

Our results do not show that a linear model could not perform as well as the neural network,

as less effort was given to this model. Improving the linear baseline would however mean additional engineering of the data to include time dependencies, patient static information, and perform feature selection.

These results suggest that combining, even weak, signals from remote monitoring in the homecare context can outperform simple manual baselines which open the door to better models.

Nonetheless, a self-fulfilling prophecy problem could occur with a better prediction accuracy. Care workers using machine learning generated alerts would prevent events from happening and reduce the observations labeled as events. A potential alternative is to ask care workers if the prediction was useful or not, *i.e.*, if they want such prediction happening again in the future. While far from perfect, this methodology has the advantage of enabling model retraining as the data is gathered. More research is required to evaluate the risk of this problem and performance of the proposed alternative.

In addition, further research is needed to better understand the factors that contribute to higher risk days for home telemonitoring patients. Indeed, the black-box nature of neural networks makes them difficult to implement in the health care industry since physicians and other care workers generally want to understand why an adverse event probability is predicted. For example, what action should a care worker take if manual alerts are triggered but neural networks say that nothing is happening? The model performance suggests that no action should be taken, but this is clearly a difficult call.

# CHAPTER 5 ARTICLE 2 - MACHINE LEARNING FOR COMBINATORIAL OPTIMIZATION: A METHODOLOGICAL TOUR D'HORIZON

Authors: Yoshua Bengio, Andrea Lodi, Antoine Prouvost
Published in *European Journal of Operational Research* [37].

**Abstract** This paper surveys the recent attempts, both from the machine learning and operations research communities, at leveraging machine learning to solve combinatorial optimization problems. Given the hard nature of these problems, state-of-the-art algorithms rely on handcrafted heuristics for making decisions that are otherwise too expensive to compute or mathematically not well defined. Thus, machine learning looks like a natural candidate to make such decisions in a more principled and optimized way. We advocate for pushing further the integration of machine learning and combinatorial optimization and detail a methodology to do so. A main point of the paper is seeing generic optimization problems as data points and inquiring what is the relevant distribution of problems to use for learning on a given task.

## 5.1 Introduction

Operations research, also referred to as prescriptive analytics, started in the second world war as an initiative to use mathematics and computer science to assist military planners in their decisions [101]. Nowadays, it is widely used in the industry, including but not limited to transportation, supply chain, energy, finance, and scheduling. In this paper, we focus on discrete optimization problems formulated as integer constrained optimization, *i.e.*, with integral or binary variables (called decision variables). While not all such problems are hard to solve (*e.g.*, shortest path problems), we concentrate on CO problems (NP-hard). This is bad news, in the sense that, for those problems, it is considered unlikely that an algorithm whose running time is polynomial in the size of the input exists. However, in practice, CO algorithms can solve instances with up to millions of decision variables and constraints.

*How is it possible to solve NP-hard problems in practical time?* Let us look at the example of the TSP, a NP-hard problem defined on a graph where we are searching for a cycle of minimum length visiting once and only once every node. A particular case is that of the *Euclidian* TSP. In this version, each node is assigned coordinates in a plane,[1] and the cost on an edge

[1] Or more generally in a vector space of arbitrary dimension.

connecting two nodes is the Euclidian distance between them. While theoretically as hard as the general TSP, good approximate solution can be found more efficiently in the Euclidian case by leveraging the *structure* of the graph [102, Chapter 6.4.7]. Likewise, diverse types of problems are solved by leveraging their special structure. Other algorithms, designed to be general, are found in hindsight to be empirically more efficient on particular problems types. The scientific literature covers the rich set of techniques researchers have developed to tackle different CO problems. An expert will know how to further refine algorithm parameters to different behaviors of the optimization process, thus extending this knowledge with unwritten intuition. These techniques, and the parameters controlling them, have been collectively *learned* by the community to perform on the inaccessible distribution of problem instances deemed valuable. The focus of this paper is on CO algorithms that automatically perform learning on a chosen implicit distribution of problems. Incorporating ML components in the algorithm can achieve this.

Conversely, ML focuses on performing a task given some (finite and usually noisy) data. It is well suited for natural signals for which no clear mathematical formulation emerges because the true data distribution is not known analytically, such as when processing images, text, voice or molecules, or with recommender systems, social networks or financial predictions. Most of the times, the learning problem has a statistical formulation that is solved through mathematical optimization. Recently, dramatic progress has been achieved with deep learning, an ML sub-field building large parametric approximators by composing simpler functions. Deep learning excels when applied in high dimensional spaces with a large number of data points.

### 5.1.1 Motivation

From the CO point of view, machine learning can help improve an algorithm on a distribution of problem instances in two ways. On the one side, the researcher assumes expert knowledge[2] about the optimization algorithm, but wants to replace some heavy computations by a fast approximation. Learning can be used to build such approximations in a generic way, *i.e.*, without the need to derive new explicit algorithms. On the other side, expert knowledge may not be sufficient and some algorithmic decisions may be unsatisfactory. The goal is therefore to explore the space of these decisions, and learn out of this experience the best performing behavior (policy), hopefully improving on the state of the art. Even though ML is approximate, we will demonstrate through the examples surveyed in this paper that this does not systematically mean that incorporating learning will compromise overall theoretical

[2]Theoretical and/or empirical.

guarantees. From the point of view of using ML to tackle a combinatorial problem, CO can decompose the problem into smaller, hopefully simpler, learning tasks. The CO structure therefore acts as a relevant prior for the model. It is also an opportunity to leverage the CO literature, notably in terms of theoretical guarantees (*e.g.*, feasibility and optimality).

### 5.1.2 Setting

Imagine a delivery company in Montreal that needs to solve TSPs. Every day, the customers may vary, but usually, many are downtown and few on top of the Mont Royal mountain. Furthermore, Montreal streets are laid on a grid, making the distances close to the $\ell_1$ distance. How close? Not as much as Phoenix, but certainly more than Paris. The company does not care about solving all possible TSPs, but only *theirs*. Explicitly defining what makes a TSP a likely one for the company is tedious, does not scale, and it is not clear how it can be leveraged when explicitly writing an optimization algorithm. We would like to automatically specialize TSP algorithms for this company.

The true probability distribution of likely TSPs in the Montreal scenario is defining the instances on which we would like our algorithm to perform well. This is unknown, and cannot even be mathematically characterized in an explicit way. Because we do not know what is in this distribution, we can only learn an algorithm that performs well on a finite set of TSPs sampled from this distribution (for instance, a set of historical instances collected by the company), thus implicitly incorporating the desired information about the distribution of instances. As a comparison, in traditional ML tasks, the true distribution could be that of all possible images of cats, while the training distribution is a finite set of such images. The challenge in learning is that an algorithm that performs well on problem instances used for learning may not work properly on other instances from the true probability distribution. For the company, this would mean the algorithm only does well on past problems, but not on the future ones. To control this, we monitor the performance of the learned algorithm over another independent set of *unseen* problem instances. Keeping the performances similar between the instances used for learning and the unseen ones is known in ML as *generalizing*. Current ML algorithms can generalize to examples from the same distribution, but tend to have more difficulty generalizing out-of-distribution (although this is a topic of intense research in ML), and so we may expect CO algorithms that leverage ML models to fail when evaluated on unseen problem instances that are too far from what has been used for training the ML predictor. As previously motivated, it is also worth noting that traditional CO algorithms might not even work consistently across all possible instances of a problem family, but rather tend to be more adapted to particular structures of problems, *e.g.*, Euclidean TSPs.

Finally, the implicit knowledge extracted by ML algorithms is complementary to the hard-won explicit expertise extracted through CO research. Rather, it aims to augment and automate the unwritten expert intuition (or lack of) on various existing algorithms. Given that these problems are highly structured, we believe it is relevant to augment solving algorithms with machine learning – and especially deep learning to address the high dimensionality of such problems.

In the following, we survey the attempts in the literature to achieve such automation and augmentation, and we present a methodological overview of those approaches. In light of the current state of the field, the literature we survey is exploratory, *i.e.*, we aim at highlighting promising research directions in the use of ML within CO, instead of reporting on already mature algorithms.

### 5.1.3 Outline

We have introduced the context and motivations for building combinatorial optimization algorithms together with machine learning. The remainder of this paper is organized as follows. Section 5.2 provides minimal prerequisites in combinatorial optimization, machine learning, deep learning, and reinforcement learning necessary to fully grasp the content of the paper. Section 5.3 surveys the recent literature and derives two distinctive, orthogonal, views: Section 5.3.1 shows how machine learning policies can either be learned by imitating an expert or discovered through experience, while Section 5.3.2 discusses the interplay between the ML and CO components. Section 5.4 details the theoretical learning framework for using ML inside CO algorithms. Section 5.5 pushes further the reflection and brings to the fore some methodological points. In Section 5.6, we detail critical practical challenges of the field. Finally, some conclusions are drawn in Section 5.7.

## 5.2 Preliminaries

In this section, we give a basic (sometimes rough) overview of combinatorial optimization and machine learning, with the unique aim of introducing concepts that are strictly required to understand the remainder of the paper.

### 5.2.1 Combinatorial Optimization

Without loss of generality, a CO problem can be formulated as a constrained min-optimization program. Constraints model natural or imposed restrictions of the problem, variables define the decisions to be made, while the objective function, generally a cost to be minimized,

defines the measure of the quality of every feasible assignment of values to variables. If the objective and constraints are linear, the problem is called a linear programming (LP) problem. If, in addition, some variables are also restricted to only assume integer values, then the problem is a mixed-integer linear programming (MILP) problem.

The set of points that satisfy the constraints is the feasible region. Every point in that set (often referred to as a feasible solution) yields an upper bound on the objective value of the optimal solution. Exact solving is an important aspect of the field, hence a lot of attention is also given to find lower bounds to the optimal cost. The tighter the lower bounds, with respect to the optimal solution value, the higher the chances that the current algorithmic approaches to tackle MILPs described in the following could be successful, *i.e.*, effective if not efficient.

Linear and mixed-integer linear programming problems are the workhorse of CO because they can model a wide variety of problems and are the best understood, *i.e.*, there are reliable algorithms and software tools to solve them. We give them special considerations in this paper but, of course, they do not represent the entire CO, mixed-integer nonlinear programming being a rapidly expanding and very significant area both in theory and in practical applications. With respect to complexity and solution methods, LP is polynomially solved, well solved, in theory and in practice, through the simplex algorithm or interior points methods. Mixed-integer linear programming, on the other hand, is an NP-hard problem, which does not make it hopeless. Indeed, it is easy to see that the complexity of MILP is associated with the integrality requirement on (some of) the variables, which makes the MILP feasible region nonconvex. However, dropping the integrality requirement (i) defines a proper relaxation of MILP (*i.e.*, an optimization problem whose feasible region contains the MILP feasible region), which (ii) happens to be an LP problem, *i.e.*, polynomially solvable. This immediately suggests the algorithmic line of attack that is used to solve MILP through a whole ecosystem of B&B techniques to perform implicit enumeration. Branch and bound implemements a divide-and-conquer type of algorithm representable by a search tree in which, at every node, an LP relaxation of the problem (possibly augmented by branching decisions, see below) is efficiently computed. If the relaxation is infeasible, or if the solution of the relaxation is naturally (mixed-)integer, *i.e.*, MILP feasible, the node does not need to be expanded. Otherwise, there exists at least one variable, among those supposed to be integer, taking a fractional value in the LP solution and that variable can be chosen for branching (enumeration), *i.e.*, by restricting its value in such a way that two child nodes are created. The two child nodes have disjoint feasible regions, none of which contains the solution of the previous LP relaxation. We use Figure 5.1 to illustrate the B&B algorithm for a minimization MILP. At the root node in the figure, the variable $x_2$ has a fractional value in the LP solution

(not represented), thus branching is done on the floor (here zero) and ceiling (here one) of this value. When an integer solution is found, we also get an upper bound (denoted as $\overline{z}$) on the optimal solution value of the problem. At every node, we can then compare the solution value of the relaxation (denoted as $\underline{z}$) with the minimum upper bound found so far, called the incumbent solution value. If the latter is smaller than the former for a specific node, no better (mixed-)integer solution can be found in the sub-tree originated by the node itself, and it can be pruned.

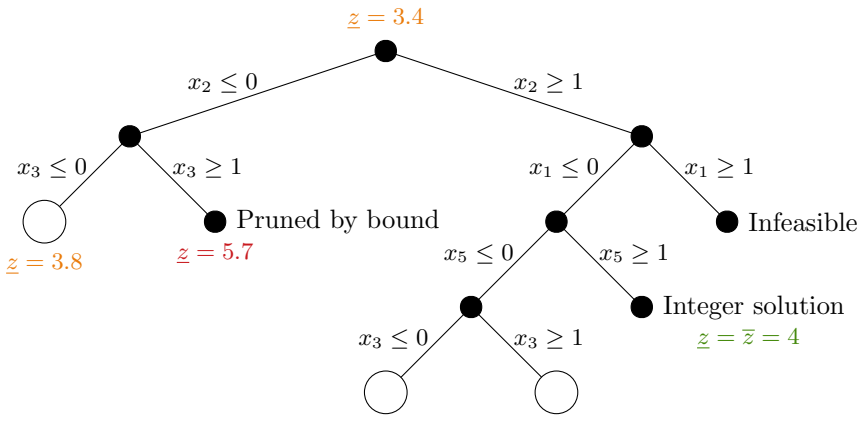



Figure 5.1 A branch-and-bound tree for MILP. The LP relaxation is computed at every node (only partially shown in the figure). Nodes still open for exploration are represented as blank.

All commercial and noncommercial MILP solvers enhance the above enumeration framework with the extensive use of cutting planes, *i.e.*, valid linear inequalities that are added to the original formulation (especially at the root of the B&B tree) in the attempt of strengthening its LP relaxation. The resulting framework, referred to as the branch-and-cut algorithm, is then further enhanced by additional algorithmic components, preprocessing and primal heuristics being the most crucial ones. The reader is referred to *Integer Programming* [14] and *Integer Programming* [103] for extensive textbooks on MILP and to "MIP Computation" [104] for a detailed description of the algorithmic components of the MILP solvers.

We end the section by noting that there is a vast literature devoted to (primal) heuristics, *i.e.*, algorithms designed to compute "good in practice" solutions to CO problems without optimality guarantee. Although a general discussion on them is outside the scope here, those heuristic methods play a central role in CO and will be considered in specific contexts in the present paper. The interested reader is referred to "Heuristics in Mixed Integer Programming" [13] and *Handbook of metaheuristics* [105].

### 5.2.2 Machine Learning

**Supervised learning** In supervised learning, a set of input (features) / target pairs is provided and the task is to find a function that for every input has a predicted output as close as possible to the provided target. Finding such a function is called learning and is solved through an optimization problem over a family of functions. The loss function, *i.e.*, the measure of discrepancy between the output and the target, can be chosen depending on the task (regression, classification, *etc.*) and on the optimization methods. However, this approach is not enough because the problem has a statistical nature. It is usually easy enough to achieve a good score on the given examples but one wants to achieve a good score on unseen examples (test data). This is known as generalization.

Mathematically speaking, let $X$ and $Y$, following a joint probability distribution $P$, be random variables representing the input features and the target. Let $\ell$ be the per sample loss function to minimize, and let $\{f_\theta \,|\, \theta \in \mathbb{R}^p\}$ be the family of ML models (parametric in this case) to optimize over. The supervised learning problem is framed as

$$\min_{\theta \in \mathbb{R}^p} \mathbb{E}_{X,Y \sim P}\, \ell(Y, f_\theta(X)). \tag{5.1}$$

For instance, $f_\theta$ could be a linear model with weights $\theta$ that we wish to *learn*. The loss function $\ell$ is task dependent (*e.g.*, classification error) and can sometimes be replaced by a surrogate one (*e.g.*, a differentiable one). The probability distribution is unknown and inaccessible. For example, it can be the probability distribution of all natural images. Therefore, it is approximated by the empirical probability distribution over a finite dataset $D_{train} = \{(x_i, y_i)\}_i$ and the optimization problem solved is

$$\min_{\theta \in \mathbb{R}^p} \sum_{(x,y) \in D_{train}} \frac{1}{|D_{train}|} \ell(y, f_\theta(x)). \tag{5.2}$$

A model is said to generalize, if low objective values of (5.2) translate in low objective values of (5.1). Because (5.1) remains inaccessible, we estimate the generalization error by evaluating the trained model on a separate test dataset $D_{test}$ with

$$\sum_{(x,y) \in D_{test}} \frac{1}{|D_{test}|} \ell(y, f_\theta(x)). \tag{5.3}$$

If a model (*i.e.*, a family of functions) can represent many different functions, the model is said to have high capacity and is prone to overfitting: doing well on the training data but not generalizing to the test data. Regularization is anything that can improve the test score

at the expense of the training score and is used to restrict the practical capacity of a model. On the contrary, if the capacity is too low, the model underfits and performs poorly on both sets. The boundary between overfitting and underfitting can be estimated by changing the effective capacity (the richness of the family of functions reachable by training): below the critical capacity one underfits and test error decreases with increases in capacity, while above that critical capacity one overfits and test error increases with increases in capacity.

Selecting the best among various trained models cannot be done on the test set. Selection is a form of optimization, and doing so on the test set would bias the estimator in (5.2). This is a common form of data dredging, and a mistake to be avoided. To perform model selection, a validation dataset $D_{valid}$ is used to estimate the generalization error of different ML models. Model selection can be done based on these estimates, and the final unbiased generalization error of the selected model can be computed on the test set. The validation set is therefore often used to select effective capacity, *e.g.*, by changing the amount of training, the number of parameters $\theta$, and the amount of regularization imposed to the model.

**Unsupervised learning** In unsupervised learning, one does not have targets for the task one wants to solve, but rather tries to capture some characteristics of the joint distribution of the observed random variables. The variety of tasks include density estimation, dimensionality reduction, and clustering. Because unsupervised learning has received so far little attention in conjunction with CO and its immediate use seems difficult, we are not discussing it any further. The reader is referred to *Pattern Recognition and Machine Learning* [19], *Machine Learning: A Probabilistic Perspective* [20] and *Deep learning* [9] for textbooks on machine learning.

**Reinforcement learning** In RL, an agent interacts with an environment through a MDP, as illustrated in Figure 5.2. At every time step, the agent is in a given state of the environment and chooses an action according to its (possibly stochastic) policy. As a result, it receives from the environment a reward and enters a new state. The goal in RL is to train the agent to maximize the expected sum of future rewards, called the return. For a given policy, the expected return given a current state (resp. state and action pair) is known as the value function (resp. state action value function). Value functions follow the Bellman equation, hence the problem can be formulated as dynamic programming, and solved approximately. The dynamics of the environment need not be known by the agent and are learned directly or indirectly, yielding an exploration *vs* exploitation dilemma: choosing between exploring new states for refining the knowledge of the environment for possible long-term improvements, or exploiting the best-known scenario learned so far (which tends to be in already visited or

predictable states).

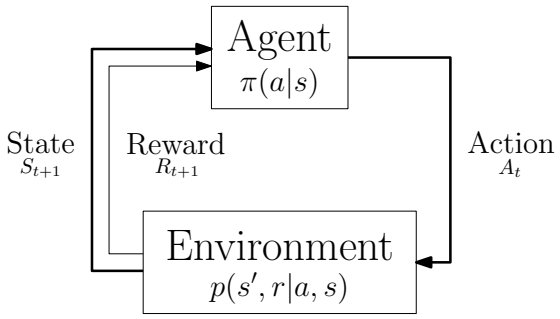



Figure 5.2 The Markov decision process associated with reinforcement learning, modified from *Reinforcement Learning: An Introduction* [21]. The agent behavior is defined by its policy $\pi$, while the environment evolution is defined by the dynamics $p$. Note that the reward is not necessary to define the evolution and is provided only as a learning mechanism for the agent. Actions, states, and rewards are random variables in the general framework.

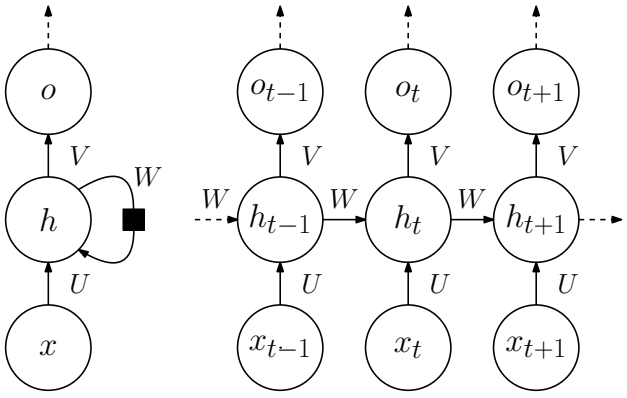
The state should fully characterize the environment at every step, in the sense that future states only depend on past states via the current state (the Markov property). When this is not the case, similar methods can be applied but we say that the agent receives an *observation* of the state. The Markov property no longer holds and the MDP is said to be partially observable.

Defining a reward function is not always easy. Sometimes one would like to define a very sparse reward, such as 1 when the agent solves the problem, and 0 otherwise. Because of its underlying dynamic programming process, RL is naturally able to credit states/actions that lead to future rewards. Nonetheless, the aforementioned setting is challenging as it provides no learning opportunity until the agent (randomly, or through advanced approaches) solves the problem. Furthermore, when the policy is approximated (for instance, by a linear function), the learning is not guaranteed to converge and may fall into local minima. For example, an autonomous car may decide not to drive anywhere for fear of hitting a pedestrian and receiving a dramatic negative reward. These challenges are strongly related to the aforementioned exploration dilemma. The reader is referred to *Reinforcement Learning: An Introduction* [21] for an extensive textbook on the matter.

**Deep learning** Deep learning is a successful method for building parametric composable functions in high dimensional spaces. In the case of the simplest neural network architecture, the feedforward neural network (also called an MLP), the input data is successively passed through a number of layers. For every layer, an affine transformation is applied on

the input vector, followed by a non-linear scalar function (named activation function) applied element-wise. The output of a layer, called intermediate activations, is passed on to the next layer. All affine transformations are independent and represented in practice as different matrices of coefficients. They are learned, *i.e.*, optimized over, through SGD, the optimization algorithm used to minimize the selected loss function. The stochasticity comes from the limited number of data points used to compute the loss before applying a gradient update. In practice, gradients are computed using reverse mode automatic differentiation, a practical algorithm based on the chain rule, also known as back-propagation. Deep neural networks can be difficult to optimize, and a large variety of techniques have been developed to make the optimization behave better, often by changing architectural designs of the network. Because neural networks have dramatic capacities, *i.e.*, they can essentially match any dataset, thus being prone to overfitting, they are also heavily regularized. Training them by SGD also regularizes them because of the noise in the gradient, making neural networks generally robust to overfitting issues, even when they are very large and would overfit if trained with more aggressive optimization. In addition, many hyper-parameters exist and different combinations are evaluated (known as hyper-parameters optimization). Deep learning also sets itself apart from more traditional ML techniques by taking as inputs all available raw features of the data, *e.g.*, all pixels of an image, while traditional ML typically requires to engineer a limited number of domain-specific features.

Deep learning researchers have developed different techniques to tackle this variety of structured data in a manner that can handle variable-size data structures, *e.g.*, variable-length sequences. In this paragraph, and in the next, we present such state-of-the-art techniques. These are complex topics, but lack of comprehension does not hinder the reading of the paper. At a high level, it is enough to comprehend that these are architectures designed to handle different structures of data. Their usage, and in particular the way they are learned, remains very similar to plain feedforward neural networks introduced above. The first architectures presented are the RNNs. These models can operate on sequence data by *sharing parameters* across different sequence steps. More precisely, a same neural network block is successively applied at every step of the sequence, *i.e.*, with the same architecture and parameter values at each time step. The specificity of such a network is the presence of recurrent layers: layers that take as input both the activation vector of the previous layer and its own activation vector on the preceding sequence step (called a hidden state vector), as illustrated in Figure 5.3.




(a) Folded (b) Unfolded

Figure 5.3 A vanilla RNN modified from *Deep learning* [9]. On the left, the black square indicates a one step delay. On the right, the same RNN is shown unfolded. Three sets $U$, $V$, and $W$ of parameters are represented and re-used at every time step.

Another important size-invariant technique are *attention mechanisms*. They can be used to process data where each data point corresponds to a set. In that context, parameter sharing is used to address the fact that different sets need not to be of the same size. Attention is used to query information about all elements in the set, and merge it for downstream processing in the neural network, as depicted in Figure 5.4. An affinity function takes as input the query (which represents any kind of contextual information which informs where attention should be concentrated) and a representation of an element of the set (both are activation vectors) and outputs a scalar. This is repeated over all elements in the set for the same query. Those scalars are normalized (for instance with a softmax function) and used to define a weighted sum of the representations of elements in the set that can, in turn, be used in the neural network making the query. This form of content-based soft attention was introduced by Bahdanau *et al.* [23]. A general explanation of attention mechanisms is given by Vaswani *et al.* [106]. Attention can be used to build GNNs, *i.e.*, neural networks able to process graph structured input data, as done by Veličković *et al.* [107]. In this architecture, every node attends over the set of its neighbors. The process is repeated multiple times to gather information about nodes further away. GNNs can also be understood as a form of message passing [108].

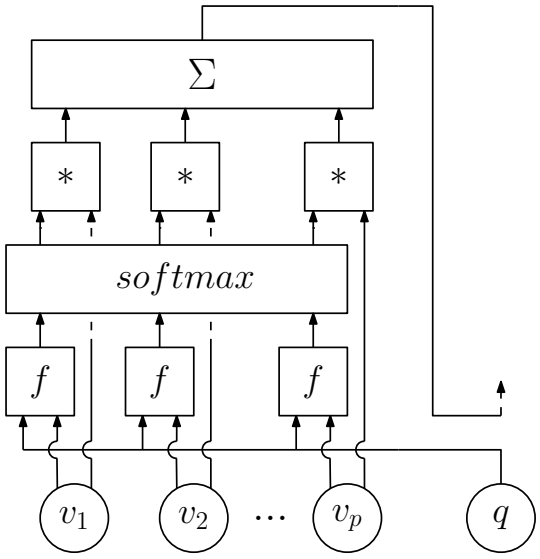



Figure 5.4 A vanilla attention mechanism where a query $q$ is computed against a set of values $(v_i)_i$. An affinity function $f$, such as a dot product, is used on query and value pairs. If it includes some parameters, the mechanism can be learned.

Deep learning and back-propagation can be used in supervised, unsupervised, or reinforcement learning. The reader is referred to *Deep learning* [9] for a machine learning textbook devoted to deep learning.

## 5.3 Recent approaches

We survey different uses of ML to help solve combinatorial optimization problems and organize them along two orthogonal axes. First, in Section 5.3.1 we illustrate the two main motivations for using learning: approximation and discovery of new policies. Then, in Section 5.3.2, we show examples of different ways to combine learned and traditional algorithmic elements.

### 5.3.1 Learning methods

This section relates to the two motivations reported in Section 5.1.1 for using ML in CO. In some works, the researcher assumes theoretical and/or empirical knowledge about the decisions to be made for a CO algorithm, but wants to alleviate the computational burden by approximating some of those decisions with machine learning. On the contrary, we are also motivated by the fact that, sometimes, expert knowledge is not satisfactory and the researcher wishes to find better ways of making decisions. Thus, ML can come into play to

train a model through trial and error reinforcement learning.

We frame both these motivations in the state/action MDP framework introduced in section 5.2.2, where the environment is the internal state of the algorithm. We care about learning algorithmic decisions utilized by a CO algorithm and we call the function making the decision a *policy*, that, given all available information,[3] returns (possibly stochastically) the action to be taken. The policy is the function that we want to learn using ML and we show in the following how the two motivations naturally yield two learning settings. Note that the case where the length of the trajectory of the MDP has value 1 is a common edge case (called the bandit setting) where this formulation can seem excessive, but it nonetheless helps comparing methods.

In the case of using ML to approximate decisions, the policy is often learned by *imitation learning*, thanks to *demonstrations*, because the expected behavior is shown (demonstrated) to the ML model by an expert (also called oracle, even though it is not necessarily optimal), as shown in Figure 5.5. In this setting, the learner is not trained to optimize a performance measure, but to *blindly* mimic the expert.

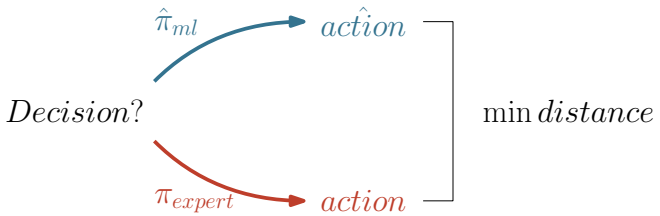



Figure 5.5 In the demonstration setting, the policy is trained to reproduce the action of an expert policy by minimizing some discrepancy in the action space.

In the case where one cares about discovering new policies, *i.e.*, optimizing an algorithmic decision function from the ground up, the policy may be learned by reinforcement learning through *experience*, as shown in Figure 5.6. Even though we present the learning problem under the fundamental MDP of RL, this does not constrain one to use the major RL algorithms (approximate dynamic programming and policy gradients) to maximize the expected sum of rewards. Alternative optimization methods, such as bandit algorithms, genetic algorithms, direct/local search, can also be used to solve the RL problem.[4]

[3] A *state* if the information is sufficient to fully characterize the environment at that time in a Markov decision process setting.

[4] In general, identifying which algorithm will perform best is an open research question unlikely to have a simple answer, and is outside of the scope of the methodology presented here.


Figure 5.6 When learning through a reward signal, no expert is involved; only maximizing the expected sum of future rewards (the return) matters.

It is critical to understand that in the imitation setting, the policy is learned through supervised targets provided by an expert for every action (and without a reward), whereas in the experience setting, the policy is learned from a reward (possibly delayed) signal using RL (and without an expert). In imitation, the agent is taught *what* to do, whereas in RL, the agent is encouraged to quickly *accumulate* rewards. The distinction between these two settings is far more complex than the distinction made here. We explore some of this complexity, including their strengths and weaknesses, in Section 5.5.1.

In the following, few papers demonstrating the different settings are surveyed.

#### 5.3.1.1 Demonstration

Baltean-Lugojan *et al.* [109] use a neural network to approximate the lower bound improvement generated by tightening the current relaxation via cutting planes (cuts, for short). More precisely, Baltean-Lugojan *et al.* [109] consider non-convex quadratic programming problems and aim at approximating the associated semidefinite programming (SDP) relaxation, known to be strong but time-consuming, by a linear program. A straightforward way of doing that is to iteratively add (linear) cutting planes associated with negative eigenvalues, especially considering small-size (square) submatrices of the original quadratic objective function. That approach has the advantage of generating sparse cuts[5] but it is computationally challenging because of the exponential number of those submatrices and because of the difficulty of finding the right metrics to select among the violated cuts. The authors propose to solve SDPs to compute the bound improvement associated with considering specific submatrices, which is also a proxy on the quality of the cuts that could be separated from the same submatrices. In this context, supervised (imitation) learning is applied offline to approximate the objective value of the SDP problem associated with a submatrix selection and, afterward, the model can be rapidly applied to select the most promising submatrices without the very significant computational burden of solving SDPs. Of course, the rational is that the most promising submatrices correspond to the most promising cutting planes and Baltean-Lugojan *et al.*

[5]The reader is referred to "Theoretical Challenges towards Cutting-Plane Selection" [110] for a detailed discussion on the importance of sparse cutting planes in MILP.

[109] train a model to estimate the objective of an SDP problem only in order to decide to add the most promising cutting planes. Hence, cutting plane selection is the ultimate policy learned.

Another example of demonstration is found in the context of branching policies in B&B trees of MILPs. The choice of variables to branch on can dramatically change the size of the B&B tree, hence the solving time. Among many heuristics, a well-performing approach is *strong branching* [111]. Namely, for every branching decision to be made, strong branching performs a one step look-ahead by tentatively branching on many candidate variables, computes the LP relaxations to get the potential lower bound improvements, and eventually branches on the variable providing the best improvement. Even if not all variables are explored, and the LP value can be approximated, this is still a computationally expensive strategy. For these reasons, Marcos Alvarez *et al.* [112, 113] use a special type of decision tree (a classical model in supervised learning) to approximate strong branching decisions using supervised learning. Khalil *et al.* [59] propose a similar approach, where a linear model is learned on the fly for every instance by using strong branching at the top of the tree, and eventually replacing it by its ML approximation. The linear approximator of strong branching introduced by Marcos Alvarez *et al.* [114] is learned in an active fashion: when the estimator is deemed unreliable, the algorithm falls back to true strong branching and the results are then used for both branching and learning. In all the branching algorithms presented here, inputs to the ML model are engineered as a vector of fixed length with static features descriptive of the instance, and dynamic features providing information about the state of the B&B process. Gasse *et al.* [32] use a neural network to learn an offline approximation to strong branching, but, contrary to the aforementioned papers, the authors use a raw exhaustive representation (*i.e.*, they do not discard nor aggregate any information) of the sub-problem associated with the current branching node as input to the ML model. Namely, an MILP sub-problem is represented as a bipartite graph on variables and constraints, with edges representing non-zero coefficients in the constraint matrix. Each node is augmented with a set of features to fully describe the sub-problem, and a GNN is used to build an ML approximator able to process this type of structured data. Node selection, *i.e.*, deciding on the next branching node to explore in a B&B tree, is also a critical decision in MILP. He *et al.* [115] learn a policy to select among the open branching nodes the one that contains the optimal solution in its sub-tree. The training algorithm is an online learning method collecting expert behaviors throughout the entire learning phase. The reader is referred to "On Learning and Branching" [36] for an extended survey on learning and branching in MILPs.

Branch and bound is a technique not limited to MILP and can be use for general tree search. Hottung *et al.* [116] build a tree search procedure for the container pre-marshalling problem

in which they aim to learn, not only a branching policy (similar in principle to what has been discussed in the previous paragraph), but also a value network to estimate the value of partial solutions and used for bounding. The authors leverage a form of convolutional neural network (CNN)[6] for both networks and train them in a supervised fashion using pre-computed solutions of the problem. The resulting algorithm is heuristic due the approximations made while bounding.

As already mentioned at the beginning of Section 5.3.1, learning a policy by demonstration is identical to supervised learning, where training pairs of input state and target actions are provided by the expert. In the simplest case, expert decisions are collected beforehand, but more advanced methods can collect them online to increase stability as previously shown by Marcos Alvarez *et al.* [114] and He *et al.* [115].

#### 5.3.1.2 Experience

Considering the TSP on a graph, it is easy to devise a greedy heuristic that builds a tour by sequentially picking the nodes among those that have not been visited yet, hence defining a permutation. If the criterion for selecting the next node is to take the closest one, then the heuristic is known as the nearest neighbor. This simple heuristic has poor practical performance and many other heuristics perform better empirically, *i.e.*, build cheaper tours. Selecting the nearest node is a fair intuition but turns out to be far from satisfactory. Khalil *et al.* [27] suggest learning the criterion for this selection. They build a greedy heuristic framework, where the node selection policy is learned using a GNN [117], a type of neural network able to process input graphs of any finite size by a mechanism of message passing [108]. For every node to select, the authors feed to the network the graph representation of the problem – augmented with features indicating which of the nodes have already been visited – and receive back an action value for every node. Action values are used to train the network through RL (Q-learning in particular) and the partial tour length is used as a reward.

This example does not do justice to the rich TSP literature that has developed far more advanced algorithms performing orders of magnitude better than ML ones. Nevertheless, the point we are trying to highlight here is that given a fixed context, and a decision to be made, ML can be used to discover new, potentially better performing policies. Even on state-of-the-art TSP algorithms (*i.e.*, when exact solving is taken to its limits), many decisions are made in heuristic ways, *e.g.*, cutting plane selection, thus leaving room for ML to assist in

[6] A type of neural network, usually used on image input, that leverages parameter sharing to extract local information.

making these decisions.

Once again, we stress that learning a policy by experience is well described by the MDP framework of reinforcement learning, where an agent maximizes the return (defined in Section 5.2.2). By matching the reward signal with the optimization objective, the goal of the learning agent becomes to solve the problem, without assuming any expert knowledge. Some methods that were not presented as RL can also be cast in this MDP formulation, even if the optimization methods are not those of the RL community. For instance, part of the CO literature is dedicated to automatically build specialized heuristics for different problems. The heuristics are build by orchestrating a set of moves, or subroutines, from a pre-defined domain-specific collections. For instance, to tackle bipartite boolean quadratic programming problems, Karapetyan *et al.* [118] represent this orchestration as a Markov chain where the states are the subroutines. One Markov chain is parametrized by its transition probabilities. Mascia *et al.* [119], on the other hand, define valid succession of moves through a grammar, where words are moves and sentences correspond to heuristics. The authors introduce a parametric space to represent sentences of a grammar. In both cases, the setting is very close to the MDP of RL, but the parameters are learned though direct optimization of the performances of their associated heuristic through so-called *automatic configuration tools* (usually based on genetic or local search, and exploiting parallel computing). Note that the learning setting is rather simple as the parameters do not adapt to the problem instance, but are fixed for various clusters. From the ML point of view, this is equivalent to a piece-wise constant regression. If more complex models were to be used, direct optimization may not scale adequately to obtain good performances. The same approach to building heuristics can be brought one level up if, instead of orchestrating sets of moves, it arranges predefined heuristics. The resulting heuristic is then called a *hyper-heuristic*. Özcan *et al.* [120] build a hyper-heuristic for examination timetabling by learning to combine existing heuristics. They use a bandit algorithm, a stateless form of RL [21, Chapter 2], to learn online a value function for each heuristic.

We close this section by noting that demonstration and experience are not mutually exclusive and most learning tasks can be tackled in both ways. In the case of selecting the branching variables in an MILP branch-and-bound tree, one could adopt anyone of the two prior strategies. On the one hand, Khalil *et al.* [59] and Marcos Alvarez *et al.* [112–114] estimate that strong branching is an effective branching strategy but computationally too expensive and build a machine learning model to approximate it. On the other hand, one could believe that no branching strategy is good enough and try to learn one from the ground up, for instance through reinforcement learning as suggested (but not implemented) by Khalil *et al.*

[59]. An intermediary approach is proposed by Liberto *et al.* [121]. The authors recognize that, among traditional variable selection policies, the ones performing well at the top of the B&B tree are not necessarily the same as the ones performing well deeper down. Hence, the authors learn a model to dynamically switch among predefined policies during B&B based on the current state of the tree. While this seems like a case of imitation learning, given that traditional branching policies can be thought of as experts, this is actually not the case. In fact, the model is not learning *from* any expert, but really learning to choose between pre-existing policies. This is technically not a branching variable selection, but rather a branching heuristic selection policy. Each sub-tree is represented by a vector of handcrafted features, and a clustering of these vectors is performed. Similarly to what was detailed in the previous paragraph about the work of Karapetyan *et al.* [118] and Mascia *et al.* [119], automatic configuration tools are then used to assign the best branching policy to each cluster. When branching at a given node, the cluster the closest to the current sub-tree is retrieved, and its assigned policy is used.

### 5.3.2 Algorithmic structure

In this section, we survey how the learned policies (whether from demonstration or experience) are combined with traditional CO algorithms, *i.e.*, considering ML and explicit algorithms as building blocks, we survey how they can be laid out in different templates. The three following sections are not necessarily disjoint nor exhaustive but are a natural way to look at the literature.

#### 5.3.2.1 End to end learning

A first idea to leverage machine learning to solve discrete optimization problems is to train the ML model to output solutions directly from the input instance, as shown in Figure 5.7.

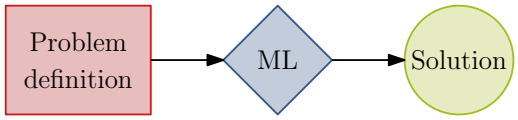



Figure 5.7 Machine learning acts alone to provide a solution to the problem.

This approach has been explored recently, especially on Euclidean TSPs. To tackle the problem with deep learning, Vinyals *et al.* [24] introduce the pointer network wherein an encoder, namely an RNN, is used to parse all the TSP nodes in the input graph and produces an encoding (a vector of activations) for each of them. Afterward, a decoder, also an RNN,

uses an attention mechanism similar to the one of Bahdanau *et al.* [23] (Section 5.2.2) over the previously encoded nodes in the graph to produce a probability distribution over these nodes (through the softmax layer previously illustrated in Figure 5.4). Repeating this decoding step, it is possible for the network to output a permutation over its inputs (the TSP nodes). This method makes it possible to use the network over different input graph sizes. The authors train the model through supervised learning with precomputed TSP solutions as targets. Bello *et al.* [122] use a similar model and train it with reinforcement learning using the tour length as a reward signal. They address some limitations of supervised (imitation) learning, such as the need to compute optimal (or at least high quality) TSP solutions (the targets), that in turn, may be ill-defined when those solutions are not computed exactly, or when multiple solutions exist. Kool and Welling [123] introduce more prior knowledge in the model using a GNN instead of an RNN to process the input. Emami and Ranka [124] and Nowak *et al.* [125] explore a different approach by directly approximating a double stochastic matrix in the output of the neural network to characterize the permutation. The work of Khalil *et al.* [27], introduced in Section 5.3.1.2, can also be understood as an end to end method to tackle the TSP, but we prefer to see it under the eye of Section 5.3.2.3. It is worth noting that tackling the TSP through ML is not new. Earlier work from the nineties focused on Hopfield neural networks and self organizing neural networks, the interested reader is referred to the survey of Smith [22].

In another example, Larsen *et al.* [126] train a neural network to predict the solution of a stochastic load planning problem for which a deterministic MILP formulation exists. Their main motivation is that the application needs to make decisions at a tactical level, *i.e.*, under incomplete information, and machine learning is used to address the stochasticity of the problem arising from missing some of the state variables in the observed input. The authors use operational solutions, *i.e.*, solutions to the deterministic version of the problem, and aggregate them to provide (tactical) solution targets to the ML model. As explained in their paper, the highest level of description of the solution is its cost, whereas the lowest (operational) is the knowledge of values for all its variables. Then, the authors place themselves in the middle and predict an aggregation of variables (tactical) that corresponds to the stochastic version of their specific problem. Furthermore, the nature of the application requires to output solutions in real time, which is not possible either for the stochastic version of the load planning problem or its deterministic variant when using state-of-the-art MILP solvers. Then, ML turns out to be suitable for obtaining accurate solutions with short computing times because some of the complexity is addressed offline, *i.e.*, in the learning phase, and the run-time (inference) phase is extremely quick. Finally, note that in the work of Larsen *et al.* [126] an MLP, *i.e.*, a feedforward neural network, is used to process the input instance as a

vector, hence integrating very little prior knowledge about the problem structure.

#### 5.3.2.2 Learning to configure algorithms

In many cases, using only machine learning to tackle the problem may not be the most suitable approach. Instead, ML can be applied to provide additional pieces of information to a CO algorithm as illustrated in Figure 5.8. For example, ML can provide a parametrization of the algorithm (in a very broad sense).

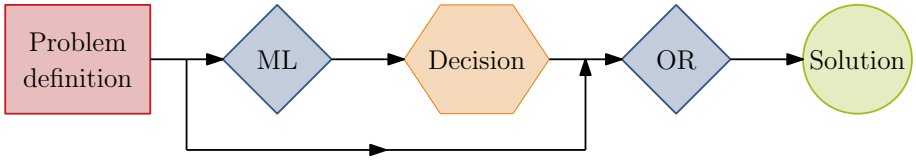



Figure 5.8 The machine learning model is used to augment an operations research algorithm with valuable pieces of information.

Algorithm configuration, detailed by Hoos [28] and Bischl *et al.* [29], is a well studied area that captures the setting presented here. Complex optimization algorithms usually have a set of parameters left constant during optimization (in machine learning they are called hyper-parameters). For instance, this can be the aggressiveness of the pre-solving operations (usually controlled by a single parameter) of an MILP solver, or the learning rate / step size in gradient descent methods. Carefully selecting their value can dramatically change the performance of the optimization algorithm. Hence, the algorithm configuration community started looking for good default parameters. Then good default parameters for different cluster of similar problem instances. From the ML point of view, the former is a constant regression, while the second is a piece-wise constant nearest neighbors regression. The natural continuation was to learn a regression mapping problem instances to algorithm parameters.

In this context, Kruber *et al.* [127] use machine learning on MILP instances to estimate beforehand whether or not applying a Dantzig-Wolfe decomposition will be effective, *i.e.*, will make the solving time faster. Decomposition methods can be powerful but deciding if and how to apply them depends on many ingredients of the instance and of its formulation and there is no clear cut way of optimally making such a decision. In their work, a data point is represented as a fixed length vector with features representing instance and tentative decomposition statistics. In another example, in the context of mixed-integer quadratic programming, Bonami *et al.* [128] use machine learning to decide if linearizing the problem will solve faster. When the quadratic programming (QP) problem given by the relaxation

is convex, *i.e.*, the quadratic objective matrix is semidefinite positive, one could address the problem by a B&B algorithm that solves QP relaxations[7] to provide lower bounds. Even in this convex case, it is not clear if QP B&B would solve faster than linearizing the problem (by using McCormick inequalities [129]) and solving an equivalent MILP. This is why ML is a great candidate here to fill the knowledge gap. In both papers [127, 128], the authors experiment with different ML models, such as support vector machines and random forests, as is good practice when no prior knowledge is embedded in the model.

The heuristic building framework used by Karapetyan *et al.* [118] and Mascia *et al.* [119], already presented in Section 5.3.1.2, can be understood under this eye. Indeed, it can be seen as a large parametric heuristic, configured by the transition probabilities in the former case, and by the parameter representing a sentence in the latter.

As previously stated, the parametrization of the CO algorithm provided by ML is to be understood in a very broad sense. For instance, in the case of radiation therapy for cancer treatment, Mahmood *et al.* [55] use ML to produce candidate therapies that are afterward refined by a CO algorithm into a deliverable plan. Namely, a GAN is used to color CT scan images into a potential radiation plan, then, inverse optimization [130] is applied on the result to make the plan feasible [131]. In general, GANs are made of two distinct networks: one (the generator) generates images, and another one (the discriminator) discriminates between the generated images and a dataset of real images. Both are trained alternatively: the discriminator through a usual supervised objective, while the generator is trained to fool the discriminator. Mahmood *et al.* [55] use a particular type of GAN (conditional GAN) to provide coloring instead of random images. The interested reader is referred to "Generative Adversarial Networks" [132] for an overview on GANs.

We end this section by noting that an ML model used for learning some representation may in turn use as features pieces of information given by another CO algorithm, such as the decomposition statistics used by Kruber *et al.* [127], or the LP information by Bonami *et al.* [128]. Moreover, we remark that, in the satisfiability context, the learning of the type of algorithm to execute on a particular cluster of instances has been paired with the learning of the parameters of the algorithm itself, see, *e.g.*, the work of Ansótegui *et al.* [133, 134].

#### 5.3.2.3 Machine learning alongside optimization algorithms

To generalize the context of the previous section to its full potential, one can build CO algorithms that repeatedly call an ML model throughout their execution, as illustrated in

[7] Note that convex QPs can be solved in polynomial time.

Figure 5.9. A master algorithm controls the high-level structure while frequently calling an ML model to assist in lower level decisions. The key difference between this approach and the examples discussed in the previous section is that the *same ML model* is used by the CO algorithm to make the same type of decisions a number of times in the order of the number of iterations of the algorithm. As in the previous section, nothing prevents one from applying additional steps before or after such an algorithm.

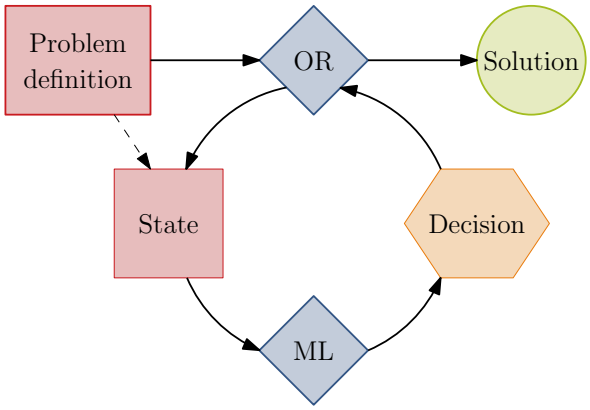



Figure 5.9 The CO algorithm repeatedly queries the same ML model to make decisions. The ML model takes as input the current state of the algorithm, which may include the problem definition.

This is clearly the context of the branch-and-bound tree for MILP, where we already mentioned how the task of selecting the branching variable is either too heuristic or too slow, and is therefore a good candidate for learning [36]. In this case, the general algorithm remains a branch-and-bound framework, with the same software architecture and the same guarantees on lower and upper bounds, but the branching decisions made at every node are left to be learned. Likewise, the work of Hottung *et al.* [116] learning both a branching policy *and* value network for heuristic tree search undeniably fits in this context. Another important aspect in solving MILPs is the use of primal heuristics, *i.e.*, algorithms that are applied in the B&B nodes to find feasible solutions, without guarantee of success. On top of their obvious advantages, good solutions also give tighter upper bounds (for minimization problems) on the solution value and make more pruning of the tree possible. Heuristics depend on the branching node (as branching fix some variables to specific values), so they need to be run frequently. However, running them too often can slow down the exploration of the tree, especially if their outcome is negative, *i.e.*, no better upper bound is detected. Khalil *et al.* [31] build an ML model to predict whether or not running a given heuristic will yield a better solution than the best one found so far and then greedily run that heuristic whenever the outcome of the model is positive.

The approximation used by Baltean-Lugojan *et al.* [109], already discussed in Section 5.3.2.1, is an example of predicting a high-level description of the solution to an optimization problem, namely the objective value. Nonetheless, the goal is to solve the original QP. Thus, the learned model is queried repeatedly to select promising cutting planes. The ML model is used only to select promising cuts, but once selected, cuts are added to the LP relaxation, thus embedding the ML outcome into an exact algorithm. This approach highlights promising directions for this type of algorithm. The decision learned is critical because adding the best cutting planes is necessary for solving the problem fast (or fast enough, because in the presence of NP-hard problems, optimization may time out before any meaningful solving). At the same time, the approximate decision (often in the form of a probability) does not compromise the exactness of the algorithm: any cut added is guaranteed to be valid. This setting leaves room for ML to thrive, while reducing the need for guarantees from the ML algorithms (an active and difficult area of research). In addition, note that, the approach by Larsen *et al.* [126] is part of a master algorithm in which the ML is iteratively invoked to make booking decisions in real time. The work of Khalil *et al.* [27], presented in Section 5.3.1.2, also belongs to this setting, even if the resulting algorithm is heuristic. Indeed, an ML model is asked to select the most relevant node, while a master algorithm maintains the partial tour, computes its length, *etc.* Because the master algorithm is very simple, it is possible to see the contribution as an end-to-end method, as stated in Section 5.3.2.1, but it can also be interpreted more generally as done here.

Presented in Section 5.3.1.2, and mentioned in the previous section, the Markov Chain framework for building heuristics of Karapetyan *et al.* [118] can also be framed as repeated decisions. The transition matrix can be queried and sampled from in order to transition from one state to another, *i.e.*, to make the low level decisions of choosing the next move. The three distinctions made in this Section 5.3.2 are general enough that they can overlap. Here, the fact that the model operates on internal state transitions, yet is learned globally, is what makes it hard to analyze.

Before ending this section, it is worth mentioning that learning recurrent algorithmic decisions is also used in the deep learning community, for instance in the field of meta-learning to decide how to apply gradient updates in stochastic gradient descent [34, 35, 135].

## 5.4 Learning objective

In the previous section, we have surveyed the existing literature by orthogonally grouping the main contributions of ML for CO into families of approaches, sometimes with overlaps. In this section, we formulate and study the objective that drives the learning process.

### 5.4.1 Multi-instance formulation

In the following, we introduce an abstract learning formulation (inspired by Bischl *et al.* [29]). How would an ML practitioner compare optimization algorithms? Let us define $\mathcal{I}$ to be a set of problem instances, and $P$ a probability distribution over $\mathcal{I}$. These are the problems that we care about, weighted by a probability distribution, reflecting the fact that, in a practical application, not all problems are as likely. In practice, $\mathcal{I}$ or $P$ are inaccessible, but we can observe some samples from $P$, as motivated in the introduction with the Montreal delivery company. For a set of algorithms $\mathcal{A}$, let $m : \mathcal{I} \times \mathcal{A} \to \mathbb{R}$ be a measure of the performance of an algorithm on a problem instance (lower is better). This could be the objective value of the best solution found, but could also incorporate elements from optimality bounds, absence of results, running times, and resource usage. To compare $a_1,\ a_2 \in \mathcal{A}$, an ML practitioner would compare $\mathbb{E}_{i\sim P}\ m(i, a_1)$ and $\mathbb{E}_{i\sim P}\ m(i, a_2)$, or equivalently

$$\min_{a\in\{a_1,a_2\}} \mathbb{E}_{i\sim P}\ m(i, a). \tag{5.4}$$

Because measuring these quantities is not tractable, one will typically use empirical estimates instead, by using a finite dataset $D_{train}$ of independent instances sampled from $P$

$$\min_{a\in\{a_1,a_2\}} \sum_{i\in D_{train}} \frac{1}{|D_{train}|} m(i, a). \tag{5.5}$$

This is intuitive and done in practice: collect a dataset of problem instances and compare say, average running times. Of course, such expectation can be computed for different datasets (different $\mathcal{I}$'s and $P$'s), and different measures (different $m$'s).

This is already a learning problem. The more general one that we want to solve through leaning is

$$\min_{a\in\mathcal{A}} \mathbb{E}_{i\sim P}\ m(i, a). \tag{5.6}$$

Instead of comparing between two algorithms, we may compare among an uncountable, maybe non-parametric, space of algorithms. To see how we come up with so many algorithms, we have to look at the algorithms in Section 5.3, and think of the ML model space over which we learn as parametrizing the algorithm space $\mathcal{A}$. For instance, consider the case of learning a branching policy $\pi$ for B&B. If we define the policy to be a neural network with a set of weights $\theta \in \mathbb{R}^p$, then we obtain a parametric B&B algorithm $a(\pi_\theta)$ and (5.6) becomes

$$\min_{\theta\in\mathbb{R}^p} \mathbb{E}_{i\sim P}\ m(i, a(\pi_\theta)). \tag{5.7}$$

Unfortunately, solving this problem is hard. On the one hand, the performance measure $m$ is most often not differentiable and without closed form expression. We discuss this in Section 5.4.2. On the other hand, computing the expectation in (5.6) is intractable. As in (5.5), one can use an empirical distribution using a finite dataset, but that leads to *generalization* considerations, as explained in Section 5.4.3.

Before we move on, let us introduce a new element to make (5.6) more general. That formula suggests that, once given an instance, the outcome of the performance measure is deterministic. That is unrealistic for multiple reasons. The performance measure could itself incorporate some source of randomness due to external factors, for instance with running times which are hardware and system dependent. The algorithm could also incorporate non negligible sources of randomness, if it is designed to be stochastic, or if some operations are non deterministic, or to express the fact that the algorithm should be robust to the choice of some external parameters. Let $\tau$ be that source of randomness, $\pi \in \Pi$ the internal policy being learned, and $a(\pi, \tau)$ the resulting algorithm, then we can reformulate (5.6) as

$$\min_{\pi \in \Pi} \; \mathbb{E}_{i \sim P} \left[ \, \mathbb{E}_{\tau} [ \, m(i, \, a(\pi, \tau)) \mid i \, ] \, \right]. \tag{5.8}$$

In particular, when learning repeated decisions, as in Section 5.3.2.3, this source of randomness can be expressed along the trajectory followed in the MDP, using the dynamics of the environment $p(s', r|a, s)$ (see Figure 5.2). The addition made in (5.8) will be useful for the discussion on generalization in Section 5.4.3.

### 5.4.2 Surrogate objectives

In the previous section, we have formulated a proper learning objective. Here, we try to relate that objective to the learning methods of Section 5.3.1, namely, demonstration and experience. If the usual learning metrics of an ML model, *e.g.*, accuracy for classification in supervised (imitation) learning, are improving, does it mean that the performance metric of (5.6) is also improving?

A straightforward approach for solving (5.8) is that of reinforcement learning (including direct optimization methods), as surveyed in Section 5.3.1.2. The objective from (5.6) can be optimized directly on experience data by matching the total return to the performance measure. Sometimes, a single final reward can naturally be decoupled across the trajectory. For instance, if the performance objective of a B&B variable selection policy is to minimize the number of opened nodes, then the policy can receive a reward discouraging an increase in the number of nodes, hence giving an incentive to select variables that lead to pruning. However,

that may not be always possible, leaving only the option of delaying a single reward to the end of the trajectory. This sparse reward setting is challenging for RL algorithms, and one might want to design a surrogate reward signal to encourage intermediate accomplishments. This introduces some discrepancies, and the policy being optimized may learn a behavior not intended by the algorithm designer. There is *a priori* no relationship between two reward signals. One needs to make use of their intuition to design surrogate signals, *e.g.*, minimizing the number of B&B nodes *should* lead to smaller running times. Reward shaping is an active area of research in RL, yet it is often performed by a number of engineering tricks.

In the case of learning a policy from a supervised signal from expert demonstration, the performance measure $m$ does not even appear in the learning problem that is solved. In this context, the goal is to optimize a policy $\pi \in \Pi$ in the action space to mimic an expert policy $\pi_e$ (as first introduced with Figure 5.5)

$$\min_{\pi \in \Pi} \mathbb{E}_{i \sim P} \left[ \mathbb{E}_s [ \ell(\pi(s), \pi_e(s)) \mid i, \pi_e ] \right], \tag{5.9}$$

where $\ell$ is a task dependent loss (classification, regression, *etc.*). We have emphasized that the state $S$ is conditional, not only on the instance, but also on the expert policy $\pi_e$ used to collect the data. Intuitively, the better the ML model learns, *i.e.*, the better the policy imitates the expert, the closer the final performance of the learned policy should be to the performance of the expert. Under some conditions, it is possible to relate the performance of the learned policy to the performance of the expert policy, but covering this aspect is out of the scope of this paper. The opposite is not true, if learning fails, the policy may still turn out to perform well (by encountering an alternative good decision). Indeed, when making a decision with high surrogate objective error, the learning will be fully penalized when, in fact, the decision could have good performances by the original metric. For that reason, it is capital to report the performance metrics. For example, we surveyed in Section 5.3.2.2 the work of Bonami *et al.* [128] where the authors train a classifier to predict if a mixed integer quadratic problem instance should be linearized or not. The targets used for the learner are computed optimally by solving the problem instance in both configurations. Simply reporting the classification accuracy is not enough. Indeed, this metric gives no information on the impact a misclassification has on running times, the metric used to compute the targets. In the binary classification, a properly classified example could also happen to have unsignificant difference between the running times of the two configurations. To alleviate this issue, the authors also introduce a category where running times are not significatively different (and report the real running times). A continuous extension would be to learn a regression of the solving time. However, learning this regression now means that the final algorithm needs to

optimize over the set of decisions to find the best one. In RL, this is analoguous to learning a value function (see Section 5.2.2). Applying the same reasoning to repeated decisions is better understood with the complete RL theory.

### 5.4.3 On generalization

In Section 5.4.1, we have claimed that the probability distribution in (5.6) is inaccessible and needs to be replaced by the empirical probability distribution over a finite dataset $D_{train}$. The optimization problem solved is

$$\min_{a \in \mathcal{A}} \sum_{i \in D_{train}} \frac{1}{|D_{train}|} m(i, a). \tag{5.10}$$

As pointed out in Section 5.2.2, when optimizing over the empirical probability distribution, we risk having a low performance measure on the finite number of problem instances, *regardless of the true probability distribution.* In this case, the *generalization* error is high because of the discrepancy between the training performances and the true expected performances (overfitting). To control this aspect, a validation set $D_{valid}$ is introduced to compare a finite number of candidate algorithms based on estimates of generalization performances, and a test set $D_{test}$ is used for estimating the generalization performances of the selected algorithm.

In the following, we look more intuitively at generalization in ML for CO, and its consequences. To make it easier, let us recall different learning scenarios. In the introduction, we have motivated the Montreal delivery company example, where the problems of interest are from an unknown probability distribution of Montreal TSPs. This is a very restricted set of problems, but enough to deliver value for this business. Much more ambitious, we may want our policy learned on a finite set of instances to perform well (generalize) to any "*real-world*" MILP instance. This is of interest if you are in the business of selling MILP solvers and want the branching policy to perform well for as many of your clients as possible. In both cases, generalization applies to the instances that are not known to the algorithm implementer. These are the only instances that we care about; the ones used for training are already solved. The topic of probability distribution of instances also appears naturally in stochastic programming/optimization, where uncertainty about the problem is modeled through probability distributions. Scenario generation, an essential way to solve this type of optimization programs, require sampling from this distribution and solving the associated problem multiple times. Nair *et al.* [136] take advantage of this repetitive process to learn an end-to-end model to solve the problem. Their model is composed of a local search and a local improvement policy and is trained through RL. Here, generalization means that, during sce-

nario generation, the learned search beats other approaches, hence delivering an overall faster stochastic programming algorithm. In short, *learning without generalization is pointless!*

When the policy generalizes to other problem instances, it is no longer a problem if training requires additional computation for solving problem instances because, learning can be decoupled from solving as it can be done offline. This setting is promising as it could give a policy to use out of the box for similar instances, while keeping the learning problem to be handled beforehand and remaining hopefully reasonable. When the model learned is a simple mapping, as is the case in Section 5.3.2.2, generalization to new instances, as previously explained, can be easily understood. However, when learning sequential decisions, as in Section 5.3.2.3, there are intricate levels of generalization. We said that we want the policy to generalize to new instances, but the policy also needs to generalize to internal states of the algorithm for a single instance, even if the model can be learned from complete optimization trajectories, as formulated by (5.8). Indeed, complex algorithms can have unexpected sources of randomness, even if they are designed to be deterministic. For instance, a numerical approximation may perform differently if the version of some underlying numerical library is changed or because of asynchronous computing, such as when using Graphical Processing Units [137]. Furthermore, even if we can achieve perfect replicability, we do not want the branching policy to break if some other parameters of the solver are set (slightly) differently. At the very least, we want the policy to be robust to the choice of the random seed present in many algorithms, including MILP solvers. These parameters can therefore be modeled as random variables. Because of these nested levels of generalization, one appealing way to think about the training data from multiple instances is like separate tasks of a multi-task learning setting. The different tasks have underlying aspects in common, and they may also have their own peculiar quirks. One way to learn a single policy that generalizes within a distribution of instances is to take advantage of these commonalities. Generalization in RL remains a challenging topic, probably because of the fuzzy distinction between a multi-task setting, and a large environment encompassing all of the tasks.

Choosing how ambitious one should be in defining the characteristics of the distribution is a hard question. For instance, if the Montreal company expands its business to other cities, should they be considered as separate distributions, and learn one branching policy per city, or only a single one? Maybe one per continent? Generalization to a larger variety of instances is challenging and requires more advanced and expensive learning algorithms. Learning an array of ML models for different distributions asociated with a same task means of course more models to train, maintain, and deploy. The same goes with traditional CO algorithms, an MILP solver on its own is not the best performing algorithm to solve TSPs, but it works across all MILP problems. It is too early to provide insights about how broad the considered

distributions should be, given the limited literature in the field. For scholars generating synthetic distributions, two intuitive axes of investigation are "*structure*" and "*size*". A TSP and a scheduling problem seem to have fairly different structure, and one can think of two planar euclidean TSPs to be way more similar. Still, two of these TSPs can have dramatically different sizes (number of nodes). For instance, Gasse *et al.* [32] assess their methodology independently on three distributions. Each training dataset has a specific problem structure (set covering, combinatorial auction, and capacitated facility location), and a fixed problem size. The problem instance generators used are state-of-art and representative of real-world instances. Nonetheless, when they evaluate their learned algorithm, the authors push the test distributions to larger sizes. The idea behind this is to gauge if the model learned is able to generalize to a larger, more practical, distribution, or only perform well on the restricted distribution of problems of the same size.

### 5.4.4 Single instance learning

An edge case that we have not much discussed yet is the single instance learning framework. This might be the case for instance for planning the design of a single factory. The factory would only be built once, with very peculiar requirements, and the planners are not interested to relate this to other problems. In this case, one can make as many runs (episodes) and as many calls to a potential expert or simulator as one wants, but ultimately one only cares about solving this one instance. Learning a policy for a single instance should require a simpler ML model, which could thus require less training examples. Nonetheless, in the single instance case, one learns the policy from scratch at every new instance, actually incorporating learning (not learned models but really the learning process itself) into the end algorithm. This means starting the timer at the beginning of learning and competing with other solvers to get the solution the fastest (or get the best results within a time limit). This is an edge scenario that can only be employed in the setting of the Section 5.3.2.3, where ML is embedded *inside* a CO algorithm; otherwise there would be only one training example! There is therefore no notion of generalization to other problem instances, so (5.6) is not the learning problem being solved. Nonetheless, the model still needs to generalize to *unseen states* of the algorithm. Indeed, if the model was learned from all states of the algorithm that are needed to solve the problem, then the problem is already solved at training time and learning is therefore fruitless. This is the methodology followed by Khalil *et al.* [59], introduced in Section 5.3.1.1, to learn an instance-specific branching policy. The policy is learned from strong-branching at the top of the B&B tree, but needs to generalize to the state of the algorithm at the bottom of the tree, where it is used. However, as for all CO algorithms, a fair comparison to another algorithm can only be done on an independent dataset of instances, as in (5.4).

This is because through human trials and errors, the data used when building the algorithm leaks into the design of the algorithm, even without explicit learning components.

### 5.4.5 Fine tuning and meta-learning

A compromise between instance-specific learning and learning a generic policy is what we typically have in multi-task learning: some parameters are shared across tasks and some are specific to each task. A common way to do that (in the transfer learning scenario) is to start from a generic policy and then adapt it to the particular instance by a form of *fine-tuning* procedure: training proceeds in two stages, first training the generic policy across many instances from the same distribution, and then continuing training on the examples associated with a given instance on which we are hoping to get more specialized and accurate predictions.

Machine learning advances in the areas of meta-learning and transfer learning are particularly interesting to consider here. Meta-learning considers two levels of optimization: the inner loop trains the parameters of a model on the training set in a way that depends on meta-parameters, which are themselves optimized in an outer loop (*i.e.*, obtaining a gradient for each completed inner-loop training or update). When the outer loop's objective function is performance on a validation set, we end up training a system so that it will generalize well. This can be a successful strategy for generalizing from very few examples if we have access to many such training tasks. It is related to transfer learning, where we want that what has been learned in one or many tasks helps improve generalization on another. These approaches can help rapidly adapt to a new problem, which would be useful in the context of solving many MILP instances, seen as many related tasks.

To stay with the branching example on MILPs, one may not want the policy to perform well out of the box on new instances (from the given distribution). Instead, one may want to learn a policy offline that can be adapted to a new instance in a few training steps, every time it is given one. Similar topics have been explored in the context of automatic configuration tools. Fitzgerald *et al.* [138] study the automatic configuration in the lifelong learning context (a form of sequential transfer learning). The automatic configuration algorithm is augmented with a set of previous configurations that are prioritized on any new problem instance. A score reflecting past performances is kept along every configuration. It is designed to retain configurations that performed well in the past, while letting new ones a chance to be properly evaluated. The automatic configuration optimization algorithm used by Lindauer and Hutter [139] requires training an empirical cost model mapping the Cartesian product of parameter configurations and problem instances to expected algorithmic performance. Such a model

is usually learned for every cluster of problem instances that requires configuring. Instead, when presented with a new cluster, the authors combine the previously learned cost models and the new one to build an ensemble model. As done by Fitzgerald *et al.* [138], the authors also build a set of previous configurations to prioritize, using an empirical cost model to fill the missing data. This setting, which is more general than not performing any adaptation of the policy, has potential for better generalization. Once again, the scale on which this is applied can vary depending on ambition. One can transfer on very similar instances, or learn a policy that transfers to a vast range of instances.

Meta-learning algorithms were first introduced in the 1990s [140–142] and have since then become particularly popular in ML, including, but not limited to, learning a gradient update rule [33, 34], few shot learning [143], and multi-task RL [144].

### 5.4.6 Other metrics

Other metrics from the process of learning itself are also relevant, such as how fast the learning process is, the sample complexity (number of examples required to properly fit the model), *etc.* As opposed to the metrics suggested earlier in this section, these metrics provide us with information not about final performance, but about offline computation or the number of training examples required to obtain the desired policy. This information is, of course, useful to calibrate the effort in integrating ML into CO algorithms.

## 5.5 Methodology

In the previous section, we have detailed the theoretical learning framework of using ML in CO algorithms. Here, we provide some additional discussion broadening some previously made claims.

### 5.5.1 Demonstration and experience

In order to learn a policy, we have highlighted two methodologies: demonstration, where the expected behavior is shown by an expert or oracle (sometimes at a considerable computational cost), and experience, where the policy is learned through trial and error with a reward signal.

In the demonstration setting, the performance of the learned policy is bounded by the expert, which is a limitation when the expert is not optimal. More precisely, without a reward signal, the imitation policy can only hope to marginally outperform the expert (for example because the learner can reduce the variance of the answers across similarly-performing experts). The

better the learning, the closer the performance of the learner to the expert's. This means that imitation alone should be used only if it is significantly faster than the expert to compute the policy. Furthermore, the performance of the learned policy may not generalize well to unseen examples and small variations of the task and may be unstable due to accumulation of errors. This is because in (5.9), the data was collected according to the expert policy $\pi_e$, but when run over multiple repeated decisions, the distribution of states becomes that of the learned policy. Some downsides of supervised (imitation) learning can be overcome with more advanced algorithms, including active methods to query the expert as an oracle to improve behavior in uncertain states. The part of imitation learning presented here is limited compared to the current literature in ML.

On the contrary, with a reward, the algorithm learns to optimize for that signal and can potentially outperform any expert, at the cost of a much longer training time. Learning from a reward signal (experience) is also more flexible when multiple decisions are (almost) equally good in comparison with an expert that would favor one (arbitrary) decision. Experience is not without flaws. In the case where policies are approximated (*e.g.*, with a neural network), the learning process may get stuck around poor solutions if exploration is not sufficient or solutions which do not generalize well are found. Furthermore, it may not always be straightforward to define a reward signal. For instance, sparse rewards may be augmented using reward shaping or a curriculum in order to value intermediate accomplishments (see Section 5.2.2).

Often, it is a good idea to start learning from demonstrations by an expert, then refine the policy using experience and a reward signal. This is what was done in the original AlphaGo paper [145], where human knowledge is combined with reinforcement learning. The reader is referred to "Imitation Learning" [146] for a survey on imitation learning covering most of the discussion in this section.

### 5.5.2 Partial observability

We mentioned in Section 5.2.2 that sometimes the states of an MDP are not fully observed and the Markov property does not hold, *i.e.*, the probability of the next observation, conditioned on the current observation and action, is not equal to the probability of the next observation, conditioned on all past observations and actions. An immediate example of this can be found in any environment simulating physics: a single frame/image of such an environment is not sufficient to grasp notions such as velocity and is therefore not sufficient to properly estimate the future trajectory of objects. It turns out that, on real applications, partial observability is closer to the norm than to the exception, either because one does not have access to a

true state of the environment, or because it is not computationally tractable to represent and needs to be approximated. A straightforward way to tackle the problem is to compress all previous observations using an RNN. This can be applied in the imitation learning setting, as well as in RL, for instance by learning a recurrent policy [147].

How does this apply in the case where we want to learn a policy function making decisions for a CO algorithm? On the one hand, one has full access to the state of the algorithm because it is represented in exact mathematical concepts, such as constraints, cuts, solutions, B&B tree, *etc.* On the other hand, these states can be exponentially large. This is an issue in terms of computations and generalization. Indeed, if one does want to solve problems quickly, one needs to have a policy that is also fast to compute, especially if it is called frequently as is the case for, say, branching decisions. Furthermore, considering too high-dimensional states is also a statistical problem for learning, as it may dramatically increase the required number of samples, decrease the learning speed, or fail altogether. Hence, it is necessary to keep these aspects in mind while experimenting with different representations of the data.

### 5.5.3 Exactness and approximation

In the different examples we have surveyed, ML is used in both exact and heuristic frameworks, for example by Baltean-Lugojan *et al.* [109] and Larsen *et al.* [126], respectively. Getting the output of an ML model to respect advanced types of constraints is a hard task. In order to build exact algorithms with ML components, it is necessary to apply the ML where all possible decisions are valid. Using only ML as surveyed in Section 5.3.2.1 cannot give any optimality guarantee, and only weak feasibility guarantees (see Section 5.6.1). However, applying ML to select or parametrize a CO algorithm as in Section 5.3.2.2 will keep exactness if all possible choices that ML discriminate lead to complete algorithms. Finally, in the case of repeated interactions between ML and CO surveyed in Section 5.3.2.3, all possible decisions must be valid. For instance, in the case of MILPs, this includes branching *among fractional variables* of the LP relaxation, selecting the node to explore *among open branching nodes* [115], deciding on the frequency to run heuristics on the B&B nodes [31], selecting cutting planes *among valid inequalities* [109], removing previous cutting planes *if they are not original constraints or branching decision*, *etc.* A counter-example can be found in the work of Hottung *et al.* [116], presented in Section 5.3.1.1. In their branch-and-bound framework, bounding is performed by an approximate ML model that can overestimate lower bounds, resulting in invalid pruning. The resulting algorithm is therefore not an exact one.

## 5.6 Challenges

In this section, we are reviewing some of the algorithmic concepts previously introduced by taking the viewpoint of their associated challenges.

### 5.6.1 Feasibility

In Section 5.3.2.1, we pointed out how ML can be used to directly output solutions to optimization problems. Rather than learning the solution, it would be more precise to say that the algorithm is learning a *heuristic*. As already repeatedly noted, the learned algorithm does not give any guarantee in terms of optimality, but it is even more critical that feasibility is not guaranteed either. Indeed, we do not know how far the output of the heuristic is from the optimal solution, or if it even respects the constraints of the problem. This can be the case for every heuristic and the issue can be mitigated by using the heuristic within an exact optimization algorithm (such as branch and bound).

Finding feasible solutions is not an easy problem (theoretically NP-hard for MILPs), but it is even more challenging in ML, especially by using neural networks. Indeed, trained with gradient descent, neural architectures must be designed carefully in order not to break differentiability. For instance, both pointer networks [24] and the Sinkhorn layer [124] are complex architectures used to make a network output a permutation, a constraint easy to satisfy when writing a classical CO heuristic.

### 5.6.2 Modelling

In ML, in general, and in deep learning, in particular, we know some good prior for some given problems. For instance, we know that a CNN is an architecture that will learn and generalize more easily than others on image data. The problems studied in CO are different from the ones currently being addressed in ML, where most successful applications target natural signals. The architectures used to learn good policies in combinatorial optimization might be very different from what is currently used with deep learning. This might also be true in more subtle or unexpected ways: it is conceivable that, in turn, the optimization components of deep learning algorithms (say, modifications to SGD) could be different when deep learning is applied to the CO context.

Current deep learning already provides many techniques and architectures for tackling problems of interest in CO. As pointed out in Section 5.2.2, techniques such as parameter sharing made it possible for neural networks to process sequences of variable length with RNNs or, more recently, to process graph structured data through GNNs. Processing graph data is of

uttermost importance in CO because many problems are formulated (represented) on graphs. For a very general example, Selsam *et al.* [148] represent a satisfiability problem using a bipartite graph on variables and clauses. This can generalize to MILPs, where the constraint matrix can be represented as the adjacency matrix of a bipartite graph on variables and constraints, as done by Gasse *et al.* [32].

### 5.6.3 Scaling

Scaling to larger problems can be a challenge. If a model trained on instances up to some size, say TSPs up to size fifty nodes, is evaluated on larger instances, say TSPs of size a hundred, five hundred nodes, *etc*, the challenge exists in terms of generalization, as mentioned in Section 5.4.3. Indeed, all of the papers tackling TSP through ML and attempting to solve larger instances see degrading performance as size increases much beyond the sizes seen during training [24, 27, 122, 123]. To tackle this issue, one may try to learn on larger instances, but this may turn out to be a computational and generalization issue. Except for very simple ML models and strong assumptions about the data distribution, it is impossible to know the computational complexity and the sample complexity, i.e. the number of observations that learning requires, because one is unaware of the exact problem one is trying to solve, *i.e.*, the true data generating distribution.

### 5.6.4 Data generation

Collecting data (for example instances of optimization problems) is a subtle task. Larsen *et al.* [126] claim that "*sampling from historical data is appropriate when attempting to mimic a behavior reflected in such data*". In other words, given an external process on which we observe instances of an optimization problem, we can collect data to train some policy needed for optimization, and expect the policy to generalize on future instances of this application. A practical example would be a business that frequently encounters optimization problems related to their activities, such as the Montreal delivery company example used in the introduction.

In other cases, *i.e.*, when we are not targeting a specific application for which we would have historical data, how can we proactively train a policy for problems that we do not yet know of? As partially discussed in Section 5.4.3, we first need to define to which family of instances we want to generalize over. For instance, we might decide to learn a cutting plane selection policy for Euclidian TSP problems. Even so, it remains a complex effort to generate problems that capture the essence of real applications. Moreover, CO problems are high dimensional, highly structured, and troublesome to visualize. The sole exercise of

generating graphs is already a complicated one! The topic has nonetheless received some interest. Smith-Miles and Bowly [149] claim that the confidence we can put in an algorithm "*depends on how carefully we select test instances*", but note however that too often, a new algorithm is claimed "*to be superior by showing that it outperforms previous approaches on a set of well-studied instances*". The authors propose a problem instance generating method that consists of: defining an instance feature space, visualizing it in two dimensions (using dimensionality reduction techniques such as principal component analysis), and using an evolutionary algorithm to drive the instance generation toward a pre-defined sub-space. The authors argue that the method is successful if the easy and hard instances can be easily separated in the reduced instance space. The methodology is then fruitfully applied to graph-based problems, but would require redefining evolution primitives in order to be applied to other type of problems. On the contrary, Malitsky *et al.* [150] propose a method to generate problem instances from the same probability distribution, in that case, the one of "*industrial*" boolean satisfiability problem instances. The authors use a large neighborhood search, using destruction and reparation primitives, to search for new instances. Some instance features are computed to classify whether the new instances fall under the same cluster as the target one.

Deciding how to represent the data is also not an easy task, but can have a dramatic impact on learning. For instance, how does one properly represent a B&B node, or even the whole B&B tree? These representations need to be expressive enough for learning, but at the same time, concise enough to be used frequently without excessive computations.

## 5.7 Conclusions

We have surveyed and highlighted how machine learning can be used to build combinatorial optimization algorithms that are partially learned. We have suggested that imitation learning alone can be valuable if the policy learned is significantly faster to compute than the original one provided by an expert, in this case a combinatorial optimization algorithm. On the contrary, models trained with a reward signal have the potential to outperform current policies, given enough training and a supervised initialization. Training a policy that generalizes to unseen problems is a challenge, this is why we believe learning should occur on a distribution small enough that the policy could fully exploit the structure of the problem and give better results. We believe end-to-end machine learning approaches to combinatorial optimization can be improved by using machine learning in combination with current combinatorial optimization algorithms to benefit from the theoretical guarantees and state-of-the-art algorithms already available.

Other than performance incentives, there is also interest in using machine learning as a modelling tool for discrete optimization, as done by Lombardi and Milano [54], or to extract intuition and knowledge about algorithms as mentioned by Khalil *et al.* [27] and Bonami *et al.* [128].

Although most of the approaches we discussed in this paper are still at an exploratory level of deployment, at least in terms of their use in general-purpose (commercial) solvers, we strongly believe that this is just the beginning of a new era for combinatorial optimization algorithms.

**Acknowledgments**

The authors are grateful to Emma Frejinger, Simon Lacoste-Julien, Jason Jo, Laurent Charlin, Matteo Fischetti, Rémi Leblond, Michela Milano, Sébastien Lachapelle, Eric Larsen, Pierre Bonami, Martina Fischetti, Elias Khalil, Bistra Dilkina, Sebastian Pokutta, Marco Lübbecke, Andrea Tramontani, Dimitris Bertsimas and the entire CERC team for endless discussions on the subject and for reading and commenting a preliminary version of the paper.

# CHAPTER 6 ARTICLE 3 - ECOLE: A LIBRARY FOR LEARNING INSIDE MILP SOLVERS

Authors: Antoine Prouvost, Justin Dumouchelle, Maxime Gasse, Didier Chételat, Andrea Lodi
Under review in *INFORMS Journal of Computing* [51].

**Abstract** In this paper we describe Ecole (Extensible Combinatorial Optimization Learning Environments), a library to facilitate integration of machine learning in combinatorial optimization solvers. It exposes sequential decision making that must be performed in the process of solving as Markov decision process. This means that, rather than trying to predict solutions to combinatorial optimization problems directly, Ecole allows machine learning to work in cooperation with a state-of-the-art mixed-integer linear programming solver that acts as a controllable algorithm. Ecole provides a collection of computationally efficient, ready to use learning environments, which are also easy to extend to define novel training tasks. Documentation and code can be found at `https://www.ecole.ai`.

## 6.1 Introduction

Combinatorial optimization algorithms play a crucial role in our societies, for tackling a wide range of decision problems arising in, but not limited to, transportation, supply chain, energy, finance and scheduling [151]. These optimization problems, framed as mathematical programs, are inherently hard to solve, forcing practitioners to constantly develop and improve existing algorithms. As a result, general-purpose mathematical solvers typically rely on a large number of handcrafted heuristics that are critical to efficient problem solving, but whose interplay is usually not well understood and is exponentially hard to analyse. These heuristics can be regarded as having been *learned* by human experts through trial and error, on public (or private) data-sets of problems such as MIPLIB [8].

At the same time, traditional solvers typically disregard the fact that some applications require to solve similar problems repeatedly, and tackle each new problem independently, without leveraging any knowledge from the past. In this context, applying ML to CO appears as a natural idea, and has actually been a topic of interest for quite some time [22]. With the recent success of ML, especially the DL sub-field, there is renewed appeal to replace some of the heuristic rules inside traditional solvers by statistical models learned from data. The result would be a solver whose performance could be automatically tailored to a

given distribution of mathematical optimization problems, which could be either application-specific or general-purpose ones. The reader is referred to "Machine learning for combinatorial optimization: a methodological tour d'horizon" [37] for a detailed survey on the topic.

In this article, we present Ecole, an open-source library aimed at facilitating the development of ML approaches within general-purpose MILP solvers based on the B&B algorithm. The remainder of this article is organized as follows. In Section 6.2, we detail the challenges faced by practitioners for applying ML inside combinatorial optimization (CO) solvers. In Section 6.3, we present existing software that also aim at facilitating the development of ML solutions for CO. In Section 6.4, we provide background on mixed-integer linear programming and the branch-and-bound algorithm, as well as the concepts of Markov decision process and reinforcement learning. In Section 6.5, we present our formulation of control problems arising in mathematical solvers as Markov decision processes, and in Section 6.6, we showcase the Ecole interface and how it relates to this formulation. In Section 6.7, we compare the computing performance of Ecole for extracting solver features, compared to existing implementations from the literature. Finally, we conclude with a discussion on future plans for Ecole in Section 6.8.

## 6.2 Motivation

Building the appropriate software to apply ML inside of a B&B solver is not an easy task, and requires a deep knowledge of the solver. It may take months of software engineering before researchers can focus on the actual ML algorithm, and the engineering endeavors can be dissuasive. For example, it suffices to look at research articles with public software implementation [32, 38, 40, 115] to get an idea of the complexity of the required code base. Not to mention the fact that such implementations can themselves contain bugs, and will quickly become outdated.

Solvers such as SCIP [1], CPLEX [2], and Gurobi [3], expose their application programming interface (API) in the `c` programming language, while the state-of-the-art tools for ML such as Scikit-Learn [96], PyTorch [91], and TensorFlow [152] exist primarily in Python. Advanced software engineering skills are necessary to interface both ecosystems, and the room for errors is large, especially if additional time is not invested to write tests for the code. Once these hardships are overcome, the resulting implementation may still be slow and lack advanced features such as parallelization (in particular due to the Python global interpreter lock (GIL) that prevents multi-threaded code executions). Furthermore, research software written for particular projects is often difficult to reuse without copy-editing code, as they lack extensible concept abstractions, proper software packaging, and code maintenance.

Ecole is a free and open-source library built around the SCIP solver to address the aforementioned issues. Several decision problems of interest that arise inside the solver are exposed through an extensible interface akin to OpenAI Gym library [49], a library familiar to ML practitionners. Going further, Ecole aims at improving the reproducibility of scientific research in the area with unified problem benchmarks and metrics, and provides strong default options for new researchers to start with, without the need for an expert knowledge of the inner workings of a mathematical solvers.

## 6.3 Related Work

Other libraries have been introduced recently to facilitate the application of ML to operations research. MipLearn [50] is basically aimed at the same goals as Ecole, with a strong focus on customization and extensibility. It supports two competitive commercial solvers, namely CPLEX and Gurobi, but as a result is limited in the type of interactions it offers, and only allows for using ML for solver configuration. In contrast, Ecole only supports the open-source solver SCIP, but allows for repeated decision making, such the selection of branching variables during B&B, which is a cornerstone of the algorithm. ORGym [47] and OpenGraphGym [48] also offer Gym-like learning environments, for general OR problems and for graph-based problems, respectively. Both are aimed at using ML to produce feasible solutions directly, without the need for an MILP solver. As such they do not allow for the exact solving of CO problems. Ecole, on the other hand, benefits from the inherent mathematical guarantees of a mathematical solver, which include the possibility of exact solving. As such Ecole does not necessarily offer a replacement to the existing software in the ML for OR ecosystem, but rather a (nice) complement that fills some existing gaps. For instance, practitioners can use one of the problem benchmarks from MipLearn, ORGym or OpenGraphGym, to generate a collection of instances in a standard format, and then use Ecole for learning to branch via ML.

## 6.4 Background

We now introduce formally some key concepts that are relevant for describing Ecole, related to both combinatorial optimization and reinforcement learning.

### 6.4.1 Combinatorial Optimization

Mathematical optimization can be used to model a variety of problems. Variables model the decisions to be made, constraints represent physical or structural restrictions, while the objec-

tive function defines a measure of cost to be minimized. When the objective and constraints are linear, and some variables are restricted to be integer, the problem is a mixed-integer linear programming (MILP) problem. MILP is an important class of decision problems but MILP problems are, in general, $\mathcal{NP}$-hard to solve. The B&B algorithm [4] is an implicit enumeration scheme that is generally at the core of the mathematical solvers designed to tackle these problems. The algorithm starts by computing the (continuous) LP relaxation, typically using the Simplex algorithm [153]. If the solution respects the integrality constraints, then the solution is optimal. Otherwise, the feasible space is split by branching on (*i.e.*, partitioning the domain of) an integer variable in a way that excludes the solution to the current LP relaxation. The algorithm is then recursively applied to the sub-domain. If a sub-domain LP relaxation is infeasible, or if its objective value does not improve on the best feasible solution, then the algorithm can stop exploring it.

General-purpose solvers, such as SCIP [1], CPLEX [2], and Gurobi [3], have emerged to provide an enhanced version of the B&B algorithm. They include additional techniques such as presolving [10], cutting planes [11, 12], and primal heuristics [13], which together with B&B have contributed to drastically reduce the solving time of MILP in the last decades [7].

### 6.4.2 Reinforcement Learning

In RL, an agent interacts with an environment and receive rewards as a (indirect) consequence of its actions. The frameworks is defined by MDP problems as follows. At every time step $t$, the agent is in a given state $S_t$ and decides on an action $A_t$. The agent decisions are modeled by a probabilistic policy function: for a given state $s$ that the agent is in, the agent takes an action $a$ with probability $\pi(a|s)$. As a result, and depending on the unknown dynamics of the environment, the agent transitions into a new state $S_{t+1}$ and receives a reward $R_{t+1}$. If the agent is in a state $s$, and takes an action $a$, then the probability to transition into a new state $s'$ and receiving a reward $r$ is denoted by $\mathbb{P}(s', r|a, s)$. This is illustrated in Figure 6.1. The process ends when reaching a state defined to be terminal, where the set of possible actions is empty. The objective for the agent is to maximize the expected sum of future rewards.

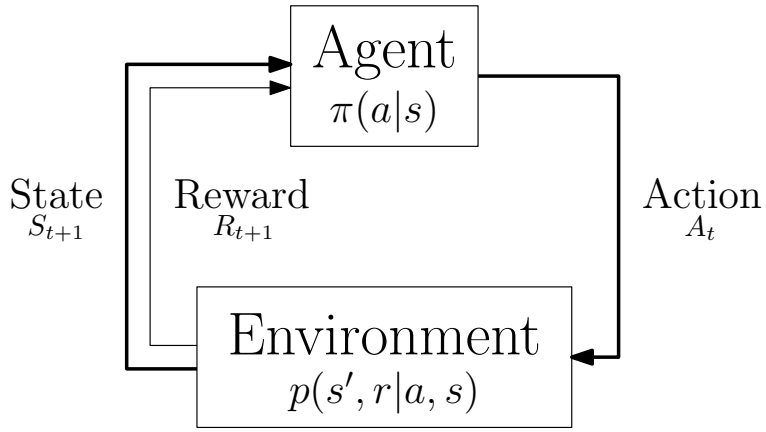



Figure 6.1 The MDP associated with reinforcement learning, modified from *Reinforcement Learning: An Introduction* [21].

A sequence of actions and transitions $\tau$ is called an *episode*, or *trajectory*. The probability of a given trajectory is given by

$$\mathbb{P}(\tau) = \mathbb{P}(S_0) \prod_t \pi(A_t|S_t)\mathbb{P}(S_{t+1}, R_{t+1}|A_t, S_t). \tag{6.1}$$

Common RL algorithms fall in two categories. Policy methods try to directly learn an approximation of the agent policy function $\pi$. Value based methods estimate the state-action value function $Q(s, a)$ defined as the expected sum of future rewards given that the agent is in a state $s$ and takes an action $a$. The associated policy is then computed through approximate dynamic programming [154]. Finally, modern approaches such as actor-critic methods both train a policy (the actor) and an estimate of the value function of the policy (the critic) in a single procedure.

### 6.4.3 Imitation Learning

An alternative to find good policies in a MDP is to learn to imitate another expert policy, an approach usually referred to as imitation learning. In this scenario, something usually prevents using the expert directly: for example, the expert might be a human, or might be an expensive algorithm that takes excellent decisions at high computational cost.

The simplest possible approach for imitation learning is behavioral cloning, where state-action pairs are collected by running the expert for a while, and a ML model is trained by imitation learning to predict actions that would be taken by the expert in those states. A downside of this approach is the state distribution mismatch problem: as the student will

usually make mistakes, it will deviate from the kinds of states likely to be encountered by the expert and will soon end up in new states much unlike those seen during training. So called active imitation learning methods, such as DAGGER [155], try to correct this mismatch to improve final policy performance.

## 6.5 Solver Control Problem

Although MILP solvers are usually deterministic processes, in Ecole we adopt a general MDP formulation, introduced in Section 6.4.2, which allows for non-deterministic processes.

To better map the Ecole interface presented in the next section, we further refine probability (6.1). First, we split the initial state distribution $\mathbb{P}(S_0)$ using Bayes' rule to introduce the probability distribution of the problem instance $I$ being solved, which can be formulated as $\mathbb{P}(S_0) = \mathbb{P}(I)\mathbb{P}(S_0|I)$. Second, we borrow from the partially-observable MDP framework [156] and introduce an observation function $\mathcal{O}$ of the state that is used by the agent to make decisions, using all of the past trajectory, that is $\pi(A_t|\mathcal{O}(S_0), ..., \mathcal{O}(S_t))$. Similarly, we reformulate rewards as a function $\mathcal{R}$ of the state and leave it out of the transition probability $\mathbb{P}(S_{t+1}|A_t, S_t)$. This gives us the following final probability distribution of a trajectory:

$$\mathbb{P}(\tau) = \mathbb{P}(I)\mathbb{P}(S_0|I) \prod_t \pi(A_t|H_t)\ \mathbb{P}(S_{t+1}|A_t, S_t), \tag{6.2}$$

where the history $H_t$ is given by

$$H_t = \{\mathcal{O}(S_0), \mathcal{R}(S_0), A_0, ..., \mathcal{O}(S_{t-1}), \mathcal{R}(S_{t-1}), A_{t-1}, \mathcal{O}(S_t)\}. \tag{6.3}$$

Ecole provides an API to define environments inside the solver. It lets users sample from those environments according to (6.2), as well as extract rewards and observations using the functions $\mathcal{R}$ and $\mathcal{O}$. Two environments are currently implemented in Ecole.

**Configuring** The first environment expose the task of algorithm configuration [28]. The goal is to find well performing SCIP parameters, then let the solver run its course without intervention. This task is akin to what is learned by Hutter *et al.* [30]. In this scenario, finding optimal parameters can be framed as a contextual bandit problem[1]. Contextual bandit problems are special cases of RL where the underlying MDP has unit episode length, so it fits naturally in the framework adopted by Ecole as a special case. In this environment, the one and only action to be taken is a mapping of SCIP parameters and associated values.

[1] A good reference on the topic is *Bandit algorithms*, Part V [157].

**Branching** The second environment implemented allows users to select the variable to branch on each B&B node, as used by Gasse *et al.* [32]. The state $S_t$ is defined as the state of the solver on the $t^{\text{th}}$ node, and is equivalent to the branching rule callback available in SCIP. In the work of Gasse *et al.* [32], the observation function $\mathcal{O}$ is extracting a bipartite graph representation of the solver, and $\pi$ is a GNN.

Plans for future environments are suggested in Section 6.8.

## 6.6 Design Decisions

The API of Ecole is designed for ease of use and extensibility. In this section, we detail some of the key features of Ecole.

### 6.6.1 Environment Interface

**Environments** The interface for using an environment is inspired by the OpenAI Gym [49] library. The main abstraction is the `Environment` class, which is used to encapsulate any control problem, as formulated in Section 6.5. The Listing 6.1 provides an example of using Ecole with the `Branching` environment for B&B variable selection. The inner `while` loop spans over a full episode, while the outer `for` loop repeats it multiple times for different problem instances. An episode, starts with a call to `reset`. The method takes as parameter an MILP problem instance from which an initial state will be sampled, and returns an observation of that state, and a Boolean flag indicating whether that state is terminal. Transitions are performed by calling the `step` method, with the action provided by the user (the `policy` function). It returns the observation of the new state, the reward, the flag indicating whether the new state is terminal, and a dictionary of additional non-essential information about the transition.

```
import ecole

env = ecole.environment.Branching(
    reward_function=ecole.reward.LpIterations() ** 2,
    observation_function=ecole.observation.NodeBipartite(),
)
instances = ecole.instance.IndependentSetGenerator(n_nodes=100)

for _ in range(10):
    obs, action_set, reward_offset, done, info = env.reset(next(instances))
    while not done:
        action = policy(obs, action_set)
        obs, action_set, reward, done, info = env.step(action)
```

Listing 6.1 Default usage of environments in Python.

**Reward and Observation Functions** Furthermore, Listing 6.1 shows how the constructor of `Environment` can be used to specify the rewards and observations to compute. Some observation functions from the literature are provided in Ecole, such as the ones used by Gasse *et al.* [32] and Khalil *et al.* [59]. The listing also demonstrates how new reward functions can be dynamically created by applying mathematical operations (`+`, `-`, `*`, `/`, `**`, `.exp()`...) on them.

**Instance Generators** Learning inside a solver may require large amount of training instances. Although industry applications can provide extensive datasets of instances of interest, it is also valuable to have on hand generators with good defaults to quickly experiment ideas. Out of convenience, we provide four families of generators for users to learn from. These are the same families of instances that were used to benchmark the imitation learning method of Gasse *et al.* [32].

- Set covering MILP problems generated following the procedure of Balas and Ho [158];
- Combinatorial auction MILP problems generated following the procedure of Leyton-Brown *et al.* [159, Section 4.3];
- Capacitated facility location MILP problems generated following the procedure of Cornuéjols *et al.* [160];
- Independent set MILP problems generated following the procedure of Bergman *et al.* [161, Section 4.6.4] with both Erdos-Renyi [162] and Barabasi-Albert [163] graphs.

Table 6.1 summarizes the key abstractions in Ecole and how they map to the mathematical formulation presented in Section 6.5. Some elements are further explained in the next section.

Table 6.1 Comparing Ecole API to its mathematical formulation.

| | | |
|---|---|---|
| Observation function | $\mathcal{O}$ | `NodeBipartite()` |
| Reward function | $\mathcal{R}$ | `LpIterations() ** 2` |
| Instance dist. | $\mathbb{P}(I)$ | `IndependentSetGenerator(n_nodes=100)` |
| State | $S_t$ | `Model` |
| Cond. initial state dist. | $\mathbb{P}(S_0\|I)$ | `BranchingDynamics.reset_dynamics` |
| Policy | $\pi(A_t\|H_t)$ | `policy` |
| Transition dist. | $\mathbb{P}(S_{t+1}\|A_t, S_t)$ | `BranchingDynamics.step_dynamics` |

### 6.6.2 Extensibility

OpenAI Gym is designed as a set of benchmarks for ML practitioners. However, in Ecole the tasks also have industry applications, and therefore it was an essential design principle to allow users flexibility in designing the environments. For example, although good defaults bring value, if users decide to train an agent with a customized observations or reward function and this leads in the end to better solving times, this is a net gain and Ecole should allow for it. Thus, environments were made to be customizable, unlike in OpenAI Gym.

In this section, we explain how users can customize the reward function $\mathcal{R}$, the observation function $\mathcal{O}$, the instance distribution $\mathbb{P}(I)$, and the transition dynamics $\mathbb{P}(S_0|I)$ and $\mathbb{P}(S_{t+1}|A_t, S_t)$.

**Reward and Observation Functions** Users can create observation or reward functions by creating a class with two methods, as shown in Listing 6.2. They have access to the state of the MDP, *i.e.*, the underlying SCIP solver, through the `ecole.scip.Model`, or equivalently a `pyscipopt.Model` object (both being wrappers of a `SCIP*` pointer in `C`). There is no limitation to what an observation can be because they are an abstraction used exclusively by the user.

```
class Observation:
    ...

class ObservationFunction:
    def before_reset(model: ecole.scip.Model) -> Observation:
        ...

    def extract(model: ecole.scip.Model) -> Observation:
        ...
```

Listing 6.2 Python interface of an observation function.

**Transitions and Initial State** The initial state and transition probability distribution can be customized by creating a `EnvironmentDynamics` object. Its API is similar to that of the `Environment`, with the exception that the two methods `reset_dynamics` and `step_dynamics` solely manipulate the solver, without computing either observations or rewards. Environments are actually wrappers around dynamics that also call the reward and observation functions.

**Instance Generators** Instance generators are regular Python generators that output an `ecole.scip.Model`. The users are free to define any new generators that can create problem instances, or read them from file.

For all the abstractions above, existing components of Ecole can easily be reused through composition or inheritance to speed up development.

### 6.6.3 Comparison with OpenAI Gym

One objective of developing Ecole is to provide an interface close to the popular OpenAI Gym library. Nonetheless, some differences exist.

**Condition Initial State distribution** The `reset` method in Gym does not accept any parameters. In Ecole, it makes little sense to solve the same MILP instance over and over, hence conditioning the initial state probability distribution on the instance is mandatory.

**Initial Terminal States** In the Gym interface, initial states cannot be terminal. As a result the `reset` method does not return the boolean termination flag. However, terminal initial states do arise in the environments defined in Ecole. For example, in B&B variable selection, the instance could be solved through preprocessing and never require any branching.

**Reward Offsets** In Ecole, rewards are also returned on `reset`. This is because rewards usually come from differences of metrics, but users cannot compute the total metric without knowing how the metric evolved until the first decision point. The reset reward is this missing information. For example, the solving time reward reports how much time is spent between each decision and the next state. Summing these rewards give the total time spent solving since the agent was first asked for a decision, but it does not take into account time spent before that point. In B&B variable selection, this time would include preprocessing, and root node cutting plane calculation and LP solving. This pre-decision time is what is returned by reset, allowing it to be summed to the rest of the rewards to find the total solving time.

**Action Spaces** In OpenAI Gym, `action_space` is a property of the environment class, that is, it does not change between episodes. In Ecole, not only the set of actions can change from one instance to the next, it can also change between transitions. For instance, in B&B variable selection, the set of valid branching candidates changes on every node.

### 6.6.4 Ecosystem

The largest part of Ecole is written in `C++`, and exposes bindings to Python through PyBind11 [164]. Xtensor [165] is used for high level multi-dimensional arrays, and NumPy [93] is used for binding them to Python.

Writing Ecole in `C++` provides solid ground for a computationally efficient library that can be made available in multiple programming languages [166].

## 6.7 Performance Experiments

Speed is a core principle of Ecole. Besides having most of its codebase in `C++`, selecting carefully the third-party libraries on which it relies (see Section 6.6.4) and tuning compilation options, other optimizations were leveraged to further speedup Ecole. When using Ecole from Python, calls into the `C++` code are made without copying parameters and return values unless necessary. Furthermore, any function doing significant work, such as data extraction or calls to the SCIP solver, are performed without holding the GIL, allowing Python users to use Python threading capabilities and avoid the overhead that comes with multiprocessing parallelization.

We propose two experiments to illustrate the gains provided by these optimizations and benchmark Ecole performance.[2] The benchmarks were run on a server with Linux OpenSUSE

[2]The code of the experiments is available at `https://github.com/ds4dm/ecole-paper`.

Tumbleweed 20210114, with 32GB of RAM, and and Intel i7-6700K (8 cores, 4.0 GHz) CPU. To analyze the results, we used SciPy [94], Jupyter [98], Pandas [95], Matplotlib [99], and Seaborn [100].

### 6.7.1 Ecole Overhead Experiment

The first concern was to understand whether using Ecole without extracting any data creates any overhead compared to using SCIP directly. In particular, SCIP offers the possibility to select a branching variable through the use of a callback function, while the Ecole interface presented in Section 6.6.1 wraps the callback function to select branching variables iteratively (effectively transforming the callback in a stackful coroutine, *i.e.*, a function that can be suspended and resumed).

Using the four instance generators from Gasse *et al.* [32] (implemented in Ecole and presented in Section 6.6.2), we compare branching on the first available branching candidate using Ecole and vanilla SCIP. To speedup the benchmark, we disable presolving, cutting planes, and limit the number of nodes to 100. Over a total of 4500 instances, a one sample t-test shows that the ratio of the wall times is not significantly different from 1.0 (with p-value $< 10^{-50}$). Thus, we can conclude that for a typical usage, Ecole produces no overhead. This is explained by the fact that any possible overhead time is dwarfed in comparison to the time spent solving the problem.

### 6.7.2 Observation Functions Experiment

Our second experiment aims at measuring the sole execution time of two observation functions implemented in Ecole: `NodeBipartite` from Gasse *et al.* [32] and `Khalil2016` from Khalil *et al.* [59] and implemented by Gasse *et al.* [32]. Using the four instance generators from Gasse *et al.* [32], with no presolving, no cutting planes, and limiting to 100 nodes, we measure their sole execution time. A total of 439 instances were generated.

The results are given in Table 6.2 for `Khalil2016`. For `NodeBipartite`, it is possible to cache some information computed at the root node for future use but this is incompatible with cutting planes. The results without and with a cache are given in Tables 6.3 and 6.4, respectively. In all tables, the average time is given along the standard deviations. The ratio of the two first column is given in the third one.

Table 6.2 Comparison of execution times for `Khalil2016`.

| Generator | Wall time Gasse (s) | Wall time Ecole (s) | Time Ratio |
|---|---|---|---|
| CFLP 100-100 | $(1.067 \pm 0.072) * 10^2$ | $1.050 \pm 0.117$ | $102.2 \pm 5.9$ |
| CFLP 200-100 | $(4.381 \pm 0.641) * 10^2$ | $2.580 \pm 0.420$ | $170.2 \pm 8.8$ |
| CFLP 400-100 | $(2.468 \pm 0.354) * 10^3$ | $6.151 \pm 0.768$ | $400.3 \pm 13.7$ |
| CAuction 100-500 | $(5.682 \pm 0.856) * 10^{-1}$ | $(3.752 \pm 0.605) * 10^{-2}$ | $15.19 \pm 0.77$ |
| CAuction 200-1000 | $2.654 \pm 0.437$ | $(1.067 \pm 0.190) * 10^{-1}$ | $24.93 \pm 0.74$ |
| CAuction 300-1500 | $7.207 \pm 0.718$ | $(1.967 \pm 0.209) * 10^{-1}$ | $36.68 \pm 0.98$ |
| IndependentSet 500 | $1.531 \pm 0.192$ | $(3.210 \pm 0.285) * 10^{-1}$ | $4.755 \pm 0.212$ |
| IndependentSet 1000 | $8.209 \pm 1.463$ | $1.424 \pm 0.146$ | $5.726 \pm 0.512$ |
| IndependentSet 1500 | $(2.226 \pm 0.407) * 10$ | $3.060 \pm 0.284$ | $7.228 \pm 0.724$ |
| SetCover 500-1000 | $(9.962 \pm 6.410) * 10^{-2}$ | $(9.261 \pm 6.197) * 10^{-3}$ | $10.26 \pm 2.99$ |
| SetCover 1000-1000 | $(4.351 \pm 2.812) * 10^{-1}$ | $(5.136 \pm 3.403) * 10^{-2}$ | $8.613 \pm 0.540$ |
| SetCover 2000-1000 | $1.210 \pm 0.627$ | $(1.980 \pm 1.135) * 10^{-1}$ | $6.274 \pm 0.555$ |

Table 6.3 Comparison of execution times for `NodeBipartite` without cache.

| Generator | Wall time Gasse (s) | Wall time Ecole (s) | Time Ratio |
|---|---|---|---|
| CFLP 100-100 | $(4.232 \pm 0.070) * 10^{-1}$ | $(1.209 \pm 0.038) * 10^{-1}$ | $3.505 \pm 0.113$ |
| CFLP 200-100 | $(8.017 \pm 0.384) * 10^{-1}$ | $(2.815 \pm 0.125) * 10^{-1}$ | $2.848 \pm 0.058$ |
| CFLP 400-100 | $1.557 \pm 0.016$ | $(6.179 \pm 0.101) * 10^{-1}$ | $2.520 \pm 0.037$ |
| CAuction 100-500 | $(7.376 \pm 0.549) * 10^{-2}$ | $(4.074 \pm 0.348) * 10^{-3}$ | $18.14 \pm 0.89$ |
| CAuction 200-1000 | $(9.797 \pm 0.473) * 10^{-2}$ | $(8.753 \pm 0.566) * 10^{-3}$ | $11.21 \pm 0.36$ |
| CAuction 300-1500 | $(1.160 \pm 0.016) * 10^{-1}$ | $(1.356 \pm 0.047) * 10^{-2}$ | $8.564 \pm 0.226$ |
| IndependentSet 500 | $(6.806 \pm 0.043) * 10^{-1}$ | $(1.679 \pm 0.017) * 10^{-1}$ | $4.053 \pm 0.028$ |
| IndependentSet 1000 | $2.564 \pm 0.012$ | $(8.849 \pm 0.090) * 10^{-1}$ | $2.898 \pm 0.026$ |
| IndependentSet 1500 | $6.697 \pm 0.421$ | $2.046 \pm 0.015$ | $3.273 \pm 0.201$ |
| SetCover 500-1000 | $(4.347 \pm 2.556) * 10^{-2}$ | $(3.986 \pm 2.413) * 10^{-3}$ | $10.25 \pm 2.94$ |
| SetCover 1000-1000 | $(1.015 \pm 0.283) * 10^{-1}$ | $(1.238 \pm 0.412) * 10^{-2}$ | $8.368 \pm 0.749$ |
| SetCover 2000-1000 | $(1.768 \pm 0.304) * 10^{-1}$ | $(2.896 \pm 0.628) * 10^{-2}$ | $6.179 \pm 0.469$ |

Table 6.4 Comparison of execution time for `NodeBipartite` with cache.

| Generator | Wall time Gasse (s) | Wall time Ecole (s) | Time Ratio |
|---|---|---|---|
| CFLP 100-100 | $(1.472 \pm 0.019) * 10^{-1}$ | $(9.381 \pm 0.287) * 10^{-2}$ | $1.570 \pm 0.047$ |
| CFLP 200-100 | $(2.891 \pm 0.123) * 10^{-1}$ | $(1.958 \pm 0.083) * 10^{-1}$ | $1.477 \pm 0.031$ |
| CFLP 400-100 | $(6.093 \pm 0.058) * 10^{-1}$ | $(3.818 \pm 0.065) * 10^{-1}$ | $1.596 \pm 0.024$ |
| CAuction 100-500 | $(1.407 \pm 0.095) * 10^{-2}$ | $(2.983 \pm 0.224) * 10^{-3}$ | $4.720 \pm 0.098$ |
| CAuction 200-1000 | $(1.988 \pm 0.081) * 10^{-2}$ | $(6.327 \pm 0.308) * 10^{-3}$ | $3.144 \pm 0.050$ |
| CAuction 300-1500 | $(2.484 \pm 0.021) * 10^{-2}$ | $(9.486 \pm 0.152) * 10^{-3}$ | $2.619 \pm 0.030$ |
| IndependentSet 500 | $(1.371 \pm 0.019) * 10^{-1}$ | $(7.254 \pm 0.090) * 10^{-2}$ | $1.890 \pm 0.018$ |
| IndependentSet 1000 | $(5.268 \pm 0.047) * 10^{-1}$ | $(3.117 \pm 0.032) * 10^{-1}$ | $1.690 \pm 0.016$ |
| IndependentSet 1500 | $1.305 \pm 0.008$ | $(7.088 \pm 0.046) * 10^{-1}$ | $1.842 \pm 0.012$ |
| SetCover 500-1000 | $(9.406 \pm 4.883) * 10^{-3}$ | $(2.836 \pm 1.678) * 10^{-3}$ | $3.435 \pm 1.160$ |
| SetCover 1000-1000 | $(1.851 \pm 0.432) * 10^{-2}$ | $(7.153 \pm 2.052) * 10^{-3}$ | $2.673 \pm 0.374$ |
| SetCover 2000-1000 | $(2.791 \pm 0.399) * 10^{-2}$ | $(1.422 \pm 0.267) * 10^{-2}$ | $1.985 \pm 0.145$ |

As can be seen, Ecole is systematically faster than the literature. The shorter execution times in Ecole can be explained by the fact that the code from Gasse *et al.* [32] was a proof of concept and was not extensively optimized.

## 6.8 Conclusions and Future Work

Ecole offers researchers an efficient and well designed interface to the SCIP solver without compromising on customizability.

A current limitation of the library is that only SCIP is supported as a back-end solver. Since commercial, closed-source solvers such as CPLEX [2] and Gurobi [3] are very popular in industry, it would be natural to extend the library to support them as potential back-ends. For now, their closed-source nature limits this possibility, but we hope that interest in the current version of Ecole will lead solver developers to facilitate interfacing with ML libraries in the future.

Future work on Ecole will involve developing new environments, such as node selection and cutting planes selection, as well as new observation and reward functions, such as primal, dual and primal-dual integral metrics. In addition, support for out-of-the-box parallelism would be useful to cope with computationally expensive environments.

**Acknowledgments**

This work was supported by the Canada Excellence Research Chair (CERC) in "Data Science for Real-Time Decision Making", Mila - Quebec Artificial Intelligence Institute, and IVADO - Institut de valorisation des données. We are grateful to the SCIP team at the Zuse Institute Berlin for their support in developing Ecole.

# CHAPTER 7 GENERAL DISCUSSION

The three articles presented in Chapters 4, 5, and 6 demonstrate the potential of applying ML to solve practical CO applications. Throughout the thesis, the methodology we present grows more and more extensive. It starts from the sequential application of ML and CO, to a complete integration of learning inside optimization algorithms. The present chapter highlights such connections among the three contributions.

The most straightforward way to think of the combination of ML and CO to solve operational problems is sequential. Real-world applications are not as well defined as academic optimization problems. They stem from data that is noisy but with identifiable statistical properties. Hence, the fist step before solving an optimization problem is to apply statistical methods, such as ML, to retrieve the underlying properties of the data. In Chapter 4, we presented a DL model to estimate health care patients' risk of adverse event. This work is actually set in a more general optimization context. The partner company uses similar prediction to plan the visits done by care workers. Hence, the ML model used for risk estimation can be seen as a way to find the parameters of the routing problem. The latter is subsequently passed to an optimization pipeline.

Optimization problems arising from the world differ not only from their problem type, but also from the problem distribution from which they arise. The latter is not something that can be explicitly formulated, and thus applying ML to the optimization algorithm itself, as we survey in Chapter 5, is a promising way to tune the algorithm to this implicit distribution. The article surveys complex approaches, not only sequential, that feature the use of ML models for repeated decision making inside optimization algorithms. Taking a step back, we were able to explain the development of the field under the framework of MDPs. While the use of RL for CO is still in its infancy, the framework is also adapted to imitation learning methods. We expect that the use of RL will keep growing as researchers grow confident that improvements can be made, and as knowledge about good ML priors for CO is built. We noticed a similar pattern with the game of Go. At first, AlphaGo [145] relied on expert play and imitation learning, but was later outperformed by RL alone in Alphazero [167]. Similarly, the work on B&B variable selection from Gasse *et al.* [32] was quickly followed by a RL methodology from Sun *et al.* [41]. We also expect that GNNs will play a significant role in scaling up the field to better DL models, as argued by Cappart *et al.* [45].

Yet, one question remained. With or without ML, how can we solve (faster and faster) CO problems despite them being $\mathcal{NP}$-hard? The answer came from ML, and the no free lunch

theorem. Predictive models, when averaged over all possible problems, perform all equally well. But all possible problems do not matter. When we focus on a restricted distribution, such as the set of natural images in computer vision, some models can significantly outperform others. In this view, all CO problems are also not relevant. For practical CO algorithms, performing well on a distribution of "*natural*" problems (or even application-specific problems) is what matters. With this knowledge, we can view the CO algorithms evolution as a weak form of rule-based learning, trained through human trials and errors on specific problem datasets. With these strong foundations, ML then appears as the natural evolution for solving CO problems.

Armed with a complete methodology, we began applying it to the topic of cutting plane selection in MILP. Our approach featured a GNN policy, trained through RL policy gradients and self-supervised learning. On top of combining three challenging research fields, we realized that large engineering requirements inhibited our ability to quickly iterate over different ideas. We noticed that we were not alone in this situation [32, 40] and that the most impactful way to move the field forward was to give researchers (including ourselves) proper software tools to focus on experimentation, in the same way that Theano [168] helped the DL community focus on research rather than engineering. With democratization of software programming to wider audiences, and the fast pace of doing research, many researchers either lack knowledge or time to work with C codebases. Ecole, presented in Chapter 6, was built out of this common need. Its interface is the practical implementation of the MDP framework presented in Chapter 5. They are the two sides of a same coin. More than providing reusable code components, the interface of Ecole was carefully designed to map to relevant concepts of ML and CO. We believe that the interface defined in Ecole could easily be adapted to apply our methodology to other solvers or algorithms.

# CHAPTER 8 CONCLUSION AND RECOMMENDATIONS

With the present thesis, we have demonstrated the value of using ML for CO, and methodologically built a framework to understand the growing diversity of research emerging in the field. In this chapter, we compile a summary of our three contributions, highlight limitations of our framework, and suggest future areas of research.

## 8.1 Summary of Works

Our first work, presented in Chapter 4, was motivated by a practical problem in healthcare: planning care-workers routes for home-recovery patients. We proposed a DL model to estimate the risk of patients from their daily medical records. Our approach parses historical patient data through a RNN and derives a loss from survival analysis to rank patients without having to estimate an absolute risk, as well as accounts for censor data. The DL model we proposed was able to outperform threshold alerts set by care workers. Although not detailed in the publication, a risk model is needed by the partner company to prioritize the planning of the care worker routes. Therefore, the predictive model can be framed as a processing step of the optimization pipeline to estimate the problem parameters.

This initial work has led us to consider the use of ML, not only for processing stochastic data in operational problems, but also as a way to solve CO problems themselves. Our second contribution (Chapter 5) surveys the recent literature that aim to tackle CO problems with statistical learning methods. We introduce a methodological scheme to frame the rich bibliography of ML for CO as control problems through a MDP formulation. Moreover, we transcribe the statistical learning theory to optimization. We consider the development in CO as a weak form of rule-based learning, found through research trial and error, and study algorithms performances under the lens of statistical generalization.

Finally, we presented in Chapter 6 a software framework to apply statistical learning methods inside a CO solver. Noticing the engineering hardships of current solutions, we built a library that offers the familiar abstractions of RL and MDPs for CO. Inspired by OpenAI Gym, the software is well tested, easy to install, and fast. It is able to reproduce existing articles with a significant speedup factor, and a dramatic reduction in code complexity. Our software aims to be a cornerstone of future research in the area by providing a reusable and highly customizable interface.

## 8.2 Limitations and future research

### 8.2.1 Multi-policy

The ML for CO framework introduced in Chapter 5 and implemented in Ecole suggests many heuristics that could be learn to improve MILP solvers, such as B&B variable selection, node selection, cutting planes selection (and generation), preprocessing routines, and primal heuristics to apply. However, as research in each of these topics improves, the problem of combining multiple policies will emerge. Will separately trained policies keep their performances when combined? Or will they impinge on one another? Better yet, can we achieve superior improvements if we learn the policies together? This can be framed as a single enveloping policy with heterogeneous action space, or as distinct cooperating agents, as described by the multi-agent RL framework.

Compiling learned policies for different solver heuristics will be an unavoidable milestone in the development of *fully learnable solvers*. Such solvers would not only deliver extreme adaptability to problem distributions, but also remove the burden on developers to keep track of how heuristics (learned or hand-coded) interact together.

### 8.2.2 Meta and Transfer Learning

Learning one (or more) CO heuristic is a computationally and data expensive task. Computational resources, or large sets of instances to train a policy on, might not be available, ruling out DL approaches. Once strong ML model priors, such as GNNs, are properly established, a first possibility to fast train heuristics could be achieved by fine tuning a pre-trained policy for the new task at hand.

Learning to transfer a policy can be more profoundly addressed through meta-learning, a blooming DL topic that has been mostly left out of our framework (if not for some high-level ideas in Chapter 5). In this setting, optimization of a policy depends on meta-parameters, themselves optimized to maximize fast generalization (transfer) to new problem distributions.

Reducing the computation and data costs is not only a matter of savings. It also opens new possibilities, such as solving single instances, or real-time optimization.

### 8.2.3 Implicit Modeling

The topic of modeling has been mostly unaddressed in the present thesis. Yet, in practice the problems solved in everyday scenarios are not mathematically well defined. They are modeled by optimization and domain experts in a time-consuming back and forth. ML can

be used to assist in transforming domain data into optimization problems.

Just like expert modeling and optimization are often considered together (the problem formulation impacts its resolution), so should learned models and their optimization. Otherwise some data might be lost in the process, that could otherwise be leveraged by learned optimization algorithms. Furthermore, it is possible that producing human-intelligible models as an intermediary output might even, in some cases, be a bottleneck for an (extraordinary) ML model. This is the approach taken by Ferber *et al.* [169], which envision MILP as a NN layer. Such advances could also benefit to predictive problems that have an unsuspected underlying optimization structure.

# REFERENCES


[1] G. Gamrath, D. Anderson, K. Bestuzheva, W.-K. Chen, L. Eifler, M. Gasse, P. Gemander, A. Gleixner, L. Gottwald, K. Halbig, G. Hendel, C. Hojny, T. Koch, P. Le Bodic, S. J. Maher, F. Matter, M. Miltenberger, E. Mühmer, B. Müller, M. E. Pfetsch, F. Schlösser, F. Serrano, Y. Shinano, C. Tawfik, S. Vigerske, F. Wegscheider, D. Weninger, and J. Witzig, "The SCIP Optimization Suite 7.0", Zuse Institute Berlin, ZIB-Report 20-10, Mar. 2020.

[2] IBM, *CPLEX Optimizer User Manual*, 2020. [Online]. Available: `https://www.ibm.com/analytics/cplex-optimizer`.

[3] Gurobi Optimization LLC, *Gurobi Optimizer Reference Manual*, 2020. [Online]. Available: `http://www.gurobi.com`.

[4] A. H. Land and A. G. Doig, "An Automatic Method of Solving Discrete Programming Problems", *Econometrica*, vol. 28, no. 3, pp. 497–520, 1960, ISSN: 0012-9682. DOI: `10.2307/1910129`.

[5] A. Lodi, "Mixed Integer Programming Computation", en, in *50 Years of Integer Programming 1958-2008*, M. Jünger, T. M. Liebling, D. Naddef, G. L. Nemhauser, W. R. Pulleyblank, G. Reinelt, G. Rinaldi, and L. A. Wolsey, Eds., Berlin, Heidelberg: Springer, 2010, pp. 619–645, ISBN: 9783540682790. DOI: `10.1007/978-3-540-68279-0_16`.

[6] A. Lodi, "The Heuristic (Dark) Side of MIP Solvers", en, in *Hybrid Metaheuristics*, ser. Studies in Computational Intelligence, E.-G. Talbi, Ed., Berlin, Heidelberg: Springer, 2013, pp. 273–284, ISBN: 9783642306716. DOI: `10.1007/978-3-642-30671-6_10`. [Online]. Available: `https://doi.org/10.1007/978-3-642-30671-6_10` (visited on 08/20/2020).

[7] R. Bixby and E. Rothberg, "Progress in computational mixed integer programming—a look back from the other side of the tipping point", *Annals of Operations Research*, vol. 149, no. 1, pp. 37–41, Feb. 2007, ISSN: 1572-9338. DOI: `10.1007/s10479-006-0091-y`. [Online]. Available: `https://doi.org/10.1007/s10479-006-0091-y`.

[8] A. Gleixner, G. Hendel, G. Gamrath, T. Achterberg, M. Bastubbe, T. Berthold, P. M. Christophel, K. Jarck, T. Koch, J. Linderoth, M. Lübbecke, H. D. Mittelmann, D. Ozyurt, T. K. Ralphs, D. Salvagnin, and Y. Shinano, "MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library", *Mathematical Program-*

*ming Computation*, 2021. DOI: 10.1007/s12532-020-00194-3. [Online]. Available: https://doi.org/10.1007/s12532-020-00194-3.

[9] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[10] T. Achterberg, R. E. Bixby, Z. Gu, E. Rothberg, and D. Weninger, "Presolve reductions in mixed integer programming", *INFORMS Journal on Computing*, vol. 32, no. 2, pp. 473–506, 2020. DOI: 10.1287/ijoc.2018.0857.

[11] R. E. Gomory, "Outline of an algorithm for integer solutions to linear programs and an algorithm for the mixed integer problem", in *50 Years of Integer Programming 1958-2008*, Springer, 2010, pp. 77–103.

[12] E. Balas, S. Ceria, and G. Cornuéjols, "A lift-and-project cutting plane algorithm for mixed 0–1 programs", *Mathematical programming*, vol. 58, no. 1, pp. 295–324, 1993.

[13] M. Fischetti and A. Lodi, "Heuristics in mixed integer programming", in *Wiley Encyclopedia of Operations Research and Management Science*, J. J. Cochran, L. A. C. Jr., P. Keskinocak, J. P. Kharoufeh, and J. C. Smith, Eds. Wiley Online Library, 2011, vol. 3, pp. 2199–2204, ISBN: 9780470400531. DOI: 10.1002/9780470400531.eorms0376. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470400531.eorms0376.

[14] L. A. Wolsey, *Integer Programming*. Wiley, 1998.

[15] A. Lodi and A. Tramontani, "Performance variability in mixed-integer programming", in *Theory driven by influential applications*, INFORMS, 2013, pp. 1–12.

[16] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets", in *Advances in Neural Information Processing Systems*, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Q. Weinberger, Eds., vol. 27, Curran Associates, Inc., 2014. [Online]. Available: https://proceedings.neurips.cc/paper/2014/file/5ca3e9b122f61f8f06494c97b1afccf3-Paper.pdf.

[17] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space", *arXiv preprint arXiv:1301.3781*, 2013.

[18] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding", in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–

4186. DOI: 10.18653/v1/N19-1423. [Online]. Available: https://www.aclweb.org/anthology/N19-1423.

[19] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.

[20] K. P. Murphy, *Machine Learning: A Probabilistic Perspective*. MIT press, 2012.

[21] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, Second. MIT press Cambridge, 2018, ISBN: 9780262039246. [Online]. Available: http://incompleteideas.net/book/the-book-2nd.html.

[22] K. A. Smith, "Neural Networks for Combinatorial Optimization: A Review of More Than a Decade of Research", *INFORMS Journal on Computing*, vol. 11, no. 1, pp. 15–34, Feb. 1999, ISSN: 1091-9856. DOI: 10.1287/ijoc.11.1.15.

[23] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate", in *ICLR'2015, arXiv:1409.0473*, 2015.

[24] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer Networks", in *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds., Curran Associates, Inc., 2015, pp. 2692–2700.

[25] A. Graves, G. Wayne, and I. Danihelka, "Neural turing machines", *arXiv preprint arXiv:1410.5401*, 2014.

[26] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks", *arXiv preprint arXiv:1609.02907*, 2016.

[27] E. Khalil, H. Dai, Y. Zhang, B. Dilkina, and L. Song, "Learning Combinatorial Optimization Algorithms over Graphs", in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., Curran Associates, Inc., 2017, pp. 6348–6358.

[28] H. H. Hoos, "Automated Algorithm Configuration and Parameter Tuning", en, in *Autonomous Search*, Y. Hamadi, E. Monfroy, and F. Saubion, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 37–71, ISBN: 978-3-642-21434-9. DOI: 10.1007/978-3-642-21434-9_3. [Online]. Available: https://doi.org/10.1007/978-3-642-21434-9_3 (visited on 05/23/2019).

[29] B. Bischl, P. Kerschke, L. Kotthoff, M. Lindauer, Y. Malitsky, A. Fréchette, H. Hoos, F. Hutter, K. Leyton-Brown, K. Tierney, and J. Vanschoren, "ASlib: A benchmark library for algorithm selection", *Artificial Intelligence*, vol. 237, pp. 41–58, Aug. 2016, ISSN: 0004-3702. DOI: 10.1016/j.artint.2016.04.003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0004370216300388 (visited on 05/23/2019).

[30] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Automated configuration of mixed integer programming solvers", in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, A. Lodi, M. Milano, and P. Toth, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 186–202.

[31] E. B. Khalil, B. Dilkina, G. L. Nemhauser, S. Ahmed, and Y. Shao, "Learning to Run Heuristics in Tree Search", in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, 2017, pp. 659–666.

[32] 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 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 15 580–15 592.

[33] S. Hochreiter, A. S. Younger, and P. R. Conwell, "Learning to learn using gradient descent", in *Artificial Neural Networks — ICANN 2001*, G. Dorffner, H. Bischof, and K. Hornik, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 87–94, ISBN: 978-3-540-44668-2.

[34] M. Andrychowicz, M. Denil, S. Gómez, M. W. Hoffman, D. Pfau, T. Schaul, B. Shillingford, and N. de Freitas, "Learning to learn by gradient descent by gradient descent", in *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, Eds., Curran Associates, Inc., 2016, pp. 3981–3989.

[35] O. Wichrowska, N. Maheswaranathan, M. W. Hoffman, S. G. Colmenarejo, M. Denil, N. de Freitas, and J. Sohl-Dickstein, "Learned Optimizers that Scale and Generalize", in *Proceedings of the 34th International Conference on Machine Learning*, D. Precup and Y. W. Teh, Eds., ser. Proceedings of Machine Learning Research, vol. 70, International Convention Centre, Sydney, Australia: PMLR, Aug. 2017, pp. 3751–3760.

[36] A. Lodi and G. Zarpellon, "On learning and branching: a survey", en, *TOP*, vol. 25, no. 2, pp. 207–236, Jul. 2017, ISSN: 1134-5764, 1863-8279. DOI: 10.1007/s11750-017-0451-6.

[37] Y. Bengio, A. Lodi, and A. Prouvost, "Machine learning for combinatorial optimization: a methodological tour d'horizon", *European Journal of Operational Research*, 2020.

[38] P. Gupta, M. Gasse, E. Khalil, P. Mudigonda, A. Lodi, and Y. Bengio, "Hybrid models for learning to branch", in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33, Curran

Associates, Inc., 2020, pp. 18 087–18 097. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/d1e946f4e67db4b362ad23818a6fb78a-Paper.pdf.

[39] V. Nair, S. Bartunov, F. Gimeno, I. von Glehn, P. Lichocki, I. Lobov, B. O'Donoghue, N. Sonnerat, C. Tjandraatmadja, P. Wang, *et al.*, "Solving mixed integer programs using neural networks", *arXiv preprint arXiv:2012.13349*, 2020.

[40] G. Zarpellon, J. Jo, A. Lodi, and Y. Bengio, "Parameterizing branch-and-bound search trees to learn branching policies", *arXiv preprint arXiv:2002.05120*, 2020.

[41] H. Sun, W. Chen, H. Li, and L. Song, "Improving learning to branch via reinforcement learning", in *Learning Meets Combinatorial Algorithms at NeurIPS2020*, 2020. [Online]. Available: https://openreview.net/forum?id=z4D7-PTxTb.

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

[43] Y. Tang, S. Agrawal, and Y. Faenza, "Reinforcement learning for integer programming: learning to cut", in *International Conference on Machine Learning*, PMLR, 2020, pp. 9367–9376.

[44] K. Yilmaz and N. Yorke-Smith, "Learning efficient search approximation in mixed integer branch and bound", *arXiv preprint arXiv:2007.03948*, 2020.

[45] Q. Cappart, D. Chételat, E. Khalil, A. Lodi, C. Morris, and P. Veličković, "Combinatorial optimization and reasoning with graph neural networks", *arXiv preprint arXiv:2102.09544*, 2021.

[46] M. Fischetti, A. Lodi, and G. Zarpellon, "Learning milp resolution outcomes before reaching time-limit", in *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, L.-M. Rousseau and K. Stergiou, Eds., Cham: Springer International Publishing, 2019, pp. 275–291, ISBN: 978-3-030-19212-9.

[47] C. D. Hubbs, H. D. Perez, O. Sarwar, N. V. Sahinidis, I. E. Grossmann, and J. M. Wassick, *Or-gym: a reinforcement learning library for operations research problems*, 2020. eprint: arXiv:2008.06319.

[48] W. Zheng, D. Wang, and F. Song, "OpenGraphGym: A parallel reinforcement learning framework for graph optimization problems", in *International Conference on Computational Science*, Springer, 2020, pp. 439–452.

[49] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, *Openai gym*, 2016. eprint: arXiv:1606.01540.

[50] A. S. Xavier and F. Qiu, *MIPLearn*, 2020. [Online]. Available: https://anl-ceeesa.github.io/MIPLearn.

[51] A. Prouvost, J. Dumouchelle, M. Gasse, D. Chételat, and A. Lodi, *Ecole: a library for learning inside milp solvers*, 2021. arXiv: 2104.02828 [cs.LG].

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

[53] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, *et al.*, "Deep graph library: a graph-centric, highly-performant package for graph neural networks", *arXiv preprint arXiv:1909.01315*, 2019.

[54] M. Lombardi and M. Milano, "Boosting Combinatorial Problem Modeling with Machine Learning", in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, International Joint Conferences on Artificial Intelligence Organization, Jul. 2018, pp. 5472–5478. DOI: 10.24963/ijcai.2018/772.

[55] R. Mahmood, A. Babier, A. McNiven, A. Diamant, and T. C. Y. Chan, "Automated Treatment Planning in Radiation Therapy using Generative Adversarial Networks", in *Proceedings of Machine Learning for Health Care*, ser. Proceedings of Machine Learning Research, vol. 85, 2018.

[56] A. Babier, T. C. Chan, A. Diamant, and R. Mahmood, "Learning to optimize with hidden constraints", *arXiv preprint arXiv:1805.09293*, 2018.

[57] A. Prouvost, A. Lodi, L.-M. Rousseau, and J. Vallee, "Adverse event prediction by telemonitoring and deep learning", in *International Conference on Human-Centred Software Engineering*, Springer, 2019, pp. 205–215.

[58] P. Toth and D. Vigo, *The vehicle routing problem*. SIAM, 2002.

[59] E. B. Khalil, P. L. Bodic, L. Song, G. Nemhauser, and B. Dilkina, "Learning to Branch in Mixed Integer Programming", in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, ser. AAAI'16, Phoenix, Arizona: AAAI Press, 2016, pp. 724–731.

[60] B. W. Ward, J. S. Schiller, and R. A. Goodman, "Peer reviewed: multiple chronic conditions among us adults: a 2012 update", *Preventing chronic disease*, vol. 11, 2014.

[61] L. Gavrilov and P. Heuveline, "Aging of population", Longevity Science, Tech. Rep., 2003. [Online]. Available: http://longevity-science.org/Population%5C_Aging.htm.

[62] G. Paré, M. Jaana, and C. Sicotte, "Systematic review of home telemonitoring for chronic diseases: the evidence base.", *J Am Med Inform Assoc*, vol. 14, no. 3, pp. 269–77, 2007, ISSN: 1067-5027. DOI: 10.1197/jamia.M2270.

[63] M. S. Rasmussen, T. Justesen, A. Dohn, and J. Larsen, "The home care crew scheduling problem: preference-based visit clustering and temporal dependencies", *European Journal of Operational Research*, vol. 219, no. 3, pp. 598–610, 2012.

[64] M.-k. .-.-. K. Suh, L. S. Evangelista, C.-A. .-.-. A. Chen, K. Han, J. Kang, M. K. Tu, V. Chen, A. Nahapetian, and M. Sarrafzadeh, "An automated vital sign monitoring system for congestive heart failure patients", in *Proceedings of the 1st ACM International Health Informatics Symposium*, ACM, 2010, pp. 108–117.

[65] D. M. Berwick, T. W. Nolan, and J. Whittington, "The triple aim: care, health, and cost", *Health Affairs*, vol. 27, no. 3, pp. 759–769, 2008.

[66] S. F. Jencks, M. V. Williams, and E. A. Coleman, "Rehospitalizations among patients in the medicare fee-for-service program", *New England Journal of Medicine*, vol. 360, no. 14, pp. 1418–1428, 2009.

[67] W. V. Huntington, L. A. Covington, P. P. Center, and L. Manchikanti, "Patient protection and affordable care act of 2010: reforming the health care reform for the new decade", *Pain Physician*, vol. 14, no. 1, E35–E67, 2011.

[68] J. S. Ross, G. K. Mulvey, B. Stauffer, V. Patlolla, S. M. Bernheim, P. S. Keenan, and H. M. Krumholz, "Statistical models and patient predictors of readmission for heart failure: a systematic review", *Archives of internal medicine*, vol. 168, no. 13, pp. 1371–1386, 2008.

[69] L. O. Hansen, R. S. Young, K. Hinami, A. Leung, and M. V. Williams, "Interventions to reduce 30-day rehospitalization: a systematic review", *Annals of internal medicine*, vol. 155, no. 8, pp. 520–528, 2011.

[70] E. Wallace, E. Stuart, N. Vaughan, K. Bennett, T. Fahey, and S. M. Smith, "Risk prediction models to predict emergency hospital admission in community-dwelling adults: a systematic review", *Medical care*, vol. 52, no. 8, p. 751, 2014.

[71] D. Kansagara, H. Englander, A. Salanitro, D. Kagen, C. Theobald, M. Freeman, and S. Kripalani, "Risk prediction models for hospital readmission: a systematic review", *Jama*, vol. 306, no. 15, pp. 1688–1698, 2011.

[72] M. Zhu, L. Cheng, J. J. Armstrong, J. W. Poss, J. P. Hirdes, and P. Stolee, *Machine Learning in Healthcare Informatics*. Springer, 2014, pp. 181–207.

[73] W. G. Baxt, F. S. Shofer, F. D. Sites, and J. E. Hollander, "A neural network aid for the early diagnosis of cardiac ischemia in patients presenting to the emergency department with chest pain", *Annals of Emergency Medicine*, vol. 40, no. 6, pp. 575–583, Dec. 2002, ISSN: 0196-0644. DOI: 10.1067/mem.2002.129171.

[74] R. F. Harrison and R. L. Kennedy, "Artificial neural network models for prediction of acute coronary syndromes using clinical data from the time of presentation", *Annals of emergency medicine*, vol. 46, no. 5, pp. 431–439, 2005.

[75] M. Luck, T. Sylvain, H. Cardinal, A. Lodi, and Y. Bengio, "Deep learning for patient-specific kidney graft survival analysis", *arXiv*, no. 1705.1024, 2017.

[76] M. Swiercz, Z. Mariak, J. Lewko, K. Chojnacki, A. Kozlowski, and P. Piekarski, "Neural network technique for detecting emergency states in neurosurgical patients", *Medical and Biological Engineering and Computing*, vol. 36, no. 6, pp. 717–722, 1998.

[77] R. K. Price, E. L. Spitznagel, T. J. Downey, D. J. Meyer, N. K. Risk, and O. G. El-Ghazzawy, "Applying artificial neural network models to clinical decision making.", *Psychological assessment*, vol. 12, no. 1, p. 40, 2000.

[78] A. Rajkomar, E. Oren, K. Chen, A. M. Dai, N. Hajaj, M. Hardt, P. J. Liu, X. Liu, J. Marcus, M. Sun, P. Sundberg, H. Yee, K. Zhang, Y. Zhang, G. Flores, G. E. Duggan, J. Irvine, Q. Le, K. Litsch, A. Mossin, J. Tansuwan, D. Wang, J. Wexler, J. Wilson, D. Ludwig, S. L. Volchenboum, K. Chou, M. Pearson, S. Madabushi, N. H. Shah, A. J. Butte, M. D. Howell, C. Cui, G. S. Corrado, and J. Dean, "Scalable and accurate deep learning with electronic health records", en, *npj Digital Medicine*, vol. 1, no. 1, p. 18, May 2018, ISSN: 2398-6352. DOI: 10.1038/s41746-018-0029-1. [Online]. Available: https://www.nature.com/articles/s41746-018-0029-1 (visited on 05/28/2018).

[79] A. Avati, K. Jung, S. Harman, L. Downing, A. Ng, and N. H. Shah, "Improving palliative care with deep learning", *BMC Medical Informatics and Decision Making*, vol. 18, no. 4, p. 122, Dec. 2018, ISSN: 1472-6947. DOI: 10.1186/s12911-018-0677-8. [Online]. Available: https://doi.org/10.1186/s12911-018-0677-8.

[80] C. Esteban, O. Staeck, S. Baier, Y. Yang, and V. Tresp, "Predicting clinical events by combining static and dynamic information using recurrent neural networks", in *2016 IEEE International Conference on Healthcare Informatics (ICHI)*, Oct. 2016, pp. 93–101. DOI: 10.1109/ICHI.2016.16.

[81] S. Hochreiter and J. Schmidhuber, "Long short-term memory", *Neural computation*, vol. 9, pp. 1735–80, Dec. 1997. DOI: 10.1162/neco.1997.9.8.1735.

[82] K. Cho, B. van Merrienboer, C. Gulcehre, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation", English (US), in *Conference on Empirical Methods in Natural Language Processing (EMNLP 2014)*, 2014.

[83] R. J. Williams and J. Peng, "An efficient gradient-based algorithm for on-line training of recurrent network trajectories", *Neural Computation*, vol. 2, no. 4, pp. 490–501, 1990. DOI: 10.1162/neco.1990.2.4.490. eprint: https://doi.org/10.1162/neco.1990.2.4.490. [Online]. Available: https://doi.org/10.1162/neco.1990.2.4.490.

[84] J. Harrell Frank E., R. M. Califf, D. B. Pryor, K. L. Lee, and R. A. Rosati, "Evaluating the Yield of Medical Tests", *JAMA*, vol. 247, no. 18, pp. 2543–2546, May 1982, ISSN: 0098-7484. DOI: 10.1001/jama.1982.03320430047030. eprint: https://jamanetwork.com/journals/jama/articlepdf/372568/jama_247_18_030.pdf. [Online]. Available: https://dx.doi.org/10.1001/jama.1982.03320430047030.

[85] D. R. Cox, "Regression models and life-tables", English, *Journal of the Royal Statistical Society. Series B*, vol. 34, pp. 187–220, 1972, ISSN: 0035-9246. [Online]. Available: https://zbmath.org/?q=an:0243.62041 (visited on 05/28/2018).

[86] J. L. Katzman, U. Shaham, A. Cloninger, J. Bates, T. Jiang, and Y. Kluger, "Deepsurv: personalized treatment recommender system using a cox proportional hazards deep neural network", *BMC Medical Research Methodology*, vol. 18, no. 1, p. 24, Feb. 2018, ISSN: 1471-2288. DOI: 10.1186/s12874-018-0482-1. [Online]. Available: https://doi.org/10.1186/s12874-018-0482-1.

[87] W. H. Organization, *International statistical classification of diseases and related health problems*. World Health Organization, 2004, vol. 1.

[88] Y. Choi, C. Y.-I. Chiu, and D. Sontag, "Learning Low-Dimensional Representations of Medical Concepts", *AMIA Summits on Translational Science Proceedings*, vol. 2016, pp. 41–50, Jul. 2016, ISSN: 2153-4063. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5001761/ (visited on 12/05/2017).

[89] Z. Che, S. Purushotham, K. Cho, D. Sontag, and Y. Liu, "Recurrent Neural Networks for Multivariate Time Series with Missing Values", en, *Scientific Reports*, vol. 8, no. 1, May 2018, ISSN: 2045-2322. DOI: 10.1038/s41598-018-24271-9. [Online]. Available: http://www.nature.com/articles/s41598-018-24271-9 (visited on 04/27/2018).

[90] D. P. Kingma and J. Ba, "Adam: a method for stochastic optimization", *arXiv preprint arXiv:1412.6980*, 2014.

[91] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch", in *NeurIPS 2017 Autodiff Workshop*, 2017. [Online]. Available: https://openreview.net/forum?id=BJJsrmfCZ.

[92] C. Davidson-Pilon, J. Kalderstam, P. Zivich, B. Kuhn, A. Fiore-Gartland, L. Moneda, Gabriel, D. WIlson, A. Parij, K. Stark, S. Anton, L. Besson, Jona, H. Gadgil, D. Golland, S. Hussey, J. Noorbakhsh, A. Klintberg, J. Jordan, J. Rose, I. Slavitt, E. Martin, E. Ochoa, D. Albrecht, dhuynh, D. Zgonjanin, D. Chen, C. Fournier, Arturo, and A. F. Rendeiro, *Camdavidsonpilon/lifelines: v0.19.2*, Feb. 2019. DOI: 10.5281/zenodo.2575709. [Online]. Available: https://doi.org/10.5281/zenodo.2575709.

[93] S. Van Der Walt, S. C. Colbert, and G. Varoquaux, "The numpy array: a structure for efficient numerical computation", *Computing in Science & Engineering*, vol. 13, no. 2, p. 22, 2011.

[94] E. Jones, T. Oliphant, P. Peterson, *et al.*, *SciPy: open source scientific tools for Python*, [Online; accessed <today>], 2001. [Online]. Available: http://www.scipy.org/.

[95] W. McKinney, "Data structures for statistical computing in python", in *Proceedings of the 9th Python in Science Conference*, S. van der Walt and J. Millman, Eds., 2010, pp. 51–56.

[96] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, *et al.*, "Scikit-learn: machine learning in python", *Journal of machine learning research*, vol. 12, no. 10, pp. 2825–2830, 2011.

[97] F. Perez and B. E. Granger, "Ipython: a system for interactive scientific computing", *Computing in Science Engineering*, vol. 9, no. 3, pp. 21–29, May 2007, ISSN: 1521-9615. DOI: 10.1109/MCSE.2007.53.

[98] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, and C. Willing, "Jupyter notebooks – a publishing format for reproducible computational workflows", in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt, Eds., IOS Press, 2016, pp. 87–90.

[99] J. D. Hunter, "Matplotlib: a 2d graphics environment", *Computing in Science Engineering*, vol. 9, no. 3, pp. 90–95, May 2007, ISSN: 1521-9615. DOI: 10.1109/MCSE.2007.55.

[100] M. Waskom, O. Botvinnik, D. O'Kane, P. Hobson, J. Ostblom, S. Lukauskas, D. C. Gemperline, T. Augspurger, Y. Halchenko, J. B. Cole, J. Warmenhoven, J. de Ruiter, C. Pye, S. Hoyer, J. Vanderplas, S. Villalba, G. Kunter, E. Quintero, P. Bachant, M. Martin, K. Meyer, A. Miles, Y. Ram, T. Brunner, T. Yarkoni, M. L. Williams, C. Evans, C. Fitzgerald, Brian, and A. Qalieh, *Mwaskom/seaborn: v0.9.0 (july 2018)*, Jul. 2018. DOI: 10.5281/zenodo.1313201. [Online]. Available: https://doi.org/10.5281/zenodo.1313201.

[101] M. Fortun and S. S. Schweber, "Scientists and the legacy of world war ii: the case of operations research (or)", *Social Studies of Science*, vol. 23, no. 4, pp. 595–642, 1993. DOI: 10.1177/030631293023004001. eprint: https://doi.org/10.1177/030631293023004001. [Online]. Available: https://doi.org/10.1177/030631293023004001.

[102] R. C. Larson and A. R. Odoni, *Urban operations research*, Monograph. Prentice-Hall, 1981, ISBN: 9780139394478.

[103] M. Conforti, G. Conrnuéjols, and G. Zambelli, *Integer Programming*. Springer, 2014.

[104] A. Lodi, "MIP computation", in *50 Years of Integer Programming 1958-2008*, M. Jünger, T. Liebling, D. Naddef, G. Nemhauser, W. Pulleyblank, G. Reinelt, G. Rinaldi, and L. Wolsey, Eds., Springer-Verlag, 2009, pp. 619–645.

[105] M. Gendreau and J.-Y. Potvin, Eds., *Handbook of metaheuristics*. Springer, 2010, vol. 2, ISBN: 978-3-319-91086-4. DOI: 10.1007/978-3-319-91086-4. [Online]. Available: https://www.springer.com/gp/book/9783319910857.

[106] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is All you Need", in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., Curran Associates, Inc., 2017, pp. 5998–6008.

[107] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks", in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=rJXMpikCZ.

[108] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural Message Passing for Quantum Chemistry", in *Proceedings of the 34th International Conference on Machine Learning*, D. Precup and Y. W. Teh, Eds., ser. Proceedings of Machine Learning Research, vol. 70, International Convention Centre, Sydney, Australia: PMLR, Aug. 2017, pp. 1263–1272.

[109] R. Baltean-Lugojan, R. Misener, P. Bonami, and A. Tramontani, "Strong sparse cut selection via trained neural nets for quadratic semidefinite outer-approximations", en, Imperial College, London, Tech. Rep., 2018.

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

[111] D. Applegate, R. Bixby, V. Chvátal, and W. Cook, *The traveling salesman problem. A computational study*. Princeton University Press, 2007.

[112] A. Marcos Alvarez, Q. Louveaux, and L. Wehenkel, "A supervised machine learning approach to variable branching in branch-and-bound", Université de Liège, Tech. Rep., 2014.

[113] A. Marcos Alvarez, Q. Louveaux, and L. Wehenkel, "A Machine Learning-Based Approximation of Strong Branching", *INFORMS Journal on Computing*, vol. 29, no. 1, pp. 185–195, Jan. 2017, ISSN: 1091-9856. DOI: 10.1287/ijoc.2016.0723.

[114] A. Marcos Alvarez, L. Wehenkel, and Q. Louveaux, "Online Learning for Strong Branching Approximation in Branch-and-Bound", en, Université de Liège, Tech. Rep., 2016.

[115] H. He, H. Daume III, and J. M. Eisner, "Learning to Search in Branch and Bound Algorithms", in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds., Curran Associates, Inc., 2014, pp. 3293–3301.

[116] A. Hottung, S. Tanaka, and K. Tierney, "Deep Learning Assisted Heuristic Tree Search for the Container Pre-marshalling Problem", *arXiv:1709.09972 [cs]*, Sep. 2017, arXiv: 1709.09972. [Online]. Available: http://arxiv.org/abs/1709.09972 (visited on 05/23/2019).

[117] H. Dai, B. Dai, and L. Song, "Discriminative Embeddings of Latent Variable Models for Structured Data", in *Proceedings of The 33rd International Conference on Machine Learning*, M. F. Balcan and K. Q. Weinberger, Eds., ser. Proceedings of Machine Learning Research, vol. 48, New York, New York, USA: PMLR, Jun. 2016, pp. 2702–2711.

[118] D. Karapetyan, A. P. Punnen, and A. J. Parkes, "Markov Chain methods for the Bipartite Boolean Quadratic Programming Problem", *European Journal of Operational Research*, vol. 260, no. 2, pp. 494–506, Jul. 2017, ISSN: 0377-2217. DOI: 10.1016/j.ejor.2017.01.001. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0377221717300061 (visited on 05/23/2019).

[119] F. Mascia, M. López-Ibáñez, J. Dubois-Lacoste, and T. Stützle, "Grammar-based generation of stochastic local search heuristics through automatic algorithm configuration tools", *Computers & Operations Research*, vol. 51, pp. 190–199, Nov. 2014, ISSN: 0305-0548. DOI: 10.1016/j.cor.2014.05.020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0305054814001555 (visited on 05/23/2019).

[120] E. Özcan, M. Misir, G. Ochoa, and E. K. Burke, "A Reinforcement Learning: Great-Deluge Hyper-Heuristic for Examination Timetabling", en, *Modeling, Analysis, and Applications in Metaheuristic Computing: Advancements and Trends*, pp. 34–55, 2012. DOI: 10.4018/978-1-4666-0270-0.ch003. [Online]. Available: https://www.igi-global.com/chapter/reinforcement-learning-great-deluge-hyper/63803 (visited on 05/23/2019).

[121] G. D. Liberto, S. Kadioglu, K. Leo, and Y. Malitsky, "DASH: Dynamic Approach for Switching Heuristics", *European Journal of Operational Research*, vol. 248, no. 3, pp. 943–953, Feb. 2016, ISSN: 0377-2217. DOI: 10.1016/j.ejor.2015.08.018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0377221715007559 (visited on 05/23/2019).

[122] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio, "Neural Combinatorial Optimization with Reinforcement Learning", in *International Conference on Learning Representations*, 2017. [Online]. Available: https://openreview.net/forum?id=Bk9mxlSFx.

[123] W. W. M. Kool and M. Welling, "Attention Solves Your TSP, Approximately", *arXiv:1803.08475 [cs, stat]*, Mar. 2018. arXiv: 1803.08475 [cs, stat].

[124] P. Emami and S. Ranka, "Learning Permutations with Sinkhorn Policy Gradient", *arXiv:1805.07010 [cs, stat]*, May 2018. arXiv: 1805.07010 [cs, stat].

[125] A. Nowak, S. Villar, A. S. Bandeira, and J. Bruna, "A Note on Learning Algorithms for Quadratic Assignment with Graph Neural Networks", *arXiv:1706.07450 [cs, stat]*, Jun. 2017. arXiv: 1706.07450 [cs, stat].

[126] E. Larsen, S. Lachapelle, Y. Bengio, E. Frejinger, S. Lacoste-Julien, and A. Lodi, "Predicting Solution Summaries to Integer Linear Programs under Imperfect Information with Machine Learning", *arXiv:1807.11876 [cs, stat]*, Jul. 2018. arXiv: 1807.11876 [cs, stat].

[127] M. Kruber, M. E. Lübbecke, and A. Parmentier, "Learning When to Use a Decomposition", en, in *Integration of AI and OR Techniques in Constraint Programming*, ser. Lecture Notes in Computer Science, Springer, Cham, Jun. 2017, pp. 202–210, ISBN: 978-3-319-59776-8. DOI: 10.1007/978-3-319-59776-8_16.

[128] P. Bonami, A. Lodi, and G. Zarpellon, "Learning a Classification of Mixed-Integer Quadratic Programming Problems", en, in *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, ser. Lecture Notes in Computer Science, Springer, Cham, Jun. 2018, pp. 595–604, ISBN: 978-3-319-93030-5. DOI: 10.1007/978-3-319-93031-2_43.

[129] G. P. McCormick, "Computability of global solutions to factorable nonconvex programs: Part I — Convex underestimating problems", en, *Mathematical Programming*, vol. 10, no. 1, pp. 147–175, Dec. 1976, ISSN: 1436-4646. DOI: 10.1007/BF01580665.

[130] R. K. Ahuja and J. B. Orlin, "Inverse Optimization", *Operations Research*, vol. 49, no. 5, pp. 771–783, Oct. 2001, ISSN: 0030-364X. DOI: 10.1287/opre.49.5.771.10607.

[131] T. C. Y. Chan, T. Craig, T. Lee, and M. B. Sharpe, "Generalized Inverse Multiobjective Optimization with Application to Cancer Therapy", *Operations Research*, vol. 62, no. 3, pp. 680–695, Apr. 2014, ISSN: 0030-364X. DOI: 10.1287/opre.2014.1267.

[132] A. Creswell, T. White, V. Dumoulin, K. Arulkumaran, B. Sengupta, and A. A. Bharath, "Generative Adversarial Networks: An Overview", *IEEE Signal Processing Magazine*, vol. 35, no. 1, pp. 53–65, Jan. 2018, ISSN: 1053-5888. DOI: 10.1109/MSP.2017.2765202.

[133] C. Ansótegui, J. Pon, M. Sellmann, and K. Tierney, "Reactive Dialectic Search Portfolios for MaxSAT", en, in *Thirty-First AAAI Conference on Artificial Intelligence*, Feb. 2017. [Online]. Available: https://www.aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14872 (visited on 05/23/2019).

[134] C. Ansótegui, B. Heymann, J. Pon, M. Sellmann, and K. Tierney, "Hyper-Reactive Tabu Search for MaxSAT", en, in *Learning and Intelligent Optimization*, R. Battiti, M. Brunato, I. Kotsireas, and P. M. Pardalos, Eds., ser. Lecture Notes in Computer Science, Springer International Publishing, 2019, pp. 309–325, ISBN: 978-3-030-05348-2.

[135] K. Li and J. Malik, "Learning to Optimize Neural Nets", *arXiv:1703.00441 [cs, math, stat]*, Mar. 2017. arXiv: 1703.00441 [cs, math, stat].

[136] V. Nair, D. Dvijotham, I. Dunning, and O. Vinyals, "Learning fast optimizers for contextual stochastic integer programs.", in *Conference on Uncertainty in Artifical Intelligence*, Monterey, CA, 2018, pp. 591–600. [Online]. Available: http://auai.org/uai2018/proceedings/papers/217.pdf.

[137] P. Nagarajan, G. Warnell, and P. Stone, "Deterministic implementations for reproducibility in deep reinforcement learning", in *AAAI 2019 Workshop on Reproducible AI*, Honolulu, Hawaii, Jan. 2019.

[138] T. Fitzgerald, Y. Malitsky, B. O'Sullivan, and K. Tierney, "ReACT: Real-Time Algorithm Configuration through Tournaments", en, in *Seventh Annual Symposium on Combinatorial Search*, Jul. 2014. [Online]. Available: https://www.aaai.org/ocs/index.php/SOCS/SOCS14/paper/view/8910 (visited on 05/23/2019).

[139] M. Lindauer and F. Hutter, "Warmstarting of Model-Based Algorithm Configuration", en, in *Thirty-Second AAAI Conference on Artificial Intelligence*, Apr. 2018. (visited on 05/23/2019).

[140] Y. Bengio, S. Bengio, J. Cloutier, and J. Gecsei, "Learning a synaptic learning rule", in *IJCNN*, Seattle, WA, 1991, II–A969.

[141] J. Schmidhuber, "Learning to control fast-weight memories: an alternative to dynamic recurrent networks", *Neural Computation*, vol. 4, no. 1, pp. 131–139, 1992. DOI: 10.1162/neco.1992.4.1.131. eprint: https://doi.org/10.1162/neco.1992.4.1.131. [Online]. Available: https://doi.org/10.1162/neco.1992.4.1.131.

[142] S. Thrun and L. Y. Pratt, Eds., *Learning to Learn*. Kluwer Academic, 1998.

[143] S. Ravi and H. Larochelle, "Optimization as a model for few-shot learning", in *International Conference on Learning Representations*, 2017.

[144] C. Finn, P. Abbeel, and S. Levine, "Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks", in *Proceedings of the 34th International Conference on Machine Learning*, D. Precup and Y. W. Teh, Eds., ser. Proceedings of Machine Learning Research, vol. 70, International Convention Centre, Sydney, Australia: PMLR, Aug. 2017, pp. 1126–1135.

[145] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search", en, *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016, ISSN: 1476-4687. DOI: 10.1038/nature16961.

[146] A. Hussein, M. M. Gaber, E. Elyan, and C. Jayne, "Imitation Learning: A Survey of Learning Methods", *ACM Computing Surveys*, vol. 50, no. 2, 21:1–21:35, Apr. 2017, ISSN: 0360-0300. DOI: 10.1145/3054912.

[147] D. Wierstra, A. Förster, J. Peters, and J. Schmidhuber, "Recurrent policy gradients", *Logic Journal of the IGPL*, vol. 18, no. 5, pp. 620–634, 2010. DOI: 10.1093/jigpal/jzp049. [Online]. Available: http://dx.doi.org/10.1093/jigpal/jzp049.

[148] D. Selsam, M. Lamm, B. Bünz, P. Liang, L. de Moura, and D. L. Dill, "Learning a SAT Solver from Single-Bit Supervision", *arXiv:1802.03685 [cs]*, Feb. 2018. arXiv: 1802.03685 [cs].

[149] K. Smith-Miles and S. Bowly, "Generating new test instances by evolving in instance space", *Computers & Operations Research*, vol. 63, pp. 102–113, Nov. 2015, ISSN: 0305-0548. DOI: 10.1016/j.cor.2015.04.022. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0305054815001136 (visited on 05/23/2019).

[150] Y. Malitsky, M. Merschformann, B. O'Sullivan, and K. Tierney, "Structure-Preserving Instance Generation", en, in *Learning and Intelligent Optimization*, P. Festa, M. Sellmann, and J. Vanschoren, Eds., ser. Lecture Notes in Computer Science, Springer International Publishing, 2016, pp. 123–140, ISBN: 978-3-319-50349-3.

[151] V. T. Paschos, *Applications of combinatorial optimization*. John Wiley & Sons, 2013.

[152] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: a system for large-scale machine learning", in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.

[153] G. B. Dantzig, "Origins of the simplex method", in *A history of scientific computing*, Association for Computing Machinery, 1990, pp. 141–151.

[154] D. P. Bertsekas, *Approximate dynamic programming*, 2008.

[155] 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*, JMLR Workshop and Conference Proceedings, 2011, pp. 627–635.

[156] M. T. Spaan, "Partially observable markov decision processes", in *Reinforcement Learning*, Springer, 2012, pp. 387–414.

[157] T. Lattimore and C. Szepesvári, *Bandit algorithms*. Cambridge University Press, 2020.

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

[159] 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*, 2000, pp. 66–76.

[160] G. Cornuéjols, R. Sridharan, and J.-M. Thizy, "A comparison of heuristics and relaxations for the capacitated plant location problem", *European journal of operational research*, vol. 50, no. 3, pp. 280–297, 1991.

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

[162] P. Erdos and A. Renyi, "On random graphs", *Publicationes Mathematicae*, pp. 290–297, 1959.

[163] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks", *science*, vol. 286, no. 5439, pp. 509–512, 1999.

[164] W. Jakob, J. Rhinelander, and D. Moldovan, *Pybind11 – seamless operability between c++11 and python*, 2017. [Online]. Available: `https://github.com/pybind/pybind11`.

[165] J. Mabille, S. Corlay, and W. Vollprecht, *Xtensor: multi-dimensional arrays with broadcasting and lazy computing*, 2016. [Online]. Available: `https://github.com/xtensor-stack/xtensor`.

[166] W. Vollprecht, *The xtensor vision*, 2018. [Online]. Available: `https://towardsdatascience.com/the-xtensor-vision-552dd978e9ad`.

[167] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, *et al.*, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play", *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[168] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio, "Theano: a cpu and gpu math expression compiler", in *Proceedings of the Python for scientific computing conference (SciPy)*, Austin, TX, vol. 4, 2010, pp. 1–7.

[169] A. Ferber, B. Wilder, B. Dilkina, and M. Tambe, "Mipaal: mixed integer program as a layer", in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, 2020, pp. 1504–1511.