Artificial Intelligence 244 (2017) 343–367

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

Empirical decision model learning

Michele Lombardi [a,*], Michela Milano [a], Andrea Bartolini [b]

[a] DISI, University of Bologna, Italy
[b] DEI, University of Bologna, Italy

ARTICLE INFO

Article history:
Received in revised form 21 December 2015
Accepted 10 January 2016
Available online 13 January 2016

Keywords:
Combinatorial optimization
Machine learning
Complex systems
Local search
Constraint programming
Mixed integer non-linear programming
SAT modulo theories
Artificial neural networks
Decision trees

ABSTRACT

One of the biggest challenges in the design of real-world decision support systems is coming up with a good combinatorial optimization model. Often enough, accurate predictive models (e.g. simulators) can be devised, but they are too complex or too slow to be employed in combinatorial optimization.

In this paper, we propose a methodology called Empirical Model Learning (EML) that relies on Machine Learning for obtaining *components* of a prescriptive model, using data either extracted from a predictive model or harvested from a real system. In a way, *EML can be considered as a technique to merge predictive and prescriptive analytics*.

All models introduce some form of approximation. Citing G.E.P. Box [1] "Essentially, all models are wrong, but some of them are useful". In EML, models are useful if they provide adequate accuracy, and if they can be *effectively exploited by solvers for finding high-quality solutions*.

We show how to ground EML on a case study of thermal-aware workload dispatching. We use two learning methods, namely Artificial Neural Networks and Decision Trees and we show how to encapsulate the learned model in a number of optimization techniques, namely Local Search, Constraint Programming, Mixed Integer Non-Linear Programming and SAT Modulo Theories. We demonstrate the effectiveness of the EML approach by comparing our results with those obtained using expert-designed models.

© 2016 Elsevier B.V. All rights reserved.

1. Introduction

Advances in Combinatorial Optimization methods in the last decades have enabled their successful application to a broad range of industrial problems. Many of such approaches rely on the availability of some declarative system description. This typically consists of a hand-crafted mathematical model, obtained after thorough discussion with the domain experts by introducing some simplifying assumptions.

Devising a good model is a complex task, especially challenging when dealing with real-world systems. A good model finds a proper balance between model complexity and model accuracy: on the one hand, excessive simplification may lead to "optimal" – but completely useless – solutions. On the other hand, incorporating too many details results in extremely hard computational issues. Despite this, a number of successful optimization approaches have been proposed in the literature and

* Corresponding author.
E-mail addresses: michele.lombardi2@unibo.it (M. Lombardi), michela.milano@unibo.it (M. Milano), a.bartolini@unibo.it (A. Bartolini).

http://dx.doi.org/10.1016/j.artint.2016.01.005
0004-3702/© 2016 Elsevier B.V. All rights reserved.

applied to real-life industrial problems, enabling in many cases[1] huge savings in terms of resources (time, money, machines, energy).

Nevertheless, many systems are still impervious to approaches such as Mixed Integer Linear Programming (MILP), Constraint Programming (CP), or SAT (propositional SATisfiability) and this is often due to modeling issues. There are basically two kinds of "high-complexity systems" that are out-of-reach for traditional combinatorial approaches: (1) Complex Systems, which exhibit phenomena that emerge from a collection of interacting objects capable of self-organization and affected by memory or feedback; and (2) physical systems whose dynamic model is known, but its embedding in a combinatorial model is computationally intractable.

A very common way for supporting decision-making in these systems is to design a predictive model (e.g., a simulator) based on real data and to use it via what-if analysis (see [2] for a recent reference). In what-if analysis, the decision maker repeatedly feeds scenarios (i.e. sets of decisions) to the predictive model to extract the values of certain observables of interest (e.g. quality measures). Inevitably, only a limited number of scenarios is investigated, and then the decision maker commits to the one showing the best behavior. In combinatorial problems the decision space might be so large that selecting scenarios manually or in isolation results in far-from-optimal choices.

The aim of this paper is to bring such high-complexity systems within the reach of combinatorial decision making and optimization. The idea is to use Machine Learning (ML) to learn an approximate relation between decisions and their impact on the system. In particular, we devise a methodology, called *Empirical Model Learning (EML)* that: (1) learns relations between decidables and observables[2] from data, and (2) encapsulates these relations into components of an optimization model, namely objective functions or constraints. The training data for the learning techniques can be harvested from the real system or extracted from a predictive model (e.g. simulator). The integration into model components is not merely a matter of encoding, since in some cases an operational semantics for the efficient use of the component should be defined.

The ability to integrate Machine Learning models in combinatorial optimization has the potential to play a major role in *bridging the gap between predictive and prescriptive analytics*. An EML based system may be capable of suggesting optimal decisions in a complex real-world setting, by taking advantage of recent developments in big data analysis and predictive model design.

This paper provides three main contributions. First, we introduce the Empirical Model Learning approach in a general fashion. Second, we present a number of methods for embedding Machine Learning models (namely Decision Trees and Artificial Neural Networks) into several optimization techniques (Local Search, Mixed Integer Non-Linear Programming, Constraint Programming, SAT Modulo Theories). Some of our embedding techniques have been presented in previous papers of ours [4,5]. Third, we show that despite the main idea behind EML being very simple, its application requires some care for obtaining an effective optimization approach. We highlight the main difficulties and suggest possible solutions by applying the EML approach on two practical examples.

As motivating (and running) examples, we use two thermal-aware workload dispatching problems, defined over an experimental multicore Intel CPU called "Single-chip Cloud Computer" (SCC, see [6]). Both problems consist in mapping a set of heterogeneous jobs on the platform cores so as to maximize some cost metric involving the platform efficiency. The efficiency of each core is affected by a number of complex factors including the thermal dynamics of the chip, the workload distribution, and the presence of low-level schedulers and thermal controllers. Although an accurate system simulator for the platform is available, it cannot be inserted into a decision model due to its high complexity and large run time. We show that EML allows considerable improvements over simpler optimization approaches either based on expert-designed heuristics, or on expert-designed models refined via function fitting.

The paper is structured as follows: in Section 2 we provide a comparative analysis of related work. In Section 3 we introduce the example problems. In Section 4 we give a brief overview of the EML approach. Section 5 presents techniques for embedding Machine Learning models into Combinatorial Optimization models. Sections 6 and 7 discuss respectively how to design the *core combinatorial structure* of the optimization problem, and how to extract a system model from data: in both cases, our example problems are employed to present the process. We provide experimental results in Section 8 and concluding remarks in Section 9.

2. Comparative analysis of related work

The EML approach combines elements of Combinatorial Optimization, Machine Learning, and Complex Systems/Simulation. In this section we provide a brief overview of approaches related to the integration of such research fields.

Loosely related approaches Researchers have been interested for a long time in the integration of optimization techniques in Machine Learning. This is not surprising, given that training problems are fundamentally (very peculiar) optimization problems. Works such as [7,8] have studied the core optimization problems in ML algorithms and proposed efficient methods

[1] The reader may find some examples on the web page dedicated to the Franz Edelman Award at https://www.informs.org/Recognize-Excellence/Franz-Edelman-Award.

[2] The names decidables and observables have been suggested by Peter Flach [3].

for extracting knowledge from huge volumes of data. Other works (e.g. [9,10] and those presented in [11]) have applied Constraint Programming to Machine Learning tasks.

In the optimization community, a substantial effort has been recently dedicated to using learning to improve a solution approach. Clustering methods have been employed for automatic algorithm selection (e.g. [12]). Several Machine Learning techniques have been used for predicting the run time of optimization algorithms, once again with the aim to perform algorithm selection (e.g. [13,14]). A few works have focused on learning customized optimization problem instances for testing new techniques (see [15]).

Constraint acquisition Some papers [16–18] have focused on learning a set of constraints that match a number of positive and negative examples, a task known as *constraint acquisition*. An extension to these approaches is represented by the QuAcq system [19], which requires only partial queries on subsets of problem variables (with no need of positive examples). Other constraint acquisition approaches are surveyed in [20]. All such approaches attempt to combine constraints from a given library to build a model that is compatible with the available data. Conversely, in EML the emphasis is on enabling the integration of a "standard" Machine Learning model (e.g. a Neural Network) into a combinatorial problem. Our approach requires an additional design effort, but it is also more flexible and better suited for dealing with practical applications, especially in cases where a good Machine Learning model is already available.

Many constraint acquisition approaches make use of *active learning*: the candidate solutions found by a solver are evaluated (e.g. via a simulator) and then serve as new examples for the training set. This kind of approach allows one to tune the model for the specific instance being solved. As a drawback, the response times can become impractically large if evaluating a data point is expensive. Moreover, if a simulator is not available, then active learning would require one to deploy the *intermediate* solutions on the real system, which may be unreasonable in practice. In this work, we assume that the training is done entirely off-line. Integrating active learning in our method is possible, but left as a subject of future research.

Surrogate models Surrogate models (see [21] for an excellent overview) are approximate system models. They are typically employed on design problems for which declarative models are difficult to obtain, but simulation is viable. A surrogate model takes the form of a function with pre-defined structure and unknown parameters, or of a combination of kernel functions (e.g. Gaussian processes). Surrogate models are tuned over a training set and then are typically employed for blackbox optimization (see the next paragraph). In some approaches (e.g. ALAMO [22]) optimization and simulation are jointly employed to explore alternative surrogate model structures and tune the parameters.

The idea of using surrogate models is closely related to the one we propose, but there are important differences in terms of focus, scope, and (ideal) generality. Most works on surrogate models focus: (1) on a specific class of Machine Learning techniques, e.g. relatively simple compositions of non-linear, continuous, functions (only a minority of works have considered Artificial Neural Networks [23,24]); (2) on specific problems, typically with continuous variables and range constraints; and (3) on specific solution techniques, e.g. Genetic Algorithms, blackbox/derivative-free optimization, sometimes Mixed Integer Non-Linear Programming. Conversely, in EML we aim at enabling the use of as many Machine Learning techniques as possible, within as many optimization methods as possible: the goal is having the ability to choose the most adequate solution approach for each problem. As a consequence, our focus is mostly on *handling the integration* of Machine Learning models in optimization: for example, we emphasize the importance of model embedding techniques (in Section 5), and of methods to exploit the structure of the extracted model for boosting the search process. This paper will specifically focus on problems, Machine Learning models, and techniques that are outside the typical scope of surrogate models.

The LION approach Off-line learning is employed in the LION approach [25] to extract an approximate cost function from abundant data. The authors stress that there are practical cases (e.g. when user preferences are involved) where there is no obvious numeric approach to rank solutions. In this situation, using historical data is the only way to obtain an approximate cost function without repeatedly querying the user at solution time. The LION approach relies on model fitting to obtain close-to-optimal solutions: this makes it difficult to target decision problems with a complex combinatorial structure.

Black box optimization Black-box optimization approaches are concerned with finding solutions for optimization problems having cost or constraint functions with unknown structure: the typical case is that of systems lacking a declarative model, but for which a simulator is available. This is the same use case of surrogate models, which in fact are often employed in black box optimization.

If the black box function is fast enough to evaluate, then local search approaches, meta-heuristics, or Genetic Algorithms can be directly applied (e.g. [26]). If the black box evaluation is expensive to evaluate (this is frequent with simulators), then it becomes necessary to employ more advanced techniques to limit the number of simulator calls.

Most of the methods for black box optimization with expensive cost functions have been developed in the field of derivative free optimization. Within that field, the book [27] makes a distinction between Directional Search and Model-based approaches. Informally speaking, Directional Search methods (see the survey by [28]) rely on discretization to limit the number of black box evaluations. Model-based approaches, instead, employ active learning to fit an internal surrogate

model (often referred to as *response surface*). The internal model is used to guide the search process. In [29] and in Simulation for Optimization [30] the internal model is a stochastic process, which allows the search algorithm to take into account both the estimated solution quality and the estimated model accuracy in order to identify promising solutions. The OptQuest [31] system integrates in a closed loop simulation and meta-heuristics and relies on a Neural Network for quick, approximate, solution checking.

Black box optimization approaches differ from EML in three important regards. The most important difference is that most black-box optimization approaches are designed for problems without a complex combinatorial structure. Hence, they are likely to be ineffective (or inapplicable) for problems with discrete variables and non-trivial constraints. This is exactly the class of problems where techniques such as CP, MILP, or SAT (i.e. the most interesting for our technique) tend to provide the best results. Second, black-box optimization often relies on performing simulation during the search process: this may be excessively time-consuming, or even impossible if a simulator is not available. In EML, the simulation time has no direct impact on the solver performance, and the model extraction can rely on historical data, or on experiments performed on the real system. Third, in EML the function that describes the system behavior is not a black box: on the contrary, the structure of the Empirical Model is well known, and we wish to exploit it for boosting the search process.

3. Motivating example

As a case study and motivating example, we consider two problems related to thermal-aware workload dispatching over an experimental multicore CPU by Intel, called Single-chip Cloud Computer (SCC [6]). The two problems share the same combinatorial structure, but are different w.r.t. the objective function and the observables of interest. In this section we describe the problems, identify the critical difficulties for defining a model, and outline possible solution approaches.

Problem description The SCC platform has 24 dual-core tiles arranged in a 4×6 grid (overall, 48 cores in a 8×6 grid). Each core runs an independent instance of the Linux kernel. Inter-tile communication occurs via message passing on a network interface, hence tasks cannot easily migrate between cores. The chip is designed to accept job batches from an external node (e.g. a separate computer). Due to the large number of cores packed on a single silicon die, the platform is prone to overheating. Since the chip is a prototype, Intel has chosen to make it thermally stable via an overly large (and noisy) fan, rather than by implementing a more advanced thermal controller.

With the aim of studying temperature control policies on SCC, researchers at the University of Bologna (the domain experts in our case) have devised an accurate simulator based on Matlab and the Hotspot thermal modeling tool [32]. The simulator characterizes the jobs in terms of their CPI value (Clocks Per Instruction) at maximum frequency: jobs with low CPI make a more intense use of the CPU and generate more heat, whereas jobs with high CPI are comparatively colder. On each core of the simulated platform, two thermal control approaches have been introduced:

1. A preemptive thermal-aware scheduler that interleaves the execution of “hot” and “cold” jobs in an effort to keep the temperature stable. Jobs are classified at run time based on their effect on the current temperature.
2. A thermal controller that dramatically lowers the core frequency if the temperature exceeds a safety threshold.

The jobs are assumed to run indefinitely: hence, the domain experts have decided to avoid overloading by running the same number of jobs on each core.

Problem variants Both our dispatching problems consist in mapping a set of heterogeneous jobs on the platform cores so as to maximize an objective related to the core efficiencies. Due to the presence of a thermal controller, the efficiency of each core depends on its temperature, and on a number of complex factors including: (1) the workload, (2) the temperature of the other cores, (3) the core position on the die, (4) the action of the thermal-aware scheduler and of the threshold controller. As already mentioned, we consider two problem variants:

1. In the first variant (referred to as WDP_{bal}), the goal is to balance the platform efficiency and to avoid the occurrence of hot spots (abnormally warm cores). Together with the domain experts, we have formalized this objective as that of maximizing the worst-case core efficiency.
2. In the second variant (referred to as WDP_{max}), the goal is to have a portion of the platform as large as possible that operates at high efficiency. Together with the domain experts, we have formalized this objective as that of maximizing the number of cores having an efficiency larger than a certain threshold.

Base models As a preliminary step for defining a solution approach, we introduce a base model for each of our example problems. In both the base models, the relation between the mapping decisions and the efficiency is captured by means of generic functions. Formally, let n and m respectively be the number of jobs and cores ($m = 48$ for SCC). Let us introduce a

set of binary variables x_{ik} such that $x_{ik} = 1$ iff job i is mapped on core k and 0 otherwise. Then a possible formulation for WDP_{bal} is:

Base Model for the WDP_{bal}

$$\max z = \min_{k=0..m-1} \left(eff_k\right) \tag{1}$$

$$\text{subject to: } \boxed{eff_k = h_k^{bal}(x)} \qquad \forall k = 0..m-1 \tag{2}$$

$$\sum_{k=0}^{m-1} x_{ik} = 1 \qquad \forall i = 0..n-1 \tag{3}$$

$$\sum_{i=0}^{n-1} x_{ik} = \frac{n}{m} \qquad \forall k = 0..m-1 \tag{4}$$

$$x_i \in \{0, 1\} \qquad \forall i = 0..n-1 \tag{5}$$

Constraints (3) ensure that each job is mapped on a single core and Constraints (4) force the same number of jobs (i.e. n/m) to run on each core.[3]

Constraints (2) define the behavior of the target CPU. This is done by relying on a set of functions $h_k^{bal} : \{0, 1\}^n \rightarrow (0, 1]$, each of which associates a mapping of all jobs to a value for the eff_k variable, representing the efficiency of core k. A base model for the second problem variant can be formulated as follows:

Base Model for the WDP_{max}

$$\max z = \sum_{k=0}^{m-1} heff_k \tag{6}$$

$$\text{subject to: } \boxed{heff_k = h_k^{max}(x)} \qquad \forall k = 0..m-1 \tag{7}$$

$$\text{Constraints (3), (4), and (5)} \tag{8}$$

The model is identical to the one for the WDP_{bal}, except for the cost function and the functions h_k^{max} that define the system behavior. In particular, we assume here that each $heff_k$ is binary and such that $heff_k = 1$ iff the efficiency of core k is above the threshold value. Therefore, we have that each h_k^{max} is a function $h_k^{max} : \{0, 1\}^n \rightarrow \{0, 1\}$.

Modeling the system behavior The critical step for defining a solution approach for the WDP_{bal} and the WDP_{max} is finding a suitable embodiment of the h^{bal} and h^{max} functions. Here we survey the main alternatives.

First, the h^{bal} and h^{max} functions can be evaluated in an exact fashion by simply *running the SCC simulator*. This is the typical scenario in black box optimization (see Section 2), and a viable approach when using GAs or Local Search as solution methods. Unfortunately, in our case *each simulation run requires several minutes, while the acceptable response time for the dispatcher is in the order of (tens of) seconds*. Amortizing the simulation time (by limiting simulator calls or relying on an internal model) is not enough to solve the issue, and therefore direct evaluations via the simulator are simply not viable.

Second, it is possible to *use a heuristic measure* as a "proxy" for the exact system behavior. Heuristics of this kind are often easy to embed in a combinatorial model. More importantly, such heuristics are easy to define for domain experts, who typically base their decisions on some kind of simple rule. For example, in the case of the WDP_{bal}, the designer of the simulator suggested to use the average CPI (Clocks Per Instruction) of the jobs mapped on a core as a proxy for its efficiency. This approach has been considered in our experimentation in Section 8. The main drawback of using a heuristic is that it becomes usually impossible to a-priori quantify the degree of approximation. This is undesirable in general, and may have an adverse affect on the solution quality.

Third, it is possible to define the h^{bal} and h^{max} functions by using (e.g.) linear regression for *fitting an expert-design model*: we rely on the knowledge of the domain expert for defining the model structure, while we extract the parameter values from data. The accuracy of the model can be evaluated over the training set or on a separate test set. We have considered this approach (which is in fact a simple form of model learning) in our experimentation.

[3] For sake of simplicity we assume n to be a multiple of m. In the general case, computing the features that will be introduced in Section 7.2 requires one to introduce some (simple) non-linear constraints. This can be done via standard modeling constructs in Local Search or MINLP. In CP, the computation can be done via the Weighted Average Constraint from [33].

Finally, *we can use Empirical Model Learning* and employ Machine Learning to *extract from data both the structure and the parameters of the system model*. This approach is applicable even when the structure of h^{bal} and h^{max} is not obvious, and it will be discussed in detail in the remainder of the paper.

4. The Empirical Model Learning approach

In EML, we are interested in solving optimization problems defined over high-complexity systems, which typically have the following structure:

$$\min f(x, z) \tag{9}$$

$$\text{s.t. } g_j(x, z) \qquad \forall j \in J \tag{10}$$

$$z = h(x) \tag{11}$$

$$x_i \in D_i \qquad \forall x_i \in x \tag{12}$$

where x is a vector of decision variables x_i with domain D_i, and z is a vector of observables related to the target system. We make no special assumption on the domains D_i in the general case. The cost function f may depend on both the decision variables and the observables.

All problem variables may be subject to constraints, represented as logical predicates $g_j(x, z)$. The predicates may correspond to classical inequalities from Mathematical Programming, or to combinatorial restrictions, such as Global Constraints in Constraint Programming (e.g. ALLDIFF, ELEMENT). Equations (9), (10), and (12) represent the *core combinatorial structure* of the optimization problem. The $h(x)$ function describes the (approximate) behavior of the high-complexity system and specifies how the observables depend on the decision variables. The function h corresponds to the encoding of the *Empirical Model*, obtained via Machine Learning.

Designing an optimization approach based on EML requires to take care of three main activities:

1. Defining the core combinatorial structure of the problem.
2. Obtaining a Machine Learning model.
3. Embedding the Empirical Model in the combinatorial problem.

The third step is the critical one, and it will be the first to be presented (in Section 5). It is possible to embed a Machine Learning model into a combinatorial problem only if a suitable encoding has been defined. Such an encoding should be *designed so that it can be exploited by the optimization approach* for boosting the search process (e.g. via bound computation or constraint propagation).

Step 1 consists in defining a combinatorial model and Step 2 is a classical Machine Learning task (regression or classification). Both activities are non-trivial and may be time consuming, but they have been extensively studied in the past. In the context of EML, they require some special attention: they will be discussed in Sections 6 and 7, using the WDP_{bal} and WDP_{max} as running examples.

5. Embedding the Empirical Model

Embedding the extracted EM into an optimization model requires: (1) to encode the Empirical Model in terms of variables and constraints; (2) to define an *operational semantics* for such an encoding.

By operational semantics we refer to any procedure that can improve the optimization process by reasoning over the EM. This operational semantics may be provided implicitly by the underlying solver (e.g. convex envelope bounding in some MINLP solvers) or may be defined explicitly together with the encoding (e.g. ad hoc filtering algorithms in CP).

In this section, we will show embedding techniques for two types of Machine Learning models – namely Artificial Neural Networks (ANNs) and Decision Trees (DTs) – and four Combinatorial Optimization approaches – namely Local Search (LS), Mixed Integer Non-Linear Programming (MINLP), Constraint Programming (CP), and SAT Modulo Theories (SMT). The exact combinations that we consider are reported in Table 1: we provide no technique for integrating ANNs in SMT, because solvers with support for non-linear theories are not widely available. We did not try to encode Decision Trees in MINLP, because it would require extensive linearization of disjunctions, likely leading to poor bounds. Additionally, the interested reader may find a technique for embedding Random Forests in CP in work [5], by some of the authors of this paper. All our encoding techniques are general and not restricted to our example problems.

As a baseline case, we consider embedding a Machine Learning model in Local Search. The core idea in Local Search is to improve iteratively an incumbent solution, by exploring (evaluating) a set of neighbor solutions. Local Search methods are originally designed for problems with discrete variables (unlike many blackbox optimization methods), and they can deal with non-trivial constraints (either via violation-based cost functions or by incorporating the constraints in the neighborhood definition).

LS approaches require only the ability to *evaluate* the cost and constraint functions, and they always manipulate *fully-instantiated solutions*. For these reasons, embedding a Machine Learning model requires simply to implement a function

Table 1
Supported combinations of learning and optimization techniques.

	LS	MINLP	CP	SMT
Artificial Neural Networks	×	×	×	
Decision Trees	×		×	×

evaluator. As a downside, LS approaches are limited in their ability to exploit the model structure for boosting the search process.

5.1. Embedding artificial neural networks in optimization methods

We briefly recall that ANNs (see [34] for a comprehensive reference) are computational systems consisting in networks of basic units called *artificial neurons*. Technically, an artificial neuron is a function with vector input x and scalar output y, corresponding to the equation:

$$y = \phi\left(b + \sum_i w_i x_i\right) \tag{13}$$

where x_i denotes a single element in x, the w_i terms are the input *weights* (obtained via training), and b is a *bias*. The term ϕ is a monotonic, non-decreasing, often non-linear, *activation function*. Its argument is known as the neuron *activity*.

5.1.1. Embedding ANNs in MINLP

Mixed Integer Non-Linear Programming (MINLP, see [35]) is a field of Mathematical Programming that is concerned with finding extreme points of non-linear functions subject to linear, non-linear, or integrality constraints. Modern MINLP solvers can take advantage of the problem structure (constraints and cost function) via convex envelope approximation, linearization, cutting planes, constraint propagation, and branching.

An ANN can be embedded in a MINLP model by introducing variables to model the input and output of each neuron, and then by directly inserting the neuron equations in the model. This is easy as long as the considered MINLP solver supports the activation functions employed in the neurons. Once the EM has been encoded, its equations will be automatically taken it into account by the solver for computing bounds and generating cuts.

There are a few aspects that deserve some care. First, several MINLP solvers rely on convexity for providing globally optimal results. Any such MINLP solver will converge to a local optimum (possibly different from the global optimum), if the neurons use non-linear functions and the network has enough layers.

Second, numerical stability may be an issue. For example, some MINLP solvers need at some point to invert the model functions. In principle, most activation function types can be inverted. In practice, however, due to the finite precision of the underlying machine, the inversion may be possible only on a restriction of the function domain, possibly leading to software crashes and missed solutions. The issue can be addressed by restricting the domain of the input/output variables of each neuron: this approach may accidentally eliminate a high-quality solution, but the risk should be very low.

5.1.2. Embedding ANNs in CP

Constraint Programming (CP, see [36]) is an Artificial Intelligence technique designed to solve Constraint Satisfaction Problems (CSP) and Constraint Optimization Problems (COP). A CSP is defined as a set of variables, subject to a set of constraints. Each constraint has an associated filtering algorithm that can prune, at search time, provably infeasible values from the variable domains. Pruning a value may trigger other filtering algorithms in a process known as propagation. A constraint solver combines filtering, propagation, and search (which is highly customizable) to find solutions for a combinatorial problem. Optimization can be performed by dynamically adding bounding constraints whenever a feasible solution is found.

We have shown in [4] that an ANN can be encoded in CP by introducing additional variables like in MINLP, and then by encoding each neuron as a global *Neuron Constraint*. A Neuron Constraint has signature:

$$\phi(y, x, w, b) \tag{14}$$

where ϕ denotes the activation function type, y is the output variable, x is the vector of input variables, w is the vector of weights, and b is the bias. Using a constraint to model each neuron allows one to encode complex networks (even recurrent ones) with a limited number of basic modeling components (i.e. a constraint for each type of activation function).

A Neuron Constraint maintains Bound Consistency on Equation (13), i.e. its filtering algorithm is capable of pruning the x and y variables so that the extremes of their domains are guaranteed to be consistent. Bound Consistency can be enforced for a Neuron Constraint by filtering the neuron activity and the activation function separately. Formally, we consider the decomposition:

$$y = \phi(y') \tag{15}$$

$$y' = b + \sum_i w_i \cdot x_i \tag{16}$$

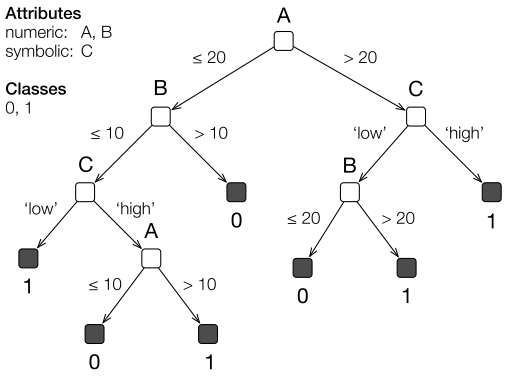

Fig. 1. An example of Decision Tree.

where y' is a real-valued variable representing the neuron activity. The variable is introduced to explain the filtering method, and typically it does not appear in the model. Equation (16) is a linear expression, for which classical filtering techniques exist. Since the activation function is monotonic and non-decreasing, Bound Consistency on Equation (15) can be enforced by means of the following rules:

$$\overline{y}' \text{ updated} \longrightarrow \quad \overline{y} = \min(\overline{y}, \phi(\overline{y}')) \tag{17}$$

$$\overline{y} \text{ updated} \longrightarrow \quad \overline{y}' = \min(\overline{y}', \phi^{-1}(\overline{y})) \tag{18}$$

where the notation $\overline{y}, \overline{y}'$ denotes the maximum in the domain of y and y'. The key idea is that whenever the domain maximum of y' is narrowed, this triggers a reduction of the domain maximum of y, and vice-versa. The rules for filtering the domain minima are analogous.

We recall that the ϕ function may be invertible in practice for a restricted range of values, due to the finite precision of the underlying machine. This issue can be addressed by replacing $\phi^{-1}(\overline{y})$ in Equation (18) with the expression $\max\{v \in \mathbb{R} : \phi(v) = \overline{y}\}$. Moreover, since many constraint solvers do not provide support for real-valued variables, it is often necessary to use integer variables and a finite-precision representation. More details about practical implementation issues can be found in [4].

Using a global constraint for each neuron is not the only viable CP encoding. An alternative approach consists in using a single global constraint to capture a whole network: this method is less flexible, but can provide stronger filtering. In [37] one of the authors of this paper has devised a global propagator based on a Lagrangian relaxation for a very common class of networks, i.e. two-layer, feed-forward ANNs. For simplicity, however, in this paper we limit ourselves to the original approach using one constraint per neuron.

5.2. Embedding Decision Trees in optimization methods

Decision Trees (DT) are a type of Machine Learning model typically employed for classification tasks (see [38] for a comprehensive overview). Each leaf of a DT is labeled with a *class*. Each node is labeled with one of a set of *attributes* x_i that are used to described the DT input. Attributes can be numeric or symbolic. The outgoing branches of a node are labeled with conditions over its attribute. The conditions form a partition of the attribute domain: every branch b_j over a symbolic attribute is labeled with a set $L(b_j)$ of acceptable symbolic values; every branch over a numeric attribute is labeled with a splitting condition over a threshold $\theta(b_j)$, in the form $x_i \leq \theta(b_j)$ or $x_i > \theta(b_j)$. A simple Decision Tree is depicted in Fig. 1. An example is classified by starting from the root node and traversing the tree, always taking the branches whose condition is satisfied by the values of the attributes. For a fully specified example, this process leads to a single leaf, corresponding to the predicted class.

5.2.1. Embedding Decision Trees in SMT

Satisfiability Modulo Theories (SMT) is an extension of SAT that allows one to include non-boolean predicates (e.g. linear inequalities over integer variables) in a logical formula. An SMT solver [39] combines a SAT solver and one or more theory solvers, each providing support for some constraints via dedicated approaches (e.g. the Simplex algorithm). The SAT solver manages the boolean representation, taking care of (e.g.) unit propagation and conflict learning. Several search strategies are possible, ranging from lazy approaches where all the boolean decisions are instantiated before the theory variables, to more complex schemes where the SAT and theory solvers work in an interleaved fashion. Optimization can be performed by posting a bounding constraint whenever a solution is found, and then restarting the search process.

SMT allows one to treat constraints over non-logical domains as boolean predicates, enabling a convenient encoding of DTs. First, we introduce variables to model the input attributes and the class (let them be x and y respectively). Symbolic

attributes and classes can be modeled as integer variables. Then, we can obtain a simple rule-based encoding based on the observation that each path π from root to leaf can be viewed as a logical implication:

$$\bigwedge_{b_j \in \pi} cst(b_j) \Rightarrow [\![y = C(\pi)]\!] \tag{19}$$

where the notation $[\![-]\!]$ refers to the boolean predicate corresponding to the constraint enclosed in the brackets. The term $C(\pi)$ is the class corresponding to the leaf in the path π, and each b_j is a branch along the path. Each expression $cst(b_j)$ is in the form:

$$cst(b_j) = \begin{cases} \bigvee_{v \in L(b_j)} [\![x(b_j) = v]\!] & \text{if } x(b_j) \text{ is symbolic} \\ [\![x(b_j) \leq \theta(b_j)]\!] & \text{if } x(b_j) \text{ is numeric and } b_j \text{ is a left-branch} \\ [\![x(b_j) > \theta(b_j)]\!] & \text{if } x(b_j) \text{ is numeric and } b_j \text{ is a right-branch} \end{cases}$$

where $x(b_j)$ refers in this case to the attribute variable associated to branch b_j.

Posting Clause (19) for each path in the tree is sufficient to encode a DT. However, it is possible to obtain a formulation leading to stronger propagation. The key observation is that the set of leaves labeled with a certain class specifies *all* and *only* the input configurations that should be mapped to such class. This allows one to encode a whole tree as a set of clauses:

$$\forall \text{ class } c_h: \quad [\![y = c_h]\!] \Leftrightarrow \bigvee_{\pi_k : C(\pi_k) = c_h} \left(\bigwedge_{b_j \in \pi_k} cst(b_j) \right) \tag{20}$$

In other words, the class variable y takes the value c_h iff at least one of the implications associated to the paths π_k labeled with c_h is true. This formulation allows the SMT solver to perform more powerful deductions via unit propagation, and it is therefore the one we employ in the remainder of the paper.

5.2.2. Embedding Decision Trees in CP

If the Constraint Programming solver used for the implementation supports logical expressions, then it is possible to directly employ in CP the encoding from Equation (20). Indeed, such an encoding was originally designed for CP and presented by some of the authors of this paper in [5].

If logical constraints are not supported, it is still possible to obtain an equivalent reformulation of Equation (20), by separately modeling the left-to-right and right-to-left implications associated with the $\Leftrightarrow$ operator. In particular, the right-to-left implication corresponds to Equation (19) and for the whole DT translates to:

$$\forall \text{ path } \pi_k: \quad \prod_{b_j \in \pi_k} cst(b_j) \leq [\![y = C(\pi_k)]\!] \tag{21}$$

where the notation $[\![-]\!]$ refers here to a reified constraint, which denotes 1 if the expression between the double square brackets is true and 0 if the expression is false. Informally, Equation (21) forces the class variable y to take the value $C(\pi_k)$ if the current domain of the attribute variables is such that all the $cst(b_j)$ constraints are necessarily satisfied. Conversely, if y takes the value $C(\pi_k)$, then at least one of the conjunctions of $cst(b_j)$ constraints must be true. This leads to:

$$\forall \text{ class } c_h: \quad [\![y = c_h]\!] \leq \sum_{\pi_k : C(\pi_k) = c_h} \left(\prod_{b_j \in \pi_k} cst(b_j) \right) \tag{22}$$

None of the presented encodings is capable of enforcing Generalized Arc Consistency (GAC). GAC can be actually achieved in CP using the two encodings that we have presented in [5] (respectively based on a TABLE and an MDD constraint). For sake of simplicity in this paper we restrict ourselves to the simple encodings that we have presented, and we refer the interested reader to [5].

6. Design of the optimization model

In this section we discuss Step 1 of the EML design process, i.e. the definition of the *core combinatorial structure* of the optimization problem. Informally, this is the part of the optimization model that can be designed by a domain expert in a traditional fashion. Defining the combinatorial structure requires one to *identify the input and output of the Empirical Model*, i.e. to define which part of the final model should be extracted from data. This decision affects the Machine Learning techniques that can be used to extract the Empirical Model.

For our example problems, we have already provided a first definition of the core combinatorial structure in the two base models from Section 3. In both cases, the input of the Empirical Model is given by the mapping variables. The EM output for the WDP_{bal} is given by the efficiency of each core: this implies that some kind of regression technique must be used to obtain the Empirical Model. The output for the WDP_{max} is a vector of binary variables ($eff_k = 1$ iff core k has "high" efficiency), hence in this case the Empirical Model is a classifier.

In the remainder of this section we will present models for the core combinatorial structure of the WDP_{bal} and the WDP_{max}, using a variety of optimization techniques (LS, MINLP, CP, and SMT). We refer to each model with the notation $\mathbf{T_p(h)}$, where **T** identifies the solution technique, and **p** the problem type (*bal* for the WDP_{bal} and *max* for the WDP_{max}). The **h** term identifies how the relation between the mapping and the efficiency variables is modeled, which is left unspecified in this section (the possible alternatives will be discussed in Sections 7 and then again in Section 8).

Core model for Local Search The Local Search model for the WDP_{bal} is identical to the base model presented in Section 3, and reported here for convenience:

LS_{bal}(h)

$$\max z = \min_{k=0..m-1} (eff_k) \tag{23}$$

$$\text{subject to: } \boxed{eff_k = h_k(x) \text{ [empirical model]}} \qquad \forall k = 0..m-1 \tag{24}$$

$$\sum_{k=0}^{m-1} x_{ik} = 1 \qquad \forall i = 0..n-1 \tag{25}$$

$$\sum_{i=0}^{n-1} x_{ik} = \frac{n}{m} \qquad \forall k = 0..m-1 \tag{26}$$

$$x_i \in \{0, 1\} \qquad \forall i = 0..n-1 \tag{27}$$

where we recall that n is the number of jobs and m is the number of cores. The h_k functions represent the Empirical Model, which will be discussed in detail in Section 7. Analogously, the LS model for the WDP_{max} is:

LS_{max}(h)

$$\max z = \sum_{k=0}^{m-1} (heff_k) \tag{28}$$

$$\text{subject to: } \boxed{heff_k = h_k(x) \text{ [empirical model]}} \qquad \forall k = 0..m-1 \tag{29}$$

Constraints (25), (26), (27)

Core model for MINLP A MINLP model for the WDP_{bal} can be obtained from the base model by linearizing the objective function:

$MINLP_{bal}$(h)

$$\max z \tag{30}$$

$$\text{subject to: } z \leq eff_k \qquad \forall k = 0..m-1 \tag{31}$$

$$\boxed{eff_k = h_k(x) \text{ [empirical model]}} \qquad \forall k = 0..m-1 \tag{32}$$

Constraints (25), (26), (27)

We did not design a MINLP model for our second example problem, because we have not defined a technique for encoding in MINLP a Decision Tree, i.e. the Empirical Model that we employ for the WDP_{max} (see Section 7).

Core model for CP We have designed CP models for the two dispatching problems presented in Section 3. Unlike the LS and MINLP case, in the CP models we represent the mapping decisions via integer variables $x_i \in \{0, .., m-1\}$, such that $x_i = k$ iff job i is mapped on core k. This encoding ensures that a job is always mapped to a single core. The full model for WDP_{bal} is as follows:

CP$_{bal}$(**h**)

$$\max z = \min_{k=0..m-1} (eff_k) \tag{33}$$

$$\text{subject to: } \boxed{eff_k = h_k(x) \text{ [empirical model]}} \quad \forall k = 0..m-1 \tag{34}$$

$$\text{GCC}\left(x, \{0, .., m-1\}, \frac{n}{m}\right) \tag{35}$$

$$x_i \in \{0, .., m-1\} \quad \forall i = 0..n-1 \tag{36}$$

$$eff_k \in (0, 1] \quad \forall k = 0, ..m-1 \tag{37}$$

Constraints (35) employ a GCC (Global Cardinality Constraint [40]) to ensure that the same number of jobs is mapped to each core. The GCC constraint forces each value of the set $\{0, ..m-1\}$ to be taken by exactly n/m of the x variables. The model for the WDP_{max} is identical, except for the cost function and the definition of the efficiency variables:

CP$_{max}$(**h**)

$$\max z = \sum_{k=0}^{m-1} (heff_k) \tag{38}$$

$$\text{subject to: } \boxed{heff_k = h_k(x) \text{ [empirical model]}} \quad \forall k = 0..m-1 \tag{39}$$

$$\text{Constraints (35), (36)} \tag{40}$$

$$heff_k \in \{0, 1\} \quad \forall k = 0, ..m-1 \tag{41}$$

Core model for SMT We did not design an SMT model for the WDP_{bal}, because the Empirical Model that we employ in this case is a Neural Network and we have not devised a technique for embedding ANNs in SMT. We did however define an SMT model for the WDP_{max}. In the model we represent the mapping decisions via integer variables $x_{i,k}$ such that $x_{i,k} = 1$ iff job i runs on core k, similarly to what was done for LS and MINLP. The resulting SMT model is:

SMT$_{max}$(**h**)

$$\max z = \sum_{k=0} m - 1 \, (heff_k) \tag{42}$$

$$\text{subject to: } \boxed{heff_k = h_k(x) \text{ [empirical model]}} \quad \forall k = 0..m-1 \tag{43}$$

$$\sum_{k=0}^{m-1} x_{ik} = 1 \quad \forall i = 0..n-1 \tag{44}$$

$$\sum_{i=0}^{n-1} x_{ik} = \frac{n}{m} \quad \forall k = 0..m-1 \tag{45}$$

$$[\![x_{ik} = 0]\!] \vee [\![x_{ik} = 1]\!] \quad \forall i = 0..n-1, \; k = 0..m-1 \tag{46}$$

where $[\![x_{i,k} = 0]\!]$ and $[\![x_{i,k} = 1]\!]$ are boolean predicates associated to non-boolean constraints (in this case, linear equations over integer variables). The boolean clauses (46) model the binary domain of the $x_{i,k}$ variables.

7. Extracting an Empirical Model

In this section we discuss Step 2 of the EML design process, i.e. obtaining the Empirical Model. In principle, extracting the Empirical Model is simply a supervised learning task: given a *training set* containing known input/output pairs (i.e. *examples*) for the considered system, the goal is to obtain a function to predict the system behavior on unseen examples. This goal can be achieved via a number of powerful techniques from the Machine Learning domain. In this work, we will limit ourselves to Artificial Neural Networks (ANNs) and Decision Trees (DTs), i.e. the Machine Learning models for which an embedding technique has been provided in Section 5.

In general, extracting a Machine Learning model requires one to: (1) obtain a training set, (2) choose the features to be used as input, (3) choose a Machine Learning technique, and then (4) proceed with the training and the evaluation.

Extracting an Empirical Model is done in the same way, except that each step requires some special care because of the peculiarities of the EML context. In the remainder of this section we identify such peculiarities and how to deal with them, using the two dispatching problems from Section 3 as running examples.

7.1. EM extraction, step 1: obtaining a training set

In EML, a training set can be obtained either by collecting historical data or by running ad-hoc experiments. This can be a costly operation, but it needs to be performed only once,[4] for obtaining the Empirical Model.

A good training set should be representative of the instances (input configurations) for which a prediction needs to be made. In a classical Machine Learning scenario, the set should include instances that are likely to occur in practice, so as to introduce a controlled bias that simplifies the learning problem. In EML, however, *the instances to be evaluated are generated by an optimization system*: in our example problems, given a specific set of jobs, the search engine will try to explore the space of possible mappings as extensively as possible. In this kind of situation, defining which input configurations are more likely to be generated can be extremely difficult.

We address this issue by designing a training set that is as unbiased as possible *with respect to the decisions made by the search engine*. For example, we can (1) collect groups of jobs that are likely to be submitted to the system, and then (2) build a training set by generating mappings at random so as to "cover" the set of possible mappings as uniformly as possible. This approach yields a training set that is biased towards likely sets of jobs, but unbiased w.r.t. how such jobs can be mapped.

The training set for the example problems For our dispatching problems, we learn a different Machine Learning model for each core. Using different models is necessary since the cores have non-homogeneous behavior, due to unique factors such as their position on the chip, or variability in the manufacturing process. Using a model for each core rather than a single model for the whole platform allowed us to obtain high accuracy using a limited number of inputs, leading to a dramatic reduction of the training set size.

We built the training set for each core by generating random groups of mapped jobs. Then, the corresponding efficiency values were obtained via the simulator. The efficiency values were employed directly in the training set for the WDP_{bal}, while for the WDP_{max} the efficiency was discretized into a 0-1 class using the value 0.97 (i.e. 97%) as a threshold. The threshold was chosen together with domain experts. The efficiency measure that we consider is the average efficiency of the mapped jobs, computed as:

$$eff_k = \frac{1}{n/m} \sum_{\text{job } i \text{ on core } k} \frac{cpi_i}{\widehat{cpi}_i} \tag{47}$$

where n/m is the number of jobs mapped on the core, cpi_i is the nominal CPI of job i (which assumes maximum operating frequency), and $\widehat{cpi}_i$ is the job CPI as measured during simulation (which includes the slow-down due to the thermal controller). The formula provides an indication of the main factors that affect the core efficiency. In particular, we observe that:

1. Because of the thermal controller, the efficiency of a core depends on its temperature, which is affected by how much the running jobs stress the CPU. In particular, there is a known correlation between the temperature of a core and the average CPI of the running jobs (see Section 8).
2. The temperature of a core is also subject to thermal interactions from the other cores, in particular the nearest ones.
3. The temperature of a core depends on its position on the silicon die, which affects its ability to disperse heat: this dependency is taken into account by learning a different model for each core.
4. The temperature of a core is affected by that of the computer room. For sake of simplicity, we disregard this factor in the paper (it could be taken into account – for example – by learning different models for different room temperature ranges).
5. Finally, not all jobs are affected by the thermal controller in the same way. Jobs with higher cpi_i values spend a lot of clock cycles waiting for memory operations to conclude: as a consequence, their actual CPI value is less affected by the thermal controller (which acts on the frequency of the CPU and not of the memory). Conversely, jobs with low CPI are more affected by frequency reductions.

Based on these observations, we obtained training sets via a factorial design (similarly to what is done for surrogate models [21]). In particular, for each core k we generated multiple random sets of 288 jobs (6 per core) so as to cover the possible values of some key parameters, reported in Table 2. Each parameter in the table is an approximation for one of the main factors affecting the core efficiency. The p_0, p_1, p_2 parameters are related to the *average* CPI of the jobs on each core, which is used as a proxy for the temperature. The *actual* CPI values on core k are randomly generated following a beta distribution, with 4 different parameterizations (see p_3 in the table). The CPI values on the remaining cores are also beta distributed, but a single parameterization is used in this case (chosen uniformly at random from those considered for core k).

[4] Unless active learning is employed, but we leave this as a subject for future research.

Table 2
Factorial design of the training set for core k.

Parameter	Corresponding factor	#Values	
p_0	Average CPI of the jobs mapped on core k	Power consumption of core k	3
p_1	Average CPI of the jobs mapped on neighboring cores	Power consumption of nearby cores	up to 21
p_2	Average CPI of the jobs mapped on the remaining cores	Power consumption of far-off cores	3
p_3	α and β values for a beta distribution, for generating CPIs values	Effect of the individual CPI values of the jobs on core k	4 α, β pairs

Overall, for each core we obtained (up to) $3 \times 21 \times 3 \times 4 = 756$ sets of mapped jobs, which were simulated in order to obtain the corresponding efficiency values. The process was quite time consuming, given that each simulation takes two-to-three minutes of computation time. However, the training set needs to be generated only once, during the system setup: the performance of the EML based solution approach is not affected by the training time, but rather by the complexity of the extracted model. All our training sets are publicly available.[5]

7.2. EM extraction, step 2: choosing the input features

In EML, the input of the Empirical Model consists in principle of the decision variables themselves. In practice, feeding the decisions (e.g. the job-core mapping) directly to a Machine Learning model would make it dependent on the problem size and reduce the generality of the method. In Machine Learning, this issue is usually addressed by using aggregation functions (e.g. average, standard deviation...) to obtain features that are then fed to the model.

In EML, this means that the Empirical Model constraints in the formulation from Section 4 are in fact a combination of two relations:

$$z = h(x) \text{ is equivalent to } \begin{cases} y = h_{feat}(x) \\ z = h_{EM}(y) \end{cases} \tag{48}$$

where y is a vector of variables representing the features, h_{EM} is the encoding of the Machine Learning model, and h_{feat} are *feature extraction constraints*. The need to encode in the final model also the feature extraction constraints may lead to accuracy/effectiveness trade-offs: for example, an important feature may be difficult to encode in the optimization technique that is best suited for the combinatorial part of problem.

Input features for the example problems We designed and tested several input features, based on the factors affecting the efficiency that have been identified in Section 7.1. After several attempts, we settled for the following list:

1. The average CPI of the jobs on core k: $avgcpi_k = \frac{1}{m} \sum_{\text{job } j \text{ on } k} cpi_j$
2. The minimum CPI of the jobs on core k: $mincpi_k = \min_{\text{job } j \text{ on } k} (cpi_j)$
3. The average of the average CPI of the neighboring cores: $neighcpi_k = \frac{1}{|N(k)|} \sum_{\text{core } h \in N(k)} avgcpi_h$
4. The average of the average CPI of all the other cores: $othercpi_k = \frac{1}{m-1-|N(k)|} \sum_{\text{core } h \neq k, h \notin N(k)} avgcpi_h$

where $N(k)$ is the set of cores having H_∞ distance (i.e. the maximum between the distances along the x and y coordinates on the chip grid) equal to one from the target k. Most of the input features correspond directly to the parameters from Table 2, which is quite natural. The $mincpi_k$ feature was introduced (among other, eventually discarded, ones) to provide the Machine Learning model with information about the individual CPI values.

The feature extraction constraints for the Local Search models are:

$$avgcpi_k = \frac{1}{m} \sum_{i=0}^{n-1} cpi_i \cdot x_{i,k} \qquad \forall k = 0..m-1 \tag{49}$$

$$mincpi_k = \min_{i=0..n-1} (max_{cpi} - (max_{cpi} - cpi_i) \cdot x_{i,k}) \qquad \forall k = 0..m-1 \tag{50}$$

$$neighcpi_k = \frac{1}{|N(k)|} \sum_{h \in N(k)} avgcpi_h \qquad \forall k = 0..m-1 \tag{51}$$

$$othercpi_k = \frac{1}{m-1-|N(k)|} \sum_{h \neq k, h \notin N(k)} avgcpi_h \qquad \forall k = 0..m-1 \tag{52}$$

[5] In the git repository https://bitbucket.org/m_lombardi/eml-aij-2015-resources.

where $avgcpi_k$, $mincpi_k$, $neighcpi_k$, $othercpi_k$ are real-valued variables, and max_{cpi} is the maximum possible CPI (a constant). The feature extraction constraints for the CP models are identical to those used for Local Search, except that in place of the $x_{i,k}$ variables we use reified constraints in the form $[\![x_i = k]\!]$.

Constraints (49), (51), and (52) can be directly inserted in a MINLP model. This is not the case for Constraints (50), because of the presence of the "min" operator. It may be possible to encode the minimum operator in a MINLP via integer variables and linearization, with adverse effects on the bound quality. For sake of simplicity, however, we have decided to extract two groups of Empirical Models, respectively with and without the $mincpi_k$ input feature. Only the second group has been embedded in MINLP.

In the SMT model from Section 6, linear predicates that define $avgcpi_k$, $neighcpi_k$, $othercpi_k$ can be obtained from Constraints (49), (51), and (52) by replacing the $x_{i,k}$ variables with the LIA (Linear Integer Arithmetic) predicates $[\![x_{i,k} = 1]\!]$. For defining the feature extraction constraints of $mincpi_k$, we first build a vector s containing all job indices, sorted by increasing CPI value. Then, we post the hybrid linear/logical predicate:

$$mincpi_k = ite([\![x_{s_0} = 1]\!], cpi_{s_0}, ite([\![x_{s_1} = 1]\!], cpi_{s_1}, \ldots)) \tag{53}$$

where *ite* is an if-then-else statement: the first parameter is the condition to test, the second and third parameter are the expressions to be denoted if the condition is respectively true or false.

Normalization For many Machine Learning approaches (e.g. the Artificial Neural Networks that we will employ in the next section), it is beneficial to normalize the input features in a fixed range (typically $[-1, 1]$). In our case, this was done using the formula:

$$avgcpi'_k = \frac{1}{span_1}(avgcpi_k - span_0) \qquad \forall k = 0..m-1 \tag{54}$$

for $avgcpi_k$ and analogously for all other features. In the formula, $span_0 = (max_{cpi} + min_{cpi})/2$ and $span_1 = (max_{cpi} - min_{cpi})/2$. The value min_{cpi} represents the minimum possible CPI value, which is assumed to be 0 in a slightly conservative fashion (CPI values are always strictly positive). The normalization Formula (54) was included in the feature extraction constraints for all our approaches using ANNs for the Empirical Model.

7.3. EM extraction, step 3: training and quality assessment

Designing and training a model are very well studied topics in Machine Learning (see [41] for an excellent overview). Both steps require one to assess the quality of the extracted model, which is measured via metrics such as the Mean Squared Error or the number of correctly classified instances. The assessment is typically done on a separate *test set* or via cross-validation.

In EML, the fact that the input of the Machine Learning model is generated by a search approach makes the quality assessment more complicated. Suppose for example that the extracted model has a large prediction error for a certain range of values of the input features that is not very frequent in the training set. If the same values happen to be favorable in terms of cost *in the problem model*, then such input configurations, despite being uncommon in the training set, will be actively sought by the optimizer and will be much more likely to appear at search time. Because of this, the actual error in the final system has a chance to be similar to the *maximum* error on the test set.

This issue could be addressed by using (for example) the maximum error as an evaluation metric during training. Alternatively, one could dynamically generate new examples that are similar to the ones leading to the largest errors, and then repeat the training. For sake of simplicity, in this paper we limit ourselves to classical training techniques: this means for example that we use the Mean Squared Error to evaluate the quality of ANNs at training time. Since the Mean Squared Error assigns a higher penalty to errors with large absolute value, it also tends to mitigate the issue that we have described.

In traditional Machine Learning applications, evaluating the model (i.e. making a prediction) is usually a simple operation. In EML the extracted model is not simply evaluated, but rather it is exploited in a number of ways in order to boost the search process. This makes the size and complexity of the extracted model particularly important, since they determine the computation effort required for (e.g.) bounding or constraint propagation. This can lead to a trade-off between model size and accuracy, which may be critical if the final optimization system operates under tight time constraints. For our example problems, it was possible to obtain accurate and yet quite small EMs, and therefore we leave a thorough evaluation of this trade-off for future research.

Empirical models for the WDP_{bal} We used Artificial Neural Networks (ANNs) for the Empirical Model of the WDP_{bal}, for several reasons. First, we needed a regression technique because the EM output had to be an efficiency value. Moreover, ANNs are a classical technique in Machine Learning, they require little domain knowledge, and their modular nature makes them easier to embed in Combinatorial Optimization (see Section 5).

We trained for each core a feed-forward network with one hidden layer. We used bipolar sigmoid neurons (with tanh activation function) in both the hidden and the output layers: the use of alternative activation functions (e.g. rectifiers) may be considered as part of future research. The output of each network is therefore in the $(-1, 1)$ range, and must be scaled in order to obtain an easily understandable efficiency value. We used the Encog framework [42] for training the ANNs, via the

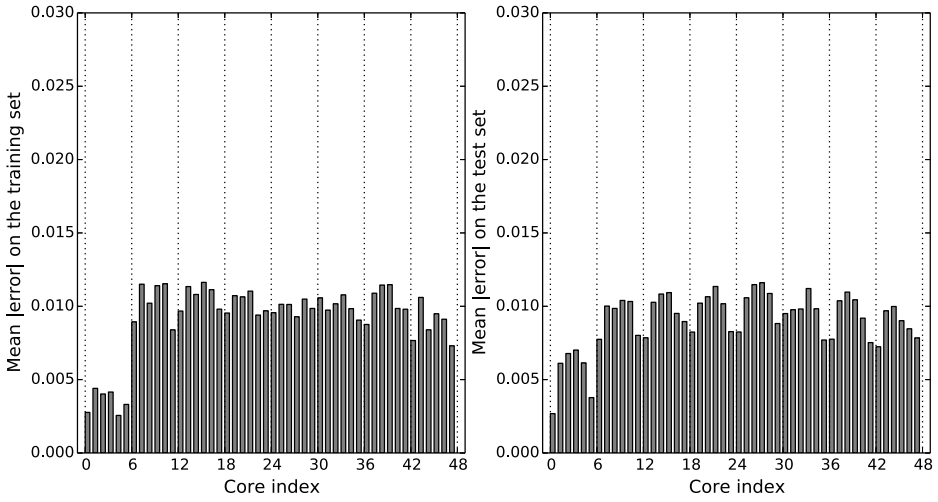

Fig. 2. Mean prediction error for the main ANN0.

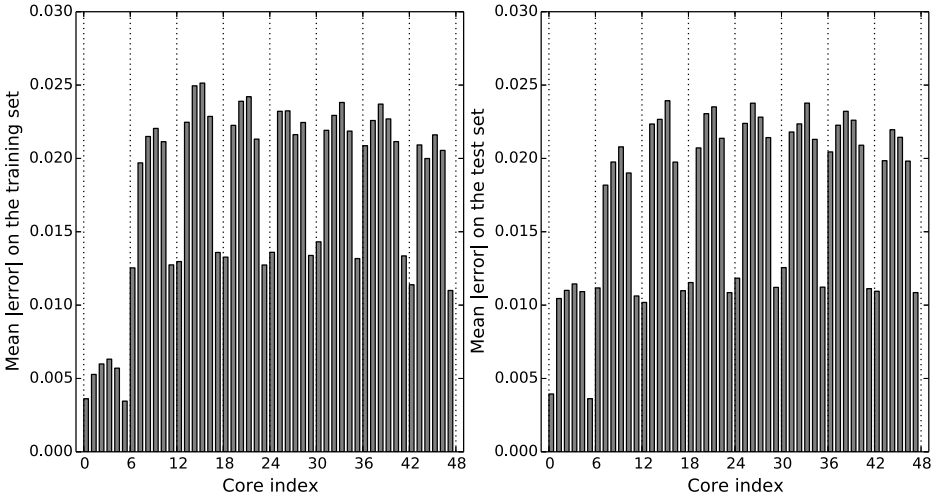

Fig. 3. Mean prediction error for the ANN1 lacking the "min" input.

Levenberg–Marquardt algorithm. After some experiments, we settled for using two neurons in the hidden layer (increasing the size did not significantly improve the accuracy).

We trained two ANN variants for each core, both having the same number and type of neurons. The first variant, referred to as ANN0, takes as inputs all the features from Equations (49)–(52). The second variant is referred to as ANN1 and lacks the $mincpi_k$ input, which is the most difficult to encode in Combinatorial Optimization.

We evaluated the ANNs over their training sets and over separate test sets. The test set for each core was obtained by sampling 50 examples at random from the training sets of the *other* cores. Therefore, each test set contains $50 \times 47 = 2350$ unseen examples, it is considerably different from the corresponding training set and considerably challenging.

Fig. 2 reports the mean absolute values of the prediction errors for ANN0 on each core. The cores are indexed by rows over an 8×6 grid, hence cores 0–5 refer to the first row, 6–11 to the second and so on. The errors refer to efficiency values in the range $(0, 1]$, hence a mean error of 0.01 corresponds to a 1% efficiency difference. The mean prediction errors are very low for both the training and the test set. Moreover, the efficiency of the cores on the first row and of those on the left and right chip borders is considerably easier to predict.

Fig. 3 reports the mean absolute values of the prediction errors for the ANN1 networks, i.e. those without the $mincpi_k$ input. Their average accuracy is still quite good, but definitely worse than ANN0 ($\sim 2.5\%$ against $\sim 1\%$). On the other hand, the ANN1 networks are easier to embed and may be employed with more effective optimization techniques. *This kind of trade-off is not unique to EML.* Rather, it is common when dealing with real world problems that involve approximate models. In this context, EML provides a flexible approach to balance model accuracy and solver efficacy.

Fig. 4 reports the error histogram over the training set for the ANN0 and ANN1 networks corresponding to core #20 (located close to the center of the chip). Actual errors are considered here rather than their absolute values. In most cases,

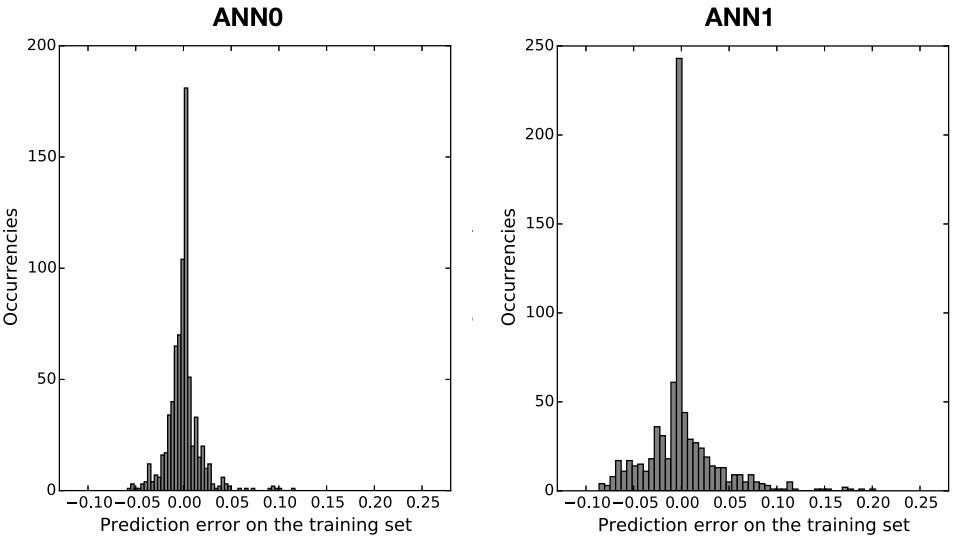

Fig. 4. Prediction error histograms for the ANNs corresponding to core 20.

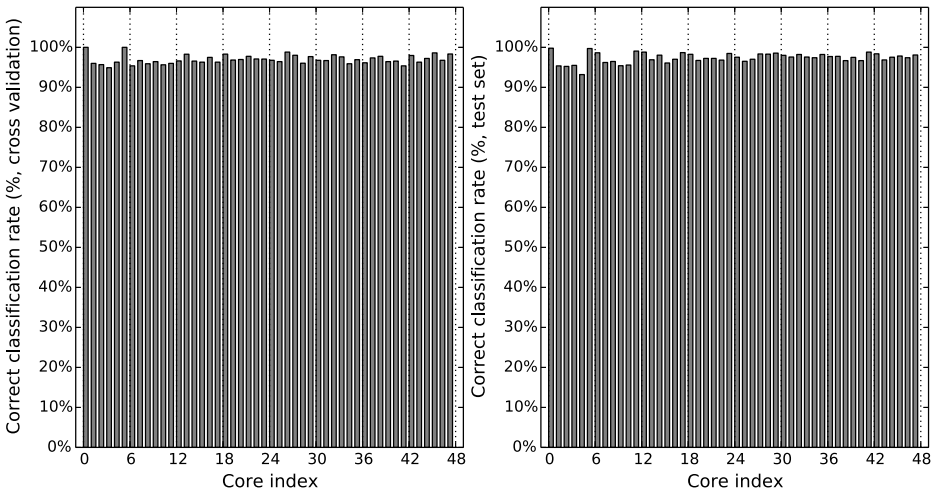

Fig. 5. Percentage of correctly classified instances for the Decision Trees.

both networks provide very accurate predictions, but larger error values (e.g. in the order of 10%) are more likely to occur for ANN1. This is potentially an issue, given the importance of maximum errors in EML (see the beginning of this Section 7.3).

Finally, we have investigated the impact that changing the size of the training set has on the accuracy of the ANN1 networks. In particular, we have tried to: (1) decrease the size of each training set by removing a (varying) percentage of instances, chosen uniformly at random; and (2) increase the size of each training set, by adding a (varying) number of instances chosen uniformly at random from the other training sets. The networks trained on these modified sets have been tested on the original test sets. We have found that both removing and adding as little as 10% instances has a significant *adverse* effect on the accuracy, leading to much larger average errors for some cores. The rationale is that both removing and adding instances causes an alteration of the factorial designed employed in the original training sets, making some region of the input space either over- or under-represented. The accuracy plots for this experimentation are not reported in this paper, but they are available on-line.[6]

Empirical models for the WDP_{max} For the second dispatching problem, we can use a classifier as an Empirical Model, trained to distinguish low- and high-efficiency cores. Specifically, our Empirical Model consists of a Decision Tree (DT) for each core. We chose DTs for two main reasons: first, DTs were accurate enough for the problem at hand. Second, our choice gives the

[6] In the git repository https://bitbucket.org/m_lombardi/eml-aij-2015-resources.

opportunity to discuss the EML approach on a Machine Learning technique that is radically different from Artificial Neural Networks.

We trained our Decision Trees using the C4.5 algorithm [43], implemented in Weka [44] with the name J48, with the default parameters (in particular, all trees are pruned). The performance of the DTs was assessed both via 10-fold cross-validation over the training set, and via evaluation over the test sets used for the ANNs. The outcome of the two evaluations is reported in Fig. 5, which shows the percentage of correctly classified instances for the DT corresponding to each core. The accuracy is actually very high in both cases.

8. Experimental results

In this section, we present an experimental evaluation of several solution methods for our example problems. We consider all the alternative modeling approaches discussed in Section 3, namely: (1) using a heuristic as a proxy for the efficiency values; (2) using fitting to adjust the parameters of an expert-designed model; and finally (3) several solution methods based on EML.

The benchmark for the experimentation is a set of 20 instances, each with 288 jobs to be mapped on 48 cores (i.e. the full SCC platform). The instances were designed to be as diverse as possible. As a common trait, each instance includes a majority of low-CPI jobs, plus a small number of jobs with a relatively high CPI: this was done to make the benchmark challenging, leaving at the same time sufficient room for improvements. The instances and the detailed results are publicly available for download.[7]

All the solution approaches that we present are based on approximate models, since relying on the simulator during search would lead to unacceptable response times. As a consequence, it is necessary to distinguish two levels in our experimental comparisons:

1. First, we compare the effectiveness of different solution approaches in terms of the value of the cost function, i.e. the *predicted solution quality*.
2. Second, we compare solution approaches in terms of the *"real" solution quality*, which in our case is obtained from the simulator.

The first level of comparison is the one adopted by many combinatorial optimization papers: it allows one to assess the effectiveness of a given approach in finding solutions for a given model. This comparison is meaningful *only for solution approaches based on the same approximate system model*. The second type of comparison evaluates the combined effectiveness of the solution method and the approximate model. We will use the first type of comparison in Section 8.3, in order to determine the best solution method for each modeling approach (heuristic, model fitting, EML); the second type of evaluation is used in Section 8.4 to compare optimization approaches based on different models.

8.1. Approximate system models

In this section, we briefly present/review the approximate system models considered in our experimentation.

A heuristic as a proxy for the efficiency The computation workload of each job is measured by its CPI value, and jobs are executed on each core in an interleaved fashion by the thermal aware scheduler from Section 3. As a consequence, it seems reasonable to expect on each core k a correlation between the efficiency and the *average job CPI*.

The existence of such a correlation has been checked empirically by performing some pilot experiments. Fig. 6 shows the results for core #41, and contains a scatter plot with the efficiency on the y-axis and the average CPI of the jobs on the x-axis. Each point corresponds to a different set of jobs, and the color reflects the density of the data points (red denotes higher values, and hence the most typical region of operation). There is indeed a strong correlation between the efficiency and the average CPI, which is almost linear for low average CPI values (i.e. the most interesting region for the considered problems). The results for the other cores are not always as clean as those obtained for core #41, but a good degree of correlation is always present.

Based on these observations, our domain experts (i.e. the authors of the platform simulator) have suggested to employ the average job CPI on each core as a proxy for its efficiency in the WDP_{bal}. The resulting system model is:

$$eff = h_{ACPI}(x) = w^T avgcpi \tag{55}$$

where $avgcpi$ is the vector of the $avgcpi_k$ values, as specified in the feature definitions (i.e. Equation (49)), and w is a vector of weights that identifies how sensitive each core is to changes of the average CPI. Assuming we want to avoid the use of model fitting (which will be considered in the next paragraph), the value of w must be fixed based on the knowledge of the domain expert. In our experiments, we have considered two possibilities:

[7] In the git repository https://bitbucket.org/m_lombardi/eml-aij-2015-resources.

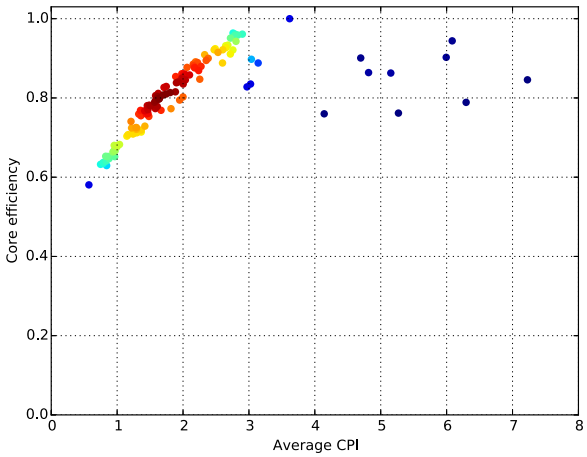

Fig. 6. Core efficiency over the average CPI of the mapped jobs. (For interpretation of the references to color in this figure, the reader is referred to the web version of this article.)

Table 3
Solution approaches considered in the experimentation.

	Model	Opt. technique				
		LS	MILP	MINLP	CP	SMT
WDP_{bal}	ACPI	×	×		×	
	wACPI	×	×		×	
	FEAT	×	×		×	
	ANN0	×			×	
	ANN1	×		×	×	
WDP_{max}	DT	×			×	×

1. $w = 1$ for all cores;
2. $w = 1$ for internal cores, $w = 0.75$ for cores on the chip borders, $w = 0.5$ for cores on the chip corners.

the first choice (referred to as *ACPI* model) is appropriate if reliable information is missing. The second choice (referred to as *wACPI* model) takes into account that heat dissipation is more efficient for the cores along the border of the silicon die. We have employed the heuristic approach only for the WPD_{bal} problem.

Using fitting to adjust an expert-designed model A compromise between using EML and relying on a heuristic consists in employing function fitting to adjust the parameters of an expert-designed model. With the aim to evaluate this approach, we have devised a linear model for the WDP_{bal} based on the features from Section 7. The idea is that pilot experiments and the observations from Section 7.1 should be sufficient to define this model structure, without relying on more powerful learning techniques. The resulting model is referred to as FEAT and is given by:

$$eff = h_{FEAT}(x) = wa^T avgcpi + wm^T mincpi + wn^T ngbcpi + wo^T otrcpi$$

where wa, wm, wn, wo are vectors of weights that are obtained via linear regression. Linear regression makes it feasible to obtain values for all the parameters; however, it also requires one to build a training set, and therefore an overall design effort much closer to that of extracting an Artificial Neural Network. Still, in the FEAT model the cost function is linear, which may allow the optimization approach to get considerably closer to the optimum (predicted) cost. We have employed the model fitting approach only for the WDP_{bal} problem.

Using an Empirical Model Finally, we have modeled the system behavior via Empirical Model Learning. The details of our Empirical Models have already been given in Section 7. Here we simply recall that we employed the ANN0 and ANN1 Neural Networks for the WDP_{bal}, while we employed Decision Trees (referred to as DT) for the WDP_{max}. We recall also that the ANN1 model differs from ANN0 because it lacks the $mincpi_k$ input feature.

8.2. The solution approaches

A summary of the solution approaches considered in our experimentation is reported in Table 3. The table highlights the solution techniques, the problems they were applied to (i.e. WDP_{bal} and WDP_{max}), and the models employed for the system

Algorithm 1 Find Initial Solution(n, m, cpi).

$sum_k = 0, cnt_k = 0 \quad \forall k = 0..m-1$
repeat
  select the job i^* with lowest cpi_i
  map job i^* on the core k^* with min cnt_k, break ties by max sum_k
  $sum_{k^*} = sum_{k^*} + cpi_{i^*}$
  $cnt_{k^*} = cnt_{k^*} + 1$
until all jobs are mapped

behavior. Only two of the solution techniques (LS and CP) have been applied in all configurations: all the remaining ones were restricted to linear formulations (MILP) or to specific system models (MINLP and SMT).

We used Localsolver v3.0 [45] to implement our LS approach. We defined our MINLP model in GAMS [46] and we solved it using BARON, via the Neos Servers for Optimization [47]. As a CP system, we chose Google or-tools,[8] and we employed the Neuron Constraint implementation that we developed in [4]. In the CP model, we use integer variables with a fixed precision factor to represent real-valued variables. Finally, we chose Z3 by Microsoft Research for our SMT models [39] and IBM ILOG CPLEX[9] v12.5 for the MILP models. All solvers were chosen based on their effectiveness (tested in some pilot experiments) and on the accessibility of their modeling interfaces.

We use the default configuration of Localsolver, CPLEX, and BARON to solve our LS, MILP, and MINLP models. The CP and SMT models are instead solved via a Randomized Adaptive Decomposition approach, similar in spirit to Large Neighborhood Search and described in the next paragraph. This was done in order to boost their ability to find high quality solutions in a reasonably small time (a few tens of seconds in our case). The decomposition technique was not applied in LS (because Localsolver is already designed to be scalable), nor in MINLP (because we had access to BARON via a slow, http-based, API), nor in MILP (this could be a topic for future research).

Randomized adaptive decomposition Our CP and SMT approaches are wrapped in a decomposition method similar to Large Neighborhood Search to reduce the time for obtaining high-quality solutions. The method starts from an initial solution, which is obtained via the simple greedy heuristic presented in Algorithm 1. The heuristic works by iteratively mapping the most computation-intensive job (i.e. the one with lowest CPI) on the least loaded cores. A core k_0 is considered to be less loaded than a core k_1 if: (1) there are fewer jobs on k_0 than on k_1; (2) the number of jobs is identical, but the sum of job CPIs on k_0 is lower than the sum for k_1. This approach tries to balance the average CPI value on each core, essentially following the same idea at the basis of the ACPI model.

Then, the decomposition method proceeds analogously to Large Neighborhood Search, by iteratively: (1) selecting a small set C_R of cores; (2) relaxing the mapping decisions for all jobs running on such cores; (3) solving a restricted problem to re-map the jobs within the cores in C_R, with the aim to improve the solution quality. The difference with LNS is that, when solving the restricted subproblem, we change the objective function to:

$$\max z = \min_{k \in C_R} eff_k \quad (WDP_{bal}) \qquad \max z = \sum_{k \in C_R} eff_k \quad (WDP_{max})$$

i.e. in the cost function we neglect the efficiency of all cores, except those in the set C_R. In other words, when solving the restricted problem at each iteration we look for local improvements rather than for global ones. This is particularly useful for the WDP_{bal}, for which it may happen that during search multiple cores have a similarly low efficiency. In this situation, the decomposition approach can focus on each of those critical cores individually.

We stop each iteration as soon as one improving solution is found, or when a time limit is reached. In the SMT approach, the restricted subproblem is solved via Z3 with default parameters. In the CP case we use restarts and depth first search, with random variable and value selection.

For choosing the cores to relax, we always select a number n_{bad} of "bad" cores, of which we want to improve the efficiency, plus a number n_{good} of "good" cores, for which we can accept an efficiency reduction. In practice we employ a slightly different criterion on the two example problems.

For the WDP_{bal}: the "bad" cores are chosen randomly (with uniform probability) among those with minimum efficiency; the "good" cores are chosen randomly among the rest, with a selection probability given by the (normalized) ratio ${}^{eff_k}/_{avgcpi'_k}$, where $avgcpi'_k$ is equal to ${}^{avgcpi_k}/_{max_{cpi}}$.

For the WDP_{max}: the "bad" cores are chosen randomly (with uniform probability) among those having low efficiency; the "good" cores are chosen at random among the rest, with a selection probability given by the (normalized) value of $avgcpi_k$.

In our experiments, we have $n_{bad} = 1$ and $n_{good} = 3$ for the WDP_{bal}, and we have $n_{bad} = 2$ and $n_{good} = 3$ for the WDP_{max}.

[8] The software and the existing documentation are available at: https://developers.google.com/optimization/.
[9] See the software page at http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/.

Table 4
Results for the WDP_{bal} with the ACPI, ANN0, and ANN1 model.

#	ACPI			ANN0		ANN1		
	LS_{bal}	$MILP_{bal}$		CP_{bal}	LS_{bal}	CP_{bal}	LS_{bal}	$MINLP_{bal}$
	obj	obj	UB	obj (%)	obj (%)	obj (%)	obj (%)	UB
0	2.90	2.88	2.93	74.6 (0.4)	73.6 (1.1)	70.0 (0.2)	67.1 (1.0)	*72.1*
1	2.93	2.92	3.58	78.8 (0.6)	75.5 (2.1)	77.4 (0.4)	68.1 (2.5)	84.0
2	2.35	2.33	2.39	71.9 (0.3)	71.5 (1.3)	66.9 (0.7)	65.8 (0.5)	85.4
3	2.18	2.17	2.18	70.5 (0.4)	68.0 (4.6)	66.2 (0.1)	65.4 (0.6)	86.9
4	2.00	2.00	2.00	68.8 (0.4)	68.2 (0.8)	64.5 (0.1)	64.6 (0.6)	*66.2*
5	2.83	2.82	2.88	73.1 (0.5)	73.5 (1.0)	69.2 (0.3)	66.9 (0.6)	*72.3*
6	4.19	4.19	5.05	84.7 (0.3)	77.8 (5.5)	85.8 (0.4)	76.8 (6.5)	95.2
7	2.72	2.70	2.76	72.9 (0.4)	73.3 (0.8)	68.2 (0.7)	66.8 (0.8)	*71.0*
8	2.65	2.63	3.01	76.7 (0.3)	73.4 (3.2)	73.0 (0.5)	66.5 (1.8)	83.3
9	3.62	3.57	4.34	82.9 (0.5)	76.4 (4.0)	83.6 (0.6)	74.6 (4.8)	95.4
10	3.08	3.05	3.80	80.5 (0.5)	75.3 (1.9)	79.7 (0.5)	69.7 (3.1)	91.9
11	2.00	2.00	2.00	69.0 (0.1)	68.2 (1.1)	64.5 (0.1)	65.0 (0.4)	*66.3*
12	2.68	2.67	2.71	77.1 (0.4)	72.7 (1.5)	72.7 (0.6)	67.0 (1.0)	85.5
13	2.15	2.13	2.16	70.1 (0.3)	69.9 (0.8)	65.7 (0.3)	65.2 (0.4)	*66.7*
14	2.90	2.89	2.96	74.8 (0.5)	74.5 (0.7)	70.2 (0.1)	66.2 (0.9)	*72.3*
15	2.76	2.75	2.80	76.5 (0.6)	74.2 (3.1)	73.5 (0.6)	66.2 (1.4)	87.8
16	2.17	2.16	2.22	70.4 (0.2)	69.9 (1.8)	66.1 (0.1)	65.6 (0.4)	83.8
17	4.22	4.22	5.10	84.8 (0.2)	78.9 (3.5)	85.7 (0.4)	77.9 (6.5)	*87.2*
18	2.60	2.58	2.62	72.3 (0.3)	72.2 (0.8)	67.3 (0.8)	65.8 (0.8)	*69.9*
19	2.76	2.75	3.12	77.1 (0.6)	74.4 (1.8)	74.9 (0.7)	67.5 (0.9)	*79.1*

8.3. Evaluation based on the predicted quality

In this section, we compare the solution approaches from Table 3 in terms of the predicted solution quality: this allow us to assess the effectiveness of each approach at finding good solutions for specific system models. The wACPI and FEAT models were solved via Local Search, since such an approach performed slightly better than MILP on the (very similar) ACPI model.

Results for the WDP_{bal} We obtained a LS approach based on the ACPI model for the WDP_{bal} by inserting the $h_{ACPI}(x)$ function into the core Local Search model from Section 6. An analogous process was followed to obtain a MILP approach. We solved the problem using a time limit of 90 seconds for Localsolver (after that time no significant improvement was obtained) and of 1 hour for CPLEX (in an effort to prove optimality).

The results of this experimentation are reported in Table 4, in the **ACPI** columns. For both the approaches we report the minimum average CPI for each instance (i.e. the value of the objective function). For MILP we also report the value of the best upper bound at the end of the search process: as can be seen, it was not possible to prove optimality within the time limit in most cases. However, the bound values provide an indication that the solutions found by both approaches are indeed very good, with Localsolver having a slight edge.

Table 4 presents also the results of the EML approaches, based on the ANN0 and ANN1 model, respectively in the **ANN0** and **ANN1** columns. The problem objective (lowest core efficiency) is reported as a percentage value. It is important to realize that only approaches operating on the same system model can be directly compared in terms of their predicted quality. Comparisons between approaches based on different system models will be taken into account in Section 8.4, and are meaningless at this stage: this is emphasized in the table by using double-lines to separate the **ACPI**, **ANN0**, and **ANN1** columns.

We recall that a MINLP model has not been designed for the "full" Artificial Neural Network (i.e. ANN0), but only for the simplified version without the $mincpi_k$ input (i.e. ANN1). Moreover, since we rely on a remote server for solving MILP problems, it was impossible for us to use the MINLP model within our Randomized Adaptive Decomposition scheme, which severely affected the performance of the approach.

Eventually, we decided to make a different use of the MINLP model, namely we employed it for computing a bound on the solution quality. This was done by removing the integrality restriction from the mapping variables and by solving the resulting Non-Linear Programming model. The best bound found after 2 hours is shown in Table 4 in the **MINLP**$_{bal}$/**UB** column, where values in italic font denote optimal (non-integer) solutions. We believe this – somehow disappointing – result provides actually a good example of how the flexibility of EML allows one to obtain valuable information that would be inaccessible if a single solution technique was used.

Localsolver makes use of a randomized search algorithm, and the decomposition method that we employ for solving our CP (and SMT) model is also randomized. Therefore, for the LS and CP approaches we performed 10 runs per instance with a time limit of 90 seconds. In CP, a timeout of 2 seconds was enforced on each iteration of the decomposition method. For LS and CP, the table reports in the **obj** columns the average value of the objective function (i.e. minimum core efficiency) over the 10 runs, and within round brackets the standard deviation of the objective value over the 10 runs.

Table 5
Results for the WDP_{max}.

#	DT		
	CP_{max}	LS_{max}	SMT_{max}
	obj	obj	obj
0	29.60 (0.917)	27.70 (0.458)	23.60 (0.800)
1	33.60 (0.800)	32.40 (0.663)	24.80 (0.980)
2	28.70 (1.005)	28.10 (0.700)	22.50 (1.025)
3	25.10 (0.700)	25.10 (0.539)	18.10 (0.943)
4	23.30 (0.458)	23.50 (0.500)	15.00 (2.236)
5	30.50 (1.118)	29.00 (0.632)	23.30 (0.900)
6	39.10 (0.700)	36.90 (0.700)	26.20 (1.778)
7	28.60 (0.800)	27.90 (1.044)	23.00 (1.183)
8	32.90 (1.300)	30.70 (0.640)	24.10 (0.700)
9	37.30 (0.640)	34.90 (1.044)	25.80 (1.600)
10	35.60 (0.917)	33.20 (0.872)	24.10 (0.943)
11	24.30 (0.781)	24.00 (0.000)	15.00 (1.000)
12	33.10 (0.831)	31.30 (0.781)	24.00 (1.095)
13	24.00 (1.000)	24.00 (0.000)	17.40 (1.497)
14	29.60 (0.663)	28.60 (0.917)	23.20 (0.980)
15	32.90 (0.831)	30.80 (0.600)	23.70 (1.100)
16	25.10 (0.700)	24.70 (0.640)	17.20 (2.088)
17	38.90 (0.539)	36.60 (1.020)	26.90 (1.221)
18	26.40 (1.020)	25.20 (0.400)	21.80 (0.748)
19	32.90 (1.136)	31.80 (0.748)	24.70 (1.100)

Both the CP and the LS approaches managed to obtain high quality solutions, as it can be seen on the instances for which an optimal NLP bound is available. The CP approach performed usually better, with gaps as large as $\sim 10\%$ for the simplified model ANN1, and as large as $\sim 7\%$ for the "full" model ANN0, which is significant enough for this application. The performance of the CP approach was also considerably more robust over different runs.

The difference in performance stems from two main reasons: first, the propagation of Neuron Constraints often allows the CP solver to quickly terminate many iterations of the decomposition method. Second, the incorporation of expert knowledge in the selection of the cores to be relaxed allows one to steer the search toward promising regions of solution space.

Results for the WDP_{max} Here we present the results for the second dispatching problem from Section 3. As we anticipated, on this setting we have tested a single Empirical Model, i.e. the Decision Trees extracted in Section 7. We solved the benchmark instances with LS, CP, and SMT. The CP and SMT approaches make use of the decomposition method described at the beginning of Section 8. Each instance was solved 10 times with different random seeds, and with a time limit of 90 seconds. Again a time limit of 2 seconds was enforced on each iteration of the decomposition method.

Table 5 reports, like in previous cases, the average value of the problem objective over the 10 runs and its standard deviation. Similarly to the WDP_{bal}, propagation and domain knowledge allow the CP approach to perform generally better than LS, although the gap is smaller on this problem. LS seems to be slightly more stable than CP, as highlighted by the standard deviation values.

The SMT approach was the worst performer, despite being based on the same decomposition method as CP. The reason is that the Z3 solver had difficulties in proving infeasibility when solving subproblems corresponding to an unlucky selection of the relaxed cores. The CP approach, conversely, was often able to close these subproblems within the 2 seconds time limit.

8.4. Evaluation on the real system

In this section, we compare the solutions provided by different approaches in terms of their quality over the target system, i.e. the SCC simulator from Section 3. By doing so, we evaluate at the same time both the optimization approach and the accuracy of the system model. This allows one to compare radically different approaches designed to solve the same practical problem. This evaluation is restricted to the WDP_{bal}.

We compare solutions obtained using the ACPI, wACPI, FEAT, ANN0, and ANN1 system models. For each of them, we employed the approach that worked best in terms of *predicted* solution quality, hence LS was used for ACPI, wACPI, and FEAT, while CP was used for ANN0 and ANN1. Each instance in the benchmark was solved only once, with a time limit of 90 seconds (and 2 seconds for each decomposition iteration).

The results for this experimentation are reported in Table 6. For each approach we report the minimum (simulated) core efficiency as a percentage. We also report between brackets the standard deviation *of the core efficiencies* (this is not the standard deviation of the cost over different runs, as each instance is solved only once). The reason for showing this standard deviation is that the minimum core efficiency, when used as a quality measure of real solutions, is not very robust w.r.t. approximation errors: a very well balanced solution with a single large approximation error could appear as having very poor quality. Adding the standard deviation allows one to detect this kind of situation.

Table 6
Solution quality of several approaches as measured on the target system. The values within square brackets are standard deviations of the efficiency of the platform cores (for a single solution), rather than standard deviations taken over multiple runs.

#	ACPI	wACPI	FEAT	ANN0	ANN1
	LS	LS	LS	CP	CP
	sim (%)	sim (%)	sim (%)	sim (%)	sim (%)
0	53.5 (14.3)	53.1 (8.3)	58.5 (7.7)	63.8 (6.3)	60.8 (7.0)
1	56.8 (11.1)	52.1 (10.8)	62.8 (7.5)	63.4 (6.7)	65.0 (7.5)
2	53.9 (13.8)	59.7 (9.0)	56.4 (9.1)	58.4 (7.7)	58.0 (8.5)
3	50.2 (13.7)	54.0 (9.2)	48.7 (7.8)	64.3 (7.0)	60.9 (9.1)
4	61.0 (10.8)	53.2 (8.1)	50.0 (8.5)	60.0 (7.2)	46.7 (10.7)
5	52.5 (13.2)	52.5 (8.4)	56.7 (7.6)	65.7 (6.7)	59.2 (7.9)
6	58.1 (12.5)	57.1 (10.6)	56.6 (8.1)	62.9 (7.3)	60.2 (9.1)
7	51.9 (13.3)	58.3 (8.9)	59.6 (8.5)	65.1 (6.4)	67.2 (7.0)
8	60.3 (11.1)	58.7 (8.7)	56.4 (10.5)	63.4 (6.5)	65.2 (7.7)
9	60.1 (11.1)	63.0 (10.1)	60.0 (7.2)	67.7 (5.7)	65.4 (6.6)
10	61.8 (11.6)	53.4 (12.0)	56.2 (9.0)	65.2 (6.7)	62.9 (7.9)
11	59.2 (11.8)	50.1 (8.4)	48.8 (8.3)	57.1 (6.1)	56.0 (10.3)
12	57.0 (11.9)	59.2 (9.2)	56.1 (9.8)	65.2 (7.8)	63.5 (8.3)
13	50.0 (11.0)	54.2 (7.8)	55.1 (8.4)	63.9 (6.4)	59.1 (8.5)
14	51.6 (14.9)	55.6 (8.7)	57.2 (8.9)	59.5 (7.7)	65.2 (7.3)
15	55.3 (11.6)	55.2 (10.1)	60.2 (9.6)	63.1 (7.3)	61.7 (7.1)
16	51.8 (12.1)	56.6 (8.3)	50.8 (7.9)	63.9 (8.0)	56.9 (9.6)
17	60.5 (12.3)	64.7 (9.8)	60.0 (7.2)	67.5 (6.2)	65.7 (8.0)
18	51.0 (12.8)	51.7 (9.0)	55.6 (8.0)	63.5 (6.8)	66.5 (7.3)
19	54.3 (12.5)	53.1 (10.8)	56.6 (8.4)	63.8 (8.3)	66.0 (8.5)

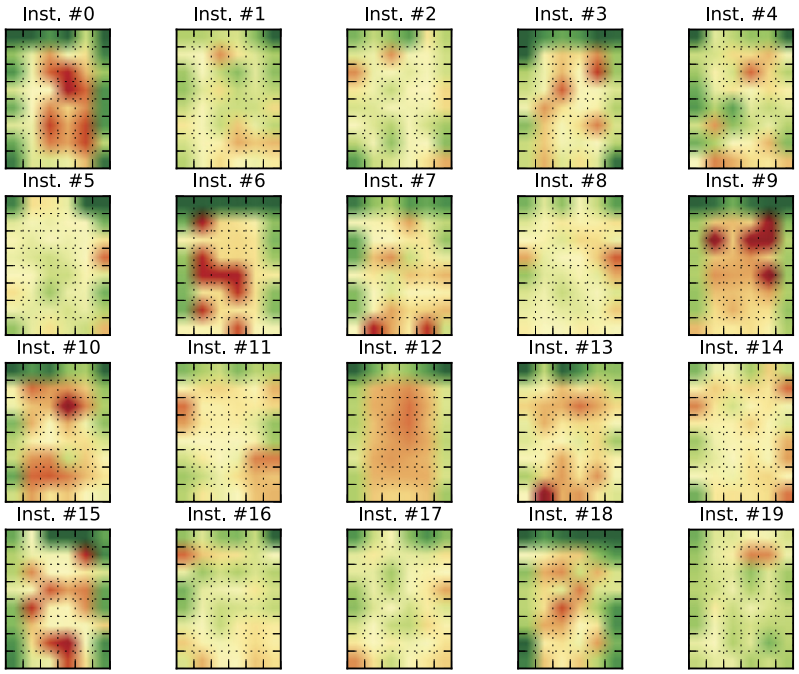

Fig. 7. Simulated efficiency values for $LS_{bal}(ACPI)$. (For interpretation of the references to color in this figure, the reader is referred to the web version of this article.)

The solution approaches based on ANN models obtained in general better results than those with a linear objective, including the FEAT model for which the coefficients were obtained via linear regression. The advantage is in terms of both the minimum efficiency and the standard deviation. The results obtained with the more accurate network (ANN0) tend in turn to be better than those obtained with the simplified ANN1.

The results of the simulation for the ACPI and the ANN0 models can be observed in detail in Fig. 7 and Fig. 8. Each subfigure corresponds to an instance, and each tile within a subfigure corresponds to a core (we recall that the platform has an 8×6 layout). Green/red tiles respectively denote higher/lower efficiency values: a full red corresponds to 50% efficiency, while a full green to 100%. The $LS_{bal}(ACPI)$ approach (Fig. 7) is often able to obtain fairly balanced workloads. However, the produced mappings do occasionally lead to abnormally low efficiency for some cores. This happens even in cases where the

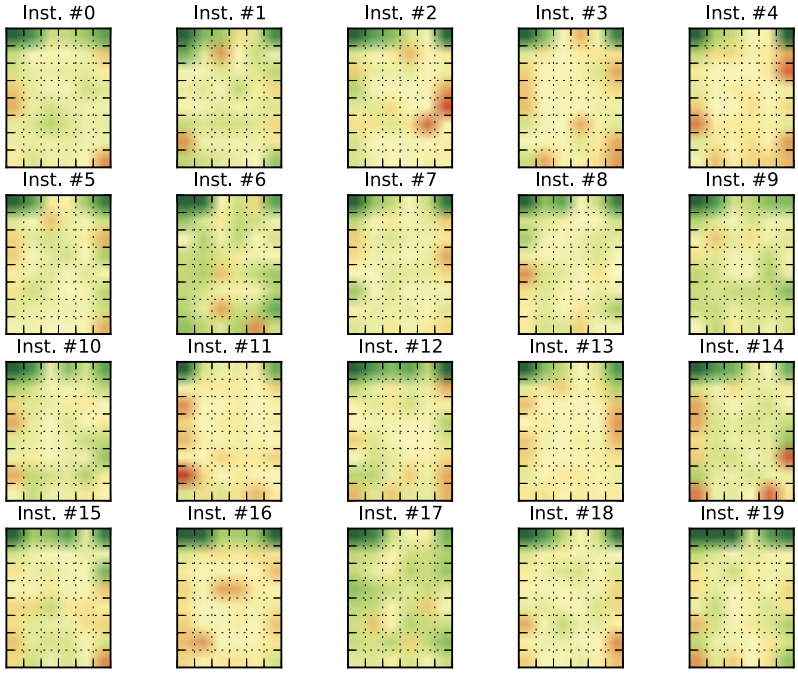

Fig. 8. Simulated efficiency values for $CP_{bal}(ANN0)$. (For interpretation of the references to color in this figure, the reader is referred to the web version of this article.)

solution is close to optimal: by cross-checking the results with those of Table 4, it is possible to see for example that very low efficiency values occur on instance #15, where the cost function has value 2.76 against a bound of 2.80. The efficiency maps from Fig. 8, corresponding to the solutions provided by $CP_{bal}(ANN0)$, are much more uniform.

By comparing the results of Table 6 with those from Table 4, it can be seen that the predicted (minimum) efficiency levels tend to be 5–10% higher than the real values. On this regard, there are two interesting facts to observe: first, despite the significant error level, the CP solutions (with the ANN0 model) are very well balanced in terms of efficiency: this suggests that the overestimation tends to be similar over all cores. Second, the error level is of the same order of the largest errors in the EM evaluation from Section 7. This strengthens the idea that the optimizer may be attracted by solutions with a high predicted quality, but also a large margin of error. This behavior, although certainly not ideal, is actually not totally undesirable: in fact, it may provide a systematic approach to make the Empirical Model more robust by including the simulated solutions in the training set. This method may allow us to improve the model robustness without the need to increase dramatically the training set size. We leave the investigation of this idea as a topic for future research.

9. Concluding remarks

We have proposed here a methodology (called Empirical Model Learning) for merging Machine Learning and optimization by extracting decision model components from data, which may come from a simulator or from the real system. Our emphasis is on defining techniques for embedding ML models in Combinatorial Optimization, which should be designed so that the optimization engine can exploit the ML model for boosting the search process.

We have discussed the main steps of the methodology, using as motivating and running examples two thermal-aware workload dispatching problems. The ML techniques adopted are Artificial Neural Network and Decision Trees that have been encoded in Local Search, Constraint Programming, Mixed Integer Non-Linear Programming (only ANNs) and SAT Modulo Theory (only DTs).

Designing a good empirical model may still be a non-trivial task, but allows for better accuracy w.r.t. to expert designed heuristics. Experiments show the clear advantages of using a data-extracted model in terms of quality of the final solutions. Constraint Programming has been shown to be particularly effective, due to the expressive modeling language and the filtering algorithms.

Further improvements may be obtained by increasing the accuracy of the Empirical Models: for example, using Instructions Per Clock rather than CPIs allows to better characterize computation-intensive jobs, i.e. the most critical for efficiency prediction. We have tested this approach and found that it leads to significant improvements over ANN1, but not over ANN0.

The Empirical Model Learning approach enables the application of optimization techniques to complex real world problems that used to be either very hard or impossible to tackle. As a main benefit, the approach opens up new application areas. In particular, EML may be instrumental in bridging the gap between predictive and prescriptive analytics.

As a particular case, EML could be used to perform decision making over a controlled system. In such a situation, EML allows one to use a high-level optimizer to steer the behavior of the existing controller, with no need to know its internal details and potentially without direct communication. This property could be exploited to ease the integration with legacy systems, and to tackle large scale problems via multi-level optimization.

Active learning could be incorporated in the EML approach by periodically re-training the ML model to improve its accuracy. The same approach could be employed to design self-adapting systems, capable of tracking changes in their operating conditions. In principle, this could include also disruptive events, such as the addition of new system components.

Finally, another interesting topic for future research concerns the relation between predictive models and uncertainty: many Machine Learning models are designed to deal with uncertain data, and can provide information (e.g. probability distributions or confidence intervals) about the robustness of their predictions. These capabilities could be exploited to design more robust and reliable decision making systems for real world problems.

Acknowledgements

This research was partly funded by the Google Focused Grant Program on Mathematical Optimization and Combinatorial Optimization in Europe, with title “Model Learning in Combinatorial Optimization: A Case Study on Thermal Aware Dispatching”.

References

[1] G.E. Box, N.R. Draper, Empirical Model-Building and Response Surfaces, Wiley, New York, 1987, p. 424.
[2] S. Brailsford, L. Churilov, B. Dangerfield, Discrete-Event Simulation and System Dynamics for Management Decision Making, John Wiley & Sons, 2014.
[3] P. Flach, personal communication, 2014.
[4] A. Bartolini, M. Lombardi, M. Milano, L. Benini, Neuron constraints to model complex real-world problems, in: Principles and Practice of Constraint Programming, CP, 2011, pp. 115–129.
[5] A. Bonfietti, M. Lombardi, M. Milano, Embedding decision trees and random forests in constraint programming, in: Integration of AI and OR Techniques in Constraint Programming, CPAIOR, 2015, pp. 74–90.
[6] J. Howard, S. Dighe, et al., A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS, in: International Solid-State Circuits Conference, ISSCC, 2010, pp. 108–109.
[7] K.P. Bennett, E. Parrado-Hernández, The interplay of optimization and machine learning research, J. Mach. Learn. Res. 7 (2006) 1265–1281.
[8] S. Sra, S. Nowozin, S.J. Wright, Optimization for Machine Learning, MIT Press, 2011.
[9] L.D. Raedt, T. Guns, S. Nijssen, Constraint programming for data mining and machine learning, in: Proc. of 24th AAAI Conference on Artificial Intelligence, AAAI, 2010, pp. 1671–1675.
[10] T. Guns, S. Nijssen, L.D. Raedt, k-Pattern set mining under constraints, IEEE Trans. Knowl. Data Eng. 25 (2) (2013) 402–418.
[11] L.D. Raedt, S. Nijssen, B. O'Sullivan, P.V. Hentenryck, Constraint programming meets machine learning and data mining, available at: http://vesta.informatik.rwth-aachen.de/opus/volltexte/2011/3207/pdf/dagrep_v001_i005_p061_s11201.pdf, 2011.
[12] Y. Malitsky, M. Sellmann, Instance-specific algorithm configuration as a method for non-model-based portfolio generation, in: Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR, 2012, pp. 244–259.
[13] F. Hutter, L. Xu, H. Hoos, K. Leyton-Brown, Algorithm runtime prediction: methods & evaluation, Artif. Intell. 206 (2014) 79–111.
[14] L. Kotthoff, I.P. Gent, I. Miguel, An evaluation of machine learning in algorithm selection for search problems, AI Commun. 25 (3) (2012) 257–270.
[15] L. Hernando, A. Mendiburu, J.A. Lozano, Generating customized landscapes in permutation-based combinatorial optimization problems, in: Learning and Intelligent Optimization – LION, in: Lecture Notes in Computer Science, vol. 7997, 2013, pp. 299–303.
[16] C. Bessière, R. Coletta, E.C. Freuder, B. O'Sullivan, Leveraging the learning power of examples in automated constraint acquisition, in: Principles and Practice of Constraint Programming, CP, 2004, pp. 123–137.
[17] C. Bessière, R. Coletta, B. O'Sullivan, M. Paulin, Query-driven constraint acquisition, in: International Joint Conference on Artificial Intelligence, IJCAI, 2007, pp. 50–55.
[18] N. Beldiceanu, H. Simonis, A model seeker: Extracting global constraint models from positive examples, in: Principles and Practice of Constraint Programming, IJCAI, 2012, pp. 141–157.
[19] C. Bessiere, R. Coletta, E. Hebrard, G. Katsirelos, N. Lazaar, N. Narodytska, C.-G. Quimper, T. Walsh, Constraint acquisition via partial queries, in: International Joint Conference on Artificial Intelligence, 2013, pp. 475–481.
[20] B. O'Sullivan, Automated modelling and solving in constraint programming, in: Proc. of the 24th AAAI Conference, 2010, pp. 1493–1497.
[21] N.V. Queipo, R.T. Haftka, W. Shyy, T. Goel, R. Vaidyanathan, P.K. Tucker, Surrogate-based analysis and optimization, Prog. Aerosp. Sci. 41 (1) (2005) 1–28.
[22] A. Cozad, N.V. Sahinidis, D.C. Miller, Learning surrogate models for simulation-based optimization, AIChE J. 60 (6) (2014) 2211–2227.
[23] K. Gopalakrishnan, A.M. Asce, Neural network – swarm intelligence hybrid nonlinear optimization algorithm for pavement moduli back-calculation, J. Transp. Eng. 136 (6) (2009) 528–536.
[24] A.H. Zaabab, Q. Zhang, M. Nakhla, A neural network modeling approach to circuit optimization and statistical design, IEEE Trans. Microw. Theory Tech. 43 (6) (1995) 1349–1358.
[25] R. Battiti, M. Brunato, The LION way: machine learning plus intelligent optimization, LIONlab, University of Trento, 2014.
[26] M. Gavanelli, M. Nonato, A. Peano, S. Alvisi, M. Franchini, Genetic algorithms for scheduling devices operation in a water distribution system in response to contamination events, in: Evolutionary Computation in Combinatorial Optimization, EvoCOP, 2012, pp. 124–135.
[27] A.R. Conn, K. Scheinberg, L.N. Vicente, Introduction to Derivative-Free Optimization, MPS/SIAM Series on Optimization, vol. 8, SIAM, 2009.
[28] C. Audet, A survey on direct search methods for blackbox optimization and their applications, in: Mathematics Without Boundaries, Springer, 2014, pp. 31–56.
[29] D.R. Jones, M. Schonlau, W.J. Welch, Efficient global optimization of expensive black-box functions, J. Glob. Optim. 13 (1998) 455–492.
[30] M.C. Fu, Optimization via simulation: a review, Ann. Oper. Res. 53 (1994) 199–248.
[31] F. Glover, J.P. Kelly, M. Laguna, New advances for wedding optimization and simulation, in: Winter Simulation Conference, vol. 1, WSC, IEEE, 1999, pp. 255–260.
[32] W. Huang, S. Ghosh, S. Velusamy, Hotspot: a compact thermal modeling methodology for early-stage VLSI design, IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 14 (5) (2006) 501–513.

[33] A. Bonfietti, M. Lombardi, The weighted average constraint, in: Principles and Practice of Constraint Programming, CP, 2012, pp. 191–206.
[34] S. Haykin, Neural Networks: A Comprehensive Foundation, 2nd edition, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1998.
[35] P. Belotti, C. Kirches, S. Leyffer, J. Linderoth, J. Luedtke, A. Mahajan, Mixed-integer nonlinear optimization, Acta Numer. 22 (2013) 1–131.
[36] F. Rossi, P.V. Beek, T. Walsh, Handbook of Constraint Programming, Elsevier, 2006.
[37] M. Lombardi, S. Gualandi, A new propagator for two-layer neural networks in empirical model learning, in: Principles and Practice of Constraint Programming, CP, 2013, pp. 448–463.
[38] L. Rokach, O. Maimon, Data Mining with Decision Trees: Theory and Applications, World Scientific Publishing Co., Inc., River Edge, NJ, USA, 2008.
[39] L.D. Moura, N. Bjørner, Z3: an efficient smt solver, in: Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2008, pp. 337–340.
[40] J.C. Régin, Generalized arc consistency for global cardinality constraint, in: Proc. of the 13th National Conference on Artificial Intelligence, vol. 1, AAAI Press, 1996, pp. 209–215.
[41] P. Flach, Machine Learning: The Art and Science of Algorithms That Make Sense of Data, Cambridge University Press, 2012.
[42] J. Heaton, Encog Java and DotNet neural network framework, Heaton Research, Inc., retrieved on July 20, 2010.
[43] J. Quinlan, C4.5: Programs for Machine Learning, Morgan Kaufmann, 1993.
[44] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, I.H. Witten, The WEKA data mining software: an update, ACM SIGKDD Explor. Newsl. 11 (1) (2009) 10–18.
[45] T. Benoist, B. Estellon, F. Gardi, R. Megel, K. Nouioua, Localsolver 1.x: a black-box local-search solver for 0-1 programming, 4OR 9 (3) (2011) 299–316.
[46] M.R. Bussieck, A. Pruessner, Mixed-integer nonlinear programming, SIAG/OPT Newsl. Views News 14 (1) (2003) 19–22.
[47] N.V. Sahinidis, BARON 12.1.0: global optimization of mixed-integer nonlinear programs, User's manual, available at: http://www.gams.com/dd/docs/solvers/baron.pdf, 2013.