Dynamic Algorithm Configuration: Foundation of a New Meta-Algorithmic Framework

André Biedenkapp[1] and **H. Furkan Bozkurt**[1] and **Theresa Eimer**[3] and
Frank Hutter[1,2] and **Marius Lindauer**[3]

Abstract. The performance of many algorithms in the fields of hard combinatorial problem solving, machine learning or AI in general depends on parameter tuning. Automated methods have been proposed to alleviate users from the tedious and error-prone task of manually searching for performance-optimized configurations across a set of problem instances. However, there is still a lot of untapped potential through adjusting an algorithm's parameters *online* since different parameter values can be optimal at different stages of the algorithm. Prior work showed that reinforcement learning is an effective approach to learn policies for online adjustments of algorithm parameters in a data-driven way. We extend that approach by formulating the resulting *dynamic algorithm configuration* as a contextual MDP, such that RL not only learns a policy for a single instance, but across a set of instances. To lay the foundation for studying dynamic algorithm configuration with RL in a controlled setting, we propose white-box benchmarks covering major aspects that make dynamic algorithm configuration a hard problem in practice and study the performance of various types of configuration strategies for them. On these white-box benchmarks, we show that (i) RL is a robust candidate for learning configuration policies, outperforming standard parameter optimization approaches, such as classical algorithm configuration; (ii) based on function approximation, RL agents can learn to generalize to new types of instances; and (iii) self-paced learning can substantially improve the performance by selecting a useful sequence of training instances automatically.

1 Introduction

To achieve peak performance of an algorithm, it is often crucial to tune its parameters. Manually searching for performance-optimizing parameter configurations is a complex and error prone task. General algorithm configuration tools [4, 16, 27] free users from the manual search for well-performing parameters. Such tools have been successfully applied to state-of-the-art AI algorithms of various problem domains, such as mixed integer programming [15], AI planning [12], machine learning [35], or propositional satisfiability solving [18]. One drawback of classical algorithm configuration, however, is that it only yields a fixed configuration that is used during the entire run of the optimized algorithm. It does not take into account that most AI algorithms are iterative in nature and thereby ignores that the optimal target parameter configuration may change over time.

From the field of adaptive and reactive heuristics, we already know that non-stationary parameter configurations can indeed improve the performance of algorithms substantially. To automatically obtain policies adjusting parameter configurations online, prior work showed that reinforcement learning (RL) can learn those in a data-driven way and thus the performance of a variety of different algorithms can be automatically improved [23, 32, 6, 10, 34].

Extending prior approaches to be applicable across instances, we formalize the problem of learning dynamic configuration policies of an algorithm's parameters across instance sets (in short *dynamic algorithm configuration* or *DAC*) as a contextual Markov decision process (MDP) and apply reinforcement learning (RL) to it. Our formulation of DAC as a contextual MDP allows explicit handling of instances, which we combine with the self-paced learning scheme [22] to focus on subsets of instances, facilitating faster learning of configuration policies. Furthermore, we propose white-box benchmarks explicitly designed to study dynamic algorithm configuration in a principled manner without confounding factors. On these benchmarks, we study the potential and challenges of our approach. Specifically, our contributions are as follows:

1. We formalize the dynamic configuration of algorithm parameters as a contextual MDP, taking instances into account;
2. We propose new and highly flexible white-box benchmarks that allow to study DAC for scenarios involving: (i) budget constraints, (ii) short effective sequences, (iii) noisy rewards, (iv) different degrees of homogeneity of training and testing instances, as well as (v) strong parameter interaction effects;
3. We propose to use self-paced learning to order instances from easy to complex, facilitating faster transfer learning, compared to learning on an unordered set.
4. We are the first to study dynamic algorithm configuration with reinforcement learning in a controlled setting to shed light on its strengths and weaknesses.

2 Related Work

Meta-algorithmic Frameworks The goal of algorithm selection (AS; [31]) is to learn a selection mechanism, that decides which algorithm, out of a finite set of algorithms is most suited to solve a given instance. Algorithm configuration (AC; [17]) however, not only deals with one-dimensional categorical spaces, but with high-dimensional, conditional and mixed categorical/continuous spaces. AC by itself struggles with heterogeneous instance sets (in which different configurations work best for different instances), but it can be combined with AS to search for multiple well-performing configurations and select which of these to apply to new instances [39, 20]. For

---

[1] University of Freiburg, Germany,
   email: {biedenka, bozkurf, fh}@cs.uni-freiburg.de
[2] Bosch Center for Artificial Intelligence, Germany
[3] University of Hannover, Germany, email: lastname@tnt.uni-hannover.de

each problem instance, even this more general form of per-instance algorithm configuration (PIAC) still uses stationary configurations[4]. However for different AI applications, dynamic configuration can be more powerful than static or stationary ones.

Adaptive Configurations in Practice   A prominent example for parameters that need to be dynamically adjusted is the learning rate in deep learning: a static learning rate can lead to sub-optimal training results and training times [30]. To facilitate fast training and convergence, various learning rate schedules or adaptation schemes have been proposed, but only a few are data-driven [10]. Contrary to hand-designed adaptation schemes, a learned one was much less sensitive to initial starting points. Further a learned configuration policy could generalize to new architectures and larger networks.

In the field of EAs, self-adaptive strategies can change parameters on the fly [21, 11]. These methods, however, are often tailored to one individual problem, rely on heuristics and are also only rarely learned in a data-driven fashion [32], making them applicable only to homogeneous instances. A learned (and even a random) dynamic configuration policy that adjusts the mutation strategy in differential evolution has been shown to outperform non-adaptive strategies [34].

Similarly, reactive search [5] uses handcrafted heuristics to adapt an algorithm's parameters online. To adapt heuristics to the task at hand, hyper-reactive search [3] parameterizes these reactive heuristics and applies PIAC. In contrast, we propose to not only learn which heuristic to apply, given an instance, but to learn how to configure online without the need of hand-designed reactive heuristics.

Relation to Learning to Learn   The work we present here can be seen as orthogonal to work presented under the heading of learning to learn (L2L; [2, 25, 8]). Both lines of work intend to learn optimal instantiations of algorithms. The goal of a L2L agent is to learn how to traverse a search space and how to directly modify solution candidates. In contrast, a dynamic configurator learns how a specific algorithm behaves in a search space, based on which the optimal algorithm parameters are selected;[5] modifications of solution candidates are still handled by the configured algorithm.

For example when configuring iterative optimization heuristics, DAC learns when to switch between heuristics given the observed behaviour when applying the heuristics. L2L in essence would learn or discover new heuristics and thus would directly output how to traverse through the search space.

By exploiting existing algorithms and only focusing on dynamically configuring their parameters, DAC may be more sample efficient and generalize better than directly learning algorithms entirely from data, while also preserving guarantees that hold for the existing algorithm regardless of its parameter settings.

3   DAC as Contextual MDP

Definition 3.1 (DAC: Dynamic Algorithm Configuration). *Given a parameterized algorithm A with a configuration space Θ, a probability distribution p over instances $\mathcal{I}$ (which correspond to different inputs to A), a state description $s_t \in \mathcal{S}$ of A solving an instance $i \in \mathcal{I}$ at time point t, and a cost metric $c : \Pi \times \mathcal{I} \to \mathbb{R}$ assessing*

the cost of a dynamic configuration policy $\pi \in \Pi$ on instance i (e.g., runtime to solve an instance, cost of a finally returned solution, or the empirical loss of a predictive model) the goal is to obtain a policy $\pi^* : \mathcal{S} \times \mathcal{I} \to \Theta$, that adapts a parameter configuration $\theta \in \Theta$ at time point t, given a state s_t of A solving instance i, by optimizing its cost across a distribution of instances:

$$\pi^* \in \arg\min_{\pi \in \Pi} \int_{\mathcal{I}} p(i)c(\pi, i)\, \mathrm{d}i \qquad (1)$$

Contextual MDP   We propose to formulate DAC as a contextual Markov Decision Process (MDP) $\mathcal{M}_{\mathcal{I}} := \{\mathcal{M}_i\}_{i \sim \mathcal{I}}$ with $\mathcal{M}_i :=$ $(\mathcal{S}, \mathcal{A}, \mathcal{T}_i, \mathcal{R}_i)$. The notion of context $\mathcal{I}$ induces multiple MDPs $\mathcal{M}_i$ with shared action and state spaces, but with different transition and reward functions for a given instance i sampled from a distribution $\mathcal{I}$. The MDP $\mathcal{M}_i$ is a 4-tuple, consisting of a state space $\mathcal{S}$ describing the algorithm state, an action space $\mathcal{A}$ changing the algorithm's parameter settings, a probability distribution $\mathcal{T}_i$ of algorithm state transitions, and a reward function $\mathcal{R}_i$ indicating the progress of the algorithm. Algorithms are often tasked with solving varied problem instances from the same, or similar domains. Searching for well-performing parameter settings on only one instance might lead to a strong performance on that individual instance but might not generalize to new instances. In order to facilitate generalization, we therefore explicitly take instance distributions $\mathcal{I}$ as context into account. In the following, we describe in detail how this context $\mathcal{I}$ influences parts of the individual MDPs.

State and Action Spaces   At each time-step t, in order to make informed choices about the parameter values to use, the dynamic configurator needs to be informed about the internal state s_t of the dynamically configured algorithm. Many algorithms collect various statistics that are available at each time-step. For example, a SAT solver might track how variable assignments change over time. This information could be used to inform the dynamic configurator about the algorithm's current behaviour.

The theoretically possible state space does not change when switching between instances, and is shared between all MDPs induced by the context. Thus we consider the same state features which will allow us to learn useful relations across instances. To enrich the state space, we could also add instance-specific information, so-called instance features (e.g. problem size), that could allow us to reason across instances, which could be useful in particular for heterogeneous instance sets [24, 33].

Given a state s_t, the dynamic configurator has to decide how to change the value $v \in \mathcal{A}_h$ of a parameter h or directly assign a value to that parameter, out of a range of valid choices. This gives rise to the overall action space $\mathcal{A} = \mathcal{A}_{h_1} \times \mathcal{A}_{h_2} \times \ldots \times \mathcal{A}_{h_n}$ for n parameters. The action space solely depends on the algorithm at hand and is also shared across all MDPs in $\mathcal{M}_{\mathcal{I}}$, similar to the state space.

Transition Function   The transition function describes the dynamics of the system at hand. The probability of reaching state s_{t+1} after applying action a_t in state s_t can be expressed as $p(s_{t+1}|a_t, s_t)$. For simple algorithms and a small instance space, it might be possible to derive the transition function directly from the source code of the algorithm. However, we believe that the transition function cannot be explicitly modelled for most interesting algorithms. Nevertheless, even if the dynamics are not modelled, RL can learn how to optimize policies directly from observed transitions.

---

[4]   Static configurations are unchanged throughout the solving process and are not adjusted to new instances. Stationary configurations stay constant throughout the solving process but might adapt to the instance at hand.

[5]   We emphasize that we refer to hyperparameters as algorithm parameters. The goal of DAC is not to update weights (sometimes called the parameters) of a neural network directly.
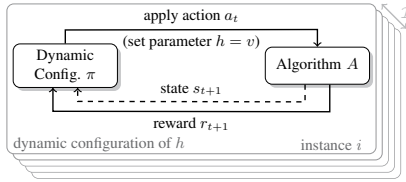
Figure 1: Dynamic configuration of parameter h of an algorithm A on a given instance $i \in \mathcal{I}$, at time-step $t \in T$. Until i is solved or a maximum budget reached, the dynamic configurator decides to change value v of parameter h, based on the internal state s_t of A on the given instance i.

Contrary to the state and action space, the transition function depends on the given instance. For example, an algorithm might be faced with different search landscapes where applying different parameter settings could lead to different state transitions.

Reward Function   In order for the dynamic configurator to learn which actions are better suited for a given state, the dynamic configurator receives a reward signal $\mathcal{R}_i(s_t, a_t) \in \mathbb{R}$. Reward functions for DAC include either sparse rewards, e.g., runtime at the end of the algorithm run, or dense rewards, e.g., distance estimations to some goal state or intermediate solution qualities, such as validation error of a partially trained neural network.

As the transition function depends on the instance at hand, so does the reward function. Transitions deemed beneficial by the dynamic configurator on one instance might become unfavorable on another instance, which is reflected by the reward signal.

Interaction of Dynamic Configurator and Algorithm   The dynamic configurator's goal is to learn a policy that can be applied to various problem instances i out of a set of instances $\mathcal{I}$, treated as the context of the MDP, see Figure 1. Given an instance i, at time-step t, the dynamic configurator applies action a_t to the algorithm, e.g., setting parameter h to value v. Given this input, the algorithm advances to state s_{t+1} producing a reward signal r_{t+1}, based on which the dynamic configurator will make its next decision. The instance stays fixed throughout the algorithm run.

Learning Policies across Instances   Given the MDP and a distribution of instances $\mathcal{I}$, the goal is to find a policy π^* from a space of possible policies Π that performs well across all instances i from a probability distribution $p(i)$ over $\mathcal{I}$. Formally,

$$\mathcal{V}_i^\pi(s_t) = \mathbb{E}\left[r_{t+1}(i) + \gamma \mathcal{V}_i^\pi(s_{t+1}) | s_{t+1} \sim \mathcal{T}_i(s_t, \pi(s_t))\right] \quad (2)$$

$$= \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}(i) | s_t = s\right] \quad (3)$$

$$\pi^* \in \arg\max_{\pi \in \Pi} \int_{\mathcal{I}} p(i) \int_{\mathcal{S}_0} \Pr(s_0) \cdot \mathcal{V}_i^\pi(s_0) \, \mathrm{d}s_0 \, \mathrm{d}i \quad (4)$$

$\mathcal{V}_i^\pi$ is the value function, giving the expected discounted future reward, starting from s_t, following policy π (i.e. advancing through $s_{t+1}, s_{t+2}, \ldots$ and adjusting the parameters according to π) on instance i with discounting-rate γ until a termination criterion is met. For finding the optimal configuration policy, we limit ourselves to the set of possible start-states $\mathcal{S}_0$ of the algorithms, which might depend on stochastic initialization or pre-processing of a given instance. In practice, we use simply a Monte-Carlo estimate by performing several runs with different seeds of the algorithm at hand.

Relation to Algorithm Configuration and Selection   This formulation of DAC allows to recover classical algorithm configuration (AC) as a special case: in AC, the optimal policy would simply always return the same action, for each state and instance. Further, this formulation also allows to recover per-instance algorithm configuration (PIAC) as a special case: in PIAC, the optimal policy would always return the same action for all states, but potentially different actions across different instances. Finally, algorithm selection (AS) is a special case of PIAC with a 1-dimensional categorical action that merely chooses out of a finite set of algorithms.

Markov Property   We argue that most developers in fact already assume the Markov property by using manually designed rule based reactive-heuristics that change parameters if certain conditions are met, independent on how these conditions were met. This behavior satisfies the Markov property since future states (behavior after adaptations) are independent of past states (prior algorithm behavior), given the present (state features).

4   Reinforcement Learning for DAC

Why Reinforcement Learning?   In algorithm configuration, a typical black-box optimizer (i) has no access to state information and (ii) sets the parameters only once in the beginning. Both shortcomings hinder classical black-box optimizers from learning optimal sequences of parameters. As a proof of concept, context-oblivious agents [1] take state information into account when selecting which action to play next. This enabled these agents to learn sequences of parameters. However, only a history of previous actions was used and employing a richer state information would enable to learn dynamic policies, that are capable of adapting to the context at hand.

RL is a promising candidate to learn DAC policies in a data driven fashion as we showed how to formulate it as a contextual MDP. It has been demonstrated that RL is capable of generalizing to new tasks given enough examples [9]. Given DAC as an MDP we can sample large numbers of episodes given enough compute resources. For small action and state spaces, RL agents can be easily implemented using table lookups, for large spaces, function approximation methods can make learning feasible. In our experiments, we evaluated ϵ-greedy Q-learning [38] in the tabular setting as well as using function approximation inspired by DQN [29].

Self-Paced Learning for Dynamic Algorithm Configuration   Since evaluating a policy on a single instance can already require quite some time (e.g., solving an NP-hard problem), evaluating a policy on all instances is often not feasible in practice. As, shown for classical algorithm configuration, using too few instances likelys result in overfitting and too many instances is too costly [17]. Therefore, we need an efficient, dynamic approach for selecting a subset of instances for training a dynamic configurator.

Similarly to curriculum learning [7], self-paced learning (SPL; [22]) aims to order tasks from easy to complex such that a configurator can transfer knowledge from easier to harder tasks, improving the overall learning. In DAC, these tasks relate to the instances the algorithm has to solve. In contrast to curriculum learning, however, the curriculum is dynamically adjusted to the pace of the learning process. In SPL, the goal is to maximize the reward achievable by a dynamic configurator on the current curriculum by jointly learning the dynamic configuration policy π and the curriculum $\mathbf{v} \in [0, 1]^{|\mathcal{I}|}$

Benchmark Outline 1: Luby

---

1 Benchmark Parameters: *minimal episode length L,*
   maximal episode length T, noise level σ;
2 $i \sim$ sample_instance;
3 Actions: $a_t \in \{0, 1, \ldots, \lfloor \log_2 T \rfloor\}$ *for all* $0 \le t \le L \le T$;
4 States: $s_t \in \{t, \text{Hist}(a_{t-4}, a_{t-3}, \ldots, a_t), i\}$;
5 for $t \in \{0, 1, \ldots, L\}$ **do**
6 $l_t \leftarrow \text{luby}(t, i)$;
7   **if** $a_t \ne l_t$ **then**
8 $\text{reward}_t \sim \mathcal{N}(-1, \sigma^2)$;
9 $L \leftarrow \min(L + |a_t - l_t|, T)$;
10   **else** $\text{reward}_t \leftarrow 0$;
11 end

Benchmark Outline 2: Sigmoid

---

1 Benchmark Parameters: *number of actions H, number of*
   action values C_h, episode length T;
2 $s_i \sim \mathcal{U}(-100, 100, H)$;
3 $p_i \sim \mathcal{N}(T/2, T/4, H)$;
4 Actions:
    $a_{h,t} \in \left\{ \frac{0}{C_h}, \frac{1}{C_h}, \ldots, \frac{C_h}{C_h} \right\} \forall\, 0 \le h < H; 0 \le t \le T$;
5 States: $s_t \in s_i \cup p_i \cup \{t\}$;
6 for $t \in \{0, 1, \ldots, T\}$ **do**
7 $\text{reward}_t \leftarrow \prod_{h=0}^{H-1} 1 - abs(sig(t, s_{i,h}, p_{i,h}) - a_{h,t})$;
8 end

(the i-th element $\mathbf{v}_i$ indicates if instance i belongs to the curriculum):

$$\max_{\pi, \mathbf{v}} \mathcal{C}(\pi, \mathbf{v}, K) = \sum_{i=1}^{|\mathcal{I}|} \mathbf{v}_i \mathcal{R}_i(\pi) - \frac{1}{K} \sum_{i=1}^{|\mathcal{I}|} \mathbf{v}_i \qquad (5)$$

where $\mathcal{R}_i(\pi)$ is the reward of following the dynamic configuration policy π on instance i. The term $-\frac{1}{K} \sum_{j=1}^{|\mathcal{I}|} \mathbf{v}_i$ regulates the curriculum size, moving from smaller to larger subsets, given a suitable increasing schedule of K.

Instead of evaluating the dynamic configurator's performance on all instances to determine the true reward $\mathcal{R}_i(\mathbf{w})$, we propose to use the expected reward as given by the $\mathcal{Q}$-function. Easy instances, for which the dynamic configurator already knows well-performing policies, will quickly lead to good rewards which will quickly be reflected in the $\mathcal{Q}$-function. This then lets us efficiently determine which instances should be included in the current curriculum as:

$$\mathbf{v}_i := \begin{cases} 1, & \text{if} \quad \mathcal{C}(\mathbf{w}, \mathbf{v}_i := 0, K) \le \mathcal{C}(\mathbf{w}, \mathbf{v}_i := 1, K) \\ 0, & \text{otherwise} \end{cases} \qquad (6)$$

where $\mathbf{v}_i := 0$ excludes the instance in computing the expected reward and $\mathbf{v}_i := 1$ includes it. In each training-iteration we greedily construct the set of training instances from scratch, such that instances are only included if they are expected to improve the reward of the dynamic configurator and then train the dynamic configurator on that set of training instances. If no instance at all is expected to improve the reward, an instance is randomly sampled.

5 White-Box Benchmarks for DAC

As discussed in the related work, various shades of DAC have already been applied to a wide range of AI problems. While they already

yielded improved performance in several applications, none of them studied the general DAC problem, and none of them employed a set of carefully-controlled benchmarks with ground truth data to allow a scientific study of when which approaches work well. To remedy this, and to enable an evaluation of DAC policies with full control over all aspects and characteristics of the environment, we propose two highly flexible, white-box benchmarks.

Our benchmarks are designed based on typical challenges in DAC on real algorithms, such as, (i) budget constraints for running an algorithm until a cutoff is reached, (ii) varying lengths of algorithm runs depending on the effectiveness of the chosen parameter settings, (iii) strong parameter interaction effects where the choice of one parameter value influences others, (iv) varying degrees of homogeneity of the instances or (v) noisy rewards because of non-deterministic behavior of algorithms. We focus here on a setting with dense rewards, since in many domains we can approximate the quality of a solution candidate, e.g., validation performance of partially trained deep neural networks or plan quality in optimal AI planning.

Luby To evaluate the ability of agents to dynamically configure algorithms with budget constraints, short effective sequences and noisy rewards across instances of varying degree of heterogeneity, we introduce benchmark *Luby* (see Benchmark Outline 1). The underlying task requires an agent to learn the values in a Luby sequence [28], which is, for example, used for restarting SAT solvers. The sequence is 1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, ...; formally, the t-th value in the sequence can be computed as:

$$l_t = \begin{cases} 2^{k-1} & \text{if } t = 2^k - 1, \\ l_{t-2^{k-1}+1} & \text{if } 2^{k-1} \le t < 2^k - 1. \end{cases} \qquad (7)$$

This gives rise to an action space for sequences of length T with $\mathcal{A} := \{0, 1, \ldots, \lfloor \log_2 T \rfloor\}$ for all time-steps $t \le T$, with the action values giving the exponents used in the Luby sequence. State information includes the time-step t, the history of actions, and an instance feature describing how the original Luby sequence is shifted.[6]

Inspired by running real algorithms, this benchmark simulates different execution times that depend on the quality of the used policy. The short horizon L (which we dub *short effective sequences*) refers to the minimal time required to solve an instance and thus determines the minimal number of configuration steps. The long term horizon T is equivalent to the cutoff (the maximal time a user wants to run an algorithm) and thus limits the total number of steps. For real algorithm runs, suboptimal parameter settings can lead to longer execution times. To reflect that, in our benchmark, L is increased by the severity of each suboptimal choice, i.e. $L \leftarrow \min(L + |a_t - l_t|, T)$.

Since most algorithms in AI are non-deterministic and do not provide a reliable reward signal, the benchmark uses a fuzzy reward $\text{N}(-1, \sigma^2)$ to penalize wrong action choices, i.e. not the true Luby value l_t at time-step t, in a stochastic way. E.g., setting σ^2 to 1.5 results in a reward where roughly $\frac{3}{4}$ of wrong action choices are correctly penalized and the rest return a false positive signal.

Finally, we task the agent to learn across a distribution of instances. To generate homogeneous instances, every m-th element of the sequence either skips or repeats an element of the true Luby sequence, leading to largely overlapping instances. To generate heterogeneous instances, we sample different starting points of the Luby sequence, leading to little overlap in the resulting instances. For details of the sampling strategies, we refer to the appendix.

---

[6] $\text{luby}(t, i = 0)$ from Benchmark Outline 1 is given in Equation 7. See appendix for details on $i \ne 0$.

Sigmoid Our second benchmark Sigmoid (see Benchmark Outline 2) allows to study DAC across instance distributions for a varying number of parameters (determined by the scalar H) and varying number of choices per parameter h (determined by C_h). Policies depend on the sampled instance i, which is described by independent sigmoid functions $sig(t; s_{i,h}, p_{i,h}) = \frac{1}{1+e^{-s_{i,h} \cdot (t - p_{i,h})}}$, each of which can be characterized through its inflection points $p_{i,h}$ and scaling factors $s_{i,h}$. The state consists of a time feature, as well as the instance information $s_{i,h}$ and $p_{i,h}$ for each parameter dimension h.

In order to be successful, for each parameter dimension h, an agent has to approximate the sigmoid $sig(t; s_{i,h}, p_{i,h})$ at each time-step t and choose the action $a_{h,t}$ closest to it. For example, for a single parameter ($H = 1$) with only two action values $a_{0,t} \in \{0, 1\}$, an agent would need to learn which value to play first and when to switch to the other value (a concrete example is given in the appendix).

In a multi-parameter setting (i.e. $H > 1$) an agent not only has to learn a simple policy switching between two actions but to learn to follow the shape of each sigmoid function that describe the instance at hand. To simulate interaction effects of the individual parameters, the reward is computed as the product of the individual approximation errors, i.e. $\text{reward}_t \leftarrow \prod_{h=0}^{H-1} 1 - abs(sig(t, s_{i,h}, p_{i,h}) - a_{h,t})$. Further, the granularity of the discretization of the action space can be adjusted by C_h, such that an agent can follow the sigmoid more or less closely, directly affecting its reward.

6 Baselines

As the simplest baseline, we present the best static policy; this defines an upper bound to the performance that could be reached by static algorithm configuration methods, such as SMAC [16]. Our agents, in contrast, can find non-stationary policies that outperform (even optimal) static choices. One could also use SMAC to learn such non-stationary policies by searching for an optimal *sequence* of parameter values (which we dub *parameter scheduling SMAC*, short PS-SMAC). For each time-step, PS-SMAC sets parameter values, making the problem exponentially harder when increasing the episode length. This relates to an optimized schedule of static parameter configurations which ignores all instance features and state information, similar to previous approaches, such as aspeed [14].

As second baseline we consider *context-oblivious agents* [1]. As state information they only take a history of actions into account. During training the agents keep track of the number of times an action lead from one state to another, as well as the average reward this transition produced. This tabular approach limits the agents to small state and action spaces. In our experiments we include *URS*, which selects an action uniformly at random during the training phase; and during the evaluation phase, URS greedily selects the best action given the observations recorded during training. We did not evaluate additional context-oblivious agents due to their limitations on our challenging benchmarks, see the appendix for details.

7 Experimental Study

Setup We used SMAC [16, 26] as a state-of-the-art algorithm configurator and black-box optimizer. We implemented URS using simple tabular 1-greedy Q-learning. Q-learning based approaches (such as URS and our RL-agents) were evaluated using a discounting factor of 0.99 and a constant learning rate of 1.0. The ϵ-greedy agent was trained using a constant $\epsilon = 0.1$. To facilitate generalization to un-

seen test instances, we include Q-learning using function approximation in the form of a double DQN [37] implemented in chainer [36].[7]

In each training iteration (10^5 in total) each agent observed a full episode. Training runs for all methods were repeated 25 times using different random seeds and each agent was evaluated after updating its policy. When evaluating on the benchmarks we performed 10 evaluation runs of which we report the mean reward. When using a fixed instance set of size 100 on *Sigmoid* we evaluated the agents once on each instance. To allow the tabular Q-learning approaches to work on this continuous state-space we round the scaling factor and inflection point to the closest integer values. We provide further details and results in the supplementary material.[8]

Effect of Short Effective Sequence Length On the *Luby* benchmark with a fixed noise level, we first study the effect of changing the short effective sequence length L, i.e., the minimal sequence length to solve an instance, see Table 1. By construction of the Luby sequence, the optimal static policy is to play the most frequent element in the sequence, i.e. the lowest value, as it makes up roughly 50% of the sequence. With increasing length of the short effective sequence, the reward achievable by this simple policy quickly approaches this 50% threshold (i.e. a reward of 0.5). Given its random behaviour, URS is only able to learn a random policy, which performs much worse than the optimal static policy. Increasing the short effective sequence length degrades PS-SMAC's result as it is only able to find a local optimum, i.e. SMAC identifies that action 0 needs to be played often but not *when* it should be played. Therefore PS-SMAC is unable to outperform the simple static policy. Contrary to the results of PS-SMAC, our ϵ-greedy RL agent is able to adjust its policies better to the presented instance, regardless of the effective sequence length, consistently achieving the best anytime and final reward, readily outperforming the best static policy. However, the greater the intended short effective sequence length, the longer it takes the ϵ-greedy agent to learn (see the appendix for details).

Stochasticity of Reward Signal To study the impact of the stochasticity, we evaluated the agents with different noise levels of the reward and a fixed short effective sequence length, see Table 2. Given very low noise-levels, URS achieves slightly better any-time performance than purely random policies, but still is far off the best static policy. With increasing noise-level, however, URS quickly degrades to a random policy. Due to its black-box nature, PS-SMAC is less affected by the noise coming from a symmetric Gaussian; since it optimizes the cumulative reward of the sequence, the noise is nearly averaged out. In contrast, the RL agent learns to average out the noise for each individual state transition. As a result, our ϵ-greedy agent is hardly more influenced by the noise-level than PS-SMAC, with a drop in AUC by 0.20 compared to PS-SMAC's drop to 0.16.

Homogeneity of Instances The observations made above hold both for more homogeneous and more heterogeneous instance distributions, see Table 1a and 1b as well as Table 2a and 2b. Only PS-SMAC is affected by the change of instance distributions; this is expected since SMAC uses a racing algorithm that assumes a certain degree of homogeneity and given its static algorithm configuration view, PS-SMAC cannot return instance-specific configurations.

---

[7] We expect that proper tuning of these hyperparameters would further improve the performance of the RL agents, but would be fairly expensive in a real application of DAC.

[8] Appendix and code: `https://github.com/automl/DAC`

	8	16	32
ϵ-greedy	**0.86** (**0.93**)	**0.72** (**0.82**)	**0.47** (**0.65**)
PS-SMAC	0.62 (0.72)	0.39 (0.40)	0.39 (0.40)
URS	0.17 (0.17)	0.17 (0.17)	0.17 (0.17)

(a) Homogeneous

	8	16	32
ϵ-greedy	**0.89** (**0.96**)	**0.75** (**0.84**)	**0.47** (**0.66**)
PS-SMAC	0.56 (0.69)	0.37 (0.39)	0.37 (0.39)
URS	0.17 (0.17)	0.17 (0.17)	0.17 (0.17)

(b) Heterogeneous

Table 1: Results on *Luby* with fuzzy rewards for $L \in \{8, 16, 32\}$ with $T = 64$ on two instance distributions and a noise factor leading to roughly 15% of the actions returning a false positive reward. The values represent the normalized area under the learning curve for 10^5 training episodes. A random policy would achieve 0.17 and the optimal one 1.0. The normalized final performance is given in brackets. The best achieved rewards are highlighted in bold. Respectively, the performance (on both sets) of the best static policy are 0.88, 0.59 and 0.52.

	$p(r_t > 0)$				
	0.01	0.08	0.15	0.20	0.25
ϵ-greedy	**0.96**	**0.92**	**0.86**	**0.81**	**0.76**
PS-SMAC	0.71	0.63	0.62	0.62	0.55
URS	0.21	0.18	0.17	0.17	0.16

(a) Homogeneous

	$p(r_t > 0)$				
	0.01	0.08	0.15	0.20	0.25
ϵ-greedy	**0.97**	**0.94**	**0.89**	**0.84**	**0.80**
PS-SMAC	0.60	0.63	0.56	0.61	0.52
URS	0.21	0.19	0.17	0.17	0.16

(b) Heterogeneous

Table 2: Sensitivity analysis of the presented agents for varying degrees of noise on *Luby*. The short effective sequence was set to 8 with a cutoff of 64. The values represent the normalized area under the learning curve for 10^5 training episodes. The corresponding standard errors and plots are contained in the supplementary material. The first columns in Table 1 correspond to the third columns here.

This effect is amplified in the experiments on *Sigmoid* where PS-SMAC cannot find a policy better than random (see Figure 2 and 3), since it does not take instance features into account and thus cannot distinguish between a positive and negative slope of the sigmoid. Roughly half the instances need completely orthogonal policies to be solved optimally, as the scaling factor is uniformly sampled.

Generalization We study the ability of generalization to unseen instances on the *Sigmoid* benchmark with a single parameter. We note that we evaluated our RL agent not only based on tabular ϵ-greedy, but also based on DQN as we expect function approximation to be crucial for generalization. For this benchmark, the best static policy is to play the action value that is closest to 0.5, as it results in the smallest approximation error on average. This is due to the sampling of the scaling factor and inflection point, where the scaling factor is uniformly sampled between -100 and 100 with the inflection point being normally distributed with a mean at $\frac{T}{2}$. In the binary case both action values are equally preferable. Learning on a distribution of instances (see Figure 2a), DQN learns faster than either tabular approaches and is able to learn an instance-dependent optimal policy, whereas the tabular ϵ-greedy agent gets stuck in a local optimum[9]. Being completely exploratory, URS does not suffer from this problem and recovers the optimal policy. The optimal static policy and PS-SMAC are unable to adapt to the task at hand, resulting in the same reward as a random non-stationary policy.

On the fixed training set (see Figure 2b), results are very similar to the case of learning on a distribution of instances (see Figure 2a); the exception are the tabular agents (URS and ϵ-greedy), which learn much faster (since the possible state-space is much smaller), but which are not able to recover the optimal policy and end up in a local optimum. Furthermore, on the test instances, these tabular agents are incapable of generalization (see Figure 2c), whereas DQN, using function approximation, is able to generalize. Our DQN can quickly generalize from observations on the training set to those on the test instances, resulting in a performance on the test set that only slightly lacks behind the performance on the training instances.

Scaling with the Number of Parameters To study the ability of agents to dynamically configure multiple, strongly interacting parameters, we evaluated them for an increasing number of parameters on the *Sigmoid* benchmark, see Figure 3.

PS-SMAC slowly approaches the same performance as the optimal static policy. For an action space of size 3, PS-SMAC and the static policy are able to achieve a better reward than a random policy. With increasing dimensionality, the static policy outperforms both tabular-agents. Our DQN agent is capable of learning instance-dependent policies even on moderately higher dimensional action spaces. With strong parameter interactions, (see reward of Benchmark Outline 2) learning policies for multiple parameters across a distribution of instances quickly becomes challenging. However, even on the highest presented dimensionality, our DQN is able to outperform the best static policy, while still improving at the end of training. Without longer training nor tuning of the agents' parameters, configuration of five very strongly coupled parameters proves very difficult for the presented agents. Parameter interactions are quite severe, as incrementing the number of parameters roughly halves the reward achievable by a random policy. If one parameter is adjusted suboptimally, this can drastically, negatively influence the overall achievable reward. All agents struggle to cope with such strong interaction effects. Our DQN agent scales best with the number of parameters, as tabular agents cannot model interaction effects.

Effect of Self-Paced Learning We study the effect of using SPL to present a learning agent with new instances ordered from easy to hard and compare it to a simple round-robin (RR) scheme, see Figure 4. SPL first performs poorly but then learns to transfer its learned policies to larger sets of instances. Compared to RR, this substantially improves the final reward, approaching the optimal reward.

8 Discussion

In practice, the feasibility of RL for DAC depends on several factors. First of all, computing state information and querying the policy to make a decision will induce some overhead. In scenarios with
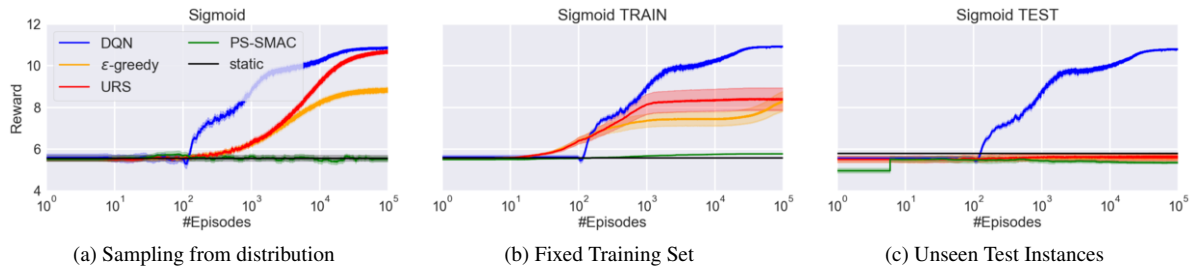
---

[9] A tuned epsilon schedule might mitigate this problem.

(a) Sampling from distribution   (b) Fixed Training Set   (c) Unseen Test Instances

Figure 2: Comparison of generalization to new instances on 1D-*Sigmoid* with binary action space and $T = 11$. The solid line represents the mean reward and the shaded area the standard error over 25 repetitions. To estimate the performance over the distribution of instances in (a), we sample 10 new random sigmoid functions for evaluation. In (b) we evaluate the agents on 100 training instances. In (c) we show the generalization capability by evaluating the agents on 100 new, prior unseen test instances, evaluating them after every training-step in (b) without additional training. A random policy could only achieve a reward of 5.5. The performance of the optimal static policy is given in black.
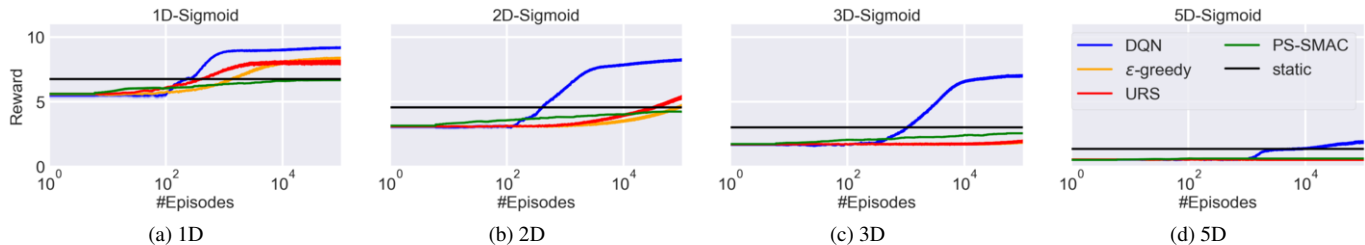

(a) 1D   (b) 2D   (c) 3D   (d) 5D

Figure 3: Comparison on higher dimensional dynamic configuration problems on $\{1, 2, 3, 5\}D$-*Sigmoid* with $|a_{h,t}| = 3$ and $T = 10$. The solid line is the average performance and the shaded area the standard error over 25 repeated experiments. Due to the interaction effects of the parameters the reward for random policies is halved when incrementing the number of parameters.
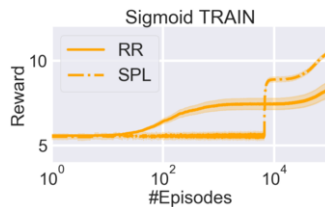

Figure 4: Comparison of training rewards for the ϵ-greedy agent using a round robin (RR) scheme against the same agent using self-paced learning (SPL) on a 1D-Sigmoid with binary actions and $T = 11$.

runtime as a performance metric, it will therefore be of importance to find a good trade-off between the granularity of making decisions and minimizing the overhead. In future work, we plan to jointly learn the optimal parameter value as well as when to adjust the parameter value using recent advances in hierarchical RL.

Furthermore, it is important to have informative state features based on which a policy can change parameter configurations. This is a known problem for RL in general. However, we argue that most AI algorithms anyway collect information for reactive heuristics which could also be used as state information in DAC. Regarding context information, there exists a plethora of work on descriptive instance features, e.g. for AI-planning [13], mixed integer programming [20, 19] or propositional satisfiability solving [40], which can be used for configuration of algorithms from their respective domains.

As always with RL, the reward function is crucial for learning a correct behavior. If an algorithm is able to approximate the quality of solution candidates well, this can be directly used as a reward signal. However, for some algorithms, the quality of solution candidates is hard to approximate and in some domains, runtime-related performance metrics are relevant, e.g., in SAT solving, which cannot be

easily approximated ahead of time. Nevertheless even for SAT solving, proxy reward functions were proposed [6] which led to well-performing SAT solvers. Therefore, we believe it viable in future work to carefully design reward functions for many AI domains.

9 Conclusion

We proposed a general framework that enables us to learn configuration policies across instances. To the best of our knowledge we are the first to formalize the dynamic algorithm configuration problem as a contextual MDP, explicitly taking problem instances into account. To study different agent types for the problem of DAC in a controlled setting, we introduced new white-box benchmarks, which enabled us to study DAC with a variety of different properties.

Using these white-box benchmarks, we demonstrated the robustness of using RL for DAC in scenarios with budget constraints, short effective sequences, noisy rewards and demonstrate the ability of RL to handle not only homogeneous but also heterogeneous instances, readily outperforming classical algorithm configuration. We showed the effectiveness of function approximation to handle more challenging state and configuration spaces. We explored the open issue of handling high-dimensional strong parameter interaction effects, where out-of-the box RL methods struggled to scale to higher dimensions. Finally we showed the efficacy of self-paced learning for dynamic algorithm configuration, ordering instances from easy to hard to facilitate faster transfer across instances.

REFERENCES

[1]   S. Adriaensen and A. Nowé, 'Towards a white box approach to auto-mated algorithm design.', in *Proc. of IJCAI'16*, pp. 554–560, (2016).

[2]   M. Andrychowicz, M. Denil, S. Gomez, M. W. Hoffman, D. Pfau, T. Schaul, B. Shillingford, and N. De Freitas, 'Learning to learn by gra-dient descent by gradient descent', in *Proc. of NeurIPS'16*, pp. 3981–3989, (2016).

[3]   C. Ansótegui, J. Pon, M. Sellmann, and K. Tierney, 'Reactive dialectic search portfolios for maxsat', in *Proc. of AAAI'17*, (2017).

[4]   C. Ansótegui, M. Sellmann, and K. Tierney, 'A gender-based genetic algorithm for the automatic configuration of algorithms', in *Proc. of CP'09*, pp. 142–157, (2009).

[5]   R. Battiti, M. Brunato, and F. Mascia, *Reactive search and intelligent optimization*, volume 45, Springer Science & Business Media, 2008.

[6]   R. Battiti and P. Campigotto, 'An investigation of reinforcement learn-ing for reactive search optimization', in *Autonomous Search*, 131–160, Springer, (2011).

[7]   Y. Bengio, J. Louradour, R. Collobert, and J. Weston, 'Curriculum learning', in *Proc. of ICML'09*, pp. 41–48, (2009).

[8]   Y. Chen, M. W. Hoffman, S. G. Colmenarejo, M. Denil, T. P. Lillicrap, M. Botvinick, and N. De Freitas, 'Learning to learn without gradient de-scent by gradient descent', in *Proc. of ICML'17*, pp. 748–756, (2017).

[9]   K. Cobbe, O. Klimov, C. Hesse, T. Kim, and J. Schulman, 'Quantify-ing generalization in reinforcement learning', in *Proc. of ICML'19*, pp. 1282–1289, (2019).

[10] C. Daniel, J. Taylor, and S. Nowozin, 'Learning step size controllers for robust neural network training', in *Proc. of AAAI'16*, (2016).

[11] B. Doerr and C. Doerr, 'Theory of parameter control for discrete black-box optimization: Provable performance gains through dynamic param-eter choices', **arXiv:1804.05650**, (2018).

[12] C. Fawcett, M. Helmert, H. Hoos, E. Karpas, G. Roger, and J. Seipp, 'Fd-autotune: Domain-specific configuration using fast-downward', in *Proc. of ICAPS'11*, (2011).

[13] C. Fawcett, M. Vallati, F. Hutter, J. Hoffmann, H. Hoos, and K. Leyton-Brown, 'Improved features for runtime prediction of domain-independent planners', in *Proc. of ICAPS'14*, pp. 355–359, (2014).

[14] H. Hoos, R. Kaminski, M. Lindauer, and T. Schaub, 'aspeed: Solver scheduling via answer set programming', *TPLP*, **15**, 117–142, (2015).

[15] F. Hutter, H. Hoos, and K. Leyton-Brown, 'Automated configuration of mixed integer programming solvers', in *Proc. of CPAIOR'10*, pp. 186–202, (2010).

[16] F. Hutter, H. Hoos, and K. Leyton-Brown, 'Sequential model-based op-timization for general algorithm configuration', in *Proc. of LION'11*, pp. 507–523, (2011).

[17] F. Hutter, H. Hoos, K. Leyton-Brown, and T. Stützle, 'ParamILS: An automatic algorithm configuration framework', *JAIR*, **36**, 267–306, (2009).

[18] F. Hutter, M. Lindauer, A. Balint, S. Bayless, H. Hoos, and K. Leyton-Brown, 'The configurable SAT solver challenge (CSSC)', *AIJ*, **243**, 1–25, (2017).

[19] F. Hutter, L. Xu, H. Hoos, and K. Leyton-Brown, 'Algorithm runtime prediction: Methods and evaluation', *AIJ*, **206**, 79–111, (2014).

[20] S. Kadioglu, Y. Malitsky, M. Sellmann, and K. Tierney, 'ISAC - instance-specific algorithm configuration', in *Proc. of ECAI'10*, pp. 751–756, (2010).

[21] G. Karafotias, M. Hoogendoorn, and A. E. Eiben, 'Parameter control in evolutionary algorithms: Trends and challenges', *IEEE Transactions on Evolutionary Computation*, **19**(2), 167–187, (2015).

[22] M. P. Kumar, B. Packer, and D. Koller, 'Self-paced learning for latent variable models', in *Proc. of NeurIPS'10*, pp. 1189–1197, (2010).

[23] M. G. Lagoudakis and M. L. Littman, 'Learning to select branching rules in the DPLL procedure for satisfiability', *Electronic Notes in Dis-crete Mathematics*, **9**, 344–359, (2001).

[24] K. Leyton-Brown, E. Nudelman, and Y. Shoham, 'Empirical hardness models: Methodology and a case study on combinatorial auctions', *Journal of ACM*, **56**(4), 1–52, (2009).

[25] K. Li and J. Malik, 'Learning to optimize', in *Proc. of ICLR'17*, (2017).

[26] M. Lindauer, K. Eggensperger, M. Feurer, S. Falkner, A. Biedenkapp, and F. Hutter. SMAC v3: Algorithm configuration in Python. `https://github.com/automl/SMAC3`, 2017.

[27] M. López-Ibáñez, J. Dubois-Lacoste, L. Perez Caceres, M. Birattari, and T. Stützle, 'The irace package: Iterated racing for automatic al-gorithm configuration', *Operations Research Perspectives*, **3**, 43–58, (2016).

[28] M. Luby, A. Sinclair, and D. Zuckerman, 'Optimal speedup of las vegas algorithms', *Information Processing Letters*, **47**(4), 173–180, (1993).

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

[30] E. Moulines and F. R. Bach, 'Non-asymptotic analysis of stochastic ap-proximation algorithms for machine learning', in *Proc. of NeurIPS'11*, pp. 451–459, (2011).

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

[32] Y. Sakurai, K. Takada, T. Kawabe, and S. Tsuruta, 'A method to control parameters of evolutionary algorithms by using reinforcement learn-ing', in *Proc. of SITIS*, pp. 74–79, (2010).

[33] M. Schneider and H. Hoos, 'Quantifying homogeneity of instance sets for algorithm configuration', in *Proc. of LION'12*, pp. 190–204, (2012).

[34] M. Sharma, A. Komninos, M. López-Ibáñez, and D. Kazakov, 'Deep re-inforcement learning based parameter control in differential evolution', in *Proc. of GECCO'19*, pp. 709–717, (2019).

[35] J. Snoek, H. Larochelle, and R. Adams, 'Practical Bayesian optimiza-tion of machine learning algorithms', in *Proc. of NeurIPS'12*, pp. 2960–2968, (2012).

[36] S. Tokui, R. Okuta, T. Akiba, Y. Niitani, T. Ogawa, S. Saito, S. Suzuki, K. Uenishi, B. Vogel, and H. V. Yamazaki, 'Chainer: A deep learning framework for accelerating the research cycle', in *Proc. of KDD'19*, pp. 2002–2011, (2019).

[37] H. van Hasselt, A. Guez, and D. Silver, 'Deep reinforcement learning with double q-learning', in *Proc. of AAAI'16*, pp. 2094–2100, (2016).

[38] C. Watkins and P. Dayan, 'Q-learning', *Machine learning*, **8**(3-4), 279–292, (1992).

[39] L. Xu, H. Hoos, and K. Leyton-Brown, 'Hydra: Automatically config-uring algorithms for portfolio-based selection', in *Proc. of AAAI'10*, pp. 210–216, (2010).

[40] L. Xu, F. Hutter, H. Hoos, and K. Leyton-Brown, 'SATzilla: Portfolio-based algorithm selection for SAT', *JAIR*, **32**, 565–606, (2008).