Journal of Artificial Intelligence Research 75 (2022) 1633-1699 Submitted 05/22; published 12/22

Automated Dynamic Algorithm Configuration

Steven Adriaensen ADRIAENS@CS.UNI-FREIBURG.DE
André Biedenkapp BIEDENKA@CS.UNI-FREIBURG.DE
Gresa Shala SHALAG@CS.UNI-FREIBURG.DE
Noor Awad AWAD@CS.UNI-FREIBURG.DE
University of Freiburg, Machine Learning Lab

Theresa Eimer T.EIMER@AI.UNI-HANNOVER.DE
Marius Lindauer M.LINDAUER@AI.UNI-HANNOVER.DE
Leibniz University Hannover, Institute of Artificial Intelligence

Frank Hutter FH@CS.UNI-FREIBURG.DE
University of Freiburg, Machine Learning Lab &
Bosch Center for Artificial Intelligence

Abstract

The performance of an algorithm often critically depends on its parameter configuration. While a variety of automated algorithm configuration methods have been proposed to relieve users from the tedious and error-prone task of manually tuning parameters, there is still a lot of untapped potential as the learned configuration is *static*, i.e., parameter settings remain fixed throughout the run. However, it has been shown that some algorithm parameters are best adjusted *dynamically* during execution. Thus far, this is most commonly achieved through hand-crafted heuristics. A promising recent alternative is to automatically *learn* such dynamic parameter adaptation policies from data. In this article, we give the first comprehensive account of this new field of *automated dynamic algorithm configuration (DAC)*, present a series of recent advances, and provide a solid foundation for future research in this field. Specifically, we (i) situate DAC in the broader historical context of AI research; (ii) formalize DAC as a computational problem; (iii) identify the methods used in prior art to tackle this problem; and (iv) conduct empirical case studies for using DAC in evolutionary optimization, AI planning, and machine learning.

1. Introduction

Designing robust, state-of-the-art algorithms requires careful design of multiple components. It is infeasible to know how these components will interact for all possible applications. This is particularly true in the field of artificial intelligence (AI), pursuing ever more general problem-solving methods. This generality necessarily comes at the cost of an increased uncertainty about the problem instances the algorithm will have to solve in practice. To account for this uncertainty, it is common practice to expose difficult design choices as parameters of the algorithm, allowing users to customize them to their specific use case. These algorithm parameters can be numerical (e.g., crossover rate or population size in evolutionary algorithms, and the learning rate or batch size in deep learning), but also categorical (e.g., the choice of optimizer in deep learning, or the choice of heuristic or search operator in classical planning and meta-heuristics).

It is widely recognized that appropriate parameter settings are often instrumental for AI algorithms to reach a desired performance level (Birattari et al., 2002; Hutter et al., 2010;

Probst et al., 2019). In this paper, we will use the term *algorithm configuration* (AC) to refer to the process of determining a policy for setting algorithm parameters as to maximize performance across a problem instance distribution. AC has been widely studied, both in general (Birattari et al., 2002; Hutter et al., 2009; Ansótegui et al., 2009; Hutter et al., 2011; López-Ibáñez et al., 2016), as well as in specific research communities (Lobo et al., 2007; Snoek et al., 2012; Feurer & Hutter, 2019). In this work, we focus on a particular kind of AC that is both (i) automated and (ii) dynamic. This general framework was recently proposed in a conference paper by Biedenkapp et al. (2020), and in this article we provide the first comprehensive treatment of the topic.

Dynamic vs. Static AC: In *static* AC, parameter settings are *fixed prior to execution*, using the information available at that time, and remain invariant during execution. For example, in evolutionary optimization, the population size is commonly set statically, e.g., as a function of the input dimensionality. In contrast, in *dynamic* AC (DAC), parameter settings are *varied during execution* using information that becomes available at run time. For example, in machine learning, while static AC would choose a learning rate, possibly dependent on meta-data (e.g., size or modality of the dataset), DAC would propose a learning rate *schedule* that could additionally be a function of time, alignment of past gradients, training/validation losses, etc. While not all parameters can be varied dynamically, in practice many can, and it often makes sense to do so. As a general motivating use case, consider parameters that (indirectly) control the exploration/exploitation trade-off: Often, it makes sense to explore more early on, and to exploit this knowledge in later stages. Even if the optimal configuration happens to be static, predicting it *upfront* may be very hard, yet the best static configuration may quickly become apparent *while solving* the problem. For instance, if our learning rate is too high, training loss may diverge (Bengio, 2012). Another use case arises in the context of anytime algorithms, that aim to return an as good as possible solution, under an unknown termination criterion (Aine et al., 2009; López-Ibánez & Stützle, 2014). Here, early exploitation to quickly find a good solution, and continual, delayed exploration is often desirable. DAC has been an active research area that has produced various highly practical algorithms leveraging dynamic parameter adaptation mechanisms to empirically outperform their static counterparts, e.g., Reactive Tabu Search (Battiti & Tecchiolli, 1994), CMA-ES (Hansen et al., 2003), and Adam (Kingma & Ba, 2015). Beyond these empirical successes and dedicated case studies (e.g., Senior et al., 2013; van Rijn et al., 2018), the potential of DAC has also been shown theoretically (Moulines & Bach, 2011; Warwicker, 2019; Doerr & Doerr, 2020; Speck et al., 2021).

Automated vs. Manual AC: The difference between manual and automated AC is *who performs AC*: A human or a machine. Over the last two decades, a variety of general-purpose automated algorithm configurators have been proposed that effectively relieve users from the tedious and time-consuming task of optimizing parameter settings manually (Hutter et al., 2009; Ansótegui et al., 2009; Kadioglu et al., 2010; Xu et al., 2010; Hutter et al., 2011; Seipp et al., 2015; López-Ibáñez et al., 2016; Falkner et al., 2018; Pushak & Hoos, 2020). However, there is still a lot of untapped potential, as all of these tools perform static AC. In contrast, *dynamic* AC is mostly done manually. Clearly, the human does not directly adjust the parameters during execution; rather, the mechanisms doing this automatically, e.g., learning rate schedules, are products of human engineering. In this work, we will consider deriving such dynamic configuration policies in an automated and data-driven fashion.

Summary of Contributions: In this article, we provide the first comprehensive account of automated DAC. It subsumes and extends four prior conference papers, in which we

1. established DAC as a new meta-algorithmic framework and proposed solving it using contextual reinforcement learning (Biedenkapp et al., 2020);
2. applied DAC to evolutionary optimization, tackling the problem of step-size adaptation in CMA-ES (Hansen et al., 2003), and showed that existing manually-designed heuristics can be used to guide learning of DAC policies (Shala et al., 2020);
3. applied DAC to AI planning, tackling the problem of heuristic selection in FastDownward (Helmert, 2006), and showed how DAC subsumes static algorithm configuration and can improve upon the best possible algorithm selector (Speck et al., 2021); and
4. presented DACBench, the first benchmark library for DAC, facilitating reproducible results through a unified interface (Eimer et al., 2021b).

Here, we go well beyond this previous work, by

i more thoroughly discussing and classifying related work in different areas (Section 2), placing recent work on automated DAC in its scientific and historical context;

ii establishing a formal problem formulation (Section 3), offering a novel theoretical perspective on DAC and its relation to existing computational problems;

iii discussing possible methods for solving DAC problems (Section 4), beyond reinforcement learning, and classifying previous work according to their methodology;

iv extending and using DACBench (Section 5) to perform empirical case studies that

- demonstrate recent successes of automated DAC
- provide empirical validation for the benchmark library, and
- show that DAC presents a practical alternative to static AC, in various areas of AI: evolutionary optimization (Section 6.1), AI planning (Section 6.2), and machine learning (Section 6.3); and

v discussing current limitations of DAC (Section 7).

As such, we provide the first comprehensive overview of automated DAC, a standard reference and a solid foundation for future research in this area.

2. Related Work

Automated DAC is a recent and underexplored research area. However, it did not arise out of thin air, rather it closely relates to, builds on, and tries to consolidate past research efforts. In this section, we place recent work on automated DAC in its scientific and historical context. We start by introducing the terminology we use (Section 2.1). Then, we situate DAC in the broader context of AI (Section 2.2). Finally, we discuss some specific prior art, covering historical work as well as the most recent developments (Section 2.3).

2.1 Terminology

Algorithm parameters are omnipresent in computer science. Unsurprisingly, no single set of terms has been consistently used when discussing the problem of how to best set them. In this section, we briefly clarify some of the terms we use, relating them to known alternatives.

We use the term *algorithm configuration* (AC) to refer to the process of determining a policy for setting an algorithm's parameters as to maximize performance (or equivalently, minimize cost) across an input distribution. In the classical AC literature (Birattari et al., 2002; Hutter et al., 2009; Ansótegui et al., 2009), this process results in a single parameter setting (i.e., a complete assignment of values to parameters) and is called a *configuration*. Later work generalized AC to produce configurations that are a function of the context in which they are used, e.g., the problem instance at hand (Kadioglu et al., 2010; Xu et al., 2010), and most recently the dynamic execution state (Biedenkapp et al., 2020). We will use the term *configuration policy* to refer to the result of AC in general. To disambiguate the aforementioned AC variants, we add the prefixes *per-distribution* (or also *classical*), *per-instance* and *dynamic*, respectively. Finally, while AC terminology was introduced in the context of attempts to automate this process, the term itself does not imply automation, i.e., we add prefixes *automated* and *manual* to specify whether configuration policies are determined automatically or through a manual engineering process, respectively.

In this work, we follow a meta-algorithmic approach to automating AC: We will treat AC as a computational problem to be solved by executing an algorithm. Hence, we have problem instances and algorithms at two different levels and will use the prefixes *(D)AC* and *target* to disambiguate these: For example, research on automated DAC aims to find a DAC algorithm for tackling the general DAC problem. In a given DAC problem instance, we aim to find a policy for configuring the parameters of a given target algorithm as to optimize its performance across a distribution of target problem instances. We also use *DAC method* and *DAC scenario* as a synonym for DAC algorithm and DAC problem instance, respectively.

In machine learning, the problem of setting the hyperparameters of the learning pipeline is known as *hyperparameter optimization* (HPO, Feurer & Hutter, 2019). We consider the more general problem of setting the parameters of *any* target algorithm and therefore adopt a more general terminology (Eggensperger et al., 2018). In meta-learning terms, *AC problem solving* corresponds to the *outer-loop* and *target problem solving* to the *inner-loop*. More generally, assuming the target is optimization, AC can be seen as a *bilevel* optimization problem (Bard, 2013), where the target algorithm optimizes the inner objective and the AC algorithm the outer objective.

When a parameter maps directly onto the use of a certain sub-procedure, AC has also been called *operator selection*. In metaheuristics (Glover & Kochenberger, 2006), these sub-procedures are commonly called *heuristics* and in hyper-heuristic literature (Pillay & Qu, 2018) the procedure selecting the (low-level) heuristic called a *selection hyper-heuristic* (Drake et al., 2020) corresponding to the configuration policy in our terminology.

In heuristic optimization, the terms *parameter tuning* and *parameter control* are commonly used to refer to static and dynamic algorithm configuration, respectively (Lobo et al., 2007). Also, the terms *online* (during use) and *offline* (before use) are sometimes used as synonyms for *dynamic* and *static*, respectively. In this work, we refrain from doing so, reserving these terms to refer to *when (D)AC takes place* (see Figure 1). In the offline setting, AC takes place in a dedicated *configuration phase* (similar to training in machine learning) where we determine which configuration to use later to solve the problems of actual interest to the user (i.e., at *use time*). In the online setting, AC happens at use time (Fitzgerald, 2021). In that sense, offline and dynamic are not mutually exclusive. In fact, most prior art does DAC offline, determining a dynamic policy offline by using a training set, and at use time simply executing that dynamic policy on new problem instances.

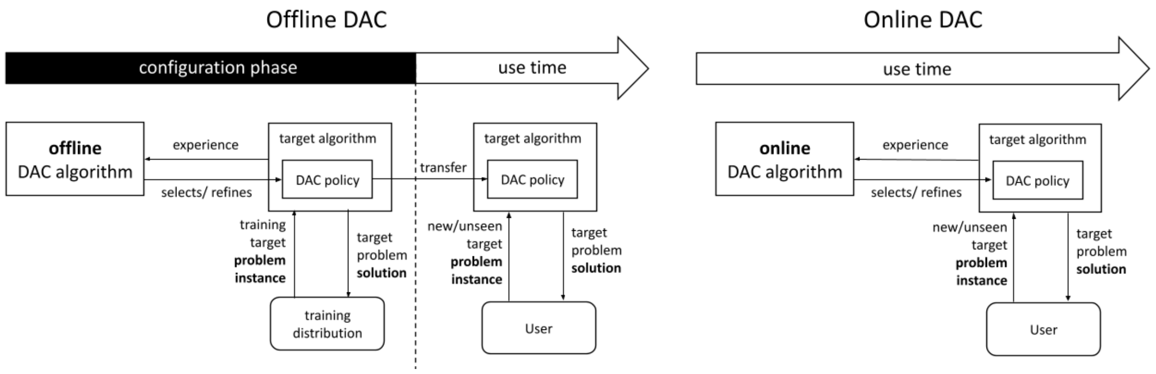


Figure 1: Offline vs. online learning of DAC policies.

2.2 Related Research Areas

While automating DAC is a relatively understudied problem, much research has been performed studying related problems. In what follows, we briefly characterize this work and how it relates to automating DAC. See Appendix B for a more formal treatment of this topic, where we provide problem definitions, possible reductions, and prove their correctness.

2.2.1 Automated Design of Algorithms / Components

The idea of letting computers, rather than humans, design algorithms has been studied in many different communities, using a variety of different methods. Some well-known, historical examples are *program synthesis*, using logical inference (Manna & Waldinger, 1980), and *genetic programming*, using evolutionary algorithms (Koza, 1992). Recent advances in machine learning have prompted a surge in approaches that *learn* algorithms, e.g., *Neural Turing machines* (Graves et al., 2014), *learning-to-learn* (L2L , Andrychowicz et al., 2016; Lv et al., 2017; Bello et al., 2017; Metz et al., 2020), and *learning-to-optimize* (L2O, Li & Malik, 2017; Kool et al., 2018; Chen et al., 2021).

Generally speaking, algorithm parameters can be seen as "algorithmic design choices" that are left open at design time. In that sense, automated configuration is naturally viewed as a way of automating *part of* the algorithm design process. This approach has been referred to as "programming by optimization" (PbO, Hoos, 2012). While previous PbO applications used static AC approaches, the original PbO philosophy envisioned the possibility of varying design decisions at runtime, something naturally achieved by DAC.[1]

1. While we are not aware of any work "under the PbO banner" considering the dynamic setting, most prior art automating DAC (Section 2.3) could be viewed as PbO when taking an algorithm design perspective.

A key difference between PbO and the aforementioned design automation approaches is that in PbO algorithms are not designed "from scratch", instead only design choices that are "difficult" for the human designer are made automatically by the configurator. For instance, DAC aims to design learning rate schedules (e.g., Daniel et al., 2016), but not entire optimizers as in L2L/L2O. In summary, DAC can be viewed as automatically designing parameter controlling components, and "DAC powered PbO" as a general *semi-automated* algorithm design approach that enables the human designer to bias the design process by embedding prior knowledge (e.g., obtained through decades of algorithmic research), thereby reducing the computational requirements and improving generalization.

It is worth noting that other semi-automated algorithm design approaches exist. For instance, genetic programming is nowadays often used to design specific algorithm components, e.g., in the hyper-heuristic literature (Pillay & Qu, 2018), as a procedure for automatically *designing* heuristics, referred to as a *generation hyper-heuristic* (Burke et al., 2009). Finally, if the designed components are reusable and control parameters (e.g., are reusable selection hyper-heuristics as in Fontoura et al., 2017), we view this approach as automated DAC.

2.2.2 Meta-Algorithmic Frameworks

Algorithm Selection Problems can be solved using a variety of different algorithms. For example, if we want to sort a sequence of numbers, we could do so using insertion sort, merge sort, quick sort, etc. In *algorithm selection*, we determine a mapping from features of the problem instance (e.g., sequence length) to the algorithm best suited to solve it (e.g., that sorts the sequence fastest). While first formalized by Rice (1976), this computational problem only received wide-spread attention two decades later, when it was independently rediscovered by Fink (1998), and Leyton-Brown et al. (2003) proposed to solve it using machine learning methods. This approach has resulted in various successful applications, e.g., SATzilla (Xu et al., 2008), a portfolio solver selecting between state-of-the-art SAT solvers to win multiple (gold) medals at the 2007 and 2009 SAT competitions. We refer to Kotthoff (2014) and Kerschke et al. (2019) for surveys on this topic.

Algorithm Scheduling It is often difficult to efficiently predict which algorithm will perform best on a given problem instance. In many settings, poor choices may require orders of magnitude longer than optimal choices, and tend to dominate average performance. In *algorithm scheduling*, instead of selecting a single algorithm, we aim to find an optimal time allocation. Automated algorithm scheduling was first extensively studied in seminal work by Huberman et al. (1997) and Gomes and Selman (2001), and follow-up work, e.g., by Hoos et al. (2015), typically focuses on finding a fixed time allocation that works best on average across instances (i.e., per-distribution). These kind of algorithm schedules are also very popular in the AI planning community, e.g., in Fast Downward Stonesoup (Helmert et al., 2011). Scheduling has also been combined with algorithm selection to find instance-specific schedules (Kadioglu et al., 2011; Lindauer et al., 2016). Dynamic scheduling approaches allocate resources to the algorithms based on runtime information (e.g., Carchrae & Beck, 2004; Gagliolo & Schmidhuber, 2006; Kadioglu et al., 2017; Nguyen et al., 2021). This allows them to exploit the fact that, while it may be difficult to predict which algorithm performs best *in advance*, their relative performance may become apparent early-on in their executions. DAC also takes advantage of this property. However, unlike DAC, dynamic scheduling is restricted to allocating resources to *independent* processes; i.e., in scheduling, no information is exchanged between algorithm runs, and resources allocated to all but the one producing the eventual solution are effectively wasted.

Algorithm Configuration While algorithm selection chooses between multiple target algorithms on a per-instance basis, classical per-distribution algorithm configuration (AC) is concerned with finding the parameter setting of a single algorithm that performs best across all given instances. As the space of possible configurations grows exponentially in terms of the number of parameters, research on AC has traditionally focused on (i) efficient search methods, e.g., local search (Hutter et al., 2009), genetic algorithms (Ansótegui et al., 2009) and Bayesian optimization (Hutter et al., 2011); and (ii) efficient evaluation of configurations, e.g., using racing (Birattari et al., 2002), adaptive capping (Hutter et al., 2009), structured procrastination (Kleinberg et al., 2017) and multi-fidelity optimization (Li et al., 2018). This line of work has resulted in a variety of automated tools known as *configurators* that for any given target algorithm quickly find a configuration that performs well *on average* across a set of target problem instances, e.g., ParamILS (Hutter et al., 2009), GGA (Ansótegui et al., 2009, 2015, 2021), SMAC (Hutter et al., 2011), iRace (López-Ibáñez et al., 2016), and Golden Parameter Search (Pushak & Hoos, 2020); as well as various theoretical insights (Kleinberg et al., 2017; Weisz et al., 2019; Hall et al., 2019, 2020). Configuration has further been combined with algorithm selection (Kadioglu et al., 2010; Xu et al., 2010), and algorithm scheduling (Seipp et al., 2015). However, all of these consider determining a static configuration policy, and the pursuit of similar automated tools and theory for DAC is a natural extension of this line of work. That being said, other works have previously explored tackling the dynamic setting (see Section 2.3) some even using the aforementioned static AC tools (see Section 4.2).

2.2.3 Adaptive Operator Selection and Parameter Control

Heuristic Approaches The potential of varying parameters during execution time is widely recognized and has been extensively studied in various areas of AI. For instance, in heuristic optimization, this problem has been studied in the context of parameter control for evolutionary algorithms (Aleti & Moser, 2016), reactive search (Battiti et al., 2008), and selection hyper-heuristics (Drake et al., 2020). In machine learning, one hyperparameter that is typically varied is the learning rate, e.g., using global learning rate schedules (Loshchilov & Hutter, 2017; Smith, 2017) or adaptive gradient methods (Kingma & Ba, 2015) adopting weight-specific step-sizes. These works typically consider the dynamic configuration policy as a given and present an empirical/theoretical analysis thereof. Furthermore, the policies themselves were designed by human experts. In contrast, automated DAC is concerned with finding such policies automatically in a data-driven fashion. That being said, prior art automating DAC does exist and is discussed in Section 2.3. Before doing so, we will briefly discuss a broad class of methods that rely less on human expert knowledge, but that we nonetheless do not generally regard as automated DAC.

Online Learning Approaches Many parameter control mechanisms integrate complex feedback loops, learning and optimization mechanisms, creating the potential that the DAC policy is not entirely predetermined by the human, but is rather learned online, while solving the problem instance at hand. All depends on the relative contribution to performance due to (i) the exploration of the hand-crafted DAC algorithm, and (ii) the exploitation of the DAC policy it learns. In an offline setting, distinguishing between (i) and (ii) is easy, as (i) does not occur at test/use time. In online settings, both are intertwined by nature. Note that this does not rule out "online DAC", but rather necessitates dedicated analysis that learning indeed takes place. Furthermore, in Section 3.2, we will define DAC as the problem of finding dynamic configuration policies "that generalize across a distribution of target problem instances". Therefore, in our nomenclature, *in order to qualify as automated DAC, an approach must demonstrate the ability to successfully transfer experience across runs of the target algorithm on target problem instances drawn from the same distribution.* In machine learning terms, automated DAC does not only require learning, but also *meta-learning.* Please note that most previous online learning approaches to parameter control (e.g., Müller et al., 2002; Carchrae & Beck, 2004; Chen et al., 2005; Eiben et al., 2006; Prestwich, 2008; Sakurai et al., 2010; Wessing et al., 2011; Gaspero & Urli, 2012; Sabar et al., 2013; Schaul et al., 2013; Karafotias et al., 2014; Baydin et al., 2018) trivially do not meet this criterion, as no information is transferred across runs on different instances. Note that massive parallel online HPO methods such as Population Based Training (PBT, Jaderberg et al., 2017) also fall into this category.

2.3 Prior Art: Automated Dynamic Algorithm Configuration

The term *dynamic algorithm configuration* (DAC) was only recently introduced by Biedenkapp et al. (2020). However, various authors had previously (or, in a few cases, concurrently) investigated the possibility of automatically determining policies for varying the configuration of an algorithm on-the-fly. In what follows, we give a brief overview of literature on automated DAC ("avant-la-lettre").[2] Here, we discuss these by application domain, a methodological overview is presented in Section 4.

Pioneering work by Lagoudakis and Littman (2000, 2001) explored this setting in the context of recursive algorithm selection, observing that sub-problems are better solved using different algorithms (e.g., sorting sub-sequences using different sorting algorithms). While initial results were promising, their approach was limited to recursive target algorithms. Pettinger and Everson (2002) considered a more general setting, learning a policy jointly selecting mutation and crossover operators in a genetic algorithm, per generation, based on statistics of the current population.[3] Various other works have explored automating DAC in the context of genetic algorithms (Fialho et al., 2010; Andersson et al., 2016), evolution strategies (López-Ibáñez et al., 2012; Sharma et al., 2019), and heuristic optimization in general (Battiti & Campigotto, 2012; López-Ibánez & Stützle, 2014; Ansótegui et al., 2017; Fontoura et al., 2017; Sae-Dan et al., 2020). Similar investigations were also conducted in various other communities, e.g., machine learning (Daniel et al., 2016; Hansen, 2016; Fu, 2016; Xu et al., 2019; Almeida et al., 2021), AI planning (Gomoluch et al., 2019, 2020), exact search (Bhatia et al., 2021), and quadratic programming (Getzelman & Balaprakash, 2021; Ichnowski et al., 2021).

2. Appendix A provides a more detailed description. We also maintain a list of work on automated DAC: `https://www.automl.org/automated-algorithm-design/dac/literature-overview/`
3. Notably, direct follow-up works (Chen et al., 2005; Sakurai et al., 2010), no longer transferred experience across runs and is therefore not considered prior art automating DAC (see Section 2.2.3).

Biedenkapp et al. (2020) introduced DAC in an attempt to consolidate these isolated efforts and to raise the level of generality in pursuit of algorithms similar to those that exist for static AC. Direct follow-up work has provided additional evidence for the practicality of DAC by learning step-size adaptation in CMA-ES (Shala et al., 2020), and by learning to select heuristics in the FastDownward planner (Speck et al., 2021). These application domains, together with the learning rate control setting from (Daniel et al., 2016), have later been released as part of a benchmark suite, called DACbench (Eimer et al., 2021b), offering a unified interface that facilitates comparisons between different DAC methods across different DAC scenarios. In this article, we extend this initial discussion of Biedenkapp et al. (2020) and present a thorough empirical comparison of AC and DAC on these three different real-world DAC applications (Daniel et al., 2016; Shala et al., 2020; Speck et al., 2021) using the unified DACbench interface.

3. Problem Definition

In this section, we formalize the computational problem underlying DAC. Here, we first introduce formulations for static AC variants (Section 3.1), and then define the dynamic AC problem (Section 3.2).

3.1 Static Algorithm Configuration

In algorithm configuration, we have some target algorithm $\mathcal{A}$ with parameters $p_1, p_2, \ldots, p_k$ that we would like to configure, i.e., assign a value in the domains $\Theta_1, \Theta_2, \ldots, \Theta_k$, respectively. Furthermore, we may wish to exclude certain invalid combinations, giving rise to the space of candidate configurations $\Theta \subseteq \Theta_1 \times \Theta_2 \times \cdots \times \Theta_k$, called the *configuration space* of $\mathcal{A}$. In classical per-distribution algorithm configuration, we aim to determine a single $\boldsymbol{\theta}^* \in \Theta$ that minimizes a given cost metric c in expectation across instances $i \in I$ of our target problem distribution $\mathcal{D}$. This problem can be formalized as follows:

Definition 1: Classical / Per-distribution Algorithm Configuration (AC)

Given $\langle \mathcal{A}, \Theta, \mathcal{D}, c \rangle$:

- A target algorithm $\mathcal{A}$ with configuration space Θ
- A distribution $\mathcal{D}$ over target problem instances with domain I
- A cost metric $c : \Theta \times I \to \mathbb{R}$ assessing the cost of using $\mathcal{A}$ with $\boldsymbol{\theta} \in \Theta$ on $i \in I$

Find a $\boldsymbol{\theta}^* \in \arg\min_{\boldsymbol{\theta} \in \Theta} \mathbb{E}_{i \sim \mathcal{D}} [c(\boldsymbol{\theta}, i)]$.

In practice, $\mathcal{A}$, $\mathcal{D}$, and c are not given in closed form. Instead, c is typically a black-box procedure that executes $\mathcal{A}$ with configuration $\boldsymbol{\theta}$ on a problem instance i and quantifies cost as a function of the desirability of this execution, e.g., how long the execution took, the quality of the solution it found, or a measure of anytime performance (López-Ibánez & Stützle, 2014). Note that $\mathcal{D}$ is our true target distribution, i.e., the likelihood $\mathcal{A}$ is presented with an instance i at use time. In the online setting, we are given a sequence of samples from the actual distribution in real time (Fitzgerald, 2021). In the offline setting, we typically do not have access to $i \sim \mathcal{D}$, and are given a set of instances I' sampled i.i.d. from some representative training distribution $\mathcal{D}' \approx \mathcal{D}$ instead.

Note that unless a single configuration is non-dominated on all instances, better performance may be achieved by making the choice of $\boldsymbol{\theta}$ dependent on the problem instance i at hand. This extension is known as:

Definition 2: Per-instance Algorithm Configuration (PIAC)

Given $\langle \mathcal{A}, \Theta, \mathcal{D}, \Psi, c \rangle$:

- A target algorithm $\mathcal{A}$ with configuration space Θ
- A distribution $\mathcal{D}$ over target problem instances with domain I
- A space of *per-instance configuration policies* $\psi \in \Psi$ with $\psi : I \to \Theta$ that choose a configuration $\boldsymbol{\theta} \in \Theta$ for each instance $i \in I$.
- A cost metric $c : \Psi \times I \to \mathbb{R}$ assessing the cost of using $\mathcal{A}$ with $\psi \in \Psi$ on $i \in I$

Find a $\psi^* \in \arg\min_{\psi \in \Psi} \mathbb{E}_{i \sim \mathcal{D}} [c(\psi, i)]$

Note that the definition above is highly general. For example, by specifying Ψ accordingly PIAC can put arbitrary hard constraints on the configuration policies of interest. As a consequence, classical per-distribution AC can be seen as a special case of PIAC only considering constant ψ, i.e., $\Psi = \{\psi \,|\, \psi(i) = \psi(i'), \forall\, i, i' \in I\}$. More generally, configuration policies could be restricted to be a function of specific features of i, or to belong to a specific (e.g., linear) function class. Note that this definition is also strictly more general than *unconstrained* PIAC, which is itself a special case. Also worth noting is that the cost metric c in this definition can be any function of ψ (and i), allowing PIAC to put arbitrary soft constraints on Ψ. In particular, we do not constrain c to be a function of $\psi(i)$, but allow it to quantify non-functional aspects of ψ, e.g., its minimal description length or computational complexity. That being said, most practical PIAC approaches are limited to minimizing $\mathbb{E}_{i \sim \mathcal{D}} [c'(\psi(i), i)]$, given some $c' : \Theta \times I \to \mathbb{R}$ ($\sim$ *pure functional* PIAC).

Algorithm 1 Stepwise execution of a dynamically configured target algorithm $\mathcal{A}$

Input: Dynamic configuration policy $\pi \in \Pi$; target problem instance $i \in I$
Output: Solution for i found by $\mathcal{A}$ (following π)

```
1: procedure $\mathcal{A}(\pi, i)$
2:     $s \leftarrow$ init($i$)                ▷ Initial state by starting the execution of $\mathcal{A}$ on $i$
3:     while $\neg$ is_final($s, i$) do
4:         $\boldsymbol{\theta} \leftarrow \pi(s, i)$          ▷ Reconfiguration point: Use $\pi$ to choose next $\boldsymbol{\theta}$
5:         $s \leftarrow$ step($s, i, \boldsymbol{\theta}$)                  ▷ Continue executing $\mathcal{A}$ using $\boldsymbol{\theta}$
6:     return s                             ▷ Execution terminated: Return solution
```

3.2 Dynamic Algorithm Configuration

In dynamic AC, we aim to optimally vary $\boldsymbol{\theta} \in \Theta$ while executing $\mathcal{A}$. In order to formalize this problem, we need to define points of interaction where $\mathcal{A}$ can be reconfigured. To this end, we decompose the execution of $\mathcal{A}$ with dynamic configuration policy $\pi \in \Pi$ on problem instance $i \in I$ as shown in Algorithm 1. Here, we start executing an "`init`" sub-routine bringing $\mathcal{A}$ in some initial state $s \in \mathcal{S}$ only depending on i. Subsequently, we iteratively execute "`step`" to determine the next state $s' \in \mathcal{S}$ of $\mathcal{A}$ as a function the current state s, instance i, and configuration $\pi(s, i) \in \Theta$. This process continues until $\texttt{is\_final}(s, i)$ signalling termination and s is returned as solution. When such decomposition $\langle \texttt{init}, \texttt{step}, \texttt{is\_final} \rangle$ is given, we will call $\mathcal{A}$ *stepwise reconfigurable* and define DAC as follows:

Definition 3: Dynamic Algorithm Configuration (DAC)

Given $\langle \mathcal{A}, \Theta, \mathcal{D}, \Pi, c \rangle$:

- A *stepwise reconfigurable* target algorithm $\mathcal{A}$ with configuration space Θ.
- A distribution $\mathcal{D}$ over target problem instances with domain I
- A space of *dynamic configuration policies* $\pi \in \Pi$ with $\pi : \mathcal{S} \times I \to \Theta$ that choose a configuration $\boldsymbol{\theta} \in \Theta$ for each instance $i \in I$ and state $s \in \mathcal{S}$ of $\mathcal{A}$
- A cost metric $c : \Pi \times I \to \mathbb{R}$ assessing the cost of using $\pi \in \Pi$ on $i \in I$.

Find a $\pi^* \in \arg\min_{\pi \in \Pi} \mathbb{E}_{i \sim \mathcal{D}} [c(\pi, i)]$

Here, we define DAC as a generalization of PIAC, considering configuration policies that do not only depend on i, but also the dynamically changing state $s \in \mathcal{S}$ of the target algorithm $\mathcal{A}$, i.e., $\Psi \subseteq \{\pi \,|\, \pi(i, s) = \pi(i, s'), \forall\, s, s' \in \mathcal{S}, \forall\, i \in I\}$. This dynamic state, by definition, provides all information required for continuing the execution of $\mathcal{A}$, however can additionally contain arbitrary features of the execution thus far. As in PIAC, c can be an arbitrary function of π (and i). However, often the total cost of executing $\mathcal{A}$ with π on i can be decomposed and attributed to the T individual execution steps. Formally: In DAC with *stepwise decomposable cost*, we are given functions $\langle \mathrm{c}_{\mathrm{init}}, \mathrm{c}_{\mathrm{step}} \rangle$, such that

$$c(\pi, i) = \mathrm{c}_{\mathrm{init}}(i) + \sum_{t=0}^{T-1} \mathrm{c}_{\mathrm{step}}(s_t, i, \pi(s_t, i))$$

$$\text{where} \qquad s_t = \begin{cases} \texttt{init}(i) & t = 0 \\ \texttt{step}(s_{t-1}, i, \pi(s_{t-1}, i)) & t > 0 \end{cases} \quad \wedge \quad \texttt{is\_final}(s_t, i) \Leftrightarrow t = T.$$

Note that, by this definition, a stepwise decomposable cost metric can, but does not have to be dense, e.g., attributing the full cost (e.g., final solution quality) to the final step (zero otherwise) is a valid decomposition. However, since $\mathrm{c}_{\mathrm{init}}$ and $\mathrm{c}_{\mathrm{step}}$ only depend on i and $\pi(s_t, i)$, a stepwise decomposable cost metric cannot measure non-functional aspects of π. Rather, it captures the notion of *pure functional* DAC and subsumes *pure functional* PIAC, but not PIAC in general.

4. Solution Methods

In this section, we discuss methods for solving DAC. As discussed in Section 2.2.3, DAC has so far been primarily solved manually, i.e., dynamic configuration policies have been determined by humans and not in an automatic and data-driven way. In Section 2.3, we discussed previous work exploring *automated* DAC, and in what follows we will give an overview of the methods they used for doing so. Please note that no dedicated, general DAC solvers exist to date. Instead, prior art can be viewed as solving DAC *by reduction* to some other well-studied computational problem.[4] Considering the fact that most of this work has been performed in isolation and tackles very different DAC scenarios, the high-level solution approaches followed are remarkably similar. In particular, we will roughly distinguish between two approaches: "DAC by reinforcement learning" (Section 4.1) and "DAC by optimization" (Section 4.2), and discuss their relative strengths and weaknesses (Section 4.3).

4.1 DAC by Reinforcement Learning

In reinforcement learning (RL, Sutton & Barto, 2018), an agent learns to optimize an unknown reward signal by means of interaction with an unknown environment. The RL agent takes actions $a \in A$, observes a transition T from the current state $s \in S$ of the environment to $T(s, a) \in S$, receives a reward $R(s, a) \in \mathbb{R}$, and learns for any state the action maximizing its expected future reward. Formally, the RL agent solves a Markov decision problem $\langle S, A, T, R \rangle$ (MDP, Definition 10 in Appendix B.3.5). Here, the transition T and reward R are given in the form of a black box method. Also, the state space S is typically not given explicitly; instead, we are given a procedure for generating initial states and can generate further states using T.

The RL problem described above is closely related to DAC, and prior art has commonly solved DAC using reinforcement learning methods. In DAC by RL, the environment consists of the target algorithm $\mathcal{A}$ solving some target problem instance $i \in I$. The state of this environment is $s = (s', i) \in S$ with $s' \in \mathcal{S}$ the state of the algorithm, and initial states $(\texttt{init}(i), i)$ with $i \sim \mathcal{D}$. At every reconfiguration point, the RL agent interacts with this algorithm choosing a configuration $\theta \in \Theta$ as action. The transition dynamics of the environment are fully determined by stepwise algorithm execution, i.e., $T((s', i), \theta) = (\texttt{step}(s', i, \theta), i)$, and the reward is $R((s', i), \theta) = -\,\mathrm{c}_{\text{step}}(s', i, \theta)$. See Figure 2 for an illustration of this approach. The power of this reduction lies in the fact that the resulting MDP can be solved using the full gamut of existing reinforcement learning methods.

Traditional RL Early *DAC by RL* work (e.g., Lagoudakis & Littman, 2000, 2001; Pettinger & Everson, 2002; Battiti & Campigotto, 2012) used traditional value-based RL methods that learn the optimal state-action value function $Q^*(s, a)$ and return the policy $\pi(s) \in \arg\max_{a \in A} Q^*(s, a)$. These methods work well when $S \times A$ is small enough to be represented explicitly by a table, but do not scale up. Note that both S and $A = \Theta$ are typically too large in DAC to be modelled in tables.

Modern RL Over the last decade, a series of methodological advances have given rise to a new generation of RL methods that can tackle complex real-world problems (Mnih

4. In Appendix B, we define these related computational problems and discuss reductions more formally.

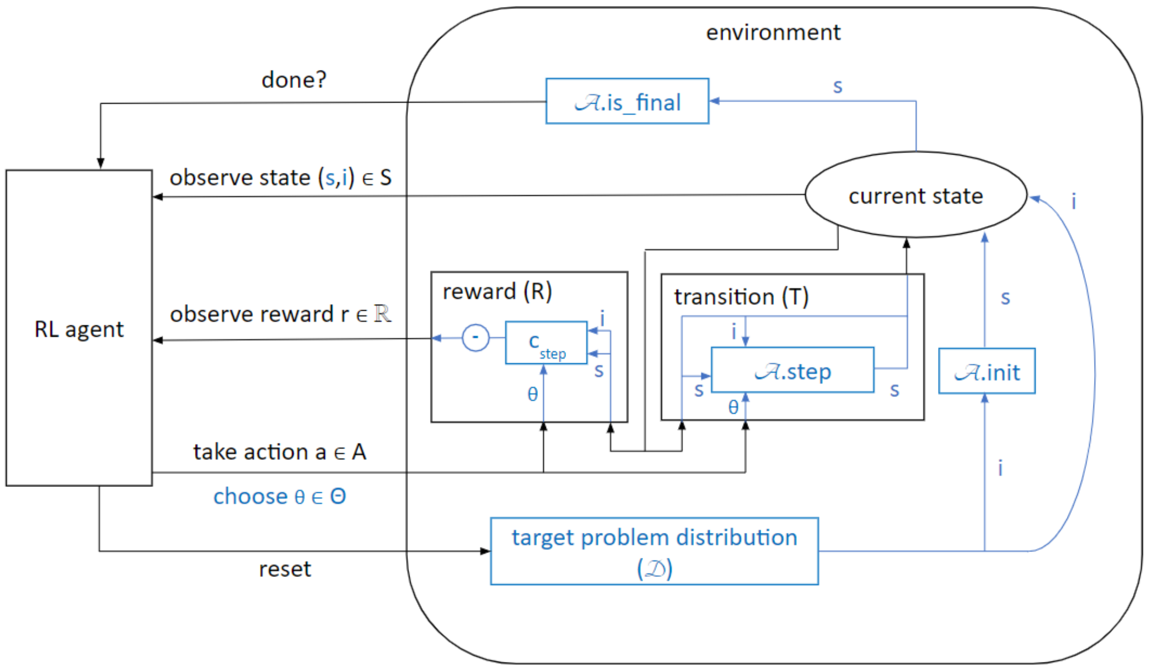


Figure 2: Illustration of DAC by Reinforcement Learning (DAC components in blue)

et al., 2015; Silver et al., 2016; Barozet et al., 2020; Lee et al., 2020), and that have also been successfully applied to DAC. In particular, modern RL methods based on deep neural networks can effectively learn useful representations that allow them to handle complex state and action spaces, using, e.g., (double) deep Q-learning (DDQN, Hansen, 2016; Sharma et al., 2019; Speck et al., 2021; Bhatia et al., 2021), modern actor critic (Ichnowski et al., 2021), and policy gradient methods (Daniel et al., 2016; Xu et al., 2019; Gomoluch et al., 2019; Shala et al., 2020; Getzelman & Balaprakash, 2021; Almeida et al., 2021).

Contextual RL It is worth noting that standard RL methods are not *instance-aware* and will generally not choose their initial state (see Figure 2, where i is hidden inside the environment). This is one of the reasons Biedenkapp et al. (2020) proposed to model DAC as a *contextual* MDP (cMDP, Hallak et al., 2015), which consists of a collection of MDPs, one for each instance i (see Definition 11 in Appendix B.3.5). Each MDP $\mathcal{M}(i)$ shares a common action space S and state space A as in traditional RL, but possesses an instance-specific transition function T_i and reward function R_i. This more general formulation allows DAC practitioners to explicitly model variation between instances: Variations in transition dynamics model the differences in target algorithm behaviour between instances (i.e., how the target algorithm progresses in solving an instance) while different reward functions reflect the instance-specific objectives. Although a single MDP can capture these dependencies implicitly, the explicit model allows the contextual RL agent to directly exploit this knowledge. For example, instances may vary in difficulty. A contextual RL agent, being aware of different instances and their characteristics, can more easily learn this, allowing the agent to more accurately assign credit for high/low rewards to (i) following a good/poor policy or (ii) solving easy/hard instances. Furthermore, the agent can choose which MDP $\mathcal{M}(i)$ it interacts with, e.g., to gather more experience on harder instances (Klink et al., 2020; Eimer et al., 2021a).

4.2 DAC by Optimization

Not all prior art automating DAC has done so using reinforcement learning. Instead, some previous works can be viewed as reformulating DAC as a (non-sequential) optimization problem: Given a search space Π and an objective function $f(\pi) = \mathbb{E}_{i\sim\mathcal{D}}\left[c(\pi, i)\right]$, find $\pi^* \in \arg\min_{\pi\in\Pi} f(\pi)$. This approach is illustrated in Figure 3. Optimization covers a wide variety of different methods. In what follows, we give an overview of those used in prior art for "DAC by optimization", and distinguish between different variants of optimization depending on (i) search space representation, and (ii) what information about f is used.

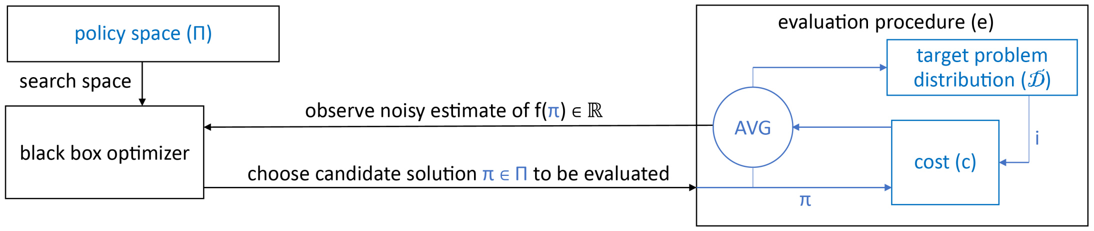

policy space (Π)
search space
black box optimizer
observe noisy estimate of f(π) ∈ ℝ
choose candidate solution π ∈ Π to be evaluated
evaluation procedure (e)
target problem distribution (𝒟)
AVG
cost (c)
i
π

Figure 3: Illustration of DAC by Optimization (DAC components in blue)

Noisy Black Box Optimization In black box optimization (BBO), the only interaction between f and the optimizer is through an evaluation procedure e that returns $f(\pi)$ for any given $\pi \in \Pi$. A wide variety of black box optimizers exist, specialized for particular kinds of representations. In the reduction, dynamic configuration policies can be represented in a variety of different ways. For example, prior art (Gomoluch et al., 2020) represents policies as real-valued vectors that correspond to the weights of a neural network policy, and optimizes these using evolution strategies. It is worth noting that a similar approach is currently state-of-the-art in learning-to-learn (Metz et al., 2020) (see Section 2.2.1). However, one could go further and also vary the architecture and optimize directly in the space of neural networks, e.g., using methods from neuroevolution (Stanley & Miikkulainen, 2002; Stanley et al., 2021). Alternatively, one could follow a symbolic approach, like Fontoura et al. (2017), representing policies as programs and use genetic programming (Koza, 1992). Remark that this freedom comes with responsibility, i.e., making an appropriate choice of representation may be crucial to achieve satisfactory performance. Next to representation, another difficult choice in this reduction is the evaluation procedure. Since $\mathcal{D}$ is unknown, e cannot evaluate f exactly in general. Instead, we typically evaluate the cost on some finite sample of target problem instances $I' \subseteq I$ with $\forall i \in I' : i \sim \mathcal{D}$, and $e(\pi) = \frac{1}{|I'|}\sum_{i\in I'} c(\pi, i)$. However, the choice of $|I'|$ still poses a trade-off between accuracy and cost of evaluation to the DAC by BBO practitioner.

Static Algorithm Configuration We can also solve DAC using classical static algorithm configurators (e.g., SMAC and irace). Assuming we choose a parametric representation Λ for the policy space, i.e., $\Pi = \{\pi_{\boldsymbol{\lambda}} \,|\, \boldsymbol{\lambda} \in \Lambda\}$, the DAC problem can be reformulated as classical AC, where we configure the parameters $\boldsymbol{\lambda}$ of the dynamic configuration policy $\pi_{\boldsymbol{\lambda}}$, instead of configuring the parameters $\boldsymbol{\theta}$ of the target algorithm.[5] While solving DAC using static AC may at first sight seem contradictory, this reduction gives rise to a highly practical solution approach that has been explored extensively in prior art (Fialho et al., 2010; López-Ibáñez et al., 2012; López-Ibánez & Stützle, 2014; Andersson et al., 2016; Ansótegui et al., 2017; Sae-Dan et al., 2020). An important benefit specific to this approach is that algorithm configurators are *instance-aware* and therefore automate the trade-off between the accuracy/cost of evaluation (using so-called *racing* mechanisms), and can even vary $I' \subset I$ dynamically to focus evaluation on those instances providing the most useful information.

Gradient-based Optimization In AC, we typically use gradient-free optimization. The motivation is that we cannot generally compute analytical gradients. While this is true *in general*, we would like to argue that the specific cases where we can actually compute them are more prominent than one might expect. Assuming a stepwise decomposable cost, we can compute the derivative $\nabla_{\boldsymbol{\lambda}} c_i = \frac{\partial c(\pi_{\boldsymbol{\lambda}}, i)}{\partial \boldsymbol{\lambda}}$ from the derivatives of the stepwise cost, the step, and the policy, using the chain rule (see Appendix B.3.6). When $\mathrm{c}_{\mathrm{step}}$, `step`, and π can be implemented using the operations in an automated differentiation framework (e.g., autograd in Pytorch, Paszke et al., 2017), these gradients can be calculated efficiently, reliably, without requiring any additional mathematical knowledge from the DAC practitioner. In fact, in the machine learning community, in particular meta-learning, differentiating through the entire learning process is almost standard practice (Maclaurin et al., 2015; Andrychowicz et al., 2016; Finn et al., 2017). The potential benefit of this extra piece of information is not to be underestimated. DAC policies may have many hyperparameters, e.g., a neural network with thousands of weights. Gradient-based optimization is an efficient way to navigate extremely high-dimensional spaces, as is evidenced by deep neural networks with millions of parameters being trained almost exclusively using simple first order optimization methods. That being said, gradients for DAC are no silver bullet. Computing them, if possible, may require too many computational resources. Furthermore, gradients only provide local information, i.e., an infinitesimal change to every parameter that is guaranteed to reduce cost. When f is particularly rugged, gradients may not provide information about the effect of any reasonably sized change. This phenomenon has, in fact, been observed in the context of learning-to-learn (Metz et al., 2019).

4.3 Reinforcement Learning vs. Optimization?

Now, we discuss the relative strengths and weaknesses, and argue for the potential of combining both approaches.

Why DAC by RL? The sequential nature of the problem is arguably the key feature that sets DAC apart from static AC: In static AC, we only have to select a single configuration, while in DAC we must select a sequence of such configurations. RL provides a very general framework for tackling sequential decision problems and was presented as the method of choice for DAC by Biedenkapp et al. (2020). DAC by optimization approaches reduce DAC to a non-sequential optimization problem. In doing so, valuable information about

5. We prove the correctness of this reduction in Appendix B.3.1

the problem is lost that may otherwise be used to solve it more efficiently (Adriaensen & Nowé, 2016). While executing a target algorithm, an RL agent observes at *every step* what configurations were used, the (immediate) costs this incurred, and how this affected the dynamic state of the algorithm. In contrast, the same evaluation provides a black box optimizer with a single value (i.e., the sum of costs incurred), at the end of the run. This inherent relative sample-inefficiency of black box optimization is particularly problematic when target algorithm execution is costly, e.g., takes multiple hours, and/or policy spaces are extremely large / unconstrained.

Why DAC by Optimization? Previous work has shown that optimization can be a practical alternative to RL in simulated environments (Mannor et al., 2003; Szita & Lörincz, 2006; Salimans et al., 2017; Chrabaszcz et al., 2018; Majid, 2021). While RL aims to exploit sequential information, contemporary RL methods do not always do so successfully. Also, in some scenarios, this information may not add much value, or may even be deceptive (e.g., delayed rewards). Finally, these mechanisms add considerable computational overhead, and complicate implementation. In contrast, optimization methods tend to be simpler, have fewer failure modes, and their often-parallel nature makes them well-suited for modern high-performance computing infrastructure. Adding to these limitations of RL methods are limitations of the reduction. While DAC is generally reducible to a (noisy) black box optimization problem, the previously discussed reduction to an MDP implicitly assumes (i) the cost function c to be *stepwise decomposable*[6] and (ii) the space of policies Π to be *unconstrained*.[7] As a consequence, it cannot be used when optimizing non-functional aspects of the policy (e.g., resources it requires to make decisions) or to impose arbitrary hard constraints on Π (e.g., which of these N policies is best?).

Beyond RL *or* Optimization Our discussion thus far focused on contrasting both approaches. In what remains, we look at their relation, and argue for the potential of combining them. First, our "sequential vs. non-sequential" discussion can be extended to "a method's ability to exploit a certain characteristic of DAC", or not. A good example of a cross-cutting characteristic is *instance-awareness*, both contextual RL and static AC can be viewed as instance-aware extensions of RL and black box optimization, respectively. Second, the pitfalls of RL also apply to approaches exploiting other characteristics. For example, gradients in optimization can be similarly deceptive (e.g, exploding/vanishing gradient problem) as immediate rewards. Therefore, while artificially hiding information is useless, blindly relying on it introduces failure modes, and general DAC methods should be carefully designed to only rely on information that is available and useful for the scenario at hand. In the context of "sequential vs. non-sequential", this suggests the importance of combining reinforcement learning *and* optimization. Further underpinning this conjecture, is the observation that state-of-the-art *static* AC methods combine optimization with machine learning, and reinforcement learning is essentially a dynamic extension of the latter.

6. As defined at the end of Section 3.2 ($\sim$ *pure functional* DAC). This is not to be confused with a dense reward signal that, while desirable for RL, is not a requirement for MDP reducibility.
7. MDP extensions exist, e.g., *constrained MDPs* (CMDPs, Altman, 1999), that support hard constraints. However, CMDPs conventionally constrain agent behavior (actions taken in encountered states) and still cannot encode arbitrary non-functional hard constraints in policy space.

5. Benchmark Library

In this section, we present DACBench (Eimer et al., 2021b), a novel benchmark library for DAC that we will be using in our experiments in Section 6. We have seen related fields like hyperparameter optimization, static algorithm configuration and algorithm selection profit greatly from focusing on shared benchmark problems (Eggensperger et al., 2013; Hutter et al., 2014; Bischl et al., 2016). In these meta-algorithmic domains, standardizing the target algorithm setup did not only increase the accessibility of the field by reducing some of the specialized knowledge required to get started in the field, but it also made comparisons between different methods more reliable and reproducible. DACBench provides such a standard for DAC. In what follows, we give a brief overview of the interface it provides, the benchmarks it implements, and prior empirical validation it has undergone. We also discuss novel developments and highlight extensions that were motivated by and/or made specifically in the context of this work.[8]

Interface DACBench builds upon a common RL interface, OpenAI's gym (Brockman et al., 2016), as it provides a flexible template for stepwise interaction with the target algorithm. The target algorithm `init` is handled in the `gym.Env.reset` method, with each stepwise interaction handled by the `gym.Env.step` method. DACBench extends the `gym.Env.reset` method to provide the ability to select the problem instance i to be solved. Listing 1 shows how DAC components are mapped onto the gym interface in the benchmarks. These essentially implement the DAC by contextual RL reduction, discussed in Section 4.1. The result is a simple-to-use interface, allowing DAC researchers to work across application domains, without requiring domain expertise, and providing an easy-to-use template for applying DAC to new domains. While the interface is modelled after the RL formulation of DAC, it can be used with a variety of approaches described in Section 4.2. That being said, the original DACbench interface strongly focused on conventional RL. In the scope of this work, we have extended the interface from the first release of DACBench. In accordance with our proposed definition of DAC, we have taken a broader perspective beyond standard RL, and made various interface changes to provide better support for alternative approaches. For example, users can now specify rich structured configuration spaces as opposed to the simplistic action spaces supported by conventional RL methods. Directly controlling instance progression is easier now as well, providing a better base for developing instance-aware solution methods for DAC.

Benchmarks An overview of the benchmarks currently included in DACBench is given in Table 1. It includes several benchmarks that we have either added in the latest release or at least improved significantly. The original SGD-DL benchmark (see Section 6.3 for a thorough description) was extended to mimic the experimental setup from Daniel et al. (2016) as closely as possible. The CMA-ES benchmarks (CMAStepSize and ModCMA) are now based on IOHProfiler (Doerr et al., 2018) and thus provide a DAC interface for a well-known and important tool in the EA community.[9] TheoryBench is a completely new benchmark, published by Biedenkapp et al. (2022), where one is to dynamically configure the mutation rate of a (1+1) random local search algorithm for the LeadingOnes problem.

8. A new version of `https://github.com/automl/DACBench` (v. 0.1) was released alongside this article.
9. The original *pycma* version of CMAStepSize is still supported and used in Section 6.1.

This is a particularly interesting setting as the exact runtime distribution is very well understood (Doerr, 2019). In particular, it is possible to compute optimal dynamic configuration policies for various different problem sizes and configuration spaces. Finally, a continuous

```
class DACEnv(gym.Env):

    def __init__(self, 𝒜, Θ, 𝒟, Π_φ, c_step):
        self.𝒜, self.𝒟, self.c_step, self.φ = 𝒜, 𝒟, c_step, Π_φ.φ
        self.action_space = Θ

    def reset(self, i=sample(self.𝒟)):
        s = self.𝒜.init(i)
        self.state = (s, i)
        return self.φ(s, i)

    def step(self, θ):
        s, i = self.state
        s = self.𝒜.step(s, i, θ)
        r = self.c_step(s, i, θ)
        done = self.𝒜.is_final(s, i)
        self.state = (s, i)
        return self.φ(s, i), r, done, None
```

Listing 1: A generic python implementation of a gym environment using components of a DAC scenario (in blue) with decomposable cost and an input-constrained policy space Π_ϕ. Practical DACBench benchmarks implement a similar mapping, but DAC components are typically not strictly separated, e.g., $\mathcal{A}$.`step` and c_{step} would typically be calculated jointly. Note that the `gym.Env.step` method, despite its name, does far more than merely computing $\mathcal{A}$.`step`: It implements the transition dynamics (T) and reward signal (R). Furthermore, unlike $\mathcal{A}$.`step`, it is stateful, does not take the state (s, i) as input, and does not necessarily return the new state. Instead, it more generally returns what is called an observation $\phi(s, i)$ which may *abstract* arbitrary aspects of the internal state, i.e., DACBench technically reduces DAC to a contextual *partially observable* MDP (cPOMDP). Note that the learned policy in POMDPs is a function of the *observable* state, and hence ϕ can be viewed as modeling a policy space constraint of the form $\Pi_\phi = \{\pi \mid \phi(s, i) = \phi(s', i') \implies \pi(s, i) = \pi(s', i')\}$.

Benchmark	Domain	Status	Description
Sigmoid	Toy	Extended	Control k parameters to trace a different sigmoids each (Biedenkapp et al., 2020).
Luby	Toy	Original	Select the correct next term in a shifted luby sequence (Biedenkapp et al., 2020).
CMAStepSize	EA	Extended	Control the step size in CMA-ES (Shala et al., 2020).
FastDownward	Planning	Original	Control heuristic selection in FastDownward (Speck et al., 2021).
SGD-DL	DL	Extended	Control the SGD for neural network training (Daniel et al., 2016).
TheoryBench	EA	New	Control the mutation rate of (1+1)RLS for LeadingOnes (Biedenkapp et al., 2022).
ModCMA	EA	New	Control design choices (e.g., base sampler used) of CMA-ES (Vermetten et al., 2019).
ToyGD	Toy	New	Control the learning rate of gradient descent on polynomial functions.

Table 1: DACBench Benchmarks. "Status" compares the current state of each benchmark to the benchmarks originally introduced by Eimer et al. (2021b).

Sigmoid variation and SGD on polynomials provide additional artificial benchmarks for efficient evaluation of DAC algorithms.

Empirical Validation DACBench is a very recent library. As a consequence, it has not yet been used in prior art. Eimer et al. (2021b) focused on providing a unified interface to a variety of benchmarks and analyzed specific properties of these benchmarks based on the behavior of static policies and simple hand-crafted dynamic policies. Here, we describe the first applications of practical DAC methods to these benchmarks, and provide important empirical validation for DACBench.

6. Empirical Case Studies

In this section, we discuss in more detail three successful applications of automated DAC in different areas of AI: evolutionary optimization (Shala et al., 2020, Section 6.1), AI planning (Speck et al., 2021, Section 6.2), and machine learning (Daniel et al., 2016, Section 6.3). The primary purpose of this section is to complement the general, big picture discussions in previous sections with some concrete practical examples of automated DAC. Here, we cover our own work in this area (Shala et al., 2020; Speck et al., 2021), supplemented with a machine learning application (Daniel et al., 2016) for diversity. In these case studies, we also conducted additional experiments to answer the following research questions.

RQ1: Can we reproduce the main results of the original papers using DACBench?
Since it is well known that RL results are hard to reproduce (Henderson et al., 2018), in order to provide a solid foundation for experimental work in the field we believe it to be important to repeat the original experiments, this time using the publicly available re-implementations provided by DACBench (i.e., the CMAStepSize, FastDownward, and SGD-DL benchmarks) and to compare the results obtained to those of the original papers. Beyond insights into the reproducibility of the prior work, this analysis provides empirical validation for DACBench: This is the first study investigating whether, and to what extent, the benchmarks in DACBench permit reproducing the original results. Further, it is worth noting that the work by Daniel et al. (2016) is closed source, and that this is the first reproduction of their experiments with open-source code.

RQ2: Does DAC outperform static AC in practice?
Theoretically, an optimal DAC policy will be at least as good as an optimal static AC policy. In practice, however, the superiority of DAC is not guaranteed, since practical DAC methods may not be capable of finding an optimal/better DAC policy and/or doing so may require more computational resources than available. To investigate this, for each scenario in our case studies, we compare the anytime performance of the DAC method used to that of static AC baselines: We run SMAC (as a classical AC method, Hutter et al., 2011; Lindauer et al., 2022) and Hydra[10] (as a PIAC method, Xu et al., 2010) on the same problem, and compare the performance of the best dynamic/static policies found at any time during the configuration process. We further include the theoretical upper bounds for

10. Hydra combines SMAC with an algorithm selection method of choice. Since most of the considered benchmarks do not have instance features, we will assume an oracle selecting the best configuration in the portfolio. We treat the maximum size of the portfolio as a case study dependent hyperparameter and detail this choice in the respective experimental setups.

classical AC (SBS $= \min_{\boldsymbol{\theta} \in \Theta} \frac{1}{|I'|} \sum_{i \in I'} c(\boldsymbol{\theta}, i)$) and PIAC (VBS $= \frac{1}{|I'|} \sum_{i \in I'} \min_{\boldsymbol{\theta} \in \Theta} c(\boldsymbol{\theta}, i)$) as reference, to distinguish practical from inherent limitations of static AC.[11]

In what follows, we discuss our three case studies, in each case presenting an introduction to the domain, the problem formulation as an instance of DAC, the solution method, the experimental setup, the results, and a discussion thereof.[12]

6.1 Step Size Adaptation in CMA-ES

The first problem we consider is to dynamically set the step-size parameter of the Covariance Matrix Adaptation Evolution Strategy (CMA-ES, Hansen et al., 2003), an evolutionary algorithm for continuous black box optimization. Each generation g, CMA-ES evaluates the objective value f of λ individuals $x_1^{(g+1)}, ..., x_\lambda^{(g+1)}$ sampled from a non-stationary multivariate Gaussian distribution $\mathcal{N}(\boldsymbol{\mu}^{(g)}, {\sigma^{(g)}}^2 \cdot C^{(g)})$. Then, based on the outcome of these evaluations, CMA-ES heuristically adapts the parameters $\boldsymbol{\mu}$, σ, C of the search distribution aiming to increase the likelihood of generating better individuals next generation. In particular, the step-size parameter σ controls the scale of the search distribution and CMA-ES by default adjusts it using Cumulative Step Length Adaptation (CSA, Hansen & Ostermeier, 1996). Note that variants of CMA-ES, e.g., IPOP-CMA-ES (Auger & Hansen, 2005), exist that also adapt the population size λ dynamically. While these hand-crafted parameter control heuristics are arguably key to CMA-ES's success, they are by no means optimal and implicitly make assumptions about the task distribution. For instance, these heuristics have themselves (hyper)parameters, and it has been shown that tuning these can further improve IPOP-CMA-ES's performance (Liao et al., 2011; López-Ibáñez et al., 2012). In Shala et al. (2020), we investigated the possibility of learning step-size adaptation in a data-driven fashion, optimized for the task distribution at hand, i.e. automated DAC.

Problem Formulation: Below, we briefly detail each of the DAC components:

$\mathcal{A}$, Θ: The target algorithm to configure in this scenario is CMA-ES. As in Shala et al. (2020), we use the *pycma* distribution of CMA-ES. Its interface allows for stepwise execution of CMA-ES. CMA-ES is initialized with a given initial mean $\boldsymbol{\mu}^{(0)}$ and step-size $\sigma^{(0)}$ ($C^{(0)} = \mathbb{1}$). Each generation g, we

1. sample λ individuals $x_1^{(g+1)}, ..., x_\lambda^{(g+1)}$ from $\mathcal{N}(\boldsymbol{\mu}^{(g)}, {\sigma^{(g)}}^2 \cdot C^{(g)})$
2. evaluate the objective function values $f(x_1^{(g+1)}), ..., f(x_\lambda^{(g+1)})$ of these individuals
3. adapt the distribution parameters $\boldsymbol{\mu}^{(g+1)}$, $\sigma^{(g+1)}$, $C^{(g+1)}$ for the next generation.

In this final step, the mean $\boldsymbol{\mu}$ and covariance C are adapted as usual in CMA-ES, while the step-size σ is to be reconfigured dynamically in the range $\Theta = \mathbb{R}^+$.

11. Note that the acronyms SBS (single best solver) and VBS (virtual best solver) stem from the algorithm selection literature. More details on how these theoretical bounds were determined can be found in the experimental setup of the respective case studies.
12. Code for reproducing these experiments is publicly available: `https://github.com/automl/2022_JAIR_DAC_experiments`

$\mathcal{D}, I$**:** Instances correspond to tuples consisting of a black box function f and an initial search distribution. Here, the latter is isotropic and defined by an initial mean $m^{(0)}$ and step-size $\sigma^{(0)}$.

Π : The policies are constrained to be functions of a specific observable state composed of: (i) the current step-size value $\sigma^{(g)}$; (ii) the current cumulative path length $p_\sigma^{(g)}$ (Hansen & Ostermeier, 1996); (iii) the history of changes in objective value, i.e., the normalized differences between successive objective values, from h previous iterations; and (iv) the history of step-sizes from h previous iterations.

c : The cost metric used is "the likelihood of outperforming CSA". Assuming we perform two runs of CMA-ES, one using π, the other CSA, it measures how likely the latter is to obtain a better final solution than the former. We estimate this probability based on pairwise comparisons of $n = 25$ runs varying only the random seed of CMA-ES, i.e.,

$$c(\pi, i) = \frac{\sum_j^n \sum_k^n \mathbb{1}_{\pi_j < CSA_k}}{n^2}$$

where $\mathbb{1}_{\pi_j < CSA_k}$ is the function indicating that our policy resulted in a lower final objective value than the baseline using CSA, when comparing runs j and k. Note that a benefit of this cost metric is that it is easy to interpret, both conceptually and in terms of statistical significance. As explained in more detail in the original publication (Shala et al., 2020, Appendix C), it has a direct correspondence with the Wilcoxon rank sum statistic. For $n = 25$, estimates $c(\pi, i) \geq 0.64$ imply π significantly outperformed CSA (at 95% confidence, one-sided).

Solution Method: In Shala et al. (2020), we proposed to use existing hand-crafted heuristics to warm-start DAC. To this end, we adopted the methodology proposed by Li and Malik (2017) in the context of L2O and used guided policy search (GPS, Levine & Abbeel, 2014), a reinforcement learning method originating from the robotics community. In GPS, a teacher (typically a human) provides some initial sample trajectories that the RL agent first learns to imitate and then iteratively refines without further interaction with the teacher. To learn step-size adaptation policies, in Shala et al. (2020), we used CSA as a teacher and extended GPS with *persistent teaching*, meaning that at each iteration the GPS agent obtains a fixed fraction (the *sampling rate*, a hyperparameter) of its sample trajectories from the teacher, encouraging it to continually learn from CSA. Following Li and Malik, the area under the curve (AUC) was used as a reward signal for GPS, instead of negated cost. Here, the reward at step t is $-\min_{x \in X_t} f(x)$ where $X_t = \{x_i^{(g)} \mid g \leq t\}$ is the set of individuals evaluated up until step t. This reward signal, unlike negated cost, is dense and actively encourages learning policies with good anytime performance.[13]

Experimental Setup In our experiments, we used the DACBench implementation of the CMAStepSize benchmark. Replicating the original setup, we set population size $\lambda = 10$, history length $h = 40$, terminate CMA-ES after 50 generations, and model policies as fully connected feed-forward neural networks having two hidden layers with 50 hidden units each

13. Taking a multi-objective perspective (López-Ibáñez et al., 2012; López-Ibánez & Stützle, 2014), AUC can be seen as measuring the hypervolume using fitness 0 and the maximum budget as a reference point.

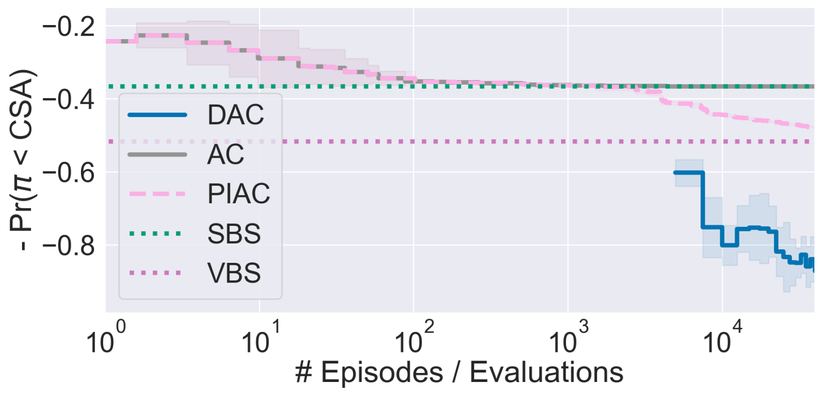


Figure 4: Incumbent performance of DAC (GPS), PIAC (Hydra), and classical AC (SMAC) when determining a step-size configuration policy for CMA-ES. Solid lines depict the mean of five independent configuration runs and the shaded area the standard deviation. SBS depicts the single best configuration and VBS the oracle configuration portfolio across all instances.

and ReLU activations. Note that in Shala et al. (2020), we considered a collection of different scenarios varying in target distribution: (i) single black box function, different initial search distributions; (ii) black box functions of the same type, but different dimensionalities and initial search distributions; and (iii) black box functions of different types and initial search distributions. In our case study here, we only reproduce and discuss the results for the third scenario, as it considers learning policies that generalize across different black box functions. Here, the training setup consists of 100 training instances: 10 different black box functions, with 10 different initial search distributions each. For testing, 12 other black box functions were used with a specific initial search distribution. In both cases, the functions used were taken from the BBOB-2009 competition (Hansen et al., 2009). We perform five independent GPS training runs using the original hyperparameters, each performing a total of 40000 CMA-ES runs and taking 8-10 CPU hours on our system. In our comparison of anytime performance to static AC, the same budget was used for classical AC (SMAC) and PIAC (Hydra). A maximum portfolio size of 10 was used for Hydra. To determine SBS and VBS, we discretized Θ (1000 values equidistant in [0.1, 2.0]) and evaluated $c(\boldsymbol{\theta}, i)$ for all (1000×100) combinations of $\boldsymbol{\theta} \in \Theta$ and $i \in I'$.

Results Figure 4 compares the anytime training performance of DAC (GPS) to that of classical AC (SMAC) and PIAC (Hydra) when learning step-size adaptation. Classical AC and PIAC initially show similar anytime behavior, where the former reaches SBS performance after 1000 evaluations, the latter further improves, but does not reach VBS performance within the given budget of 40000 evaluations. In contrast, DAC (GPS) has a minimum budget of 5000 evaluations, however, the initial incumbent immediately outperforms the VBS and further improves to eventually find a DAC policy that on average outperforms CSA on 87% of the runs on the training setting. Figure 5 shows the likelihood of the five learned policies outperforming CSA on the 12 unseen test functions. Here, for

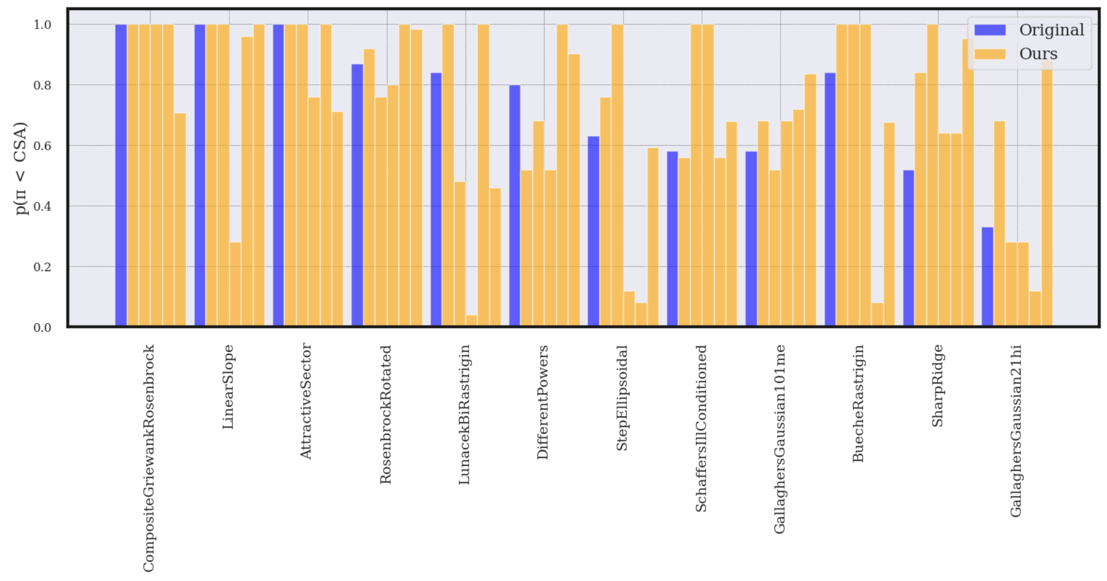


Figure 5: Likelihood of the policies learned by GPS (for five runs) outperforming CSA on 12 unseen test functions. The reported values from (Shala et al., 2020) are shown in blue, whereas the results for the five learned policies are shown in yellow, in a consistent order.

each of five individual seeds, we observe that the learned policies significantly (p($\pi <$ CSA) ≥ 0.64, $\alpha = 0.05$) outperformed CSA on $10, 9, 10, 8, 8$ of the 12 unseen test functions, while being significantly outperformed on $0, 1, 0, 3, 3$, respectively. In comparison to the original, the learned policies performed similarly when averaging costs across all test functions/policies (0.74 vs 0.75 originally). However, it is worth noting that the average performance of the individual policies and the performance profile across the test functions varies more strongly.

Discussion On a high level, we could reproduce our results from Shala et al. (2020), showing that the learned policies for step-size adaptation can outperform CSA on functions not seen during training. Since the DACBench implementation, to the best of our knowledge, exactly replicates the original setup, we assume the observed differences to be a consequence of variability across training runs. This is supported by our observation that the five different runs of GPS (varying only in random seed) resulted in policies whose test cost ranged from -0.83 to -0.65 (vs. -0.75 originally). Our analysis of the anytime performance revealed another weakness of the approach: Its relatively high up-front cost. It is worth noting that this cost includes the teacher runs (25×100 runs using CSA) we performed to warm-start GPS. Nonetheless, since GPS maintains an independent controller per instance, its computational cost will generally scale linearly with the number of training instances. Further, it is difficult to predict in advance how many training instances and runs per training instance suffice. In comparison, the static approaches in our comparison follow a more incremental approach resulting in a better anytime performance. That being said, the best static policy did not significantly outperform CSA. As such, independent of the specific approaches, our results provide further evidence of the importance of dynamic step-size adaptation, showing that DAC policies (learned, but also CSA) are competitive with and/or outperform their static counterparts, even on relatively short CMA-ES runs. Finally, we note that CSA itself has various configurable parameters (e.g., the initial step-size, backward time horizon, and dampening parameters) and that these were not optimized for our baseline in Figure 5. While tuning these offline is not standard practice, doing so may result in a stronger baseline (Liao et al., 2011; López-Ibáñez et al., 2012), potentially even outperforming our learned policy. In fact, this could itself be seen as an instance of "DAC by static AC" constrained to the parametric CSA policy space.

6.2 Heuristic Selection in FastDownward

Heuristic search is one of the most widely used and successful approaches to AI planning. This type of search makes use of heuristics to estimate the distance to some desired goal state as a cheap proxy of having to directly evaluate the true distance. Over decades of research, many different heuristics have been developed for a variety of problem domains. Since no single heuristic works best on all problem instances, the AI planning community has made use of meta-algorithmic approaches such as algorithm selection, algorithm scheduling and algorithm configuration (Helmert et al., 2011; Seipp et al., 2014; Fawcett et al., 2014; Seipp et al., 2015; Sievers et al., 2019). However, one limiting factor of these approaches is that they do not take the internal dynamics of the planning system into account and only adapt to a set of problem instances (per-distribution) or individual problem instances (per-instance). Gomoluch et al. (2019, 2020) first investigated automated dynamic algorithm configuration (DAC) in AI planning, in the context of switching between different search strategies. However, in our case study below, we will focus on DAC for heuristic selection problem (Speck et al., 2021). It has been shown that using hand-crafted policies to switch between heuristics to adapt to changing conditions can greatly improve performance (Richter & Helmert, 2009; Röger & Helmert, 2010). Speck et al. (2021) proposed to use reinforcement learning to automatically determine a policy that selects at each individual planning step which heuristic to follow, out of a set of heuristics sharing their progress. That work showed that DAC is in theory able to outperform prior meta-algorithmic approaches and empirically validated this by outperforming the theoretical best algorithm selector (a.k.a. virtual best solver) on multiple domains.

Problem Formulation Below, we briefly detail each of the DAC components:

$\mathcal{A}$, Θ: The target algorithm to configure in this scenario is the popular FastDownward Planner (Helmert, 2006). To make it stepwise executable, and to allow communication with a dynamic configuration policy, Speck et al. (2021) proposed to set up a socket communication such that DAC can change heuristics after each node expansion. The configuration space consists of four heuristics[14] (i.e., a single categorical parameter), commonly used in satisficing planning: (i) the FF heuristic h_{ff} (Hoffmann & Nebel,

14. In an additional experiment, Speck et al. (2021) showed that even with an increased action space, including the landmark-count heuristic (Richter et al., 2008), DAC was still capable of learning better policies than the considered baselines. Here, we limit ourselves to the original configuration space which only includes four heuristics.

2001), (ii) the causal graph heuristic h_{cg} (Helmert, 2004), (iii) the context-enhanced additive heuristic h_{cea} (Helmert & Geffner, 2008), and (iv) the additive heuristic h_{add} (Bonet & Geffner, 2001). The planning system is terminated when a solution is found. Since some runs may fail to find a solution, Speck et al. (2021) also limited the maximal run length. During the configuration phase (training of the RL agent) an individual solution attempt can run for at most 7500 steps. During evaluation, this conservative step-limit of 7500 steps is removed and instead a maximum of five minutes running time is used.

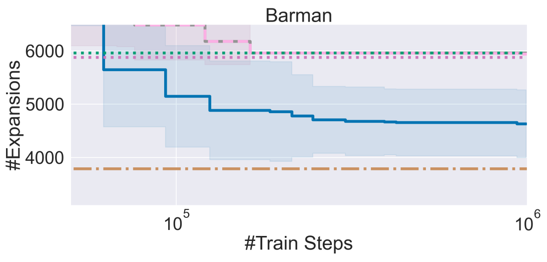
$\mathcal{D}, I$: The target problems consist of 100 training and 100 disjoint test problem instances taken from each of six different domains from the international planning competition (IPC). The instances, however, were not taken from a particular round of the IPC as some domains only contain few instances. Instead, Speck et al. (2021) used instance generators to generate instances that resemble those of the IPC tracks.

Π: The policies are constrained to be a function of a specified observable state. The observable state consists of simple statistics about the heuristics in the configuration space. Specifically, for every heuristic h, it contains the (i) maximum h value; (ii) minimum h value; (iii) average h value; (iv) variance of h over all possible next states; (v) number of possible next states as determined by h; and (vi) current expansion step t. In order to encode progress towards solving a problem instance, Speck et al. (2021) did not use these state features as is, but rather their change w.r.t. the previous step (i.e., the difference between consecutive observations).

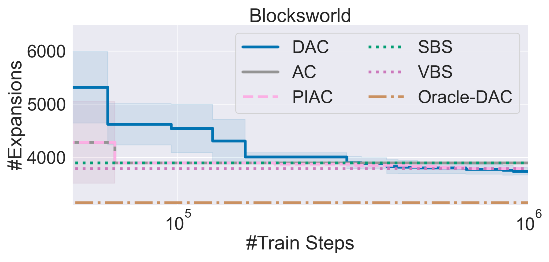
c: The considered cost metric is the total number of node expansions, i.e., the number of planning steps until a solution is found. The decomposed cost metric is +1 for every step. Thus, configuration policies that minimize the average number of planning steps are preferential. Note that, given the termination criterion of $\mathcal{A}$, the maximal cost at configuration time is 7500, corresponding to not finding a solution in time. During evaluation, coverage is analyzed instead, i.e., the number of instances solved within the five minute budget.

Solution Method The proposed solution approach by Speck et al. (2021) uses a small double deep Q-network (DDQN, van Hasselt et al., 2016) to learn a dynamic configuration policy via reinforcement learning. In our experiments, we use the original reinforcement learning code with the exact same hyperparameters as provided by the original authors. Since DACBench offers a standard RL interface (see Section 5), the original RL code could be reused without modification.

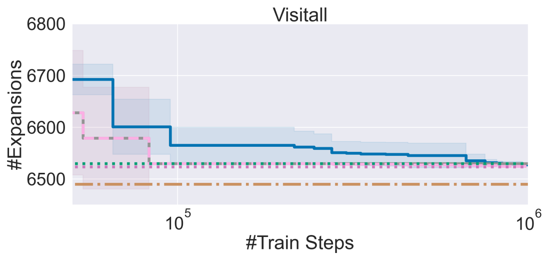
Experimental Setup In our experiments, we make use of the implementation of the interface as provided via DACBench (FastDownward benchmark). Following Speck et al. (2021), we learn a separate policy for each domain, however, to reduce the computational cost, we limit ourselves to a representative set of three out of six domains. Following Speck et al. (2021), we perform five independent training runs for each domain. In each training run, an RL agent experiences 10^6 steps of the planning system, taking 8-12 hours on our system. Since $|\Theta| = 4$, SBS and VBS could be determined exactly for each domain by evaluating $c(\boldsymbol{\theta}, i)$ for all (4×100) combinations. For Hydra, we used a maximum portfolio size of three which is sufficient to cover the optimal portfolio.



(a) barman domain



(b) blocksworld domain



(c) visitall domain

Figure 6: Incumbent performance of DAC (DDQN), PIAC (Hydra), and classical AC (SMAC) when determining a heuristic selection policy for FastDownward on (a) the barman, (b) blocksworld, and (c) visitall domains. Solid lines depict the mean of five independent configuration runs and the shaded area the standard deviation. SBS depicts the single best configuration and VBS the oracle configuration portfolio across all instances. Oracle-DAC is the oracle portfolio of the policies learned by each DAC (DDQN) training run, i.e., out of the five different learned policies this oracle selects the best performing policy for each individual instance. Thus, this provides a pessimistic performance estimate of an optimal dynamic configuration policy.

Results Figure 6 compares the anytime performance of DAC (DDQN) to that of classical AC (SMAC) and PIAC (Hydra) for all three domains. On the barman domain, DAC finds policies that on average clearly outperform the best static baseline in less than 10% of the total budget. On the blocksworld domain, DAC almost needs the full budget, but eventually finds policies that marginally outperform the VBS. The visitall domain is even slightly harder and DAC (DDQN) does not confidently find policies outperforming the static baselines within the limited budget. For lower budgets, classical AC and PIAC obtain clearly better policies on visitall / blocksworld, and PIAC eventually approaches VBS performance on both domains. Table 2 compares the coverage results for all learned policies on the test problem instances with a static baseline. Here, we find that our learned policies generalize well to the test scenarios and achieve similar coverage as reported in the original paper. In the barman domain, DAC policies dominate, and while we achieve a slightly lower coverage on this domain than originally, this can largely be attributed to an individual training run of ours performing worse than the others, with the individual coverages 85.00, 88.32, 67.00, 84.00, and 84.00. In the other two domains, we achieve slightly higher coverages than originally, and the DAC policies perform similarly well as the best static policies.

Algorithm	DAC POLICY						SINGLE HEURISTIC				AS ORACLE
Domain (# Inst.)	RL					RL†	h_{ff}	h_{cg}	h_{cea}	h_{add}	SINGLE h
	Run#1	Run#2	Run#3	Run#4	Run#5						
BARMAN (100)	81.7					84.4	66.0	17.0	18.0	18.0	67.0
	85.0	88.3	67.0	84.0	84.0						
BLOCKSWORLD (100)	93.6					92.9	75.0	60.0	92.0	92.0	93.0
	95.0	95.0	91.0	94.0	93.0						
VISITALL (100)	58.6					56.9	37.0	60.0	60.0	60.0	60.0
	58.1	56.1	57.8	60.0	61.0						
SUM (300)	233.9					234.2	178.0	137.0	170.0	170.0	220.0

Table 2: Number of solved *unseen* test problem instances averaged over five independently repeated training runs. Column RL provides the results of our experiment, with the results of the individual runs given in a smaller font, whereas RL† contains the original coverage values as reported by Speck et al. (2021). All h_i columns contain the number of solved problem instances when only using the specific heuristic. AS ORACLE reports the coverage results of the theoretically best algorithm selector.

Discussion Our results confirm the results of Speck et al. (2021) where the DAC policies (i) obtain slightly lower coverage than the single best heuristic in the VISITALL domain, (ii) outperform the single best heuristic and are close in performance to the theoretical best algorithm selector on the BLOCKSWORLD domain and (iii) provide the best coverage by far in the BARMAN domain. Most notably, on average, the learned DAC policies are capable of solving more problem instances than the theoretical best algorithm selector, which already provides a significant improvement over using the single best heuristic. Our analysis of the approach's anytime performance also revealed that when less time is available, static AC approaches, in particular PIAC (Hydra), outperform DAC (DDQN) on two of the three domains. However, on the remaining domain (BARMAN), superior dynamic policies are easily found. It is worth noting that on all three domains, oracle-DAC is clearly superior, suggesting the potential to further improve performance by using better DAC methods and/or more informative state features.

6.3 Learning Rate Control in Neural Network Training

Daniel et al. (2016) investigated meta-learning a controller for the learning rate hyperparameter η in Stochastic Gradient Descent (SGD) style neural network optimizers. SGD is the method of choice for optimizing the parameters $\boldsymbol{w}$ of deep neural networks, i.e., solve $\arg\min_{\boldsymbol{w}} L(\boldsymbol{w}, D)$, where L is some differentiable measure of loss on the training data D. In deep learning, it is common to have millions of parameters. To scale up to such extremely high-dimensional $\boldsymbol{w}$, SGD exploits the fact that $\nabla_{\boldsymbol{w}} L(\boldsymbol{w}, D) = \frac{\partial L(\boldsymbol{w}, D)}{\partial \boldsymbol{w}}$ can be computed exactly, and updates $\boldsymbol{w}$ in the opposite direction of the gradient. As datasets in deep learning are huge, computing the "full batch" gradient is typically too expensive. Instead, SGD computes the gradient at every optimization step for a different randomly selected "mini-batch" $B \subset D$. While this gradient is an unbiased estimate of the actual gradient, i.e., $\mathbb{E}[\nabla_{\boldsymbol{w}} L(\boldsymbol{w}, B)] = \nabla_{\boldsymbol{w}} L(w, D)$, variance can cause gradients to occasionally point in the wrong direction. Furthermore, gradients only provide local information and do not tell us how far we can move without overshooting. Moreover, the optimal step sizes per dimension may vary strongly, a problem known as ill-conditioning. Over the last decade a variety of different variants of SGD, e.g., Momentum (Jacobs, 1988), RMSprop (Tieleman et al., 2012), and Adam (Kingma & Ba, 2015), have been proposed that aim to address these and other issues. However, despite their popularity, modern SGD variants are still sensitive to their hyperparameter settings. In particular, they still have a global/initial learning rate η, that uniformly scales the step taken in each dimension, and that must typically be optimized for the problem at hand (Bengio, 2012). When setting η too low, optimization is slow, while too high η might even lead to divergence. To the best of our knowledge, Daniel et al. (2016) was the first work that explored replacing η by a meta-learned controller, producing more robust SGD methods. Xu et al. (2019) followed up on this idea, and most recently Almeida et al. (2021) considered meta-learning the control of learning rate *and* various other hyperparameters (e.g., weight-decay and gradient clipping).

Problem Formulation The meta-learning approach by Daniel et al. (2016) is readily seen as automated DAC. Below, we briefly detail each of the DAC components:

$\mathcal{A}$, Θ: Daniel et al. (2016) present a general method for dynamically configuring the learning rate $\eta_t \in \mathbb{R}^+$ at every optimization step of SGD. In their experiments, they do this for two SGD variants: RMSprop and Momentum.[15] Note that in the first optimization step, a fixed learning rate η_0 is used.

$\mathcal{D}, I$: Instances correspond to neural network optimization problems, and are represented by the quadruple $\langle D, L, k, \xi \rangle$, where

- D is the data we want to fit the neural network to. In their experiments, Daniel et al. (2016) consider image classification, using examples from the MNIST and CIFAR-10 datasets.
- L is the differentiable loss function to be minimized. Daniel et al. used cross-entropy loss, i.e., the negative log-likelihood of the data D under the model with parameters w, where this model can be any parametric model. Daniel et al. (2016) used small convolutional neural networks (CNNs).
- k is the cutoff: SGD is terminated after k optimization steps.
- ξ is the seed of the pseudo-random number generator used for random neural network initialisation and mini-batch sampling.

Π : Daniel et al. (2016) considered dynamic configuration policies that are a log-linear function $\pi_{\boldsymbol{\lambda}}(\boldsymbol{\phi}) = \exp(\lambda_0 + \sum_{j=1}^{4} \lambda_j \phi_j)$ of four expert features $\boldsymbol{\phi}$ that in turn depend on the previous learning rate η_{t-1} and the current loss/gradients for each data point. See the original paper for a detailed description of $\boldsymbol{\phi}$.

c : Daniel et al. (2016) aim to control the learning rate η as to maximally reduce the training loss. Specifically, the cost of a run is quantified as $\min(\frac{1}{k-1}\log(\frac{E_k}{E_1}), 0)$, where E_t is the full batch training loss after t optimization steps. Note that we handle divergence cases by setting the costs of runs that fail to reduce the training loss to 0.

15. In the original paper, Momentum was simply referred to as "SGD".

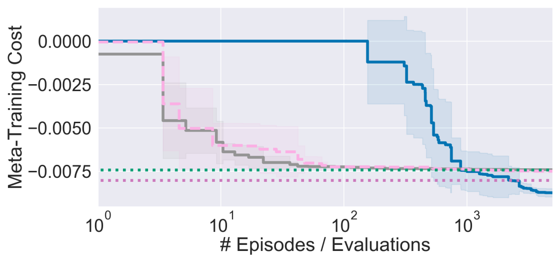
	$\mathcal{A}$	D	L	k	ξ
meta-training	RMSprop	MNIST-small	c-p-c-p-c-r-fc-s (varying # filters)	300-1000	varying
	Momentum	MNIST-small	c-p-c-p-c-r-fc-s (varying # filters)	300-1000	varying
meta-testing	RMSprop	MNIST	c-p-c-p-c-r-fc-s (20-50-200 filters)	2000	fixed
	Momentum	MNIST	c-p-c-p-c-r-fc-s (20-50-200 filters)	2000	fixed
	RMSprop	CIFAR-10	c-p-r-c-r-p-c-r-p-c-r-fc-s (32-32-64-64 filters)	6000	fixed
	Momentum	CIFAR-10	c-p-r-c-r-p-c-r-p-c-r-fc-s (32-32-64-64 filters)	12000	fixed

Table 3: A summary of the six different DAC setups used in (Daniel et al., 2016). During meta-training, 100 target problem instances are considered, generated by randomly varying D (dataset), L (loss), k (cutoff), and ξ (seed). The meta-testing setups consider a single instance. MNIST-small: To avoid bias towards specific training examples, a randomly varied subset of 6K-30K of the 60K MNIST training examples is used during meta-training. The losses L differ only in the predictive model. All use CNNs, but a different layered architecture (c: same convolution with 3x3 filter, r: ReLU, fc: fully connected, s: softmax). To avoid bias towards specific architectures, the number of filters used is varied randomly in meta-training (in ranges [2-10]-[5-25]-[50-250]).

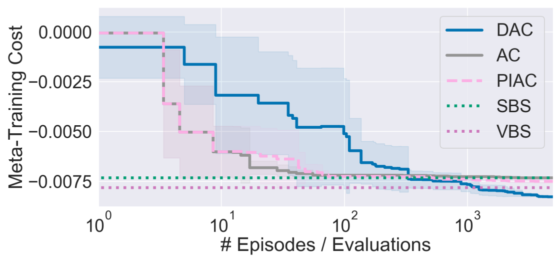
Solution Method Daniel et al. (2016) solved this DAC problem by directly optimizing the policy parameters $\boldsymbol{\lambda}$ using the Relative Entropy Policy Search (REPS) policy gradient method. In our experiments, we will also optimize $\boldsymbol{\lambda}$ directly, but instead use Sequential Model-based Algorithm Configuration (SMAC, Hutter et al., 2011). Note that we follow a *DAC by static AC*, instead of a *DAC by RL* approach (see Section 4.2). This decision was motivated by the fact that Daniel et al. (2016) provide too little details about the method and its implementation, to allow us to confidently reproduce the original meta-training pipeline. On the other hand, SMAC is a popular open source (Lindauer et al., 2022) tool for Bayesian optimization that we conjecture to be suitable to reliably and globally optimize $\boldsymbol{\lambda}$ within a reasonable time frame.

Experimental Setup In our experiments, we used the DACBench implementation of the DAC scenario described above (SGD-DL). Apart from using SMAC instead of REPS, we aimed to maximally replicate the setup used in the original paper. Note that Daniel et al. (2016) actually considered six slightly different scenarios: Two for learning the η-controller for RMSprop/Momentum, resp., and two for testing each of the meta-learned controllers on MNIST/CIFAR-10, resp. The differences between these setups are summarized in Table 3. As we did not have access to the original code, replication was restricted by the details disclosed in the original paper.[16] The remaining design choices were mostly made heuristically. Some had to be optimized to obtain similar baseline behavior. Here, we found the use of a sufficiently large mini-batch size (64 at meta-training, 512 at meta-testing), and Xavier weight initialisation, to be particularly important. For meta-training the two η-controllers, we used a meta-training set $I' \sim \mathcal{D}$ of 100 instances and the default parameter settings of SMAC, and optimized $\boldsymbol{\lambda} \in [-10, 10]^5$, using a symmetric log-scale with linear threshold 10^{-6}, for 5000 inner training runs. Each SMAC run took less than 2 CPU-days on our system. To assess meta-training variability, we performed five such runs in parallel,

16. We also contacted Chris Daniel, the first author, but he did not have access to the proprietary code anymore, either, and was thus not able to help us replicate the original setup.



(a) Meta-Training RMSprop



(b) Meta-Training Momentum

Figure 7: Incumbent performance of DAC (SMAC configuring a parametric DAC policy), PIAC, and classical AC when meta-learning learning rate configuration for RMSprop (left) and Momentum (right). Solid lines depict the mean of five independent meta-learning runs and the shaded area the standard deviation. SBS depicts the single best configuration and VBS the oracle configuration selection portfolio across all instances.

selecting the configuration with the highest meta-training performance for meta-testing. For Hydra, we used the same parameters as SMAC, and a maximum portfolio size of 10. Finally, to determine SBS and VBS, we discretized Θ (1000 values, log-scale in $[10^{-5}, 10^{0}]$) and evaluated $c(\boldsymbol{\theta}, i)$ for all (1000×100) combinations.

Results Figure 7 compares the anytime performance of DAC (SMAC) to that of PIAC (Hydra) and classical AC (SMAC) for RMSprop (left) and Momentum (right). In both cases, DAC's initial performance is worse than its static counterparts. This difference in relative performance is most blatant for RMSprop, where DAC takes over 100 evaluations to find a non-diverging policy (i.e., with negative average cost), while classical AC achieves near SBS performance in that time. PIAC (Hydra) only marginally improves upon classical AC (SMAC) and SBS, and does not attain VBS performance. Despite the slow start, all DAC runs eventually outperform all classical AC and PIAC runs, ultimately attaining a policy that reduces training loss 0.71% (RMSprop) and 0.46% (Momentum) more per step than the VBS on average ($\sim$ 58% and 35% after 650 steps). Figure 8 shows the full batch training loss $L(w_t, D)$ at each optimization step using the meta-learned η-controller that performed best in meta-training, and various static baselines, in each of the four meta-test setups. Overall, the training curves for our baselines look similar to the original, both in terms of absolute and relative performance. An exception are high learning rates. For RMSprop, our curves look quite different, but are similarly chaotic. For momentum, the highest learning rate performs best for us, while the original diverges. On MNIST, both meta-learned controllers (π) clearly outperform the best static baseline, even though the cutoff k is two times higher than the highest cutoff considered during meta-training. This result is similar to that of the original paper, but our learned controller arguably even does better. On CIFAR-10, the meta-learned controller (π) performs similar to (RMSprop), or better than (Momentum) the best baseline in the first 1000 update steps, but fails to achieve the best final performance. Here, our results differ from the original, where the final performance was similar (RMSprop) or better (Momentum) than the best static baseline.

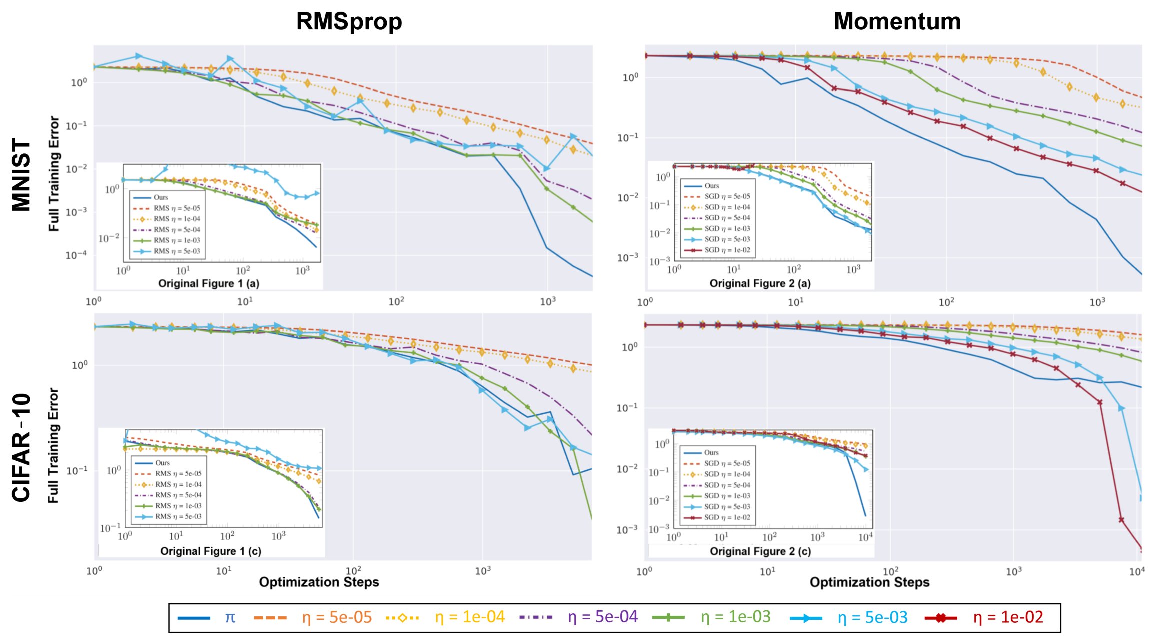


Figure 8: Comparison of learning curves for RMSprop/Momentum using the meta-learned η-controllers (π) vs. several static baselines, on MNIST/CIFAR-10. Each dataset/optimizer combination appears in its own sub-figure. For ease of comparison, the corresponding figure in the original paper is shown in the bottom left corner of each sub-figure.

Discussion The "flavour of AC" prevailing on these scenarios depends on the budget available: For sufficiently large budgets (> 1000 evaluations), DAC confidently outperforms static AC. However, DAC is clearly outperformed by classical AC for smaller budgets. While the best DAC policies are better, arbitrary static policies tend to outperform arbitrary DAC policies for this scenario, e.g., the vast majority of DAC policies diverge for RMSprop. Nonetheless, the poor relative initial performance of DAC is not inherent, and could, e.g., be addressed by using a different initial design that prioritizes static policies ($\pi_{\boldsymbol{\lambda}} : \lambda_k = 0, \forall\, k > 0$). Also note that we cannot compare our meta-training results to those obtained by REPS, since Daniel et al. (2016) did not analyze meta-training. Our meta-testing results, however, validate that the SGD-DL benchmark considers a highly similar setup and that it can be used to learn controllers that perform similarly well as in the original paper. On the other hand, we also observed differences that are unlikely explained by random noise alone. For example, momentum seems to prefer higher learning rates in our experiments, and our meta-learned controller does not transfer as well to higher cutoffs on CIFAR-10. Finally, the configurations $\boldsymbol{\lambda}$ we found differ strongly from those reported in the original paper, and using the latter even caused divergence in our experiments. We currently cannot explain those differences, and lacking the original code, further insight can only be gained through trial & error. We emphasize that, in contrast to the original code, our benchmark is publicly available to facilitate future research on DAC.

7. Conclusion

To conclude, we again summarize our main insights and results, and discuss possible future research directions opened up by this work.

7.1 Summary

In this article, we presented the first comprehensive overview of automated Dynamic Algorithm Configuration (DAC), a novel meta-algorithmic framework proposed by Biedenkapp et al. (2020). To this end, we introduced automated DAC as a natural extension of previous research efforts in automated static algorithm configuration and manual DAC. Furthermore, we situated automated DAC in a broader context of AI, discussing how it can be viewed as a form of "semi-automated" programming, as a generalization of existing meta-algorithmic frameworks, and as an automated approach to the design of operator selection and parameter control mechanisms. After formalizing DAC further, we introduced its methodology and showed how prior art can be roughly subdivided in two schools, tackling the problem using *reinforcement learning* and *optimization* methods, respectively. On the empirical side, we presented and extended DACBench, a novel benchmark library for DAC proposed by Eimer et al. (2021b) and showed that DAC can be successfully applied to evolutionary optimization, AI planning, and machine learning. As the first paper, we provided thorough empirical evidence that automated DAC can outperform prior static AC methods. In summary, we found that on all scenarios considered, automated DAC discovered policies that were at least as good as, and typically better than, their static counterparts. Depending on the scenario, this sometimes required less, but usually more (up to $10\times$) computational budget than a state-of-the-art static AC method needed to converge on the same scenario, on average.

7.2 Limitations and Further Research

While these case studies and other previous applications provide a "proof of concept" for automated DAC, we point out that much remains to be done to unlock its full potential, and we hope that this work may serve as a stepping stone for further exploring this promising line of research. In what remains, we will discuss some of the limitations of contemporary work and provide specific directions for future research.

Jointly configuring many parameters While static approaches are capable of jointly configuring hundreds of parameters, the configuration space in contemporary DAC is typically much smaller, often considering only a single parameter. While the configuration space is smaller, the candidate solution space (i.e., the dynamic configuration policy space) grows exponentially with the number of reconfiguration points, in the worst case, and is thus typically drastically larger than static configuration policy spaces. Although modern techniques from reinforcement learning scale much better than ever before, we still know too little about the internal structure of DAC problems to handle this exploding space of possible policies. For example, not much is known regarding interaction effects of parameters in the DAC setting. If there should be only a few interaction effects between parameters as in static AC (Hutter et al., 2014; Wang et al., 2016), learning several independent policies might be a way forward.

Temporally sparse DAC Note that not all parameters can/must be reconfigured at every time step. Also, our results suggest that an initial bias towards static configuration policies could improve the anytime performance of DAC in various scenarios. Mixed static and dynamic configuration, and learning "when to reconfigure" (Biedenkapp et al., 2021) therefore present one opportunity to scaling up DAC. Furthermore, we plan to extend the DAC formalism with *partial* reconfiguration to capture intrinsic temporal conditionalities, e.g., a parameter not being used in some execution steps.

Warm-starting DAC Most prior art derives dynamic configuration policies from scratch, while in many cases good default parameter control mechanisms are known. Beyond strong baselines, these existing policies could also be used to warm-start the automated process. This idea has already been explored by Shala et al. (2020) (see also Section 6.1), but could be extended in various ways. For example, we could learn from an ensemble of teachers to exploit their complementary strengths.

Online DAC Most prior art performs algorithm configuration offline, i.e., the optimal static/dynamic configuration policy is derived in a dedicated configuration phase proceeding use/test time (see Figure 1).[17] However, when using the target algorithm, more information about the target problem distribution and relative performance of candidate policies becomes available, and *online algorithm configuration* approaches (Fitzgerald, 2021) capable of exploiting this information and transferring experience across test instances, continually refining the policy, are an interesting direction of future research.

Better DAC methods Successful DAC requires more than just computational resources. To apply DAC, a practitioner must make many choices that critically affect not just its efficiency, but also its effectiveness. As a consequence, key ingredients for successful DAC are currently (i) target domain expertise, (ii) DAC methodology expertise, and (iii) trial and error. Note that this conflicts with the main objective of automated DAC, i.e., reducing reliance on human effort and expertise. To address this shortcoming, we need better methods. In particular, as discussed in Section 4.3, we believe that there is a need for dedicated dynamic algorithm configuration packages capable of combining the strengths of the contemporary DAC by reinforcement learning and optimization approaches.

Domain-Expert driven DAC From working on static AC for more than a decade, we know that a challenge AC poses to users is to specify the inputs, including questions such as: (i) Which instances will reflect future real-world use cases well? (ii) Which parameters are important and should be configured, and using which domain (upper & lower bounds, etc)? (iii) Which metric will accurately quantify the true desirability of a configuration or policy? This hinders the adoption of such meta-algorithmic approaches in practice. To tackle this problem, we envision a new paradigm which is driven by the domain expert and allows for monitoring of the training and deployment performance, and for live adjustments of training distributions, configuration spaces and performance metrics by the domain expert. Likewise, we would like to enable experts to express priors over the policies they would expect to work well, extending similar work in static AC (Hvarfner et al., 2022). Finally, we would like domain experts to not only be able to steer DAC, but to also gain new and

17. As discussed in Section 2.2.3, we do not regard the majority of previous "online learning approaches" to parameter control as prior art in *automating* DAC.

deeper insights from the automated search process, similar to various existing methods that capture the importance of hyperparameters in static AC (Hutter et al., 2014; Biedenkapp et al., 2017, 2018; van Rijn & Hutter, 2018; Probst et al., 2019).

Extending DAC Benchmarks To stay relevant, these future directions will also have to be reflected in a set of contemporary DAC benchmarks, such as in DACBench, alongside continuing work on further expanding the scope of existing benchmarks. While DACBench covers a range of domains, some like SAT or MIP, which are commonly used in AC, are absent at the moment. Partnering with domain experts could help broaden the scope of DACBench and thus DAC in general. Beyond real-world benchmarks, there is also a need for additional "toy" benchmarks that permit efficient evaluation of DAC methods, something especially crucial in (i) the early stages of developing new methods and (ii) enabling meta-algoritmics to be applied to DAC itself. Finally, prior art, our own work included, rarely compares different DAC methods. To facilitate this, we need more than just benchmarks, we need a library of DAC methods and standard protocols to compare them.

Acknowledgments

All authors acknowledge funding by the Robert Bosch GmbH. Theresa Eimer and Marius Lindauer acknowledge funding by the German Research Foundation (DFG) under LI 2801/4-1. We thank Maximilian Reimer, Rishan Senanayake, Göktuğ Karakaşlı, Nguyen Dang, Diederick Vermetten, Jacob de Nobel and Carolin Benjamins for their contributions to DACBench, and Carola Doerr for the many discussions on related work and problem formulation.

Appendix A. History of Automated Dynamic Algorithm Configuration

As discussed in Section 2.3, the story of automating dynamic algorithm configuration (DAC) did not start when Biedenkapp et al. first introduced the DAC meta-algorithmic framework in 2020. Instead, various works have over the last decade explored automated dynamic algorithm configuration, under different names, in more specific contexts, often in isolation, and pioneering work can be traced at least 20 years back. In the main text, we reference the relevant works and characterize the field both in terms of application areas (Section 2.3) and methodology (Section 4). In what follows we complement this high-level perspective, briefly summarizing and highlighting the contributions of each individual work to the field, in chronological order.[18]

June 2000 Lagoudakis and Littman consider algorithm selection for recursive algorithms. In particular, motivated by the observation that every subproblem may be best solved using a different algorithm, formulate algorithm selection in this context as a sequential decision problem and propose the use of reinforcement learning to derive dynamic algorithm selection policies that generalize across target problem instances. They illustrate this approach on classic problems of order statistic selection and sorting. This is the first work extending a general meta-algorithmic framework to the dynamic setting and following a solution approach that, in many ways, resembles current "DAC by reinforcement learning" practice (see Section 4.1). The work is limited in its focus on algorithm selection ($\sim$ single categorical parameter) and the recursive setting. In particular, their methodology exploits the tree structure of the computational graph and that optimal decisions in different branches are independent.

June 2001 Lagoudakis and Littman apply their aforementioned approach to learn a policy that dynamically selects amongst seven possible branching rules for DPLL, a complete SAT solver. It is worth noting that DPLL is not a recursive algorithm, but that backtracking algorithms produce a similar computational graph.

July 2002 Pettinger and Everson present, in a *single* page extended abstract, what we regard to be the first general instance of automated dynamic algorithm configuration. They describe a study using $Q(\lambda)$ reinforcement learning to determine a policy that jointly selects mutation and crossover operators ($|\Theta| = 20$) in a genetic algorithm, per generation, based on statistics of the current population (not further specified) and show that the learned policy generalizes to a single unseen 40-city TSP problem. Notably, direct follow-up work by Chen et al. (2005), no longer transferred experience across runs and is therefore not considered prior art automating DAC (see Section 2.2.3). Similarly, Sakurai et al. (2010) propose further extensions to this line of work, without discussing generalization or presenting empirical results.

18. *Disclaimer*: This is the result of an extensive literature review we performed in the context of this article, but should by no means be regarded as being complete. Given the wide applicability, the lack of consistent terminology, and the sometimes subtle distinction between manual/automated and static/dynamic AC, we can unfortunately not guarantee completeness of this overview.

Automated DAC winter? Interestingly, following aforementioned pioneering work, we did not find published work exploring automated DAC in a period spanning more than eight years. At the same time, manual DAC and automated static AC were booming. One possible explanation is that the DAC methodology and computational resources available at the time were insufficient to achieve performance competitive with optimized static configurations and/or hand-crafted parameter control policies.

September 2010 Fialho et al. present an analysis of different multi-armed bandit (MAB) approaches for adaptive operator selection in evolutionary algorithms. Note that since none of the MAB approaches transfer experience across runs, we do not regard these as automated DAC. However, these MAB strategies themselves have (hyper)parameters and for the sake of fair comparison, the authors tune these using F-race (Birattari et al., 2002), a methodology that can be regarded as the first account of "DAC by static AC" (see Section 4.2).

January 2012 Battiti and Campigotto present the first extensive investigation using reinforcement learning for parameter adaptation in the context of reactive optimization. In particular, they present case studies using Least Squares Policy Iteration to learn to adapt a single real-valued parameter in a variety of (reactive) stochastic local search methods for the MAX-SAT problem: Prohibition value of Reactive Tabu Search (RTS), noise value of Adaptive Walksat, and the smoothing probability of Reactive Scaling and Probabilistic Smoothing (RSAPS). Unlike earlier work, their study includes learning curves and generalization experiments show that learning takes place for two of the three methods (RTS and RSAPS) and improves upon the default configuration, however, improvements compared to statically configured variants were inconsistent.

May 2012 López-Ibánez and Stützle, in a technical report that was later published as a journal article (López-Ibánez & Stützle, 2014), propose a multi-objective perspective on anytime performance, viewing an algorithm's performance profile as a pareto front. The authors argue for the importance of dynamic parameter adaptation in this context, and suggest the use of static algorithm configurators and the hypervolume metric to automatically design parameter adaptation strategies optimized for anytime performance. They present a case study, using irace (López-Ibáñez et al., 2016) to develop parameter adaptation strategies for four parameters of the MAX-MIN Ant System on the Traveling Salesman Problem. This is the first work explicitly advocating what we call the "DAC by static AC" approach. The case study is limited in that each parameter was controlled independently as a simple function (linear decay, or single switch, 5 (hyper)parameters per parameter) of time.

September 2012 López-Ibáñez et al. use irace to optimize several parameters of IPOP-CMA-ES for anytime performance. While this work could also be regarded as static AC, the fact that the authors explicitly chose parameters that indirectly dynamically control internal parameters of IPOP-CMA-ES (e.g., population size) we view it as another instance of "DAC by static AC".

February 2016a Daniel et al. propose meta-learning a controller for the learning rate hyperparameter in Stochastic Gradient Descent (SGD) style neural network optimizers. Here, they use Relative Entropy Policy Search (REPS) to train a simple log-linear controller using four dynamic expert features. We reproduce the main results of this case study in Section 6.3, but use SMAC (Hutter et al., 2011) to optimize the five parameters of this policy, instead of REPS.

February 2016b Hansen, concurrently, considered a setup similar to Daniel et al.'s. However, they use Deep Q-learning (DQN) to train a neural network controller that performs a line search in *full batch* gradient descent. Their policy network has three outputs, corresponding to trying a twice as small, double as large, or accepting, the current learning rate.

June 2016 Fu, like Hansen (2016), explore using DQN to meta-learn learning rate control in neural networks. The work differs in that it considers *stochastic* gradient descent, without line search, training larger networks (wide ResNets), and a policy network mapping lower level input features (elementary weight statistics) to two outputs, either decreasing the current learning rate with 3%, or resetting to the initial learning rate, to facilitate meta-learning SGDR-like (Loshchilov & Hutter, 2017) warm restart schedules.

July 2016a Adriaensen and Nowé propose to view semi-automated algorithm design as a sequential decision problem, and argue for the potential benefits of solving it using white box methods (exploiting this sequential nature), e.g., reinforcement learning. In this perspective, an algorithm is executed with open design choices (e.g., an unassigned parameter) until the next instruction executed depends on the decisions made, and an agent must choose between possible continuations. This concept is formalized as a non-deterministic Turing machine, where non-deterministic transitions correspond to choice points induced by design decisions left open at design time, and a formal reduction to the MDP is presented. Despite the abstract nature of this work, its focus on algorithm design, the problem formalized is closely related to DAC, and the work provides important motivation for DAC by RL, and DAC in general.

July 2016b Andersson et al. propose a temporally sparse approach to dynamically configure multiple parameters. Here, the DAC policies switch between a limited number of k parameter sets (i.e., configurations) and the dynamic switching policy $S \to \{\theta_1, \ldots, \theta_k\}$ is itself a parametric function whose parameters are jointly optimized with each parameter set ($\sim$ static algorithm configuration). The authors illustrate this approach using it to jointly control six parameters of NSGA-II, considering policies switching once/twice ($k = 2/k = 3$) when a specific tuned threshold of solution quality is reached (tuned, one per switch). The resulting DAC policies have 13/20 (hyper)parameters and were optimized using a custom evolutionary optimization method.

February 2017 Ansótegui et al. first explore a per-instance algorithm configuration (PIAC) approach to DAC. The authors consider the problem of dynamically controlling seven parameters of the dialectic search metaheuristic applied to the MAX-SAT domain. In particular, they use ISAC (Kadioglu et al., 2010) to learn a per-instance configuration policy selecting the best configuration of a DAC policy, based on the static features of the problem instance at hand. Each DAC policy in the portfolio has 84 configurable (hyper)parameters, 12 per parameter, corresponding to the weights (and bias) of a logistic regression map determining the value of each individual parameter, as a linear combination of 11 dynamic state features, followed by a logistic sigmoid output activation. Kadioglu et al. (2017) later followed a similar methodology to learn a reactive restart portfolio, which can be viewed as an instance of automated dynamic algorithm scheduling, where independent algorithm runs differ in random seed only.

June 2017 Fontoura et al. use genetic programming, more specifically grammatical evolution, as a *generation* hyper-heuristic to design a *selection* hyper-heuristic for protein structure prediction. Unlike related work by Sabar et al. (2013), Fontoura et al. apply this approach offline and validate the generality of this selection hyper-heuristic. The learned selection heuristic is shown to be competitive with, but does not outperform, a state-of-the-art cross-domain hyper-heuristic.

July 2019a Sharma et al. use Double Deep Q-learning (DDQN) to learn a dynamic configuration policy controlling the choice of the mutation strategy in the Differential Evolution (DE) algorithm. Here, the policy network learns to select, at each DE generation, and for each individual, between four frequently used mutation strategies, based on 99 dynamic state features.

July 2019b Gomoluch et al. use the classical REINFORCE (Williams, 1992) algorithm, a policy gradient method, to learn policies that switch between discrete choices of forward search algorithms. As part of this work, they discuss the influence the choice of reward function has on RL-based solution approaches for DAC. Their experiments show this RL-based approach to be capable of learning domain specific search policies that improve over simple static baselines.

September 2019 Xu et al. present another study meta-learning learning rate control for SGD using reinforcement learning. Novelty, compared to previous works (Daniel et al., 2016; Hansen, 2016; Fu, 2016), is the use Proximal Policy Optimization (PPO), as opposed to REPS or DQN. Also, they first explore meta-training a *recurrent* learning rate controller network (LSTM), enabling the learned policy to use the full history of state features. Finally, they meta-train to minimize the *validation* loss (as opposed to training loss) to avoid meta-learning controllers that "learn to overfit".

July 2020 Sae-Dan et al. propose an approach similar to Andersson et al. (2016)'s and demonstrate its effectiveness in deriving a dynamic configuration policy for four parameters of an iterated local search metaheuristic, for three different combinatorial optimization problems (TSP, PFSP, QAP). Unlike Andersson et al. (2016), the switching criterion is time, and irace is used to optimize parameter sets and switching points.

August 2020 Biedenkapp et al. establish DAC as a new meta-algorithmic framework, extending classical and per-instance algorithm configuration. In doing so, they aim to consolidate aforementioned efforts and raise the level of generality in pursuit of algorithms similar to those that exist for static AC, and a deeper understanding of the problem itself. Biedenkapp et al. (2020) also propose a concrete solution approach, using *contextual* reinforcement learning, and demonstrate its effectiveness empirically. However, experiments were limited to two toy scenarios, and the paper's focus on "DAC by RL", left alternative solution approaches somewhat underrepresented. Note that we address both limitations of the original work in this journal paper.

September 2020 Shala et al. consider the problem of learning to adapt the global step-size parameter of CMA-ES, a popular continuous black box optimizer. Here, they use guided policy search (GPS, Levine & Abbeel, 2014) and compare the learned policies to CMA-ES's default, hand-crafted step-size adaption heuristic (CSA) on functions taken from the BBOB benchmark (Hansen et al., 2009). Results show that the learned policies generalize to longer runs, and to different function families, as those seen during training. We reproduce the main results of this case study in Section 6.1

October 2020 Gomoluch et al. extend the idea from their previous work (Gomoluch et al., 2019). Instead of discrete choices, they construct a parametrized search algorithm, such that different configurations correspond to (and interpolate between) various different search techniques. They continue to learn to control these parameters dynamically, this time using the cross-entropy black box optimization method (CEM) to optimize the parameters of a simple neural controller. Next to this "DAC by optimization" approach, they also consider a static AC baseline, where CEM is used to optimize the parametrized search algorithm directly (per-distribution). Their experiments show both approaches to be capable of outperforming fixed search techniques, without a clear winner between the two AC variants.

June 2021 Almeida et al. revisit the problem of meta-learning hyperparameter control for training deep neural networks. Like Xu et al. (2019), they train an LSTM using PPO for relative multiplicative control. Unlike previous work (Daniel et al., 2016; Hansen, 2016; Fu, 2016; Xu et al., 2019), they present a case study controlling eight different hyperparameters jointly (including the learning rate). In addition, an adaptive check-pointing strategy is learned that also controls backtracking. Finally, the potential of the approach is shown, training larger models and on a wider range of datasets. Apart from its extended scope and scale, a key insight of this work is the importance of feature normalization. In particular, the work normalizes features as to factor out problem-specific info, to avoid overfitting the meta-training set.

August 2021a Bhatia et al. propose to use DQN to dynamically tune the weighting of the anytime weighted A* method. Their experiments show that DQN can serve as an effective controller, switching between five discrete weights and thereby producing plans of higher solution quality when compared to five typically considered static weights for anytime weighted A*. Their analysis shows that the learned policies are dynamic and make use of most available choices. Further, the authors provide insights into which features were most important for learning the final policies.

August 2021b Speck et al. (first published as preprint in June 2020 and as workshop version in October 2020) make use of RL to learn policies that can switch between different heuristics when searching for satisficing plans. As such, it is a direct follow up to the earlier work by Biedenkapp et al. (2020) and discusses how to design the components needed to be able to use RL as a solution method. Further, this work provides theoretical analysis on the potential gains of using DAC when switching between heuristics. For example, the work proves that, for a particular family of planning problems, optimal DAC policies drastically reduce the number of required planning steps when compared to static counter parts. Finally, their empirical analysis shows that RL based DAC policies are capable of outperforming even the theoretical best algorithm selector. We reproduce the main results of this case study in Section 6.2.

August 2021c Eimer et al. present DACBench, the first benchmark library for DAC providing unified interface to a collection of open source implementations replicating scenarios considered in literature (Daniel et al., 2016; Vermetten et al., 2019; Biedenkapp et al., 2020; Shala et al., 2020; Speck et al., 2021; Biedenkapp et al., 2022). We discuss DACBench in Section 5 and our case studies in Section 6 provide important empirical validation for DACBench.

November 2021 Getzelman and Balaprakash use reinforcement learning (PPO) to learn a stochastic policy to dynamically switch between three pre-existing optimization algorithms (Adam, gradient descent, random search) to solve quadratic programming problem instances. While this work profiles itself as *learning-to-optimize* (L2O, Li & Malik, 2017), it is also readily seen as automated DAC for a single parameter determining the optimizer used in each iteration.

December 2021 Ichnowski et al., like Getzelman and Balaprakash (2021), explore automated DAC, using reinforcement learning, in the context of quadratic programming. They use Twin Delayed DDPG (TD3) to learn a policy controlling the step-size parameter (vector) ρ of the Alternating Direction Method of Multipliers (ADMM).

July 2022 Biedenkapp et al. present a new benchmark (extending DACBench), where one is to dynamically configure the mutation rate of a (1+1) random local search algorithm for the LeadingOnes problem. This is a particularly interesting setting as the exact runtime distribution is very well understood (Doerr, 2019) and optimal dynamic configuration policies can be derived efficiently for various different problem sizes and configuration spaces. Experiments show that DDQN can reliably derive near-optimal policies for smaller problems/configuration spaces, but fails to scale up.

These are all the works we are aware of being published at the time of submitting this manuscript (May 2022). Despite this prior art, many open research challenges still remain (see Section 7.2) and we hope this article may provide a standard reference and a solid foundation for future research in this area.

Appendix B. Problem-Theoretical Perspective on DAC

In this Appendix, we present a theoretical motivation of why DAC (Definition 3) is a problem worth studying. In doing so, we provide grounding for many of the higher-level discussions in the main text. Since this kind of analysis is hardly standard, we start by introducing some fundamental concepts (Section B.1), then discuss the main results (Section B.2), and end with the formal justification (Section B.3).

B.1 Fundamental Concepts

B.1.1 Computational Problems

In this work, we formalized dynamic algorithm configuration (DAC) and related computational problems as follows:

> **Definition 4: Computational Problem**
>
> In a computational problem $(\mathcal{X}, \mathcal{R})$, given any input $x \in X$, we are to compute an output y satisfying $(x, y) \in \mathcal{R}$.

Conceptually, each $(x, y) \in \mathcal{R}$ represents a problem instance x and a solution y thereof. Note that instances may have more than one admissible solution, or even none at all. We will also use $R(x) = \{y | (x, y) \in R\}$ to denote the solution set for x. All problem definitions in this paper are structured syntactically as "Given x find a $y \in R(x)$".

Digression on Problem Classes: Problems of this form are also known as "search problems". To avoid confusion, problem *classes* group problems (e.g., DAC), not problem instances (e.g., DAC scenarios), i.e., when viewing DAC as a search problem (as in Definition 3), $\mathcal{X}$ would correspond to the set of all possible DAC scenarios and $\mathcal{R}(x)$ the set of optimal policies for some DAC scenario x. The choice to restrict ourselves to search problems was a trade-off between (i) theoretical convenience / simplicity and (ii) expressiveness, i.e., alternative formulations exist that better model many of the problems we consider:

Optimization: Can express that not every solution is equally good. For instance, in this work we use (i) "find x satisfying $x \in \arg\max_x f(x)$" rather than (ii) "find x maximizing $f(x)$". Note that this difference, while subtle, is important for problem theory: For instance, if $\arg\max_x f(x) = \emptyset$, in (i) we should return that no solution exists, while in (ii) we should return an as good as possible solution.

Distributional: Can express that all inputs are *not* equally likely, by modelling inputs as a distribution $\mathcal{D}$ rather than a set $\mathcal{X}$. Note that while we do not use distributional problems on the meta-level, we do use them as target problems.

However, for these problem classes, standard definitions for theoretical concepts such as *reducibility* do not exist and any satisfactory definition would significantly complicate the reduction proofs in this appendix.

B.1.2 Reducibility

Problem formalization enables formal reasoning about the relationship between problems. In this appendix, we focus on a specific kind of relationship: Reducibility, as defined by Papadimitriou (1994, p. 506):

Definition 5: Many-one Reducibility (m-reducibility)

Let (X, R) and (X', R') be two computational problems, with $R : X \times Y$ and $R' : X' \times Y'$. We say that (X, R) is many-one reducible to (X', R'), or also m-reducible, which we denote $(X, R) \leq_m (X', R')$, if and only if computable functions

`formulate`: $X \to X'$

`interpret`: $X \times Y' \to Y$

exist such that $\forall\ x \in X$ holds:

1. $(\texttt{formulate}(x), y') \in R' \implies (x, \texttt{interpret}(x, y')) \in R$
2. $R'(\texttt{formulate}(x)) = \emptyset \implies R(x) = \emptyset$

Conceptually, (1) all solutions of the reformulated problem instance can be interpreted as a solution to the original problem instance, and (2) if the reformulated problem instance does not have any solutions, the original problem instance should neither.

When a problem R is m-reducible to another R', R can be *solved by reduction* to R', i.e., given an algorithm a' for R', $a(x) = \texttt{interpret}(x, a'(\texttt{formulate}(x)))$ is an algorithm for R. It is worth noting that the existence of a many-one reduction does not necessarily render R irrelevant: Solving R by reduction to R' may inherently increase computational *complexity* since (i) the reduction itself (i.e., `formulate`, `interpret`) may be costly (ii) the reduction may abstract relevant info, making the reduced problem harder to solve. The practical relevance of a reduction is further limited by the performance of known algorithms for R'.

Note that sometimes, even though one problem is not generally reducible to another, a special case is. We define this notion as

Definition 6: Conditional Reducibility

Let (X, R) and (X', R') be two computational problems, with $R : X \times Y$ and $R' : X' \times Y'$. We say that (X, R) is *conditionally* many-one reducible to (X', R') under *preconditions c*, which we denote $(X, R) \leq_m^c (X', R')$, if and only if $(\{x \in X \mid c(x)\}, R) \leq_m (X', R')$, where c is a Boolean function on X.

The practical relevance of a conditional reduction additionally depends on how commonly its preconditions are satisfied. It is worth noting that many-one reducibility is *transitive*, while conditional reducibility is not. However, the following holds:

$$(X, R) \leq_m^c (X', R') \quad \wedge \quad (X', R') \leq_m (X'', R'') \quad \implies \quad (X, R) \leq_m^c (X'', R'')$$

B.2 Reducibility Results

Figure 9 shows an overview of the reducibility relationships between DAC and all the other computational problems we discussed in the main text.

Reducibility to DAC: We observe that all problems in meta-algorithmics, discussed in Section 2.2.2, can be shown to be generally reducible to DAC. This suggests that research towards solving DAC in general, will indirectly find applications in solving many of these problems. Note that the *conditional* reduction from algorithm design corresponds to the "DAC powered PbO", discussed in Section 2.2.1, despite being conditional (i.e., not general), presents a highly practical approach to automating algorithm design.

Reducibility from DAC: We observe that DAC is generally reducible to "algorithm design" (as in Definition 7). However, this reduction is not practical since no general solvers for this problem are known. DAC is also generally reducible to "noisy black box optimization". While this reduction is more practical (see Section 4.2), a lot of information is lost in the process. Beyond these two problems, DAC is conditionally reducible. While many of these conditional reductions give rise to practical solution approaches (see Section 4), they are nonetheless limited both in terms of generality and the information they can exploit.

Conclusion: While a general DAC solver would allow us to solve many well-known computational problems, no such solver exists to date, and is therefore a research direction worth exploring. Note that various existing solvers can solve special cases of DAC, suggesting that another line of research would be to identify further special cases that can be solved more efficiently.

B.3 Reducibility Proofs

In this subsection, we formalize the reducibility relationships between DAC (Definition 3) and all problems discussed in the main text. Here, we first formally define each problem and then show reducibility by describing one possible reduction, i.e., defining `formulate` and `interpret` functions. Finally, we present a formal argument (*proof sketch*) for the correctness of each reduction.[19]

B.3.1 Algorithm Configuration

All discussed algorithm configuration variants were already defined in the main text:

- classical / per-distribution algorithm configuration (AC, Definition 1)
- per-instance algorithm configuration (PIAC, Definition 2)
- dynamic algorithm configuration (DAC, Definition 3)

In what follows, we formalize their relation.

19. For brevity, we generally prove (1) in Definition 5, but not corner case (2).

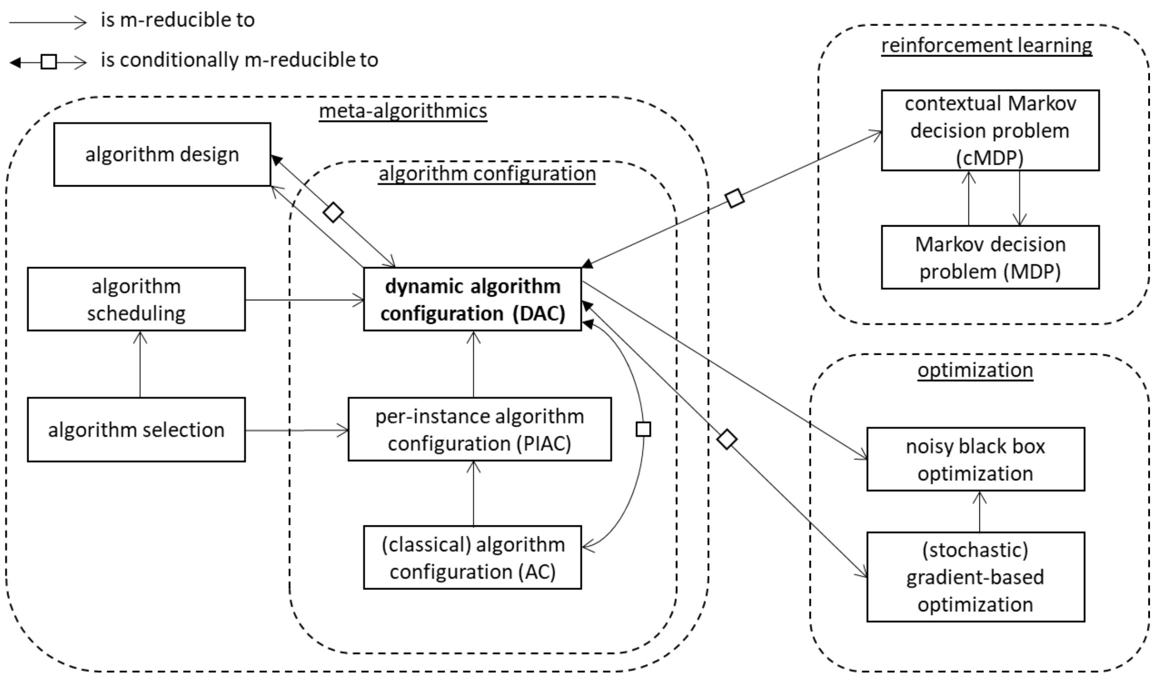

is m-reducible to
is conditionally m-reducible to
meta-algorithmics
algorithm design
algorithm configuration
algorithm scheduling
dynamic algorithm configuration (DAC)
algorithm selection
per-instance algorithm configuration (PIAC)
(classical) algorithm configuration (AC)
reinforcement learning
contextual Markov decision problem (cMDP)
Markov decision problem (MDP)
optimization
noisy black box optimization
(stochastic) gradient-based optimization

Figure 9: An overview of the reducibility relations between DAC and problems previously studied in the meta-algorithmics, reinforcement learning, and optimization communities; that we prove to exist in Section B.3. Note that arrows implied by the transitive/reflexive property of m-reducibility are not shown. Conditional reducibility indicates that a problem is only reducible to another, under specific conditions (i.e., not generally).

AC $\leq_m$ PIAC:

`formulate`$(\langle \mathcal{A}, \Theta, \mathcal{D}, c\rangle) = \langle \mathcal{A}, \Theta, \mathcal{D}, \Psi, c'\rangle$ with

- $\Psi = \{\psi_{\boldsymbol{\theta}} \,|\, \psi_{\boldsymbol{\theta}}(i) = \boldsymbol{\theta}, \forall \boldsymbol{\theta} \in \Theta\}$
- $c'(\psi_{\boldsymbol{\theta}}, i) = c(\boldsymbol{\theta}, i)$

`interpret`$(\langle \mathcal{A}, \Theta, \mathcal{D}, c\rangle, \psi_{\boldsymbol{\theta}^*}) = \boldsymbol{\theta}^*$

Proof Sketch: $\boldsymbol{\theta}^* \in \arg\min_{\boldsymbol{\theta}\in\Theta} \mathbb{E}_{i\sim\mathcal{D}}[c(\boldsymbol{\theta}, i)]$
By contradiction, assume $\boldsymbol{\theta}^* \notin \arg\min_{\boldsymbol{\theta}\in\Theta} \mathbb{E}_{i\sim\mathcal{D}}[c(\boldsymbol{\theta}, i)]$. This implies that there exists $\boldsymbol{\theta}' : \mathbb{E}_{i\sim\mathcal{D}}[c(\boldsymbol{\theta}', i)] < \mathbb{E}_{i\sim\mathcal{D}}[c(\boldsymbol{\theta}^*, i)]$. Since $\boldsymbol{\theta}' \in \Theta$ and $c(\boldsymbol{\theta}, i) = c'(\psi_{\boldsymbol{\theta}}, i)$, there must exist $\psi_{\boldsymbol{\theta}'} \in \Psi$ having $\mathbb{E}_{i\sim\mathcal{D}}[c'(\psi_{\boldsymbol{\theta}'}, i)] < \mathbb{E}_{i\sim\mathcal{D}}[c'(\psi_{\boldsymbol{\theta}^*}, i)]$, contradicting $\psi_{\boldsymbol{\theta}^*} \in \arg\min_{\psi\in\Psi} \mathbb{E}_{i\sim\mathcal{D}}[c'(\psi, i)]$. □

PIAC $\leq_m$ DAC:

`formulate`$(\langle \mathcal{A}, \Theta, \mathcal{D}, \Psi, c\rangle) = \langle \mathcal{A}', \Theta, \mathcal{D}, \Pi, c'\rangle$ with

- $\mathcal{A}'.\texttt{step}(s, i, \boldsymbol{\theta}) = \mathcal{A}(i, \boldsymbol{\theta})$ and $\mathcal{A}'.\texttt{init}(i) = \upsilon$ and $\mathcal{A}'.\texttt{is\_final}(s, i) \iff s \neq \upsilon$ (for some υ not being an output of $\mathcal{A}$, i.e., we perform exactly one step)
- $\Pi = \{\pi_\psi | \pi_\psi(s, i) = \psi(i), \forall \psi \in \Psi\}$
- $c'(\pi_\psi, i) = c(\psi, i)$

`interpret`$(\langle \mathcal{A}, \Theta, \mathcal{D}, \Psi, c\rangle, \pi_{\psi^*}) = \psi^*$

Proof Sketch: $\psi^* \in \arg\min_{\psi\in\Psi} \mathbb{E}_{i\sim\mathcal{D}}[c(\psi, i)]$.
By contradiction, assume $\psi^* \notin \arg\min_{\psi\in\Psi} \mathbb{E}_{i\sim\mathcal{D}}[c(\psi, i)]$. This implies that there exists $\psi' : \mathbb{E}_{i\sim\mathcal{D}}[c(\psi', i)] < \mathbb{E}_{i\sim\mathcal{D}}[c(\psi^*, i)]$. Since $\psi' \in \Psi$, and $c(\psi, i) = c'(\pi_\psi, i)$, there must exist a $\pi_{\psi'} \in \Pi$ having $\mathbb{E}_{i\sim\mathcal{D}}[c'(\pi_{\psi'}, i)] < \mathbb{E}_{i\sim\mathcal{D}}[c'(\pi_{\psi^*}, i)]$, contradicting $\pi_{\psi^*} \in \arg\min_{\pi\in\Pi} \mathbb{E}_{i\sim\mathcal{D}}[c'(\pi, i)]$. □

DAC $\leq_m^c$ AC:

Preconditions: We assume to be given a parametric representation Λ of the policy space, i.e., $\Pi = \{\pi_{\boldsymbol{\lambda}} \,|\, \boldsymbol{\lambda} \in \Lambda\}$.

`formulate`$(\langle \mathcal{A}, \Theta, \mathcal{D}, \Pi, c\rangle) = \langle \mathcal{A}', \Lambda, \mathcal{D}, c'\rangle$ with

- $\mathcal{A}'(i, \boldsymbol{\lambda}) = \mathcal{A}(i, \pi_{\boldsymbol{\lambda}})$.
- $c'(\boldsymbol{\lambda}, i) = c(\pi_{\boldsymbol{\lambda}}, i)$

`interpret`$(\langle \mathcal{A}, \Theta, \mathcal{D}, \Pi, c\rangle, \boldsymbol{\lambda}^*) = \pi_{\boldsymbol{\lambda}^*}$

Proof Sketch: $\pi_{\boldsymbol{\lambda}^*} \in \arg\min_{\pi_{\boldsymbol{\lambda}}\in\Pi} \mathbb{E}_{i\sim\mathcal{D}}[c(\pi_{\boldsymbol{\lambda}}, i)]$

$$\boldsymbol{\lambda}^* \in \arg\min_{\boldsymbol{\lambda}\in\Lambda} \mathbb{E}_{i\sim\mathcal{D}}[c'(\boldsymbol{\lambda}, i)] \implies$$
$$\boldsymbol{\lambda}^* \in \arg\min_{\boldsymbol{\lambda}\in\Lambda} \mathbb{E}_{i\sim\mathcal{D}}[c(\pi_{\boldsymbol{\lambda}}, i)] \implies$$
$$\pi_{\boldsymbol{\lambda}^*} \in \arg\min_{\pi_{\boldsymbol{\lambda}}\in\Pi} \mathbb{E}_{i\sim\mathcal{D}}[c(\pi_{\boldsymbol{\lambda}}, i)]$$

□

B.3.2 Algorithm Design

We formalize algorithm design as in previous work (Adriaensen, 2018):

Definition 7: Algorithm Design

Let A_U be the space of all algorithms.[a] Given a preference relation over $\preceq$[b] over A_U, find a $a^* \in A_U : a^* \not\prec a, \forall a \in A_U$.

a. A_U is a universal set containing "any procedure that solves some problem". Further formalization of this notion is hindered by the lack of a generally accepted, formal definition of "an algorithm".
b. $\preceq$ is assumed to be a preorder, i.e., a binary relation that is reflexive and transitive.

algorithm design $\leq_m^c$ DAC:

Preconditions: We assume we are given `init`, `step`, `is_final`, and a set of Π sub-routines such that algorithms $a_\pi \in A_\Pi \subset A_U$ that can be decompose in $\langle \texttt{init}, \texttt{step}, \texttt{is\_final}, \pi \rangle$ as in Algorithm 1 are at least as preferable as any other: $\forall a_\pi \in A_\Pi : a \preceq a_\pi, \forall a \in A_U \setminus A_\Pi$.

`formulate`$(\langle \preceq, \texttt{init}, \texttt{step}, \texttt{is\_final}, \Pi \rangle) = \langle \mathcal{A}, \Theta, \mathcal{D}, \Pi, c \rangle$ with

- $\mathcal{A}.\texttt{step} = \texttt{step}$ and $\mathcal{A}.\texttt{init} = \texttt{init}$ and $\mathcal{A}.\texttt{is\_final} = \texttt{is\_final}$
- Any choice of $\mathcal{D}$ and c such that $a_{\pi'} \prec a_\pi \implies \mathbb{E}_{i \sim \mathcal{D}}[c(\pi, i)] < \mathbb{E}_{i \sim \mathcal{D}}[c(\pi', i)]$. This can always be achieved, as a c that is solely a function of π can impose an arbitrary total order on Π and therefore also one consistent with $\preceq$.

`interpret`$(\langle \preceq, \texttt{init}, \texttt{step}, \texttt{is\_final}, \Pi \rangle, \pi^*) = a_{\pi^*}$

Proof Sketch: $a_{\pi^*} \not\prec a, \forall a \in A_U$
By contradiction, assume there exists $a' \in A_U : a' \succ a_{\pi^*}$. Given our pre-condition, we have $a' = a_{\pi'} \in A_\Pi$. From our choice of c follows that $\mathbb{E}_{i \sim \mathcal{D}}[c(\pi', i)] < \mathbb{E}_{i \sim \mathcal{D}}[c(\pi^*, i)]$, contradicting $\pi^* \in \arg\min_{\pi \in \Pi} \mathbb{E}_{i \sim \mathcal{D}}[c(\pi, i)]$. □

DAC $\leq_m$ algorithm design:

`formulate`$(\langle \mathcal{A}, \Theta, \mathcal{D}, \Pi, c \rangle) = \preceq$
satisfying

1. $\forall \pi \in \Pi : a \prec \pi, \forall a \in A_U \setminus \Pi$ and
2. $\forall \pi, \pi' \in \Pi : \mathbb{E}_{i \sim \mathcal{D}}[c(\pi, i)] < \mathbb{E}_{i \sim \mathcal{D}}[c(\pi', i)] \implies \pi' \prec \pi$.

`interpret`$(\langle \mathcal{A}, \Theta, \mathcal{D}, \Pi, c \rangle, a^*) = a^*$

Proof Sketch: $a^* \in \arg\min_{\pi \in \Pi} \mathbb{E}_{i \sim \mathcal{D}}[c(\pi, i)]$
By contradiction, assume $a^* \notin \arg\min_{\pi \in \Pi} \mathbb{E}_{i \sim \mathcal{D}}[c(\pi, i)]$. First, note that $a^* \in A_U \setminus \Pi$ and (1) would contradict $a^* \in A_U : a^* \not\prec a, \forall a \in A_U$. This implies there exists $\pi' \in \Pi : \mathbb{E}_{i \sim \mathcal{D}}[c(\pi', i)] < \mathbb{E}_{i \sim \mathcal{D}}[c(a^*, i)]$, implying $a^* \prec \pi'$ by (2) and contradicting $a^* \in A_U : a^* \not\prec a, \forall a \in A_U$. □

B.3.3 Algorithm Selection

For algorithm selection, we adopt the classical definition by Rice (1976):[20]

Definition 8: Algorithm Selection

Given $\langle A, I, c \rangle$:

- A finite set A of target algorithms
- A target problem space I
- A cost metric $c : A \times I \to \mathbb{R}$ assessing the cost of solving $i \in I$ using $a \in A$.[a]

Find a selection mapping $S^* : I \to A$ satisfying $S^*(i) \in \arg\min_{a \in A} c(a, i), \forall i \in I$.

a. The original definition uses a performance metric p (to be maximized).

The reducability algorithm selection to DAC follows by transitive property from its reducability to PIAC.

algorithm selection $\leq_m$ PIAC:

`formulate`$(\langle A, I, c \rangle) = \langle \mathcal{A}', \Theta, \mathcal{D}, \Psi, c' \rangle$ with

- $\mathcal{A}'(i, k) = a_k(i)$
- $\Theta = \{k \mid a_k \in A\}$ (single categorical parameter)
- $\mathcal{D} = U(I)$
- $\Psi : I \to \Theta$ (unconstrained)
- $c'(\psi, i) = c(\psi(i), i)$

`interpret`$(\langle A, I, c \rangle, \psi^*) = S^*$ with $S^*(i) = a_{\psi^*(i)}$

Proof Sketch: $S^*(i) \in \arg\min_{a \in A} c(a, i), \forall i \in I$
By contradiction, assume $\exists j \in I : S^*(j) \notin \arg\min_{a \in A} c(a, j)$. This implies there exists $S' : S'(j) \in \arg\min_{a \in A} c(a, j) \wedge S'(i') = S(i'), \forall i' \in I \setminus \{j\}$. Since Ψ is unconstrained every selection mapping S has its corresponding $\psi \in \Psi : S(i) = a_{\psi(i)}$ and therefore $\exists\, \psi' \in \Psi : c(\psi'(j), j) < c(\psi^*(j), j) \wedge c(\psi'(i'), i') = c(\psi^*(i), i'), \forall i' \in I \setminus \{j\}$. From $c'(\psi, i) = c(\psi(i), i)$ and $\mathcal{D}(j) > 0$ follows that $\mathbb{E}_{i \sim \mathcal{D}}[c'(\psi', i)] < \mathbb{E}_{i \sim \mathcal{D}}[c'(\psi^*, i)]$, contradicting $\psi^* \in \arg\min_{\psi \in \Psi} \mathbb{E}_{i \sim \mathcal{D}}[c'(\psi, i)]$. $\square$

B.3.4 Algorithm Scheduling

We define a variant of algorithm scheduling that considers allocating a fixed time budget to a finite set of target algorithms in an instance-aware and dynamic fashion:

20. Rice (1976) defines many different more general variants of the problem. However, we will restrict ourselves to a canonical variant, i.e., selecting the best algorithm, per-instance, from finite alternatives, without constraints on the selection mappings.

Definition 9: Algorithm Scheduling

Given $\langle A, B, I, \Delta, c\rangle$:

– A finite set A of stepwise executable target algorithms such that the execution of the k^{th} algorithm $a_k \in A$ can be decomposed as a consecutive application of a sub-routine $a_k.\texttt{tstep}$ such that the state of algorithm a_k when solving a problem instance i, after t time steps, is given by $s_{k,i,t} = a_k.\texttt{tstep}(s_{k,i,t-1})$ with $s_{k,i,0} = i$.

– A finite budget B of time steps to be allocated to algorithms in A.

– A target problem space I

– A space of *dynamic scheduling policies* $\delta \in \Delta$ with $\delta : \mathcal{S}^{|A|} \times I \times \mathbb{N} \to A$ choosing which algorithm $a_k \in A$ to resume executing in the next time step, as a function of the total time $T \in \mathbb{N}$ elapsed thus far, the instance $i \in I$ being solved and the vector of states s_{m,i,t_m} of each algorithm $a_m \in A$.

– A cost metric $c : A \times I \times \mathbb{N} \to \mathbb{R}$ assessing the cost of solving $i \in I$ using $a \in A$ for $t \in \mathbb{N}$ time steps.

Find a dynamic scheduling policy $\delta^* \in \arg\min_{\delta \in \Delta} \sum_{i \in I}(\min_{a_k \in A} c(a_k, i, t_k^{B,\delta,i}))$ where $t_k^{T,\delta,i}$ is the total time allocated to a_k by δ after T scheduling steps on instance i, and is given by

$$t_k^{T,\delta,i} = \begin{cases} 0 & T = 0 \\ t_k^{T-1,\delta,i} & T > 0 \quad \wedge \quad a_k \neq \delta(\boldsymbol{s}^{T-1,\delta,i}, i, T-1) \\ t_k^{T-1,\delta,i} + 1 & T > 0 \quad \wedge \quad a_k = \delta(\boldsymbol{s}^{T-1,\delta,i}, i, T-1) \end{cases}$$

with

$$s_k^{T,\delta,i} = \begin{cases} i & T = 0 \\ s_k^{T-1,\delta,i} & T > 0 \quad \wedge \quad a_k \neq \delta(\boldsymbol{s}^{T-1,\delta,i}, i, T-1) \\ a_k.\texttt{tstep}(s_k^{T-1,\delta,i}) & T > 0 \quad \wedge \quad a_k = \delta(\boldsymbol{s}^{T-1,\delta,i}, i, T-1) \end{cases}$$

algorithm scheduling $\leq_m$ DAC:

$\texttt{formulate}(\langle A, B, I, \Delta, c\rangle) = \langle \mathcal{A}, \Theta, \mathcal{D}, \Pi, c'\rangle$ with

- $\mathcal{A}.\texttt{step}((\boldsymbol{s}, i, T), i, \theta) = (\boldsymbol{s}', i, T+1)$ with $s'_k = \begin{cases} s_k & \theta \neq k \\ a_k.\texttt{tstep}(s_k) & \theta = k \end{cases}$

  $\mathcal{A}.\texttt{init}(i) = (\boldsymbol{s}, i, 0)$ with $s_k = i$
  $\mathcal{A}.\texttt{is\_final}((\boldsymbol{s}, i, T), i) \Leftrightarrow T = B$.
- $\Theta = \{k \mid a_k \in A\}$
- $\mathcal{D} = U(I)$
- $\Pi = \{\pi_\delta \mid \delta \in \Delta\}$ with $\pi_\delta((\boldsymbol{s}, i, T), i) = k \Leftrightarrow \delta(\boldsymbol{s}, i, T) = a_k$
- $c'(\pi_\delta, i) = \min_{a_k \in A} c(a_k, i, t_k^{B,\delta,i})$

$\texttt{interpret}(\langle A, B, I, \Delta, c\rangle, \pi_{\delta^*}) = \delta^*$

Proof Sketch: $\delta^* \in \arg\min_{\delta \in \Delta} \sum_{i \in I}(\min_{a_k \in A} c(a_k, i, t_k^{B,\delta,i}))$

$$\pi_{\delta^*} \in \underset{\pi_\delta \in \Pi}{\arg\min}\ \mathbb{E}_{i \sim \mathcal{D}}\left[c'(\pi_\delta, i)\right] \implies$$

$$\delta^* \in \underset{\delta \in \Delta}{\arg\min}\ \mathbb{E}_{i \sim \mathcal{D}}\left[\min_{a_k \in A} c(a_k, i, t_k^{B,\delta,i})\right] \implies$$

$$\delta^* \in \underset{\delta \in \Delta}{\arg\min}\ \frac{1}{|I|}\sum_{i \in I}\left[\min_{a_k \in A} c(a_k, i, t_k^{B,\delta,i})\right] \implies$$

$$\delta^* \in \underset{\delta \in \Delta}{\arg\min} \sum_{i \in I}(\min_{a_k \in A} c(a_k, i, t_k^{B,\delta,i}))$$

□

B.3.5 Reinforcement Learning

In Section 4.1, we discussed the relation between DAC and the

Definition 10: Markov Decision Problem (MDP)

Given $\langle S, A, T, R\rangle$:

- A state space S.
- An action space A
- A transition function $T : S \times A \to S$
- A reward function $R : S \times A \to \mathbb{R}$

Find a policy $\pi : S \to A$ satisfying $\pi^*(s) \in \arg\max_{a \in A} R(s,a) + V^*(T(s,a))$ with V^* being the optimal value-state function, i.e., $V^*(s) = \max_{a \in A} R(s,a) + V^*(T(s,a))$.

Note that we will restrict ourselves to episodic MDPs, where we have absorbing states $S_{\mathcal{H}} = \{s \,|\, T(s,a) = s, \forall a \in A\}$ having $R(s,a) = 0, \forall s \in S_{\mathcal{H}}$ that are reached within an arbitrarily large, but finite horizon $\mathcal{H}$ and therefore $V^*(s) = \max_\pi \sum_{t=0}^{\mathcal{H}-1} R(s_{\pi,t}, \pi(s_{\pi,t}))$ where $s_{\pi,t} = T(s_{\pi,t-1}, \pi(s_{\pi,t-1}))$ is the t^{th} state encountered when following π starting in $s_{\pi,0} = s$.

Since standard RL methods are not instance-aware, Biedenkapp et al. (2020) proposed to model DAC as a

Definition 11: Contextual Markov Decision Problem (cMDP, Hallak et al., 2015)

Given $\langle \mathcal{C}, S, A, \mathcal{M} \rangle$:

- A context space $\mathcal{C}$
- A shared state space S
- A shared action space A
- A function $\mathcal{M}$ mapping any $c \in \mathcal{C}$ to an MDP $\mathcal{M}(c) = \langle S, A, T_c, R_c \rangle$ with
  - a context-dependent transition function $T_c : S \times A \to S$
  - a context-dependent reward function $R_c : S \times A \to \mathbb{R}$

Find a policy $\pi : S \times \mathcal{C} \to A$ satisfying $\pi^*(s, c) \in \arg\max_{a \in A} R_c(s, a) + V_c^*(T_c(s, a))$ with $V_c^*(s) = \max_{a \in A} R_c(s, a) + V_c^*(T_c(s, a))$.

Please remark that we assume the context to be observable and our objective to be finding an optimal *context-dependent* policy. We will also assume MDP $\mathcal{M}(i)$ to be episodic. As a consequence, this formulation is m-equivalent to that of an ordinary episodic MDP. The cMDP formulation is nonetheless interesting in that it can capture various aspects of DAC abstracted in the MDP reduction: DAC $\leq_m$ MDP $\leq_m$ cMDP.

DAC $\leq_m^c$ cMDP:

Preconditions:

1. The cost function c is stepwise decomposable, i.e., we are given functions $\langle \mathrm{c_{init}}, \mathrm{c_{step}} \rangle$, such that

$$c(\pi, i) = \mathrm{c_{init}}(i) + \sum_{t=0}^{T-1} \mathrm{c_{step}}(s_t, i, \pi(s_t, i))$$

   where

$$s_0 = \texttt{init}(i) \quad \wedge \quad s_t = \texttt{step}(s_{t-1}, i, \pi(s_{t-1}, i)) \quad \wedge \quad \texttt{is\_final}(s_t, i) \Leftrightarrow t = T$$

2. The policy space is unconstrained, i.e., $\Pi = \{\pi \mid \pi(s, i) \in \Theta, \ \forall s \in \mathcal{S} \wedge i \in I\}$

$\texttt{formulate}(\langle \mathcal{A}, \Theta, \mathcal{D}, \Pi, c \rangle) = \langle \mathcal{C}, S, A, \mathcal{M} \rangle$ with

- $\mathcal{C} = I$ the domain of $\mathcal{D}$.
- $S = \mathcal{S}$ the set of algorithm states.
- $A = \Theta$
- $\mathcal{M}(i) = \langle S, A, T_i, R_i \rangle$ with

- $T_i(s, \boldsymbol{\theta}) = \begin{cases} s & \texttt{is\_final}(s, i) \\ \texttt{step}(s, i, \boldsymbol{\theta}) & \neg\, \texttt{is\_final}(s, i) \end{cases}$
- $R_i(s, \boldsymbol{\theta}) = \begin{cases} 0 & \texttt{is\_final}(s, i) \\ \mathrm{c}_{\text{step}}(s, i, \boldsymbol{\theta}) & \neg\, \texttt{is\_final}(s, i) \end{cases}$

$\texttt{interpret}(\langle \mathcal{A}, \Theta, \mathcal{D}, \Pi, c\rangle, \pi^*) \;= \pi^*$

Proof Sketch: $\pi^* \in \arg\max_\pi \mathbb{E}_{i\sim\mathcal{D}}\, c(\pi, i)$

We first note that since $S = \mathcal{S} \wedge A = \Theta \wedge \mathcal{C} = I$ and given precondition (2), it follows that both problems have the exact same policy space Π. It remains to show that optimality in the resulting cMDP implies optimality in the original DAC:

$$\pi^*(s,i) \in \underset{\boldsymbol{\theta}\in\Theta}{\arg\max}\; R_i(s, \boldsymbol{\theta}) + V_i^*(T_i(s, \boldsymbol{\theta})) \quad (\forall s \in S, \forall i \in I) \implies$$
$$\pi^*(\texttt{init}(i), i) \in \underset{\boldsymbol{\theta}\in\Theta}{\arg\max}\; R_i(\texttt{init}(i), \boldsymbol{\theta}) + V_i^*(T_i(\texttt{init}(i), \boldsymbol{\theta})) \quad (\forall i \in I) \implies$$
$$\pi^* \in \underset{\pi}{\arg\max}\; \mathbb{E}_{i\sim\mathcal{D}}\, R_i(\texttt{init}(i), \pi(\texttt{init}(i), i)) + V_i^*(T_i(\texttt{init}(i), \pi(\texttt{init}(i), i)))$$

Since each $\mathcal{M}(i)$ is episodic, this objective can be rewritten as:

$$\pi^* \in \underset{\pi}{\arg\max}\; \mathbb{E}_{i\sim\mathcal{D}} \sum_{t=0}^{\mathcal{H}-1} R_i(s_{\pi,t,i}, \pi(s_{\pi,t,i})) \text{ where } s_{\pi,t,i} = \begin{cases} \texttt{init}(i) & t = 0 \\ T_i(s_{\pi,t-1,i}, \pi(s_{\pi,t-1,i})) & t > 0 \end{cases}$$
$$\pi^* \in \underset{\pi}{\arg\max}\; \mathbb{E}_{i\sim\mathcal{D}} \sum_{t=0}^{T-1} \mathrm{c}_{\text{step}}(s_{\pi,t,i}, i, \pi(s_{\pi,t,i})) \text{ where } s_{\pi,t,i} = \begin{cases} \texttt{init}(i) & t = 0 \\ \texttt{step}(s_{\pi,t-1,i}, i, \pi(s_{\pi,t-1,i})) & t > 0 \end{cases}$$
$$\pi^* \in \underset{\pi}{\arg\max}\; \mathbb{E}_{i\sim\mathcal{D}}\, c(\pi, i)$$

□

It is worth noting that the optimality of π^* does not depend on $\mathcal{D}$, `init`, or c_{init}.
Conditional reducability to an ordinary MDP follows from the transitive property and

cMDP $\leq_m$ MDP:

$\texttt{formulate}(\langle \mathcal{C}, S, A, \mathcal{M}\rangle) \;= \langle S', A, T, R\rangle$ with

- $S' = S \times \mathcal{C}$
- $T((s, c), a) = T_c(s, a)$
- $R((s, c), a) = R_c(s, a)$

where $\mathcal{M}(c) = \langle S, A, T_c, R_c\rangle$.

$\texttt{interpret}(\langle \mathcal{C}, S, A, \mathcal{M}\rangle, \pi^*) \;= \pi'^*$ with $\pi'^*(s, c) = \pi^*((s, c))$

Proof Sketch: $\pi'^*(s,c) \in \arg\max_{a \in A} R_c(s,a) + V_c^*(s)$

$$\pi^*((s,c)) \in \arg\max_{a \in A} R((s,c),a) + V^*(T((s,c),a)) \implies$$
$$\pi'^*(s,c) \in \arg\max_{a \in A} R_c(s,a) + V^*(T_c(s,a))$$

It remains to show that $V^*((s,c)) = V_c^*(s)$. We can prove this by induction on the maximum number of steps $\mathcal{H}_{s,c}$ before reaching an absorbing state following any policy starting from state s in context c. The base case $\mathcal{H}_{s,c} = 0$ follows from $T((s,c),a) = (s,c) \iff T_c(s,a) = s$. Now, for the recursive case, assuming this holds for $\mathcal{H}_{s,c} \leq n-1$, it also holds for $\mathcal{H}_{s,c} = n$

$$\begin{aligned} V^*((s,c)) &= \max_{a \in A} R((s,c),a) + V^*(T((s,c),a)) \\ &= \max_{a \in A} R_c(s,a) + V^*(T_c(s,a)) \\ &= \max_{a \in A} R_c(s,a) + V_c^*(T_c(s,a)) = V_c^*(s) \end{aligned}$$

since $\mathcal{H}_{T_c(s,a),c} \leq n-1$. $\square$

and both are m-equivalent since

MDP $\leq_m$ cMDP:

`formulate(`$\langle S, A, T, R \rangle$`)` $= \langle \mathcal{C}, S, A, \mathcal{M} \rangle$ with

- $\mathcal{C} = \{0\}$
- $T_0(s,a) = T(s,a)$
- $R_0(s,a) = R(s,a)$

where $\mathcal{M}(0) = \langle S, A, T_0, R_0 \rangle$.

`interpret(`$\langle S, A, T, R \rangle, \pi^*$`)` $= \pi'^*$ with $\pi'^*(s) = \pi^*(s, 0)$

Proof Sketch: $\pi'^*(s) \in \arg\max_{a \in A} \ R(s,a) + V^*(s)$

$$\pi^*(s,0) \in \arg\max_{a \in A} \ R_0(s,a) + V_0^*(T_0(s,a)) \implies$$
$$\pi'^*(s) \in \arg\max_{a \in A} \ R(s,a) + V_0^*(T(s,a))$$

It remains to show that $V^*(s) = V_0^*(s)$. We can prove this by induction on the maximum number of steps $\mathcal{H}_s$ before reaching an absorbing state following any policy starting from state s. The base case $\mathcal{H}_s = 0$ follows from $T(s,a) = s \Leftrightarrow T_0(s,a) = s$. Now the recursive case, assuming this holds for $\mathcal{H}_s \leq n-1$, it also holds for $\mathcal{H}_s = n$

$$\begin{aligned} V^*(s) &= \max_{a \in A} R(s,a) + V^*(T(s,a)) \\ &= \max_{a \in A} R_0(s,a) + V^*(T_0(s,a)) \\ &= \max_{a \in A} R_0(s,a) + V_0^*(T_0(s,a)) = V_0^*(s) \end{aligned}$$

since $\mathcal{H}_{T_0(s,a)} \leq n-1$. $\square$

B.3.6 Optimization

In Section 4.2, we discussed the relation between DAC and

Definition 12: Noisy Black Box Optimization

Given $\langle X, e \rangle$:

- A search space X
- A noisy evaluation sub-routine e

Find a $x^* \in \arg\min_{x \in X} \mathbb{E}[e(x)]$.

DAC $\leq_m$ noisy black box optimization:

`formulate`$(\langle \mathcal{A}, \Theta, \mathcal{D}, \Pi, c \rangle) = \langle \Pi, e \rangle$ where $e(\pi) = c(\pi, i)$ with $i \sim \mathcal{D}$.

`interpret`$(\langle \mathcal{A}, \Theta, \mathcal{D}, \Pi, c \rangle), \pi^*) = \pi^*$

Proof Sketch: $\pi^* \in \arg\min_{\pi \in \Pi} \mathbb{E}_{i \sim \mathcal{D}} [c(\pi, i)]$
We have $\pi^* \in \arg\min_{\pi \in \Pi} \mathbb{E}[e(\pi)]$, since π^* is a solution for noisy black box optimization problem. Since $e(\pi) = c(\pi, i)$ and $i \sim \mathcal{D}$, this is equivalent to $\pi^* \in \arg\min_{\pi \in \Pi} \mathbb{E}_{i \sim \mathcal{D}} [c(\pi, i)]$.
$\square$

Also in Section 4.2, we discussed the possibility of solving DAC using

Definition 13: Stochastic Gradient-Based Optimization

Given $\langle X, e, e' \rangle$:

- X and e as in Definition 12.
- A stochastic differentiation sub-routine e' satisfying $\mathbb{E}[e'(x)] = \mathbb{E}[\frac{\partial e(x)}{\partial x}]$.

Find a $x^* \in \arg\min_{x \in X} \mathbb{E}[e(x)]$.

DAC $\leq_m^c$ stochastic gradient-based optimization:

Preconditions:

1. We assume to be given a parametric representation Λ of the policy space, i.e., $\Pi = \{\pi_{\boldsymbol{\lambda}} \,|\, \boldsymbol{\lambda} \in \Lambda\}$.
2. We assume $c(\pi_{\boldsymbol{\lambda}}, i)$ to be piece-wise differentiable w.r.t. $\boldsymbol{\lambda}$ and a sub-routine for calculating $\frac{\partial c(\pi_{\boldsymbol{\lambda}},i)}{\partial \boldsymbol{\lambda}}$ to be given.

`formulate`$(\langle \mathcal{A}, \Theta, \mathcal{D}, \Pi, c\rangle) = \langle \Lambda, e, e' \rangle$ with

- $e(\boldsymbol{\lambda}) = c(\pi_{\boldsymbol{\lambda}}, i)$ with $i \sim \mathcal{D}$.
- $e'(\boldsymbol{\lambda}) = \frac{\partial c(\pi_{\boldsymbol{\lambda}},i)}{\partial \boldsymbol{\lambda}}$ with $i \sim \mathcal{D}$.

`interpret`$(\langle \mathcal{A}, \Theta, \mathcal{D}, \Pi, c\rangle, \boldsymbol{\lambda}^*) = \pi_{\boldsymbol{\lambda}^*}$

Proof Sketch: $\pi_{\boldsymbol{\lambda}^*} \in \arg\min_{\pi_{\boldsymbol{\lambda}} \in \Pi} \mathbb{E}_{i \sim \mathcal{D}} \left[c(\pi_{\boldsymbol{\lambda}}, i)\right]$

We have $\boldsymbol{\lambda}^* \in \arg\min_{\boldsymbol{\lambda} \in \Lambda} \mathbb{E}[e(\boldsymbol{\lambda})]$, since $\boldsymbol{\lambda}^*$ is a solution for black box optimization problem. Since $e(\boldsymbol{\lambda}) = c(\pi_{\boldsymbol{\lambda}}, i)$ and $i \sim \mathcal{D}$, this is equivalent to $\boldsymbol{\lambda}^* \in \arg\min_{\boldsymbol{\lambda} \in \Lambda} \mathbb{E}_{i \sim \mathcal{D}} \left[c(\pi_{\boldsymbol{\lambda}}, i)\right]$. Substituting every $\boldsymbol{\lambda}$ by its corresponding policy $\pi_{\boldsymbol{\lambda}}$, we get $\pi_{\boldsymbol{\lambda}^*} \in \arg\min_{\pi_{\boldsymbol{\lambda}} \in \Pi} \mathbb{E}_{i \sim \mathcal{D}} \left[c(\pi_{\boldsymbol{\lambda}}, i)\right]$
$\square$

At first sight, precondition (2) may seem very strong. In what follows, we show a sufficient condition that is arguably not so strong.

Sufficient Conditions: Next to the precondition (1), we assume

3. to be given a parametric representation of the state space.
4. the cost function c to be stepwise decomposable in $\langle \mathrm{c_{init}}, \mathrm{c_{step}} \rangle$ such that

$$c(\pi_{\boldsymbol{\lambda}}, i) = \mathrm{c_{init}}(i) + \sum_{t=0}^{T-1} \mathrm{c_{step}}(s_t, i, \pi_{\boldsymbol{\lambda}}(s_t, i))$$

where

$$s_0 = \texttt{init}(i) \quad \wedge \quad s_t = \texttt{step}(s_{t-1}, i, \pi_{\boldsymbol{\lambda}}(s_{t-1}, i)) \quad \wedge \quad \texttt{is\_final}(s_t, i) \Leftrightarrow t = T$$

5. to be given sub-routines for calculating the following partial derivatives: $\frac{\partial \pi_{\boldsymbol{\lambda}}(s,i)}{\partial \boldsymbol{\lambda}}$, $\frac{\partial\, \mathrm{c_{step}}(s,i,\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$, $\frac{\partial\, \mathrm{c_{step}}(s,i,\boldsymbol{\theta})}{\partial s}$, $\frac{\partial\, \texttt{step}(s,i,\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$, and $\frac{\partial\, \texttt{step}(s,i,\boldsymbol{\theta})}{\partial s}$.

Proof Sketch: Under these conditions precondition 2 also holds.

The piece-wise derivative of c w.r.t. $\boldsymbol{\lambda}$ can be calculated as follows

$$\begin{aligned}
\frac{\partial c(\pi_{\boldsymbol{\lambda}}, i)}{\partial \boldsymbol{\lambda}} &= \frac{\partial \left(\mathrm{c_{init}}(i) + \sum_{t=0}^{T-1} \mathrm{c_{step}}(s_t, i, \pi_{\boldsymbol{\lambda}}(s_t, i))\right)}{\partial \boldsymbol{\lambda}} \\
&= \frac{\partial\, \mathrm{c_{init}}(i)}{\partial \boldsymbol{\lambda}} + \sum_{t=0}^{T-1} \frac{\partial\, \mathrm{c_{step}}(s_t, i, \pi_{\boldsymbol{\lambda}}(s_t, i))}{\partial \boldsymbol{\lambda}} \\
&= \sum_{t=0}^{T-1} \frac{\partial\, \mathrm{c_{step}}(s_t, i, \pi_{\boldsymbol{\lambda}}(s_t, i))}{\partial \boldsymbol{\lambda}}
\end{aligned}$$

where

$$\begin{aligned}\frac{\partial\, \mathrm{c}_{\text{step}}(s_t, i, \pi_{\boldsymbol{\lambda}}(s_t, i))}{\partial \boldsymbol{\lambda}} =& \frac{\partial\, \mathrm{c}_{\text{step}}(s_t, i, \pi_{\boldsymbol{\lambda}}(s_t, i))}{\partial \pi_{\boldsymbol{\lambda}}(s_t, i)} \cdot \frac{\partial \pi_{\boldsymbol{\lambda}}(s_t, i)}{\partial \boldsymbol{\lambda}} \\ &+ \frac{\partial s_t}{\partial \boldsymbol{\lambda}} \cdot \left(\frac{\partial\, \mathrm{c}_{\text{step}}(s_t, i, \pi_{\boldsymbol{\lambda}}(s_t, i))}{\partial s_t} + \frac{\partial\, \mathrm{c}_{\text{step}}(s_t, i, \pi_{\boldsymbol{\lambda}}(s_t, i))}{\partial \pi_{\boldsymbol{\lambda}}(s_t, i)} \cdot \frac{\partial \pi_{\boldsymbol{\lambda}}(s_t, i)}{\partial s_t} \right)\end{aligned}$$

with

$$\begin{aligned}\frac{\partial s_t}{\partial \boldsymbol{\lambda}} =& \frac{\partial\, \mathtt{step}(s_{t-1}, i, \pi_{\boldsymbol{\lambda}}(s_{t-1}, i))}{\pi_{\boldsymbol{\lambda}}(s_{t-1}, i)} \cdot \frac{\partial \pi_{\boldsymbol{\lambda}}(s_{t-1}, i)}{\partial \boldsymbol{\lambda}} \\ &+ \frac{\partial s_{t-1}}{\partial \boldsymbol{\lambda}} \cdot \left(\frac{\partial\, \mathtt{step}(s_{t-1}, i, \pi_{\boldsymbol{\lambda}}(s_{t-1}, i))}{\partial s_{t-1}} + \frac{\partial\, \mathtt{step}(s_{t-1}, i, \pi_{\boldsymbol{\lambda}}(s_{t-1}, i))}{\partial \pi_{\boldsymbol{\lambda}}(s_{t-1}, i)} \cdot \frac{\partial \pi_{\boldsymbol{\lambda}}(s_{t-1}, i)}{\partial s_{t-1}} \right)\end{aligned}$$

if $t > 0$ and $\frac{\partial s_0}{\partial \boldsymbol{\lambda}} = 0$

Note that we can reuse the quantities calculated in the previous step, resulting in a procedure known as *forward-mode* differentiation. While we will not derive the formulas here, $\frac{\partial c(\pi_{\boldsymbol{\lambda}}, i)}{\partial \boldsymbol{\lambda}}$ can also be calculated using *reverse-mode* differentiation, a procedure also known as *backpropagation* in the context of neural networks. □

References

Adriaensen, S., & Nowé, A. (2016). Towards a white box approach to automated algorithm design.. In Kambhampati, S. (Ed.), *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI'16)*, pp. 554–560.

Adriaensen, S. (2018). *On the Semi-automated Design of Reusable Heuristics*. Ph.D. thesis, Vrije Universiteit Brussel.

Aine, S., Kumar, R., & Chakrabarti, P. (2009). Adaptive parameter control of evolutionary algorithms to improve quality-time trade-off. *Applied Soft Computing*, *9*(2), 527–540.

Aleti, A., & Moser, I. (2016). A systematic literature review of adaptive parameter control methods for evolutionary algorithms. *ACM Comput. Surv.*, *49*(3), 1–35.

Almeida, D., Winter, C., Tang, J., & Zaremba, W. (2021). A generalizable approach to learning optimizers. *arXiv preprint arXiv:2106.00958, [cs.LG]*.

Altman, E. (1999). *Constrained Markov decision processes: stochastic modeling*. Routledge.

Andersson, M., Bandaru, S., & Ng, A. H. (2016). Tuning of multiple parameter sets in evolutionary algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pp. 533–540.

Andrychowicz, M., Denil, M., Colmenarejo, S. G., Hoffman, M., Pfau, D., Schaul, T., & de Freitas, N. (2016). Learning to learn by gradient descent by gradient descent. In Lee, D., Sugiyama, M., von Luxburg, U., Guyon, I., & Garnett, R. (Eds.), *Proceedings of the 29th International Conference on Advances in Neural Information Processing Systems (NeurIPS'16)*, pp. 3981–3989. Curran Associates.

Ansótegui, C., Malitsky, Y., Sellmann, M., & Tierney, K. (2015). Model-based genetic algorithms for algorithm configuration. In Yang, Q., & Wooldridge, M. (Eds.), *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI'15)*, pp. 733–739.

Ansótegui, C., Pon, J., Sellmann, M., & Tierney, K. (2021). PyDGGA: Distributed GGA for automatic configuration. In Li, C., & Manyà, F. (Eds.), *Theory and Applications of Satisfiability Testing - SAT*, Vol. 12831 of *Lecture Notes in Computer Science*, pp. 11–20. Springer.

Ansótegui, C., Sellmann, M., & Tierney, K. (2009). A gender-based genetic algorithm for the automatic configuration of algorithms. In Gent, I. (Ed.), *Proceedings of the Fifteenth International Conference on Principles and Practice of Constraint Programming (CP'09)*, Vol. 5732 of *Lecture Notes in Computer Science*, pp. 142–157. Springer.

Ansótegui, C., Pon, J., Sellmann, M., & Tierney, K. (2017). Reactive dialectic search portfolios for MaxSAT. In S.Singh, & Markovitch, S. (Eds.), *Proceedings of the Thirty-First Conference on Artificial Intelligence (AAAI'17)*, pp. 765–772. AAAI Press.

Auger, A., & Hansen, N. (2005). A restart CMA evolution strategy with increasing population size. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC*, pp. 1769–1776. IEEE.

Bard, J. F. (2013). *Practical bilevel optimization: algorithms and applications*, Vol. 30. Springer Science & Business Media.

Barozet, A., Molloy, K., Vaisset, M., Siméon, T., & Cortés, J. (2020). A reinforcement-learning-based approach to enhance exhaustive protein loop sampling. *Bioinform.*, *36*(4), 1099–1106.

Battiti, R., & Campigotto, P. (2012). An investigation of reinforcement learning for reactive search optimization. In Hamadi, Y., Monfroy, E., & Saubion, F. (Eds.), *Autonomous Search*, pp. 131–160. Springer.

Battiti, R., Brunato, M., & Mascia, F. (2008). *Reactive search and intelligent optimization*, Vol. 45. Springer Science & Business Media.

Battiti, R., & Tecchiolli, G. (1994). The reactive tabu search. *ORSA journal on computing*, *6*(2), 126–140.

Baydin, A., Cornish, R., Rubio, D., Schmidt, M., & Wood, F. (2018). Online learning rate adaption with hypergradient descent. In *Proceedings of the International Conference on Learning Representations (ICLR'18)*. Published online: `iclr.cc`.

Bello, I., Zoph, B., Vasudevan, V., & Le, Q. V. (2017). Neural optimizer search with reinforcement learning. In *International Conference on Machine Learning*, pp. 459–468. PMLR.

Bengio, Y. (2012). Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*, pp. 437–478. Springer.

Bhatia, A., Svegliato, J., & Zilberstein, S. (2021). Tuning the hyperparameters of anytime planning: A deep reinforcement learning approach. In *ICAPS 2021 Workshop on Heuristics and Search for Domain-independent Planning*.

Biedenkapp, A., Bozkurt, H. F., Eimer, T., Hutter, F., & Lindauer, M. (2020). Dynamic algorithm configuration: Foundation of a new meta-algorithmic framework. In Lang, J., Giacomo, G. D., Dilkina, B., & Milano, M. (Eds.), *Proceedings of the Twenty-fourth European Conference on Artificial Intelligence (ECAI'20)*, pp. 427–434.

Biedenkapp, A., Dang, N., Krejca, M. S., Hutter, F., & Doerr, C. (2022). Theory-inspired parameter control benchmarks for dynamic algorithm configuration. In Fieldsend, J. (Ed.), *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'22)*. ACM Press.

Biedenkapp, A., Lindauer, M., Eggensperger, K., Fawcett, C., Hoos, H., & Hutter, F. (2017). Efficient parameter importance analysis via ablation with surrogates. In S.Singh, & Markovitch, S. (Eds.), *Proceedings of the Thirty-First Conference on Artificial Intelligence (AAAI'17)*, pp. 773–779. AAAI Press.

Biedenkapp, A., Marben, J., Lindauer, M., & Hutter, F. (2018). CAVE: Configuration assessment, visualization and evaluation. In Battiti, R., Brunato, M., Kotsireas, I., & Pardalos, P. (Eds.), *Proceedings of the International Conference on Learning and Intelligent Optimization (LION)*, Lecture Notes in Computer Science. Springer.

Biedenkapp, A., Rajan, R., Hutter, F., & Lindauer, M. (2021). TempoRL: Learning when to act. In *Proceedings of the 38th International Conference on Machine Learning (ICML 2021)*.

Birattari, M., Stützle, T., Paquete, L., & Varrentrapp, K. (2002). A racing algorithm for configuring metaheuristics. In Langdon, W., Cantu-Paz, E., Mathias, K., Roy, R., Davis, D., Poli, R., Balakrishnan, K., Honavar, V., Rudolph, G., Wegener, J., Bull, L., Potter, M., Schultz, A., Miller, J., Burke, E., & Jonoska, N. (Eds.), *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'02)*, pp. 11–18. Morgan Kaufmann Publishers.

Bischl, B., Kerschke, P., Kotthoff, L., Lindauer, M., Malitsky, Y., Frechétte, A., Hoos, H., Hutter, F., Leyton-Brown, K., Tierney, K., & Vanschoren, J. (2016). ASlib: A benchmark library for algorithm selection. *Artificial Intelligence*, *237*, 41–58.

Bonet, B., & Geffner, H. (2001). Planning as heuristic search. *Artificial Intelligence*, *129*(1), 5–33.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). OpenAI Gym. *arXiv preprint arXiv:1606.01540, [cs.LG]*.

Burke, E. K., Hyde, M. R., Kendall, G., Ochoa, G., Ozcan, E., & Woodward, J. R. (2009). Exploring hyper-heuristic methodologies with genetic programming. In *Computational intelligence*, pp. 177–201. Springer.

Carchrae, T., & Beck, J. (2004). Low-knowledge algorithm control. In *Proceedings of the 19th National Conference on Artifical Intelligence*, AAAI'04, p. 49–54. AAAI Press.

Chen, F., Gao, Y., qian Chen, Z., & fu Chen, S. (2005). SCGA: Controlling genetic algorithms with sarsa(0). In *International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce (CIMCA-IAWTIC'06)*, Vol. 1, pp. 1177–1183.

Chen, T., Chen, X., Chen, W., Heaton, H., Liu, J., Wang, Z., & Yin, W. (2021). Learning to optimize: A primer and a benchmark. *arXiv preprint arXiv:2103.12828*, *[cs.LG]*.

Chrabaszcz, P., Loshchilov, I., & Hutter, F. (2018). Back to basics: Benchmarking canonical evolution strategies for playing atari. In Lang, J. (Ed.), *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI'18)*, pp. 1419–1426.

Daniel, C., Taylor, J., & Nowozin, S. (2016). Learning step size controllers for robust neural network training. In Schuurmans, D., & Wellman, M. (Eds.), *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI'16)*. AAAI Press.

Doerr, B. (2019). Analyzing randomized search heuristics via stochastic domination. *Theoretical Computer Science*, *773*, 115–137.

Doerr, B., & Doerr, C. (2020). Theory of parameter control for discrete black-box optimization: Provable performance gains through dynamic parameter choices. In Doerr, B., & Neumann, F. (Eds.), *Theory of Evolutionary Computation*, pp. 271–321. Springer.

Doerr, C., Wang, H., Ye, F., van Rijn, S., & Bäck, T. (2018). IOHprofiler: A benchmarking and profiling tool for iterative optimization heuristics. *arXiv preprint arXiv:1810.05281*, *[cs.NE]*.

Drake, J. H., Kheiri, A., Özcan, E., & Burke, E. K. (2020). Recent advances in selection hyper-heuristics. *European Journal of Operational Research*, *285*(2), 405–428.

Eggensperger, K., Feurer, M., Hutter, F., Bergstra, J., Snoek, J., Hoos, H., & Leyton-Brown, K. (2013). Towards an empirical foundation for assessing Bayesian optimization of hyperparameters. In *NeurIPS Workshop on Bayesian Optimization in Theory and Practice (BayesOpt'13)*.

Eggensperger, K., Lindauer, M., Hoos, H. H., Hutter, F., & Leyton-Brown, K. (2018). Efficient benchmarking of algorithm configurators via model-based surrogates. *Machine Learning*, *107*(1), 15–41.

Eiben, A., Horvath, M., Kowalczyk, W., & Schut, M. (2006). Reinforcement learning for online control of evolutionary algorithms. In *International Workshop on Engineering Self-Organising Applications*, pp. 151–160. Springer.

Eimer, T., Biedenkapp, A., Hutter, F., & Lindauer, M. (2021a). Self-paced context evaluation for contextual reinforcement learning. In Meila, M., & Zhang, T. (Eds.), *Proceedings of the 38th International Conference on Machine Learning (ICML'21)*, Vol. 139 of *Proceedings of Machine Learning Research*, pp. 2948–2958. PMLR.

Eimer, T., Biedenkapp, A., Reimer, M., Adriaensen, S., Hutter, F., & Lindauer, M. (2021b). DACBench: A benchmark library for dynamic algorithm configuration. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence (IJCAI'21)*. ijcai.org.

Falkner, S., Klein, A., & Hutter, F. (2018). BOHB: Robust and efficient hyperparameter optimization at scale. In Dy, J., & Krause, A. (Eds.), *Proceedings of the 35th International Conference on Machine Learning (ICML'18)*, Vol. 80, pp. 1437–1446. Proceedings of Machine Learning Research.

Fawcett, C., Vallati, M., Hutter, F., Hoffmann, J., Hoos, H., & Leyton-Brown, K. (2014). Improved features for runtime prediction of domain-independent planners. In Chien, S., Minh, D., Fern, A., & Ruml, W. (Eds.), *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS-14)*, pp. 355–359. AAAI.

Feurer, M., & Hutter, F. (2019). Hyperparameter optimization. In Hutter, F., Kotthoff, L., & Vanschoren, J. (Eds.), *Automated Machine Learning: Methods, Systems, Challenges*, pp. 3–38. Springer. Available for free at http://automl.org/book.

Fialho, A., Da Costa, L., Schoenauer, M., & Sebag, M. (2010). Analyzing bandit-based adaptive operator selection mechanisms. *Annals of Mathematics and Artificial Intelligence*, *60*(1), 25–64.

Fink, E. (1998). How to solve it automatically: Selection among problem solving methods.. In *AIPS*, pp. 128–136.

Finn, C., Abbeel, P., & Levine, S. (2017). Model-agnostic meta-learning for fast adaptation of deep networks. In Precup, D., & Teh, Y. W. (Eds.), *Proceedings of the 34th International Conference on Machine Learning*, Vol. 70 of *Proceedings of Machine Learning Research*, pp. 1126–1135. PMLR.

Fitzgerald, T. (2021). *Real-time algorithm configuration.* Ph.D. thesis, University College Cork.

Fontoura, V. D., Pozo, A. T. R., & Santana, R. (2017). Automated design of hyper-heuristics components to solve the PSP problem with HP model. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC*, pp. 1848–1855. IEEE.

Fu, J. (2016). Deep q-networks for accelerating the training of deep neural networks. *arXiv preprint arXiv:1606.01467, [cs.LG].*

Gagliolo, M., & Schmidhuber, J. (2006). Learning dynamic algorithm portfolios. *Annals of Mathematics and Artificial Intelligence*, *47*(3-4), 295–328.

Gaspero, L. D., & Urli, T. (2012). Evaluation of a family of reinforcement learning cross-domain optimization heuristics. In Hamadi, Y., & Schoenauer, M. (Eds.), *Proceedings of the Sixth International Conference on Learning and Intelligent Optimization (LION'12)*, Vol. 7219 of *Lecture Notes in Computer Science*, pp. 384–389. Springer.

Getzelman, G., & Balaprakash, P. (2021). Learning to switch optimizers for quadratic programming. In *Asian Conference on Machine Learning*, pp. 1553–1568. PMLR.

Glover, F. W., & Kochenberger, G. A. (2006). *Handbook of metaheuristics*, Vol. 57. Springer Science & Business Media.

Gomes, C., & Selman, B. (2001). Algorithm portfolios. *Artificial Intelligence*, *126*(1-2), 43–62.

Gomoluch, P., Alrajeh, D., & Russo, A. (2019). Learning classical planning strategies with policy gradient. In *Proceedings of the International Conference on Automated Planning and Scheduling*, Vol. 29, pp. 637–645.

Gomoluch, P., Alrajeh, D., Russo, A., & Bucchiarone, A. (2020). Learning neural search policies for classical planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, Vol. 30, pp. 522–530.

Graves, A., Wayne, G., & Danihelka, I. (2014). Neural turing machines. *arXiv preprint arXiv:1410.5401, [cs.NE].*

Hall, G. T., Oliveto, P. S., & Sudholt, D. (2019). On the impact of the cutoff time on the performance of algorithm configurators. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 907–915.

Hall, G. T., Oliveto, P. S., & Sudholt, D. (2020). Fast perturbative algorithm configurators. In *International Conference on Parallel Problem Solving from Nature*, pp. 19–32. Springer.

Hallak, A., Di Castro, D., & Mannor, S. (2015). Contextual markov decision processes. *arXiv preprint arXiv:1502.02259, [stat.ML].*

Hansen, N., Finck, S., Ros, R., & Auger, A. (2009). Real-Parameter Black-Box Optimization Benchmarking 2009: Noiseless Functions Definitions. Research report RR-6829, INRIA.

Hansen, N., Müller, S. D., & Koumoutsakos, P. (2003). Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (CMA-ES). *Evolutionary Computing*, *11*(1), 1–18.

Hansen, N., & Ostermeier, A. (1996). Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaptation. In *Proceedings of IEEE international conference on evolutionary computation*, pp. 312–317. IEEE.

Hansen, S. (2016). Using deep q-learning to control optimization hyperparameters. *arXiv preprint arXiv:1602.04062, [math.OC].*

Helmert, M. (2006). The fast downward planning system. *Journal of Artificial Intelligence Research*, *26*, 191–246.

Helmert, M., Röger, G., & Karpas, E. (2011). Fast downward stone soup: A baseline for building planner portfolios. In *ICAPS-2011 Workshop on Planning and Learning (PAL)*, pp. 28–35.

Helmert, M. (2004). A planning heuristic based on causal graph analysis. In Zilberstein, S., Koehler, J., & Koenig, S. (Eds.), *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS-04)*, pp. 161–170. AAAI Press.

Helmert, M., & Geffner, H. (2008). Unifying the causal graph and additive heuristics. In Rintanen, J., Nebel, B., Beck, J. C., & Hansen, E. (Eds.), *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS-08)*, pp. 140–147. AAAI Press.

Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., & Meger, D. (2018). Deep reinforcement learning that matters. In McIlraith, S., & Weinberger, K. (Eds.), *Proceedings of the Conference on Artificial Intelligence (AAAI'18)*. AAAI Press.

Hoffmann, J., & Nebel, B. (2001). The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, *14*, 253–302.

Hoos, H. (2012). Programming by optimization. *Communications of the ACM*, *55*(2), 70–80.

Hoos, H., Kaminski, R., Lindauer, M., & Schaub, T. (2015). aspeed: Solver scheduling via answer set programming. *Theory and Practice of Logic Programming*, *15*, 117–142.

Huberman, B., Lukose, R., & Hogg, T. (1997). An economic approach to hard computational problems. *Science*, *275*, 51–54.

Hutter, F., Hoos, H., & Leyton-Brown, K. (2010). Automated configuration of mixed integer programming solvers. In Lodi, A., Milano, M., & Toth, P. (Eds.), *Proceedings of the Seventh International Conference on Integration of AI and OR Techniques in Constraint Programming (CPAIOR'10)*, Vol. 6140 of *Lecture Notes in Computer Science*, pp. 186–202. Springer.

Hutter, F., Hoos, H., & Leyton-Brown, K. (2011). Sequential model-based optimization for general algorithm configuration. In Coello, C. (Ed.), *Proceedings of the Fifth International Conference on Learning and Intelligent Optimization (LION'11)*, Vol. 6683 of *Lecture Notes in Computer Science*, pp. 507–523. Springer.

Hutter, F., Hoos, H., & Leyton-Brown, K. (2014). An efficient approach for assessing hyperparameter importance. In Xing, E., & Jebara, T. (Eds.), *Proceedings of the 31th International Conference on Machine Learning, (ICML'14)*, pp. 754–762. Omnipress.

Hutter, F., Hoos, H., Leyton-Brown, K., & Stützle, T. (2009). ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, *36*, 267–306.

Hutter, F., López-Ibánez, M., Fawcett, C., Lindauer, M., Hoos, H., Leyton-Brown, K., & Stützle, T. (2014). AClib: a benchmark library for algorithm configuration. In Pardalos, P., & Resende, M. (Eds.), *Proceedings of the Eighth International Conference on Learning and Intelligent Optimization (LION'14)*, Lecture Notes in Computer Science, pp. 36–40. Springer.

Hvarfner, C., Stoll, D., Souza, A., Nardi, L., Lindauer, M., & Hutter, F. (2022). PiBO: Augmenting Acquisition Functions with User Beliefs for Bayesian Optimization. In *International Conference on Learning Representations*.

Ichnowski, J., Jain, P., Stellato, B., Banjac, G., Luo, M., Borrelli, F., Gonzalez, J. E., Stoica, I., & Goldberg, K. (2021). Accelerating quadratic optimization with reinforcement learning. In *Thirty-Fifth Conference on Neural Information Processing Systems*.

Jacobs, R. A. (1988). Increased rates of convergence through learning rate adaptation. *Neural networks*, *1*(4), 295–307.

Jaderberg, M., Dalibard, V., Osindero, S., Czarnecki, W., Donahue, J., Razavi, A., Vinyals, O., Green, T., Dunning, I., Simonyan, K., Fernando, C., & Kavukcuoglu, K. (2017). Population based training of neural networks. *arXiv preprint arXiv:1711.09846, [cs.LG]*.

Kadioglu, S., Malitsky, Y., Sabharwal, A., Samulowitz, H., & Sellmann, M. (2011). Algorithm selection and scheduling. In Lee, J. (Ed.), *Proceedings of the Seventeenth International Conference on Principles and Practice of Constraint Programming (CP'11)*, Vol. 6876 of *Lecture Notes in Computer Science*, pp. 454–469. Springer.

Kadioglu, S., Malitsky, Y., Sellmann, M., & Tierney, K. (2010). ISAC - instance-specific algorithm configuration. In Coelho, H., Studer, R., & Wooldridge, M. (Eds.), *Proceedings of the Nineteenth European Conference on Artificial Intelligence (ECAI'10)*, pp. 751–756. IOS Press.

Kadioglu, S., Sellmann, M., & Wagner, M. (2017). Learning a reactive restart strategy to improve stochastic search. In *International Conference on Learning and Intelligent Optimization*, pp. 109–123. Springer.

Karafotias, G., Eiben, A. E., & Hoogendoorn, M. (2014). Generic parameter control with reinforcement learning. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, pp. 1319–1326.

Kerschke, P., Hoos, H. H., Neumann, F., & Trautmann, H. (2019). Automated algorithm selection: Survey and perspectives. *Evolutionary computation*, *27*(1), 3–45.

Kingma, D., & Ba, J. (2015). Adam: A method for stochastic optimization. In *Proceedings of the International Conference on Learning Representations (ICLR'15)*. Published online: `iclr.cc`.

Kleinberg, R., Leyton-Brown, K., & Lucier, B. (2017). Efficiency through procrastination: Approximately optimal algorithm configuration with runtime guarantees.. In Sierra, C. (Ed.), *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI'17)*, pp. 2023–2031.

Klink, P., D'Eramo, C., Peters, J., & Pajarinen, J. (2020). Self-paced deep reinforcement learning. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS*.

Kool, W., van Hoof, H., & Welling, M. (2018). Attention, learn to solve routing problems!. In *International Conference on Learning Representations*.

Kotthoff, L. (2014). Algorithm selection for combinatorial search problems: A survey. *AI Magazine*, *35*(3), 48–60.

Koza, J. R. (1992). *Genetic programming*, Vol. 4. MIT Press.

Lagoudakis, M., & Littman, M. (2001). Learning to select branching rules in the DPLL procedure for satisfiability. *Electronic Notes in Discrete Mathematics*, *9*, 344–359.

Lagoudakis, M. G., & Littman, M. L. (2000). Algorithm selection using reinforcement learning.. In *ICML*, pp. 511–518.

Lee, J., Hwangbo, J., Wellhausen, L., Koltun, V., & Hutter, M. (2020). Learning quadrupedal locomotion over challenging terrain. *Science in Robotics*, *5*.

Levine, S., & Abbeel, P. (2014). Learning neural network policies with guided policy search under unknown dynamics. In Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N., & Weinberger, K. (Eds.), *Proceedings of the 27th International Conference on Advances in Neural Information Processing Systems (NeurIPS'14)*, pp. 1071–1079. Curran Associates.

Leyton-Brown, K., Nudelman, E., Andrew, G., McFadden, J., & Shoham, Y. (2003). A portfolio approach to algorithm selection. In *IJCAI*, Vol. 3, pp. 1542–1543.

Li, K., & Malik, J. (2017). Learning to optimize. In *Proceedings of the International Conference on Learning Representations (ICLR'17)*.

Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., & Talwalkar, A. (2018). Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, *18*(185), 1–52.

Liao, T., de Oca, M. A. M., & Stützle, T. (2011). Computational results for an automatically tuned IPOP-CMA-ES on the CEC'05 benchmark set. Tech. rep., TR/IRIDIA/2011-022, IRIDIA, Université Libre de Bruxelles, Belgium.

Lindauer, M., Bergdoll, D., & Hutter, F. (2016). An empirical study of per-instance algorithm scheduling. In Festa, P., Sellmann, M., & Vanschoren, J. (Eds.), *Proceedings of the Tenth International Conference on Learning and Intelligent Optimization (LION'16)*, Lecture Notes in Computer Science. Springer. to appear.

Lindauer, M., Eggensperger, K., Feurer, M., Biedenkapp, A., Deng, D., Benjamins, C., Ruhkopf, T., Sass, R., & Hutter, F. (2022). SMAC3: A versatile bayesian optimization package for hyperparameter optimization. *Journal of Machine Learning Research (JMLR) – MLOSS*, *23*(54), 1–9.

Lobo, F., Lima, C. F., & Michalewicz, Z. (2007). *Parameter setting in evolutionary algorithms*, Vol. 54. Springer Science & Business Media.

López-Ibáñez, M., Dubois-Lacoste, J., Caceres, L. P., Birattari, M., & Stützle, T. (2016). The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, *3*, 43–58.

López-Ibáñez, M., Liao, T., & Stützle, T. (2012). On the anytime behavior of IPOP-CMA-ES. In *International Conference on Parallel Problem Solving from Nature*, pp. 357–366. Springer.

López-Ibánez, M., & Stützle, T. (2014). Automatically improving the anytime behaviour of optimisation algorithms. *European Journal of Operational Research*, *235*(3), 569–582.

Loshchilov, I., & Hutter, F. (2017). SGDR: Stochastic gradient descent with warm restarts. In *Proceedings of the International Conference on Learning Representations (ICLR'17)*.

Lv, K., Jiang, S., & Li, J. (2017). Learning gradient descent: Better generalization and longer horizons. In *International Conference on Machine Learning*, pp. 2247–2255. PMLR.

Maclaurin, D., Duvenaud, D., & Adams, R. (2015). Gradient-based Hyperparameter Optimization through Reversible Learning. In Bach, F., & Blei, D. (Eds.), *Proceedings of the 32nd International Conference on Machine Learning (ICML'15)*, Vol. 37, pp. 2113–2122. Omnipress.

Majid, A. (2021). Deep reinforcement learning versus evolution strategies: A comparative survey. *arXiv preprint arXiv:2110.01411, [cs.LG]*.

Manna, Z., & Waldinger, R. (1980). A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *2*(1), 90–121.

Mannor, S., Rubinstein, R. Y., & Gat, Y. (2003). The cross entropy method for fast policy search. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pp. 512–519.

Metz, L., Maheswaranathan, N., Freeman, C., Poole, B., & Sohl-Dickstein, J. (2020). Tasks, stability, architecture, and compute: Training more effective learned optimizers, and using them to train themselves. *arXiv preprint arXiv:2009.11243, [cs.LG]*.

Metz, L., Maheswaranathan, N., Nixon, J., Freeman, D., & Sohl-Dickstein, J. (2019). Understanding and correcting pathologies in the training of learned optimizers. In Chaudhuri, K., & Salakhutdinov, R. (Eds.), *Proceedings of the 36th International Conference on Machine Learning (ICML'19)*, Vol. 97. Proceedings of Machine Learning Research.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M. A., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., & Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529–533.

Moulines, E., & Bach, F. R. (2011). Non-asymptotic analysis of stochastic approximation algorithms for machine learning. In Shawe-Taylor, J., Zemel, R., Bartlett, P., Pereira, F., & Weinberger, K. (Eds.), *Proceedings of the 24th International Conference on Advances in Neural Information Processing Systems (NeurIPS'11)*, pp. 451–459. Curran Associates.

Müller, S. D., Schraudolph, N. N., & Koumoutsakos, P. D. (2002). Step size adaptation in evolution strategies using reinforcement learning. In *Proceedings of the 2002 Congress on Evolutionary Computation, CEC*, pp. 151–156. IEEE.

Nguyen, M. H., Grinsztajn, N., Guyon, I., & Sun-Hosoya, L. (2021). MetaREVEAL: RL-based meta-learning from learning curves. In *Workshop on Interactive Adaptive Learning*.

Papadimitriou, C. H. (1994). On the complexity of the parity argument and other inefficient proofs of existence. *Journal of Computer and system Sciences*, *48*(3), 498–532.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., & Lerer, A. (2017). Automatic differentiation in pytorch. In Guyon, I., von Luxburg, U., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., & Garnett, R. (Eds.), *Proceedings of the 30th International Conference on Advances in Neural Information Processing Systems (NeurIPS'17)*. Curran Associates.

Pettinger, J., & Everson, R. (2002). Controlling genetic algorithms with reinforcement learning. In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, pp. 692–692.

Pillay, N., & Qu, R. (2018). *Hyper-heuristics: theory and applications*. Springer.

Prestwich, S. (2008). Tuning local search by average-reward reinforcement learning. In Maniezzo, V., Battiti, R., & Watson, J. (Eds.), *Proceedings of the International Conference on Learning and Intelligent Optimization (LION)*, Vol. 5313 of *Lecture Notes in Computer Science*, pp. 192–205. Springer.

Probst, P., Boulesteix, A., & Bischl, B. (2019). Tunability: Importance of hyperparameters of machine learning algorithms. *Journal of Machine Learning Research*, *20*(53), 1–32.

Pushak, Y., & Hoos, H. (2020). Golden parameter search: exploiting structure to quickly configure parameters in parallel. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, pp. 245–253.

Rice, J. (1976). The algorithm selection problem. *Advances in Computers*, *15*, 65–118.

Richter, S., & Helmert, M. (2009). Preferred operators and deferred evaluation in satisficing planning. In Gerevini, A., Howe, A., Cesta, A., & Refanidis, I. (Eds.), *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS-09)*, pp. 273–280. AAAI Press.

Richter, S., Helmert, M., & Westphal, M. (2008). Landmarks revisited. In Fox, D., & Gomes, C. P. (Eds.), *Proceedings of the Twenty-third Conference on Artificial Intelligence (AAAI'08)*, pp. 975–982. AAAI Press.

Röger, G., & Helmert, M. (2010). The more, the merrier: Combining heuristic estimators for satisficing planning. In Brafman, R., Geffner, H., Hoffmann, J., & Kautz, H. (Eds.), *Working notes of the Twenty-first International Conference on Automated Planning and Scheduling (ICAPS-10), Workshop on Planning and Learning.*, pp. 246–249.

Sabar, N. R., Ayob, M., Kendall, G., & Qu, R. (2013). Grammatical evolution hyper-heuristic for combinatorial optimization problems. *IEEE Transactions on Evolutionary Computation*, *17*(6), 840–861.

Sae-Dan, W., Kessaci, M.-E., Veerapen, N., & Jourdan, L. (2020). Time-dependent automatic parameter configuration of a local search algorithm. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, GECCO '20, p. 1898–1905.

Sakurai, Y., Takada, K., Kawabe, T., & Tsuruta, S. (2010). A method to control parameters of evolutionary algorithms by using reinforcement learning. In Yétongnon, K., Dipanda, A., & Chbeir, R. (Eds.), *Proceedings of Sixth International Conference on Signal-Image Technology and Internet-Based Systems (SITIS)*, pp. 74–79. IEEE Computer Society.

Salimans, T., Ho, J., Chen, X., & Sutskever, I. (2017). Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, *[stat.ML]*.

Schaul, T., Zhang, S., & LeCun, Y. (2013). No More Pesky Learning Rates. In Dasgupta, S., & McAllester, D. (Eds.), *Proceedings of the 30th International Conference on Machine Learning (ICML'13)*. Omnipress.

Seipp, J., Sievers, S., Helmert, M., & Hutter, F. (2015). Automatic configuration of sequential planning portfolios. In Bonet, B., & Koenig, S. (Eds.), *Proceedings of the Twenty-ninth AAAI Conference on Artificial Intelligence (AAAI'15)*. AAAI Press.

Seipp, J., Sievers, S., & Hutter, F. (2014). Fast downward SMAC.. Planner abstract, IPC 2014 Planning and Learning Track.

Senior, A., Heigold, G., Ranzato, M., & Yang, K. (2013). An empirical study of learning rates in deep neural networks for speech recognition. In *Proc. of ICASSP*.

Shala, G., Biedenkapp, A., Awad, N., Adriaensen, S., Lindauer, M., & Hutter, F. (2020). Learning step-size adaptation in CMA-ES. In Bäck, T., Preuss, M., Deutz, A., Wang, H., Doerr, C., Emmerich, M., & Trautmann, H. (Eds.), *Proceedings of the Sixteenth International Conference on Parallel Problem Solving from Nature (PPSN'20)*, Lecture Notes in Computer Science, pp. 691–706. Springer.

Sharma, M., Komninos, A., López-Ibáñez, M., & Kazakov, D. (2019). Deep reinforcement learning based parameter control in differential evolution. In López-Ibáñez, M. (Ed.), *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 709–717. ACM Press.

Sievers, S., Katz, M., Sohrabi, S., Samulowitz, H., & Ferber, P. (2019). Deep learning for cost-optimal planning: Task-dependent planner selection. In Hentenryck, P. V., & Zhou, Z. (Eds.), *Proceedings of the Thirty-Third Conference on Artificial Intelligence (AAAI'19)*, pp. 7715–7723. AAAI Press.

Silver, D., Huang, A., Maddison, C., Guez, A., Sifre, L., Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., & Hassabis, D. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, *529*(7587), 484–489.

Smith, L. N. (2017). Cyclical learning rates for training neural networks. In *2017 IEEE winter conference on applications of computer vision (WACV)*, pp. 464–472. IEEE.

Snoek, J., Larochelle, H., & Adams, R. (2012). Practical Bayesian optimization of machine learning algorithms. In Bartlett, P., Pereira, F., Burges, C., Bottou, L., & Weinberger, K. (Eds.), *Proceedings of the 25th International Conference on Advances in Neural Information Processing Systems (NeurIPS'12)*, pp. 2960–2968. Curran Associates.

Speck, D., Biedenkapp, A., Hutter, F., Mattmüller, R., & Lindauer, M. (2021). Learning heuristic selection with dynamic algorithm configuration. In *Proceedings of the 31st International Conference on Automated Planning and Scheduling (ICAPS'21)*.

Stanley, K. O., Clune, J., Lehman, J., & Miikkulainen, R. (2021). Designing neural networks through neuroevolution. *Nature Machine Intelligence*, *1*, 24–35.

Stanley, K. O., & Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary Computation*, *10*, 99–127.

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction.* MIT press.

Szita, I., & Lörincz, A. (2006). Learning tetris using the noisy cross-entropy method. *Neural computation*, *18*(12), 2936–2941.

Tieleman, T., Hinton, G., et al. (2012). Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, *4*(2), 26–31.

van Hasselt, H., Guez, A., & Silver, D. (2016). Deep reinforcement learning with double q-learning. In Schuurmans, D., & Wellman, M. (Eds.), *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI'16).* AAAI Press.

van Rijn, J., & Hutter, F. (2018). Hyperparameter importance across datasets. In Guo, Y., & Farooq, F. (Eds.), *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'18)*, pp. 2367–2376. ACM Press.

van Rijn, S., Doerr, C., & Bäck, T. (2018). Towards an adaptive CMA-ES configurator. In Auger, A., Fonseca, C. M., Lourenço, N., Machado, P., Paquete, L., & Whitley, L. D. (Eds.), *Proceedings of the 15th International Conference on Parallel Problem Solving from Nature (PPSN'18)*, Vol. 11101 of *Lecture Notes in Computer Science*, pp. 54–65. Springer.

Vermetten, D., van Rijn, S., Bäck, T., & Doerr, C. (2019). Online selection of CMA-ES variants. In *Proc. of GECCO*, pp. 951–959. ACM.

Wang, Z., Hutter, F., Zoghi, M., Matheson, D., & de Feitas, N. (2016). Bayesian optimization in a billion dimensions via random embeddings. *Journal of Artificial Intelligence Research*, *55*, 361–387.

Warwicker, J. A. (2019). *On the Runtime Analysis of Selection Hyper-heuristics for Pseudo-Boolean Optimisation.* Ph.D. thesis, University of Sheffield.

Weisz, G., György, A., & Szepesvári, C. (2019). CapsAndRuns: An improved method for approximately optimal algorithm configuration. In Chaudhuri, K., & Salakhutdinov, R. (Eds.), *Proceedings of the 36th International Conference on Machine Learning (ICML'19)*, Vol. 97, pp. 6707–6715. Proceedings of Machine Learning Research.

Wessing, S., Preuss, M., & Rudolph, G. (2011). When parameter tuning actually is parameter control. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pp. 821–828.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, *8*, 229–256.

Xu, L., Hoos, H., & Leyton-Brown, K. (2010). Hydra: Automatically configuring algorithms for portfolio-based selection. In Fox, M., & Poole, D. (Eds.), *Proceedings of the Twenty-fourth AAAI Conference on Artificial Intelligence (AAAI'10)*, pp. 210–216. AAAI Press.

Xu, L., Hutter, F., Hoos, H., & Leyton-Brown, K. (2008). SATzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, *32*, 565–606.

Xu, Z., Dai, A. M., Kemp, J., & Metz, L. (2019). Learning an adaptive learning rate schedule. *arXiv preprint arXiv:1909.09712*, *[cs.LG]*.