Towards White-box Benchmarks for Algorithm Control

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

[1] University of Freiburg   [2] Bosch Center for Artificial Intelligence

{biedenka, bozkurth, fh, lindauer}@cs.uni-freiburg.de

Abstract

The performance of many algorithms in the fields of hard combinatorial problem solving, machine learning or AI in general depends on tuned hyperparameter configurations. 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 hyperparameters online since different hyperparameters are potentially optimal at different stages of the algorithm. We formulate the problem of adjusting an algorithm's hyperparameters for a given instance on the fly as a contextual MDP, making reinforcement learning (RL) the prime candidate to solve the resulting *algorithm control problem* in a data-driven way. Furthermore, inspired by applications of algorithm configuration, we introduce new white-box benchmarks suitable to study algorithm control. We show that on short sequences, algorithm configuration is a valid choice, but that with increasing sequence length a black-box view on the problem quickly becomes infeasible and RL performs better.

1 Introduction

To achieve peak performance of an algorithm, it is often crucial to tune its hyperparameters. Manually searching for performance-optimizing hyperparameter configurations is a complex and error prone task. General algorithm configuration tools [Ansótegui *et al.*, 2009; Hutter *et al.*, 2011; López-Ibáñez *et al.*, 2016] have been proposed to free users from the manual search for well-performing hyperparameters. Such tools have been successfully applied to state-of-the-art solvers of various problem domains, such as mixed integer programming [Hutter *et al.*, 2010], AI planning [Fawcett *et al.*, 2011], machine learning [Snoek *et al.*, 2012], or propositional satisfiability solving [Hutter *et al.*, 2017].

One drawback of algorithm configuration, however, is that it only yields a fixed configuration that is used during the en-

tire solution process of the optimized algorithm. It does not take into account that most algorithms used in machine learning, satisfiability solving (SAT), AI-planning, reinforcement learning or AI in general are iterative in nature. Thereby, these tools ignore the possible induced non-stationarity of the optimal target hyperparameter configuration.

We propose a general framework to learn to control algorithms which we dub *algorithm control*. We formulate the problem of learning dynamic algorithm control policies wrt its hyperparameters as a contextual Markov decision process (MDP) and apply reinforcement learning to it. Prior work that considered online tuning of algorithms did not explicitly take problem instances into account [Battiti and Campigotto, 2011] and did not pose this problem as a reinforcement learning problem [Adriaensen and Nowé, 2016]. To address these missing, but important components, we introduce three new white-box benchmarks suitable for algorithm control. On these benchmarks we show that, using reinforcement learning, we are able to successfully learn dynamic configurations across instance sets directly from data, yielding better performance than static configurations.

Specifically, our contributions are as follows:

1. We describe controlling algorithm hyperparameters as a contextual MDP, allowing for the notion of instances;

2. We show that black-box algorithm configuration is a well-performing option for learning short policies;

3. We demonstrate that, with increasing policy length, even in the homogeneous setting, traditional algorithm configuration becomes in-feasible;

4. We propose three new white-box benchmarks that allow to study algorithm control across instances;

5. We demonstrate that we can learn dynamic policies across a set of instances showing the robustness of applying RL for algorithm control.

2 Related Work

Since algorithm configuration by itself struggles with heterogeneous instance sets (in which different configurations work best for different instances), it was combined with algorithm selection [Rice, 1976] to search for multiple well-performing configurations and select which of these to apply to new instances [Xu *et al.*, 2010; Kadioglu *et al.*, 2010]. For

---

[*]Contact Author

each problem instance, this more general form of per-instance algorithm configuration still uses fixed configurations. However for different AI applications, dynamic configurations can be more powerful than static ones. A prominent example for hyperparameters that need to be controlled over time is the learning rate in deep learning: a static learning rate can lead to sub-optimal training results and training times [Moulines and Bach, 2011]. To facilitate fast training and convergence, various learning rate schedules or adaptation schemes have been proposed [Schaul *et al.*, 2013; Kingma and Welling, 2014; Singh *et al.*, 2015; Daniel *et al.*, 2016; Loshchilov and Hutter, 2017]. Most of these methods, however, are not data-driven.

In the context of evolutionary algorithms, various on-line hyperparameter adaptation methods have been proposed [Karafotias *et al.*, 2015; Doerr and Doerr, 2018]. These methods, however, are often tailored to one individual problem or rely on heuristics. These adaptation methods are only rarely learned in a data-driven fashion [Sakurai *et al.*, 2010].

Reactive search [Battiti *et al.*, 2008] uses handcrafted heuristics to adapt an algorithms parameters online. To adapt such heuristics to the task at hand, hyper-reactive search [Ansótegui *et al.*, 2017] parameterizes these heuristic and applies per-instance algorithm configuration.

The work we present here can be seen as orthogonal to work presented under the heading of learning to learn [Andrychowicz *et al.*, 2016; Li and Malik, 2017; Chen *et al.*, 2017]. Both lines of work intend to learn optimal instantiations of algorithms during the execution of said algorithm. The goal of learning to learn, however, is to learn an update rule in the problem space directly whereas the goal of algorithm control is to indirectly influence the update by adjusting the hyperparameters used for that update. By exploiting existing manually-derived algorithms and only controlling their hyperparameters well, algorithm control may be far more sample efficient and generalize much better than directly learning algorithms entirely from data.

3 Algorithm Control

In this section we show how algorithm control (i.e., algorithm configuration per time-step) can be formulated as a sequential decision making process. Using this process, we can learn a policy to configure an algorithm's hyperparameters on the fly, using reinforcement learning (RL).

3.1 Learning to Control Algorithms

We begin by formulating *algorithm control* as a Markov Decision Process (MDP) $\mathcal{M} := (\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R})$. An MDP is a 4-tuple, consisting of a state space $\mathcal{S}$, an action space $\mathcal{A}$, a transition function $\mathcal{T}$ and a reward function $\mathcal{R}$.

State Space At each time-step t, in order to make informed choices about the hyperparameter values to choose, the controller needs to be informed about the internal state s_t of the algorithm being controlled. Many algorithms collect various statistics that are available at each time-step. For example, a SAT solver might track how many clauses are satisfied at the current time-step. Such statistics are suitable to inform the controller about the current behaviour of the algorithm.

Action Space Given a state s_t, the controller has to decide how to change the value $v \in \mathcal{A}_h$ of a hyperparameter h or directly assign a value to that hyperparameter, 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 hyperparameters of the algorithm at hand.

Transition Function The transition function describes the dynamics of the system at hand. For example, 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 assume that the transition function cannot not be explicitly modelled for interesting algorithms. Even if the dynamics are not modelled, RL can be used to learn an optimizing policy directly from observed transitions and rewards.

Reward Function In order for the controller to learn which actions are better suited for a given state, the controller receives a reward signal $\mathcal{R}_i(s_t, a_t) \to \mathbb{R}$. On many RL domains the reward is sparse, i.e., only very few state-action pairs result in an immediate reward signal. If an algorithm already estimates the distance to some goal state well, such statistics might be suitable candidates for the reward signal, with the added benefit that such a reward signal is dense.

Learning policies Given the MDP $\mathcal{M}$ the goal of the controller is to search for a policy π^* such that

$$\pi^*(s) \in \arg\max_{a \in \mathcal{A}} \mathcal{R}(s, a) + \mathcal{Q}_{\pi^*}(s, a) \qquad (1)$$

$$\mathcal{Q}_\pi(s, a) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a \right] \qquad (2)$$

where $\mathcal{Q}_\pi$ is the *action-value function*, giving the expected discounted future reward, starting from state s, applying action a and following policy π with discounting-factor γ.

3.2 Learning to Control across Instances

Algorithms are most often tasked with solving varied problem instances from the same, or similar domains. Searching for well performing hyperparameter settings on only one instance might lead to a strong performance on that instance but might not generalize to new instances. In order to facilitate generalization of algorithm control, we explicitly take problem instances into account. The formulation of algorithm control given above does not take instances into account, treating the problem of finding well performing hyperparameters as independent of the problem instance.

To allow for algorithm control across instances, we formulate the problem as a contextual Markov Decision Process $\mathcal{M}_i := (\mathcal{S}, \mathcal{A}, \mathcal{T}_i, \mathcal{R}_i)$, for a given instance $i \in \mathcal{I}$. This notion of context induces multiple MDPs with shared action and state spaces, but with different transition and reward functions. In the following, we describe how the context influences the parts of the MDP.

Context The controller's goal is to learn a policy that can be applied to various problem instances i out of a set of instances $\mathcal{I}$. We treat the instance at hand as context to the MDP.
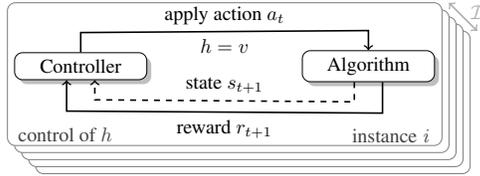
Figure 1: Control of hyperparameter h of an algorithm on a given contextual instance $i \in \mathcal{I}$, at time-step $t \in T$. Until an instance is solved or a maximum budget reached, the controller decides which value v to apply to hyperparameter h based on the internal state s_t of the algorithm, on the given instance i.

Figure 1 outlines the interaction between controller and algorithm in that setting. Given an instance i, at time-step t, the controller applies action a_t to the algorithm, i.e., setting hyperparameter 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 controller will make its next decision. The instance stays fixed during the algorithm run.

State and Action spaces The space of possible states does not change when switching between instances from the same set, and is shared between all MDPs induced by the context. Thus we consider the same state features. To enrich the state space, we could also add instance-specific information, so-called instance features such as problem size, which could be useful in particular for heterogeneous instance sets [Leyton-Brown *et al.*, 2009; Schneider and Hoos, 2012].

Similar to the state space, the action space stays fixed for all MDPs induced by the context. The action space solely depends on the algorithm at hand and is thus shared across all MDPs of the same context.

Transition Function Contrary to the state and action space, the transition function is influenced by the choice of the instance. For example, a search algorithm might be faced with completely different search spaces where applying an action could lead to different kind of states.

Reward Function As the transition function depends on the instance at hand, so does the reward function. Depending on the instance, transitions beneficial for the controller on one instance might become unfavorable or might punish the agent on another instance.

It is possible to choose a proxy reward function that is completely independent of the context, i.e., a negative reward for every step taken. This would incentivize the controller to learn a policy to quickly solve an instance which would be interesting if the real objective is to minimize runtime. However, a controller using such a reward would potentially take very long to learn a meaningful policy as the reward would not help it to easily distinguish between two observed states.

Learning policies across instances Given the MPD and a set of instances $\mathcal{I}$ the goal of the controller is to find a policy $\pi^*_{i \sim \mathcal{I}}$ such that

$$\pi^*_{i \sim \mathcal{I}}(s) \in \arg\max_{a \in \mathcal{A}} \mathcal{R}_{i \sim \mathcal{I}}(s, a) + \mathcal{Q}_{\pi^*_{i \sim \mathcal{I}}}(s, a) \quad (3)$$

$$\mathcal{Q}_{\pi_{i \sim \mathcal{I}}}(s, a) = \mathbb{E}_{\pi_{i \sim \mathcal{I}}} \left[\sum_{k=0}^{\infty} \gamma^k r_{i, t+k+1} | s_t = s, a_t = a, i \sim \mathcal{I} \right] \quad (4)$$

where $\mathcal{Q}_{\pi_{i \sim \mathcal{I}}}$ is the *action-value function*, giving the expected discounted future reward, starting from s, applying action a, following policy $\pi_{i \sim \mathcal{I}}$ on instance i with discounting-rate γ.

Relation to Algorithm Configuration and Selection This formulation of algorithm control allows to recover 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 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 space that merely chooses out of a finite set of algorithms.

4 Benchmarks

To study the algorithm control setting we use two benchmarks already proposed by Adriaensen and Nowé (2016) and introduce three new benchmarks. Our proposed benchmarks increase the complexity of the optimal policy by either increasing the action space and policy length or including instances.

Counting The first benchmark introduced by Adriaensen and Nowé (2016) requires an agent to learn a monotonically increasing sequence. The agent only receives a reward if the chosen action has been selected at the corresponding time-step. This requires the agent to learn to count, where the size of the action space is equal to the sequence length. In the original setting of Adriaensen and Nowé (2016), agents need to learn to count to five, with the optimal policy resulting in a reward of five. The state is simply given by the history of the actions chosen so far.

Fuzzy The second benchmark introduced by Adriaensen and Nowé (2016) only features two actions. Action 1 returns a fuzzy reward signal drawn from $\mathcal{N}(1, 2)$, whereas playing action 0 terminates the sequence prematurely. The maximum sequence length used in Adriaensen and Nowé (2016) is 20 with an expected reward of the optimal policy also being 20. Similar to the previous benchmark, *Fuzzy* does not include any state representation other than a history over the actions.

Luby Similar to the already presented benchmarks, the newly proposed *Luby* (see Benchmark Outline 1) does not model instances explicitly. However, it increases the complexity of learning a sequence compared to the benchmarks by Adriaensen and Nowé (2016). An agent is required to learn the values in a Luby sequence [Luby *et al.*, 1993], 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:

$$l_t = \begin{cases} 2^{k-1} & \text{if } t = 2^k - 1, \\ l_{t - 2^{k-1} + 1} & \text{if } 2^{k-1} \le t < 2^k - 1. \end{cases} \quad (5)$$

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. For such a sequence, an agent can benefit from state information about the sequence, such as the length of the sequence. For example, imagine an agent has to learn the Luby

---

Benchmark Outline 1: Luby

> **Actions:** $a_t \in \{0, 1, \ldots, \lfloor \log_2 T \rfloor\}$ *for all* $0 \le t \le T$;
> **States:** $s_t \in \{t, \text{Hist}(a_{t-4}, a_{t-3}, \ldots, a_t)\}$;
> **for** $t \in \{0, 1, \ldots, T\}$ **do**
> > **if** $a_t == \text{luby}(t)$ **then**
> > > $\text{reward}_t = 1$;
> >
> > **else**
> > > $\text{reward}_t = -1$;
> >
> > **end**
>
> **end**

Benchmark Outline 2: Sigmoid

> $s_i \sim \mathcal{U}(-100, 100)$;
> $p_i \sim \mathcal{N}(T/2, T/4)$;
> **Actions:** $a_t \in \{0, 1\}$ *for all* $0 \le t \le T$;
> **States:** $s_t \in \{t; s_i, p_i\}$;
> **for** $t \in \{0, 1, \ldots, T\}$ **do**
> > $\text{reward}_t = 1 - sig(t; s_i, p_i)$;
> > **if** $a_t == 1$ **then**
> > > $\text{reward} = sig(t; s_i, p_i)$;
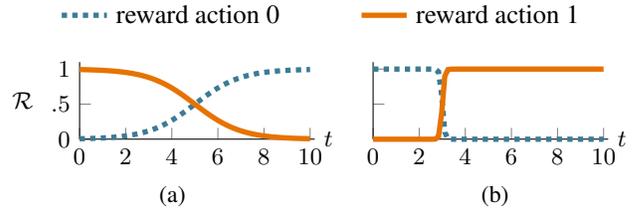> >
> > **end**
>
> **end**

Figure 2: Example rewards for Benchmark 2 with $T = 10$ on both instances, $p_{(a)} = 5, s_{(a)} = 1$ on instance (a) and $p_{(b)} = 3, s_{(b)} = 20$ on instance (b). The solid line shows the received reward when playing action 1 and the dashed line gives the reward for action 0. On the x-axis are the time steps and on the y-axis is the reward. On instance (a) it is preferable to select action 1 for the first half of the sequence whereas on instance (b) it is better to start with action 0.

Benchmark Outline 3: SigmoidMVA

> $s_i \sim \mathcal{U}(-100, 100)$;
> $p_i \sim \mathcal{N}(T/2, T/4)$;
> **Actions:** $a_t \in \left\{\frac{0}{L}, \frac{1}{L}, \ldots, \frac{L}{L}\right\}$ *for all* $0 \le t \le T$;
> **States:** $s_t \in \{t, s_i, p_i\}$;
> **for** $t \in \{0, 1, \ldots, T\}$ **do**
> > $\text{reward}_t = 1 - abs(sig(t, s_i, p_i) - a_t)$;
>
> **end**

---

sequence for length $T = 16$. Before time-step 8 the action value 3 would never have to be be played. For a real algorithm to be controlled, such a temporal feature could be encoded by the iteration number directly or some other measure of progress. The state an agent can observe therefore consists of such a time feature and a small history over the five last selected actions.

Sigmoid Benchmark Sigmoid (see Benchmark Outline 2) allows to study algorithm control across instances. Policies depend on the sampled instance i, which is described by a sigmoid $sig(t; s_i, p_i) = \frac{1}{1 + e^{-s_i \cdot (t - p_i)}}$ that can be characterized through its inflection point p_i and scaling factor s_i. The state is constructed using a time feature, as well as the instance information s_i and p_i.

At each time-step an agent has to decide between two actions. The received reward when playing action 1 is given by the function value of the sigmoid $sig(t; s_i, p_i)$ at timestep t and $1 - sig(t; s_i, p_i)$ otherwise. The scaling of the sigmoid function is sampled uniformly at random in the interval $(-100, 100)$. The sign of the scaling factor determines if an optimal policy on the instance should begin by selecting action 0 or 1. The inflection point is distributed according to $\mathcal{N}(T/2, T/4)$ and determines how often an action has to be repeated before switching to the other action. Figure 2 depicts rewards for two example instances. The sigmoid in Figure 2a is unshifted and unscaled, leading to an optimal policy of playing action 1 for the first half of the sequence and 0 for the rest of the sequence. In Figure 2b the sigmoid is shifted to the left such that the inflection point is at $t = 3$ and scaled by factor 20. The optimal policy in this case is to play action 0 for the first three steps and 1 for the rest of the sequence.

SigmoidMVA Benchmark SigmoidMVA (see Benchmark Outline 3) further increases the complexity of learning across instances by translating the setting of *Sigmoid* into a multi-valued action setting. An agent not only has to learn a simple policy switching between two actions but to learn to follow the shape of the sigmoid function used to compute the reward. The available actions an agent can choose from at each time-step are $a_t \in \left\{\frac{0}{L}, \frac{1}{L}, \ldots, \frac{L}{L}\right\}$. Note that, depending on the granularity of the discretization (determined by L) the agent can follow the sigmoid more or less closely (thereby directly affecting its reward).

5 Algorithms to be Considered

In this section we discuss the agents we want to evaluate for the task of algorithm control. We first discuss how to apply standard black-box optimization to the task of algorithm control. We then present agents that are capable of taking state information into account.

5.1 Black-Box Optimizer

In a standard black-box optimization setting, the optimizer interacts with an intended target by setting the configuration of the target at the beginning and waiting until the target returns the final reward signal. This is, e.g., the case in algorithm configuration. The same setup can be easily extended to search for sequences of configurations for online configuration of the target. Instead of setting a hyperparameter once, the optimizer would have to set a sequence of hyperparameter values, once per time-step at which the target should switch its configuration. For sequences with T such change points and large T, this drastically increases the configuration space, since the optimizer would need to treat each individual parameter as T

different hyperparameters. In addition, black-box optimizers cannot observe the state information, which are required to learn instance-specific policies.

5.2 Context-oblivious Agents

As a proof-of-concept Adriaensen and Nowé (2016) introduce context-oblivious agents that can take state information into account when selecting which action to play next. In their experiments the only state information they took into account was the history of the actions.

To move their proposed agents from a black-box setting towards a white-box setting, 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. The proposed agents include:

- URS: Selects an action uniformly at random.

- PURS: Selects a previously not selected action uniformly at random. Otherwise, actions are selected in proportion to the expected number of remaining steps.

- GR: Selects an action greedily based on the expected future reward.

During the evaluation phase, all agents greedily select the best action given the observations recorded during training.

URS and GR both are equivalent to the two extremes of ϵ-greedy Q-learning [Watkins and Dayan, 1992], with $\epsilon = 1$ and $\epsilon = 0$ respectively. PURS leverages information about the expected trajectory length, but it does not include the observed reward signal in the decision making process. For tasks where every execution path has the same length (e.g. *Counting*, *Luby*, *Sigmoid* and *SigmoidMVA*), PURS would fail to produce a policy other than a uniform random one. Further, when using PURS, we need to have some prior knowledge if shorter or longer trajectories should be preferred. For example on benchmarks like *Fuzzy*, PURS is only able to find a meaningful policy if we know that longer sequences produce better rewards.

5.3 Reinforcement Learning

Reinforcement learning (RL) is a promising candidate to learn algorithm control policies in a data driven fashion because we can formulate algorithm control as an MDP and we can sample a large number of episodes given enough compute resources. An RL agent repeatedly interacts with the target algorithm by choosing some configurations at a given time-step and observing the state transition as well as the reward. Then, the RL agent updates its believe state about how the target algorithm will behave when using the chosen configuration at that time-step. Through these interactions, over time, the agent can typically find a policy that yields higher rewards. For small action and state spaces, RL agents can be easily implemented using table lookups, whereas for larger spaces, function approximation methods can make learning feasible. We evaluated ϵ-greedy Q-learning [Watkins and Dayan, 1992] in the tabular setting as well as DQN using function approximations [Mnih *et al.*, 2016].

6 Experimental Study

To compare black-box optimizers, context-oblivious agents [Adriaensen and Nowé, 2016] and reinforcement learning agents for algorithm control, we evaluated various agents on the benchmarks discussed above.

6.1 Setup

We used SMAC [Hutter *et al.*, 2011] in version 3 (SMACv3 [Lindauer *et al.*, 2017]) as a state-of-the-art algorithm configurator and black-box optimizer. We implemented URS using simple tabular 1-greedy Q-learning. We decided against using PURS as it would only be applicable to *Fuzzy*, see Section 5.2.

Q-learning based approaches were evaluated using a discounting factor of 0.99. On benchmarks with stochastic reward we set the learning rate to 0.1 and to 1.0 otherwise. The ϵ-greedy agent was trained using a constant $\epsilon = 0.1$.

As *Sigmoid* has continuous state features we include Q-learning using function approximation in the form of a DQN [Mnih *et al.*, 2013] implemented in RLlib [Liang *et al.*, 2018]. We used the default configuration of the DQN in RLlib (0.6.6), i.e., a double dueling DQN where the target network is updated every 5 episodes and the exploration fraction ϵ of the DQN is linearly decreased from 1.0 to 0.02. We only changed the number of hidden units to 50 and the training batch size and the timesteps per training iteration to the episode length such that in each training iteration only one episode is observed.

In each training iteration 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 deterministic benchmarks (*Counting* and *Luby*) only one evaluation run was performed. On the other benchmarks we performed 10 evaluation runs of which we report the mean reward. When using a fixed instance set of size 100 on *Sigmoid* and *SigmoidMVA*, we evaluated the agents once on each instance.

All experiments were run on a compute cluster with nodes equipped with two Intel Xeon E5-2630v4 and 128GB memory running CentOS 7. The results on the benchmarks that do not model problem instances (*Counting*, *Fuzzy* and *Luby*) are plotted in Figure 3. The results for benchmarks with instances (*Sigmoid* and *SigmoidMVA*) are shown in Figures 4 and 5.

6.2 Results

Counting Figure 3a shows the evaluation results of SMAC, URS, as well as an ϵ-greedy agent on *Counting*. The agents are tasked with learning a policy of length $T = 5$ with $a_t = \{0, 1, 2, 3, 4\}$ for all $t \leq T$. On this simple benchmark, SMAC outperforms both other methods and learns the optimal policy after observing approximately 100 episodes. This is in contrast to Adriaensen and Nowé (2016), where on this benchmark they evaluated black-box optimization for a static policy producing constant reward 1. The ϵ-greedy agent quickly learns policies in which 4 out of 5 choices are set correctly but requires to observe approximately 900 episodes until it learns the optimal policy. URS purely exploratory behaviour prohibits quick learning of simple policies, requiring close to 10^4 episodes until it recovers the optimal policy.
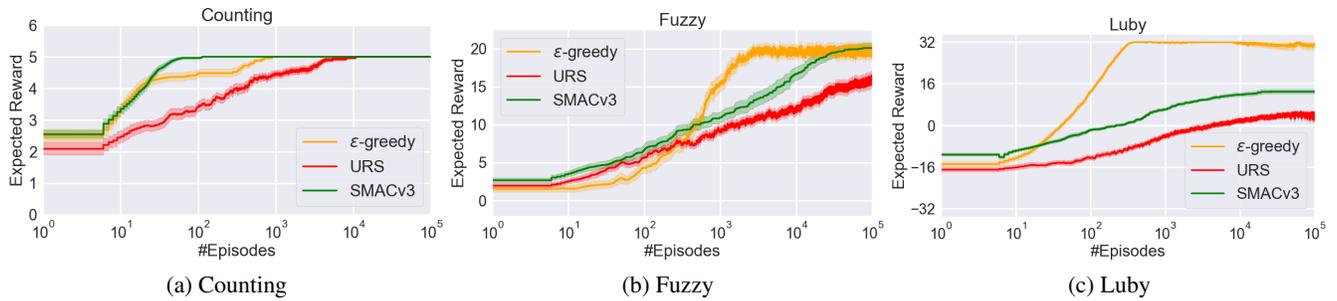
| (a) Counting | (b) Fuzzy | (c) Luby |

Figure 3: Results of SMAC, URS and tabular ϵ-greedy Q-learning, on a set of discrete benchmarks. The x-axis depicts the number of episodes seen during training and the y-axis the gained reward. The lines depict the gained reward for each agent when evaluating it after the given number of training episodes with the solid line representing the mean reward over 25 repetitions and the shaded area the standard error. The presented lines are smoothed over a window of size 10. The results in (a) are obtained on *Counting*. The results in (b) stem from *Fuzzy* and the results in (c) depict the results on *Luby*.

Fuzzy The results of the agents' behaviours on *Fuzzy* with $T = 20$ are presented in Figure 3b and extend the findings of Adriaensen and Nowé (2016). In such a noisy setting, ϵ-greedy Q-learning is faster than SMAC in learning the optimal policy, approaching it after roughly 2 000 episodes. However, SMAC is still able to learn the optimal policy after more than 24 000 episodes. URS has still not learned the optimal policy after 10^5 episodes, only learning a policy that chooses action 1 approximately 16 times before choosing action 0.

Luby Learning the optimal policy for *Luby* requires the agent to learn a policy of length $T = 32$ with $a_t = \{0, 1, 2, 3, 4, 5\}$ for all $t \leq T$. The ϵ-greedy agent already learns the optimal policy after observing about 200 episodes. In roughly the same amount of episodes SMAC found a policy in which half of the choices are set correctly, and after observing 10^5 episodes it is able to find a policy that selects roughly 70% of the actions correctly. URS was roughly 10 times slower in learning a policy that achieves a reward of 0, selecting half of the actions correctly after roughly $2 \cdot 10^3$ episodes (see Figure 3c); it found its final performance 100 times slower than SMAC.

Sigmoid Results considering instances are shown in Figure 4. In this setting the agents have to learn to adapt their policies to the presented instance, with each policy of length $T = 11$ and $a_t = \{0, 1\} \forall t \leq T$. For each episode an instance can be either directly sampled or taken out of a set of instances stemming from the same distribution. Therefore an agent that learns policies dependent on the task can achieve a maximal reward of 11 and black-box optimizers can only achieve at most 6. 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.

Figure 4a shows the gained reward of the agents when randomly drawing new instances in each training iteration. We can observe that all agents received a reward of roughly 5.5 for randomly selecting which actions to play. The DQN quickly began to learn faster than either of the tabular approaches, receiving a reward of 8 before ϵ-greedy and URS begin to learn an improving policy. After roughly 10^4 training episodes the DQN learns policies that adapt to the instance at

hand, whereas the ϵ-greedy agent gets stuck in a local optimum. Due to being completely exploratory, URS does not exhibit the same behaviour and can continue to improve its policy before learning the optimal policy after roughly $2 \cdot 10^4$ training episodes. SMAC is unable to find a policy that is able to adapt to the instances at hand. This is due to the optimizer not being able to distinguish between a positive and negative slope of the sigmoid. Therefore, it cannot decide if it should start a policy with action 0 or 1 before switching to the other. Furthermore, the agent does not know the inflection point and can only guess when to switch from one action to the other.

It is most often not the case that we have an entire distribution of instances at our disposal, but only a finite set of instances sampled from an unknown distribution. To include this setting in our evaluation, we sampled 100 training and 100 test instances from the same distribution used before. The reported results here give the performance across this whole training or test set. On the training set, the results for the DQN as well as SMAC look very similar to the results for the distribution of instances. Both tabular agents learn much faster since the possible state-space is much smaller. However both agents get stuck in a local optimum and are unable to recover the optimal policy.

On the test instances, the tabular agents are incapable of generalization (see Figure 4c), but, using function approximation, DQN is able to generalize. At first, DQN overfits on a few training instances, before it learns a robust policy for many training instances that generalizes to the test instances.

SigmoidMVA The results on *SigmoidMVA* are shown in Figure 5. Similar to *Sigmoid*, agents need to adapt their policy to a sampled instance, however, on an extended action space of size 5. Again URS benefits from its random sampling behavior, whereas the ϵ-greedy agent needs to observe roughly 10^4 episodes before improving over a random policy. Without any state information SMAC struggles to find a meaningful policy and the DQN is capable of adjusting the policy to the instance at hand even on this higher-dimensional space.
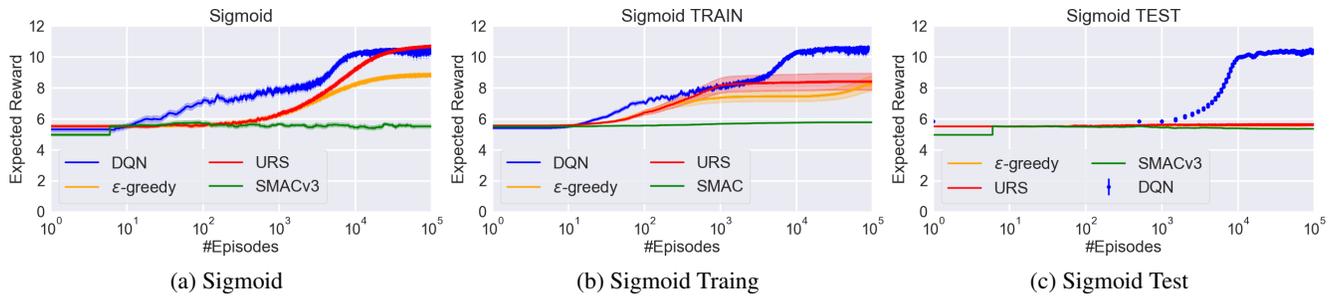
(a) Sigmoid (b) Sigmoid Traing (c) Sigmoid Test

Figure 4: Comparison of SMAC, URS and tabular ϵ-greedy Q-learning, and a DQN on *Sigmoid*. The x-axis depicts the number of episodes seen during training and the y-axis the gained reward. The lines depict the gained reward for each agent when evaluating it after the given number of training episodes with the solid line representing the mean reward over 25 training repetitions and the shaded area the standard error. To estimate the performance over the distribution of instances in (a), we sample 10 random sigmoid functions when evaluating the agents. In the case of (b) we evaluate the agents on all 100 training instances. The presented lines are smoothed over a window of size 10. (a) depicts results over a distribution of instances, (b) depicts results over a fixed set of training instances and (c) the results on prior unseen test instances, evaluated every 500 training episodes. For the tabular approaches the sate values have been rounded to the closest integer.
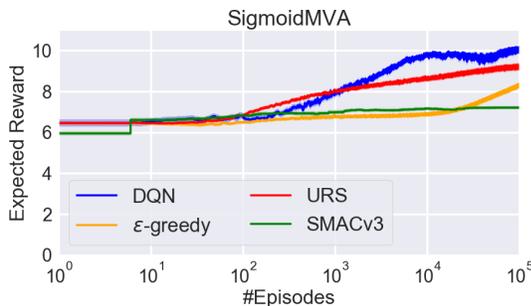

Figure 5: Comparison of the agents on *SigmoidMVA*. The x and y-axis show the number of episodes and the gained reward respectively. The lines depict the reward for each agent when evaluating it after the given number of training episodes where the line represents the mean reward over 25 training repetitions and the shaded area the standard error, smoothed over a window of size 10.

7 Conclusion

To the best of our knowledge we are the first to formalize the algorithm control problem as a contextual MDP, explicitly taking problem instances into account. To study different agents types for the problem of algorithm control with instances, we present new white-box benchmarks. Using these benchmarks, we showed that black-box optimization is a feasible candidate to learn policies for simple action spaces. With increasing complexity of the optimal policy however, black-box optimizers struggle to learn such an optimal policy. In contrast, reinforcement learning is a suitable candidate for learning more complex sequences. If heterogeneous instances are considered, black-box optimizers might struggle to learn any policy that is better than a random policy. In contrast, RL agents making use of state information are able to adapt their policies to the problem instance, which demonstrates the potential of applying RL to algorithm control.

The presented white-box benchmarks are a first step towards scenarios resembling real algorithm control for hard-combinatorial problem solvers on a set of instances. In fu-

ture work, we plan to extend our benchmarks considering mixed spaces of categorical and continuous hyperparameters and conditional dependencies. Furthermore, we plan to train cheap-to-evaluate surrogate benchmarks based on data gathered from real algorithm runs [Eggensperger *et al.*, 2018].

References

[Adriaensen and Nowé, 2016] S. Adriaensen and A. Nowé. Towards a white box approach to automated algorithm design. In *Proc. of IJCAI'16*, pages 554–560, 2016.

[Andrychowicz *et al.*, 2016] M. Andrychowicz, M. Denil, S. Gomez, M. W. Hoffman, D. Pfau, T. Schaul, B. Shillingford, and N. De Freitas. Learning to learn by gradient descent by gradient descent. In *Proc. of NIPS'16*, pages 3981–3989, 2016.

[Ansótegui *et al.*, 2009] C. Ansótegui, M. Sellmann, and K. Tierney. A gender-based genetic algorithm for the automatic configuration of algorithms. In *Proc. of CP'09*, pages 142–157, 2009.

[Ansótegui *et al.*, 2017] C. Ansótegui, J. Pon, M. Sellmann, and K. Tierney. Reactive dialectic search portfolios for maxsat. In *Proc. of AAAI'17*, 2017.

[Battiti and Campigotto, 2011] R. Battiti and P. Campigotto. An investigation of reinforcement learning for reactive search optimization. In *Autonomous Search*, pages 131–160. Springer, 2011.

[Battiti *et al.*, 2008] Roberto Battiti, Mauro Brunato, and Franco Mascia. *Reactive search and intelligent optimization*, volume 45. Springer Science & Business Media, 2008.

[Chen *et al.*, 2017] Y. Chen, M. W. Hoffman, S. G. Colmenarejo, M. Denil, T. P. Lillicrap, M. Botvinick, and

N. De Freitas. Learning to learn without gradient descent by gradient descent. In *Proc. of ICML'17*, pages 748–756, 2017.

[Daniel *et al.*, 2016] C. Daniel, J. Taylor, and S. Nowozin. Learning step size controllers for robust neural network training. In *Proc. of AAAI'16*, 2016.

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

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

[Fawcett *et al.*, 2011] 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.

[Hutter *et al.*, 2010] F. Hutter, H. Hoos, and K. Leyton-Brown. Automated configuration of mixed integer programming solvers. In *Proc. of CPAIOR'10*, pages 186–202, 2010.

[Hutter *et al.*, 2011] F. Hutter, H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proc. of LION'11*, pages 507–523, 2011.

[Hutter *et al.*, 2017] 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.

[Kadioglu *et al.*, 2010] S. Kadioglu, Y. Malitsky, M. Sellmann, and K. Tierney. ISAC - instance-specific algorithm configuration. In *Proc. of ECAI'10*, pages 751–756, 2010.

[Karafotias *et al.*, 2015] 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.

[Kingma and Welling, 2014] D. Kingma and M. Welling. Auto-encoding variational bayes. In *Proc. of ICLR'14*, 2014.

[Leyton-Brown *et al.*, 2009] 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.

[Li and Malik, 2017] K. Li and J. Malik. Learning to optimize. In *Proc. of ICLR'17*, 2017.

[Liang *et al.*, 2018] E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, K. Goldberg, J. Gonzalez, M. Jordan, and I. Stoica. Rllib: Abstractions for distributed reinforcement learning. In *Proc. of ICML'18*, pages 3059–3068, 2018.

[Lindauer *et al.*, 2017] 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.

[López-Ibáñez *et al.*, 2016] M. López-Ibáñez, J. Dubois-Lacoste, L. Perez Caceres, M. Birattari, and T. Stützle. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016.

[Loshchilov and Hutter, 2017] I. Loshchilov and F. Hutter. Sgdr: Stochastic gradient descent with warm restarts. In *Proc. of ICLR'17*, 2017.

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

[Mnih *et al.*, 2013] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. arXiv:1312.5602, 2013.

[Mnih *et al.*, 2016] V. Mnih, A. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proc. of ICML'16*, pages 1928–1937, 2016.

[Moulines and Bach, 2011] E. Moulines and F. R. Bach. Non-asymptotic analysis of stochastic approximation algorithms for machine learning. In *Proc. of NIPS'11*, pages 451–459, 2011.

[Rice, 1976] J. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.

[Sakurai *et al.*, 2010] Y. Sakurai, K. Takada, T. Kawabe, and S. Tsuruta. A method to control parameters of evolutionary algorithms by using reinforcement learning. In *Proc. of SITIS*, pages 74–79, 2010.

[Schaul *et al.*, 2013] T. Schaul, S. Zhang, and Y. LeCun. No More Pesky Learning Rates. In *Proc. of ICML'13*, 2013.

[Schneider and Hoos, 2012] M. Schneider and H. Hoos. Quantifying homogeneity of instance sets for algorithm configuration. In *Proc. of LION'12*, pages 190–204, 2012.

[Singh *et al.*, 2015] B. Singh, S. De, Y. Zhang, T. Goldstein, and G. Taylor. Layer-specific adaptive learning rates for deep networks. In *Proc. of ICMLA'15*, pages 364–368, 2015.

[Snoek *et al.*, 2012] J. Snoek, H. Larochelle, and R. Adams. Practical Bayesian optimization of machine learning algorithms. In *Proc. of NIPS'12*, pages 2960–2968, 2012.

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

[Xu *et al.*, 2010] L. Xu, H. Hoos, and K. Leyton-Brown. Hydra: Automatically configuring algorithms for portfolio-based selection. In *Proc. of AAAI'10*, pages 210–216, 2010.