Algorithmic configuration by learning and optimization

Gabriele Iommazzo

INSTITUT POLYTECHNIQUE DE PARIS

NNT : 2021IPPAX105

Thèse de doctorat

Algorithmic Configuration by Learning and Optimization

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à LIX CNRS - École Polytechnique, IP Paris
en cotutelle avec Dipartimento di Informatica - Università di Pisa

École doctorale n°626
École doctorale de l'Institut Polytechnique de Paris (ED IP Paris)
Spécialité de doctorat : Informatique

Thèse présentée et soutenue à Palaiseau, France, le 10/12/2021, par

GABRIELE IOMMAZZO

Composition du Jury :

Frédéric MESSINE Full Professor LAPLACE-CNRS, Toulouse INP, ENSEEIHT, Université de Toulouse	Président
Emilio CARRIZOSA Full professor IMUS, Universidad de Sevilla	Rapporteur
Dominique ORBAN Full professor GERAD & Department of Mathematics and Industrial Engineering, Ecole Polytechnique de Montréal	Rapporteur (absent)
Clarisse DHAENENS Full professor CRIStAL laboratory, Université de Lille, CNRS, Centrale Lille	Examinateur
Sonia VANIER Maîtresse de conférences LIX - CNRS & Université Paris I Panthéon-Sorbonne	Examinateur
Leo LIBERTI Directeur de recherche LIX - CNRS & École Polytechnique, Institut Polytechnique de Paris	Directeur de thèse
Antonio FRANGIONI Full professor Dipartimento di Informatica, Università di Pisa	Co-directeur de thèse
Claudia D'AMBROSIO Directrice de recherche LIX - CNRS & École Polytechnique, Institut Polytechnique de Paris	Co-encadrant de thèse

Algorithmic Configuration by Learning and Optimization

Doctoral Thesis, Ph.D. in Computer Science

Gabriele IOMMAZZO

Last updated on: February 2, 2022

Abstract (in english)

The research topics described in this Ph.D. thesis lie at the intersection of Machine Learning (ML) and Mathematical Programming (MP). The main contributions concern the algorithm configuration problem and the distance geometry problem.

In the first part of the manuscript, we provide introductions to MP and ML

In the second part, we survey the literature on algorithm configuration. We consider *configurable algorithms* $\mathcal{A}^c$, where c is an array of parameters that modify the behavior of the algorithm. For brevity, since we assume that all of the algorithms in this thesis are configurable, we dispense from the c suffix and only write $\mathcal{A}$ instead of $\mathcal{A}^c$.

Given a target algorithm $\mathcal{A}$ and an input π for $\mathcal{A}$, we address the issue of identifying the parameter configuration c^* of $\mathcal{A}$ ensuring the best algorithmic performance $p_{\mathcal{A}}(\pi, c)$ in solving π. This question, known as Algorithm Configuration Problem (ACP), can be formulated as an optimization problem, where the constraints define the set of feasible configurations, and the objective optimizes the performance function $p_{\mathcal{A}}$. Since most algorithms have a very large number of configurable parameters, finding their optimal values for specific algorithmic executions is usually a very hard task to tackle in practice. Therefore, the development of strategies for solving it automatically can benefit several domains.

We propose two novel MP-driven methodologies, using ML paradigms as elements appearing in an MP formulation, to address the ACP. Since algorithmic performance is usually a black-box function (i.e., its closed form expression is unknown), we first train an ML predictor to estimate the behaviour of $\mathcal{A}$. Notably, we learn either: a) an approximation of $p_{\mathcal{A}}$, or b) an approximation of a function mapping π and a required performance level to any configuration achieving it. In a second phase, we translate the mathematical properties underlying the learned approximation into MP terms. We embed these components into an MP formulation. Its objective optimizes the ML-derived predictor; its constraints encode dependency/compatibility conditions over the parameters, required for a configuration to be feasible, and, potentially, other conditions on the learned approximation. This allows us to formulate the ACP by MP and to optimize it, upon the arrival of a new instance $\bar{\pi}$, to retrieve the algorithmic configuration c^* achieving the best (estimated) algorithmic performance for $\bar{\pi}$. Hopefully, c^* is a good

configuration for actually solving the instance $\bar{\pi}$ with $\mathcal{A}$. This framework can be adapted to work with many ML paradigms, as long as one is capable of encoding the learned approximation by MP.

The most prominent methodologies in the literature, treating the ACP as a black-box problem, can only find configurations that are good for a set of instances with similar characteristics. While black-box methodologies may not scale well to settings where the set of feasible configurations is large and c^* depends on the instance at hand, MP solution algorithms should be more efficient, since they exploit the structure of the problem.

Because of our interest in MP, our work in algorithm configuration is motivated by the problem of finding the best parametrization of an MP solution algorithm, deployed on a given problem. Specifically, we employ our approaches for tuning the parameters of an optimization solver, deployed on instances of a hard mixed-integer linear programming problem (see Sec. 2.6). In particular, we investigate how implementation choices in the learning phase affect not only the accuracy of the trained predictor, but also the cost of solving the derived MP, yielding nontrivial trade-offs.

In the third part of the manuscript, we consider a methodology for finding a realization of a simple, undirected, weighted graph, in a Euclidean space of given dimension, where the edges are realized as straight segments of length equal to the edge weights. This is known as the distance geometry problem. A customary approach to it is to solve an MP formulation to determine the position of the vertices in the given Euclidean space. We propose a new MP formulation where, instead, we consider the cycles of the graph, and we decide the length of the segments modelling the edges of each cycle. Our research is partly motivated by the fact that it can serve as a graph embedding methodology, in view of applying vector-based ML paradigms to graphs.

The thesis closes with a brief outline of several possible future research directions, stemming from some of the subjects discussed in this thesis, which were investigated during my Ph. D.

Abstract (en français)

Les sujets de recherche décrits dans cette thèse de doctorat se situent à l'intersection du Machine Learning (ML) et de la Programmation Mathématique (PM). Les principales contributions concernent le problème de la configuration des algorithmes et le problème de la géométrie de la distance.

Dans la première partie du manuscrit, nous présentons la PM et le ML.

Dans la deuxième partie, nous passons en revue la littérature sur la configuration des algorithmes. Nous considérons des algorithmes *configurables* $\mathcal{A}^c$, où c est un tableau de paramètres qui modifient le comportement de l'algorithme. Par souci de concision, puisque nous supposons que tous les algorithmes de cette thèse sont configurables, nous renonçons au suffixe c et écrivons seulement $\mathcal{A}$ au lieu de $\mathcal{A}^c$.

Étant donné un algorithme cible $\mathcal{A}$ et une entrée π pour $\mathcal{A}$, nous abordons la question de l'identification de la configuration de paramètres c^* de $\mathcal{A}$ assurant la meilleure performance algorithmique $p_{\mathcal{A}}(\pi, c)$ dans la résolution de π. Cette question, connue sous le nom de Algorithm Configuration Problem (ACP), peut être formulée comme un problème d'optimisation, où les contraintes définissent l'ensemble des configurations admissibles, et l'objectif optimise la fonction de performance $p_{\mathcal{A}}$. Comme la plupart des algorithmes ont un très grand nombre de paramètres configurables, trouver leurs valeurs optimales pour des exécutions algorithmiques spécifiques est généralement une tâche très difficile à réaliser en pratique. Par conséquent, le développement de stratégies permettant de la résoudre automatiquement peut profiter à plusieurs domaines.

Nous proposons deux nouvelles méthodologies basées sur la PM, utilisant les paradigmes de ML comme éléments apparaissant dans une formulation PM, pour aborder l'ACP. Comme la performance algorithmique est généralement une fonction boîte-noire (c'est-à-dire que son expression analytique est inconnue), nous entraînons d'abord un prédicteur ML pour estimer le comportement de $\mathcal{A}$. En particulier, nous apprenons soit : a) une approximation de $p_{\mathcal{A}}$, soit b) une approximation d'une fonction associant π et un niveau de performance requis à toute configuration permettant de l'atteindre. Dans une deuxième phase, nous traduisons les propriétés mathématiques sous-jacentes à l'approximation apprise en termes de PM. Nous intégrons les composants qui en résultent dans une formulation de PM. Son objectif optimise le prédicteur dérivé par ML ; ses contraintes encodent

les conditions de dépendance/compatibilité sur les paramètres, requises pour qu'une configuration soit admissible, et, potentiellement, d'autres conditions sur l'approximation apprise. Cela nous permet de formuler l'ACP à travers la PM et de l'optimiser, à l'arrivée d'une nouvelle instance $\bar{\pi}$, afin de récupérer la configuration algorithmique c^* réalisant la meilleure performance algorithmique (estimée) pour $\bar{\pi}$. On espère que c^* est une bonne configuration pour résoudre effectivement l'instance $\bar{\pi}$ avec $\mathcal{A}$. Ce cadre peut être adapté pour fonctionner avec de nombreux paradigmes de ML, tant que l'on est capable de codifier l'approximation apprise par la PM.

Les méthodologies les plus importantes de la littérature, traitant l'ACP comme un problème de boîte-noire, permettent uniquement de trouver des configurations qui sont bonnes pour un ensemble d'instances ayant des caractéristiques similaires. Tandis que les méthodologies de type boîte-noire risquent de ne pas être adaptées à des situations où l'ensemble des configurations réalisables est vaste et où c^* dépend de l'instance en question, les algorithmes de solution pour la PM devraient être plus efficaces, puisqu'ils exploitent la structure du problème.

En raison de notre intérêt pour la PM, notre travail sur la configuration des algorithmes est motivé par le problème de la recherche de la meilleure paramétrisation d'un algorithme de solution de PM, déployé sur un problème donné. Plus précisément, nous utilisons nos approches pour régler les paramètres d'un solveur d'optimisation, déployé sur des instances d'un problème difficile de programmation linéaire en variables entières et non (voir Sec. 2.6). En particulier, nous étudions comment les choix d'implémentation dans la phase d'apprentissage affectent non seulement la précision du prédicteur entraîné, mais aussi le coût de la résolution du program mathémathique que nous en dérivons, donnant lieu à des compromis non triviaux.

Dans la troisième partie du manuscrit, nous considérons une méthodologie pour trouver une réalisation d'un graphe simple, non dirigé et pondéré, dans un espace euclidien de dimension donnée, où les arêtes sont réalisées comme des segments droits de longueur égale aux poids des arêtes. Ce problème est connu sous le nom de problème de géométrie de la distance. Une approche habituelle de ce problème consiste à résoudre une formulation de PM pour déterminer la position des sommets dans l'espace euclidien donné. Nous proposons une nouvelle formulation PM où, à la place, nous considérons les cycles du graphe, et nous décidons de la longueur des segments modélisant les arêtes de chaque cycle. Notre recherche est en partie motivée par le fait qu'elle peut servir de méthodologie de plongement de graphes, afin d'appliquer aux graphes des paradigmes de ML basés sur les vecteurs.

La thèse se termine par un bref aperçu de plusieurs directions de recherche futures possibles, découlant de certains des sujets discutés dans cette thèse, qui ont été étudiés pendant mon doctorat.

Acronyms

Below, we provide a table with with the acronyms used in the thesis.

extended name	shorthand
Algorithm Configuration Problem	ACP
Algorithm Selection Problem	ASP
Branch-and-Bound	B&B
Configuration Space Search Problem	CSSP
Convex Programming	CP
Cross-Validation	CV
Decision Tree	DT
Design Of Experiments	DOE
Distance Geometry Problem	DGP
Empirical Risk Minimization	ERM
Euclidean Distance Geometry Problem	EDGP
Feature Selection	FS
In-Sample	IS
Hydro Unit Commitment	HUC
Karush-Kuhn-Tucker	KKT
Knowledge-Encoding Process	K-EP
Linear Programming	LP
Local Search Heuristic	LSH
Logistic Regression	LR
Mixed-Integer Linear Programming	MILP
Mixed-Integer Nonlinear Programming	MINLP
Machine Learning	ML
Mathematical Programming	MP
MultiStart	MS
Nested Cross-Validation	NCV
Neural Network	NN
Nonlinear Programming	NLP
Out-Of-Sample	OS
Performance As Input	PaI
Performance As Onput	PaO
Per-Problem	PP
Per-Instance	PI
Quadratic Programming	QP
Regularized Risk Minimization	RRM
Structural Risk Minimization	SRM
Support Vector Regression	SVR
Unit Commitment Problem	UCP
Vapnik-Chervonenkis	VC
Variable Neighborhood Search	VNS

Table 1: Thesis acronyms

Contents

Chapter 1

Introduction

In this thesis, we are going to consider research topics at the interface of MP and ML.

MP, discussed in Ch. 2, is a language for formally stating and solving optimization problems; its sentences are called "formulations". A "problem" is an infinite collection of couples of the type (Q, A), all sharing certain structural properties, whereby Q is an input question and A is the output answer to that question [99]. We assume that any problem can be always represented by a collection of strings, thereby, parameterized by a set of symbols. In this setting, an "instance" is the assignment of values to the input symbols (i.e., a specific question), while a "solution" is the assignment of values to the output symbols, for a given instance.

In some problems, called "decision problems", the solution can only be YES or NO. A sample decision problem is deciding whether a natural number is prime. In this thesis, we address "optimization problems", whose solution is the best answer out of a set of feasible ones. One widely investigated optimization problem is the Travelling Salesman Problem (TSP). The TSP question is formulated as follows: given a set of locations and the distances between them, which is the shortest route for visiting all the locations once, and returning to the first city at the end of the route? The TSP is considered as an infinite collection of all finite, labelled graphs.

We remark that the structural properties shared by problem members, notably, the relationship between questions and answers, are usually represented by a sentence, parametrized by the symbols of Q and A. Since the problems we treat in this thesis can all be cast as optimization problems and, thereby, translated into MP, we call "formulation" the sentence characterizing a problem. Then, solving a problem involves interpreting its formulation according to the values of a specific instance, in order to find the corresponding solution.

ML is a set of techniques for devising and implementing algorithms, in order to construct hypotheses from data (see Ch. 3). We consider an algorithm either as pseudo-code or, more formally, as written in a high-level programming language. The ML problem

instances are data points. Its solutions, the hypotheses, are functions, parametrized by an array of coefficients, approximating some properties of the data. Since the distribution of the ML problem instances is almost always unknown (otherwise, there would be no need to solve it), the only way to build and assess a hypothesis is by using a known subset of the data, called "training set", appropriately sampled. Given a training set, the purpose of the associated "training problem" is to learn the optimal coefficients of a hypothesis, i.e., such that the hypothesis can achieve maximum generalization. A hypothesis is said to "generalize" well when it provides reasonably accurate approximation estimates, even at points outside the training set.

This thesis often refers to ML "paradigms" and "predictors". A ML paradigm is a framework for learning hypotheses, comprising (learning) algorithms, methods (for overall use) and methodologies (for application to practical cases). A trained predictor is the solution of the training problem, i.e., a hypothesis equipped with the optimal coefficients, learned by training. We also investigate the "inference problem", related to computation of a trained ML predictor for a given input. Given a training problem and its solution, the inference problem aims at determining a set of statements, written as arithmetical expressions in equalities/inequalities, or as memberships in certain sets of numbers (integer, real, etc..), encoding the predictor as accurately as possible: evaluating the system for a given input yields the predictor approximation estimate for that input.

Thus, the inference problem is a "meta-problem", in the sense that its input is another problem.

Since the training and inference problems are both optimization problems, they can be formulated by MP.

Given a configurable algorithm and its input, the main problem we consider in this thesis aims at finding the configuration which ensures the best algorithmic performance when the algorithm is run on its input. This is known as the ACP, and we survey the relevant literature in Ch. 4. Although the ACP clearly stands out as an optimization problem, the informal definition given above has many ambiguous terms: configurable algorithm, algorithmic performance, and what it means for an algorithmic performance to be "best". In the Turing Machine (TM) computational model, algorithms simply run on the TM with the input written on the tape. In most cases, part of the input of algorithms does not necessarily represent an instance of the problem being solved, but rather instructions to the algorithm itself, encoded by parameters. For example, an algorithm might be configured to print out its outputs to screen rather than send it to the printer. Then, the part of the input we single out as "algorithmic configuration" is an instantiation of the parameters of the target algorithm, determining the arrangement and interactions of its internal components. Accordingly, the algorithmic configuration itself may be a complicated array of data of different type: boolean, numeric, cate-

gorical. More relevant to the case in point, an algorithm for solving an optimization problem represented as an MP formulation, called "solver", is usually a collection of different (implemented) algorithms, each of which can be deployed together with other algorithms, or alternately to them. Since each instance may need ad-hoc solution procedures, a good solver configuration may enhance its performance, i.e., speed up the solution process or improve the quality of the solution returned by the solver within a given time limit. More in general, the "performance" of an algorithm is the outcome of executing it on an instance and with a specific configuration; it can be measured in terms of CPU time, quality of the returned solution, etc.

The implementation of an approach to the ACP requires dealing with two issues: the (potentially very large) size of the configuration set (which grows at least as an exponential function of the number of parameters), and the fact that the algorithmic performance is generally black-box (i.e. no explicit algebraic representation of it is available) and very costly to measure. The first issue requires procedures that can search large sets efficiently. The second one requires a performance estimate that can guide the exploration of the set of feasible algorithmic configurations: this is achieved by sampling the performance at a small set of carefully selected points (i.e., running the algorithm on a handful of instances and configurations) and/or by deploying more sophisticated ML techniques. In this thesis, we propose two novel methodologies for addressing the ACP, based on a combination of MP and ML. We present them in Ch. 5 and 6).

This thesis also considers simple, undirected graphs, whereby the vertex positions are unknown and the edges are labeled by a weight/length function. Given an integer K, the Distance Geometry Problem (DGP) on such a graph aims at reconstructing the position of the graph vertices, in a Euclidean space of dimension K, from the known edge weights, i.e., at finding a "realization" of the graph. In the DGP, a realization is obtained by mapping the graph to a set of vectors of dimension K such that, for any two vertices incident on an edge, their distance must be close as possible to the corresponding edge weight. This issue is discussed in Ch. 7. We propose a novel methodology for solving the DGP. Although there is a vast literature for solving the DGP, MP-based methodologies allow very robust solutions. Our contribution is a novel MP formulation. Our research is partly motivated by the fact that it can be instrumental in applying vector-based ML techniques to graphs. In fact, several ML paradigms applied on graphs require an embedding of the input graph into some topological space. MP formulations of the DGP may achieve this in a much more efficient way than other solution strategies.

1.1 List of publications

The thesis is based on the research presented in the following papers:

Paper I [94] G. Iommazzo, C. D'Ambrosio, A. Frangioni, L. Liberti (2020), *Learning to configure mathematical programming solvers by mathematical programming*. In P. Pardalos, M. Brunato (Eds.), Learning and Intelligent Optimization (LION14), Lecture Notes in Computer Science, 12096:377–389, Springer Cham., 2020.

Paper II [93] G. Iommazzo and C. D'Ambrosio and A. Frangioni and L. Liberti (2020),*A learning-based mathematical programming formulation for the automatic configuration of optimization solvers*. In G. Nicosia et al. (Eds.), Machine Learning, Optimization, and Data Science (LOD2020), Lecture Notes in Computer Science, 12565:700–712, Springer Cham.

Paper III [120] L. Liberti, G. Iommazzo, C. Lavor and N. Maculan (2021), *A cycle-based formulation for the Distance Geometry Problem*. In C. Gentile et al. (Eds.), Graphs and Combinatorial Optimization: from Theory to Applications (CTW2020), AIRO Springer Series, 5:93–106, Springer, Cham.

Paper IV L. Liberti, G. Iommazzo, C. Lavor, N. Maculan, *Cycle-based formulations in Distance Geometry*, submitted to journal.

Paper V G. Iommazzo, C. D'Ambrosio, A. Frangioni, L. Liberti, *The algorithm configuration problem*, to appear in the 3rd edition of the Encyclopedia of Optimization, Springer Nature.

Part I

Generalities

Chapter 2

An overview of mathematical programming

2.1 Introduction

A decision problem P encodes a formal YES/NO question, parametrized by a set of input and output symbols which can be assigned values. The assignment of values to the input symbols is an *instance*, and decision problems are assumed to have an infinite number of instances. The assignment of values to the output symbols is a *solution* of P.

Mathematical optimization is a field of mathematics [65] related to *optimization problems*, whereby a solution is selected from a *feasible* (i.e., admissible) set of output assignments. The interest in mathematical optimization comes from the fact that countless practical applications, arising from diverse fields such as engineering, finance, economics, chemistry, computer science, etc., can be cast as optimization problems.

MP is a formal language for describing optimization problems and a framework for solving them. Each formal MP sentence, called *formulation*, is recursively built from a set of entities: index sets and parameters (encoding the problem input), decision variables (encoding the solution for a specific instance), objective function (for assessing variable assignments), and constraints of various types (defining the available decisions). In the following, we will be adopting the abbreviation "MP" both for "mathematical programming" (the language/framework) and "mathematical program" (an MP formulation), whenever the context allows for unambiguous use of the term.

This chapter is devoted to providing some basic notions about MP. Aside from the fact that Ph.D. theses are required to contain an introduction to whatever subject they are about, we are further motivated by three observations. Firstly, we see both the ACP and the DGP as optimization problems: the former finds the configuration offering the best algorithmic performance, the latter searches for a realization of a

graph in a given Euclidean space. Further, the methodologies proposed by the authors of this Ph. D. thesis to address the ACP and the DGP adopt MP algorithms to try to solve them; in this chapter, we discuss some of these algorithms. In particular, our ACP approaches involve training ML predictors, which requires solving optimization problems. Secondly, we study the application of the ACP to solution algorithms for MP. From the point of view of MP as a formal language, we use the MP syntax (the formulations) in order to represent the ACP; and we apply the ACP to the algorithms that provide the semantics of MP (i.e., solvers, deciding the assignment of optimal values to the decision variables).

2.2 The components of a mathematical program

The MP formulation of an optimization problem is as follows:

$$\begin{aligned} \min_{x \in \mathcal{D}} \quad & f(x) \\ \forall i \in \mathcal{I} \quad & g_i(x) \leq 0 \\ \forall j \in \mathcal{E} \quad & h_j(x) = 0\,, \end{aligned} \tag{2.1}$$

whereby $|\mathcal{I}| = m$ and $|\mathcal{E}| = m'$, $f : \mathbb{R}^d \longrightarrow \mathbb{R}$, $g_i : \mathbb{R}^d \longrightarrow \mathbb{R}^d$ and $h_j : \mathbb{R}^d \longrightarrow \mathbb{R}^d$. The formulation in Eq. (2.1) is the *standard form* of an MP. Its domain is the closed set

$$\mathcal{D} = \mathsf{dom}\, f \cap \bigcap_{i \in \mathcal{I}} \mathsf{dom}\, g_i \cap \bigcap_{j \in \mathcal{E}} \mathsf{dom}\, h_j\,.$$

Throughout this chapter, we assume that we are always optimizing over closed sets, and will always refer to $\mathcal{D}$ as the domain of an MP formulation.

The components of the problem in Eq. (2.1) are [115]:

- *decision variables*: a set of quantities x, representing decisions to be taken and encoding an output solution;
- *constraints*: a set of statements defining the domain and feasible values of the variables. The constraints of an MP formulation are of two types: explicit and implicit. Explicit constraints are written by means of arithmetical expressions of variables, parameters, equality/inequality symbols and/or optimization operators (e.g., "$2x_1 - 3x_2^4 = 0$"). Implicit constraints are statements expressed as membership in certain classes of sets (e.g., "$x \in \mathbb{Z}$");
- *objective function*: a couple, containing an optimization operator (i.e., min or max) and an arithmetical expression $f(x)$ of variables, parameters and/or optimization operators. An optimization operator defines the optimization direction: it specifies whether an expression must be minimized or maximized. The purpose of the objective function is to allow the assessment of different value assignments to variables, in order to determine an optimal one;

- *parameters*: a set of given input coefficients, univocally identifying a problem instance. In Eq. (2.1), the parameters are: a function f, the finite index sets of the inequality and equality constraints $\mathcal{I}, \mathcal{E}$ the constraint functions g_i, for all $i \in \mathcal{I}$ and h_j, for all $j \in \mathcal{E}$;

A point $\hat{x}$ is *feasible* with respect to the problem in Eq. (2.1) if it satisfies all the constraints, for given input parameters. We let

$$\mathcal{F} = \{x \in \mathcal{D} \mid \forall i \in \mathcal{I}\ g_i(x) \leq 0,\ \forall j \in \mathcal{E}\ h_j(x) = 0\} \tag{2.2}$$

be the *feasible set* of Eq. (2.1), containing the points satisfying all the constraints, so that Eq. (2.1) can be rewritten, compactly, as

$$\min_{x \in \mathcal{F}} f(x)\,.$$

If $\mathcal{F} = \emptyset$, we say that the MP in Eq. (2.1) is *infeasible*. Instead, we say that Eq. (2.1) is *unbounded if, for all objective function values* m*, there always exists a point* $x' \in \mathcal{F}$ *such that* $f(x') < \mathsf{m}$. If an MP is not infeasible nor unbounded, and there an optimal solution, we call p^* the optimal value of that solution. We remark that symbols denoting optimal objective function values might be assigned two special symbols, $\varnothing$ or ∞, which means that the underlying MP is, respectively, unbounded or infeasible.

For a feasible solution $\hat{x}$ of Eq. (2.1) to be a *local optimum*, $\hat{x}$ must be such that there exist $\varepsilon > 0$ and a neighbourhood $\mathcal{U}_\varepsilon(\hat{x}) \cap \mathcal{F}$ where, for all $x \in \mathcal{U}_\varepsilon(\hat{x}),\ f(\hat{x}) \leq f(x)$. If, instead, $f(\hat{x}) \leq f(x)$ for all $x \in \mathcal{F}$, then $\hat{x}$ is also a *global minimum* of Eq. (2.1).

In MP, it is common to resort to reformulation techniques. A *reformulation* Q of a problem P is an auxiliary problem having some properties in common with P. For a detailed account of reformulations, we refer to [114]. Reformulations are customarily applied in MP to cast the original problem into one that is easier to solve. In fact, the formulation adopted for Q can capture and encode some useful structural properties of P, which can then be exploited by appropriate solution algorithms. Q may: a) preserve all or some of the (local/global) optimality properties of P, b) be obtained from P by removing some of its constraints (relaxations), or c) approximate some of the components of P (approximations, which may or may not supply optimality guarantees).

Relaxations and approximations are especially important, because they yield bounds on the optimal value of the objective function of P. These bounds can then be exploited by optimization algorithms — such as the Branch-and-Bound (B&B) [107] — to solve diverse problems in several different MP classes, such as Mixed-Integer Linear Programming (MILP), Nonlinear Programming (NLP) and Mixed-Integer Nonlinear Programming (MINLP). Another widely used reformulation is the "dual" Q of an MP P (here, the "primal"). In the words of [118], the dual Q is such that the "decision variables of P are used to index the constraints of Q, and the constraints of P are used to index the

variables of Q". Solving the dual is also conductive to the computation of the duality gap. We let $\hat{p}$ the primal objective function value at a primal-feasible point $\hat{x}$, $\hat{q}$ the dual objective function value at a dual-feasible point $\hat{y}$, and we define the *duality gap* as

$$\|\hat{p} - \hat{q}\| \geq 0\,. \tag{2.3}$$

Moreover, for any primal-dual feasible couple, duality provides a bound on the primal formulation, a property called *weak duality*. For minimization problems such as the one formulated by Eq. 2.1, weak duality corresponds to:

$$\hat{q} - \hat{p} \leq 0\,, \tag{2.4}$$

i.e., $\hat{q}$ is a lower bound on any primal solution value. For maximization problems, solving the dual formulation supplies an upper bound on primal solution values. Unboundedness of the dual (sometimes represented as $\hat{q} = +\infty$), implies infeasibility of the primal, and viceversa.

There are several algorithms for solving optimization problems in one or more MP classes. *Solvers* are software applications implementing one or more of such algorithms. They accept a problem instance and, possibly, a formulation of the problem and specific operating instructions (conditioning their behaviour) as inputs. Their outputs are: a) an optimal feasible solution (or a set thereof), which solvers attempt to compute for the given input; b) possibly, information about the solution process. Solvers provide the semantics of MP formulations, by deciding the optimal value assignment values to the decision variables.

Optimization algorithms usually start from a (possibly, feasible) point, then iteratively explore the feasible set, searching for a point capable of improving on solution value. Many of these algorithms, especially those capable of providing optimality guarantees, depend on the gap in Eq. (2.3) to assess the improvement at each iteration. In fact, their termination criterion is usually

$$\|\hat{p} - \hat{q}\| \leq \epsilon\,,$$

where $\epsilon > 0$ is a tolerance parameter; in this case, we say that an ϵ-*optimal* solution has been attained. *Strong duality* applies when the gap in Eq. (2.3) is zero, which occurs at any globally optimal primal-dual solution.

In the following sections, we give an overview of some important classes of MP. Since an extensive treatment of them is beyond the scope of this manuscript, we will avoid any proofs and provide bibliographic references for further study.

2.3 Convex Programming

Solving an MP is, in general, difficult: it has been shown that MP is uncomputable (as a class) [97, 116] and that even those MPs that are decidable are still **NP**-hard. However

for at least one MP class, Convex Programming (CP), it is always possible to guarantee the existence of a solution with certain desirable properties. Some problems in this class are still **NP**-hard [48, 160], but many others are tractable. Notably, although exact solution algorithms running in polynomial time only exist for two CP classes — namely, Linear Programming (LP) (presented in Sec. 2.5) and Quadratic Programming (QP, i.e., minimizing a quadratic function over linear constraints) [29, §4] — it is always possible to compute an approximate solution of a CP in polytime [148].

The MP defined in Eq. (2.1) is a CP if f and $\mathcal{F}$ are convex. For this to be true, it is sufficient that [150, §12.4]: a) for all $j \in \mathcal{E}$, $h_j(x) := a_j^\top x + b_j$, i.e., the equality constraints are affine (hence, convex); b) for all $i \in \mathcal{I}$, $g_i(x)$ are convex. The main property of a CP is the equivalence of local and global optimality: any local minimum is also a global minimum. For this reason, local algorithms for convex optimization are guaranteed to work globally.

The reason why CP is important for this thesis is that many optimization problems can be formulated as CPs [29], including many arising in ML such as support vector machines, logistic regression, ridge regression, and others [33]. Moreover, some well-known DGP formulations are convex; see, e.g., Eq. (7.6) in Sec. 7.4. The interested reader will find further details in textbooks such as, say, [29] or [146].

2.4 Lagrangian duality and optimality conditions

We would like to know under which conditions the gap in Eq. (2.3) is zero, i.e., when strong duality applies. Lagrangian duality (see, e.g., [20, §6]) allows us to derive these conditions: it is a reformulation technique for transforming constrained optimization problems into (partially) unconstrained ones, by placing the constraints in the objective function and penalizing any violations to them.

We apply Lagrangian duality to Eq. (2.1).

The first step is to build the *Lagrangian function*, i.e., the map

$$\begin{aligned} L &: \mathbb{R}^d \times \mathbb{R}^m \times \mathbb{R}^{m'} \longrightarrow \mathbb{R} \\ L(x, \lambda, \mu) &= f(x) + \sum_{i \in \mathcal{I}'} \lambda_i g_i(x) + \sum_{j \in \mathcal{E}'} \mu_j h_j(x)\,, \end{aligned} \tag{2.5}$$

where $\mathcal{E}' \subseteq \mathcal{E}$, $\mathcal{I}' \subseteq \mathcal{I}$, the coefficients $\{\lambda_i\}_{i \in \mathcal{I}'}$ and $\{\mu_j\}_{j \in \mathcal{E}'}$ are the *Lagrangian multipliers* and, for all $i \in \mathcal{I}'$, $\lambda_i \geq 0$.

Then, we define the *Lagrangian dual function* as the concave map

$$\begin{aligned} q &: \mathbb{R}^m \times \mathbb{R}^{m'} \longrightarrow \mathbb{R} \\ q(\lambda, \mu) &= \inf_{x \in \mathcal{F}} L(x, \lambda, \mu)\,. \end{aligned} \tag{2.6}$$

Eq. (2.6) is the *Lagrangian relaxation* of the MP in Eq. (2.1), providing a lower bound on its optimal value p^*. The Lagrangian relaxation can be seen as an infinite family of

problems, parametrized by the Lagrangian multipliers. In fact, the tightest lower bound is achieved by solving the *Lagrangian dual*

$$\begin{aligned} \max_{\lambda,\mu} \quad & q(\lambda,\mu) \\ & \lambda \geq 0\,, \end{aligned} \tag{2.7}$$

a convex MP formulation, whose optimal value we call q^*, when it exists.

The *Karush-Kuhn-Tucker* (KKT) conditions (well described in, say, [15, §3] or [150, §12]) for the problem in Eq. (2.1) are as follows:

$$\begin{aligned} \forall i \in \mathcal{I} \qquad g_i(x) &\leq 0 && (\textit{primal feasibility I}) \\ \forall j \in \mathcal{E} \qquad h_j(x) &= 0 && (\textit{primal feasibility II}) \\ \forall i \in \mathcal{I} \qquad \lambda_i &\geq 0 && (\textit{dual feasibility}) \\ \nabla_x L(x,\lambda,\mu) &= 0 && (\textit{stationarity}) \\ \forall i \in \mathcal{I} \qquad \lambda_i g_i(x) &= 0\,, && (\textit{complementary slackness}) \end{aligned} \tag{2.8}$$

We assume that the objective and all constraint functions of the MP in Eq. (2.1) are continuously differentiable in a neighbourhood of x. ∇ is the function vector

$$\nabla f = \left(\frac{\partial f(x)}{\partial x_1}, \frac{\partial f(x)}{\partial x_2}, \cdots, \frac{\partial f(x)}{\partial x_d} \right), \tag{2.9}$$

representing the gradient of a function and x being the vector $(x_1, x_2, \ldots, x_d)$. We will employ the shorthand $\nabla f(\bar{x})$ to refer to Eq. (2.9), evaluated at $\bar{x} \in \mathbb{R}^d$.

Eq. (2.8) allow us to make some statements about the optimality of Eq. (2.1). Firstly, if there exists a primal feasible point x^* and dual feasible Lagrangian multipliers (λ^*, μ^*) at which strong duality holds, then the KKT conditions are satisfied at those points (necessity). Secondly, if there exists primal-dual feasible points x^*, λ^*, μ^* satisfying the KKT conditions in Eq. (2.8), then x^*, λ^*, μ^* are primal-dual optimal (sufficiency).

A remarkable result of Eq. 2.8 is that it provides sufficient conditions for global optimality (and, thereby, strong duality), if the corresponding primal formulation is a CP, or local optimality, otherwise. However, this relies on the assumption that a KKT-compliant point exists, which may not always be the case; notably, a dual optimal point may not satisfy the complementary slackness conditions of Eq. 2.8. For such a primal-dual optimal point to exist, specific *constraint qualification conditions* must be satisfied. Among the many applicable constraint qualification conditions, the *Slater's conditions* [15, §5], for example, require that: a) for all $i \in \mathcal{I}$, the g_i are convex, b) for all $j \in \mathcal{E}$, the h_j are affine, and c) there exists a point $\hat{x}$ in the interior of $\mathcal{D}$, such that all $g_i(\hat{x}) < 0$ and all $h_j(\hat{x}) = 0$. Slater's conditions holding at a primal feasible optimum are sufficient to ensure the existence of an optimal Lagrangian multipliers, satisfying the KKT conditions in Eq. (2.8) and achieving local/global optimality.

We remark that, if the MP at hand is nonconvex, the KKT conditions are necessary for local optimality. However, information from the gradient may not be enough to

ascertain in which directions the objective function increases/decreases. Therefore, in order to identify a point as optimal, second-order conditions are required, to analyze the curvature of the Lagrangian function in Eq. (2.5) in undecided directions. These conditions "examine the terms of the second derivative in the Taylor series expansions of f, g_i and h_j, to see if this extra information resolves the issue of increase or decrease of f". See, [150, §12.4] for further details).

2.5 Linear Programming

LP is a subfield of convex optimization, for formulating problems characterized by a linear objective and linear constraints. A generic LP has the following formulation:

$$\begin{aligned} \min_{x \in \mathbb{R}^d} \quad & \langle c, x \rangle \\ & Ax = b \\ \forall i \leq d \quad & x \geq 0\,, \end{aligned} \tag{2.10}$$

for a matrix $A \in \mathbb{R}^{m \times d},\ m > 0,\ c \in \mathbb{R}^d,\ b \in \mathbb{R}^m$. The LP in Eq. (2.10) has an optimum only if it is feasible and bounded. The associated dual

$$\begin{aligned} \max_{y \in \mathbb{R}^m} \quad & \langle y, b \rangle \\ & yA^\top \leq c \\ \forall j \leq m \quad & y \geq 0 \end{aligned} \tag{2.11}$$

is also an LP. Weak duality always applies to Eq. (2.10)–(2.11), for any feasible primal-dual couple $(\bar{x}, \bar{y})$, i.e.:

$$\langle \bar{y}, b \rangle \leq \langle c, \bar{x} \rangle\,.$$

Strong duality is attained by any feasible primal (or dual) optimal solutions $\hat{x}, \hat{y}$ satisfying the complementary slackness conditions

$$\hat{y}(b - A\hat{x}) = 0\,.$$

The main families of solution algorithms for LP are the *simplex* one and the *interior point* one [150, §13–14]: they can solve large instances (up to millions of constraints/variables) quickly and efficiently. In terms of worst-case behaviour, the simplex algorithm has been proven to have exponential complexity on some problem classes. However, its average running time is polynomially bounded [170, §11], which means that it is extremely fast in practice on most problems. Interior-point methods, instead, have been demonstrated to have polynomial complexity, although they are often faster only on large and sparse problems [158, 6].

2.6 Mixed-Integer Linear Programming

MILP comes from allowing integer variables into the LP in Eq. (2.10). A generic MILP formulation is as follows:

$$\begin{aligned} \min \quad & \langle c, x \rangle \\ & Ax = b \\ \forall w \in \mathcal{W} \quad & x_w \geq 0 \\ \forall u \in \mathcal{U} \quad & x_u \in \mathbb{Z}\,, \end{aligned} \tag{2.12}$$

for a matrix $A \in \mathbb{R}^{m \times d_x + d_z}$, d_w being the number of continuous variables and d_u being the number of integer variables. MILP has been shown to be **NP**-complete [170, 102]; the main challenge it raises lies in the nonconvexities generated by the integrality constraints.

We are interested in this MP class because a wealth of practical problems can be cast as MILPs, including many nonlinear problems, which can be approximated with sufficient accuracy by MILP formulations. A well-known example of a hard-to-solve MINLP which can be formulated as a MILP is the Unit Commitment Problem (UCP) [154, 180], a broad family of optimization problems. The UCP is also one of the major problems in the field of electricity management/production. Recognising the importance of this problem, we decided to use UCP instances as a benchmark set in the computational experiments conducted to test our ACP approaches. These experiments are discussed in finer detail in Ch. 5 and 6; the selected UCP formulation is an MILP (see Sec. 5.5.1), approximating the nonlinearities through linear constraints.

Theoretical results and common algorithmic methodologies for LP are commonly applied to MILP: the solution of an MILP often involves solving smaller subproblems, formulated by LP and approximating the initial MILP. The purpose of these subproblems is to to quickly derive bounds on the objective function value of the MILP. They are solved efficiently, by deploying the fast LP algorithms mentioned in Sec. 2.5. For this reason, in general, MILP formulations are easier to solve than the MINLP ones. In Sec. 5.5.4, we will discuss MILP and MINLP formulations for the ACP, and observe that the former can be solved considerably quicker.

A customary approach to address an MILP P is to relax the integrality constraints and then consider the derived *continuous relaxation* Q. Q is an LP, hence it can be solved efficiently. As a relaxation, Q supplies a lower bound on the objective function value of P, when P is a minimization problem, or an upper bound if P is a maximization problem. If an optimal solution of Q is also integer — i.e., feasible for P — it is also an optimal solution for P. Further, under certain conditions on the structure of Q (for example, total unimodularity [170, §19]), we can be certain that its solutions are all integer. Such properties are famously exploited in the B&B algorithmic framework, which is at the core of all the most advanced MILP solvers are all based on.

2.6.1 Common solution algorithms

In this section we overview some common algorithmic techniques for solving MILP formulations.

2.6.1.1 B&B

The B&B [107] is a *divide et impera* type algorithm, which explores the search space of an MILP formulation through the enumeration of all possible solutions to the problem in question by a tree. Each node of the tree contains a subproblem; after solving the subproblem at a node, the B&B processes the children nodes recursively.

At the current subproblem, a lower and an upper bound on the optimal objective function value are computed. The former comes from solving a relaxation — usually the continuous one — whose solution, however, is not guaranteed to be feasible. The latter is usually produced by some heuristic, searching for a feasible solution. In addition to these local bounds, globally valid upper and lower bounds are kept during the tree exploration; the global bounds are computed at the root node and updated at the subproblem nodes, when possible. We call the globally valid upper bound the *incumbent bound*, and the related solution *incumbent*.

If the lower bound at the current node is larger than the incumbent one (e.g., the subproblem is infeasible), the sub-tree rooted in that node is discarded, as it is incapable of improving upon the incumbent and provide a global optimum. Instead, if the current lower bound is smaller than the incumbent one, the difference between the bounds at the current node is smaller or larger than a user-defined threshold. In the first case, the current subproblem produces an approximate global optimum, whose approximation depends on the threshold; the incumbent, incumbent bound and globally valid lower bound may be updated, if necessary. Otherwise, the search continues by *branching* on the current subproblem's feasible region, which is partitioned into two (or more) subsets (say, by imposing constraints on a variable), that are recursively explored at the children nodes.

The B&B is run until all of the subproblems have been processed, or until some termination criterion (e.g., a time or iteration limit) is verified.

Branching, for decomposing the relaxation into two subproblems, is one of the two main *constraint enforcement* techniques. Constraint enforcement is a family of techniques for eliminating relaxation solutions that are infeasible for the original problem. The other important constraint enforcement technique is *refinement*, for tightening the relaxation via cutting planes. The Branch-and-Cut algorithm is the most widely used variant of the B&B, and is is based on refinement. It is employed when the LP solution of the continuous relaxation is not MILP-feasible: it introduces additional constraints (called "cutting planes" or, simply, "cuts") to the continuous relaxation, and then tries

to solve it again.

Our argument for mentioning the B&B is that the most popular algorithms for treating (MI)NLP problems are based on variations of this algorithmic framework.

2.6.1.2 Cutting planes

Cutting planes are at the heart of the Branch-and-Cut. We let $X \subseteq \mathbb{R}^n$ be a set of points and

$$\mathsf{conv}(X) = \{\bar{x} = \textstyle\sum_i x_i\beta_i \mid \forall i, \beta_i \geq 0 \quad \text{and} \quad \sum_i \beta_i = 1\} \tag{2.13}$$

be the *convex hull* of X, that is, the (convex) set of all convex combinations of points in X. Eq. (2.13) is a polyhedron [29, §2], i.e., the set constructed by intersecting a finite number of of halfspaces (linear inequalities) and hyperplanes (linear equalities).

Given an MILP P, with feasible region $\mathcal{F}$, it is known that optimizing over

$$\tilde{\mathcal{F}} = \mathsf{conv}(\mathcal{F})$$

gives the same optimal solutions as P. Furthermore, this means solving an LP, owing to the structure of Eq. (2.13). We refer to this LP as Q. However, to actually formulate it, one needs to know all of the points in $\mathcal{F}$, which makes describing Q hard. Moreover, even assuming that such information is available (which, typically, is unlikely), if $\mathcal{F}$ contained (infinitely) many points, Q might be "too large" to be treated efficiently (although algorithmic strategies have been developed to tackle with this issue, e.g., the Danzig-Wolfe algorithm [47]).

To overcome this challenge, one can approximate Q by first computing a continuous relaxation $\bar{Q}$ of P — with feasible region $\bar{\mathcal{F}}$ — and then "slicing" $\bar{Q}$, to tighten it. The slicing is obtained by adding constraints such that the related hyperplane is close to $\tilde{\mathcal{F}}$ or belongs to it (i.e., it defines one of its facets). This is accomplished by solving the *separation problem*, i.e., given $\bar{x} \in \bar{\mathcal{F}}$, by finding a *cutting plane* $dx - \delta$, such that the inequality

$$dx \leq \delta \tag{2.14}$$

is satisfied by all points of Q and only violated by $\bar{x}$ (i.e., $d\bar{x} > \delta$). Eq. (2.14), called *valid inequality* of Q for separating $\bar{x}$ from Q, is always guaranteed to exist [170, §23].

A *cutting plane algorithm* is a procedure which, iteratively: solves the continuous relaxation $\bar{Q}$ and tests the obtained solution for belonging to $\tilde{\mathcal{F}}$. If it does not, a cut is produced (by solving the separation problem), and added to $\bar{Q}$; then, $\bar{Q}$ is solved. These steps are repeated until an optimal MILP-feasible solution is found.

2.6.1.3 Local search heuristics

Many problems in combinatorial optimization can be cast as MILPs. A *combinatorial optimization problem* is an optimization problem where the set of solutions is finite,

i.e., the feasible region $\mathcal{F}$ is discrete. Although $\mathcal{F}$ typically has a concise representation (e.g., a graph), its size may be very large [171, §1], hence exhaustively searching it is generally not possible. Instead, approximate algorithms, tailored to the problem at hand, are usually deployed. These algorithms, called *local search heuristics* (LSH), do not provide guarantees on the global optimality of the attained solutions, but are generally capable of producing a good feasible solution in a reasonable time.

The reason why we discuss LSHs is that many methodologies for addressing the ACP, that we will survey in Ch. 4, rely on them (and on metaheuristics, described in the next section), to explore the feasible region of an ACP instance, i.e., the set of combinations of parameter values of a given algorithm.

LSHs are iterative algorithms which, at each iteration, will: a) take a putative feasible starting point in input, b) start exploring the neighbourhood of that point, and c) return, as output, a locally optimal point, which is "close" to the starting point and has better objective function value. The search terminates when step c) cannot improve the objective function value. A key element of an LSH is the definition of "closeness", by a *neighbourhood function* [67, §5]

$$I : \mathcal{F} \longrightarrow 2^{|\mathcal{F}|}\,, \tag{2.15}$$

where $\mathcal{F}$ is the feasible set of the problem. The function I in Eq. (2.15) decides the membership of a feasible point $x' \in \mathcal{F}$ in the *neighbourhood* $I(x)$ of a given x. An LSH produces a sequence of feasible points $\{x^0, x^1, \ldots, \}$ such that, at each iteration $t \geq 1$:

$$x^t = \varsigma(x^{t-1}) \quad \text{and} \quad \varsigma(x) = \arg\min_{y \in I(x)} f(y)\,,$$

whereby "min f" is the objective function of the main problem. Optimizing over $I(x)$ is easier than optimizing over the whole $\mathcal{F}$. In general, $I(x)$ is chosen to be much smaller than $\mathcal{F}$; however, if $I(x)$ is large, one can decide to use local algorithms to compute $\varsigma(x)$.

An alternative, commonly used local search strategy is to iteratively solve LP relaxations of the original problem and try to improve/tighten them by fixing variables, adding cuts, branching, etc. Heuristics of this type, customarily employed to tackle MILPs, are: the relaxation induced neighbourhood search [64], the feasibility pump [61] and local branching [63], just to name a few.

2.6.1.4 Metaheuristics

The disadvantage of LSHs is that, optimizing over a (small) subset of $\mathcal{F}$, they might miss the global optima of the problem, even when the size of the neighbourhood defined by in Eq. (2.15) is large, and only produce locally optimal solutions of the problem. To overcome this issue, *metaheuristics* employ LSHs as components of a broader algorithmic framework, which also performs techniques for escaping solutions found during the

search, whenever there is reason to believe that they are not global. These procedures are applied to problems formulated both via MILP and MINLP. In Ch. 4, we will discuss how metaheuristics are exploited in ACP approaches.

LSHs promote *intensification* mechanisms: only the most promising search regions (containing the best solutions found so far) are explored. Metaheuristics, instead, combine intensification and *diversification* mechanisms. Notably, diversification leads the search into unexplored regions, so that the entire set of of solutions can be examined uniformly [182]. Metaheuristics owe their name to the fact that they are generic, high-level strategies. Since metaheuristics seldom employ assumptions about the problem at hand, they can be performed, without specific adaptations, to solve a multitude of different problems and to enhance the performance of diverse LSHs (or even to pick one from a portfolio).

Commonly used metaheuristics for (linear and nonlinear) optimization are: Multi-Start (MS), simulated annealing [132], tabu search [188], genetic algorithms. In particular, MS algorithms execute an LHS at many different starting points, sampled (partly or completely) at random, and choose the best solution out of the ones found. Another widely used metaheuristic for MILP is Variable Neighbourhood Search (VNS)[143], which is, basically, a local search where the neighborhood size is dynamically modified at each iteration.

An exhaustive description of metaheuristics is beyond the scope of this thesis, but the interested reader will find a detailed treatment in, say, [71].

2.7 Mixed-Integer Nonlinear Programming

MINLPs [152, 53] are MPs with nonlinearities in the objective function or in constraints, and whereby some of the variables are forced to take discrete integer values. MINLPs are often described by the following formulation:

$$\begin{array}{rl} \min & f(x) \\ \forall j \in \mathcal{J} & g_j(x) \leq 0 \\ \forall i \in \mathcal{I} & x_i \in \mathbb{Z}\,, \end{array} \tag{2.16}$$

where $f : \mathbb{R}^d \longrightarrow \mathbb{R}$, $g : \mathbb{R}^d \longrightarrow \mathbb{R}^d$, and $\mathcal{I}$ is the set of indices labelling the integer variables.

MINLP is a broad MP class — actually, it includes all other MP classes — with many practical applications, requiring optimization over integer variables and the treatment of nonlinear, possibly nonconvex functions. An MINLP containing no nonlinear functions is an MILP, which, as we mentioned earlier, has been proved to be **NP**-complete. An MINLP without integrality constraints (i.e., containing only continuous variables) is an NLP, which has been shown to be **NP**-hard [103]. Thus, MINLP is hard to solve: in

general, the inclusion of MILP in MINLP is sufficient to make it **NP**-complete, but some MINLP can even be undecidable [98, 116].

The reason why we care about MINLP is that the DGP naturally belongs to this class; moreover, we provide formulations approximating the ACP, which are also MINLP.

The ACP corresponds to the following statement: given a target algorithm, select the algorithmic parameter configuration that will optimize the performances of that algorithm (see Ch. 4). What is the structure of the MP corresponding to this statement? Its objective function, optimizing the behaviour of the target algorithm, is almost always unknown, which is why one often tries to approximate it from known data; in most cases, only nonlinear functions can provide a sufficiently accurate approximation. Its feasible set, containing admissible parameter values and permitted configurations, is often a discrete set. In fact, even assuming that the logical dependence relations between the parameters can be described by linear constraints — which may not always be the case — the feasible set presents nonconvexities.

Instead, given a graph, the DGP seeks to find the position of the graph vertices in a Euclidean space of given dimension. Nonlinearities often arise from enforcing conditions on the distances between adjacent vertices, usually by a norm. See Ch. 7 for more details.

2.7.1 Common solution algorithms

The major algorithmic strategies for tackling MINLP problems, as discussed in [152], are:

2.7.1.1 Relaxations

Relaxations share the same function and similar implementation as in MILP. They are auxiliary formulations, constructed from the MINLP in Eq. (2.16) by processing some of its constraints, with a feasible set usually larger than the original problem, and yielding a (globally valid) lower bound on its optimal objective function value. It is desirable for a relaxation to be easier to solve than the MINLP itself.

The MP in Eq. (2.16) can be relaxed by: disregarding variable integrality (*continuous relaxation*); constructing a piecewise linear underestimator of the convex constraints (*polyhedral relaxation*); building a convex underestimator of the nonconvex constraints (*convex relaxation*); combining the previous techniques. Continuous relaxations are used in B&B type algorithms for nonlinear problems; polyhedral relaxations are employed in outer-approximation type algorithms [52]; convex relaxations of nonconvex NLPs are deployed in a variant of the B&B, called *spatial B&B* (sB&B) [152, §5]. The sB&B computes convex relaxations of the subproblems resulting from branching at continuous variables; in this way, it produces a guaranteed lower bound on the optimal objective function value.

2.7.1.2 Constraint enforcement techniques and metaheuristics

The main constraint enforcement techniques are branching and refinement, which are deployed to tighten relaxations. Instead, metaheuristics can find a feasible solution, yielding an upper bound on the optimal objective function value of Eq. (2.16); they almost never provide global optimality guarantees on the solution found. Both constraint enforcement techniques and heuristics are borrowed from MILP and appropriately adapted to MINLP. We treated them in Sec. 2.6.1.

Commonly used metaheuristics for MINLP are: MS, tabu search and simulated annealing. We remark that the difference between the use of metaheuristics in MILP/MINLP and NLP lies in the structure of the neighbourhood function I in Eq. (2.15). Selecting a neighbourhood when the feasible set is discrete is typically non-trivial, because there is no information about the objective function at points close to the current local optimum. One would have to compute the objective function at all points to know which ones are best suited to be chosen by I. Since this is too computationally demanding, this process often involves an element of random choice. Instead, NLPs have a continuous (or partially continuous) feasible set, so the neighbourhood is simply the topological sphere. Therefore, the gradient of the objective function can be used to move around the current local optimum, and find its decreasing directions. In Sec. 2.8, we will see how information about the objective function can be used to move around the current local optimum, when searching for a solution to unconstrained optimization problems.

Finally, we note that, since MILP $\subset$ MINLP and NLP $\subset$ MINLP, all the main approaches for solving MINLP incorporate algorithms for treating MILP and NLP subproblems. Furthermore, algorithmic approaches distinguish between convex and nonconvex MINLP problems (i.e., cMINLP and nMINLP). A nMINLP has nonconvex objective and/or constraint functions.

A commonly employed strategy for nMINLP is to approximate the nonconvexities in the objective or in the constraints by piecewise linear functions, and then solve the resulting MILP [191]. However, the workhorse for nMINLP is sB&B.

Since a detailed description of the aforementioned algorithmic techniques is not our purpose, we refer to, notably, [152] and [76] for an in-depth discussion.

2.8 Some words about unconstrained optimization

Constrained optimization problems can be reformulated as unconstrained problems, whereby constraints are multiplied by a penalty term and summed to the objective function. The Lagrangian function in Eq. (2.5) is an example of this technique. Moreover, in all ML paradigms, training a predictor involves solving an unconstrained optimization problem, in which a "loss" function, measuring the error of the predictor, is minimized

(see Sec. 3.3).

In unconstrained optimization [150, 82], we deal with continuous variables, whose values are not limited, i.e.:

$$x^* := \arg\min_{x \in \mathcal{D}} f(x)\,, \tag{2.17}$$

where $f : \mathbb{R}^d \longrightarrow \mathbb{R}^s$. Since finding global solutions of Eq. (2.17) is generally problematic, most algorithmic strategies for unconstrained optimization are designed to identify local optima, through an iterative procedure that generates a sequence of iterates $\{x_k\}_{k \in \mathbb{N}}$ from a starting point x_0, such that

$$f(x_0) \geq f(x_1) \geq \cdots \geq f(x_K)\,. \tag{2.18}$$

Alternatively, Eq. (2.18) can be relaxed by requiring that f improves every few iterations instead of every iteration, i.e., $f(x_{k-k'}) \geq f(x_k)$, for some k'. The sequence in Eq. (2.18) terminates if no objective function improvements are produced for a prescribed number of iterations, although other termination criteria are can be used. At each iteration k, x_{k+1} is chosen in function of the information available about f at points $\{x_t\}_{t \leq k}$: its gradient, Hessian (if f is smooth), or an approximation thereof.

2.8.1 Line search and trust-region methods

Iterative methods for unconstrained optimization are categorised into line search and trust region algorithms [150, §2].

2.8.1.1 Line Search algorithm

At each iteration $k \geq 0$ of a line search algorithm, a *search direction* $\gamma_k \in \mathbb{R}^d$ and a *step-size* $\alpha \in \mathbb{R}$ are chosen.

The vector γ_k must be such that, when moving along it from x^k, the objective function value decreases, i.e., the new iterate

$$x^{k+1} = x^k + \alpha\gamma_k$$

must satisfy

$$f(x^{k+1}) \leq f(x^k)\,.$$

The scalar α, which determines how far along γ_k to move, is the solution of

$$\arg\min_{\alpha > 0} f(x^k + \alpha\gamma_k)\,. \tag{2.19}$$

The problem in Eq. (2.19) is usually solved by local algorithms or approximately [150, §2].

One of the most common choices for γ_k is the negative gradient of the objective function, i.e.,

$$\gamma_k = -\nabla f(x_k)\,,$$

which gives rise to the *steepest descent* (or *gradient descent*) method.

Other customary choices for γ_k are the *Newton direction*, requiring the computation of the Hessian of the objective function; several *quasi-Newton directions*, that are based on approximations of the Hessian; *conjugate gradient directions*.

2.8.1.2 Trust-region algorithm

A trust-region algorithm is based on a function m_k, approximating f near x^k, usually by a second-order approximation model. Since the accuracy of m_k decreases by getting farther from x^k, at each iteration, a new iterate

$$x^{k+1} = x^k + \gamma$$

is searched in some neighbourhood of x^k, by approximately solving the problem

$$\arg \min_{\|\gamma\| \leq \tau} m_k(x^k + \gamma)\,, \tag{2.20}$$

where τ defines a trust-region. Clearly there are more complicated strategies for defining the trust-region than simply constraining $\|\gamma\|$. If, given a user-defined threshold $\epsilon > 0$, $f(x^k) - f(x^{k+1}) > \epsilon$, τ is reduced; then, Eq. (2.20) is solved again, with the updated trust-region.

2.9 Solvers

MP solvers are complex machines, capable of handling instances of different structure. To achieve this, most state-of-the-art general-purpose MP solvers implement and combine a wide range of algorithmic components within the B&B algorithmic framework, to address the steps of the solution process: pre-processing procedures, reformulations of various types (notably, relaxations and convexifications), heuristics, cutting planes, etc. The functioning of these components is modifiable through the accompanying algorithmic parameters, that the user can set to appropriate values.

Good solvers have default parameter configurations, which are chosen by the developers so that the solver can run satisfactorily on as many instances as possible. However, default parameter configurations might be suboptimal on specific problem instances; in these cases, finding an ad-hoc configuration becomes a necessity.

Since the list of exposed parameters is often long (hence, the set of possible algorithmic configurations is large), and it is not possible to measure in advance the performance of the solver with different configurations, solving the ACP efficiently for MP solvers is nontrivial. This is especially true for MILP solvers, that nowdays operate an extremely large range of algorithmic strategies (see, e.g., [78]). Moreover, tuning a solver requires in-depth knowledge of the optimization problem at hand, and extensive expertise with

the solver itself. Thus, many research efforts have been spent in the last tens of years to try to automate the solution of the ACP.

In this thesis, we propose a general methodology for finding good configurations for optimization solvers, and we test our approach on MILP solvers.

2.9.1 MILP solvers

In recent decades, there have been enormous strides in our ability to solve MILP, even large instances. This has resulted in the development of efficient and robust solvers, such as IBM ILOG CPLEX (or, simply, CPLEX [91]), Gurobi [78], SCIP [69], just to name a few.

All general-purpose MILP solvers operate some variation of the B&B, and execute the following three major phases:

1. Preprocessing: the problem at hand is simplified and its size shrunk, when possible, by eliminating variables and constraints, tightening the bounds on the variables or fixing some of them. This phase is usually fast and does not require many computational resources.

2. Root node: initial lower and upper bounds on the objective function value, together with a preliminary feasible primal solution, are produced by solving a first continuous relaxation of the problem and performing some LSH/metaheuristic (typically exploiting information about the relaxation). Next, the solver tries to improve the computed estimates, by a) generating cuts and solving the derived LPs, and b) performing other heuristics. The root-node phase usually requires a considerable amount of time and computational resources, and it is the most important stage in the solution process.

3. Tree exploration: the solver explores the decision tree, by branching from the root node, and tries to refine the bounds yielded by the previous phase. This generally involves solving further continuous relaxations, cuts and heuristics until optimality is attained within a certain tolerance. However, since most of the computational effort is only really done at the root node, cuts and "costly" heuristics are seldom invoked at this stage. We remark that this phase can be easily parallelized.

Many MILP solvers incorporate algorithms for addressing some well-known and common NLP (usually, CP) problems, such as Conic Programming [29, §4] problems (in, e.g., CPLEX), or QP problem (in, e.g., Gurobi).

2.10 Conclusions

In this chapter, we gave an overview of MP, a formal language for describing optimization problems and a framework for solving them.

MP is important for this thesis mainly because it allows us to formally describe the main research problems addressed by this thesis, i.e., the ACP and the DGP, and it supplies algorithms to solve them. Further, one of the main contributions of this Ph.D. thesis is to devise new methodologies to deal with the ACP, and our interest in MP motivated us to test them methodologies to tune MP solvers. Finally, our ACP approaches are based on ML paradigms, and MP plays a key role in their implementation, because constructing ML-based approximations requires solving optimization problems.

In the chapter, we first described the main components of an MP formulation. Then, we introduced its main classes: LP, MILP and MINLP. For each MP class, we also discussed some of the most commonly employed solution algorithms: in particular, we focused on those algorithms that are not only important to the MP community, but also well known to ML practitioners and/or recurrent in the ACP literature.

Chapter 3

On machine learning

3.1 Introduction

ML is a branch of computer science, whose purpose is to conceive, design, and implement algorithms which are capable of constructing *hypotheses* from data, i.e., functions which can approximate some property underlying the data. The ML problem aims at finding the hypothesis providing the best such approximation. Since the data is an infinite collection of instances, hypotheses are constructed from a known subset called "training set". The characteristics of the training set determine which of the two main ML frameworks should be used:

- Supervised Learning: an instance is a pair (input, output). The training points represent the "ground truth". The training set is used to build a function capable of mapping an input to the (approximately) correct output, i.e., of approximating the ground truth. Supervised learning may be applied to classification or regression tasks (see Sec. 3.6, i.e., to predict discrete or continuous outputs;

- Unsupervised Learning: an instance is an unlabelled input, that is, no output is given and the patterns underlying the problem instances need to be identified. The training set is used to construct a representation of the input structure. This encompasses tasks such as clustering, anomaly detection and dimensionality reduction (see Sec. 3.5.4), just to name a few [81, Ch. 14].

Sometimes, one wants to learn how to solve sequential decision problems. In these cases, the reference framework is reinforcement learning. In reinforcement learninng, an agent interacts with a potentially unknown environment, by performing actions, watching the environment react accordingly, and receiving feedback. This feedback promotes actions which contribute to the achievement of some prescribed goals. The purpose of this framework is to identify a policy optimizing such feedback. This setting is typically modeled as a Markov decision process.

In this chapter, we will present several ML paradigms within the supervised learning framework, and we will see how they are used for approximating arbitrary functions. We will also discuss several techniques for improving the approximation provided by a class of hypotheses. This is instrumental for the analysis of the experiments presented in the second part of this thesis, dedicated to the ACP. In particular, in Ch. 4 we will discuss several uses of ML techniques in the ACP literature. Then, in Ch. 5 and 6, we will illustrate our two MP formulations for the ACP, where the approximation functions learned by the ML paradigms presented here appear as arithmetic expressions in closed form.

Finally, in Ch. 7 we will present the DGP. This optimization problem has many practical applications (see Sec. 7.2), and one of them is relevant for ML. In fact, solving the DGP is equivalent to computing the embedding of a graph in a Euclidean space; this is useful, in ML, when training data arrives in some structured form, such as graphs, but the chosen learning paradigm can only process "flat" input data (typically, vectors and matrices). In this case, the most direct application would be to compute the DGP embedding for a set of training input graphs, and then give them in flat form to the chosen ML paradigm. Moreover, computing the DGP embedding can be instrumental in the execution of dimensionality reduction strategies, treated in Sec. 3.5.4, which are often necessary to improve the approximation accuracy of ML hypotheses.

The following section will introduce one of the best known frameworks for supervised learning, namely, the statistical learning theory. The focus will be on the regression task.

3.2 An introduction to statistical learning theory

The statistical learning theory [189] is a framework for supervised learning, providing the theoretical foundations for building prediction functions with good generalization capabilities. *Generalization* (discussed in Sec. 3.4) is the ability of a predictor to learn, from the given training data, an output approximation function with high prediction accuracy, as measured at points outside the training set. The components of supervised learning theory are:

- an *input set* $\mathcal{X} \subseteq \mathbb{R}^d$, containing points drawn independently from a fixed but unknown probability distribution function $F(x)$;
- an *output set* $\mathcal{Y} \subseteq \mathbb{R}^t$, containing the images $y = f(x)$ by an unknown function f, for all $x \in \mathcal{X}$. The points of $\mathcal{Y}$ are drawn from an unknown conditional distribution $F(y|x)$;
- a *training set*

$$\mathcal{S} = \big\{(x_i, y_i) \mid i \in \{1, \dots, s\}\big\} \subseteq \mathcal{X} \times \mathcal{Y} \tag{3.1}$$

of independent and identically distributed (i.i.d.) observations. The set in Eq. (3.1) is sampled by the unknown joint probability density function

$$F(x, y) = F(x)F(y|x) \tag{3.2}$$

characterising the sample space of the phenomenon to be learned. In the following, we let $S = \{1, \ldots, s\}$ and $|S| = s$ and assume that the probability distribution of the training set $\mathcal{S}$ is representative of the true distribution in Eq. (3.2). In fact, a careful sampling of $\mathcal{S}$ is a critical requirement for the reliability/robustness of an ML predictor;

- a *learning algorithm*, which selects a function

$$h_\theta : \mathcal{X} \longrightarrow \mathcal{Y}\,, \tag{3.3}$$

parametrized by a coefficient vector $\theta \in \Theta$, from a *hypothesis set* $\mathcal{H}$. Each hypothesis $h_\theta \in \mathcal{H}$ maps an input $x \in \mathcal{X}$ to an approximation of $y = f(x)$. $\mathcal{H}$ is partitioned into *hypothesis classes*; each class contains functions with the same structure and different coefficients θ.

3.3 The training and inference problems

The hypothesis coefficients θ^* allowing the best generalization are determined by solving the *training problem*, an optimization problem. In Sec. 3.6 we will see how the training problem is formulated for different ML approaches.

The training problem minimizes a loss function for a given hypothesis class; its constraints describe the structure of the hypotheses in that class. The *loss* function

$$\ell : \mathcal{Y} \times \mathcal{Y} \longrightarrow \mathbb{R} \tag{3.4}$$

maps a couple $\big(f(x), h_\theta(x)\big)$ to the positive error committed by h_θ in approximating f at a point x. In general, it is a dissimilarity function that approaches zero as its two arguments get closer.

An important choice for solving the training problem is that of a hypothesis *hyperparameters*. Each θ of h_θ is a vector of two components:

$$\theta = (\theta_{\mathsf{hyper}}, \theta_{\mathsf{par}})\,.$$

The subvector θ_{hyper} contains the hyperparameters of h_θ, which control the structure of the training problem and the solution algorithm used to solve it. The subvector θ_{par} contains the proper parameters of h_θ, which are obtained by solving the training problem, for given θ_{hyper} and $\mathcal{S}$. Therefore, while θ_{hyper} is provided as an input to the training problem, θ_{par} is its output.

Hyperparameters are typically selected by solving the training problem several times, each time for different θ_{hyper}, and evaluating the prediction error incurred on a dedicated subset of the training set, called *validation set*. Then, the validation set will be ignored during training. In the rest of this chapter, we will use the abbreviation $\mathcal{S}_{\mathsf{val}}$ to denote the validation set. This procedure, called "Cross-Validation" (CV) and described in detail in Sec. 3.5.3, selects the $\theta^*_{\mathsf{hyper}}$ with the best prediction performance on $\mathcal{S}_{\mathsf{val}}$. Afterwards, $\theta^*_{\mathsf{hyper}}$ is input to the training problem, which is solved using the whole $\mathcal{S}$ (including the $\mathcal{S}_{\mathsf{val}}$) to select θ_{par^*}. One can assess the generalization capabilities of the fully trained predictor $h_{(\theta^*_{\mathsf{hyper}},\theta^*_{\mathsf{par}})}$ by computing its prediction error on a *test set* $\mathcal{T}$; $\mathcal{T}$ is such that $\mathcal{T} \cup \mathcal{S} = \emptyset$.

Given the optimal solution θ^* of the training problem, one can represent the learned approximation h_{θ^*} of f through a a set of statements, written as arithmetical expressions in equalities/inequalities, or as memberships in certain sets of numbers (integer, real, etc..), which we call the *inference problem*. While in the training problem we optimize over Θ, in the inference problem the parameters are fixed, i.e., $\theta = \theta^*$. The inference problem always contains at least one equation, describing the actual prediction function $h_{\theta^*}(x)$. The other equations encode additional constraints necessary to represent h_{θ^*}. Given an input $x \in \mathcal{X}$, $h_{\theta^*}(x)$ is computed by statically evaluating all the equations of the inference problem. Later in this chapter, we describe the inference problem of several ML paradigms. The issue of how to best encode the inference problem is relevant for this thesis, because the approaches we implemented to address the ACP employ ML paradigms. Notably, we formulate the inference problem of the trained predictors by closed-form equations, thus we can treat them as "white-box". This allows us to embed ML predictions into our MP formulations of the ACP, and directly solve these formulations with MP algorithms. Clearly, we are interested in finding the encoding of the inference problem resulting in the ACP formulation that is easier to solve; we will discuss this issue in Ch. 5. We remark that this strategy is unprecedented in the algorithm configuration literature, although it has been investigated in the optimization community in the last years (see, e.g., [106] and the references cited therein).

3.4 Risk and generalization

The loss function is instrumental in the definition of the *expected risk* (or *generalization error*) $R(h_\theta)$ of a learning hypothesis $h_\theta \in \mathcal{H}$:

$$\begin{aligned} R(h_\theta) &= \mathbb{E}_{(x,y)\sim F(x,y)}[\ell(y, h_\theta(x))] \\ &= \int \ell(y, h_\theta(x)) dF(x,y)\,. \end{aligned} \tag{3.5}$$

R in Eq. (3.5) is the loss expectation of h_θ — that is, the probability of h_θ incurring a loss — over couples (x, y) drawn from $F(x, y)$.

With the definitions above, the supervised learning problem is formulated as follows:

$$h^* = \arg\min_{h_\theta \in \mathcal{H}} R(h_\theta)\,. \tag{3.6}$$

However, $F(x, y)$ is almost always unknown, therefore computing the expected risk is impossible and Eq. (3.6) cannot be solved. Then, since the training set $\mathcal{S}$ encodes the only available information about F, the expected risk is replaced in practice by the surrogate *empirical risk*

$$\hat{R}_{\mathcal{S}}(h_\theta) = \frac{1}{s}\sum_{i \in S} \ell(y_i, h_\theta(x_i))\,, \tag{3.7}$$

measuring the average loss at the points in $\mathcal{S}$. In other words, one can solve the *Empirical Risk Minimization* (ERM) problem

$$h^*_{\mathcal{S}} = \arg\min_{h_\theta \in \mathcal{H}} \hat{R}_{\mathcal{S}}(h_\theta)\,, \tag{3.8}$$

instead of solving Eq. (3.6).

The issue of Eq. (3.8) is that, if $\mathcal{H}$ contains a hypothesis which is capable of memorizing all (or most of) the training points, that hypothesis will be chosen as $h^*_{\mathcal{S}}$, as its empirical error is (close to) zero. However, such $h^*_{\mathcal{S}}$ would be unable to generalize: in fact, training set data always contains some noise, and $h^*_{\mathcal{S}}$ would perfectly fit the noise in $\mathcal{S}$, and it would commit large prediction errors at points outside it. This phenomenon is known as *overfitting*.

Another phenomenon associated with poor generalization, although not as common as overfitting, is *underfitting*. Underfitting occurs when $h^*_{\mathcal{S}}$ is too simple to adequately process the complexity of the training data, and it therefore fails to correctly capture the underlying patterns. For example, adopting a linear function to model complicated nonlinear relations may result in underfitting.

The usual behaviour of the training and validation/test errors is represented in Fig. 3.1: in practice we observe that, while the training error decreases in function of the time/iterations spent solving the training problem, the validation/test error (computed on held-out data, not used for training purposes) reduces at first, and then goes up as the hypothesis begins to overfit/underfit the data.

3.5 The path to improving generalization

The techniques performed to address overfitting/underfitting concern the treatment of $\mathcal{H}$ and that of $\mathcal{S}$.

There are basically two ways of acting on $\mathcal{H}$ (see Sec. 3.5.1 and 3.5.2): both are designed to reduce its complexity, by restricting the set of potential solutions of the ERM problem in Eq. (3.8) to a subset of $\mathcal{H}$. The first strategy is to manually select a class $\mathcal{H}' \subset \mathcal{H}$; the second is to penalize overly complex hypotheses, by modifying the

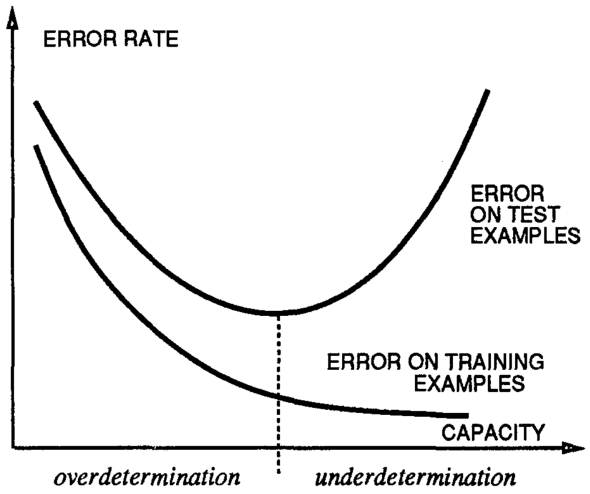

Figure 3.1: Training and generalization error of a predictor, as a function of the predictor's complexity [90]: overfitting ("underdetermination") and underfitting ("overdetermination")

objective function of the ERM problem with an additional penalty term. On the concept of "complexity", we briefly remark that it is related to how complicated the functions approximated by a hypothesis can get. There have been several attempts to provide a formal definition. For example, the Vapnik-Chervonenkis (VC) theory [187], only valid for binary classification tasks, defines complexity as the largest set of points that can be perfectly classified by a given hypothesis, for all possible output labeling assignments. An alternative definition is that of Rademacher complexity [144, 13, §3], related to the ability of a class of $\mathcal{H}$ to fit random noise in the data. However, an extensive treatment of complexity theories is beyond the scope of this thesis; the subject is treated in greater detail in texts such as, e.g., [28].

Instead, techniques targeting the training set $\mathcal{S}$ (see Sec. 3.5.3 and 3.5.4), seek to increase its size (by re-sampling, generation of additional data, data augmentation, etc.), or involve some processing of $\mathcal{S}$ to eliminate redundancies, or rely on the use of a validation set (already discussed in Sec. 3.3). Strategies of the first type are implemented in cases where $\mathcal{S}$ is not representative of the phenomenon to be learned — for example, when the training set is too small — which would lead to poor generalization.

3.5.1 Structural risk minimization

In practice, the ERM problem in Eq. (3.8) is generally solved for a given class $\mathcal{H}' \subset \mathcal{H}$; in this section, we call its solution $h^*_{\mathcal{S},\mathcal{H}'}$.

However, selecting $\mathcal{H}'$ is typically challenging in practice, as it involves a trade-off.

To show this, consider the *excess error* of a given $h_\theta \in \mathcal{H}'$, i.e., the difference between

the expected risk of h_θ and that of h^*:

$$R(h_\theta) - R(h^*) = \underbrace{R(h_\theta) - R(h^*_{\mathcal{S},\mathcal{H}'})}_{\text{estimation error}} + \underbrace{R(h^*_{\mathcal{S},\mathcal{H}'}) - R(h^*)}_{\text{approximation error}} . \tag{3.9}$$

Eq. (3.9) consists of two terms, that we call *estimation error* and *approximation error*. The *estimation error* measures the cost of choosing h_θ over the best-in-class. For a large enough training set, the estimation error can be sometimes approximated by substituting R with $\hat{R}$. In fact, the VC and Rademacher complexity theories show that, for a given $\mathcal{H}'$, the empirical risk converges to the expected risk as the training set size grows [144, §4.1-4.2]. The *approximation error* assesses the generalization of the chosen class — represented by its "best performing" member $h^*_{\mathcal{S},\mathcal{H}'}$ — compared to h^*. However, the expected error can never be computed, so h^* and the approximation error are unknown.

The trade-off of picking $\mathcal{H}'$, as represented in Eq. (3.9), is as follows: a larger $\mathcal{H}'$ is more likely to contain complex functions, with expected error close to that of h^* (defined in Eq. (3.6)); however, if $\mathcal{H}'$ is too large, the issue of finding $h^*_{\mathcal{S},\mathcal{H}'}$ might be computationally intractable. Thus, a larger $\mathcal{H}'$ is likely to reduce the approximation error, but also to increase the estimation error.

Structural Risk Minimization (SRM) is a procedure for selecting $\mathcal{H}'$. It uses: a) a function

$$\omega : \mathcal{H} \longrightarrow \mathbb{R}^+ , \tag{3.10}$$

mapping a hypothesis h_θ to a positive real number measuring its complexity (often chosen to be a norm of θ), and b) coefficients

$$c_1 \leq c_2 \leq \cdots \leq c_r, \quad c_1, \ldots, c_r \in \mathbb{R} ,$$

to define a hierarchical structure of nested subsets

$$\mathcal{H}_1 \subset \mathcal{H}_2 \subset \cdots \subset \mathcal{H}_r \subset \mathcal{H}$$

such that

$$\forall 1 \leq i \leq r \qquad \mathcal{H}_i = \{h_\theta \in \mathcal{H} \mid \omega(h_\theta) \leq c_i\} .$$

The purpose of SRM is to find, for $1 \leq r' \leq r$, the class $\mathcal{H}_{r'}$ providing the tightest upper bound on the excess error. Then, the SRM procedure finds $h^*_{\mathcal{S},\mathcal{H}_{r'}}$ over the chosen $\mathcal{H}_{r'}$.

3.5.2 Regularized risk minimization

The major downside of SRM is that the cost of calculating the bounds on the excess error can be too high [144, §4.3]. For this reason, it is sometimes preferable [28] to solve a "regularized" variant of the SRM problem:

$$\arg \min_{\substack{h_\theta \in \mathcal{H}_i \\ 1 \leq i \leq r}} \hat{R}(h_\theta) + \lambda \omega(h_\theta) . \tag{3.11}$$

The coefficient $\lambda > 0$, encoding the trade-off between complexity and empirical risk, is a hyperparameter; it is usually chosen by a procedure called "cross-validation" (illustrated in the next section), although other methods may be used [30].

The *Regularized Risk Minimization* (RRM) approach implements a special case of the problem in Eq. 3.11. The RRM is obtained by solving the regularized SRM problem in Eq. (3.11) with respect to only one hypothesis class:

$$\arg \min_{h_\theta \in \mathcal{H}'} \hat{R}_{\mathcal{S}}(h_\theta) + \lambda \omega(h_\theta)\,. \tag{3.12}$$

The RRM is famously used, for example, in the SVR training problem (see Sec. 3.6.4).

There are other approaches to regularization, such as, say, *early stopping* [22, §5], which is often used to train Neural Networks (NNs), for which we refer to Sec. 3.6.3.

3.5.3 Cross-validation

CV is a family of techniques for assessing the predictions of a hypothesis h_θ for given hyperparameters. Basically, it uses a validation set (introduced in Sec. 3.3) to evaluate the generalization of h_θ, when the hypothesis is applied to predict new data. Given $0 < \alpha < 1$, the validation set is obtained by partitioning $\mathcal{S}$ as

$$\{\mathcal{S}_{\mathsf{ptr}} \cup \mathcal{S}_{\mathsf{val}}\} \quad \text{such that} \quad \mathcal{S} = \mathcal{S}_{\mathsf{ptr}} \cup \mathcal{S}_{\mathsf{val}}\,, \tag{3.13}$$

such that

$$|\mathcal{S}_{\mathsf{ptr}}| = \alpha s \quad \text{and} \quad |\mathcal{S}_{\mathsf{val}}| = (1-\alpha)s\,. \tag{3.14}$$

We recall that $s = |\mathcal{S}|$; moreover, we remark that the shorthand "ptr" in $\mathcal{S}_{\mathsf{ptr}}$ stands for "proper training set", to distinguish it from the training set $\mathcal{S}$. CV is thus applied to select the best hyperparameters of a given hypothesis, using $\mathcal{S}_{\mathsf{ptr}}$ for training with several candidate hyperparameters, and $\mathcal{S}_{\mathsf{val}}$ to assess the empirical risk of the candidates. This is accomplished by solving the following problem, for a given class $\mathcal{H}' \subset \mathcal{H}$ [144, §4]:

$$h^*_{\mathcal{S}_{\mathsf{val}}} = \arg \inf_{h_{\bar{\theta}} \in \mathcal{H}'} \hat{R}_{\mathcal{S}_{\mathsf{val}}}(h_{\bar{\theta}})\,, \tag{3.15}$$

where: $\bar{\theta} = (\theta_{\mathsf{hyper}}, \bar{\theta}_{\mathsf{par}})$, such that θ_{hyper} is a given hyperparameter vector, and $\bar{\theta}_{\mathsf{par}}$ is the solution of the ERM training problem with $\mathcal{S}_{\mathsf{ptr}}$ and θ_{hyper} as inputs; $\hat{R}_{\mathcal{S}_{\mathsf{val}}}$ is the empirical risk on $\mathcal{S}_{\mathsf{val}}$. We call $\theta^*_{\mathsf{hyper}}$ the hyperparameter vector corresponding to $h^*_{\mathcal{S}_{\mathsf{val}}}$.

Usually, the described CV procedure is repeated on several partitions $\{\mathcal{S}_{\mathsf{ptr}}, \mathcal{S}_{\mathsf{val}}\}$; the results on the different partitions are then aggregated to select $\theta^*_{\mathsf{hyper}}$. There are many strategies for creating partitions of $\mathcal{S}$ and applying CV iteratively. In *K-fold CV* [81, §7.10], for $K \in \mathbb{N}^+$, a partition with K subsets is computed. Then, at each iteration, one subset is used for validation purposes and the remaining $K-1$ ones are used for training purposes. An alternative to K-fold CV is, for example, to apply

repeated random sub-sampling, which at each iteration creates a random partition of the training set.

This set-up is sometimes used in a larger evaluation schema, called *Nested Cross-Validation* [38, 190] (NCV). In NCV, $\mathcal{S}$ is first partitioned K times as $\{\mathcal{S}_{\mathsf{hs}}, \mathcal{S}_{\mathsf{test}}\}$, i.e., "hyperparameter selection set" and "test set". Then, the CV problem in Eq. (3.15) (or one of its variants with multiple partitions) is solved on each $\mathcal{S}_{\mathsf{hs}}$, and the error of the resulting solution is calculated on the corresponding test set $\mathcal{S}_{\mathsf{test}}$. Finally, the average prediction error is computed over the K test sets. The purpose of this procedure is to assess the generalization capabilities of the chosen hypothesis class $\mathcal{H}'$.

3.5.4 The feature space

The methodologies presented above select a class of hypotheses $\mathcal{H}' \subset \mathcal{H}$ for a fixed training set. Given $\mathcal{H}'$, instead, it is possible to act on $\mathcal{S}$ in order to construct more accurate predictors. Methodologies based on this strategy first process the input set $\mathcal{X} \subseteq \mathbb{R}^d$ by a *feature function*

$$\Phi : \mathcal{X} \longrightarrow \mathcal{F}\,, \tag{3.16}$$

which maps it into a different *feature set* $\mathcal{F}$, and then train a predictor with the training set $(\mathcal{F}, \mathcal{Y})$, rather than $(\mathcal{X}, \mathcal{Y})$.

In the following, we discuss some common approaches to construct Φ.

3.5.4.1 Dimensionality reduction techniques

Dimensionality reduction techniques are a family of methodologies for shrinking the vectors in the input set $\mathcal{X}$.

Feature selection (FS) is a subset of dimensionality reduction techniques for selecting the components of input set vectors, so as to identify a minimal set of input vector components conveying the essential information contained in the data. The elimination of redundancies and noise in the data usually improves training times and decreases prediction errors.

Formally, we let $\mathcal{I}$ be a set with the following characteristics:

$$\mathcal{I} \subset \{1, 2, \dots d\} \quad \text{s.t. } |\mathcal{I}| = d' \text{ and } 0 < d' < d\,.$$

Further, we let

$$x[\mathcal{I}] \in \mathbb{R}^{d'} \tag{3.17}$$

be the vector obtained by selecting the components of $x \in \mathcal{X}$ according to the indices in $\mathcal{I}$. The purpose of an FS methodology is to choose $\mathcal{I}$ for a given training set $\mathcal{S}$. Thus, the accompanying feature function will be

$$\Phi_{\mathcal{I}}(x) = x[\mathcal{I}] \in \mathbb{R}^{d'}\,.$$

For an overview of the different types of FS, see [79] and references therein.

FS is just one of the possibilities for performing dimensionality reduction. Other commonly used methods are, for example: principal component analysis [174] and random projection [21], i.e., mathematical methods operasting linear transformations of the input set vectors; autoencoders [73], performing a new encoding of the input set vectors by means of an NN.

For the ML-based computational experiments described in this thesis — notably, in Ch. 5 and 6 — we tested both principal component analysis and FS. However, since the former produced no significant improvements on our ML predictors, we will only present results with FS.

3.5.4.2 Feature manipulation and extraction

Feature manipulation and *feature extraction* are families of methodologies for constructing the function Φ in Eq. (3.16).

The first performs transformations of the individual components of the input set vectors, by computing, for each component, statistics over all points of the training set. Some examples are: scaling into a prescribed range; standardization, so that the component has zero mean and unit variance; logarithmic transformation.

The second applies transformations to single input components, or transformations involving several input components, to produce new prediction variables, to be added to each input vector. Usually, feature extraction techniques are deployed before dimensionality reduction is performed.

3.6 Some regression paradigms

This section covers the supervised ML paradigms used for regression purposes in the experiments reported in Ch. 5 and 6. For each one, we will formulate the associated training problem and the inference problem.

3.6.1 Logistic Regression

Logistic Regression (LR) is a supervised ML paradigm, devised for binary classification tasks [42]. Once trained, the LR maps input vectors $x \in \mathcal{X}$ to an output scalar in $[0, 1]$. In this sense, LR approximates a binary scalar; binary values are retrieved by rounding, if necessary.

Let $\mathcal{X} = (\mathcal{X}_1, \ldots, \mathcal{X}_s)$ be a vector of random variables, and let $\mathcal{Y}$ be a Bernoulli distributed random variable depending on $\mathcal{X}$. Following [100], and denoting $\mathsf{P}(\mathcal{X} = x)$ by $\mathsf{P}(x)$ and $\mathsf{P}(\mathcal{Y} = y)$ by $\mathsf{P}(y)$, the probability $\mathsf{P}(1|x)$ that $\mathcal{Y} = 1$ can be written as the composition of a logistic sigmoid function and a linear function of the input x. Since

$\mathsf{P}(0|x) = 1 - \mathsf{P}(1|x)$, we have:

$$\begin{aligned} \mathsf{P}(1|x) &= \frac{\mathsf{P}(x|1)\mathsf{P}(1)}{\mathsf{P}(x)} = \frac{\mathsf{P}(x|1)\mathsf{P}(1)}{\mathsf{P}(x|1)\mathsf{P}(1) + \mathsf{P}(x|0)\mathsf{P}(0)} \\ &= \frac{1}{1 + \frac{\mathsf{P}(x|0)\mathsf{P}(0)}{\mathsf{P}(x|1)\mathsf{P}(1)}} = \frac{1}{1 + e^{-z}} = \sigma(z)\,, \end{aligned} \tag{3.18}$$

where

$$z = \ln \frac{\mathsf{P}(x|1)}{\mathsf{P}(x|0)} + \ln \frac{\mathsf{P}(1)}{\mathsf{P}(0)}\,.$$

Furthermore, we assume that z depends linearly on x, i.e.,

$$\exists w \in \mathbb{R}^d, b \in \mathbb{R} \quad z = \langle w, x \rangle + b\,. \tag{3.19}$$

The purpose of LR is to learn, for an input $x \in \mathcal{X}$, the approximation of $\mathsf{P}(1|x)$ according to Eq. (3.18). We note that w, b are the parameters of the LR paradigm, and their optimal values w^*, b^* are found by solving the LR training problem.

3.6.1.1 The training problem

In some cases w^*, b^* can be computed explicitly. For example, if we assume that the conditional probabilities $\mathsf{P}(x|y)$ are multivariate Gaussians with means μ_y and identical covariance matrices Σ, and use the above expression for $\mathsf{P}(1|x)$, we obtain

$$\mathsf{P}(1|x) = \frac{1}{1 + e^{-\langle w^*, x \rangle - b^*}}\,,$$

where $w^* = \Sigma^{-1}(\mu_1 - \mu_0)$ and $b^* = \frac{1}{2}(\mu_0 + \mu_1)^\top \Sigma^{-1}(\mu_0 - \mu_1) + \ln(\mathsf{P}(1)/\mathsf{P}(0))$. In general, however, explicit formulæ cannot always be given, and w^*, b^* must be calculated from sampled data. In these cases, the LR training problem is typically formulated by the maximum likelihood estimation method [73, §5], i.e., the maximization of the *likelihood function*

$$L(w, b) = \prod_{i \in S} h(x_i, w, b)^{y_i} (1 - h(x_i, w, b))^{1 - y_i}\,. \tag{3.20}$$

The maximization of the likelihood function in Eq. (3.20) is performed considering w, b as variables, rather than as parameters. Since the logarithmic function is monotone in its argument, maximizing $\ln(L(w, b))$ yields the same optima as maximizing $L(w, b)$. Thus, maximum likelihood estimation is customarily formulated as:

$$\max_{w,b} \sum_{i \in S} \left[y_i \ln \left(\frac{1}{1 + e^{-(\langle w, x_i \rangle + b)}} \right) + (1 - y_i) \ln \left(1 - \frac{1}{1 + e^{-(\langle w, x_i \rangle + b)}} \right) \right]\,. \tag{3.21}$$

We recall that the functions

$$\ln \left(\frac{1}{1 + e^{-z}} \right) \quad \text{and} \quad \ln \left(1 - \frac{1}{1 + e^{-z}} \right)$$

are concave [29, Ex. 3.49(a)]. Since $0 \leq y_i \leq 1$ for each $i \in S$, Eq. (3.21) maximizes the sum of convex combinations of concave functions. Thus, it is a CP, which can be solved by local unconstrained optimization methods, such as gradient descent (see Sec. 2.8.1 or *stochastic gradient descent* (see Sec. 3.6.3.4, able to deal even with large-scale instances.

Sometimes, following the strategies outlined in Sec. 3.5.2, a "regularized" variant of Eq. (3.21) is implemented, whereby a term penalizing w, b (usually, in the form of an absolute-value norm, a euclidean norm or a convex combination of the two) is added to the problem.

3.6.1.2 The inference problem

Since we only consider two class labels $\{0, 1\}$, the LR models the probability that $\mathcal{Y} = y$, conditional to $\mathcal{X} = x$, by a Bernoulli distribution $\mathsf{P}(y \,|\, x)$. Given $\bar{x} \in \mathcal{X}$, its goal is to learn an approximation

$$h_{w^*,b^*}(\bar{x}) = \frac{1}{1 + e^{-\langle w^*, \bar{x} \rangle - b^*}} \,. \tag{3.22}$$

of $\mathsf{P}(1|\bar{x})$, such that, for $y \in \{0, 1\}$,

$$h_{w^*,b^*}(\bar{x})^y (1 - h_{w^*,b^*}(\bar{x}))^{1-y} \tag{3.23}$$

is as close as possible to $\mathsf{P}(y \,|\, \bar{x})$.

Eq. (3.22) is the LR inference problem.

3.6.2 Decision Trees

A Decision Tree (DT) [31] is a supervised ML paradigm, capable of predicting any type of output, thereby suited to both classification and regression tasks. A DT is represented by a full binary tree, i.e., a nonempty, directed graph $T = (V, A)$ such that:

- V, the vertex set, is partitioned as $\{o, V_I, V_L\}$, i.e., {*root* vertex, *intermediate* vertices, *leaf* vertices}. Vertex o has zero incoming arcs and two outgoing arcs; vertices in V_I have one incoming arc and two outgoing arcs; vertices in V_O have one incoming arc and no outgoing arcs. For each vertex in $\{o\} \cup V_I$, we call left and right *children* the two nodes adjacent to, respectively, the left and right outgoing arcs. In the following, the left and right children will be identified by the abbreviations lx and rx. We assume that the DTs we handle are processed in a depth-first fashion, from left to right;

- for each vertex $v \in \{o\} \cup V_I$, a *threshold* function

  $$\tau_v : \{o\} \cup V_I \longrightarrow \mathbb{R}$$

  and an *index* function

  $$\rho_v : \{o\} \cup V_I \longrightarrow \{1, 2, \ldots, d\}$$

are defined;

- a *vertex labeling* function

$$\lambda : V_I \cup V_L \longrightarrow \mathbb{R}$$

is defined at each vertex $v \in V_I \cup V_L$.

3.6.2.1 The training problem

The training problem is to find a training set partition

$$\mathcal{S} = \{\mathcal{S}^v\}_{v \in V_L} , \tag{3.24}$$

such that training points i with similar outputs y_i belong to the same subset, and are grouped at the same leaf vertex. For each $v \in V_L$, we call S^v the set of indices associated to the training points in $\mathcal{S}^v$. Solving the training problem yields the optimal values $\tau^*, \rho^*, \lambda^*$ of the threshold, index and vertex labeling functions.

In practice, the partition in Eq. (3.24) is obtained by implementing a branching algorithm (introduced in Sec. 2.6.1.1). The algorithm begins at the root vertex, where $\mathcal{S}$ is partitioned into two subsets by imposing the following constraints on the input set vectors $x \in \mathcal{X}$:

$$\forall v \in \{o\} \cup V_I \quad \begin{cases} x[\rho_v] \leq \tau_v & \text{at the left child } v_{\mathsf{lx}} \text{ of } v \\ x[\rho_v] > \tau_v & \text{at the right child } v_{\mathsf{rx}} \text{ of } v\,. \end{cases} \tag{3.25}$$

We note that $x[\rho_v]$ is the ρ_v-th component of x. Eq. (3.25) give the partition $\mathcal{S}^v = \mathcal{S}^{v_{\mathsf{lx}}}, \mathcal{S}^{v_{\mathsf{rx}}}$, where

$$\mathcal{S}^{v_{\mathsf{lx}}} = \{(x_i, y_i) \mid x_i[\rho_v] \leq \tau_v\} , \qquad \mathcal{S}^{v_{\mathsf{rx}}} = \{(x_i, y_i) \mid x_i[\rho_v] > \tau_v\} .$$

The algorithm assigns each subset of the partition thus produced to one of the children, and recursively processes each child in the same way as the root.

When, at a vertex, the procedure stops because some termination criterion is satisfied, that vertex is labelled as "leaf"; the subset at that leaf, which can no longer be split, becomes one of the components of the final partition in Eq. (3.24). Commonly adopted termination criteria are conditions on the maximum tree depth or on the minimum size of the $|S^v|$.

Several candidate splits are evaluated at branching decision, by considering different input components and values, i.e., different couples (τ_v, ρ_v). The optimal one is found by solving (usually, approximately) the following problem:

$$(\rho_v^*, \tau_v^*) = \arg\min_{\rho_v, \tau_v} \frac{|S^{v_{\mathsf{lx}}}|}{s} \sum_{i \in \mathcal{S}^{v_{\mathsf{lx}}}} \ell(y_i, \lambda_{v_{\mathsf{lx}}}) + \frac{|S^{v_{\mathsf{rx}}}|}{s} \sum_{i \in \mathcal{S}^{v_{\mathsf{rx}}}} \ell(y_i, \lambda_{v_{\mathsf{rx}}}) \tag{3.26}$$

where, for all $v \in V_I \cup V_L$,

$$\lambda_v = \operatorname*{\mathsf{avg}}_{i \in S^v} y_i \tag{3.27}$$

and ℓ is, often, the loss function defined by Eq. (3.4).

Eq. (3.26) also yields, for all vertices in $V_I \cup V_L$, the optimal labels $\lambda^*_{v_{\mathsf{lx}}}, \lambda^*_{v_{\mathsf{rx}}}$.

3.6.2.2 The inference problem

Given a input $\bar{x} \in \mathcal{X}$, the learned tree $T^* = (\{o^*\} \cup V_I^* \cup V_L^*, A^*)$ and the optimal training parameters $(\rho^*, \tau^*, \lambda^*)$, the inference problem is the computation of a path from o^* to one of the leaves. This is accomplished by checking, at each vertex in $\{o^*\} \cup V_I^*$, whether $\bar{x}$ satisfies the branching constraints by Eq. (3.25). The leaf label at the end of the computed path provides an approximation of $f(\bar{x})$.

To formally describe the DT inference problem, we define the binary coefficients

$$w_{uv} = \begin{cases} 1 & \text{if } v = u_{\mathsf{lx}} \\ 0 & \text{if } v = u_{\mathsf{rx}}\,. \end{cases} \tag{3.28}$$

With them, the inference problem is the following system of equations:

$$\sum_{(o^*,v)\in A^*,\ v\in V_I^*} y_{ov} = 1 \tag{3.29}$$

$$\forall (u,v) \in A^*, \quad v \in V_I^* \quad y_{uv} = \sum_{(v,j)\in A^*} y_{vj} \tag{3.30}$$

$$\forall (u,v) \in A^*, \quad u \in V_I^* \quad (2w_{uv} - 1)(2y_{uv} - 1)(x[\rho^*_u] - \tau^*_u) \le 0 \tag{3.31}$$

$$h_{\rho^*,\tau^*,\lambda^*}(x) = \sum_{(u,v)\in A^*,\ v\in V_L^*} y_{uv}\lambda^*_v \tag{3.32}$$

$$\forall (u,v) \in A^* \quad y_{uv} \in \{0,1\}\,. \tag{3.33}$$

The variables y_{uv} encode the decision "use arc (u,v)" ($y_{uv} = 1$) or "do not use arc (u,v)" ($y_{uv} = 0$). Eq. (3.29) ensure that there is exactly one path from root to leaf; Eq. (3.30) are flow conservation constraints; Eq. (3.31) enforce the learned branching constraints. Eq. (3.32) supplies the DT approximation of $f(\bar{x})$. Since $x, w, \tau^*, \rho^*, \lambda^*$ are given when the inference problem is computed, Eq. (3.29)–(3.33) are all linear.

Training several DTs — potentially, on different subsets of the training set $\mathcal{S}$ and with different features — and averaging the predictions of the trained predictors gives a *random forest* [81, §15].

3.6.3 Neural Networks

A NN $\mathcal{N}$ is a computable approximation of an unknown function f, encoded by a nonempty digraph $G = (V, A)$ such that:

- the vertex set V is partitioned as $V = \{I, H, O\}$, i.e., *input*, *hidden* and *output* vertices. H may be empty, and we assume that $|O| = 1$;

- an arc function

$$w : A \longrightarrow \mathbb{R}\,,$$

called *weight*, and a vertex function

$$b : V \setminus I \longrightarrow \mathbb{R}\,,$$

called *bias*, are defined on G;

- an *activation* function

$$\sigma_u : \mathbb{R} \longrightarrow \mathbb{R}$$

labels each vertex $u \in H \cup O$.

The tuple (G, σ) is the network's *architecture*. Further, G is such that $|I| = d$ and $|O| = t$; we recall that d, t are, respectively, the size of input and output set vectors.

3.6.3.1 The training problem

Given an architecture (G, σ), the optimal w^*, b^* are the solution of the training problem

$$\begin{aligned} \min_{w,b} \quad & \sum_{i \in S} \sum_{u \in O} \ell(y_{iu}, z_{iu}) \\ \forall i \in S, \quad \forall u \in I \quad & z_{iu} = x_i[u] \\ \forall i \in S, \quad u \in H \cup O \quad & z_{iu} = \sigma_u(b_u + \sum_{(v,u) \in A} w_{vu} z_{iv})\,, \end{aligned} \tag{3.34}$$

where: ℓ is the loss function defined in Eq. (3.4); for all $u \in I$, the $x_i[u]$ is the u-th component of input x_i; for $i \in S$ and $u \in V$, the z_{iu} are auxiliary decision variables encoding the activation of each vertex. The argument of σ is known as *preactivation*. The training problem may be highly nonconvexity and/or nonlinear, depending on the activation function σ. However, it has been proven [2, 149] that, for large enough $\mathcal{N}$, there always exist many local minima capable of providing high generalization.

However, global optima in NN are sometimes associated with overfitting. RRM and early stopping techniques (treated in Sec. 3.5.2) are often applied to regularize the problem in Eq. (3.34). However, the most popular regularization method for NNs is the *dropout* [68]: it consists of solving the training problem multiple times, each time on a different subgraph of $\mathcal{N}$, obtained by randomly removing some of its vertices.

3.6.3.2 The inference problem

The approximation produced by the trained $\mathcal{N}$ is encoded by the following system of equations, with $|V|$ equations and $|V| + d$ variables:

$$\begin{aligned} \forall u \in I \quad & z_u = x_u \\ u \in H \quad & z_u = \sigma_u(b^*_u + \sum_{(v,u)\in A} w^*_{vu} z_v) \\ & h_{w^*,b^*}(x) = \sigma_o(b^*_o + \sum_{(v,o)\in A} w^*_{vo} z_v)\,, \end{aligned} \tag{3.35}$$

where $o \in O$.

Given an input $\bar{x} \in \mathcal{X}$, Eq. (3.35) becomes a $|V| \times |V|$ determined system, with a unique solution. The approximation of $f(\bar{x})$ is the vector $(z_{o_1}, \ldots, z_{o_s})$ associated with the output vertex set O.

In this thesis, we will refer to simple *feed-forward* NNs (FNN), where: G is a k-partite, directed acyclic graph; each partition is called *layer*, the cardinality of each layer is its *width* and the number of layers is the network's *depth*; each arc can only link vertices in subsequent layers. If G is a directed acyclic graph, Eq. (3.35) can be solved by exploring it in topological forward order, from I (with no incoming arcs) to O (with no outgoing arcs). A reference text to explore the diverse NN formulations is, e.g., [73].

3.6.3.3 Back-propagation

Eq. (3.34) is usually solved by some line search type algorithm, discussed in Sec. 2.8.1), which iteratively minimizes the loss function. *Back-propagation* [165] is an algorithm for computing the gradient at each iteration (or *epoch*) of gradient-based unconstrained optimization algorithms. This is accomplished in two consecutive steps:

- the *forward pass*: given w, b, the algorithm computes the output $z_u,\ u \in O$ of the network, for one or more inputs $\{x_i\}_{i\in S}$, by evaluating Eq. (3.35). Here, the loss is seen as a function of the training inputs, parametrized by w, b;

- the *backward pass*: here, instead, the loss is seen as a function of w, b, parametrized by the training input values. This function, that we call $E(w, b)$, measures the changes in the loss as w, b vary, for fixed training inputs. The backward pass calculates the partial gradient of E with respect to w, b, given the training couples $\{(x_i, y_i)\}_{i\in S}$ employed in the forward pass.

To describe the backward pass, we redefine the vector (w, b) as "w" and, for all $u \in H \cup O$, z_u as "$(z_{u1}, z_{u2}, \ldots z_{us}, 1)$", for ease of notation. With this shorthand, we can represent the preactivation

$$b_u + \sum_{(v,u)\in A} w_{vu} z_v$$

of Eq. (3.34) as the inner product

$$a_u = \underbrace{(w_{v_1u}, w_{v_2u}, \ldots, b_u)}_{w_{vu}}^{\top} \underbrace{(z_{v_1}, z_{v_2}, \ldots, 1)}_{z_v} . \tag{3.36}$$

Moreover, in the equations below, we follow the notation of [75].

The back-propagation algorithm calculates the quantity

$$\forall (v,u) \in A \quad \text{such that} \quad u \in H \cup O \quad : \quad \tfrac{\partial E}{\partial w_{vu}} . \tag{3.37}$$

The calculations begin at those arcs such that $u \in O$, and proceed by following the reverse direction of the arcs, i.e.: one layer at a time, backwards, from the output layer to the input layer. The chain rule for derivatives yields the following reformulation of Eq. (3.37):

$$\frac{\partial E}{\partial w_{vu}} = \delta_u \frac{\partial a_u}{\partial w_{vu}}, \quad \text{where} \quad \delta_u = \frac{\partial E}{\partial a_u} . \tag{3.38}$$

The quantity δ_u measures the variations in the error with respect to variations in the activation at vertex u. Because of the definition of a_u in Eq. (3.36), we have that

$$\frac{\partial a_u}{\partial w_{vu}} = z_v ,$$

so Eq. (3.38) becomes:

$$\frac{\partial E}{\partial w_{vu}} = \delta_u z_v . \tag{3.39}$$

Since z_v is known at the beginning of the backward pass, the algorithm only needs to compute δ_u:

- for all $u \in O$

  $$\delta_u = \frac{\partial E}{\partial z_u} \frac{\partial z_u}{\partial a_u} ,$$

  by the chain rule for derivatives. Notably, $\frac{\partial z_u}{\partial a_u}$ is the derivative of $\sigma'_u(a_u)$ with respect to a_u;

- for all $u \in H$,

  $$\begin{aligned} \delta_u &= \sum_{(u,v)\in A} \frac{\partial E}{\partial a_v} \frac{\partial a_v}{\partial a_u} \\ &= \sum_{(u,v)\in A} \frac{\partial E}{\partial a_v} \frac{\partial z_u}{\partial a_u} w_{uv} \qquad \text{(by Eq. (3.36))} \\ &= \tfrac{\partial z_u}{\partial a_u} \sum_{(u,v)\in A} \delta_v w_{uv} . \end{aligned}$$

3.6.3.4 Loss optimization algorithms

If gradient descent is the algorithm chosen to minimize the loss, the k-th iteration is:

$$w_{k+1} = w_k - \gamma \nabla_w E(w_k), \quad \text{such that} \quad E(w_k) = \frac{1}{s} \sum_{i \in S} E_i(w_k)\,, \tag{3.40}$$

where γ is treated as a hyperparameter of the NN and E_i is the error for a given training input x_i.

However, gradient descent might show slow convergence, so *stochastic gradient descent* [162] is sometimes preferred, where E is computed for a single, randomly sampled point $(x_i, y_i) \in \mathcal{S}$. This algorithm is less frequently trapped in plateaus (i.e., areas where the gradient is close to zero) or saddle points [27, 166, 198], and is potentially faster because it performs only one update at a time. However, it may provide an inaccurate approximation of E, yielding large fluctuations of the weights at each epoch and slow convergence, especially if γ is not carefully tuned. A compromise between the two algorithms, the *mini-batch* gradient descent algorithm calculates $E(w_k)$ for subsets of $\mathcal{S}$. Both the stochastic and the mini-batch implementations of gradient descent are easy to parallelize.

Alternatives to gradient descent type algorithms are: [165] and Nesterov accelerated method [73, §8], which try to improve the loss optimization performance in valleys (i.e., areas such that E increases swiftly in some directions and slowly in others) [177]; Adagrad and its extension Adadelta, RMSprop, adaptive moment estimation (see, for example, [27, 164] and references therein), i.e., line search type algorithms, using different step sizes for different components of w and updating them dynamically during the optimization.

3.6.4 Support Vector Regression

SVR is a supervised learning paradigm used for regression tasks. In this section, we adopt the notation of [176].

3.6.4.1 The training problem

In its simplest form, the SVR hypothesis is a *linear regression* function

$$h_{w,b}(x) = \langle w, x \rangle + b\,, \tag{3.41}$$

for $w \in \mathbb{R}^d$ and $b \in \mathbb{R}$. This single-line system of equations is the linear SVR inference problem.

The optimal coefficients w^*, b^* are the solution of the SVR training problem, i.e.,

the convex QP:

$$\min_{w\in\mathbb{R}^d,\ \xi,\hat{\xi}\in\mathbb{R}} \quad \frac{1}{2}\|w\|_2^2 + \lambda\sum_{i\in S}(\xi_i+\hat{\xi}_i) \tag{3.42}$$

$$i\in S \quad y_i - \langle w, x_i\rangle - b \le \varepsilon + \xi_i \tag{3.43}$$

$$i\in S \quad \langle w, x_i\rangle + b - y_i \le \varepsilon + \hat{\xi}_i \tag{3.44}$$

$$i\in S \quad \xi_i,\ \hat{\xi}_i \ge 0\,. \tag{3.45}$$

Constraints (3.43)–(3.44) encode the approximation accuracy required of $h_{w,b}$, by permitting a maximum deviation of $\varepsilon > 0$ from the training labels. Thus, they define an ε-tube around $h_{w,b}$, within which there is no penalty. This requirement is relaxed by the slack variables $\xi, \hat{\xi} \ge 0$, which allow prediction errors larger than ε, and are penalised in the objective (3.42) by a coefficient $\lambda > 0$. λ and ε are hyperparameters of the SVR, usually selected by CV in the training process. The learned function must be as "flat" as possible to avoid overfitting. This is achieved by placing the regularization term $\|w\|_2^2$ (a squared euclidean norm) in the objective, although other methods are possible (see, e.g., [201]). The solution of Eq. (3.42)–(3.44) comes with strong global optimality guarantees, unlike other ML methodologies.

3.6.4.2 Dual formulation

Solution algorithms for the SVR exploit results from the duality theory outlined in Sec. 2.3. The Lagrangian function associated with the primal formulation in Eq. (3.42)–3.45 is

$$\begin{aligned} L(w,b,\xi,\hat{\xi},\eta,\hat{\eta},\alpha,\hat{\alpha}) = \tfrac{1}{2}||w||_2^2 + \lambda\sum_{i\in S}(\xi+\hat{\xi}_i) - \\ \sum_{i\in S}(\eta_i\xi + \hat{\eta}_i\hat{\xi}_i) - \sum_{i\in S}\alpha_i(\varepsilon+\xi-y_i+\langle w,x_i\rangle+b) \\ -\sum_{i\in S}\hat{\alpha}_i(\varepsilon+\hat{\xi}_i+y_i-\langle w,x_i\rangle-b)\,, \end{aligned} \tag{3.46}$$

with Lagrangian multipliers $\alpha,\ \hat{\alpha},\ \eta,\ \hat{\eta} \ge 0$. Since the primal formulation is convex, there exists a unique optimum $\left(w^*, b^*, \xi^*, \hat{\xi}^*,\ \alpha^*, \hat{\alpha}^*\right)$ satisfying the KKT conditions [35], i.e.:

$$\forall i\in S \begin{cases} y_i - \langle w^*, x_i\rangle - b^* & \le & \varepsilon + \xi_i^* \\ \langle w^*, x_i\rangle + b^* - y_i & \le & \varepsilon + \hat{\xi}_i^* \\ \xi_i^*, \hat{\xi}_i^* & \ge & 0 \end{cases} \tag{3.47}$$

$$\forall i\in S \quad \alpha_i^*, \hat{\alpha}_i^* \ \ge\ 0 \tag{3.48}$$

$$\nabla_b L \ = \ \sum_{i\in S}(\hat{\alpha}_i^* - \alpha_i^*) \ = \ 0 \tag{3.49}$$

$$\nabla_w L \ = \ w^* - \sum_{i\in S}(\alpha_i^* - \hat{\alpha}_i^*)x_i \ = \ 0 \tag{3.50}$$

$$\forall i \in S \begin{cases} \nabla_{\xi} L &= C - \alpha_i^* - \eta_i^* &= 0 \\ \nabla_{\hat{\xi}_i} L &= C - \hat{\alpha}_i^* - \hat{\eta}_i^* &= 0 \end{cases} \tag{3.51}$$

$$\forall i \in S \begin{cases} \alpha_i^* \left(\varepsilon + \xi_i^* - y_i + \langle w^*, x_i \rangle + b^*\right) &= 0 \\ \hat{\alpha}_i^* \big(\varepsilon + \hat{\xi}_i^* + y_i - \langle w^*, x_i \rangle - b^*\big) &= 0 \\ (C - \alpha_i^*)\xi_i^* &= 0 \\ (C - \hat{\alpha}_i^*)\hat{\xi}_i^* &= 0\,. \end{cases} \tag{3.52}$$

Notably, from constraints (3.47) and (3.52) we gather that $\alpha_i = 0$ (resp. $\hat{\alpha}_i = 0$) implies $\xi_i = 0$ (resp. $\hat{\xi}_i = 0$), and hence $y_i - h_\theta(x_i) \leq \epsilon$ (resp. $h_\theta(x_i) - y_i \leq \epsilon$). The training points lying outside the ε-tube, i.e., the ones with positive multipliers α, are called *support vectors*. Furthermore, α_i and $\hat{\alpha}_i$ cannot both be nonzero, as this would mean that x_i is over and under the ε-tube at the same time.

With the above KKT conditions, the dual SVR formulation is as follows:

$$\begin{array}{ll} \max\limits_{\alpha} & L(\alpha) \\ & \displaystyle\sum_{i \in S} (\alpha_i - \hat{\alpha}_i) = 0 \\ i \in S & \alpha_i,\ \hat{\alpha}_i \in [0, C]\,, \end{array} \tag{3.53}$$

where

$$\begin{aligned} L(\alpha) \quad &= -\frac{1}{2} \sum_{i,j \in S} (\alpha_i - \hat{\alpha}_i)(\alpha_j - \hat{\alpha}_j)\langle x_i, x_j \rangle \\ &\quad -\varepsilon \sum_{i \in S} (\alpha_i + \hat{\alpha}_i) + \sum_{i \in S} y_i(\alpha_i - \hat{\alpha}_i) \end{aligned}$$

is obtained by enforcing the KKT stationarity conditions (3.49)–(3.51) on the function (3.46). The dual constraints directly follow from the KKT conditions (3.48), (3.49) and (3.51).

The dual solution $\alpha^*, \hat{\alpha}^*$, together with Eq. (3.50), supply the optimal w^*, i.e.,

$$w^* = \sum_{i \in S} (\hat{\alpha}_i^* - \alpha_i^*) x_i\,. \tag{3.54}$$

In particular, w^* is completely defined by a linear combination of the training inputs, so it needs not be computed explicitly. The optimal value b^*, instead, follows from the constraints in Eq. (3.47) and (3.52):

$$\begin{aligned} \max_{i \in S} \{ y_i - \langle w^*, x_i \rangle - \varepsilon \mid \alpha_i^* < C \text{ or } \hat{\alpha}_i^* > 0 \} \ &\leq \ b^* \\ \leq \ \min_{i \in S} \{ y_i - \langle w^*, x_i \rangle - \varepsilon \mid \alpha_i^* > 0 \text{ or } \hat{\alpha}_i^* < C \}\,. \end{aligned}$$

An alternative way to find b^* is by using interior point methods to solve Eq. (3.42)–(3.45) [176].

The optimal w^* and b^* give

$$h_{w^*, b^*}(x) = \sum_{i \in S} (\hat{\alpha}_i^* - \alpha_i^*)\langle x_i, x \rangle + b^*\,. \tag{3.55}$$

3.6.4.3 The kernel trick

Duality theory allows the application of the *kernel trick*, a methodology for extending the linear SVR to make it capable of learning even complicated nonlinear functions. The kernel trick reduces problematic nonlinear transformations of vectors in $\mathcal{X}$ to the simpler computation of inner products. It is based on a feature function Φ, such as the one defined by Eq. (3.16) in Sec. 3.5.4, but where the feature set $\mathcal{F}$ is a Hilbert space. Given Φ, a *kernel* is a function

$$\kappa : \mathcal{X} \times \mathcal{X} \longrightarrow \mathbb{R} \tag{3.56}$$

such that

$$\forall x_1, x_2 \in \mathcal{X}, \quad \kappa(x_1, x_2) = \langle \Phi(x_1), \Phi(x_2) \rangle_{\mathcal{F}} \,,$$

where $\langle \cdot, \cdot \rangle_{\mathcal{F}}$ is the inner product in $\mathcal{F}$. The kernel function is symmetric and, in fact, the accompanying *Gram matrix*

$$K = \big(\kappa(x_i, x_j)\big)_{i,j \in S}$$

is positive semi-definite for any choice of Φ. We add that this is a necessary and sufficient condition for a function to be a kernel [84]. The alternative for constructing a nonlinear SVR without a kernel would be to first use a nonlinear Φ on $\mathcal{X}$ and, next, learn a linear hypothesis on the points of $\mathcal{F}$. However, computing the actual coordinates of Φ can be extremely costly. Instead, since SVR relies on inner products between the training points, one can compute the kernel function in Eq. (3.56) without even knowing Φ. This is the kernel trick. Its cost amounts to the calculation of (half of) the Gram matrix, which only involves cheap operations on the training inputs.

The Gaussian kernel

$$i \in S \quad k(x_i, x) = \exp\Big(-\gamma \|x_i - x\|_2^2 \Big) \,, \tag{3.57}$$

adds one hyperparameter to SVR ($\gamma \in \mathbb{R}^+$) and is the default choice in absence of any other meaningful prior on $\mathcal{S}$. Other kernels, and some methods to create new ones, are presented in [44].

The kernel formulation of SVR is obtained by replacing $\langle x_i, x_j \rangle$ in Eq. (3.55) with $\langle \Phi(x_i), \Phi(x_i) \rangle$. This gives the kernel-based SVR hypothesis

$$h_{\alpha^*, b^*}(x) = \sum_{i \in S} (\hat{\alpha}_i^* - \alpha_i^*) \kappa(x_i, x) + b^* \,. \tag{3.58}$$

3.7 Conclusions

In this chapter we presented a framework for supervised ML. We also described, in terms of MP, the training problem, i.e., the question of how to construct a computable

approximation of a function whose analytical expression is unknown, but which can be sampled on a set of points (called training set). We then discussed several techniques that are customarily adopted to improve the accuracy of the produced approximations. Further, we presented the ML paradigms deployed in the computational experiments that we performed to test our ACP approaches, described in Ch. 5 and 6. For each of these paradigms, we considered the issue of how to encode the mathematical properties of the learned approximation, and provided the corresponding formulation.

Part II

Algorithm configuration problem

Chapter 4

The algorithm configuration problem

4.1 Introduction

Automatic configuration of algorithmic parameters is an area of active research [77, 105], going back to the foundational work [161], published in 1976. The ACP focuses on configurable algorithms, deployed to solve instances of a given decision or optimization problem. It concerns the issue of how to identify the setup of algorithmic parameters delivering the best algorithmic performance, when the algorithm is run on its input.

Formally, we let $\mathcal{A}$ be a parametrized target algorithm. Its input consists of an instance of the problem being solved and an array of parameters. We call the latter "algorithmic configuration".

The inputs of the ACP are:

- Π: the decision/optimization problem to be solved by $\mathcal{A}$, consisting of a (potentially) infinite set of instances. Each instance is an input string for the problem; among the many encodings available for instance data, the most common one is a vector of discrete/continuous values, containing the most important attributes of the instance. In the following, we assume that several encodings can be transformed into each other efficiently (i.e., without too much loss of information) and we refer to Π as the set of *encoded* instances;

- $\mathcal{C}_{\mathcal{A}}$: the set of parameter configurations of $\mathcal{A}$, i.e., an array of data of different types (boolean, numeric, categorical), usually encoded by vectors of q continuous and/or discrete/categorical values. Not all possible parameter values may be admissible, due to logical conditions concerning multiple parameters. Thus, for simplicity, we assume that $\mathcal{C}_{\mathcal{A}}$ only contains feasible algorithmic configurations;

- $p_{\mathcal{A}}$: the performance function of $\mathcal{A}$

$$p_{\mathcal{A}} : \Pi \times \mathcal{C}_{\mathcal{A}} \longrightarrow \mathbb{R} \tag{4.1}$$

  mapping a pair (π, c) (instance, parameter configuration) to the outcome of running $\mathcal{A}$, configured by c, to solve an instance π. The encoding of $p_{\mathcal{A}}$ is a single continuous or discrete value. The performance $p_{\mathcal{A}}$ related to running $\mathcal{A}$ could be a cost measure (e.g., CPU time, number of iterations performed, etc.), or a quality measure (e.g., the accuracy achieved by a Machine Learning (ML) predictor at a certain iteration of the training process, the integrality gap reported by an optimization solver within a certain time limit, etc.). Depending on the case at hand, one aims at appropriately minimizing or maximizing it.

With the specifications given above, the ACP is formally defined as follows:

Definition 4.1.1 (ACP)**.** *Given a tuple* $(\mathcal{A}, \bar{\pi}, p_{\mathcal{A}})$, $\bar{\pi} \in \Pi$*, find the algorithmic configuration* $c^*_{\bar{\pi}} \in \mathcal{C}_{\mathcal{A}}$ *providing the optimal performance* $p_{\mathcal{A}}$ *of* $\mathcal{A}$ *on* $\bar{\pi}$*.*

A variant of the ACP is the Algorithm Selection Problem (ASP), where one seeks to pick, from a given set of configured algorithms, the best one for solving a specific instance. However, one can see the choice of which algorithm to pick as the one single parameter of a meta-algorithm for solving Π, and therefore the ASP is a special case of the ACP. The ACP is generally very hard both in theory and in practice, especially when, as it often happens, algorithms have a large number of configurable parameters. Yet, it has a large number of very relevant applications, such as the configuration for of constraint programming or mathematical optimization solvers, the hyperparameter tuning of ML pipelines, the administration of ad-hoc medical treatments, and many others; the interested reader is referred, e.g.,to [57, 105] and the references therein for a more detailed treatment of the subject.

In this chapter, we review the ACP literature. Notably, we supply a comprehensive framework for describing any approach to the ACP: we identify the core building components for designing an ACP solution strategy, and discuss their possible use patterns and implementation (Sec. 4.1–4.4). In the second part of the chapter, we illustrate how these components are combined and deployed in the various ACP methodologies (Sec. 4.5–4.7). Lastly, we introduce the approaches we devised to address the ACP and place them within the descriptive framework discussed in the first part of the chapter. This will allow us to compare our approaches with the other works in the literature, and outline our main contributions (Sec. 4.8).

Note that, despite the existence of surveys on the subject (in particular, the one in [104]), the framework we describe here offers a novel stance on the ACP. In fact, part of the content of this chapter has been accepted for publication as an entry in the next edition of the "Encyclopedia of Optimization" (whose last published version is [65]).

4.2 An algorithmic schema

Research efforts on the ACP have focused on developing methodologies to solve it efficiently in an automated fashion. Although this has been implemented in several different ways, all approaches share the same goal, i.e., the construction of a *recommender*:

Definition 4.2.1 (recommender)**.** *The recommender is a function*

$$\Psi_{\mathcal{M}} : \Pi \to \mathcal{C}_{\mathcal{A}} \tag{4.2}$$

which, given an instance $\pi \in \Pi$, is capable of selecting a configuration $c^*_\pi \in \mathcal{C}_{\mathcal{A}}$ for solving π more efficiently than with other configurations of $\mathcal{A}$.

Thus, c^*_π is, hopefully, a good approximation of the optimal configuration for π, with respect to the performance function $p_{\mathcal{A}}$.

In other words, a recommender is a heuristic for the ACP. The fundamental constraint is that the recommender should be able to produce its output in a "short" time, so as not to offset the advantages due to choosing a better configuration.

Our notation underlines the fact that the structure of $\Psi_{\mathcal{M}}$ is usually determined by a model $\mathcal{M}$ encoding some knowledge about $p_{\mathcal{A}}$. In fact, an important phase of all ACP methodologies is devoted to the building of $\mathcal{M}$. It is in general difficult (if at all possible) to construct accurate enough analytical models of the performances of complex algorithms. Thus, most practical $\mathcal{M}$ are "data-driven", in the sense that they are constructed from experiments. For the recommender to be able to choose the right c^*_π for a given π, it should be able to assess $p_{\mathcal{A}}$ anywhere on the set $\Pi \times \mathcal{C}_{\mathcal{A}}$. However, an exhaustive evaluation of $p_{\mathcal{A}}$ over $\Pi \times \mathcal{C}_{\mathcal{A}}$ is almost always impossible in practice. In fact, Π is an infinite set, and $\mathcal{C}_{\mathcal{A}}$ usually grows exponentially in the number of parameters, which can be large. Furthermore, since $p_{\mathcal{A}}$ itself is typically a black-box function (i.e., it has no analytic form), the only way to evaluate it is to directly run $\mathcal{A}$, which can be extremely costly. Therefore, a significant component of ACP approaches is how the set $\Pi \times \mathcal{C}_{\mathcal{A}}$ is explored.

Given the difficulty of assessing algorithmic performance on $\Pi \times \mathcal{C}_{\mathcal{A}}$ exhaustively, the construction of $\Psi_{\mathcal{M}}$ in Eq. (4.2) always involves the selection of sets $\Pi' \subset \Pi$ and $\mathcal{C}'_{\mathcal{A}} \subseteq \mathcal{C}_{\mathcal{A}}$. Of these, Π' is meant to be "representative" of Π, usually in the sense that it preserves the characteristics and information of Π. Sometimes, this can instead be taken as the fact that Π' contains the most "difficult" instances for $\mathcal{A}$, in that all others are solved efficiently and do not require a dedicated algorithmic configuration. Since there is no automatic way of choosing Π', most ACP approaches take it as given and rely on existing libraries, hand-picked by problem experts. Therefore, we assume that Π' is always available, or can be easily generated, for deploying an ACP methodology. Further, we assume that Π' is specified before the construction of $\mathcal{M}$, and never updated.

Moreover, the algorithmic performance of different configurations is typically unknown before launching an ACP approach, otherwise, there would be no need to even construct a recommender $\Psi_{\mathcal{M}}$. This means that $\mathcal{C}'_{\mathcal{A}}$ is often selected during the construction of $\Psi_{\mathcal{M}}$, instead of being picked *a priori*, as in the case of Π'; we remark that the (partly constructed) model $\mathcal{M}$ can also be useful in this context.

In all approaches in the literature, solving the ACP fits into the same two-stage framework (see Fig. 4.1), encompassing the ordered execution of:

a) a *Knowledge-encoding Process* (for brevity, K-EP). The K-EP builds $\mathcal{M}$ and the accompanying $\Psi_{\mathcal{M}}$. A critical step in the computation of $\mathcal{M}$ is the sampling of the performance function, i.e., the evaluation of $p_{\mathcal{A}}$ over pairs $(\pi, c) \in \Pi' \times \mathcal{C}_{\mathcal{A}}$. Since $\mathcal{C}_{\mathcal{A}}$ may be quite large, in most cases computing $p_{\mathcal{A}}$ on all the configurations is too expensive. Thus, in all ACP approaches, the selection of an appropriate subset of $\mathcal{C}_{\mathcal{A}}$ in the K-EP is a crucial task, which may require the use of $\mathcal{M}$ or additional models. Instead, when $\mathcal{C}_{\mathcal{A}}$ is small, all the $c \in \mathcal{C}_{\mathcal{A}}$ can be considered;

b) a *Recommendation Phase*. The recommendation phase deploys $\Psi_{\mathcal{M}}$, in order to produce a suitable configuration for a given instance. Thus, $\Psi_{\mathcal{M}}$ supplies a solution of the ACP for that instance.

Since, as we noted above, most $\mathcal{M}$ are data-driven, the K-EP can demand considerable computational resources. The recommendation phase can also be computationally expensive, in that exploiting $\mathcal{M}$ to produce the output configuration may involve, e.g., the solution of a nontrivial optimization problem in itself.

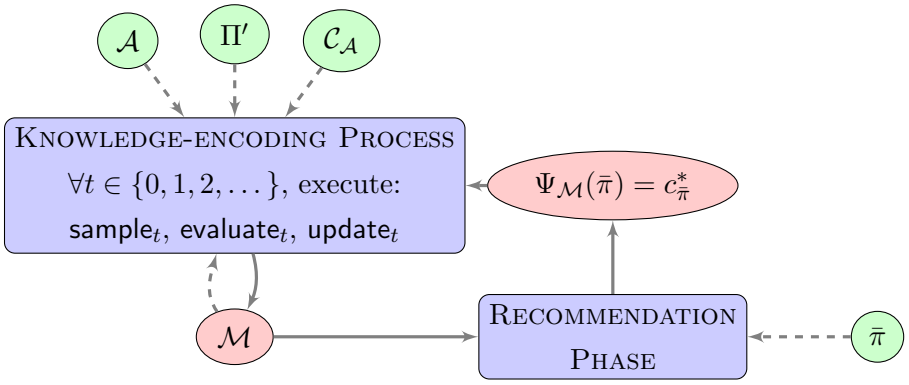

Figure 4.1: Algorithmic schema of ACP approaches

The K-EP is an iterative procedure. It cycles through three phases at each iteration $t \in \{0, \dots, T\}$, until an allotted computational budget (quantified, e.g., in terms of allowed target algorithm runs, CPU/GPU time and power, memory usage, number of K-EP iterations) is used up, or $\mathcal{M}$ has attained a desired accuracy:

1. sample_t: picks a set

$$\mathscr{S}_t \subset \Pi' \times \mathcal{C}_{\mathcal{A}} \quad s.t. \quad \mathscr{S}_t \cap \bigcup_{h<t} \mathscr{S}_h = \emptyset\,, \tag{4.3}$$

   i.e., $\mathscr{S}_t$ only contains previously unsampled couples. This selection is nontrivial, as it requires addressing the trade-off between uniformly exploring $\Pi' \times \mathcal{C}_{\mathcal{A}}$, so that no promising area is left unexplored (diversification), and concentrating on the areas containing the most promising candidates, so as to find better solutions (intensification);

2. $\mathsf{evaluate}_t$: executes (possibly, in parallel) the target algorithm $\mathcal{A}$ on all the points picked by sample_t, to compute $p_{\mathcal{A}}$ at those points and build the set

$$\mathcal{S}_t = \left\{ \left(\pi,\ c,\ p_{\mathcal{A}}(\pi, c) \right) \mid (\pi, c) \in \mathscr{S}_t \right\}; \tag{4.4}$$

3. update_t: updates the models employed in the K-EP, i.e., the model $\mathcal{M}$ of $\Psi_{\mathcal{M}}$ and, potentially, other models used for sampling purposes. For instance, it may entail training or re-training an ML model. This phase exploits the set $\bigcup_{h\leq t} \mathscr{S}_h$, the performance values computed at the points of that set and, sometimes, some other information collected in the $\mathsf{evaluate}_t$ phase.

The data generated by the recommendation phase may be employed in an adaptive "meta-sampling" loop whereby: a) when a new instance $\bar{\pi} \in \Pi \setminus \Pi'$ is given (either by the user of the recommender, or by a dedicated process aimed at improving its quality), one computes $\Psi_{\mathcal{M}}(\bar{\pi})$; b) the recommended configuration and/or $\bar{\pi}$ are fed into the K-EP, which can be performed again to improve $\mathcal{M}$. One example of this approach is presented in [96].

The implementation of the K-EP and the recommendation phase depends on the choice of the following components:

- a model $\mathcal{M}$ and the associated recommender $\Psi_{\mathcal{M}}$;
- whether $\Psi_{\mathcal{M}}$ actually depends on a specific instance or always provides the same answer for a set of instances: we call *Per-Instance* (PI) ACP approaches of the former type, and *Per-Problem* (PP) approaches of the second type;
- whether one commits to building $\mathcal{M}$ before actually solving an unknown instance by $\mathcal{A}$ (*offline* ACP methodologies) or $\mathcal{M}$ is constructed during an algorithm run (*online* methodologies).

4.3 The construction of $\mathcal{M}$

In the literature, the model $\mathcal{M}$ built in the K-EP is one of the following:

(a) a function

$$\zeta_{\mathcal{A}} : \Pi \longrightarrow \mathcal{C}_{\mathcal{A}}\,, \tag{4.5}$$

mapping an instance encoding to the configuration recommended for that instance. The function in Eq. (4.5) is usually constructed by ML techniques [24, 19, 25, 94];

(b) a function

$$\bar{p}_{\mathcal{A}} : \Pi \times \mathcal{C}_{\mathcal{A}} \longrightarrow \mathbb{R}\,, \tag{4.6}$$

computing an approximation of the performance function $p_{\mathcal{A}}$ defined by Eq. (4.1), also generally built by ML techniques [17, 23, 86, 93]. Sometimes [36], $\bar{p}_{\mathcal{A}}$ is aggregated (e.g., averaged) over the instances in Π', which yields an estimate

$$\chi_{\mathcal{A}} : \mathcal{C}_{\mathcal{A}} \longrightarrow \mathbb{R} \tag{4.7}$$

of the performance of single configurations over Π. The functions $\bar{p}_{\mathcal{A}}$ or $\chi_{\mathcal{A}}$ are then used as a proxy to recommend a configuration for the new instance, typically, by solving an optimization problem having them in the objective;

(c) a partition

$$\mathscr{P}_{\mathcal{A}} = \{\Pi'_i \subseteq \Pi'\}_{i \in C} \tag{4.8}$$

of Π' into C disjoint subsets (or "clusters") Π'_i, whereby each Π'_i is specified by choosing the corresponding recommended configuration c^*_i and, for instance, a representative instance π_i. When a new instance $\bar{\pi} \in \Pi$ has to be solved, the cluster to which it belongs is determined (e.g., by finding the closest representative π_i, under some appropriate distance metric) and the corresponding c^*_i is retrieved.

We remark that there are two "extreme" cases of Eq. (4.8):

- one in which $C = |\Pi'|$, i.e., each Π'_i contains exactly one instance, [14, 156]. We refer to this case as "$\mathscr{P}_{\mathcal{A},|\Pi'|}$";
- one whereby $\Pi'_0 = \Pi'$ and $C = 1$, i.e., the partition is trivial and a single configuration c^*_0 will be recommended regardless of the input $\bar{\pi}$. This is customary in PP approaches [3, 7, 9, 10, 45, 36, 88, 89, 147, 138]. We refer to this case as "$\mathscr{P}_{\mathcal{A},1}$".

Moreover, we refer to intermediate case, whereby $1 < C < |\Pi'|$, as "$\mathscr{P}_{\mathcal{A},C}$".

It should be noted that this choice of $\mathcal{M}$ is typically strongly coupled with the sample$_t$ phase in the K-EP, in the sense that it is often the direct result of the algorithmic decisions there.

The strategy implemented to construct $\mathscr{P}_{\mathcal{A}}$ is that of solving the problem

$$c^*_i = \arg\min\{\, \mathsf{agg}_{\pi \in \Pi_i}\, p_{\mathcal{A}}(\pi, c) \,|\, c \in \mathcal{C}_{\mathcal{A}} \,\}\,, \tag{4.9}$$

where agg is some aggregation function (say, the average). The issue here is to find the configuration providing the best aggregated performance with respect to the subset Π'_i at hand. Since $\mathcal{C}_\mathcal{A}$ is often very large and $p_\mathcal{A}$ is "black-box", the problem in Eq. (4.9) is typically treated by heuristic search algorithms, such as local searches, evolutionary algorithms or other metaheuristics, providing a local solution. Further, such heuristic algorithms can themselves naturally identify clusters on the fly, even in the extreme cases; for instance, evolutionary algorithms produce a population of the fittest individuals, each of which can be used to define a cluster representative π_i.

We remark that the strategy of building $\mathscr{P}_\mathcal{A}$ by solving Eq. (4.9), usually adopted by offline ACP methodologies, quite naturally extends to the online setting. In the online ACP, further evaluations of $p_\mathcal{A}$ are often triggered by the arrival of a new π, that restarts the optimization process.

Some ACP methodologies combine more than one model; the possible combinations are shown in Tab. 4.1.

Below, we provide further details about the sample_t and update_t phases, executed during the K-EP and introduced in Sec. 4.2. Moreover, we see how the implementation of each of these phases changes in function of the selected model $\mathcal{M}$.

4.3.1 The sample_t phase

The purpose of the sample_t phase is to pick the set $\mathscr{S}_t$ of Eq. (4.3). This is usually accomplished by random sampling or by more sophisticated techniques, i.e., Design Of Experiments (DOE) [43]. In particular, DOE is a family of methods for the design and implementation of experiments, whose purpose is to model the relationship between a set of input variables (the so-called "factors") and outputs. These methods involve systematically varying the inputs, i.e., according to a specific "design", and observing the resulting changes in the outputs. However, in some cases, alternative strategies are used in sample_t. For instance, at iteration $t = 0$, $\mathscr{S}_t$ is sometimes selected manually; this is possible only when the expertise of someone knowledgeable about the problem Π or the target algorithm $\mathcal{A}$ is available. Otherwise, at iteration $t > 0$, sample_t might employ one the models $\mathcal{M}$ described in Sec. 4.3 — instead of random sampling or DOE — for a set of sampled instances, to select the configurations in $\mathscr{S}_t$. The possible scenarios are as follows:

- sample_t with $\zeta_\mathcal{A}$ (Eq. (4.5)): for each sampled π, select the configuration $\zeta_\mathcal{A}(\pi)$ (see, e.g., [32]);

- sample_t with $\bar{p}_{\mathcal{A}}$ (Eq. (4.6)): for each sampled π, select the configuration(s) solving the problem (see, e.g., [88, 86, 36])

$$\arg\max_{c \in \mathcal{C}'_{\mathcal{A}}} \bar{p}_{\mathcal{A}}(\pi, c)\,. \tag{4.10}$$

  If $\chi_{\mathcal{A}}$ (Eq. (4.7)) is used instead of $\bar{p}_{\mathcal{A}}$, a single configuration solving the problem

$$\arg\max_{c \in \mathcal{C}'_{\mathcal{A}}} \chi_{\mathcal{A}}(c)\,. \tag{4.11}$$

  over all sampled instances, is selected (see, e.g., [147]).

In the ACP literature, Eq. (4.10) and (4.11) are often tackled by algorithms which are local and/or approximate, owing to the size of the problems and their possibly black-box nature.

4.3.2 The update_t phase

The $\mathsf{evaluate}_t$ phase concerns the construction of the set $\mathcal{S}_t$ (defined by Eq. (4.4)), which is implemented by computing $p_{\mathcal{A}}$ at all points of $\mathscr{S}_t$. Then, the update_t phase is dedicated to the initialization and update of the (one or more) $\mathcal{M}$ models built in the K-EP. We discuss the possible scenarios below:

- update_t with $\zeta_{\mathcal{A}}$ (Eq. (4.5)) or $\bar{p}_{\mathcal{A}}$ (Eq. (4.6)): $\mathcal{M}$ is an ML predictor, learned from the training set $\mathcal{S}_0$ at iteration $t = 0$; at subsequent iterations, if sample_t and $\mathsf{evaluate}_t$ are run again to update the training set, $\mathcal{M}$ is re-trained using $\bigcup_{h \leq t} \mathcal{S}_h$, in the hope of improving its generalization capabilities.

  If, instead of or in addition to $\bar{p}_{\mathcal{A}}$, the K-EP also employs $\chi_{\mathcal{A}}$ (Eq. (4.7)), the model is typically initialized as a uniform distribution, and immediately used in sample_0 to pick new points. In this case, the update_t (for $t \geq 0$) may be used to carry out adjustments of the distribution parameters, so as to focus subsequent samplings around the currently best performing configurations;

- update_t with $\mathscr{P}_{\mathcal{A}}$ (Eq. (4.8)): for each cluster $\Pi'_i \in \Pi'$, the saved configuration c_i^* is updated, if necessary, by looking at the algorithmic performance data in $\bigcup_{h \leq t} \mathcal{S}_h$.

4.4 Computing $\Psi_{\mathcal{M}}$

Given an instance $\bar{\pi} \in \Pi$, the implementation of $\Psi_{\mathcal{M}}(\bar{\pi})$ depends on the model $\mathcal{M}$ built during the K-EP:

(a) If $\mathcal{M}$ is the one in Eq. (4.5), then $\Psi_{\mathcal{M}}(\bar{\pi}) := \zeta(\bar{\pi})$;

(b) if $\mathcal{M}$ is the one in Eq. (4.6), then

$$\Psi_{\mathcal{M}}(\bar{\pi}) := \arg\min\{\, \bar{p}_{\mathcal{A}}(\bar{\pi}, c) \,|\, c \in \mathcal{C}_{\mathcal{A}} \,\}\,. \tag{4.12}$$

The problem in Eq. (4.12) may be a hard one, calling for heuristic solution algorithms akin to those possibly employed in the K-EP, i.e., treating the approximation $\bar{p}_{\mathcal{A}}$ as a "black box" (e.g., [17, 86, 195, 197, 36, 137]). However, exploiting the mathematical structure of $\bar{p}_{\mathcal{A}}$ is also possible, allowing the use of exact approaches [93];

(c) if $\mathcal{M}$ is specified by a set of clusters $\Pi'_i \subseteq \Pi'$ as in Eq. (4.8), then $\Psi_{\mathcal{M}}(\bar{\pi})$ is implemented by first solving

$$h(\bar{\pi}) = \text{argmin}_i\, \mathsf{dist}(\Pi'_i, \bar{\pi})\,, \tag{4.13}$$

for some appropriate distance function $\mathsf{dist}(\cdot,\cdot)$, and then returning $c^*_{h(\bar{\pi})}$. If the clusters are specified via a representative instance, say, π_i for cluster Π_i, then

$$\mathsf{dist}(\Pi'_i, \bar{\pi}) = \|\bar{\pi} - \pi_i\|\,,$$

for some appropriate norm $\|\cdot\|$. When the number of clusters is low, as it usually happens, Eq. (4.13) can be quickly solved by direct enumeration. In the extreme case with only one cluster, instead, the problem is trivial.

We describe special cases, where more than one model $\mathcal{M}$ is computed in the K-EP, in Sec. 4.6–4.7.

4.5 Classifying algorithm configuration approaches

PP approaches search for the configuration with the best overall performance over a problem set; they are usually based on one of the models $\mathcal{M}$ described at point (c) of Sec. 4.3. The main risk of PP approaches is that they produce suboptimal ACP solutions when the performance of the target algorithm varies considerably between instances of a problem. This is to be expected for large problem classes, e.g. MILP, where instances can represent very different problems that are hard for very different reasons (say, having very few feasible solutions, so that finding even one is challenging, or very many feasible solutions with very close objective, so that finding the optimal one is challenging).

When the risk of selecting suboptimal configurations, PI methodologies, which assume that the ACP-optimal algorithmic configuration depends on the instance at hand, are likely to achieve better results.

An ACP methodology is offline if it commits to building $\mathcal{M}$ before the recommendation phase, during which, therefore, $\mathcal{M}$ is fixed. Instead, an online methodology

performs the entire K-EP during the execution of $\mathcal{A}$. In this case, recommendation phase and K-EP coincide, and the information retrieved by running $\mathcal{A}$ is dynamically exploited, on-the-fly, to build $\mathcal{M}$. Online methods can only be employed when the target algorithm is launched to solve a sequence of instances, or is tasked with making a series of decisions during its run.

An online procedure can be used as a component of a larger offline/online approach. One way to accomplish this would be to construct, online, $\mathcal{M}$ through experiments on Π', and, then, deploy it as a recommender for instances similar to those in Π'. A very straightforward implementation of this approach would be to, say, run an optimization solver on a sequence of instances Π', within a prescribed time limit, trying different parameter configurations on each of them. This would allow the selection and storage of a set of parameter values ensuring high solver performance (as in model $\mathscr{P}_{\mathcal{A}}$ in Eq. (4.8)); they could be reused to configure the solver, and to run it on different instances in Π. For example, the CPLEX Tuning Tool [91, Ch. 10], of the IBM ILOG CPLEX solver, implements this procedure.

Another way to reuse an online $\mathcal{M}$ could be to employ it in the update_0 phase of a subsequent K-EP, to initialize the construction of a new model. Yet another option may be to use the points (instance, configuration, related performance), sampled to construct $\mathcal{M}$ online (by the sample_t and the $\mathsf{evaluate}_t$ phases), in the sample_0 phase of a following K-EP.

Since the purpose of online approaches is to solve the ACP on-the-run, they are usually based on simple algorithms, which allow for rapid decision-making. However, these approaches are often impractical to scale to large configuration sets. For this reason, they are ordinarily used for solving the ASP, rather than the ACP.

	PI		PP
$\mathcal{M}$	offline	online	offline
$\zeta_{\mathcal{A}}$ (Eq. (4.5))	[24]*, [19, 25], **[94]**		
$\bar{p}_{\mathcal{A}}$ (Eq. (4.6))	[17, 23, 86], **[93]**		
$\mathscr{P}_{\mathcal{A},C}$ (Eq. (4.8))	[40, 96]*, [101]		
$\mathscr{P}_{\mathcal{A},1}$ (Eq. (4.8))		[14]	[3, 7, 9, 10, 45] [89, 147, 138]
$\mathscr{P}_{\mathcal{A},\lvert\Pi'\rvert} + \chi_{\mathcal{A}}$ (Eq. (4.8) + (4.7))		[156]	
$\mathscr{P}_{\mathcal{A},1} + \bar{p}_{\mathcal{A}}$ (Eq. (4.8) + (4.6))	[137, 195, 197]	[137]	[88]
$\mathscr{P}_{\mathcal{A},1} + \chi_{\mathcal{A}}$ (Eq. (4.8) + (4.7))			[36]

Table 4.1: A schematic summary of the ACP literature; * indicates ASP approaches.

In Tab. 4.1, we give an overview of the main ACP approaches in the literature, subdividing them into PI, PP, offline and online, and classifying them by the models $\mathcal{M}$ produced in the K-EP. We have highlighted our contributions in bold.

From the rightmost column of the table, we gather that all PP methodologies are

based on the model $\mathscr{P}_{\mathcal{A}}$ described by Eq. (4.8), notably, on its $\mathscr{P}_{\mathcal{A},1}$ variant. Instead, the $\mathscr{P}_{\mathcal{A},C}$ and $\mathscr{P}_{\mathcal{A},|\Pi'|}$ variants are always used in the PI setting. Further, most PI approaches rely on the ML-derived models $\bar{p}_{\mathcal{A}}$ of Eq. (4.6) and $\zeta_{\mathcal{A}}$ of Eq. (4.5). In fact, PI methodologies are required to capture/encode the complicated, possibly nonlinear relationships between $p_{\mathcal{A}}$, $\mathcal{C}_{\mathcal{A}}$ and Π, and several ML paradigms are capable of producing accurate approximation. In some cases, ML-based models and variants of $\mathscr{P}_{\mathcal{A}}$ are combined, in order to implement PI procedures: see, e.g., the approaches on the second to last line of the table, which rely on both $\mathscr{P}_{\mathcal{A},1}$ and $\bar{p}_{\mathcal{A}}$ for online and offline algorithm configuration.

The rest of the chapter is devoted to a more detailed analysis of the approaches shown in Tab. 4.1.

4.6 Per-problem approaches

In this section, we present an overview of the PP methodologies that we deem to be the most significant in the ACP literature. All of them are of the offline type, and their K-EP focuses primarily on the construction of model $\mathscr{P}_{\mathcal{A},1}$ of Eq. (4.8), accomplished through the deployment of a diverse set of heuristic algorithms. Clearly, their implementation of the recommender is also the same (see Sec. 4.4): $\Psi_{\mathcal{M}}$ is a constant function, always relying on the same configuration by $\mathscr{P}_{\mathcal{A},1}$ to solve any new instances. Therefore, we will only describe their K-EP. Further, all of these approaches require that the user provides a set of instances of the optimization/decision problem solved by $\mathcal{A}$. In Sec. 4.2, we called this set Π'.

4.6.1 Approaches based on evolutionary algorithms

Several PP methodologies rely on evolutionary algorithms [199] for constructing $\mathscr{P}_{\mathcal{A},1}$ in the K-EP. Evolutionary algorithms are metaheuristics (see Sec. 2.6.1.4) for optimization problems, inspired by biological evolution. The solutions found by an evolutionary algorithm during its execution are treated as "individuals" in a population. Over the "generations" (i.e., over the course of algorithmic iterations), the population evolves through combinations of attributes of the strongest individuals and through random mutations. The "strength" of an individual is assessed via a fitness function, usually the objective function of the optimization problem to be solved, or some surrogate thereof.

An example of these approaches, called "Gender-based Genetic Algorithm" (GGA) and employing a genetic algorithm (belonging to the class of evolutionary algorithms), is presented in [7, 36]. GGA can be deployed to tune of the discrete and continuous

(although discretized) parameters of any target algorithm. To represent the logical dependencies between the parameters, GGA encodes each configuration by a tree.

In [7], the K-EP is initialized by sampling a set of configurations uniformly at random (sample_0), randomly assigning them a label comp or ncomp ("competitive", "noncompetitive") and evaluating those with label comp on a set of instances Π' ($\mathsf{evaluate}_0$). The performance $p_{\mathcal{A}}$ is chosen to be the algorithmic run time. In subsequent sample_t phases: a) 10% of the comp configurations with the smallest run time, and a smaller percentage (decreasing at each iteration) of randomly chosen ncomp configurations, are selected; b) all the selected ncomp configurations are randomly assigned one of the picked comp ones, and the remaining selected comp configurations are randomly coupled with each other; c) a new generation is produced by *crossover*, i.e., by combining the trees of each couple, and by random perturbations of the tree vertices; d) finally, each new configuration is given a label, determined by the labels of the parents. The configurations of the new generation are sampled, then evaluated ($\mathsf{evaluate}_t$). The update_t phase keeps track of and updates the best performing configuration(s), representative of $\mathscr{P}_{\mathcal{A},1}$, over the generations.

In GGA++, a variant of GGA described in [36], the K-EP is initialized (iteration $t = 0$) by training a random forest regressor (see Sec. 3.6.2). Its purpose is approximate the performance of a configuration, i.e., to learn the function $\chi_{\mathcal{A}}$ defined in Eq. (4.7). At iterations $t > 0$, the K-EP is then carried out as in [7], save for the fact that the trained predictor is used in sample_t, notably: at point b), to pick ncomp configurations by their estimated performance (rather than uniformly at random); at point c), after crossover, to compute the estimated performance of all possible tree combinations and only select the best one.

Another approach implementing evolutionary techniques is "Relevance Estimation and Value Calibration" (REVaC) [147]. The K-EP is launched by sampling candidate configurations uniformly at random (sample_0). At each iteration $t \geq 0$ and for each of the configurations sampled by sample_t, the $\mathsf{evaluate}_t$ phase computes the average performance over Π'. The configuration with the worst performance is eliminated, and replaced by a new one in the next sample_t phase, for $t > 0$. In particular, this new configuration is produced as follows: sample_t chooses the "parents", i.e., the top performing configurations of the previous iteration; then, for each algorithmic parameter, it randomly samples a value from an interval, defined by combining the parameter values of the parents. The update_0 phase saves the configuration c^* with the best performance, while update_t, for $t > 0$, checks whether of c^* can be updated.

4.6.2 Approaches based on local search heuristics

Many PP methodologies rely on LSHs, and metaheuristics other than evolutionary algorithms (presented in Sec. 2.6.1.3 and 2.6.1.4), to build $\mathscr{P}_{\mathcal{A},1}$ in the K-EP.

"OPtimization of ALgorithms" (OPAL) (see [10, 9] and references therein) is an ACP methodology based on mesh-adaptive direct search, an LSH and employed in derivative-free optimization. OPAL allows the user to encode the configuration set $\mathcal{C}_{\mathcal{A}}$ via constraints, and to even enforce additional constraints on the execution of the K-EP (say, to maximize the number of instances solved while ensuring that the solve time for each instance does not exceed a prescribed limit). In the first iteration $t = 0$ of the K-EP, the instances in Π' are solved by the default configuration of $\mathcal{A}$, and partitioned into easy (Π'_{e}) and hard (Π'_{h}) to solve, according to the solution CPU time. We call $p_{\mathcal{A}}(\Pi', c)$ the time that $\mathcal{A}$, configured by c, takes to solve the instances in Π' to a prescribed precision; in OPAL, $p_{\mathcal{A}}(\bar{\Pi}'_{\mathsf{e}}, \cdot)$ is used as a surrogate of $p_{\mathcal{A}}(\bar{\Pi}, \cdot)$. The sample_0 phase selects configurations by DOE. Then, the $\mathsf{evaluate}_0$ phase computes $p_{\mathcal{A}}(\Pi'_{\mathsf{h}}, \cdot)$ of the sampled configurations.

In subsequent iterations, sample_t performs two steps. First, a grid is created from the configurations for which $p_{\mathcal{A}}(\Pi'_{\mathsf{h}}, \cdot)$ has been previously calculated. Then, for all the configurations c on the grid, $p_{\mathcal{A}}(\Pi'_{\mathsf{e}}, c)$ is computed; only the ones with similar performance to the incumbent, i.e., for a user-defined $\varepsilon > 0$, such that

$$|p_{\mathcal{A}}(\Pi'_{\mathsf{e}}, c) - p_{\mathcal{A}}(\Pi'_{\mathsf{e}}, c^*)| \leq \varepsilon \,,$$

are sampled. The $\mathsf{evaluate}_t$ phase computes $p_{\mathcal{A}}(\Pi'_{\mathsf{h}}, \cdot)$ of the sampled configurations. Finally, the update_t phase checks whether the current best configuration of $\mathscr{P}_{\mathcal{A},1}$, i.e., the incumbent c^*, can be replaced with a better one. If c^* cannot be updated, another local search is performed around it: new configurations are sampled from a grid generated around the incumbent, $p_{\mathcal{A}}(\Pi'_{\mathsf{e}}, \cdot)$ is again used as a surrogate of $p_{\mathcal{A}}(\Pi'_{\mathsf{h}}, \cdot)$ to sample promising configurations, and c^* is updated, if necessary.

ParamILS [89] relies on the "iterated local search"metaheuristic (the "ILS" in its name), can be deployed to tune the parameters of any algorithms (although it can only manage discrete/discretized values) and allows the definition of conditional dependencies and feasibility constraints on the parameters. Furthermore, it assumes that two sets of instances are available for the K-EP, that we will refer to as Π' and Π''. In [89], the chosen $p_{\mathcal{A}}$ is the algorithmic run time, computed for each configuration and aggregated over Π'. ILS is based on the *1-neighbourhood* $\mathcal{N}_1(c)$ of a configuration c, i.e., the set of all configurations obtained by randomly modifying only one (generally, randomly picked) parameter of c.

The sample_0 phase is launched either with the default configuration of the target algorithm or with a random configuration, when the default one is unknown. This configuration is evaluated $\mathsf{evaluate}_0$ and stored as the first incumbent of $\mathscr{P}_{\mathcal{A},1}$; we call the incumbent c^*.

In the first iterations after $t = 0$, ParamILS repeats the following tasks until a better incumbent is found: a) it randomly samples a configuration $\bar{c} \in \mathcal{N}_1(c^*)$ (sample_t); b) it computes its average performance over Π' ($\mathsf{evaluate}_t$); c) it updates the incumbent, if the performance of $\bar{c}$ is higher (update_t).

In subsequent iterations, the ILS is performed again, with some modifications. Firstly, sample_t only uses a proper subset of $\mathcal{N}_1(c^*)$, rather than the whole 1-neighborhood. Secondly, each sample_t has a (user-defined) probability of being re-initialized from a random point instead of c^*. Thirdly, if the sampled candidate $\bar{c} \in \mathcal{N}_1(c^*)$ and c^* have been evaluated on the same instances but it is impossible to establish a winner, the pair is evaluated on Π'', one instance at a time, until one clearly dominates the other. The ILS is executed until no single-parameter value change can improve c^*, or until a time limit has elapsed.

A variant of ParamILS is implemented in [45], where the metaheuristic of choice is a VNS (see Sec. 2.7.1.2), based on i 1-neighbourhoods, each centered at a different configurations.

The i-race methodology [138] implements a local search procedure that the authors call "iterated race". In the first iteration of a race, randomly sampled configurations (sample_0) are evaluated, in parallel, on the instances in Π, one at a time ($\mathsf{evaluate}_0$). A statistical test, performed at prescribed intervals during the evaluations, allows the elimination of the configurations with the worst performance, while the others continue the race. This process stops when: a certain number of surviving configurations have been selected, a maximum number of training instances have been used, or a prescribed computational budget has been depleted. The surviving configurations are then selected as incumbent, representative of Π', and ranked by performance (update_0).

In subsequent iterations, sample_t unfolds in three steps: a) a configuration $\bar{c}$ is drawn from the incumbent ones, with probability increasing as the rank gets higher; b) for each parameter, a probability distribution is defined as a function of that parameter's value in $\bar{c}$; c) several new configurations are selected by this distribution, to iterate the race. A sample distribution for point b), when the parameter at hand is continuous, is a truncated normal distribution: its mean is the parameter value in $\bar{c}$; its variance is, initially, the midpoint of the parameter's feasible range, and is decreased at each iteration. After sample_t, the newly picked configurations are raced against the incumbent one(s) in $\mathsf{evaluate}_t$, and the incumbent configurations updated, if necessary, in update_t.

"Sequential Model-based Algorithm Configuration" (SMAC), illustrated in [88], differs from those outlined above in that it combines LSHs with an ML paradigm. Iteration $t = 0$ of the K-EP is dedicated to training an ML predictor $\bar{p}_{\mathcal{A}}$ (Eq. (4.6)); a random forest regressor (discussed in see Sec. 3.6.2) is the choice in [88], although other alternatives are possible. Its predictions are then aggregated over Π': we let $\hat{p}_{\mathcal{A}}(c)$ be the resulting approximation of the CPU time that $\mathcal{A}$, configured by $c \in \mathcal{C}_{\mathcal{A}}$, takes on average to solve instances in Π'. The update_0 phase also selects an incumbent configuration c^*, the one with minimum $\hat{p}_{\mathcal{A}}$, for $\mathscr{P}_{\mathcal{A},1}$.

The following iterations are based on the computation of a metric, $EI(c)$, measuring the "expected improvement" of a configuration c over c^* in terms of $\hat{p}_{\mathcal{A}}$. At each iteration $t > 0$, sample_t first computes EI for all configurations evaluated at $h < t$, then launches an MS (described in Sec. 2.7.1.2), optimizing EI and based on a 1-neighbourhood, at ten configurations with the highest EI. The ten corresponding solutions, together with several additional, randomly chosen configurations, are sampled. The update_t phase uses $\mathsf{evaluate}_t$ as a subroutine: it runs $\mathsf{evaluate}_t$ until a time budget is reached, in order to compare the performance of the sampled configurations with that of the incumbent; moreover, it replaces c^* when a better incumbent is found. We remark that $\mathsf{evaluate}_t$ is called on the on the same subset of Π' both for the incumbent and the other candidates. This process produces new training data, which can be used in the next iterations to re-train the chosen ML predictor.

4.6.3 Approaches based on experimental design

The approach presented in [3], called "CALIBRA", employs DOE techniques — notably, Taguchi's fractional factorial experimental design — for tuning up to 5 discrete/continuous parameters of a given target algorithm. We refer to the article above, and the references therein, for a closer look at the DOE techniques employed by the authors. In the first iteration of the K-EP, the algorithmic performance of 2^5 configurations, sampled by a full factorial design (sample_0) is computed on the instances in Π' ($\mathsf{evaluate}_0$); the top performing one, that we call c^*, is picked as the incumbent, representative of $\mathscr{P}_{\mathcal{A},1}$ (update_0). In subsequent iterations ($t > 0$), nine configurations are sampled in the neighbourhood of c^*, via Taguchi's design techniques (sample_t), and evaluated ($\mathsf{evaluate}_t$). If one of the sampled configurations has better performance than the incumbent, c^* is updated (update_t). The approach can be easily extended to consider more than incumbent for $\mathscr{P}_{\mathcal{A},1}$. The K-EP terminates when a given budget of algorithm runs is exhausted.

4.7 Per-instance approaches

In this section, we examine the approaches to the ACP that we consider most relevant in the PI context. Their $\mathcal{M}$ is mostly ML-based, i.e., it involves the construction of either $\zeta_{\mathcal{A}}$, $\bar{p}_{\mathcal{A}}$ or $\chi_{\mathcal{A}}$ (i.e., Eq. (4.5), Eq. (4.6) or Eq. (4.7), respectively). However, variants of model $\mathscr{P}_{\mathcal{A}}$ (Eq. (4.8)) are also employed, often in combination with ML techniques. The majority of these methodologies are of the offline type. In all of them, a set of instances and, often, even a set of configurations — we call them Π' and $\mathcal{C}'_{\mathcal{A}}$ — both necessary for performing the K-EP, are provided by the user, rather than being chosen in the sample_0 phase.

4.7.1 Approaches learning to predict the optimal configuration

Common to the methodologies in this section are: the construction of an ML-derived predictor $\zeta_{\mathcal{A}}$ (Eq. (4.5)) of the optimal configuration for a given instance; the fact that they are all offline.

The approach proposed in [19] aims to decide which, out of two *scaling* methods, an optimization solver should apply for a given problem instance. Scaling is a technique to reduce the numerical errors committed by a solver, especially in the execution of B&B-type algorithms (see Sec. 2.6.1.1), in order to improve the behaviour of the solver and shorten the solution process. The choice of the most suited scaling technique is represented as a binary classification problem, where the possible outputs $\{0, 1\}$ encode the available options. To perform classification, the K-EP learns an ML regressor, capable of approximating the numerical instability of a solver run, which is encoded by a value in $[0, 1]$. In the recommendation phase, given a new instance of the optimization problem Π, the scaling method to be applied is determined by checking whether the estimate is closer to 0 or 1, by a user-defined threshold. The K-EP usually consists of a single iteration, whereby, given Π', sample_0 and $\mathsf{evaluate}_0$ assemble a training set, and update_0 builds an ML predictor. Computational results with a Random Forest (see Sec. 3.6.2) and a linear regressor (see Sec. 3.6.4) are presented.

The approach described in [25] aims at tuning one parameter of an optimization solver. The parameter in question determines whether or not to linearize the quadratic objective function of a Mixed-Integer Quadratic Program (MIQP), i.e., a QP (see Sec. 2.3) with integer variables and linear constraints. In practice, three parameter values are considered, corresponding to three algorithmic strategies: linearize, do not linearize, leave the choice to the solver. The K-EP, carried out in a single iteration, constructs an ML classifier $\zeta_{\mathcal{A}}$ to predict the optimal parameter configuration, for a given instance. Since Π' is given, and $\mathcal{C}_{\mathcal{A}}$ consists of only three configurations, sample_0 is trivial. The

training labels, for building the classifier in update_0, are determined by running the target solver, with the three configurations, to optimize the instances in Π' ($\mathsf{evaluate}_0$). This approach can be adapted to many ML paradigms: the authors present results with a support vector machine (i.e., a classification-oriented variant of SVR, discussed in Sec. 3.6.4) and other tree-based methods (derived from the base model treated in Sec. 3.6.2).

The authors of [24] present an ASP approach, capable of picking the best algorithm for solving a given instance of an optimization problem Π. This is achieved by learning a decision tree classifier, notably, a model $\zeta_{\mathcal{A}}$ based on the DT paradigm illustrated in Sec. 3.6.2. In the K-EP, the corresponding training problem is cast as an MP and solved heuristically, by running an VNS (see Sec. 2.7.1.2). The trained predictor is such that each tree leaf is assigned a) a partition of the training instances and b) the optimal algorithm for solving them. In the recommendation phase, given a new instance, the trained predictor can directly recommend the most appropriate algorithm for solving it.

4.7.2 Approaches learning to approximate algorithmic performances

The PI approaches described in this section are all of the offline type and involve the construction of an ML-derived predictor $\bar{p}_{\mathcal{A}}$ of the performance function $p_{\mathcal{A}}$ (i.e., model $\mathcal{M}$ in Eq. (4.6)). Thus, their K-EP is dedicated to solving the training problem, which requires building a training set (sample_0, $\mathsf{evaluate}_0$) and training the chosen predictor (update_0).

The K-EP in the methodology presented in [17] is used to learn a random forest regressor (see Sec. 3.6.2), capable of approximating the algorithmic run time of a couple (instance, configuration). Given an instance of Π, the recommendation phase runs an exhaustive search, which selects the best algorithmic configuration for solving that instance by computing the trained predictor's approximation at all points of $\mathcal{C}'_{\mathcal{A}}$.

In [86], the performance function to be approximated is the same as that chosen in [17], but different ML paradigms are deployed to obtain an approximation $\bar{p}_{\mathcal{A}}$, i.e., linear basis function regression and linear bayesian regression. The recommendation phase also executes brute-force search to compute the recommender. However, the authors argue that, since scaling this method to large $\mathcal{C}_{\mathcal{A}}$ can be very challenging, more sophisticated strategies may need to be considered, such as gradient descent for continuous parameters. However, no computational experiments are provided for assessing the suitability such strategies.

The approach proposed in [23] is intended for the configuration of optimization solvers. The performance $p_{\mathcal{A}}$ is a function of different statistics concerning the solution process, i.e.: a) the time necessary to attain optimality, b) the sequence of solver integrality gaps, measured at each gap improvement, c) the time between two consecutive improvements of the sequence. A diverse set of ML regressors (locally weighted learning [8], SVR (see Sec. 3.6.4) and random committee [194]), built in the K-EP to estimate $p_{\mathcal{A}}$, are tested. Upon the arrival of a new instance, the configuration recommended in the recommendation phase is found by computing $\bar{p}_{\mathcal{A}}$ for all possible candidates.

4.7.3 Approaches building $\mathscr{P}_{\mathcal{A},|\Pi'|}$

Approaches of this type focus on the construction of a variant of model $\mathscr{P}_{\mathcal{A}}$ in Eq. (4.8), whereby Π' is partitioned into as many elements as its cardinality (see Sec. 4.3).

In the offline methodology presented in [156] each element of the partition $\mathscr{P}_{\mathcal{A},|\Pi'|}$ (i.e., each instance in Π') is specified by a Bayesian Network (BN), and all BNs are learned in one iteration of the K-EP. A BN is a probabilistic graphical model, notably, an ML approximation of the kind in Eq. (4.7), capable of assigning a high probability to configurations which are most likely to provide good performance. Its construction, taking place in update_0, requires a training set (built by sample_0 and $\mathsf{evaluate}_0$). In the recommendation phase, a configuration for a given instance of $\bar{\pi} \in \Pi$ is recommended by first finding the closest or "most similar" instance in $\mathscr{P}_{\mathcal{A},|\Pi'|}$ — according to some distance function — and then computing the prediction by the associated BN. If the instances in $\mathscr{P}_{\mathcal{A},|\Pi'|}$ are too different from $\bar{\pi}$, a new BN is learned for $\bar{\pi}$, and the pair is added to $\mathscr{P}_{\mathcal{A},|\Pi'|}$. The saved BNs are periodically re-trained, as new training data is gathered.

4.7.4 Approaches building $\mathscr{P}_{\mathcal{A},C}$

The approaches examined in this section, i.e., those presented in [40, 96] (both ASP methodologies) and [101], are all offline. Their K-EP implements the construction of $\mathscr{P}_{\mathcal{A},C}$, i.e., a variant of the model $\mathscr{P}_{\mathcal{A}}$ in Eq. (4.8), where the instances in Π' are grouped into several clusters, such that the elements of each cluster share a similar structure. In all the surveyed approaches, the K-EP begins with the execution of a procedure for clustering the instances in Π' (iteration $t = 0$). This is followed by a second phase which, for each cluster, selects a configuration considered optimal for its instances.

The adopted clustering algorithms are: the k-means (see [40, 96] and references therein), which requires the user to input the number k of cluster to create; the g-means ([101], a variation of the k-means that automatically finds k). Π' is clustered based on the instance features [40, 101], or on a combination of instance features and

metrics about the target algorithm performance on single instances [96].

The approaches differ according to the method used to select the representative configuration of each cluster. In [40], this is accomplished by a brute-force search: the performance of all candidates in $\mathcal{C}'_{\mathcal{A}}$ is computed for each cluster, and the best one is chosen. This strategy is generally computationally infeasible when $\mathcal{C}'_{\mathcal{A}}$ is large, but it works in this case, because [40] is an ASP methodology and $\mathcal{C}'_{\mathcal{A}}$ is a small set of candidate algorithms. Instead, in [96] (an ASP approach), Π' is clustered according to the performance of candidate target algorithms on single instances: for each cluster, the k-means automatically chooses the most efficient algorithm for solving the instances in that cluster. Moreover, in the K-EP of [96], a classifier is trained to assign an instance to a cluster. However, in the context of algorithm configuration, where $\mathcal{C}_{\mathcal{A}}$ is often much larger than in the ASP, more sophisticated procedures are necessary. For instance, the approach illustrated in [101] first performs clustering on Π' and then runs GGA (an ACP methodology of the PP type, discussed in Sec. 4.6.1) on each cluster.

The recommendation phase in [40] involves running, upon the arrival of a new instance, a k-nearest neighbour algorithm [22, §] to assign it a cluster. Instead, in [96], a classifier learned in the K-EP is used to assigned a cluster label to the new instance. Finally, in [101], the membership of the new instance to one of the clusters is decided according to some distance function, computed for all the cluster centroids.

We note that the approach in [96] also differs from the others in the implementation of a meta-sampling loop, which feeds the recommendation phase data (new instances and recommended configurations) into the K-EP.

4.7.5 Approaches building $\mathscr{P}_{\mathcal{A},1}$

The K-EP of the methodologies discussed in this section deals with the construction of $\mathscr{P}_{\mathcal{A},1}$, i.e., a variant of the model in Eq. (4.8), in which the instances in Π' are not partitioned and only one configuration is selected as representative of the whole Π'.

The online, PI approach discussed in [14], called "reactive search", is devised for calibrating a single parameter of a tabu-search algorithm, a metaheuristic for optimization problems (see Sec. 2.6.1.4). In reactive search, the K-EP, used to construct $\mathscr{P}_{\mathcal{A},1}$, is iterated each time a new instance $\bar{\pi} \in \Pi$ is to be solved. At iteration t, sample_t selects a value of the parameter to be configured; at $t = 0$, the choice is between a random value or the default one. Then, the tabu-search, configured with the selected parameter value, is run for a prescribed CPU time or number of iterations, to solve $\bar{\pi}$ ($\mathsf{evaluate}_t$); the corresponding solution is stored in a database containing all solutions. In the subsequent update_t phase, the database is inspected, and the parameter increased or decreased if the current solution appears too often or not too often, compared to a threshold provided by the user. The updated parameter value will be used at the next iteration.

A different approach, described in [137] and called "Auto-sklearn", relies on PP techniques — in particular, it is based on the SMAC methodology, outlined in Sec. 4.6.2 — and combines both offline and online procedures. Auto-sklearn is designed to recommend the best ML strategy to learn from a given a dataset. A strategy, that the authors refer to as "framework", comprises the choice of data pre-processing techiniques, feature creation and prediction algorithms (including the accompanying hyperparameters). The purpose of Auto-sklearn is to automatically select the framework minimizing the generalization error on the given dataset. In a first, offline phase, SMAC is run several times on different training datasets, to pick the best framework for each of them. The training datasets and the associated recommended framework, each encoded by a vector of continuous/discrete element, are then stored. The online phase is executed when a new dataset D arrives: Auto-sklearn first finds, among the datasets saved in the offline phase, k that are most similar to D (according to some distance function and threshold provided by the user), and then retrieves the associated frameworks. Subsequently, these frameworks are used to initialise SMAC, which uses the data in D to find the optimal framework for it.

In the offline approach presented in [195, 197] and called "Hydra", the K-EP output consists of two models $\mathcal{M}$, both contributing to the structure of the recommender. One model is $\mathscr{P}_{\mathcal{A},1}$ (a variant of model $\mathscr{P}_{\mathcal{A}}$ in Eq. (4.8)): although $\mathscr{P}_{\mathcal{A},1}$ is usually specified by a single configuration, representative of the instances in Π', Hydra saves a portfolio of promising configurations. The other model is a variant of $\bar{p}_{\mathcal{A}}$ in Eq. (4.7), for which here we use the shorthand $\hat{p}_{\mathcal{A},c}(\pi)$. In Hydra, $\hat{p}_{\mathcal{A},c}(\pi)$ is ML-derived and it approximates the performance of the target algorithm $\mathcal{A}$, configured by c and run to solve an instance $\pi \in \Pi$ (see [196] for details). We remark that Hydra learns a different predictor for each algorithmic configuration added to the portfolio, i.e., $\hat{p}_{\mathcal{A},c}$ is a function of π and is parametrized by c.

Before the K-EP starts, Π' is partitioned into training and test instances, that we refer to as Π'_{tr} and Π'_{te}. At each iteration of the K-EP, the following tasks are performed:

1. any PP methodology, applied to Π'_{tr}, selects a configuration c^* to be added to the portfolio accompanying $\mathscr{P}_{\mathcal{A},1}$. In iteration $t = 0$, this is achieved by using $p_{\mathcal{A}}$ as the performance function. In subsequent iterations, the PP methodology uses the improvement over the average portfolio performance as the performance function, to pick the configuration achieving the largest performance improvement;

2. $\hat{p}_{\mathcal{A},c^*}$ is learned using Π'_{tr};

3. the estimated performance of c^* by $\hat{p}_{\mathcal{A},c^*}$ is computed on Π'_{te}. Configurations with poor estimated performance are removed from the portfolio.

In [195, 197], ParamILS (Sec. 4.6.2) is the procedure chosen at step 1. In [195], step 2 is based on the ridge regression; instead, in [197], an ensemble of decision forests is deployed, which is capable of predicting the ranking of c^* in the portfolio. Furthermore, in [197] the K-EP is modified so that, at each iteration, multiple configurations are added to the portfolio: the chosen PP approach is run several times, in parallel, and a subset of promising configurations is selected from the solutions found.

Given a new instance $\bar{\pi} \in \Pi$, the recommendation phase selects, from the portfolio assembled in the K-EP, the configuration with the best estimated performance on $\bar{\pi}$, according to the learned approximations $\hat{p}_{\mathcal{A},c}$.

4.8 Our take on the algorithm configuration problem

The contribution of this thesis to the ACP is the design and implementation of two novel offline, PI solution methodologies.

In the first one (published in [93] and illustrated in detail in Ch. 5), the model $\mathcal{M}$ built in the K-EP is $\bar{p}_{\mathcal{A}}$ of Eq. (4.6). Since $\bar{p}_{\mathcal{A}}$ is an approximation of the performance function, we call this approach *Performance-as-Output* (PaO). The recommender, for computing the ACP solution, is formulated as an MP, containing the equations of the inference problem associated to $\bar{p}_{\mathcal{A}}$ (which describe the learned prediction into MP terms): its objective optimizes $\bar{p}_{\mathcal{A}}$ over the known set $\mathcal{C}_{\mathcal{A}}$ of feasible configurations; its constraints encode $\mathcal{C}_{\mathcal{A}}$.

Our methodology is unlike the other PI ones (discussed in Sec. 4.7) and, specifically, different from the ones centering on the model in Eq. (4.6) (considered in Sec. 4.7.2). In fact, we exploit the mathematical structure of the trained ML predictor to formulate the ACP as an MP. In this sense, the novelty of the formulation we propose is that it is white-box, as opposed to the other black-box approaches, prominent in the literature. For this reason, we can solve the ACP with all the available sophisticated machinery, and even off-the-shelf optimization tools, hopefully scaling more efficiently to very large configuration sets than black-box optimization algorithms.

In the second ACP methodology (published in [94] and described in Ch. 6), the model $\mathcal{M}$ to construct is a variant of $\zeta_{\mathcal{A}}$ in Eq. (4.5), i.e., an approximation $\bar{\mathfrak{c}}$ of the function

$$\mathfrak{c} : \Pi \times \mathbb{R} \longrightarrow \mathcal{C}_{\mathcal{A}}\,, \quad \mathfrak{c}(\pi, r) \mapsto c\,. \tag{4.14}$$

In Eq. (4.14), $r \in \mathbb{R}$ is a required algorithmic performance level, and $c \in \mathcal{C}_{\mathcal{A}}$ a configuration allowing $\mathcal{A}$ to attain a performance r, when it is run on π. We call this approach *Performance-as-Input* (PaI), to distinguish it from the PaO one, and because r is an

input for the ML predictor. Similarly to the PaO methodology, we translate the predictor $\bar{\mathfrak{c}}$ into MP terms, and optimize the resulting formulation, to recommend a suitable algorithmic configuration for solving a given instance.

We note that the PaI approach is in contrast to the PI methodologies outlined in Sec. 4.7 — in particular, to the ones in Sec. 4.7.1 — where $\mathcal{M}$ directly returns the optimal configuration for a given instance, by means of an ML predictor.

Finally, we observe that both of our methodologies are able to operate with any ML paradigm, as long as one is capable of describing the corresponding inference problem by means of the MP language. In this sense, the approaches we propose are "MP-driven", and parameterized by ML predictors. Moreover, both approaches can be deployed to tune the (discrete/discretized) parameters of any target algorithm, and thus have wide applicability.

4.9 Conclusions

In this chapter, we introduced an algorithmic schema, common to all ACP methodologies, for constructing and deploying a recommender, i.e., a function capable of suggesting the optimal configuration of a given algorithm $\mathcal{A}$ for solving an instance of a given decision/optimization problem Π. Furthermore, we placed the most prominent works of the ACP literature in our schema.

Despite all the research efforts in the field of algorithm configuration, the problem still remains extremely difficult to solve. In fact, it is usually impossible to know the algorithmic performance $p_{\mathcal{A}}$ over all of the many parameter configurations in $\mathcal{C}_{\mathcal{A}}$, which can be up to the hundreds or thousands (especially in general-purpose optimization solvers, equipped with a diverse set of algorithmic components to solve famously **NP**-hard problems [102, 103]). Furthermore, while the ACP can be somewhat approached if we consider a single instance or a small subset of Π', it becomes intractable when we look at the whole set Π, which is normally of infinite size.

To overcome this complication, ACP methodologies are all based on multiple forms of approximation of: a) the algorithmic performance function, or of the parameter configuration allowing to achieve specific algorithmic performances, via some computable ML approximation; b) Π, via the manual selection of a subset of representative instances and corresponding representative configurations. The methodologies we developed during my PhD are among those based on approximation type a), and relying on ML. However, they differ from the others in that they propose a white-box approach to the problem of formulating a trained ML predictor; this allows us to add, to the arsenal of techniques for dealing with the ACP, those proper to MP.

Chapter 5

Tuning algorithms: optimizing over learned predictions

5.1 Introduction

In this chapter, we discuss the PaO approach, mentioned in Sec. 4.8 of the previous chapter: an offline, PI, MP-driven, ACP methodology, parametrized by an ML paradigm, for finding instance-wise optimal parameter configurations for general MP solvers. The origin of its name ("performance-as-output") lies in the fact that the model $\mathcal{M}$ built in the K-EP is an (ML) approximation of the performance function $p_{\mathcal{A}}$. In Sec. 5.5, we report the results of the computational experiments that we carried out to test the proposed approach. A part of this chapter has been published in [93].

5.2 Motivation

Once a problem is modelled by an MP formulation, an off-the-shelf solver can be used to solve it. Off-the-shelf solvers are highly complex pieces of software, which must be general enough to encompass a significant family of problems, and yet fast enough that sufficiently large-scale instances will be solved in reasonable time. By the usual trade-off between generality and efficiency, implementing a good solver is extremely hard. Today's most successful solvers meet these specifications by actually embodying a corpus of different algorithmic components (pre-processing techniques, relaxations, heuristics, cutting planes, branching algorithms, etc., which are treated in Ch. 2), for addressing the different phases of the solution process. Each solution algorithm has its own (often very large) set of algorithmic options, and the available algorithmic choices are exposed to the user as a long list of parameters. Their default values are chosen so that the solver will operate reasonably well on a sizable library of instances. Furthermore, some commercial solvers provide automatic configuration procedures called "tuning tools"

(e.g., [91, Ch. 10]). These procedures can run the solver, with different configurations, on one or more instances within a given time limit, and record the best parameter values encountered.

Despite all this, default/tuning-tool configurations may still be highly suboptimal on specific instances. Therefore, for many problem classes, carefully adjusting the solver configuration is necessary (see, e.g., [87]) to improve the quality of the obtained solution and/or the efficiency of the solution process. Identifying good solver parameters for a given problem instance, however, is a difficult art, which requires considerable experience in solver usage, and in-depth knowledge of the application giving rise to the considered MP formulation. Further, it can be an extremely time-consuming task, due to the large amount of available parameters, whose number is sometimes in the hundreds (see, e.g., [92]). Therefore, it is of significant interest to develop general, PI approaches, capable of finding the optimal algorithmic parameters for single instances of a decision or optimization problem in an automatic way.

5.3 The Performance-as-Output approach to solver configuration

In this chapter, we will use the same notation and assumptions as in Ch. 4, notably, Sec. 4.1. Thus, we will consider a parametrized algorithm $\mathcal{A}$ (here, an optimization solver, although the PaO methodology is suitable for any algorithm) for solving instances of a problem Π, and choose a parameter configuration from an admissible set $\mathcal{C}_\mathcal{A}$, to optimize the performance $p_\mathcal{A}$ of $\mathcal{A}$ on a given problem instance. Further, we will rely on the following assumptions:

- each configuration $c \in \mathcal{C}_\mathcal{A}$ can be encoded by a vector of q binary values, representing categorical and numerical parameters (although extension to integer and continuous numerical parameters is clearly possible). Since every subset of the unitary hypercube can be described by means of a polytope [157, Cor. 1], we assume that a representation of $\mathcal{C}_\mathcal{A}$ as a bounded set of linear inequalities in binary variables, i.e.,
$$\mathcal{C}_\mathcal{A} = \{c \in \{0,1\}^q \mid Ac \leq \beta\}, \tag{5.1}$$
is known. In practice, deriving A and β from the logical conditions on the parameters can be assumed to be easy;

- a finite subset $\Pi' \subset \Pi$ is available. ML methodologies are known to perform well on training sets that are not "overly general" [73, Ch. 5.3]. Therefore, we select Π' as a set of instances belonging to the same problem, or at least to different variants of a single problem. In practice, we focus on a unit commitment problem arising in the energy industry, that we describe in Sec. 5.5.1;

- each instance $\pi \in \Pi$ can be encoded by a vector of m continuous components, i.e., $\pi \in \mathbb{R}^m$;

- the performance function that we want to learn by ML is defined as

$$p_{\mathcal{A}} : \Pi \times \mathcal{C}_{\mathcal{A}} \longrightarrow \mathbb{R}\,,$$

  mapping a couple (π, c) into the integrality gap reported by the solver after a fixed time limit. This definition takes into account the fact that the best configuration may vary among instances belonging to the same problem, which makes our approach PI.

  To build the training set for our approach, we compute $p_{\mathcal{A}}$ for all the instances in Π' and all the configurations in the considered $\mathcal{C}_{\mathcal{A}}$. We effect this by calling the solver, configured by each $c \in \mathcal{C}_{\mathcal{A}}$, on every instance $\pi \in \Pi'$. Hopefully, then, the training set thus built can be used to generalize our approach to instances outside Π', with known encoding and which are in some way similar (in size or otherwise) to those in Π'.

With the above assumptions, we describe our approach for addressing the ACP on MP solvers, and fit it into the algorithmic schema illustrated in Sec. 4.1–4.5, notably, by discussing the two phases K-EP (where we sample $p_{\mathcal{A}}$ and construct a model $\bar{p}_{\mathcal{A}}$ for it) and recommendation phase (where we build a recommender for a given instance π).

5.3.1 The construction of $\mathcal{M}$

Since $p_{\mathcal{A}}$ is usually a black-box function, the K-EP for building $\mathcal{M}$ relies on a supervised ML paradigm to learn an approximation

$$\bar{p}_{\mathcal{A}} : \mathbb{R}^m \times \mathcal{C}_{\mathcal{A}} \longrightarrow \mathbb{R} \tag{5.2}$$

of $p_{\mathcal{A}}$, from the training set

$$\mathcal{S} = \left\{ \left(\pi_i, c_i, p_{\mathcal{A}}(\pi_i, c_i)\right) \mid \pi_i \in \mathbb{R}^m,\ c_i \in \mathcal{C}_{\mathcal{A}},\ p_{\mathcal{A}} \in \mathbb{R} \right\}.$$

Eq. (5.2) is the model $\mathcal{M}$ in Eq. (4.6), so the considerations we made in Ch. 4 about the construction of this predictor — in particular, how the different phases sample_t, $\mathsf{evaluate}_t$ and update_t are implemented — also apply here. A training input vector (π_i, c_i) is the concatenation of an instance encoding and a configuration encoding; the training set labels $p_{\mathcal{A}}(\pi_i, c_i)$ are computed on the corresponding input tuples. Following the notation of Eq. (3.1) in Ch. 3, we use the shorthand $s = |\mathcal{S}|$ and $S = \{1, 2, \ldots, s\}$.

5.3.2 The recommender

The recommendation phase, in which we compute the recommender for a given instance in Π, amounts to solving the the Configuration Set Search Problem (CSSP).

After learning $\bar{p}_{\mathcal{A}}$, the formal structure underlying the trained predictor is translated into MP terms. In Sec. 3.3, we define this formulation the "inference problem" of a learned ML predictor. By adding constraints encoding the compatibility/dependency conditions on the configuration parameter values, we obtain the the CSSP, i.e.:

$$\mathsf{CSSP}(\bar{\pi}) = \arg\min_{c \in \mathcal{C}_{\mathcal{A}}} \bar{p}_{\mathcal{A}}\big(\bar{\pi}, c\big)\,, \tag{5.3}$$

for a given instance $\bar{\pi}$ and a given predictor $\bar{p}_{\mathcal{A}}$. We solve Eq. (5.3), upon the arrival of $\bar{\pi}$, to find the configuration c^* providing optimal algorithmic performance with respect to the approximated performance function $\bar{p}_{\mathcal{A}}$. Hence, in terms of the entities introduced in Ch. 4, $\mathsf{CSSP}(\bar{\pi})$ is our recommender $\Psi_{\mathcal{M}}(\bar{\pi})$. If $\bar{p}_{\mathcal{A}}$ yields an accurate estimate of $p_{\mathcal{A}}$, we expect the optimum c^* of Eq. (5.3) to be a good approximation of the true optimal configuration for solving $\bar{\pi}$.

We remark that the actual implementation of Eq. (5.3) depends on the MP encoding selected for the trained predictor, and it may include additional variables and/or constraints; see Sec. 3.6.1–3.6.4 for some examples.

The main novelty of our methodology is the fact that learned model $\mathcal{M}$ is white-box: we exploit the mathematical description of $\bar{p}_{\mathcal{A}}$ to formulate and optimize the CSSP for specific instances. This conveniently allows us to treat the ACP as an MP, by contrast with most of the existing ACP approaches, which consider the $\bar{p}_{\mathcal{A}}$ as a black-box and, hence, cannot exploit its structure. These approaches can only employ $\bar{p}_{\mathcal{A}}$, as an oracle, in LSHs or metaheuristics in the K-EP, or to compute a prediction at all candidate configurations in the recommendation phase [88, 36, 17, 86, 23, 137, 195, 197] (see Sec. 4.6–4.7). Thus, they cannot reasonably hope to find a global minimum when the number of parameters grows. Moreover, they may not scale well to settings where not only $\mathcal{C}_{\mathcal{A}}$ is large but c^* depends on the instance at hand. Further, formulating the ACP as an MP is advantageous, as it allows the seamless integration of the constraints on the configuration parameters, which is something that few other ACP methods allow. The only exceptions that we are aware of are the approaches described in [89, 7] — where, according to the authors, users can enforce constraints on feasible parameter configurations — and in [138] — where the user can forbid the sampling of specific configurations. However, these approaches are of the PP type; therefore, when the behaviour of the target algorithm varies considerably with the structure of an instance of Π, they may fail to recommend good configurations for specific instances.

A unique asset of our methodology is that it is not limited to a specific supervised ML paradigm. The fact that it can be adapted to operate with any predictor — as long as one is capable of formulating the accompanying inference problem — offers users

freedom to trial many approximation structures and pick the one providing the highest generalization. Of course, extremely accurate predictors may be also too complex for the resulting CSSP to be solved quickly. This would defeat the purpose of deploying our methodology, since, in these circumstances, one might as well solve $\bar{\pi}$ directly. Hence, incurring the additional computational overhead for solving the CSSP may be advantageous only when the instance at hand is "hard". All in all, achieving a balance between the K-EP predictor's complexity and the CSSP cost is by no means trivial, which makes the practical implementation of our PaO approach quite challenging. However, we will see in Sec. 5.3.2 that, at least for the tested ML paradigms, the associated CSSP is solved in a matter of seconds.

We remark that, although we perform the K-EP only once, offline, the approach can be easily fitted to a "dynamic" setting, in which the data obtained from solving the CSSP is iteratively used to re-train the chosen ML predictor.

Finally, we should point out that the idea of learning the unknown components (constraints, objective) of an MP has been already explored. In general, it is possible to cast the ML approximation of a mapping/relation as an MP, and optimize upon this (see, e.g., [134], [142], [62] or [186] and references therein). However, while this is in principle possible, the set of successful applications is restricted in practice. Indeed, using this approach in the ACP context is, to the best of our knowledge, novel. It also comes with some specific twists:

- different models $\mathcal{M}$ yield different CSSPs, which require ad-hoc implementations and specific solution tools;

- the performance of an optimization solver is usually expressed in terms of the gap reached within a time limit, or the time required to reach optimality. In both cases, $p_{\mathcal{A}}$ can range from very small to quite large values, which often requires designated processing procedures (e.g., outlier elimination, normalization, etc.), for the K-EP training to be carried out efficiently. We call p_{ml} the processed performance function, i.e., the one actually estimated by $\bar{p}_{\mathcal{A}}$. Since the CSSP optimizes a proxy of p_{ml} (rather than a proxy of $p_{\mathcal{A}}$) these processing procedures could affect the success of c^* in configuring $\mathcal{A}$;

- the chosen ML loss metric determines how prediction errors are penalized. This is crucial to achieve a correspondence between the set of global optima of $\bar{p}_{\mathcal{A}}$ (where the CSSP solution lies) and the set of global optima of $p_{\mathcal{A}}$ (containing the true optimal solution of the ACP);

- since dimensionality reduction techniques can help to decrease the ML predictor error, we tried several FS algorithms (see Sec. 3.5.4.1) on our training set. FS, acting on inputs (π, c) of dimension $m+q$, can select a subvector (π', c') of dimen-

sion $m' + q'$. However, FS also has an impact on the CSSP. In fact, if $q' < q$, the solution $c^* \in \{0,1\}^{q'}$ of the CSSP will have to be completed, by assigning values to the $q - q'$ left-out configuration variables. Addressing the use of dimensionality reduction techniques other than FS is left for future research.

In the next sections, we will discuss the issues above in more depth.

We tested our idea with the following components: we configured nine parameters of the IBM ILOG CPLEX solver version 12.7, which we employed to solve 250 instances of a hard MILP, the Hydro Unit Commitment (HUC) problem [26]. The experiments were conducted on an Intel Xeon CPU E5-2620 v4 @ 2.10GHz architecture, while CPLEX was run on Intel Xeon Gold 5118 CPU @ 2.30GHz. We compared the performances of linear and nonlinear SVR, and DTs, as the ML paradigms of choice. We used the open-source, MINLP solver Bonmin [153] to solve the CSSP for the nonlinear SVR, and CPLEX for the linear SVR and DTs. The pipeline was implemented in Python 3.6.8 [163] and AMPL version 20200110 [66]. In the following, we detail the algorithmic set-up that we employed. In Ch. 6, we will discuss several CSSP formulations for the PaO and PaI approaches, when the selected ML paradigm is LR; we will also describe computational experiments carried out on a smaller dataset than the one we used in this chapter. Further, in Ch. 8 we will show a CSSP-PaO formulation for a generic NN.

5.4 Implementation with different ML paradigms

In this section, we present the CSSP formulations arising from the use of SVR and DTs. The optimal training coefficients of each predictor, learned by training, are marked with an asterix. The CSSP is optimized to solve the ACP for a given instance, which we call $\bar{\pi}$. We also present the results of the computational experiments we conducted to test the PaO approach.

5.4.1 The approach with SVR

The linear SVR form yields the following MILP:

$$\mathsf{CSSP}(\bar{\pi}) \quad \equiv \quad \arg\min_{c \in \mathcal{C}_{\mathcal{A}}} \langle w^*, (\bar{\pi}, c) \rangle \, . \tag{5.4}$$

The training problem associated with Eq. (5.4) only requires solving the primal SVR formulation (stated in Sec. 3.6.4.1), a convex QP, for which many efficient algorithms exist (see Sec. 2.3). We remark that the solution returned by this CSSP is always the same for any instance, which makes it PP, rather than PI. In fact, let $w^* = (w^*_\pi, w^*_c)$, where w^*_π and w^*_c are the subvectors referring to the instance components and the configuration components, respectively. Then, carrying out the dot product in Eq. (5.4), we have:

$$\arg\min_{c \in \mathcal{C}_{\mathcal{A}}} \langle w^*, (\bar{\pi}, c) \rangle \quad = \quad \arg\min_{c \in \mathcal{C}_{\mathcal{A}}} \langle w_c, c \rangle \, .$$

We decided to test this CSSP anyway, to benchmark its performance against the PI variants of the PaO methodology, on instances of the same problem Π.

The nonlinear SVR form relies on the kernel trick (see Sec. 3.6.4.3). For our experiments, we choose a Gaussian kernel, which is the default choice in absence of any other meaningful prior [44]. This makes the CSSP a MINLP with a nonconvex objective function:

$$\mathsf{CSSP}(\bar{\pi}) \quad \equiv \quad \arg\min_{c \in \mathcal{C}_{\mathcal{A}}} \sum_{i \in S} \alpha_i^* e^{-\gamma^* \|(\pi_i, c_i) - (\bar{\pi}, c)\|_2^2} . \tag{5.5}$$

5.4.2 The approach with DTs

We use the notation of Sec. 3.6.2 to describe the CSSP with a DT. Since, in our setting, the training inputs are vectors (π, c), the intermediate vertices V_I^* of the learned tree

$$T^* = (V_I^* \cup V_L^*, A^*)$$

are partitioned as

$$V_I^* = \{V_\pi^*, V_{\mathcal{C}_{\mathcal{A}}}^*\},$$

where V_π^* are the vertices branching on instance-related components and $V_{\mathcal{C}_{\mathcal{A}}}^*$ the ones branching on configuration-related components. To formulate the CSSP for an instance $\bar{\pi}$, we first retrieve a graph from T^*, that we call $T_{\bar{\pi}}^{**} = (V_{\mathcal{C}_{\mathcal{A}}}^{**} \cup V_L^{**}, A^{**})$, whereby $V_{\mathcal{C}_{\mathcal{A}}}^{**} \subseteq V_{\mathcal{C}_{\mathcal{A}}}^*$ and $V_L^{**} \subseteq V_L^*$.

In particular, $V_{\mathcal{C}_{\mathcal{A}}}^{**}$ and A^{**} are built as follows:

1. we consider a copy $T^1 = (V_\pi^1 \cup V_{\mathcal{C}_{\mathcal{A}}}^1 \cup V_L^1, A^1)$ of T^*;

2. since $\bar{\pi}$ is known, we can evaluate the branching constraints at each $u \in V_\pi^1$, and determine which of its two children violates them: we call u_{lx} and u_{rx} the two children of u, and let $v \in \{u_{\mathsf{lx}}, u_{\mathsf{rx}}\}$ be the (only) infeasible child; then, we delete arc (u, v) from A^1, as well as the subtree of T^1 rooted in v. This yields a subgraph of T^1, that we call $T^2 = (V_\pi^2 \cup V_{\mathcal{C}_{\mathcal{A}}}^2 \cup V_L^2, A^2)$;

3. initially, $A^{**} = \emptyset$.

   Firstly, we populate it with all the arcs $(u, u') \in A^2$ such that $u, u' \in V_{\mathcal{C}_{\mathcal{A}}}^2$.

   Secondly, for each $u \in V_\pi^2$, we consider its "predecessor" and its "successors". Given the directed path $P(\mathsf{root}, u)$ in T^2 between the root and u, we define the *predecessor* of u as the vertex $u_{\mathsf{p}} \in P(\mathsf{root}, u)$ such that: a) $u_{\mathsf{p}} \in V_{\mathcal{C}_{\mathcal{A}}}^2$, b) $u_{\mathsf{p}} \neq u$ and c) u_{p} is closest to u on the path. The *successors* are found by recursively visiting each of the subtrees of T^2 rooted in u's children u_{lx} and u_{rx}: if the root of the visited subtree belongs to $V_{\mathcal{C}_{\mathcal{A}}}^2 \cup V_L^2$, then it is selected as a successor; otherwise (i.e., if the root belongs to V_π^2), the two subtrees rooted in its children are recursively explored. We remark that there can be at most four successors for

each u. After this, for all the successors, we add an arc (predecessor, successor) to A^2, and delete u.

We call the resulting tree $T^{**}_{\bar{\pi}}$;

4. finally, we retrieve the components of $\rho^*, \tau^*, \lambda^*$ related to the vertices of T^{**}.

The CSSP on T^{**} is the following ILP:

$$\arg\min_{c \in \mathcal{C}_\mathcal{A}} \quad \sum_{u \in V^1_L} y_{u_\mathsf{p},u} \lambda^*_u \tag{5.6}$$

$$y_{\mathsf{root_p},\mathsf{root}} = 1 \tag{5.7}$$

$$u \in V^{**}_{\mathcal{C}_\mathcal{A}} \quad y_{u_\mathsf{p},u} = y_{u,u_\mathsf{lx}} + y_{u,u_\mathsf{rx}} \tag{5.8}$$

$$u \in V^{**}_{\mathcal{C}_\mathcal{A}} \quad y_{u_\mathsf{p},u} - c[\rho^*_u] \leq y_{u,u_\mathsf{lx}} \tag{5.9}$$

$$u \in V^{**}_{\mathcal{C}_\mathcal{A}} \quad y_{u_\mathsf{p},u} + c[\rho^*_u] - 1 \leq y_{u,u_\mathsf{rx}} \tag{5.10}$$

$$u \in V^{**}_{\mathcal{C}_\mathcal{A}} \quad y_{u_\mathsf{p},u} \in \{0,1\}\,. \tag{5.11}$$

For each $u \in V^{**}_{\mathcal{C}_\mathcal{A}}$: we refer to its father as u_p, to its children as u_lx and u_rx, and to the corresponding path variable as $y_{u_\mathsf{p},u}$. Further, the variables $c[\rho^*_u]$ — i.e., the branching variables at the intermediate vertices of $T^{**}_{\bar{\pi}}$ — represent the CSSP choices about algorithmic configuration components. The CSSP solution also identifies a unique path in $T^{**}_{\bar{\pi}}$, which goes from root to one of the leaves V^{**}_L. We call this path $\mathcal{P}$ and the finish leaf u^*. The predicted performance $\lambda^*_{u^*}$ at u^* must be as small as possible (Eq. (5.6)). Eq. (5.7) guarantees that $\mathcal{P}$ is unique. The constraints in Eq. (5.8) make sure that only intermediate vertices u receiving flow (i.e., vertices such that $y_{u_\mathsf{p},u} = 1$) can dispatch it and belong to $\mathcal{P}$. Eq. (5.9) and (5.10) enforce the branching constraints: if $c[\rho^*_u]$ is set to 0, then the path continues to the left child; otherwise, it continues to the right child. The previous constraints also guarantee that only the flow variables corresponding to the active arcs in $\mathcal{P}$ are set to 1.

Clearly, $T^{**}_{\bar{\pi}}$ is usually much smaller than T^*. Due to its size and the fact that it is an ILP, the CSSP in Eq. (5.6)–(5.11) can be solved extremely quickly, as we will see below.

We remark that, due to the action of ρ^* on the configuration components and to the processing steps implemented to construct $T^{**}_{\bar{\pi}}$, the number of variables $c[\rho^*_u]$ might be less than q, i.e., less than the size of a configuration vector in $\mathcal{C}_\mathcal{A}$. In other words, the CSSP solution may only be a partial configuration, even when no FS is operated. The procedures we adopted to complete the CSSP solution are explained in Sec. 5.5.5.

5.5 Experimental results

In this section, we discuss the implementation details of our methodology.

5.5.1 The Hydro Unit Commitment problem

One of the most efficient and long-term techniques to store electricity is to transform it into potential energy, by pumping water up mountain valleys into water reservoirs. The HUC is the problem of finding the optimal scheduling of a multi-unit, pump-storage, hydro power station, for a short term period in which inflows and electricity prices are given (usually obtained by forecasts). The plant's operating and energy production are decided so as to maximize the revenue given by power selling (see, e.g., [192]), while satisfying diverse technical, physical and strategic constraints. The units of the plant can work as pumps or turbines and are fed by the same water reservoir. The HUC belong to the family of unit commitment problems.

In our experiments, we considered a fixed time horizon of 24h and a fixed hydro-system, with a single-pump. Therefore, all of our instances have the same size. The problem natively has a nonlinear structure, which arises from the relationship between the power output of the turbines, the corresponding water flow from the reservoir and the water volume in the reservoir. The HUC formulation that we use as reference, presented in [26], is an MILP, whereby the original nonlinearities are approximated by linear constraints.

The HUC is hard to solve in practice. In fact, even when its nonlinearities are linearized, it is often hard to find a feasible solution, let alone an optimal one. A further challenge arises from the combinatorial nature of the problem, which requires determining which units to activate, while accounting for, say, startup/shutdown costs and forbidden operational zones [181]. Finally, HUC instances are often large-scale and must be solved, in practice, in a reasonably short time, often for hundreds of times a day. In view of the mentioned complexities, the benefits of any improvements upon the solution process could be substantial. For this reason, we chose the HUC as the benchmark optimization problem (i.e., Π) to test our ACP methodology.

5.5.2 Building the training set

1. *Instance encoding* (features). The instances in Π' that we use all have the same size and their constraints do not vary overmuch. The characteristics that actually vary (and that we extracted from each instance) are mainly the coefficients of the objective function. Notwithstanding, our approach is general: the "problem structure" is encoded in the vector π, which is certainly class-specific, but need not be size-specific (one can e.g. use dimensionality reduction techniques to achieve feature vectors of the same size even for instances of varying size). Notably, for each instance, we chose to extract 55 features, varying from day to day: the date, 24 hourly prices, 24 hourly inflows, initial and target water volumes, upper and lower bound admitted on the water volumes. Thus, for

all HUC instances, $\pi \in \mathbb{R}^{55}$, with continuous/discrete components. All the instances have been randomly generated with an existing generator, which accurately reproduces realistic settings. The code of the generator can be found at `https://people.unipi.it/dimitri_thomopulos/libraries/hig/`

2. *Configuration parameters.* Our approach can help configure any subset of solver parameters. However, in order to reduce the time spent in constructing the training set, we decided to consider a reasonably small subset of parameters that are thought to have a definite impact on the problem at hand. We select 9 discrete CPLEX parameters: `fpheur`, `dive`, `probe`, `heuristicfreq`, `startalgorithm` and `subalgorithm` from `mip.strategy`; `crossover` from `barrier`; `mircuts` and `flowcovers`, from `mip.cuts`. For each of them, we consider between 2 and 4 different values. We then combine them so as to obtain 2304 parameter configurations. A configuration is encoded by a vector $c \in \{0,1\}^{23}$, where each categorical parameter is represented by its incidence vector.

3. *Performance measure.* We use CPLEX's integrality gap as $p_{\mathcal{A}}(\pi, c)$, which in [91] is defined as

$$\text{int. gap} = \frac{|\text{best integer sol.value} - \text{best relaxation value}|}{1\text{e}{-10} + |\text{best integer sol.value}|}.$$

It has been shown that MIP solvers can be affected by performance variability issues (see, e.g., [133]), owing to executing the solver on different computing platforms, permuting rows/columns of a formulation, adding valid and redundant constraints, effecting (apparently neutral) changes to the solution process, etc. In order to overcome this issue, first we sample three different random seeds. Then, for each couple (π, c), we carry out the following procedure: (i) we run CPLEX (using its Python API) three times on π, using the different random seeds, for 60 seconds; (ii) we record the middle out of the three obtained performance values, to be assigned to the pair (π, c). We call $\varrho(\pi, c)$ the performance measure thus obtained from the CPLEX output.

At this point, our dataset contains $s = 250 \times 2304 = 576k$ points, each with dimension $55 + 23 + 1 = 79$. We store this dataset as a matrix. Notably, we let $M_{\mathcal{S}}$ be the $s \times (m+q)$ matrix obtained by stacking the training inputs, so that: a) for all $i \leq s$, the i-th row displays the i-th input (π_i, c_i); b) for all $j \leq m+q$, the j-th component of (π_i, c_i) appears on the j-th column. We refer to the columns of $M_{\mathcal{S}}$ as "attributes".

We note that $\varrho(\pi, c)$ usually contains some extremely large floating point values. These values stand for "infinity" (denoted "∞" below), and appear whenever the CPLEX gap has a value close to zero in the denominator, or CPLEX cannot find

feasible solutions or valid bounds within the allotted CPU time. Moreover, these infinities may unduly bias the training process to learn $\bar{p}_{\mathcal{A}}$. Instead of scaling in the presence of these outliers, we deal with them as follows: we compute the maximum $\bar{\varrho}$ over all values of (the range of) $\varrho(\pi, c)$ lower than a given threshold (set to 1e+5 in our experiments), re-set all values of ϱ larger than the threshold to $\bar{\varrho} + 100$, then re-scale ϱ so that it lies within the interval $[0, 1]$. In the following, we call the resulting performance measure p_{ml}, to distinguish it from $p_{\mathcal{A}}$; so, $\bar{p}_{\mathcal{A}}$ is actually an approximation of p_{ml}.

We remark that setting the time limit, imposed on all of CPLEX runs, to 60 seconds provides the solver enough time to move past the preliminary processing and to begin working on closing the gap, even for very hard instances (i.e., the ones with long pre-processing times). This allows us to measure the actual impact that different parameter configurations have on the chosen performance measure. See Sec. 2.9.1 for more details on the customary approach of MILP solvers to the solution process.

4. *Feature engineering.* We process $M_{\mathcal{S}}$ to craft new learning attributes, so as to improve the approximation accuracy of our predictors. Notably, we process the date in order to extract the season, the week-day, the year-day, two flags called `isHoliday` and `isWeekend`, and we perform several sine/cosine encodings, that are customarily used to treat cyclical attributes. Moreover, we craft new features by computing statistics on the remaining 55 features. This task takes around 12 minutes to complete for the whole data set.

5. *Splitting the dataset.* Our Π' is composed of 250 HUC instances. We randomly divide them into 187 "in-sample" (IS) and 63 "out-of-sample" (OS), and split the rows of $M_{\mathcal{S}}$ accordingly (432k IS and 144k OS), into $M_{\mathcal{S}_{\mathsf{IS}}}$ and $M_{\mathcal{S}_{\mathsf{OS}}}$. We use the IS data to perform FS and to train the ML predictors. Then, we assess the performance of the whole pipeline (from the construction of $\bar{p}_{\mathcal{A}}$ to the solution of the CSSP) both on the OS instances — to test its generalization capabilities on unseen inputs — and on the IS instances — to evaluate its performance on the data that we learn from — as detailed below.

6. *Feature selection for learning $\bar{p}_{\mathcal{A}}$.* We use Python's `Pandas` `DataFrame`'s `corr` function to perform Pearson's linear correlation and `sklearn` `RandomForestRegressor`'s `feature_importances_` to perform DTs' feature importance algorithm, in order to get insights on which features contribute the most to yield accurate predictions. A detailed explanation of the employed FS techniques falls outside of the scope of this document, but the interested reader can refer to [79] for linear correlation, and [81, Ch. 10.13,15.3] for feature importance, among many others. In

order to perform FS, we use a dedicated subset of the IS dataset, composed of approximately 10k points and only employed for this task. Performing the selected FS techniques on this dataset takes around 8 minutes, and reduces π to 22 components. For the configuration vectors we consider three FS scenarios: aggFS, where FS is applied more aggressively, gentleFS and noFS, yielding c vectors with, respectively, $q' = 10, 14, 23$ components. We then filter the K-EP dataset $M_{\mathcal{S}_{\mathsf{IS}}}$ according to the selected attributes. In particular, for each instance: a) we delete all the rows of $M_{\mathcal{S}_{\mathsf{IS}}}$ where the components of the c-attributes eliminated by FS are not set to their `default` value; b) since this creates duplicate rows, for each subset of duplicate rows, we compute the average $p_{\mathcal{A}}$ keeping only one row. We call $M'_{\mathcal{S}'}$ the resulting dataset matrix. In all scenarios, we sample approximately 11k rows from $M'_{\mathcal{S}'}$ for the construction of $\bar{p}_{\mathcal{A}}$ in the K-EP.

5.5.3 Learning $\bar{p}_{\mathcal{A}}$: experimental setup and results

We present experimental results with linear SVR ("SVR lin"), SVR with a Gaussian kernel ("SVR rbf") and DTs ("DT"). We assess the prediction error of the predictors by NCV (see Sec. 3.5.3). Furthermore, our training includes a phase for determining and saving the hyperparameters and the coefficients of the ML predictor's functional form. We use Python's `sklearn.model_selection.RandomizedSearchCV` for the inner loop of the NCV and a customized implementation for the outer loop. Instead, we use `sklearn.svm.SVR`, `sklearn.svm.LinearSVR` and `sklearn.tree.DecisionTreeRegressor` to implement the ML predictors.

FS	predictor	mae$_{tr}$	mae$_{te}$	%mae$_{tr/te}$	mean p_{ml}	std p_{ml}	time
agg	SVR lin	2.248E-01	2.261E-01	99.4	6.267E-01	3.489E-01	**143.8**
	SVR rbf	**3.636E-02**	1.138E-01	32.0	6.268E-01	3.487E-01	17484.7
	DT	3.980E-02	**8.779E-02**	45.3	6.267E-01	3.489E-01	7672.6
gentle	SVR lin	2.572E-01	2.568E-01	100.2	6.280E-01	3.763E-01	**196.9**
	SVR rbf	**7.740E-02**	1.715E-01	45.1	6.272E-01	3.772E-01	14464.0
	DT	8.331E-02	**1.352E-01**	61.6	6.275E-01	3.768E-01	5512.2
no	SVR lin	2.834E-01	2.831E-01	100.1	6.045E-01	3.975E-01	**202.7**
	SVR rbf	**1.137E-01**	2.186E-01	52.0	6.031E-01	3.974E-01	15688.2
	DT	1.261E-01	**1.798E-01**	70.1	6.035E-01	3.979E-01	4438.0

Table 5.1: Prediction error of $\bar{p}_{\mathcal{A}}$, in terms of mae (measured on the training and on the test set) and training time, for different FS scenarios

In Tab. 5.1, we report the average NCV training and test mae ("mae$_{tr}$", "mae$_{tr}$"), the ratio between mae$_{tr}$ and mae$_{te}$ ("%mae$_{tr/te}$") and the seconds necessary for learning $\bar{p}_{\mathcal{A}}$ ("time"), for different FS scenarios and the three predictors. We also report the mean and standard deviation of p_{ml}, for assessing the magnitude of the prediction errors

committed. We note that the training and test sets used to obtain the results in Tab. 5.1 all have the same number of rows.

The linear SVR exhibits the largest (i.e., worst) mae, both on the training set and the test set and in every FS scenario, despite being the faster to train. The DT commits the smallest mae_{te}, so it is the paradigm with the greatest potential for generalization. The nonlinear SVR shows a slightly higher mae than the DT, but it also shows much larger training times. Training a DT takes twenty to fifty times longer than fitting a linear SVR, but only about a third of the time needed to learn a nonlinear SVR. Moreover, the DT training times do not vary overmuch (indeed, they even improve) going from agg FS to no FS. Thus, DTs provide an excellent option for producing very accurate out-of-sample predictions in relatively short time, even when trained on sizable datasets, with many attributes. "SVR lin" predictors are the least prone to overfitting: their training error is always very close to their test error, i.e., "$\%\mathsf{mae}_{tr/te}$" is always around 100%. On the other hand, the fact that "$\%\mathsf{mae}_{tr/te}$" for "SVR lin" is consistently very low points to a high risk of overfitting for predictors of this type. All in all, from the table we gather that the ability to avoid overfitting improves when more attributes are considered: evidently, a more informative (although nonredundant) dataset allows a superior generalization.

A common issue in data-driven optimization — where ML-derived predictors provide approximations of the unknown components of an optimization problem — is that using standard ML error metrics may not lead to good solutions of the optimization problem itself [55, 74]. In fact, while ML is deployed to approximate single components of the problem — i.e., its objective function and/or a subset of its constraints — the complex structure of an optimization problem is determined by the logical interactions between all of its components. However, ML predictions are natively incapable of factoring in this structure: their predictions, although capable of achieving a low generalization error (i.e., accurate by ML standards), may fail to identify the true global/local optima of the optimization problem (i.e., be inaccurate by optimization standards). Recently, an interesting work [59] addressed this problem, by devising a methodology based on NNs (see Sec. 3.6.3), that incorporates a Lagrangian relaxation of the optimization problem into the training loss function. However, this methodology is not deployable off-the-shelf, as it needs an *ad hoc* implementation for the specific optimization problem at hand.

In our PaO/PaI methodologies, the correspondence between the global optima of $\bar{p}_{\mathcal{A}}$ and those of $p_{\mathcal{A}}$ ensures that the solution of the CSSP can actually solve the ACP, for the instance at hand. However, measuring this correspondence at training time is nontrivial, as we said above. We try to overcome this difficulty by training our predictors not only with a variety of more or less customary ML metrics (mean absolute error mae, mean squared error mse, log-cosh error lce), but also with a custom metric cmae_{δ}. For

all $i \in S$, we let $p_i = p_{\mathsf{ml}}(\pi_i, c_i)$, $\bar{p}_i = \bar{p}_{\mathcal{A}}(\pi_i, c_i)$, and the prediction error be

$$\rho_i = p_i - \bar{p}_i \,.$$

With the above notation, we have that

$$\mathsf{mae} = \frac{1}{2}|\rho_i| \,, \qquad \mathsf{mse} = \frac{1}{2}(\rho_i)^2 \,, \qquad \mathsf{lce} = \log \frac{\exp^{\rho_i} + \exp^{-\rho_i}}{2} \,.$$

For $\delta \in [0, 1]$, our custom metric is defined as:

$$\mathsf{cmae}_\delta = \begin{cases} \rho_i + \frac{\rho_i}{1+\exp(\rho_i)} & \text{if } p_i \leq \delta \text{ and } \rho_i < 0 \\ \rho_i + \frac{\rho_i}{1+\exp(-\rho_i)} & \text{if } p_i \geq 1-\delta \text{ and } \rho_i > 0 \\ \rho_i & \text{if } \delta \leq p_i \leq 1-\delta \\ 0 & \text{otherwise} \,. \end{cases} \tag{5.12}$$

Note that Eq. (5.12) requires p_i to be scaled in $[0, 1]$ to work. We want $\bar{p}_{\mathcal{A}}$ to accurately predict the points around the global minimum/maximum of $p_{\mathcal{A}}$. Hence, the cmae_δ penalizes prediction overestimates, while allowing any underestimates, at δ-minima (i.e., points such that $p_i \leq \delta$), and viceversa at δ-maxima. Furthermore, it behaves exactly like the mae at points s.t. $\delta \leq p_i \leq 1 - \delta$. In the tables below, we report results for the PaO methodology, with Eq. (5.12) as the ML loss metric.

5.5.4 The CSSP: experimental setup and results

We use AMPL to formulate and solve the CSSP for the IS and OS instances. To solve the CSSP corresponding to the nonlinear SVR paradigm, we run the nonlinear solver Bonmin. Bonmin is manually configured with settings `heuristic_dive_fractional yes`, `algorithm B-Hyb`, `heuristic_feasibility_pump yes`, and with a time limit of 60 seconds. Moreover, to solve the CSSP associated to the linear SVR and DT paradigms, we run CPLEX, with its default parameter configuration.

Then, for each instance in Π', we retrieve our CSSP solution c^{**}_{cssp}. Since we have enumerated all possible configurations, we can also compute the "true" CSSP global optimum c^*, for sake of comparison.

In Tab. 5.2 and 5.3, we report the time taken to solve the CSSP ("time"), the percentage of cases where $c^{**}_{\mathsf{cssp}} = c^*$, ("%glob.") and, for all the instances where this is not true, the distance between $\bar{p}_{\mathcal{A}}(c^{**}_{\mathsf{cssp}})$ and $\bar{p}_{\mathcal{A}}(c^*)$ ("CSSP gap", averaged over all the corresponding instances of the considered IS/OS set).

From Tab. 5.2, we gather that performing more aggressive FS techniques improves both the nonlinear SVR solution quality — with higher "% glob" and lower "CSSP gap" — and time. In fact, FS reduces the size of the CSSP, by eliminating variables and related constraints, thus making it easier to optimize it. In general, the quality of c^{**}_{cssp} for "SVR rbf" is high (on average, 85% on IS instances and 90% on the OS

ones), even when c^{**}_{cssp} is only a local optimum. In fact, "CSSP gap" is always lower than 2.7E-02, i.e., a local optimum is never larger than 2.7% of the global optimum. Further, Bonmin never needs more than 15 seconds to solve the CSSP, which is well below the designated time limit of 60 seconds. The statistics reported in Tab. 5.2 also show a clear dominance of the DT and the linear SVR paradigms over the nonlinear SVR one. Their c^{**}_{cssp} is always a global optimum ("%glob" is always 100%), and their CSSP is always solved in less than one second. Moreover, the two tables above shows that another advantage of solving CSSPs corresponding to DTs and linear SVR is that its size (depending on the FS scenarios) does not affect "time".

set	FS	%glob			CSSP gap			time		
		SVR rbf	DT	SVR lin	SVR rbf	DT	SVR lin	SVR rbf	DT	SVR lin
IS	agg	86.01	100	100	2.673E-02	0	0	**5.34**	**0.04**	0.03
	gentle	**88.95**			2.583E-02			9.39	**0.04**	0.03
	no	82.00			**2.116E-02**			14.61	0.05	0.03
OS	agg	**94.18**	100	100	2.335E-02	0	0	**5.16**	0.05	0.03
	gentle	92.06			1.984E-02			9.99	**0.04**	0.03
	no	83.60			**1.733E-02**			13.06	**0.04**	0.03

Table 5.2: CSSP solution process performance, by instance set, FS scenario and predictor

set	loss	%glob			CSSP gap			time		
		SVR rbf	DT	SVR lin	SVR rbf	DT	SVR lin	SVR rbf	DT	SVR lin
IS	cmae_2	84.85	100	100	3.000E-02	0	0	11.41	0.05	0.03
	cmae_3	85.92			2.439E-02			10.06	0.05	0.03
	cmae_4	83.07			2.828E-02			10.34	0.05	0.03
	lce	**90.91**			**1.747E-02**			**6.48**	0.05	0.03
	mae	87.34			2.257E-02			10.91	**0.04**	0.03
	mse	81.82			2.474E-02			9.50	**0.04**	0.03
OS	cmae_2	**93.12**	100	100	2.317E-02	0	0	10.68	**0.04**	0.03
	cmae_3	88.36			2.154E-02			10.93	**0.04**	0.03
	cmae_4	88.89			1.753E-02			10.08	0.05	0.03
	lce	92.06			1.599E-02			**6.25**	**0.04**	0.03
	mae	88.36			**1.196E-02**			10.75	**0.04**	0.03
	mse	88.89			3.085E-02			7.74	0.05	0.03

Table 5.3: CSSP solution process performance, by instance set, K-EP loss metric and predictor

Tab. 5.3 shows the effects of adopting different K-EP loss metrics on CSSP. In the case of "SVR rbf", adopting lce yields the fastest CSSP solutions, the best "%glob" and "CSSP gap" results on IS instances, and very good results on OS instances, although no metric shows a clear advantage on the OS set. In contrast, in the case of "DT" and

"linear SVR", the same observations as in Tab. 5.2 apply.

We remark that devising more efficient techniques to solve the CSSP (say, reformulations, decomposition, etc.) might be necessary with some ML paradigms, if our approach is scaled to considerably more algorithmic parameters.

5.5.5 CSSP solution completion

To construct our ML predictors, we consider different FS scenarios for the input components of $\mathcal{S}$; then, we study how they affect the CSSP solution quality. In the training set, we use input tuples (π, c). In the CSSP, π is an input of the problem, while c encodes its variables. Thus, FS techniques that reduce π do not change the structure of the CSSP overmuch, while FS techniques operating on c alter the variables and constraints of the CSSP. In particular, upon deployment of FS, the set of feasible configurations in Eq. (5.1) becomes

$$\mathcal{C}'_{\mathcal{A}} = \{c' \in \{0,1\}^{q'} \mid A'c' \leq \beta'\}, \quad q' < q, \tag{5.13}$$

so the CSSP solution reduces to q' variables. This raises the question of how to determine the $q - q'$ left-out configuration components from a partial CSSP solution.

Consider a single parameter ω, encoded by an incidence vector of q_ω binary components summing to one. We let q'_ω be the number of components picked by FS. If $q_\omega = q'_\omega$, then all q_ω variables encoding ω are decided by solving the CSSP. Instead, if $q_\omega - q'_\omega = 1$, The value to be assigned to the only left-out component is always decided unambiguously. To see this, consider Tab. 5.4, representing a single parameter with four components (on the columns) and four configurations (on the rows). Choose any column as left-out, and then pick a row; the values on this row, excluding the one corresponding to the left-out column, encode a CSSP solution. Since each row corresponds to a unique configuration, the value of the left-out component to be determined is precisely that of the selected row. However, if $q_\omega - q'_\omega > 1$, deciding the values of the q'_ω left-out

ω			
ω_0	ω_1	ω_2	ω_3
1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

Table 5.4: A parameter ω with $q_\omega = 4$ and all of its possible configurations

components may not be straighforward. To see this, consider again Tab. 5.4 and pick ω_2 and ω_3 as left-out: if the CSSP solution was, for instance, $(\omega_0, \omega_1) = (0, 0)$ (on the

third and fourth rows), it would be unclear how to choose between $(\omega_2, \omega_3) = (1, 0)$ or $(\omega_2, \omega_3) = (0, 1)$.

To address this issue, we have devised and implemented several heuristics, which we execute once, after the K-EP. Their purpose of each heuristic is to create and store a dictionary where all possible partial CSSP solutions (i.e., vectors $c^{**}_{\mathsf{cssp}}\}$ of size q') are completed, by matching them with a vector of $q - q'$ left-out configuration components. How this matching is computed changes with each heuristic. Then, the saved dictionaries can be quickly browsed, after solving the CSSP, to complete a solution.

Below we describe `heuristic_1`, `heuristic_2`, and their variants.

5.5.5.1 `heuristic_1`

Given the q' FS-picked configurations components and the processed training matrix $M'_{\mathcal{S}'}$, separately for each parameter ω (with q_ω variables and $q_\omega - q'_\omega$ left-out components), `heuristic_1` performs the following steps:

1. it groups the rows of $M'_{\mathcal{S}'}$ by the q_ω variables;

2. for each group, it computes the mean or the minimum $p_{\mathcal{A}}$ over the corresponding rows of $M'_{\mathcal{S}'}$. This produces a new column, that we call $p'_{\mathcal{A}}$;

3. for each possible combination of the FS-picked q'_ω components, it deletes the rows of the grouped matrix such that the q'_ω component values are different than the ones in that combination. Then, the row with minimum $p'_{\mathcal{A}}$ is identified among the remaining ones, and stored in the dictionary.

5.5.5.2 `heuristic_2`

Given the q' FS-picked configurations components and the processed training matrix $M'_{\mathcal{S}'}$, `heuristic_2` executes the following tasks:

1. it groups the rows of $M'_{\mathcal{S}'}$ by all of the q' variables;

2. for each group, it retrieves the row in $M'_{\mathcal{S}'}$ with minimum $p_{\mathcal{A}}$. The $q - q'$ left-out components of that row are saved in the dictionary.

5.5.5.3 Clustering variant

We also provide an option to embed clustering into the heuristics described above: before the CSSP, the elements of $M'_{\mathcal{S}'}$ are clustered based on π; then, the heuristics are run, separately for each cluster, on the rows of $M'_{\mathcal{S}'}$ belonging to that cluster.

5.5.6 Assessing the performance of our recommender

In this section, we compare two CPLEX configurations: the CSSP solution c^{**}_{cssp} and CPLEX's default one, c_{cpx}, such that all parameters are set to "auto". We observe that, when we instruct CPLEX to use the "auto" mode, the solver automatically deploys its internal heuristics to select c_{cpx} according to the structure of the instance to be solved. As CPLEX is a commercial solver, the details of this procedure are confidential, but we can certainly consider c_{cpx} a PI configuration. We use the two configurations to configure CPLEX, that we deploy to optimize the IS and OS instances; we discuss the corresponding solver performances in terms of $p_{\mathcal{A}}$ and p_{ml}. To this end, we retrieve $p_{\mathcal{A}}(\cdot, c^{**}_{\mathsf{cssp}})$, $p_{\mathsf{ml}}(\cdot, c^{**}_{\mathsf{cssp}})$, $p_{\mathcal{A}}(\cdot, c_{\mathsf{cpx}})$ and $p_{\mathsf{ml}}(\cdot, c_{\mathsf{cpx}})$ from $M'_{\mathcal{S}}$, for every IS and OS instance.

Below, we report the percent total wins "%w" and the percent non-worsenings "%w+d", i.e., the percentage of instances such that $p_{\mathcal{A}}(\cdot, c^{**}_{\mathsf{cssp}}) < p_{\mathcal{A}}(\cdot, c_{\mathsf{cpx}})$ and $p_{\mathcal{A}}(\cdot, c^{**}_{\mathsf{cssp}}) \leq p(\cdot, c_{\mathsf{cpx}})$, by the first sixteen decimal digits of $p_{\mathcal{A}}$, in scientific notation. Furthermore, we compute the percent wins-over-nondraws "%w-d", i.e., the wins over all instances such that $p_{\mathcal{A}}(\cdot, c^{**}_{\mathsf{cssp}}) \neq p(\cdot, c_{\mathsf{cpx}})$. We also compute, for each instance with a draw, the relative gap between $p_{\mathcal{A}}(\cdot, c^{**}_{\mathsf{cssp}})$ and the minimum $p_{\mathcal{A}}$, and then average it over the IS and OS instances. Finally, we measure the time to optimality, for instances with a draw ("time d") and for the others ("time nond"). We compute the same metrics for p_{ml} too. We will pay particular attention to the statistics calculated on instances OS: they allow us to assess the generalization capabilities of our methodology, namely, its ability to solve the ACP on data unseen during the K-EP.

set	p	FS	%w			%w+d			%w-d		
			SVR rbf	DT	SVR lin	SVR rbf	DT	SVR lin	SVR rbf	DT	SVR lin
IS	$p_{\mathcal{A}}$	agg	48.69	**48.56**	41.82	**97.65**	**97.92**	89.56	**95.41**	**95.91**	80.04
		gentle	**48.59**	45.86	41.00	97.64	94.16	88.76	95.40	88.74	78.49
		no	47.33	47.33	**41.98**	96.35	95.28	**89.66**	92.84	90.99	**80.26**
	p_{ml}	agg	**82.92**	**83.18**	**70.07**	**95.07**	**95.60**	**84.98**	**94.39**	94.98	**82.34**
		gentle	81.39	77.54	69.28	93.61	92.16	84.97	92.71	90.81	82.16
		no	78.79	80.21	69.61	92.16	94.12	84.31	90.94	**93.16**	81.61
OS	$p_{\mathcal{A}}$	agg	33.99	34.61	33.60	77.29	83.51	86.33	60.06	67.95	71.10
		gentle	35.49	36.07	**33.64**	83.60	88.62	**86.42**	68.60	76.35	**71.33**
		no	**35.98**	**39.42**	32.80	**85.45**	**93.65**	84.13	**71.31**	**86.33**	67.53
	p_{ml}	agg	56.92	60.67	**64.33**	73.63	77.91	79.81	68.34	73.35	**76.11**
		gentle	60.32	63.27	64.29	76.46	80.03	**80.11**	71.89	75.85	76.37
		no	**61.38**	**75.93**	62.43	**76.98**	**91.27**	78.31	**72.86**	**89.68**	74.19

Table 5.5: CSSP solution performance, compared to CPLEX's default configuration, by instance set and FS scenario

Tab. 5.5 shows that CSSP solutions do not generally benefit from the application of aggressive FS techniques (i.e., when q' is small). In fact, although the agg and gentle scenarios report large "%w", "%w+d" and "%w-d" on IS instances, no or gentle FS are

better strategies on OS instances, for both nonlinear SVR and DTs. Only linear SVR performs better when the q' is small. The results of different FS techniques on IS and OS instances reported in Tab. 5.5 confirm what we observed about Tab. 5.1: a) the risk of overfitting is likely to increase with the use of more aggressive FS techniques, and b) the linear SVR paradigm seems to be the least vulnerable to this risk (at least, according to our experiments).

set	gap d	time d	time nond
IS	1.146E-10	24.16	34.48
OS	0	30.89	46.39

Table 5.6: Instance hardness, by time to optimality and relative gap

The fact that, in Tab. 5.6, the values of the "time d" column are always smaller than "time nond" reveal that draws occur on easier instances. Moreover, "gap d" values are always very close or equal to zero: this means that whenever CPLEX, tasked with solving those easier instances and configured by c_{cpx}, manages to close the gap within the allotted time limit, our c^{**}_{cssp} is just as efficient. In these cases, there is little room for improvement over the default CPLEX settings; the most that can be expected from the CSSP configuration is to be able to match c_{cpx}'s performance, which c^{**}_{cssp} manages to do. Further, looking at the "%w-d" column in Tab. 5.5, we see that on harder, "nondraw" instances, we beat CPLEX at least 90% of the time on the IS instances, and at least 60% of the time on OS instances.

We also remark that DTs clearly outperform the other paradigms, by their superior scores in every computational setting, in particular on OS instances and when no FS is executed.

Finally, Tab. 5.5 clearly shows that the best results are those corresponding to p_{ml}, rather than $p_{\mathcal{A}}$. Since $\bar{p}_{\mathcal{A}}$, optimized in the CSSP, is an approximation of p_{ml} and not of $p_{\mathcal{A}}$, this is not surprising. It further highlights the strong connection between the approximation of $p_{\mathcal{A}}$ and the efficacy of recommender, which exploits the structure of that approximation.

Tab. 5.7 displays the same metrics as Tab. 5.5, here aggregated by ML loss metric. The rows corresponding to p_{ml} indicate that mse is the best choice for the DT paradigm, and that cmae_2 should be selected for the linear SVR one. As for the nonlinear SVR, mae allows the best results on OS instances, so it should be preferred over mse, that is the best option on the IS ones. These observations also apply to the table entries concerning $p_{\mathcal{A}}$ and IS instances. Instead, when we consider the OS ones, the mae metric dominates the others in most of the cases. In general, we can conclude that the choice of

the K-EP loss metric has some impact on the quality of c^{**}_{cssp}, but does not significantly affect performance. Further, our custom metric often fails to provide a valid alternative to the other customary ML loss metrics.

set	p	loss	%w			%w+d			%w-d		
			SVR rbf	DT	SVR lin	SVR rbf	DT	SVR lin	SVR rbf	DT	SVR lin
IS	$p_{\mathcal{A}}$	cmae_2	47.98	46.88	41.92	96.97	95.34	**89.93**	94.07	91.08	**80.65**
		cmae_3	48.34	47.09	41.50	97.42	95.78	88.83	94.95	91.82	78.82
		cmae_4	48.37	46.82	**41.95**	97.50	95.99	89.78	95.10	92.12	80.41
		lce	47.86	**48.07**	41.33	96.76	**96.61**	88.83	93.68	**93.46**	78.75
		mae	47.86	47.50	41.35	96.70	94.83	89.04	93.57	90.28	79.07
		mse	**48.81**	47.12	41.53	**97.92**	96.17	89.54	**95.93**	92.50	79.88
	$p_{\mathtt{ml}}$	cmae_2	80.90	79.98	**70.65**	93.29	94.36	**85.44**	92.34	93.33	**82.90**
		cmae_3	81.05	80.54	68.48	93.79	94.53	83.54	92.84	93.63	80.63
		cmae_4	81.58	79.77	69.79	93.76	93.29	84.64	92.89	92.21	81.96
		lce	80.10	**81.40**	69.01	93.02	94.39	84.34	91.97	93.54	81.49
		mae	80.51	78.85	69.99	93.94	92.36	**85.44**	92.99	91.17	82.78
		mse	**82.06**	81.34	69.99	**93.88**	**94.83**	85.12	**93.06**	**94.02**	82.46
OS	$p_{\mathcal{A}}$	cmae_2	34.66	34.30	**34.13**	79.81	85.71	86.60	63.35	71.11	71.83
		cmae_3	32.80	**38.18**	32.54	80.69	88.89	83.95	63.35	78.10	67.16
		cmae_4	35.71	37.04	32.80	81.31	87.30	84.13	65.96	75.32	67.50
		lce	**36.16**	38.01	33.95	82.98	90.92	85.98	68.21	81.10	70.77
		mae	**36.16**	36.07	33.16	**85.10**	87.65	**87.30**	**71.45**	74.59	**72.44**
		mse	35.45	36.60	33.51	82.80	**91.09**	85.80	67.63	**81.06**	70.24
	$p_{\mathtt{ml}}$	cmae_2	59.26	65.17	**65.26**	76.01	84.39	**80.34**	71.22	80.29	**76.84**
		cmae_3	56.79	64.90	62.26	74.78	81.39	78.48	69.22	77.57	74.28
		cmae_4	58.11	64.02	62.43	72.66	78.84	78.22	68.01	75.14	74.14
		lce	61.82	69.40	63.23	77.34	83.86	78.84	73.19	81.00	74.92
		mae	**62.35**	66.14	64.20	**79.28**	83.69	80.25	**75.04**	80.21	76.47
		mse	58.91	**70.11**	64.73	74.07	**86.24**	**80.34**	69.50	**83.55**	76.70

Table 5.7: CSSP solution performance, compared to CPLEX's default configuration, by instance set and ML loss metric

With Tab. 5.8, we can assess the impact of different heuristics for the completion of the CSSP solution (described in Sec. 5.5.5.1–5.5.5.3). We tested six heuristics: `heuristic_1`, with mean and minimum, and `heuristic_2`, both with and without the clustering option. Furthermore, we report results corresponding to the case where no FS is carried out, hence no heuristics are executed (marked by rows whose header, under the "heur" column, is "no"). Firstly, the table shows that, on IS instances, no heuristic unequivocally dominates the others, but "clst-h1-mean" and "noclst-h2" yield slight better performances. However, when solving OS instances, not executing any FS procedure is almost always the best strategy, except for "SVR lin", for which the "noclst-h2" heuristic has a clear advantage over the others. The no FS scenario yields the most impressive OS performances, when compared to the other scenarios.

Moreover, DTs prove again to be the most effective, especially in terms of "%w-d" and for solving OS instances. All in all, they are faster to train, provide better

generalization in the K-EP (see Tab. 5.1), and also achieve excellent results in the CSSP, and even when tackling large configuration vectors.

set	p	heur	%w			%w+d			%w-d		
			SVR rbf	DT	SVR lin	SVR rbf	DT	SVR lin	SVR rbf	DT	SVR lin
IS	$p_{\mathcal{A}}$	clst-h1-mean	**49.20**	47.64	41.62	**98.40**	**96.57**	89.48	**96.85**	93.35	79.84
		clst-h1-min	48.80	47.10	41.35	97.82	96.12	89.26	95.74	92.46	79.40
		clst-h2	48.08	46.66	40.78	96.79	95.14	87.70	93.77	90.58	76.84
		no	47.33	47.33	41.98	96.35	95.28	89.66	92.84	90.99	80.26
		noclst-h1-mean	49.02	**47.77**	41.09	98.04	96.79	88.99	96.18	**93.79**	78.88
		noclst-h1-min	48.80	47.10	41.35	97.82	96.12	89.26	95.74	92.46	79.40
		noclst-h2	47.95	46.97	**42.25**	97.01	95.50	**90.24**	94.16	91.31	**81.25**
	$p_{\texttt{ml}}$	clst-h1-mean	**82.84**	81.02	69.92	94.47	**94.39**	85.12	93.74	**93.48**	82.45
		clst-h1-min	81.91	80.21	69.79	93.94	94.07	85.16	93.10	93.10	82.46
		clst-h2	81.86	79.50	67.96	**94.70**	93.09	83.73	93.92	92.00	80.69
		no	78.79	80.21	69.61	92.16	94.12	84.31	90.94	93.16	81.61
		noclst-h1-mean	82.04	**81.24**	69.25	93.94	94.16	84.89	93.11	93.24	82.09
		noclst-h1-min	81.91	80.21	69.79	93.94	94.07	85.16	93.10	93.10	82.46
		noclst-h2	82.35	79.99	**71.35**	95.05	93.49	**85.78**	**94.33**	92.46	**83.37**
OS	$p_{\mathcal{A}}$	clst-h1-mean	35.85	35.45	33.33	81.08	86.90	87.04	65.67	73.23	72.06
		clst-h1-min	35.05	34.92	33.33	81.88	85.58	85.98	66.20	71.23	70.42
		clst-h2	32.94	34.79	33.60	77.38	84.79	85.85	59.69	70.12	70.49
		no	**35.98**	**39.42**	32.80	85.45	**93.65**	84.13	**71.31**	**86.33**	67.53
		noclst-h1-mean	35.05	35.45	33.33	81.61	87.17	85.98	65.89	73.78	70.42
		noclst-h1-min	35.05	34.92	33.33	**81.88**	85.58	85.98	66.20	71.23	70.42
		noclst-h2	34.52	36.51	**34.79**	78.84	86.38	**87.43**	62.33	73.33	**73.51**
	$p_{\texttt{ml}}$	clst-h1-mean	59.79	62.30	64.81	74.74	78.97	**80.82**	70.24	74.68	77.17
		clst-h1-min	59.79	62.96	63.89	76.85	80.29	79.89	72.06	76.09	76.06
		clst-h2	54.76	59.26	62.43	71.16	76.06	77.78	65.45	71.14	73.75
		no	**61.38**	**75.93**	62.43	**76.98**	**91.27**	78.31	**72.86**	**89.68**	74.19
		noclst-h1-mean	60.05	64.15	64.68	77.12	81.35	80.69	72.38	77.40	77.01
		noclst-h1-min	59.79	62.96	63.89	76.85	80.29	79.89	72.06	76.09	76.06
		noclst-h2	57.54	60.19	**66.14**	73.54	76.85	80.69	68.52	72.20	**77.39**

Table 5.8: CSSP solution performance, compared to CPLEX's default configuration, by instance set and CSSP solution completion heuristic

In Tab. 5.9 and 5.10, we report the percentage of IS and OS instances such that CPLEX, configured by c^{**}_{cssp} and c_{cpx}, manages to find a feasible solution ("%feas c^{**}_{cssp}" and "%feas c_{cpx}"). Our results reveal that CPLEX's default configuration emphasizes feasibility, as it always ensures that the solver reaches a feasible solution within the time limit ("%feas c_{cpx}" is always 100%). Our c^{**}_{cssp} achieves slightly worse performance, and again proves to be more effective in solving IS instances than OS instances: on IS instances, the average "%feas c^{**}_{cssp}" is 98.02% for the nonlinear SVR, 96.81% for the DT and 89.88% for the linear SVR; on OS instances, it is 90.26% for the nonlinear SVR, 91.78% for the DT and 86.08% for the linear SVR. From this we gather that DT-based CSSPs solutions are most likely to help CPLEX find a feasible solution, when the solver is used to optimize $\bar{\pi}$.

In Tab. 5.9 and 5.10, we also examine the quality of the feasible solutions found by using c^{**}_{cssp} and c_{cpx}. The columns "gap$_p$ c^{**}_{cssp}", "gap$_p$ c_{cpx}", "gap$_d$ c^{**}_{cssp}" and "gap$_d$ c_{cpx}" report the primal and dual gaps achieved by CPLEX, configured by c^{**}_{cssp} or c_{cpx}, averaged over the instances for which the solver manages to find a feasible solution. Firstly, we

set	FS	loss	%feas c*			%feas c_{cpx}	gap_p c*			gap_p c_{cpx}	gap_d c*			gap_d c_{cpx}
			SVR rbf	DT	SVR lin		SVR rbf	DT	SVR lin		SVR rbf	DT	SVR lin	
IS	agg	cmae_2	98.04	98.75	90.11	100	7.850E-02	8.653E-02	4.971E-02	5.812E-01	5.179E-02	5.816E-02	**6.259E-02**	1.018E-01
		cmae_3	97.86	**98.84**	89.93		7.669E-02	8.266E-02	6.571E-02		5.111E-02	6.225E-02	7.482E-02	
		cmae_4	97.59	98.66	90.20		9.267E-02	8.058E-02	**4.897E-02**		6.680E-02	**5.408E-02**	7.362E-02	
		lce	**98.48**	98.75	90.20		7.836E-02	**7.956E-02**	4.981E-02		5.345E-02	5.514E-02	7.979E-02	
		mae	97.59	98.31	90.20		7.747E-02	8.695E-02	4.987E-02		**4.998E-02**	6.486E-02	7.689E-02	
		mse	98.13	98.48	**90.37**		**7.443E-02**	8.140E-02	6.828E-02		5.621E-02	7.140E-02	7.771E-02	
	gentle	cmae_2	98.75	92.25	**89.93**		**6.975E-02**	8.910E-02	**4.460E-02**		**3.792E-02**	5.843E-02	**5.512E-02**	
		cmae_3	99.38	**97.68**	**89.93**		8.271E-02	1.202E-01	**4.460E-02**		4.245E-02	**5.062E-02**	5.603E-02	
		cmae_4	98.48	95.01	**89.93**		7.085E-02	1.021E-01	**4.460E-02**		5.015E-02	5.516E-02	5.602E-02	
		lce	97.50	**97.68**	87.17		1.068E-01	9.698E-02	2.004E-01		5.865E-02	5.211E-02	2.124E-01	
		mae	96.43	94.65	88.32		7.183E-02	8.870E-02	5.390E-02		4.159E-02	6.327E-02	5.990E-02	
		mse	**99.55**	96.43	**89.93**		8.930E-02	**8.657E-02**	**4.460E-02**		5.533E-02	5.438E-02	5.602E-02	
	no	cmae_2	96.79	**98.40**	**91.44**		1.504E-01	7.780E-02	**4.385E-02**		9.708E-02	4.720E-02	**5.012E-02**	
		cmae_3	95.72	97.33	89.30		1.193E-01	1.168E-01	6.076E-02		8.104E-02	5.751E-02	1.797E-01	
		cmae_4	98.40	95.72	90.37		1.143E-01	6.861E-02	1.147E-01		7.253E-02	**4.577E-02**	1.291E-01	
		lce	98.40	94.65	90.37		1.454E-01	**5.558E-02**	4.458E-02		1.061E-01	5.119E-02	5.432E-02	
		mae	**98.93**	94.12	90.37		**1.058E-01**	7.090E-02	4.458E-02		**4.698E-02**	5.064E-02	5.432E-02	
		mse	98.40	96.79	89.84		1.819E-01	8.987E-02	4.465E-02		1.679E-01	4.861E-02	5.845E-02	
OS	agg	cmae_2	**89.42**	**90.74**	87.04	100	2.054E-01	1.204E-01	4.624E-02	4.633E-01	9.062E-02	6.757E+00	7.217E-02	8.121E-02
		cmae_3	88.89	90.48	87.04		2.061E-01	1.408E-01	4.643E-02		9.395E-02	1.336E-01	**6.550E-02**	
		cmae_4	88.10	86.24	86.51		2.075E-01	1.023E-01	4.660E-02		9.874E-02	1.260E-01	8.740E-02	
		lce	89.42	88.62	87.04		1.765E-01	**9.950E-02**	5.123E-02		9.301E-02	1.319E-01	9.117E-02	
		mae	88.62	90.48	**87.30**		1.792E-01	1.125E-01	4.851E-02		**7.733E-02**	9.503E-02	8.443E-02	
		mse	**89.42**	89.68	86.51		**1.744E-01**	1.125E-01	**4.609E-02**		1.021E-01	**8.426E-02**	8.239E-02	
	gentle	cmae_2	89.68	82.01	85.71		1.471E-01	9.676E-02	4.932E-02		**6.471E-02**	1.192E-01	5.373E-02	
		cmae_3	87.83	92.06	85.71		1.261E-01	1.190E-01	4.932E-02		1.189E-01	**5.017E-02**	5.373E-02	
		cmae_4	91.01	89.68	85.71		1.541E-01	1.310E-01	4.932E-02		1.283E-01	9.173E-02	5.373E-02	
		lce	**94.71**	95.50	85.98		1.922E-01	1.125E-01	**2.548E-02**		9.251E-02	6.486E-02	**4.776E-02**	
		mae	92.86	89.95	**89.68**		1.486E-01	**8.022E-02**	4.966E-02		1.078E-01	6.013E-02	7.417E-02	
		mse	90.21	**96.83**	85.71		**1.025E-01**	8.736E-02	4.932E-02		7.853E-02	7.386E-02	5.400E-02	
	no	cmae_2	87.30	93.65	**87.30**		6.968E-02	**3.875E-02**	4.908E-02		7.931E-02	3.488E-02	5.275E-02	
		cmae_3	92.06	**98.41**	84.13		1.231E-01	9.757E-02	6.368E-02		8.500E-02	**2.870E-02**	2.843E-01	
		cmae_4	88.89	96.83	80.95		1.305E-01	9.445E-02	**2.729E-02**		9.687E-02	3.758E-02	**3.943E-02**	
		lce	90.48	93.65	85.71		**6.714E-02**	4.978E-02	4.973E-02		**4.807E-02**	5.009E-02	5.672E-02	
		mae	**93.65**	90.48	85.71		1.371E-01	9.420E-02	4.973E-02		6.611E-02	1.349E-01	5.672E-02	
		mse	92.06	96.83	85.71		1.454E-01	6.355E-02	4.998E-02		6.329E-02	4.899E-02	5.333E-02	

Table 5.9: CSSP solution and CPLEX's default configuration performances, in terms of $p_{\mathcal{A}}$ gap, by instance set and ML loss metric

see that the primal/dual gaps provided by our methodologies are almost always better than those achieved by CPLEX's default setting. In fact, c^{**}_{cssp} yields gaps between 68% and 92% smaller than those obtained by c_{cpx}. The only scenario where our methodology fails to beat CPLEX's default configuration is when DTs are used in an aggressive FS scenario, to solve OS instances, particularly when cmae_2 is the ML loss metric of choice. This is evident by looking at Tab. 5.9. Moreover, since in Tab. 5.10 results are aggregated over all ML loss metrics, the poor performance of cmae_2 also impacts the "OS", "agg", "DT" rows of that table. Secondly, the "OS" entries of the two tables show that "SVR lin" gives the smallest (i.e., best) primal gaps, and often the smallest (i.e., best) dual gaps. This is unexpected, because linear SVR yields a PP CSSP. The DT paradigm manages to attain smaller dual gaps, but only in the no FS scenario. Third, from Tab. 5.9 and, especially, from Tab. 5.10, we conclude that, on OS instances, the no FS scenario provides the best primal and dual gaps. We believe that since, in this scenario, the CSSP solution does not need to be completed (heuristically), no further noise — other than the noise that $\bar{p}_{\mathcal{A}}$ already carries with it — is injected into our pipeline. We also see that no clear winner emerges among the heuristics implemented for completing the CSSP solution, nor between the several ML loss metrics tested. Consequently,

set	FS	heur	%feas c**			%feas c_{cpx}	gap_p c**			gap_p c_{cpx}	gap_d c**			gap_d c_{cpx}
			SVR rbf	DT	SVR lin		SVR rbf	DT	SVR lin		SVR rbf	DT	SVR lin	
IS	agg	clst-h1-mean	**99.11**	98.75	91.00	100	8.082E-02	8.141E-02	6.207E-02	5.812E-01	5.407E-02	5.601E-02	7.379E-02	1.018E-01
		clst-h1-min	97.77	98.57	90.37		7.523E-02	8.354E-02	4.458E-02		4.691E-02	5.588E-02	5.432E-02	
		clst-h2	97.50	98.13	87.97		8.874E-02	8.997E-02	9.263E-02		7.436E-02	7.605E-02	1.544E-01	
		noclst-h1-mean	98.22	**99.29**	89.84		**7.486E-02**	**7.935E-02**	4.465E-02		**4.648E-02**	**5.353E-02**	5.845E-02	
		noclst-h1-min	97.77	98.57	90.37		7.512E-02	8.354E-02	4.458E-02		4.692E-02	5.588E-02	5.432E-02	
		noclst-h2	97.33	98.48	**91.44**		8.334E-02	7.988E-02	**4.385E-02**		6.060E-02	6.853E-02	**5.012E-02**	
	gentle	clst-h1-mean	**98.66**	**95.90**	89.22		**8.001E-02**	9.776E-02	**7.191E-02**		4.835E-02	**5.381E-02**	8.343E-02	
		clst-h1-min	**98.66**	95.72	89.13		8.208E-02	9.805E-02	7.210E-02		**4.662E-02**	5.392E-02	8.445E-02	
		clst-h2	97.77	95.37	88.59		8.050E-02	**9.208E-02**	7.232E-02		4.920E-02	5.909E-02	7.968E-02	
		noclst-h1-mean	**98.66**	95.72	89.13		8.208E-02	9.805E-02	7.210E-02		**4.662E-02**	5.392E-02	8.445E-02	
		noclst-h1-min	**98.66**	95.72	89.13		8.208E-02	9.805E-02	7.210E-02		**4.662E-02**	5.392E-02	8.445E-02	
		noclst-h2	97.68	95.28	**90.02**		8.452E-02	9.976E-02	7.212E-02		4.868E-02	5.930E-02	**7.899E-02**	
	no	no	**97.77**	**96.17**	**90.29**		**1.362E-01**	**7.993E-02**	**5.885E-02**		**9.526E-02**	**5.015E-02**	**8.766E-02**	
OS	agg	clst-h1-mean	**91.01**	**90.48**	87.83	100	1.898E-01	1.144E-01	**4.060E-02**	4.633E-01	9.815E-02	1.167E+00	7.573E-02	8.121E-02
		clst-h1-min	88.89	88.89	85.71		1.767E-01	1.255E-01	4.973E-02		8.121E-02	1.192E+00	5.672E-02	
		clst-h2	87.04	88.10	**89.15**		2.142E-01	**1.049E-01**	4.598E-02		1.274E-01	1.342E+00	1.878E-01	
		noclst-h1-mean	89.42	90.21	85.71		1.853E-01	1.054E-01	4.998E-02		**8.072E-02**	**1.155E+00**	5.333E-02	
		noclst-h1-min	88.89	88.89	85.71		**1.767E-01**	1.255E-01	4.973E-02		8.121E-02	1.192E+00	5.672E-02	
		noclst-h2	88.62	89.68	87.30		2.064E-01	1.122E-01	4.908E-02		8.712E-02	1.281E+00	**5.275E-02**	
	gentle	clst-h1-mean	**91.80**	**91.53**	86.51		1.537E-01	**9.730E-02**	4.592E-02		9.910E-02	7.522E-02	5.636E-02	
		clst-h1-min	**91.80**	90.74	86.24		1.442E-01	1.026E-01	4.595E-02		**9.686E-02**	**7.446E-02**	5.625E-02	
		clst-h2	89.42	91.01	85.71		**1.398E-01**	1.145E-01	**4.353E-02**		1.020E-01	8.225E-02	5.710E-02	
		noclst-h1-mean	**91.80**	90.74	86.24		1.442E-01	1.026E-01	4.595E-02		**9.686E-02**	**7.446E-02**	5.625E-02	
		noclst-h1-min	**91.80**	90.74	86.24		1.442E-01	1.026E-01	4.595E-02		**9.686E-02**	**7.446E-02**	5.625E-02	
		noclst-h2	89.68	91.27	**87.57**		1.443E-01	1.072E-01	4.512E-02		9.907E-02	7.904E-02	**5.492E-02**	
	no	no	**90.74**	**94.97**	**84.92**		**1.122E-01**	**7.305E-02**	**4.825E-02**		**7.311E-02**	**5.586E-02**	**9.054E-02**	

Table 5.10: CSSP solution and CPLEX's default configuration performances, in terms of $p_{\mathcal{A}}$ gap, by instance set and CSSP solution completion heuristic

when the approach is deployed on an instance $\bar{\pi} \in \Pi$ and if the available computational resources allow it, we reckon that it might be useful to try solving different versions of the CSSP in parallel, to test different ML loss metrics, completion heuristics and even FS scenarios.

5.6 Conclusions

The methodology presented in this chapter combines ML and MP techniques to solve the ACP for optimization solvers. All in all, our computational results with CPLEX, deployed to solve a hard MILP problem, show that the approach is very promising. In fact, the configurations that it provides typically have better primal/dual gaps than those achieved by the default, PI CPLEX configuration. In a small fraction of cases, when the configuration produced by the CSSP is used to setup CPLEX, it does not allow it to reach a feasible solution (Tab. 5.9 and 5.10). A possible fix to this issue would be to use a performance measure $p_{\mathcal{A}}$ promoting feasibility, instead of the integrality gap. It is interesting to remark that using FS techniques is conducive to much easier CSSP formulations, although this can sometimes affects the quality of the accompanying recommender. However, the most important choice when implementing our approach is that of a ML paradigm for the K-EP: the tables of Sec. 5.5 show that some predictors may scale more efficiently to diverse/larger configuration spaces than others, while preserving high generalization capabilities. On our dataset, DTs generally outperformed SVR in this regard. Overall, since choices taken at any point in the pipeline affect its

final outcome (i.e., the repommended configuration c^{**}_{cssp}), a number of details have to be carefully considered for the approach to work.

In the future, we plan to extend the approach to other ML paradigms, starting with NNs (due to their powerful approximation capabilities), kernel-based ones (due to the possibility of designing and implementing *ad hoc* kernels for the application at hand) and other tree-based designs (due to the fact that DTs were so efficient in our tests). We will also try to combine several predictors.

Chapter 6

Tuning algorithms: an alternative stance

6.1 Introduction

This chapter presents the PaI approach to the ACP; this chapter was published in [94]. Like the PaO methodology (described in Ch. 5), this is an offline, PI, MP-driven approach, parametrized by an ML paradigm. Its purpose is to select the best parameters for configuring general algorithms, to solve specific instances. In the computational experiments that we conducted, we chose an MP solver as the target algorithm, and employed to optimize the instances of a hard MILP problem. As in the PaO case, the PaI methodology involves two components: the construction of a model $\mathcal{M}$ and the computation of the recommender in the CSSP. However, instead of constructing $\mathcal{M}$ as an ML approximation of $p_{\mathcal{A}}$ — as in the PaO approach — in the PaI variant we learn to select a configuration providing a desired solver performance level. To this end, we use the same training set as in the PaO one, but we treat the performance function as a part of the training input (rather than the output), hence the name "PaI" ("performance as input").

At the end of this chapter, we discuss the computational results presented in [94], with LR (see Sec. 3.6.1) as the ML paradigm of choice, and compare the derived PaI and PaO CSSP implementations. We test them on CPLEX, deployed to solve instances of the HUC problem (see Sec. 5.5.1). We remark that, in [94], we presented a first proof of concept of the PaI approach, and the experiments we illustrate Sec. 6.5 of this chapter are based on a much smaller training set than the one used for the experiments discussed in Ch. 5. Thus, the results of the two chapters will not be compared. A more extensive evaluation, say, based on the same statistics calculated for the PaO methodology in Sec. 5.5, is left for future research.

6.2 Motivation

We discuss the issue of automatically finding a good MP solver configuration for a particular instance of a given problem, and we propose a two-phase approach to solve it. In the first phase of our PaI approach, we use a supervised learning paradigm (see Ch. 3 for an overview of supervised learning) in order to learn the relationships between the features an instance $\pi \in \Pi$, a solver configuration c, and the performance $p_{\mathcal{A}}$ of the solver, configured by c, on π. Notably, we learn an approximation $\bar{\mathfrak{c}}$ of the map introduced in Eq. (4.14), that we report here for convenience:

$$\mathfrak{c} : \Pi \times \mathbb{R} \longrightarrow \mathcal{C}_{\mathcal{A}}\,, \quad \mathfrak{c}(\pi, r) \mapsto c\,.$$

In $\mathfrak{c}$, c is any configuration allowing to obtain a required performance level $r \in \mathbb{R}$ for the instance π. This is a different interpretation of the K-EP than the one used in the PaO variant of our methodology, where we construct an approximation $\bar{p}_{\mathcal{A}}$ of the performance function $p_{\mathcal{A}} : \Pi \times \mathcal{C}_{\mathcal{A}} \longrightarrow \mathbb{R}$ (see Sec. 5.3.1).

A specific difficulty of learning a good solver configuration is that parameter settings may not all be independent; this requires enforcing (hard) constraints on the learning outcome, to determine which configurations are admissible. We address this issue in the second phase of our approach, the CSSP, where we use the learnt information $\bar{\mathfrak{c}}$ to construct and solve an optimization problem having an explicit representation of the dependency/consistency/compatibility constraints on the configuration parameter settings. These constraints ensure feasibility of the produced configuration. The input of the CSSP is an encoding of the learned map, as well as the encoding of the instance to be solved. The objective function of the CSSP is the learned map: optimizing it over the the constraints yields a good solver configuration for the given instance. Our approach is therefore capable of handling configuration spaces having arbitrarily complex logical conditions. This overcomes a weakness in previous learning-based approaches to the ACP, as acknowledged e.g. by [87, 89]. To see how this weakness might adversely impact a solver configuration methodology, consider the following naive approach: learn the map $\mathfrak{c}$ using a supervised learning method, then ask the trained method to output $\mathfrak{c}(\pi, 1)$ (1 being the best possible performance) for a new, unseen instance encoded by π. Unfortunately, this approach would fail over most off-the-shelf supervised learning methodologies, which are natively unable to reliably enforce constraints on the output configuration. As discussed in Sec. 5.3.2, some attempts have been made to overcome this issue in the ACP literature. Other approaches, used in learning-based optimization, try to directly integrate a constrained optimization problem in a neural network, for example embedding it into the gradient computations of the back-propagation pass [60, 193] or into an individual layer [5], or by embedding a Lagrangian relaxation of the problem into the loss training function of an NN. However, these approaches are not

generalizable to any ML algorithm and/or MP; moreover, they have not been applied to address the ACP.

6.3 The construction of $\mathcal{M}$

The preliminary assumptions formulated in Sec. 5.3 about $\mathcal{C}_{\mathcal{A}}$, Π and $p_{\mathcal{A}}$ are also valid in this chapter. The procedures adopted to obtain p_{ml} are the same as those described in Sec. 5.5.2. However, here, the performance function approximated in the K-EP and appearing in the training set is defined as $1 - p_{\mathsf{ml}}$, rather than just p_{ml} as in Ch. 5. Therefore, the best performance is the highest $1 - p_{\mathsf{ml}}$, and the CSSP is a maximization problem, rather than a minimization one. We adopt this definition because we employ LR as the K-EP paradigm of choice in our experiments: changing the approximated performance function is the easiest way to adapt the CSSP formulation to inference problem of the LR (presented in Sec. 3.6.1.2). We do this both for the PaO and PaI variants, for a better comparison, although the performance function being part of the input in the PaI case means the alternative definition of the performance is not a requirement for the PaI CSSP.

The choice of LR for this work is motivated by the fact that: (a) the parameters chosen for automatic configuration are all binary, and LR is a good method for estimating binary values; (b) the performance function p_{ml} has range $[0, 1]$. In general, as seen for the PaO setting, LR can be replaced by other ML paradigms. This changes the technical details of the K-EP and the CSSP, but it does not change the overall approach.

6.3.1 The PaO variant

We want to produce an approximation $\bar{p}_{\mathcal{A}}$ of the performance function. The LR training and inference problem are presented in Sec. 3.6.1; we interpret the symbols introduced there using the entities defined in Sec. 5.5.2 (which addresses the construction of the training set for the PaO methodology). Thus, the training set for learning $\bar{p}_{\mathcal{A}}$ is:

$$\mathcal{S} = \left\{ \left(\pi_i, c_i, 1 - p_{\mathsf{ml}}(\pi_i, c_i)\right) \mid \pi_i \in \mathbb{R}^m,\ c_i \in \mathcal{C}_{\mathcal{A}},\ p_{\mathsf{ml}} \in [0, 1] \right\}. \tag{6.1}$$

Adopting the notation from Sec. 3.2 of Ch. 3, we have that the ML input set is $\mathcal{X} = \mathbb{R}^{m+q}$ (i.e., $\mathcal{X}$ is $\Pi \times \mathcal{C}_{\mathcal{A}}$) and the ML output set is $\mathcal{Y} = [0, 1]$. This setting is represented in the left part of Fig. 6.1.

6.3.2 The PaI variant

Here, we aim at constructing an approximation of the map $\mathfrak{c}$ in Eq. (4.14). To this end, we learn a function $\bar{\mathfrak{c}}(\pi, r)$, which, given an instance and a desired performance in $[0, 1]$, returns the a configuration capable of ensuring that performance. Here, we employ the

same training set in Eq. (6.1), used in the PaO variant, but we have that the ML input set is $\mathcal{X} = \mathbb{R}^m \times [0,1]$ (i.e., $\mathcal{X}$ is $\Pi \times [0,1]$) and the ML output set is $\mathcal{Y} = \mathbb{R}^q$ (i.e., $\mathcal{Y}$ is $\mathcal{C}_\mathcal{A}$). By the definition of $\mathcal{Y}$ in the PaI case, the LR requires multiple output vertices

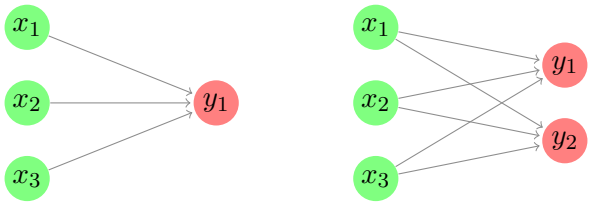

Figure 6.1: Standard (left) and multiple (right) logistic regressions.

(see Fig. 6.1, right), instead of a single one as in the PaO approach (see Fig. 6.1, left), since $q > 1$ in general. This can simply be achieved by considering q standard LRs sharing the same input vertex set.

The training problem of a multiple LR with k output vertices consists of k independent training problems for standard LRs, as in Eq. (3.21).

In fact, a multiple LR on k outputs is equivalent to k standard LRs with training sets $\mathcal{S}^1 = (\mathcal{X}, \mathcal{Y}^1), \ldots, \mathcal{S}^k = (\mathcal{X}, \mathcal{Y}^k)$ where $\mathcal{Y}^h = (y^1_h, \ldots, y^n_h)$ for all $h \leq k$ and $t \leq n$. Note that all these training sets share the same input vector set $\mathcal{X}$. For each $h \leq k$ we define Bernoulli random variables $\mathcal{Y}_h$. Then $\mathsf{P}(\mathcal{Y}_h = 1 \,|\, \mathcal{X} = x)$ (for some $x \in \mathbb{R}^d$ and some $d > 0$) is given by $\tau(x, w^h, b^h)$, where $w^h \in \mathbb{R}^d$ and $b^h \in \mathbb{R}$, for all $h \leq k$. The training problem aims at maximizing the log-likelihood functions $\ln L_{\mathcal{S}^h}(w^h, b^h)$ in Eq. (3.21) of each output vertex $h \leq k$, which yields the objective function $\max \sum_{h \leq k} \ln L_{\mathcal{S}^h}(w^h, b^h)$. Now we note that the optimum of $\sum_h \ln L_{\mathcal{S}^h}(w^h, b^h)$ is achieved by optimizing each term separately, since each term depends on separate decision variables.

It is rather hard to have the LR to produce a bona fide $y \in \{0,1\}^q$, although this might be easily solved by rounding; what is much harder to obtain is that $y \in \mathcal{C}_\mathcal{A}$, which we impose via the CSSP constraints.

6.4 The recommender

In our computational experiments, we compare the performances of the PaI and PaO variants of the CSSP. Clearly, the formulation of the CSSP depends on the output, $\bar{p}_\mathcal{A}$ or $\bar{\mathfrak{c}}$, of the K-EP.

6.4.1 The PaO variant

In this case, the most obvious version of the CSSP would be to just maximize the expected performance over the set of feasible configurations, consistently with the dynamics of the trained LR discussed in Sec. 3.6.1. As remarked above, the CSSP can be

formulated as a maximization problem because we scale p_{ml} into $[1,0]$ and use LR to produce an approximation of $1-p_{\mathsf{ml}}$. This yields the following nonconvex MINLP:

$$\mathsf{CSSP}(\bar{\pi}) \equiv \arg\max_{c\in\mathcal{C}_{\mathcal{A}}} (1+e^{-\langle w^*,(\bar{\pi},c)\rangle+b^*})^{-1}\,. \tag{6.2}$$

As already observed in Sec. 3.6.1, the objective function of Eq. (6.2) is log-concave, which means that

$$\mathsf{CSSP}(\bar{\pi}) \equiv \arg\max_{c\in\mathcal{C}_{\mathcal{A}}} \ln\frac{1}{1+e^{-\langle w^*,(\bar{\pi},c)\rangle+b^*}} \tag{6.3}$$

is a MINLP yielding the same optima as the formulation in Eq. (6.2) but is easier to solve.

We also identified a different interpretation for the CSSP objective, namely that of maximizing the likelihood that any new instance would be matched with a solver configuration and a performance value "as closely as possible" to the associations between $(\bar{\pi},c)$ and $p_{\mathcal{A}}(\bar{\pi},c)$ established during training. In other words, we maximize the likelihood given in Eq. (3.20) as a function of c and r, r being a specific performance value. In order to have the CSSP pick out a high performance, we add a term $+r$ to the objective:

$$\begin{aligned}\mathsf{CSSP}(\bar{\pi}) \equiv \quad & \arg\max_{c\in\mathcal{C}_{\mathcal{A}},r\in[0,1]} \left\{ r\ln\left(\frac{1}{1+e^{-\langle w^*,(\bar{\pi},c)\rangle+b^*}}\right)\right. \\ & \left. +(1-r)\left(1-\ln\left(\tfrac{1}{1+e^{-\langle w^*,(\bar{\pi},c)\rangle+b^*}}\right)\right)+r\right\}.\end{aligned} \tag{6.4}$$

Since the performance measure r is in $[0,1]$ (where 1 corresponds to maximum (excellent) performance), Eq. (6.4) is compatible with LR.

Finally, we propose a third CSSP interpretation, where each alternative r and $1-r$ is weighted by the corresponding conditional probability:

$$\begin{aligned}\mathsf{CSSP}(\bar{\pi}) \equiv \quad & \arg\max_{c\in\mathcal{C}_{\mathcal{A}},r\in[0,1]} \left\{ r\left(\frac{1}{1+e^{-\langle w^*,(\bar{\pi},c)\rangle+b^*}}\right)\right. \\ & \left. +(1-r)\left(1-\left(\tfrac{1}{1+e^{-\langle w^*,(\bar{\pi},c)\rangle+b^*}}\right)\right)\right\}.\end{aligned} \tag{6.5}$$

While Eq. (6.5) is nonconvex, it can still be heuristically solved efficently enough. This formulation is the one yielding the best computational results.

6.4.2 The PaI variant

Here, we consider the multiple LR setting: we correlate a given tuple (π,r), that is, (instance feature, performance), to a parameter configuration c yielding algorithmic performance r. The most direct interpretation of the CSSP in this case is the nonconvex MINLP

$$\mathsf{CSSP}(\bar{\pi}) \equiv \begin{cases} \arg\max\limits_{c\in\mathcal{A},\ r\in[0,1]} & r \\ \forall j\le q & c_j = \frac{1}{1+e^{-\langle (w^j)^*,(\bar{\pi},r)\rangle-b_j^*}}\,, \end{cases} \tag{6.6}$$

where $(w^j)^* \in \mathbb{R}^{m+1}$ is the weight vector of the j-th configuration output. However, this interpretation does not satisfy the feasibility requirements on the c_j.

Proposition 6.4.1. (6.6) *is infeasible, even if the constraint* $r \in [0,1]$ *is relaxed.*

Proof. The constraint of the problem implies for all $j \leq q$

$$\frac{1}{1+e^{-\langle (w^j)^*, (\bar{\pi}, r)\rangle - b_j^*}} \in \{0,1\}\,.$$

However, for any given $(w^j)^*$, b_j^* and $\bar{\pi}$, there is no value of $r \in \mathbb{R}$ which makes the LHS either 0 or 1, hence the result. □

Because of Prop. 6.4.1, here we consider the same interpretation of the CSSP (6.5), presented in Sec. 6.4.1), i.e., the MINLP

$$\begin{aligned} \mathsf{CSSP}(\bar{\pi}) \equiv \quad & \arg \max_{c \in \mathcal{C}_{\mathcal{A}},\ r \in [0,1]} \sum_{j \leq s} \left[c_j \left(\frac{1}{1+e^{-\langle (w^j)^*, (\bar{\pi}, r)\rangle - b_j^*}} \right) \right. \\ & \left. + (1-c_j)\left(1 - \left(\frac{1}{1+e^{-\langle (w^j)^*, (\bar{\pi}, r)\rangle - b_j^*}}\right)\right)\right], \end{aligned} \tag{6.7}$$

which, through simple rearrangements, can be reformulated as

$$\mathsf{CSSP}(\bar{\pi}) \equiv \arg \max_{c \in \mathcal{C}_{\mathcal{A}},\ r \in [0,1]} \sum_{j \leq s} \frac{\left(1-e^{-\langle (w^j)^*, (\bar{\pi}, r)\rangle - b_j^*}\right) c_j - 1}{1+e^{-\langle (w^j)^*, (\bar{\pi}, r)\rangle - b_j^*}}\,. \tag{6.8}$$

6.5 Computational experiments

We compared the PaO and PaI approaches with LR, in the following general set-up:

- we consider 41 MILP instances of the HUC, illustrated in see Sec. 5.5.1;
- the MP solver of choice is CPLEX;
- the supervised ML methodology for building $\mathcal{M}$ is LR;
- the CSSP is a MINLP, that we heuristically solve — using Bonmin — to find good parameter values for CPLEX deployed on 41 instances of the HUC problem.

All experiments were carried out on a single virtual core of a 1.4GHz Intel Core i7 of a MacBook 2017 with 16GB RAM running under macOS Mojave 10.14.6. Our implementations are based on Python 3.7, AMPL 20200430, and Bonmin 1.8.6. We implemented LR as a Keras+TensorFlow [58, 136] NN (see Sec. 3.6.3 with sigmoid activation and a stochastic gradient descent solver minimizing a loss function given by binary cross-entropy (a simple reformulation of the log-likelihood function in Eq. (3.21)).

6.5.1 The algorithmic framework

In this section we give a detailed description of the general algorithmic framework we employ.

1. *Instance encoding.* A set of $m = 55$ features was extracted from each of the 41 problem instances. For more details, see point 1. of Sec. 5.5.2.

2. *Configuration parameters.* We considered a subset of 11 CPLEX parameters (`fpheur`, `rinsheur`, `dive`, `probe`, `heuristicfreq`, `startalgorithm`, `subalgorithm` from `mip.strategy`; `crossover` from `barrier`; and `mircuts`, `flowcovers`, `pathcut` from `mip.cuts`, concerning LSHs, strategies at the root and children nodes of the B&B, cutting planes), each with a varying number of discrete settings (between 2 and 4), which we combined so as to obtain 9216 configurations. We transformed each of these settings into binary form, obtaining $q = 27$ binary parameters. These parameters were chosen because, in our experience, they were reasonably likely to have an impact on the problem we considered. Therefore, our dataset is composed of $41 \times 9216 = 377856$ points.

3. *Separating in-sample and out-of-sample sets.* We randomly choose 11 out of the 41 instances as OS ("out-of-sample"), and let the rest be the IS ("in-sample") set. The IS instances are meant for training purposes, while the OS ones are used to assess the generalization capabilities of the approach.

4. We use the `sklearn.cluster.KMeans` k-means algorithm implementation to cluster the dataset into 5 clusters. We form a training set with 75% of the vectors from each cluster, a validation set with 20%, and a test set with the remaining 5%. By using clustering, we want to ensure that, even after the sampling, the actual distribution of the instances is preserved in all sets.

5. We implement a LR using a `keras.layers.Dense` complete bipartite pair of input/output layers (for $\bar{p}_{\mathcal{A}}$, with $m + q$ input vertices and 1 output vertex; for $\bar{\mathfrak{c}}$, with $m+1$ input vertices and q output vertices), with a sigmoid activation function in the output vertices. We train the LR (by the appropriate training, validation and test sets) using the `keras` stochastic gradient descent optimizer optimizing the binary cross-entropy loss function, which corresponds to minimizing the negative of Eq. (3.21). Then, for further use in the different CSSP formulations, we save:

   - (w^*, b^*), with $w \in \mathbb{R}^{m+q}$ and $b \in \mathbb{R}$, for K-EP-PaO;
   - $(\,(w^j)^*, b_j^*\,)$, $\forall j \leq q$, with $w^j \in \mathbb{R}^{m+1}$ and $b_j \in \mathbb{R}$, for K-EP-PaI.

6. For each OS instance (feature vector) $\bar{\pi}$, we perform the following actions:

(a) we establish a link from Python to AMPL via `amplpy`;

(b) we solve the CSSP corresponding to $\bar{\pi}$ with Bonmin, and retrieve the optimal configuration c^{**}_{cssp};

(c) we retrieve the stored performances $p_{\mathcal{A}}(\bar{\pi}, c^{**}_{\mathsf{cssp}})$ and $p_{\mathcal{A}}(\bar{\pi}, c_{\mathsf{cpx}})$, where c_{cpx} is the default CPLEX configuration;

(d) if $p_{\mathcal{A}}(\bar{\pi}, c^{**}_{\mathsf{cssp}}) > p_{\mathcal{A}}(\bar{\pi}, c_{\mathsf{cpx}})$ we count an improvement;

(e) if $p_{\mathcal{A}}(\bar{\pi}, c^{**}_{\mathsf{cssp}}) \geq p_{\mathcal{A}}(\bar{\pi}, c_{\mathsf{cpx}}) - 0.001$ we count a non-worsening;

(f) we record the performance difference $|p_{\mathcal{A}}(\bar{\pi}, c^{**}_{\mathsf{cssp}}) - p_{\mathcal{A}}(\bar{\pi}, c_{\mathsf{cpx}})|$.

7. We count the number of improvements and non-worsenings over the number of successful Bonmin runs on the CSSP instances; sometimes Bonmin fails on account of the underlying NLP solver, which is why some lines of Table 6.1 consider a total of less than 11 instances.

8. We repeat this process 10 times, from Step 3, and report cumulative statistics of improvements im, non-worsenings nw, performance differences pd, and CPU times

6.5.2 Results

We first conducted experiments on the simple PaO interpretation in Eq. (6.3) of the CSSP. However, this gave very poor results in practice. The formulation in Eq. (6.4), instead, gave better computational results than those obtained optimizing Eq. (6.3), although each CSSP instance took considerably more time to solve with respect to the formulations in Eq.(6.3) and (6.5). The PaO formulation in Eq. (6.5) is the one which gave the best results, and is therefore the only one considered in Tab. 6.1. As for the PaI variant, Tab. 6.1 shows the results of formulation in Eq. (6.8). The table shows im, nw, pd and CPU times. We also report cumulative statistics (sum, mean, standard deviation) for the 10 runs of the algorithmic framework in Sect. 6.5.1 for the PaO and PaI variants. We remark that the "by-run" comparison is only meant for presentation, as the out-of-sample instances involved in each run of PaO and PaI differ.

The results show that the PaO and PaI variants are comparable. PaO improves more, but also worsens more. PaI improves slightly less, but it has three considerable advantages w.r.t. PaO: (i) it does not worsen results more than 60% of the times, which means it can be recommended for usage w.r.t. the default CPLEX configuration; (ii) it is more reliable in terms of standard deviation of improvements and non-worsening; (iii) it is faster.

	im		nw		pd		CPU	
run	PaO	PaI	PaO	PaI	PaO	PaI	PaO	PaI
1	0/08	**5/09**	0/08	**7/09**	0.63	**0.30**	41.44	**30.55**
2	0/11	**4/11**	0/11	**6/11**	**0.42**	0.47	41.62	**29.28**
3	**4/09**	4/10	5/09	**8/10**	**0.08**	0.14	43.06	**33.37**
4	0/09	**5/10**	0/09	**8/10**	0.43	**0.12**	42.65	**35.28**
5	**3/10**	1/10	**7/10**	2/10	**0.08**	0.70	43.30	**31.69**
6	**8/09**	3/11	**8/09**	9/11	0.20	**0.18**	43.69	**28.98**
7	**5/10**	1/11	**8/10**	4/11	**0.05**	0.52	45.54	**30.05**
8	**7/08**	3/09	7/08	**8/09**	0.21	**0.02**	45.49	**31.28**
9	0/09	0/10	0/09	0/10	**0.40**	0.88	43.83	**31.88**
10	**8/10**	5/08	8/10	**7/08**	0.21	**0.10**	43.50	**33.88**
sum	**35/93**	34/99	43/93	**59/99**	**2.69**	3.40	434.12	**316.24**
mean	**0.38**	0.32	0.46	**0.60**	**0.26**	0.34	43.41	**31.62**
stdev	0.36	**0.20**	0.39	**0.30**	**0.18**	0.27	**1.30**	1.94

Table 6.1: Computational results. Best results are marked in boldface.

6.6 Conclusions

We presented a general two-phase framework for learning good MP solver configurations, subject to logical constraints, using a function estimated from data. We proposed two significantly different variants of the methodology, named Pao and PaI. We tested both of them using LR as the ML paradigm of choice in the K-EP, but using different configurations for the inputs and outputs of the LR. Moreover, we conducted computational experiments to trial our approaches to efficiently solve a problem arising in the scheduling of hydro-electric generators. Both variants showed promise, although the PaI one appeared to be preferable for several reasons. We remark that these encouraging results were obtained with a relatively small number of instances.

In the future, we will conduct more extensive experiments, to fully assess the potential of the PaI methodology and compare it to the PaO variant. Moreover, in future works, we are going to investigate our framework using different ML predictors, in the same way we did for the PaO approach.

Part III

Graph embedding and Distance Geometry

Chapter 7

Cycle-based formulations in Distance Geometry

7.1 Introduction

The DGP asks to find a realization of a given simple, edge-weighted graph in a Euclidean space of given dimension K. Its goal is to reconstruct the position of the graph vertices from the (known) edge weights, such that: a) each edge is realized as a straight segment of length equal (or as close as possible) to the edge weights, and b) the distance between the corresponding incident vertices is as close as possible to the corresponding edge weight. Formally:

> Distance Geometry Problem (DGP). Given a positive integer K and a simple undirected graph $G = (V, E)$ with an edge weight function $d : E \to \mathbb{R}_{\geq 0}$, establish whether there exists a *realization* of the vertices, i.e., a function $x : V \to \mathbb{R}^K$ such that Eq. (7.1) below is satisfied:
>
> $$\forall \{i, j\} \in E \qquad \|x(i) - x(j)\| = d(\{i, j\}), \tag{7.1}$$
>
> where $x(i) \in \mathbb{R}^K$ for each $i \in V$ and $d(\{i, j\})$ is the weight on edge $\{i, j\} \in E$.

In this chapter, we will write $x(i)$ as x_i and $d(\{i, j\})$ as d_{uv}, for brevity. Although the DGP is given above in the canonical decision form of Eq. (7.1), we consider the corresponding search problem, where one has to actually find the realization x. The DGP is also known as the *graph realization problem* in geometric rigidity [108, 18, 178]. It belongs to a more general class of metric completion and embedding problems [34, 83, 169].

The problem is often modelled as an MP formulation, involving decision variables which determine the position of the vertices in the given Euclidean space. Solution algorithms are generally constructed using local or global nonlinear optimization techniques.

Our contribution is a new modelling technique for the DGP where, instead of deciding vertex positions, formulations decide the length of the segments representing the edges in each cycle in the graph, projected in every dimension. We propose a novel, exact (MP) formulation for the DGP, and a relaxation based on a Eulerian cycle. We observe that, although diverse methodologies exist for solving the DGP, MP-based ones generally allow robust solutions. We compare computational results from protein conformation instances, obtained with stochastic global optimization techniques on the new cycle-based formulation and on the existing edge-based formulation. While edge-based formulations take less time to reach termination, cycle-based formulations are generally better on solution quality measures. An abridged version of this chapter was published in [120]; a longer and updated version has been submitted to a journal.

7.2 Placing the DGP in the thesis

In general, the DGP is an inverse problem which occurs every time one can measure some of the pairwise distances in a set of entities, and needs to establish their position. Further, it is motivated by many scientific and technological applications. The clock synchronization problem, for example, aims at establishing the absolute time of a set of clocks when only the time difference between subsets of clocks can be exchanged [175]. The sensor network localization problem aims at finding the positions of moving wireless sensor on a 2D manifold given an estimation of some of the pairwise Euclidean distances [178, 95, 51]. The Molecular DGP (MDGP) aims at finding the positions of atoms in a protein, given some of the pairwise Euclidean distances [110, 112, 127, 122, 1, 139]. The position of autonomous underwater vehicles cannot be determined via GPS (since the GPS signal does not reach under water), but must rely on distances estimated using sonars: a DGP can then be solved in order to localize the fleet [12].

However, as far as this thesis is concerned, one of the most relevant applications of the DGP is in ML.

Most popular ML methods are limited to the use of "flat" data, typically encoded by vectors. However, in more complex domains, a vectorial representation may not be able to capture the richness of the underlying data. In these cases, one might resort to labelled graphs. Given a data input, graphs can represent both its single components/features — encoding them by vertices — and the relationships between them — encoding them by edges/arcs. This has been done, for instance, in [70], where instances of an optimization problem are first represented by a graph and then used to train a ML predictor. In that work, graph vertices are used to encode the constraint coefficients of the instance, or the variables involved in those constraints; then, the graph-structured data is used as training input for a graph neural network. Instead, in [114], the authors describe a framework for describing, storing and manipulating MP

formulations, providing guidelines to represent the expressions stating constraints and objective functions by a graph.

Some DL paradigms are capable of processing graph-structured inputs. Given an input graph G, their goal is to learn a function $h(G) = (\mathcal{N} \circ \Phi)(G)$, called *transduction*: given an integer $K > 0$, a trasduction is the composition of a map Φ, computing a realization of G in a space $\mathcal{X} \subseteq \mathbb{R}^K$, and an NN, i.e., a function $\mathcal{N} : \mathcal{X} \longrightarrow \mathcal{Y}$, where $\mathcal{Y} \subseteq \mathbb{R}^t$, and $\mathcal{X}, \mathcal{Y}$ are the learning input and output sets, with the notation adopted in Sec. 3.2. In particular, the topology of $\mathcal{N}$ is derived from (or is the same as) that of G, although the real-valued vectors encoding the information at the vertices may be different. See [11] for a survey on the topic, and [168, 141] for a presentation of several DL paradigms for dealing with graph inputs. We have seen that the instances of several problems can be represented by a graph, and solving the DGP amounts to finding a graph embedding. Therefore, one of the motivations for our investigation of the DGP is that it can be instrumental in deploying vector-based ML paradigms on graphs. In fact, solving the DGP (say, by the novel MP formulations that we describe in this chapter) can be seen as an alternative to computing a transduction and, notably, it should provide an efficient way of calculating Φ, compared to computationally heavier alternatives.

Another motivation lies in the fact that, since K is an input of a DGP instance, distance geometry techniques can be used to perform dimensionality reduction (discussed in Sec. 3.5.4 of this thesis) on graph-structured data, by using a small enough K. The realization found by solving the DGP can then be used as at training set input for a vector-based ML paradigm. See [119] for a survey on the issue of mapping graphs to vectors, and its relation with MP and ML (in particular, dimensionality reduction techniques). Moreover, see [117] for an application of the DGP to a vector-based ML paradigm for natural language processing.

Although the applications of the DGP in ML are of considerable interest in relation to this thesis, there was not enough time during my PhD to conduct computational experiments in this direction. The idea of applying DGP formulations to the processing of training data for an ML predictor, and the execution of test to assess the effectiveness of this strategy against more mainstream methods, is left for future research.

7.3 Generalities

In its most general form, the DGP might be parametrized over any norm [46]. In practice, the ℓ_2 norm is the most usual choice [127], and will also be employed in this paper. The DGP with the ℓ_2 norm is sometimes called the Euclidean DGP (EDGP). For the EDGP, Eq. (7.1) is often reformulated to:

$$\forall \{i, j\} \in E \qquad \|x_i - x_j\|_2^2 = d_{ij}^2 \,, \tag{7.2}$$

which is a system of quadratic polynomial equations with no linear terms [122, §2.4].

The DGP is weakly **NP**-hard even when restricted to simple cycle graphs (by reduction from Partition) and strongly **NP**-hard even when restricted to integer edge weights in $\{1, 2\}$ in general graphs (by reduction from 3Sat) [167]. It is in **NP** if $K = 1$ but not known to be in **NP** if $K > 1$ for general graphs [16], which is an interesting open question [123].

There are many approaches to solving the DGP. Generally speaking, application-specific solution algorithms exploit some of the graph structure, whenever it is induced by the application. For example, a condition often asked when reconstructing the positions of sensor networks is that the realization should be unique (as one would not know how to choose between multiple realizations), a condition called *global rigidity* [41]. This condition can, at least generically, be ensured by a specific graph rigidity structure of the unweighted input graph. For protein structures, on the other hand, which are found in nature in several isomers, one is sometimes interested in finding all (incongruent) realizations of the given protein graph [111, 145, 124]. Since such graphs are rigid, one can devise an algorithm (called Branch-and-Prune) which, following a given vertex order, branches on reflections of the position of the next vertex, which is computed using trilateration [125, 122]. In absence of any information on the graph structure, however, one can resort to Mathematical Programming (MP) formulations and corresponding solvers [128, 37, 50].

The MP formulation which is most often used reformulates Eq. (7.2) to the minimization of the sum of squared error terms:

$$\min_x \sum_{\{i,j\}\in E} (\|x_i - x_j\|_2^2 - d_{ij}^2)^2 \,. \tag{7.3}$$

This formulation describes an unconstrained polynomial minimization problem. The polynomial in question has degree 4, is always nonnegative, and generally nonconvex and multimodal. The decision variables are represented by a $n \times K$ rectangular matrix x such that x_{ik} is the k-th component of the vector x_i, which gives the position in $\mathbb{R}^K$ of vertex $i \in V$. Each solution $x^* \in \mathbb{R}^{nK}$ having global minimum value equal to zero is a realization of the given graph. Solutions with small objective function value represent approximate solutions. Because of the nonconvexity of the formulation and the hardness of the problem, Eq. (7.3) is not usually solved to guaranteed ε-optimality (e.g. using a sB&B approach [151]); rather, heuristic approaches, such as MS [121, 109], VNS [126], or relaxation-based heuristics [50, 130] may be used.

As far as we know, all existing MP formulations for the EDGP are based on the incidence of edges and vertices. In this paper we discuss a new MP formulation for the EDGP based on the incidence of cycles and edges instead, a relaxation based on Eulerian cycles, and a computational comparison with Eq. (7.3).

7.4 Some existing MP formulations

In this short section we give a minimal list of typical variants of Eq. (7.3) in order to motivate the claim that the cycle-based formulation of the DGP discussed in this paper is new. Of course, only a complete enumeration of DGP formulations in the literature could substantiate this claim. But even this short list shows that the typical modelling approach for the DGP is direct: namely, decision variables encode the realization of each vertex as a vector in $\mathbb{R}^K$. Many more formulations of the DGP and its variants, all corresponding to this criterion, are given in [109, 128, 37].

The closest variant of Eq. (7.3) simply adds a constraint ensuring that the centroid of all of the points in the realization is at the origin. This removes the degrees of freedom given by translations:

$$\begin{aligned} \min_{x} \quad & \sum_{\{i,j\}\in E} (\|x_i - x_j\|_2^2 - d_{ij}^2)^2 \\ \forall k \le K \quad & \sum_{i\in V} x_{ik} = 0\,. \end{aligned} \tag{7.4}$$

This formulation describes a linearly constrained polynomial minimization problem. Like Eq. (7.3), the polynomial in Eq. (7.4) has degree 4, is always nonnegative, and is generally nonconvex and multimodal.

Another small variant of Eq. (7.4) is achieved by adding range bounds to the the realization variables x; generally valid (but slack) bound values can be set to $\pm\frac{1}{2}\sum_{\{u,v\}\in E} d_{uv}$. This corresponds to the worst case of a single path being arranged in a straight line with unknown orientation.

Another possible formulation, derived again from Eq. (7.3), is obtained by replacing the squared error with absolute value errors (whose positive and negative parts are encoded by s^+, s^-). This yields the following formulation:

$$\begin{aligned} \min_{s,x} \quad & \sum_{\{i,j\}\in E} (s_{ij}^+ + s_{ij}^-) \\ \forall \{i,j\} \in E \quad & \|x_i - x_j\|_2^2 = d_{ij}^2 + s_{ij}^+ - s_{ij}^- \\ \forall \{i,j\} \in E \quad & s_{ij}^+, s_{ij}^- \ge 0\,. \end{aligned} \tag{7.5}$$

Note that, again, each solution s^*, x^* with zero optimal objective value makes x^* an encoding of a realization of the given graph. Thus, global optima are preserved by this reformulation, while local optimal may differ.

Yet another reformulation derived from replacing squared errors with absolute values consists in observing that the "plus" and "minus" parts of each absolute value term correspond to a convex and concave function. This yields a formulation called *push-and-pull*, since the objective pulls adjacent vertices apart, while the constraint push them back together:

$$\begin{aligned} \max_{x} \quad & \sum_{\{i,j\}\in E} \|x_i - x_j\|_2^2 \\ \forall \{i,j\} \in E \quad & \|x_i - x_j\|_2^2 \le d_{ij}^2\,. \end{aligned} \tag{7.6}$$

Eq. (7.6) is a quadratically constrained QP with concave objective and convex constraints. It was used within a Multiplicative Weights Update algorithm for the DGP in [37, 140], as well as a basis for Semidefinite Programming and Diagonally Dominant Programming relaxations [50, 130]. It can be shown that all constraints are active at global optima, which therefore correspond to realizations of the given graph [140].

7.5 A new formulation based on cycles

In this section we propose a new formulation for the EDGP, based on the fact that the quantities $x_{ik} - x_{jk}$ sum up to zero over all edges of any cycle in the given graph for each dimensional index $k \leq K$. This idea was used in [167] for proving weak **NP**-hardness of the DGP on cycle graphs. For a subgraph H of a graph $G = (V, E)$, we use $V(H)$ and $E(H)$ to denote vertex and edge set of H explicitly; given a set F of edges we use $V(F)$ to denote the set of incident vertices. Let $m = |E|$ and $n = |V|$. For a mapping $x : V \to \mathbb{R}^K$ we denote by $x[U]$ the restriction of x to a subset $U \subseteq V$.

Lemma 7.5.1. *Given an integer $K > 0$, a simple undirected weighted graph $G = (V, E, d)$ and a mapping $x : V \to \mathbb{R}^K$, then for each cycle C in G, each orientation of the edges in C given by a closed trail $W(C)$ in the cycle, and each $k \leq K$ we have*

$$\sum_{(i,j) \in W(C)} (x_{ik} - x_{jk}) = 0 \,. \tag{7.7}$$

Proof. We renumber the vertices in $V(C)$ to $1, 2, \ldots, \gamma = |V(C)|$ following the walk order in $W(C)$. Then Eq. (7.7) can be explicitly written as

$$\begin{aligned} (x_{1k} - x_{2k}) + (x_{2k} - x_{3k}) + \cdots + (x_{\gamma k} - x_{1k}) &= \\ = x_{1k} - (x_{2k} - x_{2k}) - \cdots - (x_{\gamma k} - x_{\gamma k}) - x_{1k} &= 0 \,, \end{aligned}$$

as claimed. □

We introduce new decision variables y_{ijk} replacing the terms $x_{ik} - x_{jk}$ for each $\{i, j\} \in E$ and $k \leq K$. Eq. (7.2) then becomes:

$$\forall \{i, j\} \in E \qquad \sum_{k \leq K} y_{ijk}^2 = d_{ij}^2 \,. \tag{7.8}$$

We remark that for the DGP with other norms this constraint changes. For the ℓ_1 or ℓ_∞ norms, for example, we would have:

$$\forall \{i, j\} \in E \quad \sum_{k \leq K} |y_{ijk}| = d_{ij} \quad \text{or} \quad \max_{k \leq K} |y_{ijk}| = d_{ij} \,. \tag{7.9}$$

Next, we adjoin the constraints on cycles

$$\forall k \leq K, C \subset G \quad \left(C \text{ is a cycle} \Rightarrow \sum_{\{i,j\} \in E(C)} y_{ijk} = 0 \right). \tag{7.10}$$

We also note that the feasible value of a y_{ijk} variable is the (oriented) length of the segment representing the edge $\{i, j\}$ projected on the k-th coordinate. We can therefore infer bounds for y as follows:

$$\forall k \leq K, \{i, j\} \in E \quad -d_{ij} \leq y_{ijk} \leq d_{ij} \,. \tag{7.11}$$

Although Eq. (7.11) are not necessary to solve the cycle formulation, they may improve performance of sB&B algorithms [184, 151] as well as of various "matheuristics" [129], as well as allow an exact linearization of variable products, should a y variable occur in a product with a binary variable in some DGP variant.

We now state our main result, i.e. that Eq. (7.8) and (7.10) are a valid MP formulation for the EDGP.

Theorem 7.5.2. *There exists a vector $y^* \in \mathbb{R}^{Km}$ which satisfies Eq.* (7.8) *and* (7.10)*, parametrized on K, G, if and only if (K, G) is a YES instance of the EDGP.*

The proof argues by recursion on a graph decomposition of G that a certain linear system related to the cycles of G (see Eq. (7.12) below) has a solution if and only if the given DGP instance is YES. We shall construct the proof by steps. The first step defines the graph decomposition.

Given a graph $G = (V, E)$ and a subset $U \subset V$, the subgraph $G[U]$ *induced* by U is the graph $(U, \{\{u, v\} \in E \mid u, v \in U\})$. With a slight abuse of notation we denote the vertices of a graph G' by $V(G')$ and its edges by $E(G')$. We let $\gamma(G)$ be the number of connected components of G. A vertex v of G with the property that $\gamma(G[V \setminus \{v\}]) > \gamma(G)$ is called a *cut vertex*. A graph G is *biconnected* if there is a simple cycle in G incident to any pair of distinct vertices of G.

Lemma 7.5.3. *$G = (V, E)$ is biconnected if and only if it is connected and has no cut vertices.*

Proof. Suppose G is biconnected with a cut vertex v: then the removal of v from G yields two separate connected components G_1, G_2. Let $u_1 \in V(G_1)$ and $u_2 \in V(G_2)$. Since $u_1, u_2 \in V$ and G is biconnected, there is a simple cycle in G incident to u_1, u_2, consisting of two vertex-disjoint simple paths p_1 and p_2 from u_1 to u_2. Since the removal of v can break at most one of these paths (by vertex disjointness), the other path shows that G_1, G_2 are not disconnected, against the assumption. So G cannot have cut vertices. Conversely, if G is connected and has a cut vertex v, then any pair of paths from u_1 to u_2 must necessarily pass through v, which means that they are not vertex disjoint, which implies that there is no cycle between u_1 and u_2 in G, which in turn implies that G is not biconnected. □

We now define a graph decomposition based on removal of a single cut vertex.

Definition 7.5.4. *A 1-decomposition of a graph $G = (V, E)$ is a set of subgraphs $G_1, \ldots, G_r$ (where $r \in \mathbb{N}$ with $r \geq 1$) of G such that:*

(a) G_i is either biconnected or a tree for all $i \leq r$;

(b) $\bigcup_{i \leq r} E(G_i) = E$;

(c) for any $i < j \leq r$ the intersection $V(G_i) \cap V(G_j)$ either has zero cardinality or it consists of a single cut vertex of G.

A 1-decomposition of G is nontrivial if $r > 1$. A graph G is 1-decomposable if it has a nontrivial 1-decomposition.

Lemma 7.5.5. *A connected graph $G = (V, E)$ is 1-decomposable if and only if it has a cut vertex.*

Proof. Suppose $\mathcal{C} = \{C_1, \ldots, C_t\}$ is a nontrivial 1-decomposition of G. By Def. 7.5.4 and since G is connected, it follows that G must have at least one cut vertex and as many as $|\mathcal{C}| - 1$. Conversely, supposing that G has a cut vertex would yield a nontrivial 1-decomposition by Def. 7.5.4, i.e. G is 1-decomposable. □

Corollary 7.5.6. *No simple graph consisting of a single cycle is 1-decomposable.*

Proof. Since a cycle is biconnected, by Lemma 7.5.3 it cannot have a cut vertex, hence its only possible 1-decomposition is trivial. □

Corollary 7.5.7. *Let G be 1-decomposable, with decomposition $\mathcal{G} = \{G_1, \ldots, G_r\}$, and C be a cycle in G. Then there is an index $i \leq r$ s.t. C is a subgraph of G_i.*

Proof. Consider there were two subgraphs G_i, G_j in $\mathcal{G}$ both incident to the edges of C. Then there is a nontrivial path p in C, with at least two edges, joining a vertex u in G_i to a vertex v in G_j. Therefore there must be a cut vertex of G on p, which implies that there is a cut vertex in C, which is impossible by Cor. 7.5.6. □

Corollary 7.5.8. *No biconnected graph G is 1-decomposable.*

Proof. By Lemma 7.5.3, if G is biconnected it cannot have a cut vertex, therefore any 1-decomposition must necessarily be trivial. □

Proposition 7.5.9. *Any simple graph $G = (V, E)$ has a 1-decomposition consisting of biconnected subgraphs and tree subgraphs.*

Proof. We prove this result by induction on the number β of biconnected subgraphs in a 1-decomposition $\mathcal{C} = \{G_1, \ldots, G_r\}$ of G for some $r \in \mathbb{N}$. We first deal with the base case, where $\beta = 0$. We claim that G must be a tree: supposing G has a cycle G', by Cor. 7.5.6 and part ((c)) of Def. 7.5.4, G' must be one of the $G_1, \ldots, G_r$. But then $\beta = 1$

against the assumption. Therefore, the trivial 1-decomposition $\mathcal{C} = \{G\}$ is a valid 1-decomposition of G. We now tackle the induction step. Consider the largest biconnected subgraph B of G: then $\tilde{G} = G[V \setminus V(B)]$ has one fewer biconnected components than G, so, by induction, $\tilde{G}$ has a 1-decomposition $\mathcal{D}' = \{G'_1, \ldots, G'_{t-1}\}$ for some $t \in \mathbb{N}$ with $t > 1$. We prove that $\mathcal{D} = \mathcal{D}' \cup \{B\}$ is a valid 1-decomposition of G. Condition ((a)) is verified since $\mathcal{D}'$ is a valid 1-decomposition by induction, and B is biconnected; condition ((b)) is verified since the union of the graph in $\mathcal{D}$ is $\mathcal{G}$ by construction; for condition ((c)), suppose there is $i < t$ s.t. $|V(G_i) \cap V(B)| \geq 2$: this means there are two distinct vertices u, v in both $V(G_i)$ and $V(B)$. Since G_i is connected, there must be a path p from u to v in G_i, hence $G[B \cup V(p)]$ is a biconnected graph larger than B. But B was assumed to be largest, so this is not possible, and ((c)) holds, which concludes the proof. □

The second step proves the easier ($\Leftarrow$) direction of Thm. 7.5.2.

Proposition 7.5.10. *For any YES instance (K, G) of the EDGP there is a vector $y^* \in \mathbb{R}^{Km}$ which satisfies Eq. (7.8) and (7.10).*

Proof. Assume that (K, G) is a YES instance of the EDGP. Then G has a realization $x^* \in \mathbb{R}^{nK}$ in $\mathbb{R}^K$. We define $y^*_{ijk} = x^*_{ik} - x^*_{jk}$ for all $\{i, j\} \in E$ and $k \leq K$. Since x^* is a realization of G, by definition it satisfies Eq. (7.2), and, by substitution, Eq. (7.8). Moreover, any realization of G satisfies Eq. (7.7) over each cycle by Lemma 7.5.1. Hence, by replacement, it also satisfies Eq. (7.10). □

In the third step, we lay the groundwork towards the more difficult ($\Rightarrow$) direction of Thm. 7.5.2. We proceed by contradiction: we assume that (K, G) is a NO instance of the EDGP, and suppose that Eq. (7.8) and (7.10) have a nonempty feasible set Y. For every $y \in Y$ we consider the K linear systems

$$\forall \{i, j\} \in E \quad x_{ik} - x_{jk} = y_{ijk}\,, \tag{7.12}$$

for each $k \leq K$, each with n variables and m equations. We square both sides then sum over $k \leq K$ to obtain

$$\forall \{i, j\} \in E \quad \sum_{k \leq K} (x_{ik} - x_{jk})^2 = \sum_{k \leq K} y^2_{ijk}\,. \tag{7.13}$$

By Eq. (7.8) we have

$$\sum_{k \leq K} y^2_{ijk} = d^2_{ij}\,, \tag{7.14}$$

whence follows Eq. (7.2), contradicting the assumption that the EDGP is NO. So we need only show that there is a solution x to Eq. (7.12) for any given $y \in Y$. To this effect, we shall exploit the 1-decomposition of G into biconnected graphs and trees derived in Prop. 7.5.9. First, though, we have to show that Eq. (7.12) has a solution if $Y \neq \varnothing$ in the "base cases" of the 1-decomposition, namely trees and biconnected graphs.

Lemma 7.5.11. *Let $G = (V, E)$ be a tree, and $Y \neq \varnothing$ satisfying Eq.* (7.8) *and* (7.10)*. Then Eq.* (7.12) *has a solution for every $k \leq K$.*

Proof. Let M^k be the matrix of each system Eq. (7.12), for $k \leq K$; we aim at proving that M^k and (M^k, y^k) have the same rank, where $y^k = (y_{uvk} \mid \{u, v\} \in E)$, and that this rank is full. We proceed by induction on the size $|E|$ of the tree. The base case, where $|E| = 1$ and G consists of a single edge $\{u, v\}$, yields $M^k = (1, -1)$ with rank 1 for each $k \leq K$. By inspection, (M^k, y_{uvk}) also has rank 1 for any y_{uvk}. Consider a tree G' with one fewer edge (say, $\{u, v\}$) than G, such that $V \smallsetminus V(G') = \{v\}$. Let the corresponding system Eq. (7.12) $\tilde{M}^k = \tilde{y}^k$ satisfy $\mathsf{rank}(\tilde{M}^k) = \mathsf{rank}(\tilde{M}^k, \tilde{y}^k)$, for all $k \leq K$. Then the shape of M^k is:

$$M^k = \begin{pmatrix} \tilde{M}^k & 0 \\ e_u & -1 \end{pmatrix},$$

where $e_u = (0, \ldots, 0, 1_u, 0, \ldots, 0)$. This shows that $\mathsf{rank}(M^k) = \mathsf{rank}(\tilde{M}^k) + 1$, that this rank is full, and hence also that $\mathsf{rank}(M^k) = \mathsf{rank}((M^k, y^k))$, as claimed. □

Lemma 7.5.12. *Let $G = (V, E)$ be biconnected, and $Y \neq \varnothing$ satisfying Eq.* (7.8) *and* (7.10)*. Then Eq.* (7.12) *has a solution for every $k \leq K$.*

Proof. We proceed by induction on the simple cycles of G. For the base case, we consider G to be a graph consisting of a single cycle, with corresponding y satisfying Eq. (7.8) and (7.10). Since G is a cycle, it has the same number of vertices and edges, say q. This implies that, for any fixed $k \leq K$, Eq. (7.12) is a linear system $M^k x = y^k$ (where $y^k = (y_{uvk} \mid \{u, v\} \in E)$ with a $q \times q$ matrix:

$$M^k = \begin{pmatrix} 1 & -1 & & & \\ & 1 & -1 & & \\ & & 1 & \ddots & \\ & & & \ddots & -1 \\ -1 & & & & 1 \end{pmatrix}. \tag{7.15}$$

By Eq. (7.7) and by inspection of Eq. (7.15) it is clear that $\mathsf{rank}(M^k) = q - 1$: then Eq. (7.10) ensures that $\mathsf{rank}((M^k, y^k)) = \mathsf{rank}(M^k)$, and therefore that Eq. (7.12) has a solution.

We now tackle the induction step. The incidence vectors in E of the cycles of any graph are a vector space of dimension $m - n + 1$ over the finite field $\mathbb{F}_2 = \{0, 1\}$ [173]. We consider a fundamental cycle basis $\mathcal{B}$ of G (see Sec. **??**). We assume that (a) G' is a union of fundamental cycles in $\mathcal{B}' \subsetneq \mathcal{B}$, for which Eq. (7.12) has a solution x' by the induction hypothesis, and (b) that C is another fundamental cycle in $\mathcal{B} \smallsetminus \mathcal{B}'$, with a solution x^C of Eq. (7.12) which exists by the base case. We aim at proving that

Eq. (7.12) has a solution for $G' \cup C$. Since G is biconnected, the induction can proceed by ear decomposition [135], which means that G' is also biconnected, and that C is such that $E(G') \cap E(C) = F$ is a nonempty path in G'.

By Eq. (7.10) applied to C, we have

$$\forall k \leq K \quad \sum_{\{i,j\} \in C} y_{ijk} = 0\,. \tag{7.16}$$

Since x' satisfies Eq. (7.12) by the induction hypothesis,

$$\forall k \leq K, \{i,j\} \in F \quad x'_{ik} - x'_{jk} = y_{ijk}\,. \tag{7.17}$$

We replace Eq. (7.17) in Eq. (7.16), obtaining

$$\forall k \leq K \quad \sum_{\{i,j\} \in F} (x'_{ik} - x'_{jk}) = - \sum_{\{i,j\} \in E(C) \smallsetminus F} y_{ijk}\,. \tag{7.18}$$

Moreover, x^C also satisfies Eq. (7.12) over C, hence we can replace the right hand side of Eq. (7.18) with the corresponding terms in $x^C_{ik} - x^C_{jk}$, to get

$$\forall k \leq K \quad \sum_{\{i,j\} \in F} (x'_{ik} - x'_{jk}) + \sum_{\{i,j\} \in E(C) \smallsetminus F} (x^C_{ik} - x^C_{jk}) = 0. \tag{7.19}$$

We now fix x', and aim at modifying x^C so that: (a) x^C matches x' on $V(F)$, (b) the modified x^C is still a solution of Eq. (7.12) on C. We set x^C_{ik} to x'_{ik} for each $i \in V(F)$, and consider the resulting linear system Eq. (7.12) given by M^k, as in Eq. (7.15), for each $k \leq K$, where we assume without loss of generality that $V(F) = \{1, \ldots, r\}$ and $V(C) = \{r+1, \ldots, s\}$:

$$\left.\begin{array}{rrrrrrrrrrl}
x'_{1k} & - x'_{2k} & & & & & & = & y_{12k} & (1) \\
 & x'_{2k} & - x'_{3k} & & & & & = & y_{23k} & (2) \\
 & \ddots & \ddots & & & & & \vdots & \vdots & \vdots \\
 & & x'_{rk} & - x^C_{r+1,k} & & & & = & y_{r,r+1,k} & (r) \\
 & & & x^C_{r+1,k} & - x^C_{r+2,k} & & & = & y_{r+1,r+2,k} & (r{+}1) \\
 & & & & \ddots & \ddots & & \vdots & \vdots & \vdots \\
 & & & & & x^C_{s-1,k} & - x^C_{sk} & = & y_{s-1,s,k} & (s{-}1) \\
- x'_{1k} & & & & & & x^C_{sk} & = & y_{1sk}\,. & (s)
\end{array}\right\} \tag{7.20}$$

The equations from (1) to $(r-1)$ in Eq. (7.20) are satisfied by the induction hypothesis since they only depend on x', so we can remove them from the system and assume x' to be constant. We are left with:

$$\left.\begin{array}{rrrrrl}
- x^C_{r+1,k} & & & = & y_{r,r+1,k} - x'_{rk} & (r) \\
x^C_{r+1,k} & - x^C_{r+2,k} & & = & y_{r+1,r+2,k} & (r{+}1) \\
\ddots & \ddots & & \vdots & \vdots & \vdots \\
 & x^C_{s-1,k} & - x^C_{sk} & = & y_{s-1,s,k} & (s{-}1) \\
 & & x^C_{sk} & = & y_{1sk} + x'_{1k}\,. & (s)
\end{array}\right\} \tag{7.21}$$

Summing up the left hand sides of Eq. (7.21), we obtain

$$\begin{aligned} & -x^C_{r+1,k} + (x^C_{r+1,k} - x^C_{r+2,k}) + \cdots + (x^C_{s-1,k} - x^C_{sk}) + x^C_{sk} \\ = \; & (-x^C_{r+1,k} + x^C_{r+1,k}) + \cdots + (-x^C_{sk} + x^C_{sk}) = 0 \end{aligned}$$

for all $k \leq K$, so the $(s-r+1) \times (s-r+1)$ matrix $\bar{M}^k$ of the k-th linear system Eq. (7.21) has rank $\leq s-r$. On the other hand, eliminating the first or last row makes it clear by inspection that the rest of the rows are linearly independent; therefore the rank of $\bar{M}^k$ is exactly $s-r$. Summing up the components of the right hand side vector $\bar{y}^k$ of Eq. (7.21), we obtain

$$\begin{aligned} \chi &= -x'_{rk} + y_{r,r+1,k} + y_{r+1,r+2,k} + \cdots + y_{s-1,s,k} + y_{1sk} + x'_{1k} \\ &= (x'_{1k} - x'_{rk}) + \sum_{\{i,j\} \in E(C) \smallsetminus F} y_{ijk} \,. \end{aligned}$$

We remark that

$$\begin{aligned} x'_{1k} - x'_{rk} &= (x'_{1k} - x'_{2k}) + (x'_{2k} - x'_{3k}) + \cdots + (x'_{r-1,k} + x'_{rk}) \\ &= \sum_{\{i,j\} \in F} (x'_{ik} - x'_{jk}) = \sum_{\{i,j\} \in F} y_{ijk} \end{aligned}$$

since x' satisfies Eq. (7.12) by the induction hypothesis. Therefore,

$$\chi = \sum_{\{i,j\} \in F} y_{ijk} + \sum_{\{i,j\} \in E(C) \smallsetminus F} y_{ijk} = \sum_{\{i,j\} \in E(C)} y_{ijk} \,,$$

whence $\chi = 0$ by Eq. (7.16). This implies that $\mathsf{rank}((\bar{M}^k, \bar{y}^k)) = \mathsf{rank}(\bar{M}^k) = s-r$. Therefore, Eq. (7.21) has a solution, which yields the modified x^C with properties (a) and (b) given above. This concludes the induction step and the proof. □

We can finally give the proof of Thm. 7.5.2.

Proof of Thm. 7.5.2. The ($\Leftarrow$) part follows by Prop. 7.5.10. For the ($\Rightarrow$) part, we exploit a 1-decomposition of G into trees and biconnected subgraphs, derive solutions to Eq. (7.12) for each subgraph, and show that the solutions can be easily combined to yield a solution to Eq. (7.12) for the whole graph G.

We assume without loss of generality that G is connected (otherwise each connected component can be treated separately), and consider a 1-decomposition $\mathcal{D} = \{G_1, \ldots, G_r\}$ of G. By Lemmata 7.5.11 and 7.5.12, there exist solutions $x^1, \ldots x^r$ to Eq. (7.12) applied to $G_1, \ldots, G_r$ respectively. Consider the graph

$$\mathscr{D} = (\mathcal{D},\ \{\{i,j\} \mid 1 \leq i \neq j \leq r \wedge |V(G_i) \cap V(G_j)| = 1\}) \,.$$

By Cor. 7.5.7, $\mathscr{D}$ is a tree: otherwise, a cycle in $\mathcal{D}$ would be a contraction of a cycle in G not included in a single G_i, against Cor. 7.5.7. This allows us to reorder $\mathcal{D}$ so that, for each $j > 1$, there is a unique $i < i$ such that $\{i,j\} \in E(\mathscr{D})$.

We remark that, for each $i \leq r$, x^i is a realization of G_i in $\mathbb{R}^K$ by Eq. (7.12)-(7.14). More precisely, x^i is a $|V(G_i)| \times K$ matrix $x^i = (x^i_{\ell k})$ so that $x^i_\ell = (x^i_{\ell 1}, \ldots, x^i_{\ell K})$ is the position of vertex $\ell \in V(G_i)$ in $\mathbb{R}^K$. Note that the realizations $x^1, \ldots, x^r$ can be modified by translations without changing the values of y (by inspection of Eq. (7.12)).

We now construct a solution $\bar{x}$ of Eq. (7.12) for G by induction on $\mathcal{D}$ ordered as described above. For the base case $i = 1$, we fix x^1 in any way (e.g. by taking the centroid of the rows of x^1 to be the origin), and initialize the first $|V(G_1)|$ rows of $\bar{x}$ with those of x^1. For any $i > 1$, we identify the unique predecessor j of i in the order on $\mathcal{D}$. The induction hypothesis ensures the existence of a solution $\bar{x}$ of the union of $G_1, \ldots, G_j$. Consider the cut vertex v in $V(G_j) \cap V(G_i)$ guaranteed by definition of the order on $\mathcal{D}$, and let $\bar{x}_v \in \mathbb{R}^K$ be its position. Then the translation $\tilde{x}^i = x^i - \mathbf{1}(x^i_v - \bar{x}_v)^\top$ yields another valid solution of Eq. (7.12) applied to G^i by translation invariance, and this solution is such that $\tilde{x}^i_v = \bar{x}_v$. Therefore, using the rows of $\tilde{x}^i$, $\bar{x}$ can be extended to a solution of Eq. (7.12) applied to the union of $G_1, \ldots, G_j$ and G^i, as claimed. □

Thm. 7.5.2 can also be interpreted as a polynomial reduction of the EDGP to the problem of finding a solution of Eq. (7.8) and (7.10).

Corollary 7.5.13. *Deciding feasibility of Eq.* (7.8) *and* (7.10) *is* **NP***-hard.*

Proof. By reduction from EDGP using Thm. 7.5.2. □

A remarkable consequence of Thm. 7.5.2 is that it allows a decomposition of the computation of the realization x into two stages: first, solve Eq. (7.8)-(7.10) to find a feasible y^*; then solve

$$\forall k \leq K, \{i, j\} \in E \quad x_{ik} - x_{jk} = y^*_{ijk} \tag{7.22}$$

to find a realization x^*. We note that Eq. (7.22) is just a restatement of Eq. (7.12) universally quantified over k.

Corollary 7.5.14. *Given a solution y^* solving Eq.* (7.8) *and Eq.* (7.10)*, any solution x^* of Eq.* (7.22) *is a valid realization of the EDGP instance (K, G).*

Proof. The feasibility of Eq. (7.22) with the right hand side replaced by a solution y^* of Eq. (7.8) and (7.10) follows directly from Thm. 7.5.2, since if such a y^* exists then the EDGP is feasible. □

The first stage is **NP**-hard by Cor. 7.5.13, while the second stage is tractable, since solving linear systems can be done in polynomial time.

Remark 7.5.15. *Note that Eq.* (7.22) *has Km equations, but its rank may be lower, since there are only Kn variables: in particular, Eq.* (7.22) *may be an overdetermined linear system. The feasibility of this system is guaranteed by Cor. 7.5.14; in particular,*

the steps of the proof of Thm. 7.5.2 imply that Eq. (7.22) *loses rank w.r.t. Km according to the incidence of the edges in the cycles of G. In other words, any solution y' to Eq.* (7.10) *provides a right hand side to Eq.* (7.22) *that makes the system feasible.*

The issue with Thm. (7.5.2) is that it relies on the exponentially large family of constraints Eq. (7.10). While this is sometimes addressed by algorithmic techniques such as row generation, we shall see in the following that it suffices to consider a polynomial set of cycles (which, moreover, can be found in polynomial time) in the quantifier of Eq. (7.10).

7.6 The cycle vector space and its bases

We recall that incidence vectors of cycles (in a Euclidean space having $|E|$ dimensions) form a vector space over a field $\mathbb{F}$, which means that every cycle can be expressed as a weighted sum of cycles in a basis. In this interpretation, a *cycle* in G is simply a subgraph of G where each vertex has even degree: we denote their set by $\mathcal{C}$. This means that Eq. (7.10) is actually quantified over a subset of $\mathcal{C}$, namely the simple connected cycles. Every basis has cardinality $m - n + a$, where a is the number of connected components of G. If G is connected, cycle bases have cardinality $m - n + 1$ [173].

Our interest in introducing cycle bases is that we would like to quantify Eq. (7.10) polynomially rather than exponentially in the size of G. Our goal is to replace "C is any simple connected cycle in $\mathcal{C}$" by "C is a cycle in a cycle basis of G". In order to show that this limited quantification is enough to imply every constraint in Eq. (7.10), we have to show that, for each simple connected cycle $C \in \mathcal{C}$, the corresponding constraint in Eq. (7.10) can be obtained as a weighted sum of constraints corresponding to the basis elements.

Another feature of Eq. (7.10) to keep in mind is that edges are implicitly given a direction: for each cycle, the term for the *undirected* edge $\{i, j\}$ in Eq. (7.10) is $(x_{ik} - x_{jk})$. Note that while $\{i, j\}$ is exactly the same vertex set as $\{j, i\}$, the corresponding term is either positive or not, depending on the direction (i, j) or (j, i). We deal with this issue by arbitrarily directing the edges in E to obtain a set A of arcs, and considering *directed* cycles in the directed graph $\bar{G} = (V, A)$. In this interpretation, the incidence vector of a directed cycle C of $\bar{G}$ is a vector $c^C \in \mathbb{R}^m$ satisfying [179, §2, p. 201]:

$$\forall j \in V(C) \quad \sum_{(i,j)\in A} c^C_{ij} = \sum_{(j,\ell)\in A} c^C_{j\ell}\,. \tag{7.23}$$

A directed circuit D of $\bar{G}$ is obtained by applying the edge directions from $\bar{G}$ to a connected subgraph of G where each vertex has degree exactly 2 (note that a directed circuit need not be strongly connected, although its undirected version is connected).

Its incidence vector $c^D \in \{-1, 0, 1\}^m$ is defined as follows:

$$\forall (i,j) \in A\,, \quad c_{ij}^D \triangleq \begin{cases} 1 & \text{if } (i,j) \in A(D) \\ -1 & \text{if } (j,i) \in A(D) \\ 0 & \text{otherwise}\,, \end{cases}$$

where we have used $A(D)$ to mean the arcs in the subgraph D. In other words, whenever we walk over an arc (i,j) in the natural direction $i \to j$ we let the (i,j)-th component of c^D be 1; if we walk over (i,j) in the direction $j \to i$ we assign a -1, and otherwise a zero.

7.6.1 Constraints over cycle bases

The properties of undirected and directed cycle bases have been investigated in a sequence of papers by many authors, culminating with [179]. We now prove that it suffices to quantify Eq. (7.10) over a directed cycle basis.

Proposition 7.6.1. *Let $\mathcal{B}$ be a directed cycle basis of $\bar{G}$ over $\mathbb{Q}$. Then Eq.* (7.10) *holds if and only if:*

$$\forall k \le K,\, B \in \mathcal{B} \qquad \sum_{(i,j)\in A(B)} c_{ij}^B y_{ijk} = 0\,. \tag{7.24}$$

Proof. Necessity (7.10) $\Rightarrow$ (7.24) follows because Eq. (7.10) is quantified over all cycles: in particular, it follows for any undirected cycle in any undirected cycle basis. Moreover, the signs of all terms in the sum of Eq. (7.24) are consistent, by definition, with the arbitrary edge direction chosen for $\bar{G}$.

Next, we claim sufficiency (7.24) $\Rightarrow$ (7.10). Let $C \in \mathcal{C}$ be a simple cycle, and $\bar{C}$ be its directed version with the directions inherited from $\bar{G}$. Since $\mathcal{B}$ is a cycle basis, we know that there is a coefficient vector $(\gamma_B \mid B \in \mathcal{B}) \in \mathbb{R}^{|\mathcal{B}|}$ such that

$$c^{\bar{C}} = \sum_{B\in\mathcal{B}} \gamma_B c^B\,. \tag{7.25}$$

We now consider the expression

$$\forall k \le K \quad \sum_{B\in\mathcal{B}} \gamma_B \sum_{(i,j)\in A(B)} c_{ij}^B y_{ijk}\,. \tag{7.26}$$

On the one hand, by Eq. (7.25), Eq. (7.26) is identically equal to $\sum_{(i,j)\in A(\bar{C})} c_{ij}^{\bar{C}} y_{ijk}$ for each $k \le K$; on the other hand, each inner sum in Eq. (7.26) is equal to zero by Eq. (7.24). This implies $\sum_{(i,j)\in A(\bar{C})} c_{ij}^{\bar{C}} y_{ijk} = 0$ for each $k \le K$. Since C is simple and connected, $\bar{C}$ is a directed circuit. This implies that $c^{\bar{C}} \in \{-1, 0, 1\}$. Now it suffices to replace $-y_{ijk}$ with y_{jik} to obtain

$$\forall k \le K \qquad \sum_{\{i,j\}\in E(C)} y_{ijk} = 0\,,$$

where the edges on C are indexed in such a way as to ensure they appear in order of consecutive adjacency. □

Obviously, if $\mathcal{B}$ has minimum (or just small) cardinality, Eq. (7.24) will be sparsest (or just sparse), which is often a desirable property of linear constraints occurring in MP formulations. Hence we should attempt to find short cycle bases $\mathcal{B}$.

In summary, given a basis $\mathcal{B}$ of the directed cycle space of $\bar{G}$ where c^B is the incidence vector of a cycle $B \in \mathcal{B}$, the following:

$$\left.\begin{array}{rrcl} \min\limits_{s\geq 0, y} & \sum\limits_{\{i,j\}\in E} (s^+_{ij} + s^-_{ij}) & & \\ \forall (i,j) \in A(\bar{G}) & \sum\limits_{k\leq K} y^2_{ijk} - d^2_{ij} & = & s^+_{ij} - s^-_{ij} \\ \forall k \leq K, B \in \mathcal{B} & \sum\limits_{(i,j)\in A(B)} c^B_{ij} y_{ijk} & = & 0 \end{array}\right\} \tag{7.27}$$

is a valid formulation for the EDGP. The solution of Eq. (7.27) yields a feasible vector y^*. As pointed out in Cor. 7.5.14, we must then solve Eq. (7.22) to obtain a realization x^* for G.

7.6.2 How to find directed cycle bases

We require directed cycle bases over $\mathbb{Q}$. By [179, Thm. 2.4], each undirected cycle basis gives rise to a directed cycle basis (so it suffices to find a cycle basis of G and then direct the cycles using the directions in $\bar{G}$). Horton's algorithm [85] and its variants [72, 131] find a minimum cost cycle basis in polynomial time. The most efficient deterministic variant is $O(m^3 n)$ [131], and the most efficient randomized variant has the complexity of matrix multiplication. Existing approximation algorithms have marginally better complexity.

It is not clear, however, that the provably sparsest constraint system will make the DGP actually easier to solve. We therefore consider a much simpler algorithm: starting from a spanning tree, we pick the $m - n + 1$ circuits that each *chord* (i.e., non-tree) edge defines with the rest of the tree. This algorithm [155] yields a *fundamental* cycle basis. Finding the minimum fundamental cycle basis is known to be **NP**-hard [49], but heuristics based on spanning trees prove to be very easy to implement and work reasonably well [49] (optionally, their cost can be improved by an edge-swapping phase [4, 113]).

7.7 The Eulerian cycle relaxation

In this section we construct a relaxation of Eq. (7.27) that decreases the number of constraints in Eq. (7.24), which occurs as the last line in Eq. (7.27), from $|\mathcal{B}|$ to 1.

We let G' be the multigraph obtained from G by adding sufficiently many parallel edges to G, so that the degree of each vertex in G' is even. This can always be done by [56], which implies that G' is Eulerian, i.e. it has a cycle incident with every edge in G' exactly once. We let $\mathscr{E}$ be a Eulerian cycle in G', and let $\bar{\mathscr{E}}$ be either of the two orientations of $\mathscr{E}$ obtained by walking over the cycle. We let $\bar{G}'$ be the digraph induced by the Eulerian circuit $\bar{\mathscr{E}}$. For each $\{i,j\} \in E$ let H_{ij} be the number of parallel edges between i, j in G'.

We note that $\bar{G}'$ might have parallel and antiparallel arcs. Consider the family of arc subset $\mathcal{H}_{ij} = \{(i', j', h) \mid h \leq H_{ij} \wedge \{i', j'\} = \{i, j\}\}$ of $A(\bar{G}')$. We replace each arc $(i', j', h) \in \mathcal{H}_{ij}$ having $h > 1$ by an oriented 2-path $p_{i'j'h} = \{(i', v_{ijh}), (v_{ijh}, j')\}$ involving a new added vertex v_{ijh}. Call $\tilde{G}$ the digraph obtained from $\bar{G}'$ with this replacement. We remark that $\tilde{G}$ is simple (it has no parallel/antiparallel arcs) by construction. Moreover, $\tilde{G}$ is a Eulerian digraph: take the Eulerian circuit $\bar{\mathscr{E}}$ in $\bar{G}'$, and, every time it traverses a parallel/antiparallel arc $(i', j', h) \in \mathcal{H}_{ij}$ with $h > 1$, let it traverse the oriented 2-path replacement $p_{i'j'h}$ instead: this is clearly a Euclidean circuit in $\tilde{G}$, which we call $\mathscr{C}$.

Next we consider the simple graph $\hat{G}$ obtained by replacing each arc in $\tilde{G}$ with an edge. Let $\hat{V} = \{v_{ijh} \mid \{i,j\} \in E \wedge h > 1\}$, and $\hat{E}$ be the subset of edges of $E(\hat{G})$ obtained from losing the orientation of the arcs in the union

$$\bigcup_{\substack{(i',j',h)\in\mathcal{H}_{ij}\\ \{i,j\}\in E\wedge h>1}} p_{i'j'h}$$

of all the arcs from the 2-path replacements. We note that, by construction,

$$\hat{V} = V(\hat{G}) \smallsetminus V \quad \wedge \quad \hat{E} = E(\hat{G}) \smallsetminus E\,. \tag{7.28}$$

Let $c^{\mathscr{C}}_{ij} \in \{1, -1\}$ be the orientation of (i, j) in $\mathscr{C}$ w.r.t. $\hat{G}$; let $\hat{\mathscr{C}}$ be the simple Eulerian cycle in $\hat{G}$ corresponding to $\mathscr{C}$.

We can now prove the main result of this section.

Proposition 7.7.1. *The formulation*

$$\begin{aligned} \min_{s\geq 0, y} \quad & \sum_{\{i,j\}\in E} (s^+_{ij} + s^-_{ij}) \\ \forall (i,j) \in A(\bar{G}) \quad & \sum_{k\leq K} y^2_{ijk} - d^2_{ij} = s^+_{ij} - s^-_{ij} \\ \forall k \leq K \quad & \sum_{(i,j)\in\mathscr{C}} c^{\mathscr{C}}_{ij} y_{ijk} = 0 \quad (\dagger) \end{aligned} \tag{7.29}$$

is a relaxation of Eq. (7.27).

Proof. We form a variant of the cycle formulation Eq. (7.27) applied to $\hat{G}$, where, from the constraints corresponding to Eq. (7.8) (second line of Eq. (7.27)), we omit those indexed by $\hat{E}$. We call this variant ($\star$). We claim that ($\star$) is an exact reformulation of

Eq. (7.27) applied to G. The claim holds because $E(\hat{G}) \setminus \hat{E} = E$ by Eq. (7.28), and because the signs of the y variables are irrelevant in Eq. (7.8) since they are squared. Now, since $\hat{\mathscr{C}}$ is a Eulerian cycle in $\hat{G}$, Eq. (†) is an aggregation of constraints in Eq. (7.24), which occur within the reformulation ($\star$). So Eq. (7.29) is a relaxation of ($\star$). The proposition follows because of the claim. □

Note that Eq. (7.29) provides a solution $\bar{y}$ which may not satisfy Eq. (7.24), which also guarantee feasibility in Eq. (7.10) by Prop. 7.6.1. By Remark 7.5.15, this implies that Cor. 7.5.14 is no longer applicable. In other words, the realization x of G cannot in general be retrieved from $\bar{y}$ using the linear system in Eq. (7.22), since $\bar{y}$ might well make Eq. (7.22) infeasibile. Eq. (7.22), however, can instead be integrated into Eq. (7.29) as additional constraints. This invalidates the decomposition property of Cor. 7.5.14, but allows the relaxation to yield a valid realization.

We therefore define the Eulerian cycle-based relaxation formulation as follows:

$$\begin{aligned} \min_{s \geq 0, x, y} \quad & \sum_{\{i,j\} \in E} (s^+_{ij} + s^-_{ij}) \\ \forall (i,j) \in A(\bar{G}) \quad & \sum_{k \leq K} y^2_{ijk} - d^2_{ij} = s^+_{ij} - s^-_{ij} \\ \forall k \leq K \quad & \sum_{(i,j) \in A(\tilde{\mathscr{E}})} c^{\tilde{\mathscr{E}}}_{ij} y_{ijk} = 0 \\ \forall \{i,j\} \in A(\bar{G}) \quad & x_{ik} - x_{jk} = y_{ijk} \\ \forall k \leq K \quad & \sum_{i \in V} x_{ik} = 0. \end{aligned} \tag{7.30}$$

For a formulation P, we denote by $\mathsf{val}P$ its optimal objective function value. Since Eq. (7.30) has additional constraints w.r.t. Eq. (7.29), we naturally have $\mathsf{val}(7.30) \geq \mathsf{val}(7.29)$. Moreover, for every instance for which a solution $\bar{y}$ of Eq. (7.29) yields an infeasible system Eq. (7.22), by inspection $\bar{y}$ must be infeasible in Eq. (7.30), which implies that there are cases where Eq. (7.30) is a strictly tighter relaxation than Eq. (7.29). The very last constraint in Eq. (7.30) fixes the centroid of the points at the origin, as in Eq. (7.4).

7.8 Computational experiments

The aim of this section is to compare the computational performance of the following EDGP formulations:

(i) the cycle-based formulation in Eq. (7.27), where the realization is retrieved as a post-processing stage using (7.22) according to Cor. 7.5.14;

(ii) the Eulerian cycle-based relaxation in Eq. (7.30);

(iii) the classic edge-based formulation in Eq. (7.4).

All of these formulations are nonconvex NLP, which are generally NP-hard to solve. Specifically, formulations (i) and (iii) are as hard to solve as the EDGP, which is NP-hard. No specific NP-hardness proof exists for formulation (ii) yet.

As a solution algorithm, we used a very simple MS heuristic based on calling a local NLP solver from a random initial starting point at each iteration, and updating the best solution found so far as needed: although there are better heuristics around [126, 37, 140], MS is the best trade-off between implementation simplicity and efficiency. Moreover, more efficient heuristic often change the formulation during their execution, which may hinder the meaning of this computational comparison between formulations.

We evaluate the quality of a realization x of a graph G according to mean (MDE) and largest distance error (LDE), defined this way:

$$\begin{aligned} \mathsf{mde}(x, G) &= \frac{1}{|E|} \sum_{\{i,j\} \in E} \big| \|x_i - x_j\|_2 - d_{ij} \big| \\ \mathsf{lde}(x, G) &= \max_{\{i,j\} \in E} \big| \|x_i - x_j\|_2 - d_{ij} \big| \,. \end{aligned}$$

The CPU time taken to find the solution may also be important, depending on the application. In the control of underwater vehicles [12], for example, DGP instances might need to be solved in real time. In other applications, such as finding protein structure from distance data [1, 139] (our application of choice), the CPU time is not so important.

Our tests were carried out on a single CPU of a 2.1GHz 4-CPU 8-core-per-CPU machine with 64GB RAM running Linux. The local NLP solver used within the MS heuristic was the IPOpt solver [39]. We remarked in some preliminary tests that IPOpt was considerably slowed down by variants of Eq. (7.3) such as Eq. (7.5), which essentially move a nonconvexity on the objective to one in the constraints. The same holds for the cycle-based formulation in Eq. (7.27). We therefore reformulated Eq. (7.27) as follows:

$$\begin{aligned} \min_{y} \quad & \sum_{\{i,j\} \in A(\bar{G})} (\sum_{k \leq K} y_{ijk}^2 - d_{ij}^2)^2 \\ \forall k \leq K, B \in \mathcal{B} \quad & \sum_{(i,j) \in A(B)} c_{ij}^B y_{ijk} = 0 \,, \end{aligned} \tag{7.31}$$

and Eq. (7.30) similarly.

Our implementation consists of a mixture of Python 3 [163] and AMPL [66] interfaced through `amplpy`. Cycle bases and Eulerian cycles are found using `networkX` [80]. Solutions to the feasible but possibly overdetermined linear systems in Eq. (7.22) are obtained using an ℓ_1 error minimization approach reformulated as a LP problem solved with CPLEX [91].

7.8.1 Results

A benchmark on a diverse collection of randomly generated weighted graphs of small size and many different types, with a very similar set-up to the one discussed here, is

presented in [120]. It was found that the cycle formulation finds better MDE values, while the edge formulation generally finds better LDE values and is faster. Some results on proteins, obtained with only 3 MS iterations, were also presented in [120].

The benchmark we consider here contains medium to large scale protein graph instances realized in $\mathbb{R}^3$. With regard to the protein results presented in [120], we integrated one more instance, `1tii`, which, at 69800 edges and 5684 vertices, is considerably larger than all the others.

			MDE			LDE			CPU		
Instance	m	n	cycle	Eul	edge	cycle	Eul	edge	cycle	Eul	edge
`1guu`	955	150	0.086	0.069	**0.053**	1.234	1.068	**1.037**	**7.90**	553.76	290.21
`1guu-1`	959	150	0.080	0.082	**0.059**	1.013	1.069	**0.980**	9.67	23.03	**1.72**
`1guu-4000`	968	150	0.112	0.106	**0.092**	1.073	1.431	**0.936**	8.68	10.77	**1.56**
`pept`	999	107	**0.144**	0.239	0.179	2.862	**1.847**	1.943	5.52	4.72	**1.4**
`2kxa`	2711	177	**0.051**	0.119	0.172	3.705	**2.826**	3.813	21.53	25.54	**7.35**
`res_2kxa`	2627	177	**0.055**	0.237	0.156	**2.949**	3.570	3.054	20.84	21.20	**12.44**
`C0030pkl`	3247	198	**0.000**	0.145	0.211	**0.000**	3.537	3.829	29.50	26.69	**7.36**
`cassioli`	4871	281	0.146	0.113	**0.057**	3.914	3.616	**3.185**	47.23	48.44	**14.51**
`100d`	5741	488	**0.201**	-	0.251	**3.038**	-	3.987	387.32	-	**29.42**
`hlx_amb`	6265	392	**0.105**	0.214	0.119	3.836	3.888	**3.485**	120.25	80.27	**20.54**
`water`	11939	648	**0.146**	0.490	0.243	**3.579**	4.196	4.281	1346.69	399.42	**224.66**
`3al1`	17417	678	**0.062**	0.126	0.216	3.451	**3.175**	4.059	835.10	433.69	**123.45**
`1hpv`	18512	1629	**0.385**	0.402	0.416	3.847	**3.831**	4.015	10138.00	2387.29	**442.70**
`il2`	45251	2084	0.385	**0.049**	0.107	4.422	**4.204**	4.583	18141.22	9904.81	**5255.76**
`1tii`	69800	5684	0.620	0.436	**0.434**	6.755	4.492	**3.854**	18846.37	38230.21	**9039.28**
avg			**0.172**	0.202	0.184	**3.045**	3.054	3.136	3331.05	3724.99	**1031.49**
stdev			0.167	0.144	**0.118**	1.673	**1.204**	1.272	6672.49	10272.3	**2587.33**
$\lvert best \rvert$			**9**	1	5	4	5	**6**	1	0	**14**

Table 7.1: Cycle formulation vs. edge formulation performance on protein graphs (realizations in $K = 3$ dimensions).

The results are given in Table 7.1. We report instance name, instance sizes m and n, then performance measures MDE, LDE and CPU for cycle, Eulerian and edge-based formulations. In the last three lines we report average, standard deviation, and number of instances where the formulation performed best, for all performance measures. In all tested cases, finding the cycle basis, the Eulerian cycles, and solving Eq. (7.22) took a small fraction of the total solution time. The missing result for instance `100d` on the Eulerian cycle reformulation is due to a failure occurred in the `networkX` module because the graph of `100d` is not connected.

It appears that, on average, there is relatively little difference between the quality performances of these three DGP formulations on protein graphs of medium and large sizes. CPU-time wise, of course, the edge formulation is best. Cycle formulations, taken together, are definitely better than the edge formulation on quality measures. The cycle-based formulation Eq. (7.27) is slightly better than the other formulations for both MDE

and LDE. The number of instances on which Eq. (7.27) is best on quality measures is 13, against 11 for the edge-based formulation. Eq. (7.27) was the only formulation by which a global optimum was found (that of `C0030pkl`). All in all, we believe that our results show that cycle formulations are credible competitors w.r.t. the well established edge-based formulations, especially when the CPU time is not an important performance measure (which is generally the case in the protein conformation application).

7.9 Conclusions

In this chapter, we discussed the DGP, namely, the problem of finding a realization of a graph in a Euclidean space of given dimension. Our contribution is a new modelling technique for this problem, based on MP: in particular, we propose a novel MP formulation of the DGP, and a corresponding relaxation, and we perform computational experiments to test our formulations.

Our interest in DGP lies mainly in the fact that a large number of practical problems, whereby one knows the distances between entities in a set and wants to find their coordinates, can be attributed to it. Furthermore, the DGP is relevant to this thesis because of its potential use as a methodology for processing ML training data, in view of fitting paradigms that natively work with vectorial inputs to graphs.

Chapter 8

Future developments

In this chapter, we briefly discuss some research ideas that, due to lack of time, could not be implemented and/or tested properly during my Ph. D.

8.1 The PaO approach with Neural Networks

In Ch. 5, we presented the PaO approach to the ACP (Sec. 5.3) and several implementations, corresponding to different ML paradigms (Sec. 5.4).

Here, we show the CSSP formulation corresponding to the use of an NN in the K-EP, to construct an approximation of the performance $p_{\mathcal{A}}$. We refer to Sec. 3.6.3 for a presentation of NNs.

We want to solve the ACP for an instance $\bar{\pi}$ of an optimization problem Π. We consider an NN with a single vertex o in the output layer O. Moreover, since the inputs of our problem are vectors (π, c), the vertices in the NN input layer are partitioned as $I = I_\pi \cup I_{\mathcal{C}_\mathcal{A}}$. The vertices in I_π encode $\bar{\pi}$, while those in $I_{\mathcal{C}_\mathcal{A}}$ represent the configuration variables. The CSSP is as follows:

$$\mathsf{CSSP}(\bar{\pi}) \equiv \begin{cases} \arg\min\limits_{c \in \mathcal{C}_\mathcal{A}} & \sigma_o(b^*_o + \sum\limits_{(v,o)\in A} w^*_{vo} z_v) \\ \forall u \in I_\pi & z_u = \bar{\pi}[u] \\ \forall u \in I_{\mathcal{C}_\mathcal{A}} & z_u = c[u] \\ u \in H & z_u = \sigma_u(b^*_u + \sum\limits_{(v,u)\in A} w^*_{vu} z_v)\,. \end{cases} \tag{8.1}$$

The shorthand $\bar{\pi}[u],\ c[u]$ has the same meaning as in Sec. 3.6.2, i.e., for instance, $\bar{\pi}[u]$ is the u-th components of vector π. In general, Eq. (8.1) is a nonconvex MINLP, owing to the structure of the activation functions in the network vertices. However, when specific activation functions are employed (say, ReLU), the formulation can be linearized, which yields an MILP formulation of the CSSP (see, for instance, [62, 172, 159, 185]).

We are currently testing several implementations of the CSSP in Eq. (8.1), for different NN architectures and activation functions.

8.2 The multi-phase PaO approach

In this section, we consider an extension of the PaO approach. Its purpose is to solve the ACP for a B&B-based optimization solver (see Sec. 2.6.1.1), while it is running to solve a specific instance. In particular, we aim to find the best configuration of parameters at each of the different phases of the solution process. We are motivated by previous work in ML-based configuration and enhancement of MP solvers, which employs information about the execution of the solver — rather relying solely on the characteristics of the instances to be solved — to improve algorithmic performance, on-the-run (see, for instance, [200, 54, 183]).

In general, one can identify $1, 2, \dots T$ phases of the solution process; each phase is governed by specific algorithmic parameters, and we want to decide the optimal values of those parameters before that phase begins, to execute it as efficiently as possible. At the end of each phase $t < T$, executed with the recommended configuration, we encode the solver behaviour into an array of features. This framework requires that we provide an estimate of the solver performance on the remaining phases (i.e., for $t' \geq t$), given that, in the previous phases $1, \dots, t$, we have already recommended solver configurations. Moreover, this setting requires that each phase is addressed by a specific K-EP (to learn the algorithmic performance for that phase) and a specific CSSP (to recommend the optimal configuration for executing that phase). Thus we refer to this variant as "multi-phase" PaO approach.

8.2.1 Implementation

Since, in the multi-phase variant of the PaO approach, we want to execute K-EP and CSSP for each phase, the training set will be different than the one defined in Sec. 5.3.1.

Formally, we let π be an instance to solve. Furthermore, being T the number of phases, the purpose of the CSSPs is to decides the configuration vector

$$c^* = [c^1, c^2, \dots, c^T] \in [0,1]^{q_1} \times \cdot \times [0,1]^{q_T}$$

where, for $1 \leq t \leq T$, c^t, is the configuration vector used to run the solver in phase t, and q_t is its size. It is possible for the same algorithmic parameters to appear in configuration vectors of different phases.

Further, we consider feature vectors

$$\iota = [\iota^1, \iota^2, \dots, \iota^T],$$

where $\iota^1 = \pi$ are the instance features and, for $1 < t \leq T$, ι^t encodes the algorithmic behaviour of the solver, configured by c^{t-1}, at phase $t-1$.

Thus, at each phase t, the CSSP optimizes an approximation of the performance function at phase t, given the features ι^t. Its solution, that we call $\boldsymbol{c^t}$, is used to configure the solver and execute it, which also allows us to compute ι^{t+1}.

To perform the K-EP, we first sample a training set

$$\mathcal{S} = \{(\iota^1, \iota_i^2, \ldots, \iota_i^T,\ c_i^1, c_i^2, \ldots, c_i^T,\ p_{\mathcal{A}}(\iota_i, c_i)) \mid i = 1 \ldots s\}\,, \tag{8.2}$$

which is usually larger than the basic PaO training set, as there are potentially many more possible configurations to consider.

To define the performance function $p_{\mathcal{A}}$, we let

$$\begin{cases} \iota_i^{<t} = [\iota_i^1, \ldots, \iota_i^{t-1}], & \iota_i^{\leq t} = [\iota_i^1, \ldots, \iota_i^t], & \iota_i^{>t} = [\iota_i^{t+1}, \ldots, \iota_i^T] \\ c_i^{<t} = [c_i^1, \ldots, c_i^{t-1}], & c_i^{\leq t} = [c_i^1, \ldots, c_i^t], & c_i^{>t} = [c_i^{t+1}, \ldots, c_i^T]\,. \end{cases}$$

We would like to define the performance function as

$$\underline{p}_{\mathcal{A}}^t(\iota, c^{\leq t}) = \min_{i \in S}\{p_{\mathcal{A}}(\iota_i, c_i) \mid c_i^{\leq t} = \boldsymbol{c^{\leq t}}\}\,, \tag{8.3}$$

where $\boldsymbol{c^{\leq t}}$ is the configuration recommended by our methodology. However, Eq. (8.3) cannot be used because, at t, $\iota^{>t}$ is unknown. Therefore, we define the performance function as:

$$p_{\mathcal{A}}^t(\iota^{\leq t}, c^{\leq t}) = \operatorname*{\mathsf{avg}}_{i \in S}\left\{\underline{p}_{\mathcal{A}}^t(\iota_i, c^{\leq t}) \mid \iota_i^{\leq t} = \iota^{\leq t}\right\}, \tag{8.4}$$

where: if $t = 1$, $\iota^{\leq t}$ is π; otherwise, $\iota^{\leq t}$ is computed by executing the solver with the configuration $\boldsymbol{c^{<t}}$ recommended by our methodology. In Eq. (8.4), the purpose of avg is to provide an estimate of the potential algorithmic performance at subsequent phases, subject to the uncertainty about the actual realization of $\iota^{>t}$. This strategy is related to robust/uncertain optimization. Other estimation strategies could be used: for instance, if one wanted to hedge against the risk of choosing configurations that can be "very bad" in some cases, Value-at-Risk or Conditional-Value-at-Risk may be employed instead of avg.

With the performance function defined in Eq. (8.4), the K-EP for phase t relies on the training set

$$\mathcal{S}^t = \left\{\left(\iota_i^{\leq t}, c_i^{\leq t}, p_{\mathcal{A}}^t(\iota_i^{\leq t}, c_i^{\leq t})\right) \mid i \in S^j\right\},$$

to learn an ML approximation

$$\bar{p}_{\mathcal{A}}^t(\iota^{\leq t}, c^{\leq t})$$

of $p_{\mathcal{A}}^t(\iota^{\leq t}, c^{\leq t})$. The corresponding CSSP_t, solved to find $\boldsymbol{c^{\leq t}}$, is:

$$\mathsf{CSSP}_t(\bar{\pi}) \equiv \arg\min_{c \in \mathcal{C}_{\mathcal{A}}}\{\bar{p}_{\mathcal{A}}^t(\bar{\iota}^{\leq t}, [\boldsymbol{c^{<t}}, c^t])\}\,, \tag{8.5}$$

where $\bar{\iota}^{<t}$ and $\boldsymbol{c^{<t}}$ are fixed. In Eq. (8.5), one is optimizing the average estimated performance $\bar{p}_{\mathcal{A}}$, by selecting an algorithmic configuration c^t for executing the t-th phase, such that the algorithm has already been configured by $\boldsymbol{c^{<t}}$ in the previous phases, and the features $\bar{\iota}^{\leq t}$, corresponding to these choices, have already been computed.

Chapter 9

Conclusions

The contributions of this Ph. D. thesis fall in the domains of algorithm configuration and distance geometry.

The research topics covered here lie at the intersection of MP — a formal language for describing optimization problems and a framework for solving them — and ML — a set of techniques for designing and implementing algorithms to construct hypotheses from data. The text opens with an introduction to these two fields (Ch. 2 and 3), intended to make this thesis as self-contained as possible.

In the central part of the thesis, we deal with the ACP, i.e., given a target algorithm, the problem of finding the parameter configuration yielding the best algorithmic performance for solving a specific instance.

First, we review the relevant state of the art. In Ch. 4, we first present an algorithmic schema, whose purpose is the description and classification of any approach to the ACP. Then, we survey the main ACP methodologies in the literature, and fit them into our schema. Part of the chapter has been accepted for publication in a journal as, to the extent of our knowledge, the proposed schema is original.

In Ch. 5 and 6, we propose two novel MP-driven approaches to the ACP, that optimize over trained ML predictors to find good algorithmic configurations for specific instances; we also study the application of the ACP to solution algorithms for MP (i.e., MP solvers).

To deal with the ACP, we use ML predictors as elements appearing in an MP formulation. We first construct an ML approximation of the performance of the target algorithm $\mathcal{A}$, deployed to solve instances of a given optimization problem Π. Then, we translate the mathematical properties underlying the learned approximation into MP terms; finally, we embed the resulting equations into an MP formulation, approximating the ACP for a specific instance. We refer to this MP as "CSSP". The objective of the CSSP optimizes the ML-derived approximation of the algorithmic performance;

its constraints determine the set of admissible configurations (and, potentially, other conditions on the learned approximation). When a instance arrives, we optimize the CSSP to solve the ACP for that instance. One of the advantages of this framework is that it allows great freedom in choosing the most appropriate ML paradigm for Π and $\mathcal{A}$. Another quality of our approach is that it can exploit the structure of the CSSP to solve it to a global optimum. All the other ACP methodologies, prominent in the literature, treat the ACP as a black-box problem, so they solve it by local (meta)heuristics, and can only manage to find configurations that are good for a set of instances with similar characteristics. Moreover, black-box methodologies often do not scale well large sets of feasible configurations, or to situations where the ACP-optimal configuration depends on the instance at hand. In these cases, MP solution algorithms should be more efficient, since they use a problem structure.

In Ch. 5 and 6, we investigate different implementation choices for the learning phase; furthermore, we discuss the nontrivial trade-offs arising from the fact that these choices affect not only the accuracy of the learned predictor, but also the cost of solving the corresponding CSSP. The ideas and results presented in these two chapters resulted in two publications, [93] and [94].

In the last part of the thesis, we discuss the DGP.

The DGP asks to find a realization of a given graph in a Euclidean space of given dimension. In particular, it considers simple, undirected graphs, whereby the vertex positions are unknown and the edges are labeled by a weight/length function. Given an integer K, its goal is to reconstruct the position of the graph vertices in a Euclidean space of dimension K, from the known edge weights, while drawing the edges as straight segments of length equal to the edge weights.

A customary approach to the DGP is to solve an MP formulation to determine the position of the vertices. We propose an alternative MP formulation where, instead, we consider the cycles of the graph: we determine the length of the segments modelling the edges in each cycle, so that, for any two vertices incident on an edge, their distance is as close as possible to the corresponding edge weight. Beyond the novelty of the formulation that we propose, our research is also motivated by the fact that the DGP can be instrumental in applying vector-based ML paradigms to graphs. A shortened version of Ch. 7 was published in [120], and an extended version has been submitted for publication to a journal.

Bibliography

[1] A. Cassioli *et al.* An algorithm to enumerate all possible protein conformations verifying a set of distance constraints. *BMC Bioinformatics*, 16:23–38, 2015.

[2] A. Choromanska *et al.* The loss surface of multilayer networks. *Journal of Machine Learning Research*, 38:192–204, 2015.

[3] B. Adenso-Díaz and M. Laguna. Fine-tuning of algorithms using Fractional Experimental Design and Local Search. *Operations Research*, 54(1):99–114, 2006.

[4] E. Amaldi, L. Liberti, F. Maffioli, and N. Maculan. Edge-swapping algorithms for the minimum fundamental cycle basis problem. *Mathematical Methods of Operations Research*, 69:205–223, 2009.

[5] B. Amos and Z. Kolter. OptNet: Differentiable optimization as a layer in Neural Networks. In D. Precup and Y. W. Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70, pages 136–145. PMLR, 2017.

[6] E. D. Anderson, J. Gondzio, C. Mészáros, and X. Xu. *Implementation of Interior-Point Methods for Large Scale Linear Programs*, pages 189–252. Springer US, Boston, MA, 1996.

[7] C. Ansótegui, M. Sellmann, and K. Tierney. A gender-based genetic algorithm for the automatic configuration of algorithms. In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming*, pages 142–157, Berlin, Heidelberg, 2009. Springer-Verlag.

[8] C. G. Atkeson, A. W. Moore, and S. Schaal. Locally Weighted Learning. *Artificial Intelligence Review*, 11:11—-73, 1997.

[9] C. Audet, D. Kien, and D. Orban. Algorithmic parameter optimization of the DFO method with the OPAL framework. *Software Automatic Tuning: From Concepts to State-of-the-Art Results*, pages 255–274, 2010.

[10] C. Audet and D. Orban. Finding optimal algorithmic parameters using Derivative-Free Optimization. *SIAM Journal on Optimization*, 17(3):642–664, 2006.

[11] D. Bacciu, F. Errica, A. Micheli, and M. Podda. A gentle introduction to deep learning for graphs. *Neural Networks*, 129:62, 2020.

[12] A. Bahr, J. Leonard, and M. Fallon. Cooperative localization for autonomous underwater vehicles. *International Journal of Robotics Research*, 28(6):714–728, 2009.

[13] M. F. Balcan. Georgia Tech, 8803: Machine Learning Theory, 2011. Lecture notes of the course.

[14] R. Battiti and M. Brunato. Reactive Search: machine learning for memory-based heuristics. Technical report, University of Trento, 2005.

[15] M. S. Bazaraa, H. D. Sherali, and C. M. Shetty. *Nonlinear Programming: Theory and Algorithms*. Wiley Publishing, 3rd edition, 2013.

[16] N. Beeker, S. Gaubert, C. Glusa, and L. Liberti. Is the Distance Geometry Problem in NP? In A. Mucherino, C. Lavor, L. Liberti, and N. Maculan, editors, *Distance Geometry: Theory, Methods, and Applications*, pages 85–94. Springer, 2013.

[17] N. Belkhir, J. Dréo, P. Savéant, and M. Schoenauer. Per instance algorithm configuration of CMA-ES with limited budget. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 681–688, New York, NY, USA, 2017. ACM.

[18] A. Berg and T. Jordán. Algorithms for graph rigidity and scene analysis. In G. Di Battista and U. Zwick, editors, *Algorithms: Proceedings of the European Symposium on Algorithms*, volume 2832, pages 78–89, Berlin, 2003. Springer.

[19] T. Berthold and G. Hendel. Learning to scale mixed-integer programs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 5, pages 3661–3668. AAAI Press, 2021.

[20] D. P. Bertsekas. *Nonlinear Programming*. Athena Scientific, 1999.

[21] E. Bingham and H. Mannila. Random projection in dimensionality reduction: Applications to image and text data. In *Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 245––250, New York, NY, USA, 2001. Association for Computing Machinery.

[22] C. M. Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer-Verlag, Berlin, Heidelberg, 2006.

[23] M. G. Vilas Boas, H. G. Santos, R. de S. O. Martins, and L. H. C. Merschmann. Data mining approach for feature based parameter tuning for mixed-integer programming solvers. In P. Koumoutsakos *et al.*, editor, *Proceedings of the International Conference on Computational Science*, volume 108, pages 715–724. Elsevier, 2017.

[24] M. G. Vilas Boas, H. Gambini Santos, L. H. de Campos Merschmann, and G. Vanden Berghe. Optimal decision trees for the algorithm selection problem: Integer programming based approaches. *International Transactions in Operational Research*, 28, 2019.

[25] P. Bonami, A. Lodi, and G. G. Zarpellon. Learning a classification of mixed-integer quadratic programming problems. In W. J. van Hoeve, editor, *Proceedings of the International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, volume 10848 of *Lecture Notes in Control and Information Sciences*, pages 595–604. Springer, Cham, 2018.

[26] A. Borghetti, C. D'Ambrosio, A. Lodi, and S. Martello. An MILP approach for short-term hydro scheduling and unit commitment with head-dependent reservoir. *IEEE Transactions on Power Systems*, 23(3):1115–1124, 2008.

[27] L. Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[28] O. Bousquet, S. Boucheron, and G. Lugosi. Introduction to statistical learning theory. In O. Bousquet, U. von Luxburg, and G. Rätsch, editors, *Advanced Lectures on Machine Learning*, volume 3176 of *Lecture Notes in Computer Science*, pages 169–207. Springer, 2003.

[29] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, New York, NY, USA, 2004.

[30] E. Bradley. *The jackknife, the bootstrap and other resampling plans*. SIAM, 1982.

[31] L. Breiman, J. H. Friedman R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth and Brooks, Monterey, CA, 1984.

[32] M. Brendel and M. Schoenauer. Instance-based parameter tuning for Evolutionary AI Planning. In *Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation*, pages 591–598. ACM, 2011.

[33] S. Bubeck. Convex Optimization: Algorithms and complexity. *Found. Trends Mach. Learn.*, 8(3—4):231—357, 2015.

[34] M. Bukatin, R. Kopperman, S. Matthews, and H. Pajoohesh. Partial metric spaces. *American Mathematical Monthly*, 116(8):708–718, 2009.

[35] C. J. C. Burges and D. J. Crisp. Uniqueness of the SVM solution. In *Proceedings of the 12th International Conference on Neural Information Processing Systems*, pages 223—-229, Cambridge, MA, USA, 1999. MIT Press.

[36] C. Ansótegui *et al.* Model-based genetic algorithms for algorithm configuration. In *Proceedings of the 24th International Conference on Artificial Intelligence*, pages 733—739. AAAI Press, 2015.

[37] C. D'Ambrosio *et al.* New error measures and methods for realizing protein graphs from distance data. *Discrete and Computational Geometry*, 57(2):371–418, 2017.

[38] G. C. Cawley and N. L. C. Talbot. On over-fitting in model selection and subsequent selection bias in performance evaluation. *Journal of Machine Learning Reserach*, 11:2079–2107, 2010.

[39] COIN-OR. *Introduction to IPOPT: a tutorial for downloading, installing, and using IPOPT*, 2006.

[40] A. Collins, L. Tierney, and J. Beel. Per-instance algorithm selection for recommender systems via instance clustering. *CoRR*, abs/2012.15151, 2020.

[41] R. Connelly. Generic global rigidity. *Discrete Computational Geometry*, 33:549–563, 2005.

[42] D. Cox. The regression analysis of binary sequences. *Journal of the Royal Statistical Society*, 20(2):215–242, 1958.

[43] D. R. Cox and N. Reid. *The theory of the Design of Experiments.* CRC Press, 2000.

[44] N. Cristianini and J. Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods.* Cambridge University Press, 2000.

[45] L. Pérez Cáceres and T. Stützle. Exploring variable neighborhood search for automatic algorithm configuration. *Electronic Notes in Discrete Mathematics*, 58:167–174, 2017.

[46] C. D'Ambrosio and L. Liberti. Distance geometry in linearizable norms. In F. Nielsen and F. Barbaresco, editors, *Proceedings of the 3rd Conference on*

Geometric Science of Information, volume 10589, pages 830–838, Berlin, 2017. Springer.

[47] G. B. Danzig and P. Wolfe. Decomposition principle for linear programs. *Oper. Res.*, 8(1):101–111, February 1960.

[48] E. de Klerk and D. V. Pasechnik. Approximation of the stability number of a graph via copositive programming. *SIAM Journal on Optimization*, 12(4):875—-892, 2002.

[49] N. Deo, G. M. Prabhu, and M. S. Krishnamoorthy. Algorithms for generating fundamental cycles in a graph. *ACM Transactions on Mathematical Software*, 8(1):26–42, 1982.

[50] G. Dias and L. Liberti. Diagonally dominant programming in distance geometry. In R. Cerulli, S. Fujishige, and R. Mahjoub, editors, *Proceedings of the International Symposium in Combinatorial Optimization*, volume 9849 of *LNCS*, pages 225–236, New York, 2016. Springer.

[51] Y. Ding, N. Krislock, J. Qian, and H. Wolkowicz. Sensor network localization, Euclidean distance matrix completions, and graph realization. *Optimization and Engineering*, 11:45–66, 2010.

[52] M. A. Duran and I. E. Grossmann. *An outer-approximation algorithm for a class of mixed-integer nonlinear programs*, pages 307–339. Number 36. Springer, 1986.

[53] C. D'Ambrosio and A. Lodi. Mixed integer nonlinear programming tools: an updated practical overview. *Annals of Operations Research*, 204(1):301–320, 2013.

[54] E. B. Khalil *et al.* Learning to branch in Mixed Integer Programming. In *Proceedings of the 13th AAAI Conference on Artificial Intelligence*, pages 724–731. AAAI Press, 2016.

[55] E. Demirović *et al.* An investigation into prediction + optimisation for the knapsack problem. In *Proceedings of the International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 241–257. Springer Cham., 2019.

[56] J. Edmonds and E. Johnson. Matching, Euler tours, and the Chinese postman. *Mathematical Programming*, 5:88–124, 1973.

[57] K. Eggensperger, M. Lindauer, and F. Hutter. Pitfalls and best practices in algorithm configuration. *Journal of Artificial Intelligence Research*, 64(1):861––893, 2019.

[58] F. Chollet *et al.* Keras. `https://keras.io`, 2015.

[59] F. Fioretto *et al.* A Lagrangian dual framework for Deep Neural Networks with constraints optimization, 2020.

[60] 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*, volume 34, pages 1504–1511, 2019.

[61] M. Fischetti, F. Glover, and A. Lodi. The Feasibility Pump. *Mathematical Programming*, 104:91–104, 09 2005.

[62] M. Fischetti and J. Jo. Deep neural networks and mixed integer linear optimization. *Constraints*, 23:14, 2018.

[63] M. Fischetti and A. Lodi. Local branching. *Mathematical Programming*, 98:23–47, 09 2003.

[64] M. Fischetti and A. Lodi. *Heuristics in Mixed integer programming.* John Wiley and sons, Inc., 2010.

[65] C. A. Floudas and P. M. Pardalos, editors. *History of optimization*, pages 1538–1542. Springer US, Boston, MA, 2009.

[66] R. Fourer and D. Gay. *The AMPL Book.* Duxbury Press, Pacific Grove, 2002.

[67] G. Bigi *et al. Appunti di Ricerca Operativa.* Servizio editoriale universitario di Pisa, Pisa, 2018.

[68] G. E. Hinton *et al.* Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580, 2012.

[69] G. Gamrath *et al.* The SCIP Optimization Suite 7.0. ZIB-Report 20-10, Zuse Institute Berlin, 2020.

[70] M. Gasse, D. Chételat, N. Ferroni, L. Charlin, and A. Lodi. Exact combinatorial optimization with graph convolutional neural networks. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, Red Hook, NY, USA, 2019. Curran Associates Inc.

[71] M. Gendreau and J. Y. Potvin. *Handbook of Metaheuristics.* Springer Publishing Company, Incorporated, 2nd edition, 2010.

[72] A. Golynski and J. D. Horton. A polynomial time algorithm to find the minimum cycle basis of a regular matroid. In *Proceedings of the 8th Scandinavian Workshop on Algorithm Theory*, 2002.

[73] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

[74] D. Grimes, G. Ifrim, B. O'Sullivan, and H. Simonis. Analyzing the impact of electricity price forecasting on energy cost-aware scheduling. *Sustainable Computing: Informatics and Systems*, 4:276–291, 2014.

[75] L. Grippo and M. Sciandrone. Metodi di ottimizzazione per le reti neurali, 2003.

[76] I. E. Grossmann and Z. Kravanja. *Mixed-Integer Nonlinear Programming: A Survey of Algorithms and Applications*, pages 73–100. Springer New York, New York, NY, 1997.

[77] R. Gupta and T. Roughgarden. A PAC approach to application-specific algorithm selection. In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science*, pages 123—-134, New York, NY, USA, 2016. Association for Computing Machinery.

[78] Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual*, 2021.

[79] I. Guyon. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182, 2003.

[80] A. Hagberg, D. Schult, and P. Swart. Exploring network structure, dynamics, and function using `NetworkX`. In G. Varoquaux, T. Vaught, and J. Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11–15, Pasadena, CA, 2008.

[81] T. Hastie, R. Tibshirani, and J. H. Friedman. *The elements of statistical learning: data mining, inference, and prediction, 2nd Edition*. Springer series in statistics. Springer, 2009.

[82] J. Hiriart-Urruty and C. Lemarechal. *Convex Analysis and Minimization Algorithms I*, volume 305. Springer-Verlag, Berlin Heidelberg, 1993.

[83] P. Hoffman and B. Richter. Embedding graphs in surfaces. *Journal of Combinatorial Theory B*, 36:65–84, 1984.

[84] T. Hofmann, B. Schölkopf, and A. J. Smola. Kernel methods in machine learning. *Annals of Statistics*, 36(3):1171–1220, 2008.

[85] J. D. Horton. A polynomial-time algorithm to find the shortest cycle basis of a graph. *SIAM Journal of Computing*, 16(2):358–366, 1987.

[86] F. Hutter and Y. Hamadi. Parameter adjustment based on performance prediction: Towards an instance-aware problem solver. Technical Report

MSR-TR-2005125, Microsoft Research, 2005. https://www.microsoft.com/en-us/research/publication/parameter-adjustment-based-on-performance-prediction-towards-an-instance-aware-problem-solver/.

[87] F. Hutter, H. Hoos, and K. Leyton-Brown. Automated configuration of Mixed Integer Programming solvers. In A. Lodi, M. Milano, and P. Toth, editors, *Proceedings of the 7th International Conference on the Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 6140 of *LNCS*, pages 186–202, Heidelberg, 2010. Springer.

[88] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential Model-based Optimization for general algorithm configuration. In *Proceedings of the 5th International Conference on Learning and Intelligent Optimization*, pages 507–523. Springer-Verlag, 2011.

[89] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle. ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36(1):267–306, 2009.

[90] I. Guyon *et al.* Capacity control in linear classifiers for pattern recognition. In *Proceedings of the 11th International Conference on Pattern Recognition*, pages 385–388, 1992.

[91] IBM. *IBM ILOG CPLEX Optimization Studio, CPLEX 12.7 User's Manual*. IBM, 2016.

[92] IBM. *IBM ILOG CPLEX Optimization Studio CPLEX Parameters Reference*, 2016. https://www.ibm.com/docs/en/SSSA5P_12.8.0/ilog.odms.studio.help/pdf/paramcplex.pdf.

[93] G. Iommazzo, C. D'Ambrosio, A. Frangioni, and L. Liberti. A learning-based mathematical programming formulation for the automatic configuration of optimization solvers. In G. Nicosia *et al.*, editor, *Proceedings of the 6th International Conference on Machine Learning, Optimization, and Data Science*, volume 12565 of *Information Systems and Applications, incl. Internet/Web, and HCI*, pages 700–712. Springer, 2020.

[94] G. Iommazzo, C. D'Ambrosio, A. Frangioni, and L. Liberti. Learning to configure mathematical programming solvers by mathematical programming. In I. S. Kotsireas and P. M. Pardalos, editors, *Proceeedings of the 14th International Conference on Learning and Intelligent Optimization - 14th International Conference*, volume 12096 of *Lecture Notes in Computer Science*, pages 377–389. Springer, 2020.

[95] J. Aspnes *et al.* A theory of network localization. *IEEE Transactions on Mobile Computing*, 5(12):1663–1678, 2006.

[96] J. Pérez *et al.* A statistical approach for algorithm selection. In C. C. Ribeiro and S. L. Martins, editors, *Proceedings of the International Workshop on Experimental and Efficient Algorithms*, pages 417–431, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[97] R. C. Jeroslow. There cannot be any algorithm for integer programming with quadratic constraints. *Operations Research*, 21(1):221—224, February 1973.

[98] R. C. Jeroslow. There cannot be any algorithm for integer programming with quadratic constraints. *Operations Research*, 21(1):221—-224, 1973.

[99] D. S. Johnson. *A Catalog of Complexity Classes*, pages 67—161. MIT Press, Cambridge, MA, USA, 1991.

[100] M. Jordan. Why the logistic function? A tutorial discussion on probabilities and neural networks. Technical Report Computational Cognitive Science TR 9503, MIT, 1995.

[101] S. Kadioglu, Y. Malitsky, M. Sellmann, and K. Tierney. ISAC: Instance Specific Algorithm Configuration. In *Proceedings of the 19th European Conference on Artificial Intelligence*, pages 751–756, Amsterdam, The Netherlands, 2010. IOS Press.

[102] R. Kannan and C. L. Monma. On the Computational Complexity of Integer Programming Problems. In R. Henn, B. Korte, and W. Oettli, editors, *Optimization and Operations Research*, pages 161–172. Springer Berlin Heidelberg, Berlin, Heidelberg, 1978.

[103] K. G. Katta and S. N. Kabadi. Some NP-complete problems in quadratic and nonlinear programming. *Mathematical Programming*, 39(2):117–129, 1987.

[104] P. Kerschke, H. H. Hoos, F. Neumann, and H. Trautmann. Automated algorithm selection: Survey and perspectives. *Evolutionary Computation*, 27:3–45, 2018.

[105] P. Kerschke, H. H. Hoos, F. Neumann, and H. Trautmann. Automated Algorithm Selection: Survey and Perspectives. *Evolutionary Computation*, 27(1):3–45, 2019.

[106] J. Kotary, F. Fioretto, P. Van Hentenryck, and B. Wilder. End-to-end constrained optimization learning: A survey. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, 2021.

[107] A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.

[108] M. Laurent. Cuts, matrix completions and graph rigidity. *Mathematical Programming*, 79:255–283, 1997.

[109] C. Lavor, L. Liberti, and N. Maculan. Computational experience with the molecular distance geometry problem. In J. Pinter, editor, *Global Optimization: Selected Case Studies*, pages 213–225.

[110] C. Lavor, L. Liberti, and N. Maculan. Molecular distance geometry problem. In *History of Optimization*, pages 2305–2311. Springer US, Boston, MA, 2009.

[111] C. Lavor, L. Liberti, N. Maculan, and A. Mucherino. The discretizable molecular distance geometry problem. *Computational Optimization and Applications*, 52:115–146, 2012.

[112] C. Lavor, L. Liberti, N. Maculan, and A. Mucherino. Recent advances on the discretizable molecular distance geometry problem. *European Journal of Operational Research*, 219:698–706, 2012.

[113] J. Lee and L. Liberti. A matroid view of key theorems for edge-swapping algorithms. *Mathematical Methods of Operations Research*, 76:125–127, 2012.

[114] L. Liberti. Reformulations in Mathematical Programming: Definitions and systematics. *RAIRO - Operations Research*, 43(1):55—85, 2009.

[115] L. Liberti. Mathematical Programming, 2016. Lecture notes, École Polytechnique.

[116] L. Liberti. Undecidability and hardness in mixed-integer nonlinear programming. *RAIRO - Operations Research*, 53(1):81–109, 2019.

[117] L. Liberti. A new distance geometry method for constructing word and sentence vectors. In *Companion Proceedings of the Web Conference (DL4G Workshop)*, volume 20 of *WWW*, New York, 2020. ACM.

[118] L. Liberti. Advanced Mathematical Programming, 2021. Lecture notes, École Polytechnique.

[119] L. Liberti. Distance Geometry and Data Science. *TOP*, 28:271–339, 220.

[120] L. Liberti, G. Iommazzo, C. Lavor, and N. Maculan. A cycle-based formulation for the distance geometry problem. In C. Gentile, G. Stecca, and P. Ventura, editors, *Graphs and Combinatorial Optimization: from Theory to Applications*, volume 5 of *AIRO Springer Series*, pages 93–106. Springer, 2021.

[121] L. Liberti and S. Kucherenko. Comparison of deterministic and stochastic approaches to global optimization. *International Transactions in Operational Research*, 12:263–285, 2005.

[122] L. Liberti and C. Lavor. *Euclidean Distance Geometry: An Introduction*. Springer, New York, 2017.

[123] L. Liberti and C. Lavor. Open research areas in distance geometry. In A. Migalas and P. Pardalos, editors, *Open Problems in Optimization and Data Analysis*, volume 141 of *SOIA*, pages 183–223. Springer, New York, 2018.

[124] L. Liberti, C. Lavor, J. Alencar, and G. Abud. Counting the number of solutions of ^{k}DMDGP instances. In *Proceedings of the 1st International Conference on Geometric Science of Information*, pages 224–230. Springer, Berlin, 2013.

[125] L. Liberti, C. Lavor, and N. Maculan. A branch-and-prune algorithm for the molecular distance geometry problem. *International Transactions in Operational Research*, 15:1–17, 2008.

[126] L. Liberti, C. Lavor, N. Maculan, and F. Marinelli. Double Variable Neighbourhood Search with smoothing for the molecular distance geometry problem. *Journal of Global Optimization*, 43:207–218, 2009.

[127] L. Liberti, C. Lavor, N. Maculan, and A. Mucherino. Euclidean distance geometry and applications. *SIAM Review*, 56(1):3–69, 2014.

[128] L. Liberti, C. Lavor, A. Mucherino, and N. Maculan. Molecular distance geometry methods: from continuous to discrete. *International Transactions in Operational Research*, 18:33–51, 2010.

[129] L. Liberti, N. Mladenović, and G. Nannicini. *A good recipe for solving MINLPs*, pages 231–244. Annals of Information Systems. Springer, Boston, MA, 2009.

[130] L. Liberti and K. Vu. Barvinok's naive algorithm in distance geometry. *Operations Research Letters*, 46:476–481, 2018.

[131] C. Liebchen and R. Rizzi. A greedy approach to compute a minimum cycle basis of a directed graph. *Information Processing Letters*, 94:107–112, 2005.

[132] M. Locatelli. Simulated annealing algorithms for continuous global optimization. *Handbook of Global Optimization, Nonconvex Optimization and Its Applications*, 62, 2002.

[133] A. Lodi and A. Tramontani. Performance variability in Mixed-Integer Programming. *Tutorials in Operations Research*, 10:1–12, 2013.

[134] M. Lombardi, M. Milano, and A. Bartolini. Empirical decision model learning. *Artificial Intelligence*, 244:343–367, 2017.

[135] L. Lovász and M. Plummer. On minimal elementary bipartite graphs. *Journal of Combinatorial Theory B*, 23:127–138, 1977.

[136] M. Abadi *et al.* TensorFlow: Large-scale Machine Learning on heterogeneous systems. `http://tensorflow.org/`, 2015.

[137] M. Feurer *et al.* Efficient and robust automated machine learning. In *Proceedings of the 28th International Conference on Neural Information Processing Systems*, volume 2, pages 2755—2763, Cambridge, MA, USA, 2015. MIT Press.

[138] M. López-Ibáñez *et al.* The irace package: iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016.

[139] T. Malliavin, A. Mucherino, C. Lavor, and L. Liberti. Systematic exploration of protein conformational space using a distance geometry approach. *Journal of Chemical Information and Modeling*, 59:4486–4503, 2019.

[140] L. Mencarelli, Y. Sahraoui, and L. Liberti. A multiplicative weights update algorithm for MINLP. *EURO Journal on Computational Optimization*, 5:31–86, 2017.

[141] A. Micheli. Neural network for graphs: A contextual constructive approach. *IEEE Transactions on Neural Networks*, 20(3):498–511, 2009.

[142] V. Misic. Optimization of tree ensembles. *Operations Research*, 68(5):1605–1624, 2020.

[143] N. Mladenović and P. Hansen. Variable neighborhood search. *Computers & Operations Research*, 24(11):1097–1100, 1997.

[144] M. Mohri, A. Rostamizadeh, and A. Talwalkar. *Foundations of Machine Learning.* The MIT Press, 2nd edition, 2018.

[145] A. Mucherino, C. Lavor, and L. Liberti. Exploiting symmetry properties of the discretizable molecular distance geometry problem. *Journal of Bioinformatics and Computational Biology*, 10:1–15, 2012.

[146] A. Mucherino, C. Lavor, L. Liberti, and N. Maculan, editors. *Distance Geometry: Theory, Methods, and Applications.* Springer, New York, 2013.

[147] V. Nannen and A. E. Eiben. Relevance estimation and value calibration of evolutionary algorithm parameters. In *Proceedings of the 20th International Joint*

Conference on Artifical Intelligence, pages 975–980. Morgan Kaufmann Publishers Inc., 2007.

[148] Y. Nesterov and A. Nemirovski. *Interior-point polynomial algorithms in convex programming*. Society for Industrial and Applied Mathematics (SIAM), 1994.

[149] Q. Nguyen and M. Hein. The loss surface of deep and wide neural networks. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70, pages 2603—2612. JMLR, 2017.

[150] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer series in operations research and financial engineering. Springer, New York, NY, 2 ed. edition, 2006.

[151] P. Belotti *et al.* Branching and bounds tightening techniques for non-convex MINLP. *Optimization Methods and Software*, 24(4):597–634, 2009.

[152] P. Belotti *et al.* Mixed-integer nonlinear optimization. *Acta Numerica*, 22, 2013.

[153] P. Bonami *et al.* An algorithmic framework for convex mixed integer nonlinear programs. *Discrete Optimization*, 5(2):186–204, 2008.

[154] N. P. Padhy. Unit commitment: a bibliographical survey. *IEEE Transactions on Power Systems*, 19(2):1196–1205, 2004.

[155] K. Paton. An algorithm for finding a fundamental set of cycles of a graph. *Communications of the ACM*, 12(9):514–518, 1969.

[156] R. Pavón, F. Díaz, R. Laza, and V. Luzón. Automatic parameter tuning with a Bayesian case-based reasoning system. a case of study. *Expert Systems with Applications*, 36(2):3407–3420, 2009.

[157] P.-L. Poirion, S. Toubaline, C. D'Ambrosio, and L. Liberti. Algorithms and applications for a class of bilevel MILPs. *Discrete Applied Mathematics*, 2018.

[158] F. A. Potra and S. J. Wright. Interior-point methods. *Journal of Computational and Applied Mathematics*, 124(1):281–302, 2000.

[159] R. M. Anderson *et al.* Strong mixed-integer programming formulations for trained neural networks. *Mathematical Programming*, 183:3–39, 2020.

[160] M. V. Ramana. An exact duality theory for semidefinite programming and its complexity implications. *Mathematical Programming*, 77(1):129–162, 1997.

[161] J. R. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.

[162] H. Robbins and S. Monro. A stochastic approximation method. *Annals of Mathematical Statistics*, 22:400–407, 1951.

[163] G. Van Rossum and F. L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[164] S. Ruder. An overview of gradient descent optimization algorithms, 2016.

[165] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. *Learning Internal Representations by Error Propagation*, volume 1: Foundations, pages 318—-362. MIT Press, Cambridge, MA, USA, 1986.

[166] D. Saad and S. Solla. On-line learning in soft committee machines. *Physical review. E, Statistical physics, plasmas, fluids, and related interdisciplinary topics*, 52:4225–4243, 11 1995.

[167] J. Saxe. Embeddability of weighted graphs in k-space is strongly NP-hard. In *Proceedings of 17th Allerton Conference in Communications, Control and Computing*, pages 480–489, 1979.

[168] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61—-80, January 2009.

[169] G. Schaeffer. Random sampling of large planar maps and convex polyhedra. In *Proceedings of the 31st Annual ACM Symposium on the Theory of Computing*, STOC, pages 760–769, New York, 1999. ACM.

[170] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, Inc., USA, 1986.

[171] A. Schrijver. *Combinatorial Optimization*, volume 24. Springer, 2003.

[172] T. Serra, C. Tjandraatmadja, and S. Ramalingam. Bounding and counting linear regions of Deep Neural Networks. In J. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 4558–4566. PMLR, 2018.

[173] S. Seshu and M. B. Reed. *Linear Graphs and Electrical Networks*. Addison-Wesley, Reading, MA, 1961.

[174] J. Shlens. A tutorial on principal component analysis. *International Journal of Remote Sensing*, 51(2), 2014.

[175] A. Singer. Angular synchronization by eigenvectors and semidefinite programming. *Applied and Computational Harmonic Analysis*, 30:20–36, 2011.

[176] A. J. Smola and B. Schölkopf. A tutorial on Support Vector Regression. *Statistics and Computing*, 14(3):199–222, 2004.

[177] R. S. Sutton. Two problems with backpropagation and other steepest-descent learning procedures for networks. In *Proceedings of the 8th Annual Conference of the Cognitive Science Society*. Hillsdale, NJ, Erlbaum, 1986.

[178] T. Eren *et al.* Rigidity, computation, and randomization in network localization. *IEEE*, pages 2673–2684, 2004.

[179] T. Kavitha *et al.* Cycle bases in graphs: characterization, algorithms, complexity, and applications. *Computer Science Review*, 3:199–243, 2009.

[180] M. Tahanan, W. van Ackoooij, A. Frangioni, and F. Lacalandra. Large-scale Unit Commitment under uncertainty. *4OR - A Quarterly Journal of Operations Research*, 13(2):115–171, 2015.

[181] R. Taktak and C. D'Ambrosio. An overview on mathematical programming approaches for the deterministic unit commitment problem in hydro valleys. *Energy Systems*, 8(1):57–79, 2017.

[182] E. Talbi. *Metaheuristics: From Design to Implementation.* Wiley Publishing, 2009.

[183] Y. Tang, S. Agrawal, and Y. Faenza. Reinforcement learning for integer programming: Learning to cut. In H. Daumé III and A. Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 9367–9376. PMLR, 2020.

[184] M. Tawarmalani and N. V. Sahinidis. Global optimization of Mixed Integer Nonlinear Programs: A theoretical and computational study. *Mathematical Programming*, 99:563–591, 2004.

[185] V. Tjeng, K. Y. Xiao, and R. Tedrake. Evaluating robustness of Neural Networks with Mixed Integer Programming. In *Proceedings of the International Conference on Learning Representations*, 2019.

[186] C. Tsay, J. Kronqvist, A. Thebelt, and R. Misener. Partition-based formulations for mixed-integer optimization of trained relu neural networks. https://arxiv.org/abs/2102.04373, 2021.

[187] Vapnik V. N. and A. Y. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and its Applications*, 16(2):264–280, 1971.

[188] V. V. Kovačević-Vujčić *et al.* TABU search methodology in global optimization. *Computers and Mathematics with Applications*, 37(4):125–133, 1999.

[189] V. N. Vapnik. *The nature of statistical learning theory.* Springer-Verlag New York, Inc., 1995.

[190] S. Varma and R. Simon. Bias in error estimation when using cross-validation for model selection. *BMC Bioinformatics*, 7(91), 2006.

[191] J. P. Vielma, S. Ahmed, and G. Nemhauser. Mixed-Integer Models for Nonseparable Piecewise-Linear Optimization: Unifying Framework and Extensions. *Operations Research*, 58(2):303–315, 2010.

[192] W. van Ackooij *et al.* Shortest path problem variants for the Hydro Unit Commitment problem. *Electronic Notes in Discrete Mathematics*, 69:309–316, 2018. Joint EURO/ALIO International Conference 2018 on Applied Combinatorial Optimization (EURO/ALIO 2018).

[193] B. Wilder, B. Dilkina, and M. Tambe. Melding the data-decisions pipeline: Decision-focused learning for combinatorial optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1658–1665, 2019.

[194] I. H. Witten, E. Frank, and M. A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques.* Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, Amsterdam, 3 edition, 2011.

[195] L. Xu, H. H. Hoos, and K. Leyton-Brown. Hydra: Automatically configuring algorithms for portfolio-based selection. In *Proceedings of the National Conference on Artificial Intelligence*, volume 1, 2010.

[196] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32(1):565–606, 2008.

[197] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Hydra-MIP: Automated algorithm configuration and selection for mixed integer programming. In *Proceedings of the RCRA workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion at the International Joint Conference on Artificial Intelligence*, 2012.

[198] Y. N. Dauphin *et al.* Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Proceedings of the 27th International Conference on Neural Information Processing Systems*, volume 2, pages 2933—-2941, Cambridge, MA, USA, 2014. MIT Press.

[199] X. Yu and M. Gen. *Introduction to evolutionary algorithms*. Springer Science & Business Media, 2010.

[200] G. Zarpellon, J. Jo, A. Lodi, and Y. Bengio. Parameterizing branch-and-bound search trees to learn branching policies. *ArXiv*, abs/2002.05120, 2021.

[201] J. Zhu, S. Rosset, T. Hastie, and R. Tibshirani. 1-norm Support Vector Machines. In *Proceedings of the 16th International Conference on Neural Information Processing Systems*, pages 49—-56, Cambridge, MA, USA, 2003. MIT Press.

ECOLE DOCTORALE

Title : Algorithmic configuration by learning and optimization **Keywords :** algorithm configuration, machine learning, mathematical programming, optimization solvers, parameter tuning, distance geometry

Abstract : The research topics described in this Ph.D. thesis lie at the intersection of Machine Learning (ML) and Mathematical Programming (MP). The main contributions concern the Algorithm Configuration Problem (ACP) and the Distance Geometry Problem (DGP).

After providing introductions to MP and ML, we survey the ACP literature. Given a configurable algorithm A and an input P for A, the ACP addresses the issue of identifying the parameter configuration c^* of A ensuring the best algorithmic performance $p_{\mathcal{A}}$ in solving P. Since most algorithms have a very large number of configurable parameters, this is usually a very hard optimization problem to tackle in practice.

We propose two novel MP-driven methodologies, using ML paradigms as elements appearing in an MP formulation, to address the ACP. Since algorithmic performance is usually a black-box function, we first train an ML predictor to estimate it. Then, we produce an explicit description of the learned approximation. We embed this encoding into an MP formulation of the ACP : its objective optimizes the ML-derived approximation ; its constraints define the set of feasible configurations, and, potentially, other conditions on the trained predictor. Upon the arrival of a new instance P', we solve the ACP by MP techniques, to retrieve the configuration optimising the estimated algorithmic performance for P'. This framework can work with virtually any ML paradigms. We employ our approaches to tune the parameters of an optimization solver, deployed on instances of a hard mixed-integer linear programming problem.

In the last part of the manuscript, we consider a methodology for solving the DGP, i.e., for finding a realization of a weighted graph in a Euclidean space of given dimension. A customary approach is to solve an MP formulation to determine the position of the vertices in the given Euclidean space. We propose a new MP formulation where, instead, we consider the cycles of the graph, and we decide the length of the segments modelling the edges in each cycle. Our research is partly motivated by the fact that it can serve as a graph embedding methodology, in view of applying vector-based ML paradigms to graphs.

Titre : Configuration algorithmique par apprentissage et optimisation **Mots clés :** configuration des algorithmes, machine learning, programmation mathémathique, solveurs d'optimisation, géométrie des distances

Résumé : Les sujets de recherche décrits dans cette thèse se situent à l'intersection du Machine Learning (ML) et de la Programmation Mathématique (PM). Les principales contributions concernent le problème de la configuration des algorithmes (ACP) et le problème de la géométrie des distances (DGP).

Après avoir présenté la PM et le ML, nous passons en revue la littérature sur l'ACP. Étant donné un algorithme paramétré A et une entrée P pour A, l'ACP aborde la sélection de la configuration de paramètres c^* de A assurant la meilleure performance $p_{\mathcal{A}}$ pour résoudre P. Comme la plupart des algorithmes ont un très grand nombre de paramètres, cela constitue un problème d'optimisation généralement très difficile à résoudre dans la pratique.

Nous proposons deux nouvelles méthodologies, fondées sur la PM et sur des paradigmes de ML, pour résoudre l'ACP. Puisque la performance algorithmique est généralement une fonction boîte noire, nous construisons d'abord un prédicteur de ML pour estimer le comportement de A. Puis, nous produisons une description explicite de l'approximation apprise, et nous l'intégrons dans une formulation MP de l'ACP : l'objectif de cette formulation optimise l'approximation apprise ; ses contraintes définissent l'ensemble des configurations admissibles, et, potentiellement, d'autres conditions sur le prédicteur entraîné. A l'arrivée d'une nouvelle entrée P', nous résolvons l'ACP par des techniques de PM, pour trouver la configuration optimisant la performance algorithmique estimée sur P'. Ce cadre est compatible avec n'importe quel paradigme de ML. Nous utilisons nos approches pour régler les paramètres d'un solveur d'optimisation, déployé sur des instances d'un difficile problème linéaire mixte en nombres entiers.

Dans la dernière partie du manuscrit, nous considérons une méthodologie pour résoudre le DGP, c'est à dire pour trouver une réalisation d'un graphe pondéré dans un espace euclidien de dimension donnée. Une approche habituelle consiste à résoudre une formulation de PM pour déterminer la position des sommets dans l'espace euclidien donné. Nous proposons une nouvelle formulation de PM où, à la place, nous considérons les cycles du graphe, et nous décidons de la longueur des segments modélisant les arêtes de chaque cycle. Notre recherche est en partie motivée par le fait qu'elle peut servir de méthodologie de plongement de graphes, en vue d'appliquer aux graphes des paradigmes de ML basés sur les vecteurs.

Institut Polytechnique de Paris
91120 Palaiseau, France