Theory-inspired Parameter Control Benchmarks for Dynamic Algorithm Configuration

André Biedenkapp
University of Freiburg
Freiburg, Germany

Nguyen Dang
University of St Andrews
St Andrews, United Kingdom

Martin S. Krejca
Sorbonne Université, CNRS, LIP6
Paris, France

Frank Hutter
University of Freiburg, Germany
Bosch Center for Artificial Intelligence

Carola Doerr
Sorbonne Université, CNRS, LIP6
Paris, France

ABSTRACT

It has long been observed that the performance of evolutionary algorithms and other randomized search heuristics can benefit from a non-static choice of the parameters that steer their optimization behavior. Mechanisms that identify suitable configurations on the fly ("parameter control") or via a dedicated training process ("dynamic algorithm configuration") are thus an important component of modern evolutionary computation frameworks. Several approaches to address the dynamic parameter setting problem exist, but we barely understand which ones to prefer for which applications. As in classical benchmarking, problem collections with a known ground truth can offer very meaningful insights in this context. Unfortunately, settings with well-understood control policies are very rare.

One of the few exceptions for which we know which parameter settings minimize the expected runtime is the LeadingOnes problem. We extend this benchmark by analyzing optimal control policies that can select the parameters only from a given portfolio of possible values. This also allows us to compute optimal parameter portfolios of a given size. We demonstrate the usefulness of our benchmarks by analyzing the behavior of the DDQN reinforcement learning approach for dynamic algorithm configuration.

CCS CONCEPTS

• **Computing methodologies → Randomized search**.

ACM Reference Format:
André Biedenkapp, Nguyen Dang, Martin S. Krejca, Frank Hutter, and Carola Doerr. 2022. Theory-inspired Parameter Control Benchmarks for Dynamic Algorithm Configuration. In *Genetic and Evolutionary Computation Conference (GECCO '22), July 9–13, 2022, Boston, MA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3512290.3528846

1 INTRODUCTION

It is well known that the performance of evolutionary algorithms and other black-box optimization heuristics can benefit quite significantly from a non-static choice of the (hyper-)parameters that determine their search behavior [4, 10, 18, 32, 37, 42, 47, 49]. Not only does a dynamic choice of the parameters allow tailoring the search behavior to the problem instance at hand, but it can also be used to leverage complementarity between different search strategies during the different stages of the optimization process, e.g., by moving from a global to a local generation of solution candidates.

Mechanisms to identify suitable dynamic parameter values are intensively studied since decades, see [2, 19, 38] for surveys. Most works focus on generally applicable mechanisms to control the parameters *on-the-fly*, e.g., using self-adaptation [3], success-based parameter update strategies such as the one-fifth success rule [52], co-variance matrix adaptation [32], or reinforcement learning [15] (RL). However, for many practical applications of black-box optimization techniques we also have the possibility to *learn* a parameter control policy via a dedicated training process, either because we anyway need to solve several instances of the same problem or because we can generate instances that are similar to the ones that we expect to see in the future application. Our hope is then to derive structural insight into the algorithms' behavior that can be leveraged to choose their parameters in a more informed manner, just as we are used to do it for classic parameter tuning [8, 34, 58].

The study of parameter control schemes with dedicated offline training is recently enjoying growing attention in the broader AI community, where optimization heuristics are considered an interesting application of AutoML techniques [36]. Examples include the training of a controller for the mutation strategy employed by differential evolution optimizing the CEC2015 problem collection [57] and learning to control the mutation step-size parameter of CMA-ES on the BBOB benchmarks [56]. The problem of training parameter control policies for strong performance on a distribution of instances was coined *dynamic algorithm configuration (DAC)* in [6], where it is formulated as a contextual Markov Decision Process (see Section 4.1 for details). To investigate the functioning and the performance of different DAC approaches, a dedicated library of benchmark problems, *DACBench*, was suggested in [27].

With its rich history of parameter control studies, evolutionary computation has numerous exciting benchmark problems to offer for DAC, e.g., all the problems where dynamic parameter settings have been shown to outperform static ones. One such problem that is particularly well understood is the dynamic fitness-dependent selection of the mutation rates of greedy evolutionary algorithms maximizing the LeadingOnes problem (see Section 2). In particular, we know exactly how the expected runtime of these algorithms depends on the mutation rates used during the run, and this is not

GECCO '22, July 9–13, 2022, Boston, MA, USA
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9237-2/22/07…$15.00
https://doi.org/10.1145/3512290.3528846

only in asymptotic terms, but also for concrete problem dimensions n [9, 17, 25, 60]. This feature has promoted LeadingOnes as an important benchmark for parameter control studies, both for empirical [21, 25] and for rigorously proven [20, 23, 46] results.

Our in-depth knowledge for LeadingOnes makes the problem an ideal candidate for the in-depth empirical study of the pros and cons of DAC methods: not only does the setting offer relatively fast evaluation times, but we also benefit from a ground truth against which we can compare the policies that are learned during the offline training phase. Existing DAC benchmarks that give access to ground truth either abstract away the actual optimization process and replace it with a simple surrogate or they replace problem instances with unrealistic, artificial proxies. Further, many traditional deep RL benchmarks have deterministic environments, which makes them less representative for the configuration of metaheuristics. LeadingOnes can therefore fill an important gap.

Our Contributions. We demonstrate in this work how the mutation control problem for LeadingOnes can be used to investigate existing DAC approaches and their capabilities. We evaluate a commonly used RL approach using neural networks (dubbed DDQN) and investigate how it scales with different problem dimensions.

Each problem dimension of LeadingOnes provides us with a different problem instance on which we can compare the results of the DAC process to the *ground truth,* i.e., the optimal strategy.[1] To enrich the problem collection further, we also compute optimal control policies for settings in which the algorithms are only allowed to select their parameter values from a given portfolio $\mathcal{K}$ of possible values (Table 2). These results generalize previous works of Lissovoi et al. [46], who analyzed optimal policies for the portfolios that are composed of the integers $i \in [1, k] \cap \mathbb{N}$ for $k \in \Theta(1)$.

We observe for smaller settings, in terms of problem size n and portfolio size k, that the employed DAC method is capable of learning optimal policies quickly (Section 4.3). However, increasing either n or k can drastically increase the learning difficulty, resulting in potentially sub-optimal policies or even no successful learning within the given budget and hyperparameters setting (Figure 8).

Of independent interest for the runtime analysis community are the optimal parameter portfolios (Table 1) that we compute for a number of different combinations of problem dimension n, and portfolio size k. While these optimal portfolios have a large intersection with the `initial_segment` portfolio investigated by Lissovoi et al. [46], the optimal performance achieved with this portfolio is worse than the performance achieved with the portfolio of exponentially growing values $\{2^i \mid i \in [0, k-1] \cap \mathbb{N}\}$.

Outline. In Section 2, we introduce our benchmark, consisting of the LeadingOnes problem as well as the (1+1) RLS algorithm. In Section 3, we explain how to derive optimal policies for a given portfolio. Further, we analyze these policies with respect to increasing portfolio and dimension size. In Section 4, we analyze empirically how well optimal policies can be learned when using the DDQN reinforcement learning approach. Like in Section 3, we consider different portfolios as well as increasing portfolio and dimension sizes. Last, we conclude our work in Section 5.

[1]All optimality claims made here and in the remainder of the paper are always with respect to expected runtime. This is also our primary performance measure, i.e., when we speak of the *performance* of an algorithms, we refer to the expected number of fitness evaluations made before an optimal solution is evaluated for the first time.

Algorithm 1: The $(1+1)$ RLS with state space $\mathcal{S}$, portfolio $\mathcal{K} \subseteq [0..n]$, and parameter selection policy $\pi\colon \mathcal{S} \to \mathcal{K}$, maximizing a function $f\colon \{0,1\}^n \to \mathbb{R}$. See also Section 2.

```
1 $x \leftarrow$ a sample from $\{0,1\}^n$ chosen uniformly at random;
2 for $t \in \mathbb{N}$ do
3     $s \leftarrow$ current state of the algorithm;
4     $r \leftarrow \pi(s)$;
5     $y \leftarrow \text{flip}_r(x)$;
6     if $f(y) \geq f(x)$ then $x \leftarrow y$;
```

Code and Data. Our code and results are on GitHub [7].

2 PARAMETRIZED RLS FOR LEADINGONES

We consider the optimization of the LeadingOnes problem via variants of randomized local search, which we present in the following. We note that we use, for all $a, b \in \mathbb{N}$, the notation $[a..b] := [a, b] \cap \mathbb{N}$.

Parameterized Randomized Local Search. We analyze a parameterized version of the classic randomized local search (RLS) algorithm. While RLS searches only in the direct neighborhood of a current-best solution, its parameterized cousin, the $(1+1)$ RLS (Algorithm 1), can sample solution candidates at larger distances.

The $(1+1)$ RLS maintains a single bit string (the *current solution*), denoted by x in Algorithm 1, initially drawn uniformly at random from $\{0,1\}^n$. Iteratively, the $(1+1)$ RLS generates a new sample y (the *offspring*) from the current solution x, and it replaces x with y if the the objective value $f(y)$ (its *fitness*) is at least as large as $f(x)$. The offspring y is generated by the operator flip_r (the *mutation*), which, given a parameter $r \in [0..n]$, inverts exactly r pairwise different bits in y, chosen uniformly at random from all possible r-subsets of the index set $[1..n]$. We call the parameter r of the mutation the *search radius.* In each iteration, the $(1+1)$ RLS chooses the search radius to apply based on a function π that we call a *(parameter selection) policy*, given some state of the algorithm. The policy π only returns search radii from a certain set $\mathcal{K} \subseteq [0..n]$, which we call the *portfolio* of the algorithm. Note that the portfolio $\mathcal{K}$ and the policy π are part of the input of the $(1+1)$ RLS.

Although information-rich states can prove useful [12], we only have theoretical guarantees for *fitness-dependent* policies, which use exclusively the fitness of the current solution. Doerr and Lengler [24] discuss why it is hard to derive more general bounds. Thus, we assume in this article that the policies are *fitness-dependent.*

Our key performance criterion is the number of iterations until the $(1+1)$ RLS finds a global optimum of its fitness function for the first time, i.e., the smallest $t \in \mathbb{N}$ such that x is optimal at the beginning of that iteration. We refer to this number as the algorithm's *runtime*, noting that it is a random variable.

LeadingOnes. The LeadingOnes problem is defined over bit strings of length $n \in \mathbb{N}$. It asks to maximize the number of leading 1s of a bit string; the all-1s string is the unique global maximum. Formally, $\textsc{LeadingOnes}\colon \{0,1\}^n \to [0..n], x \mapsto \sum_{i\in[n]} \prod_{j\in[i]} x_j$.

LeadingOnes is a special case of maximizing the longest prefix of agreement with a hidden target bit string $z \in \{0,1\}^n$, evaluated with respect to a hidden permutation σ that shuffles the bit positions, formally defined as $\textsc{LeadingOnes}_{z,\sigma}\colon \{0,1\}^n \to [0..n], x \mapsto$

$\max\{i \in [0..n] \mid \forall j \in [i] : x_{\sigma(j)} = z_{\sigma(j)}\}$. Since the $(1+1)$ RLS is unbiased in the sense of Lehre and Witt [45], its performance is identical on each of these problem instances and we thus restrict our attention to the classic LeadingOnes instance mentioned above.

Although LeadingOnes$_{z,\sigma}$ can be solved using $\Theta(n \log\log n)$ queries in expectation [1], this runtime cannot be achieved with unary unbiased algorithms such as the $(1+1)$ RLS. Their runtime grows at least quadratically in the dimension [45]. The same bound of $\Omega(n^2)$ also applies to all (1+1) elitist algorithms [24], of which the $(1+1)$ RLS is a representative as well. The expected runtime of the classic RLS with constant search radius 1 is $n^2/2$ [17, Theorem 5].

3 OPTIMAL POLICIES AND PORTFOLIOS FOR LEADINGONES

The exact runtime distribution for LeadingOnes is well understood for the $(1+1)$ RLS [17, Section 2.3]. Its expected runtime is, besides its initialization, entirely determined by how quickly it improves the fitness of its current solution. More formally, the most important values are the n different probabilities $(p_i)_{i\in[0..n-1]}$, where, for each $i \in [0..n-1]$, the value p_i denotes the probability that the (1+1) RLS finds a strict improvement if the current solution has fitness i. Choosing for each i the search radius so that p_i is maximized results in an $(1+1)$ RLS instance with optimal runtime on LeadingOnes.

In more detail, for each $i \in [0..n-1]$ and each $r \in [0..n]$, let $q(r,i)$ denote the probability that the $(1+1)$ RLS finds a strict improvement if the current solution has fitness i and flips exactly r bits during mutation. For LeadingOnes, it holds for all $i \in [0..n-1]$ and all $r \in [0..n]$ that [17, Section 2.3]

$$q(r,i) = \frac{r}{n} \cdot \prod\nolimits_{j\in[1..r-1]} \frac{n-i-j}{n-j}. \tag{1}$$

An important property of q that allows determining optimal policies for various portfolios of the $(1+1)$ RLS is that, for all $i \in [0..n-1]$ and $r \in [0..n-1]$, it holds that [17, Section 2.3]

$$q(r,i) \leq q(r+1,i) \text{ if and only if } i \leq (n-r)/(r+1). \tag{2}$$

In Section 3.1, we discuss what an optimal policy looks like for the well understood case when permitting *all* possible search radii from 0 to n. We refer to this setting as the *full portfolio.* Afterward, we explain in Section 3.2 how to calculate optimal policies when the portfolio does not contain all search radii, that is, when it is *restricted.* Last, in Section 3.3, we compare optimal policies of different portfolios, including the optimal one, which, given a portfolio size and a problem dimension, minimizes the expected runtime.

Generalizations. Our analyses are easily extended to the $(1+\lambda)$ RLS, the variant of the $(1+1)$ RLS generating $\lambda \in \mathbb{N}_{\geq 1}$ offspring in each iteration. For it, equation (1) looks slightly different, as it includes λ, but all other arguments work out in the same way.

3.1 Full Portfolio

In the setting of $\mathcal{K} = [0..n]$, an optimal policy π_{opt} satisfies [17, 25]

$$\pi_{\text{opt}} \colon i \mapsto \lfloor n/(i+1) \rfloor \tag{3}$$

as a direct consequence of property (2), as it can be proven that this policy chooses for each fitness i the radius r that maximizes $q(r,i)$.

Note that policy (3) is monotonically decreasing. That is, the higher the fitness of the current individual, the fewer bits are flipped. This entails that not all search radii are used. For example, for a fitness of 0, it is optimal to flip all n bits. For a fitness of 1, it is optimal to flip exactly $\lfloor n/2 \rfloor$ bits. Thus, π_{opt} skips over all search radii in the range $[\lfloor n/2 \rfloor + 1..n-1]$. We further note that using π_{opt} results in an expected runtime of about $0.39n^2$ on LeadingOnes and that using only the search radius 1 results in an expected runtime of $0.5n^2$ [17, Section 2.3]. Thus, the expected runtime of *any* portfolio with search radius 1, using an optimal policy, falls into this range.

3.2 Restricted Portfolio Sizes

For $\mathcal{K} \subsetneq [0..n]$, the optimal policy $\pi_{\text{opt}}^{(\mathcal{K})}$ strongly depends on the search radii in $\mathcal{K}$. Thus, in general, the policy cannot follow an easy formula as given by π_{opt} in policy (3) but needs to be adjusted to the specific values available in $\mathcal{K}$. Further, if $1 \notin \mathcal{K}$, then the expected runtime of an algorithm using $\mathcal{K}$ can be infinite (in particular when the probability of creating a solution with fitness $n-1$ is non-zero, as such a solution can only be improved with search radius 1). Thus, we assume in the following always that $1 \in \mathcal{K}$.

3.2.1 Determining an optimal policy. Let $i \in [0..n-1]$ denote the fitness of the current individual, and assume that $\pi_{\text{opt}}(i) \notin \mathcal{K}$. Due to property (2), q is unimodal in its first component. Thus, the best possible search radius in $\mathcal{K}$ is one of the at most two values closest to $\pi_{\text{opt}}(i)$, i.e., $\pi_{\text{opt}}^{(\mathcal{K})}(i)$ is either $r_i^{\sup} := \max\{r \in \mathcal{K} \mid r < \pi_{\text{opt}}(i)\}$ or $r_i^{\inf} := \min\{r \in \mathcal{K} \mid r > \pi_{\text{opt}}(i)\}$. Thus, it holds that

$$\pi_{\text{opt}}^{(\mathcal{K})}(i) = \arg\max\nolimits_{r\in\{r_i^{\sup}, r_i^{\inf}\}} q(r,i). \tag{4}$$

Note that this implies that $\pi_{\text{opt}}^{(\mathcal{K})}$ is monotonically decreasing, as, for all $i, j \in [0..n-1]$, $i < j$, it holds that $r_i^{\sup} \geq r_j^{\sup}$ and $r_i^{\inf} \geq r_j^{\inf}$.

Let $\mathcal{D}$ denote the vector of the elements of $\mathcal{K}$ in decreasing order. The monotonicity of equation (4) allows simplifying the calculations for $\pi_{\text{opt}}^{(\mathcal{K})}$ by only determining the fitness values for which the the probability of improvement q for two consecutive elements in $\mathcal{D}$ changes. That is, we only need to determine for all $i \in [1..|\mathcal{K}|-1]$ the largest $j \in [0..n]$ such that $q(\mathcal{D}_i, j) \geq q(\mathcal{D}_{i+1}, j)$. We call each of these $|\mathcal{K}|-1$ points j a *breaking point.* We note that breaking points do not need to be unique. Algorithm 2 provides a pseudo code for how to determine the breaking points for a given portfolio $\mathcal{K}$. Note that lines 4 to 6 can be improved by applying a binary search that returns the smallest index at which the condition from line 5 holds. This is avoided here in favor of simplicity.

Given the breaking points $(b_i)_{i\in[1..|\mathcal{K}|-1]}$ of a portfolio $\mathcal{K}$ and defining $b_0 = -1$ and $b_{|\mathcal{K}|} = n-1$, the optimal policy $\pi_{\text{opt}}^{(\mathcal{K})}$ is easily calculated by noting that, for all $i \in [0..|\mathcal{K}|]$ and all $j \in [b_i+1..b_{i+1}]$, the i-th largest value in $\mathcal{K}$ is the optimal search radius when the current individual has fitness j.

3.3 Comparing Optimal Policies

We compare different portfolios of the same size k, and we compare their resulting optimal policies calculated as stated at the end of Section 3.2.1. To this end, we consider the following four portfolios. For $n \in \mathbb{N}_{\geq 2}$ and $k \in [2..n]$, we define

- `powers_of_2` to be $\{2^i \mid 2^i \leq n \wedge i \in [0..k-1]\}$,
- `initial_segment` to be $[1..k]$,

Algorithm 2: The algorithm to compute, for a given portfolio $\mathcal{K}$ with $1 \in \mathcal{K}$, the breaking points $(b_i)_{i\in[1..|\mathcal{K}|-1]}$ of the optimal policy $\pi_{\text{opt}}^{(\mathcal{K})}$, as discussed in Section 3.2. The function q is defined in equation (1).

```
1 $\mathcal{D} \leftarrow \mathcal{K}$ in descending order;
2 $c \leftarrow 0$;
3 for $i \in [1..|\mathcal{K}| - 1]$ do
4     for $j \in [1..n]$ do
5         if $q(\mathcal{D}_i, j) < q(\mathcal{D}_{i+1}, j)$ then break the loop over $j$;
6         $c \leftarrow j$;
7     $b_i \leftarrow c$;
```

Table 1: The optimal portfolios for various sizes k, for problem sizes $n \in \{50, 100\}$, and their expected runtimes (by n^2). For $k = 8$, $n = 100$, computation timed out. See also Section 3.3.

	Optimal portfolio/Expected runtime by n^2			
k	$n = 50$		$n = 100$	
2	1, 4	0.409832	1, 4	0.409897
3	1, 2, 6	0.39568	1, 2, 6	0.395987
4	1, 2, 4, 11	0.3911372	1, 2, 4, 11	0.391403
5	1, 2, 3, 6, 17	0.3895904	1, 2, 3, 6, 16	0.389892
6	1, 2, 3, 5, 9, 21	0.3888308	1, 2, 3, 5, 9, 23	0.389109
7	1, 2, 3, 4, 6, 12, 29	0.388452	1, 2, 3, 4, 6, 11, 27	0.3887584
8	1, 2, 3, 4, 6, 9, 19, 50	0.3882052	–	–

- `evenly_spread` to be $\{i \cdot \lfloor n/k \rfloor + 1 \mid i \in [0..k-1]\}$, and
- `optimal`, which we determine by a brute-force approach over all k-subsets of n that contain the search radius 1. The portfolio with the lowest expected runtime among all of these subsets is considered `optimal`.

Note that `powers_of_2` is only defined for values k of at most $\lfloor \log_2 n \rfloor$. For any larger value of k, it is not defined. Last, note that although there is only one `optimal` *portfolio*, all *policies* discussed in this section are optimal with respect to their specified portfolio.

The portfolio `optimal`. Table 1 shows optimal portfolios for $n \in \{50, 100\}$ and for $k \in [2..8]$. For these cases, the portfolio consists of the interval $[1..\lceil k/2 \rceil]$ and of some larger values that seem to grow exponentially. That is, `optimal` is a mixture of `initial-_segment` and a variant of `powers_of_2`. Interestingly, for $k = 8$, the portfolio contains the search radius $50 = n$, which is only relevant if the current individual has a fitness of 0. Due to the uniform initialization, we see this value with 50 %, and we transition to a different state with probability 1 by flipping all bits, so that the difference between the optimal expected runtime that can be achieved with a portfolio of size $k = 8$ over that for $k = 7$ is at most 0.5. Further, `optimal` is identical for $n \in \{50, 100\}$ for $k \in \{2, 3, 4\}$. For larger k, some larger search radii change slightly. This suggests that the generals range of optimal search radii to use is only slightly affected by the problem size.

Optimal policies. Table 2 shows optimal policies (depicted as their relative breaking points) for different portfolio sizes k and problem dimensions n. For `powers_of_2` and `initial_segment`,

Table 2: The breaking points (Algorithm 2) of different portfolios (Section 3.3) of size $k \in \{3, 4\}$ for $n \in \{50, 100\}$. Each breaking point is divided by n. Recall that the breaking points refer to the portfolio sorted in descending order.

k	Portfolio	$n = 50$	$n = 100$
3	`optimal`	0.22, 0.48	0.23, 0.49
	`powers_of_2`	0.26, 0.48	0.28, 0.49
	`initial_segment`	0.3, 0.48	0.32, 0.49
	`evenly_spread`	0, 0.12	0, 0.08
4	`optimal`	0.1, 0.26, 0.48	0.12, 0.28, 0.49
	`powers_of_2`	0.14, 0.26, 0.48	0.15, 0.28, 0.49
	`initial_segment`	0.22, 0.3, 0.48	0.24, 0.32, 0.49
	`evenly_spread`	0, 0.02, 0.16	0, 0.01, 0.1

when increasing k, the portfolio is extended by adding larger search radii. This is reflected in their respective (optimal) portfolio, as the breaking points are also extended. In contrast, for `evenly_spread`, a portfolio of one size is *not* an extension of one of a smaller size. This is reflected in the breaking points, which are not extended for increasing k. For all cases of n and k depicted, `powers_of_2` and `initial_segment` share at least half of their breaking points with `optimal`. This follows also from the results of Table 1, which shows that the high overlap of `optimal` with `initial_segment` continues, whereas the one with `powers_of_2` is not that prominent for larger k. Since all portfolios except for `evenly_spread` contain at least the search radii 1 and 2, the optimal policies also utilize the full range of these radii, following policy (3). For `evenly_spread`, mostly the search radius 1 is important.

Figure 1 investigates the case of $k = 3$ for $n = 50$ more closely. We computed for all $\binom{50}{2}$ portfolios of size 3 that contain the search radius 1 the expected runtime of an optimal policy. The figure depicts cumulative data of these computations. Interestingly, the curve follows an almost linear trend, except for the last 5 %, where the increase in the expected runtime is diminishing. This suggests that choosing portfolios uniformly at random has a fair chance of resulting in a good expected runtime of its optimal policy.

In Figure 2, we take a closer look at the impact of the portfolio size k on the expected runtime. It compares the expected runtimes of all four different portfolios when using an optimal policy. Interestingly, although `initial_segment` shares a large part of its search radii with `optimal` (Table 1), the expected runtime of `powers_of_2` is better than that of `initial_segment`. This suggests that having *some* larger search radii is more beneficial than covering only small search radii. However, the comparably bad expected runtime of `evenly_spread` shows that having more than a single small search radius (e.g., 1 *and* 2) drastically improves the expected runtime.

4 ALGORITHM CONFIGURATION WITH REINFORCEMENT LEARNING

Parameter control with a dedicated offline training phase has long been studied [see e.g., 10, 39, 40, 57, 63]. Recently it gained attention in the broader AI community where *dynamic algorithm configuration* (DAC) [6] was proposed as a generalization over algorithm configuration [35] and algorithm selection [53]. In DAC, reinforcement

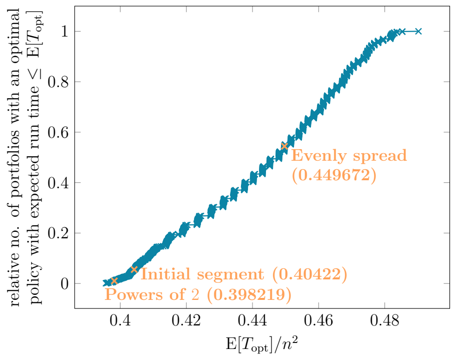


Figure 1: The cumulative fraction of how many out of *all* portfolios have at most the expected (relative) runtimes stated by the x-axis, for $n = 50$. All portfolios have cardinality exactly 3 and contain the search radius 1. Their expected runtime is determined by applying an optimal policy. See also Section 3.3.

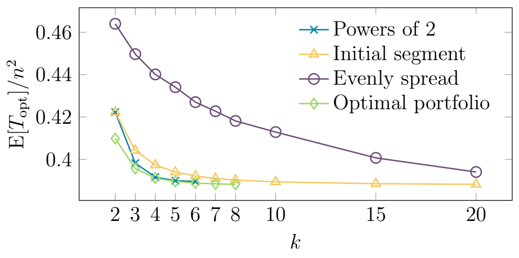


Figure 2: The expected runtimes for the optimal policies of the stated portfolios for $n = 50$. The runtime is divided by n^2. See also Section 3.3. Note that `powers_of_2` is not defined for $k > 6$. Further, we only computed `optimal` up to $k = 8$.

learning (RL) is predominantly used to learn dynamic configuration policies. In the DAC setting, our proposed benchmark is of particular interest as it readily allows us to investigate important questions such as: i) Can DAC learn optimal policies? ii) How does the choice of elements of the portfolio $\mathcal{K}$ influence the learning procedure? iii) How does the size of $\mathcal{K}$ influence the learning procedure? iv) How does the problem size influence the learning procedure?

We recap the most important definitions for DAC in Section 4.1. The experimental setup of our work is summarized in Section 4.2. Results for small portfolios $|\mathcal{K}| \in \{3, 4, 5\}$ and for fixed dimension $n = 50$ are presented in Section 4.3 and results for broader ranges of portfolio sizes and dimensions are discussed in Section 4.4.

4.1 The DAC Framework

The process of dynamically adapting hyperparameters is modeled as a contextual Markov Decision Process (cMDP) [31]. An MDP $\mathcal{M}$ is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R})$ with state space $\mathcal{S}$, action space $\mathcal{A}$, transition function $\mathcal{T}\colon \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$, and reward function $\mathcal{R}\colon \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. The transition function describes the dynamics of the process and gives the probability of reaching a state s' when playing action a in state s. Similarly, the reward function describes the reward obtained by playing action a in s. Depending on the system an MDP describes, the reward function can be stochastic. A cMDP extends this formalism through the use of so-called *context information* $i \sim \mathcal{I}$. The context influences the behavior of the reward and transition functions but leaves the state and action spaces unchanged. Thus a cMDP $\mathcal{M} = \{\mathcal{M}_i\}_{i \sim \mathcal{I}}$ is a collection of MDPs with shared state and action spaces, but with individual transition and reward functions. In DAC, the state space describes the internal behavior of an algorithm A (e.g., internal statistics of A) when running it on an instance i (i.e., the context) and the action space is given by the possible values of parameters of A. In practice, the transition and reward functions are unknown and not trivial to approximate or learn. Still, there exist solution approaches for MDPs that do not need direct access to these.

Reinforcement learning (RL) [61] has been demonstrated to be able to learn dynamic configuration policies directly from data [see e.g., 5, 6, 15, 16, 43, 44, 51, 54, 57]. In an offline learning phase, an RL agent interacts with its environment (i.e., the algorithm that is being configured) to learn which actions lead to the highest reward over multiple episodes (trajectory until a goal state or a maximal step-limit is reached). In a trial-and-error fashion, an RL agent iteratively observes the current state s_t of the environment at time t. Based on this observation it selects an action a_t which advances the environment to the next state s_{t+1} and produces a reward signal r_{t+1}. This information is sufficient to learn the value of each state and how to select the next action to maximize the expected reward.

In the commonly used Q-learning approach [64] the goal is to learn the Q-function $Q\colon \mathcal{S} \times \mathcal{A} \to \mathcal{R}$ that maps a state–action pair to the cumulative future reward that is received after playing an action a in state s. The Q-function can be learned in a typical error correction fashion. Given a state s_t and action a_t, the Q-value $Q(s_t, a_t)$ can be updated using temporal differences (TD) as

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\Bigg(\underbrace{\Big(\overbrace{r_t + \gamma \max Q(s_{t+1}, \cdot)}^{\text{TD-target}}\Big) - Q(s_t, a_t)}_{\text{TD-delta}}\Bigg),$$

where α is the *learning rate* and γ is the *discounting factor*. The TD-target is the reward r_t incurred by playing a_t in s_t together with the discounted maximal future reward. The discounting factor determines how important future rewards are when updating the Q-function. The TD-delta then describes how correct or wrong the prediction was and is used to update the Q-function accordingly. The learning rate determines the strength with which the TD-delta updates the original prediction. A reward-maximizing policy can then be defined by only using the learned Q-function as $\pi(s) = \arg\max_{a \in \mathcal{A}} Q(s, \cdot)$. For better exploration while learning, typically ϵ-greedy exploration is used, where ϵ gives the probability that an action a_t is replaced with a randomly sampled one.

Mnih et al. [48] proposed to model the Q-function as a neural network (referred to as deep Q-network) and showed that this allowed to learn Q-functions even for high-dimensional states such as frames of video games. van Hasselt et al. [62] showed that using a single network when selecting the maximizing action in the TD-target and in the prediction of the value often leads to instabilities due to overestimation during training. To mitigate this, they proposed to use a second copy of the weights of the neural network. One set is used to select the maximizing action and the other is used to predict the value. The second set of weights is kept frozen for short periods at a time and then copied over from the first set for increased stability of predictions. This extension is dubbed double deep Q-network (DDQN) and generally results in overall faster learning due to less overestimation. DDQN has been used as solution approach to DAC problems in DE [57] and AI planning [59].

4.2 Experimental Setup

Following Biedenkapp et al. [6], in our experiments we use a small DDQN with two hidden layers and 50 units each to learn the Q-function. The action space $\mathcal{A}$ is the portfolio $\mathcal{K}$. We define $s_t = f(x_t)$ and $r_t = f(x_t) - f(x_{t-1}) - 1$, where x_t is the solution found by the $(1+1)$ RLS at time step t. During the training of DDQN, we impose a cutoff time of $0.8n^2$ steps per episode to avoid wasting too much time sampling with bad policies. Recall that the expected run time of the simple setting with a constant policy $\pi\colon s \mapsto 1$ is $0.5n^2$ [17]. The episode-cutoff time for our RL training is chosen such that policies slightly worse than this trivial constant policy can still be explored during the learning phase. All DDQN agents are trained with a batch size of 2048, an ϵ-greedy value of 0.2, and a discount factor γ of 0.9998. The batch size determines how many samples are used to compute the gradients when updating the neural network. A larger batch size results in a more accurate estimation of the gradient but takes longer to compute.

It is known that hyperparameters play a crucial role in deep RL algorithms [33]. Tuning them is expensive and not trivial and many purpose-built methods exist depending on the target application [50]. It is, however, not well understood how the hyperparameters influence the learning behavior of agents, especially outside of the domain of video game playing. We built our choice of hyperparameters on prior literature using RL for dynamic tuning and adjusted batch size and γ based on results of a small prestudy (see [7]).

4.3 Results for $n = 50$

In the first set of experiments, we consider a fixed problem size of $n = 50$ as well as the three portfolio settings `initial_segment`, `powers_of_2`, and `evenly_spread` from Section 3.3. For each setting, three portfolio sizes $k \in \{3, 4, 5\}$ are considered. The aim is to study the impact of portfolio settings and portfolio sizes on DDQN's learning behaviors. For each pair of portfolio settings and sizes, a DDQN agent is trained with a budget of 1 million time steps and a walltime limit of 24 hours on an 8-core Intel Xeon E5-4650L computer (2.6 GHz). The best policy is chosen at the end of the training phase and is then evaluated and compared against the optimal policy of the same portfolio $\mathcal{K}$ via 2000 runs (per policy).

As shown in Figure 3, the performance of the DDQN policies is highly comparable to the optimal ones. DDQN is able to reach the performance of the optimal policy within 100 000 time steps in all cases. The learned policies are also quite similar to the optimal ones, with some slight discrepancy, as illustrated in Figure 4, where DDQN learned policies for two example settings (`evenly_spread` with $k = 3$, and `powers_of_2` with $k = 5$).

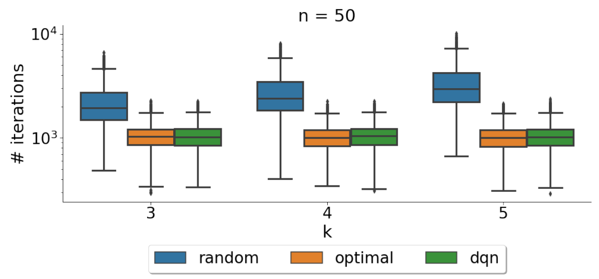


Figure 3: Performance of DDQN and optimal policies on three portfolio settings and three portfolio sizes ($n = 50$).

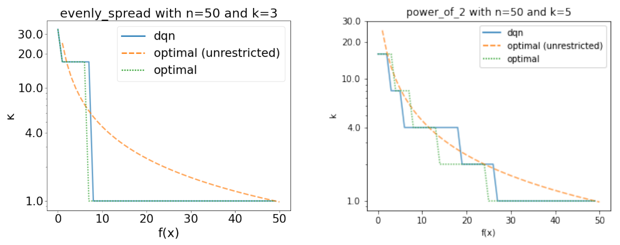
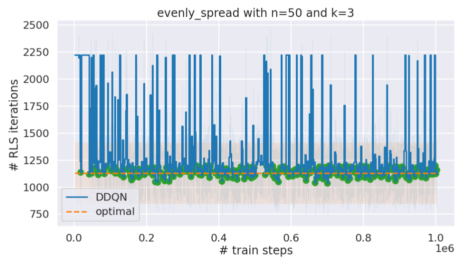


Figure 4: Two example DDQN best learned policies vs. the optimal policy for the same portfolio, and the optimal policy with unrestricted portfolio.

We now have a closer look at the training progress of each RL agent to see how different portfolio settings and portfolio sizes impact the learning behavior of DDQN. To this end, we evaluate the learned policy during each DDQN training at every 2000 time steps via 50 runs and compare it with the optimal policy. Figure 5 shows two example training progress plots of `evenly_spread` and `initial_segment`. Although DDQN frequently reaches the optimal area in both settings, there is a clear distinction between them: for `evenly_spread`, DDQN's performance constantly jumps up and down with very high variance, while for `initial_segment`, the performance progress is much smoother. To quantify these properties of the training progress, we define two metrics for each DDQN training run: (i) *hitting ratio* – the frequency of evaluations in which the expected optimal performance is reached within 0.25 % of its standard deviation; and (ii) *ruggedness* – the standard deviation of performance difference between every pair of consecutively evaluated policies. As shown in Figure 6, the RL agent gets the highest hitting ratios with `evenly_spread`, followed by `powers_of_2` and `initial_segment`. This can be explained due to the actions for `evenly_spread` being very different from each other, some of which often perform very badly in general. Such differences can result in strong signals received by the agent during the training for distinguishing between good and bad policies, which can then help speed up the learning but also causes the landscapes to be less smooth (i.e., high ruggedness) due to the large variance of performance between different policies. Similarly, `initial_segment` has the smallest difference between actions, and the RL agent has the lowest hitting ratios but smoother learning progress.



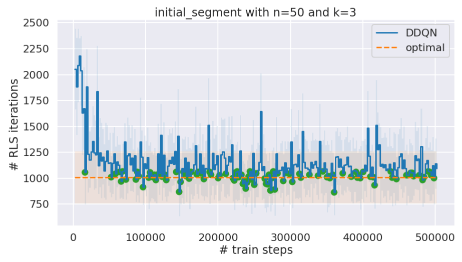


Figure 5: DDQN progress on `evenly_spread` and `initial_segment`. At the green dots, the learned policies reach $0.25\,\%$ **standard deviation of the optimal policy's performance.**

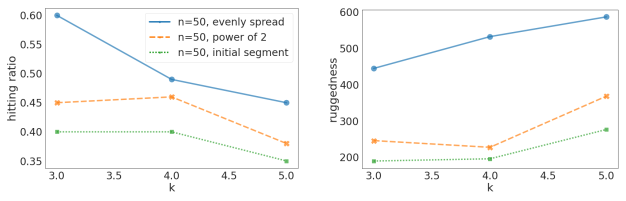


Figure 6: Hitting ratios and ruggedness of DDQN training progress for three portfolio settings ($n = 50$).

4.4 Analyzing the Impact of Portfolio Size and Problem Dimension

Figure 6 indicates a strong relation between portfolio sizes and the learning ability of DDQN agents: the larger k is, the smaller the hitting ratios. In the second set of experiments, we investigate further the impact of portfolio sizes and problem sizes on DDQN's learning behaviors. We train DDQN agents on `evenly_spread` with a wider set of portfolio sizes $k \in \{3, 4, 5, 6, 7, 8, 10, 15, 20\}$ and with two problem sizes $n \in \{50, 100\}$. For $n = 100$, we expect it to be more difficult for the RL agent to learn due to the larger episode lengths, thus, the training budget is increased to 1.4 million time steps. As shown in Figure 7, DDQN hitting ratios decrease drastically as k increases. For $n = 100$ and $k \geq 7$, the hitting ratios are very close to zero. In fact, the performance of the learned policies for $n = 100$ and $k \in \{15, 20\}$ is no longer competitive to the optimal ones, as shown in Figure 8. Looking into the detailed progress of each RL run, we find that for $k = 7$, the agent barely hits the optimal policies (only 2 times over 750 evaluations), and for $k = 15$, it has zero hitting rate.

The results so far indicate that we reach the learning limit of DDQN with the given setting. To confirm this hypothesis, we repeat

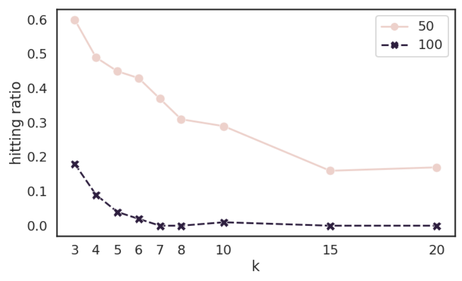
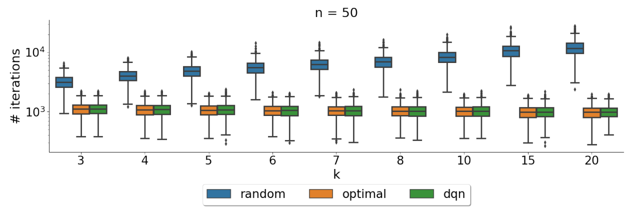


Figure 7: Hitting ratios of DDQN on `evenly_spread`, with $n \in \{50, 100\}$ and $k \in \{3, 4, 5, 6, 7, 8, 10, 15, 20\}$.



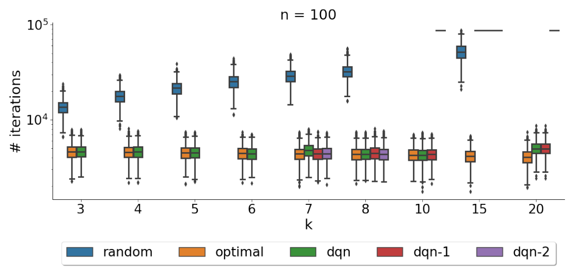


Figure 8: Performance of DDQN on `evenly_spread` setting, with $k \in \{3, 4, 5, 6, 7, 8, 10, 15, 20\}$ and $n \in \{50, 100\}$. DDQN runs failing to learn are marked with a straight line.

the DDQN training two more times for each $k \geq 7$ and $n = 100$. As shown in Figure 8, for $n = 100$ and all $k \geq 10$, there is at least one of three DDQN training runs where the agent does not learn anything, i.e., there is no progress in the entire training process.

Last, we investigate further the impact of problem dimension on the learning limit of DDQN. We train 3 DDQN agents for each pair of $n \in \{150, 200\}$ and $k \in \{3, 4, 5\}$, with a budget of 1.4 million steps and a walltime limit of 48 hours. Within the time limit, each DDQN agent can only reach 400 000 and 250 000 time steps for $n = 150$ and $n = 200$, respectively, since the length of each evaluation episode increases quadratically with n. Figure 9 shows the number of times each agent reaches the performance of the optimal policies during the entire training. These results indicate that $n = 200$ and $k = 5$ is the final limit of our DDQN agent with the chosen hyperparameters, as neither of the three runs can get close to the optimal policy.

5 CONCLUSION AND OUTLOOK

We suggested the optimization of the LeadingOnes problem via the $(1+1)$ RLS with fitness-dependent control policies as a benchmark

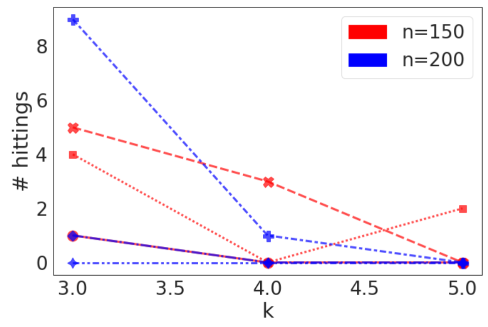


Figure 9: #times DDQN reaches performance of the optimal policy on `evenly_spread`, with $n \in \{150, 200\}$ and $k \in \{3, 4, 5\}$. Linestyles indicate individual runs with different seeds.

problem in the context of dynamic algorithm configuration (DAC). This problem setting is theoretically very well understood, to the point that we could easily extend in this work the base case with full parameter portfolio $[1..n]$ to settings in which the search radii have to be chosen from a restricted portfolio $\mathcal{K} \subsetneq [1..n]$. That is, we can compute optimal control policies for any given combination of problem dimension n and parameter portfolio $\mathcal{K}$. This allows us to create numerous problem instances of different size, which can be leveraged to gain structural insight into the behavior of DAC techniques. Empirically, we showed that DDQN efficiently learns optimal policies for the smaller LeadingOnes instances. We also explored the settings at which DDQN with the chosen parameters and budget reaches its limits, in the sense that the learned policy is not close to optimal or even fails to learn entirely.

One way to overcome the limits of DDQN for larger problem and portfolio sizes could be to use AutoRL [50] to optimize its hyperparameters, such as the batch size, discounting factor, exploration strategy, choice of algorithm, or network architecture. Although it is known that RL agents are very brittle with respect to their hyperparameters, their influence on the learning algorithm is not well understood. Our benchmark enables studying the effect of hyperparameters in a principled manner, which potentially allows us to make RL agents more robust and easier to use for dynamic algorithm configuration. A favorable aspect is that the evaluation times of the LeadingOnes benchmarks are very small, making a systematic investigation on the learning ability of RL agents computationally affordable. In fact, we can reduce the evaluation times further if we replace the actual training process by a simulation that draws the rewards from the well understood reward distribution.

Since we understand the distribution of the reward function perfectly well, no matter the problem dimension, the state, nor the played action (essentially captured by equation (1)), we believe that it is feasible to extend recent theoretical investigations of static algorithm configuration [30] to the more general DAC setting.

Regarding the DAC setting, we did not exploit the full power of DAC, as we trained and tested on the same problem instances and did not aim to derive policies that can be transferred to instances that are not part of the training set, as is classically done in algorithm configuration. Given the promising results of the DDQN agents, a reasonable next step is to investigate the generalization ability of this approach with respect to problem dimension or with respect to the portfolio. Once established, the next step are then to aim for generalizability across different problems, e.g., via a configurable benchmark generator that provides a good fit between problem representation and characteristics. The W-model [65] could be a reasonable playground for first steps in this direction. We note that generalization is an understudied topic in deep RL [41], where DAC and our proposed benchmark can help to advance the field.

Another idea we are keen on exploring is to incorporate other state information into the policy of the $(1 + 1)$ RLS than just the fitness. For example, for LeadingOnes, Buzdalov and Buzdalova [12] show that adding information about the number of correct bits in the tail allows more efficient control policies. When considering a good configuration of DDQN, this approach could also be applied in order to derive approximately optimal policies for scenarios of state information for which no theoretical guarantees are known.

We emphasize that we investigated the new benchmarks for DAC only, but they are equally interesting for the parameter control setting. Techniques that model parameter control as a multi-armed bandit problem [e.g. 15, 21, 29] can be straightforwardly applied to our benchmarks, as they typically require finite parameter portfolios. We also do not see greater obstacles to adjust other strategies, such as self-adaptive or self-adjusting parameter control mechanisms [26], although the parameter encoding and update strategies may need to be redesigned to account for the restricted portfolio.

We hope that our work initiates a fruitful exchange of benchmarks between parameter control and dynamic algorithm configuration. With the growing literature on parameter control [38] and its theoretical analysis [19], we wish to provide other use-cases with a known ground truth. However, settings for which we have such detailed knowledge as for LeadingOnes are very rare. Even for OneMax, the "drosophila of evolutionary computation" [28], the optimal mutation rates of the $(1 + 1)$ RLS and the $(1 + 1)$ evolutionary algorithm are known only in approximate terms [22] or for specific problem dimensions [11, 13, 14]. We believe that an active exchange of theoretically and automatically found policies will benefit both sides: empirical results provide guidance or inspiration for theoretical analyses, and theoretical results can be used as benchmarks with ground truth, as demonstrated in this work.

ACKNOWLEDGMENTS

André Biedenkapp and Frank Hutter acknowledge funding by the Robert Bosch GmbH. Nguyen Dang is a Leverhulme Early Career Fellow. This project has received funding from the European Union's Horizon 2020 research and innovation program under the Marie Skłodowska-Curie grant agreement No. 945298-ParisRegion-FP. It is also supported by the Paris Île-de-France region, via the DIM RFSI AlgoSelect project and is partially supported by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under GA No. 952215. The authors acknowledge the HPCaVe computing platform of Sorbonne Université for providing computational resources to this research project. The collaboration leading to this work was initiated at the 2020 Lorentz Center workshop "Benchmarked: Optimization Meets Machine Learning".

REFERENCES

[1] Peyman Afshani, Manindra Agrawal, Benjamin Doerr, Carola Doerr, Kasper Green Larsen, and Kurt Mehlhorn. 2019. The query complexity of a permutation-based variant of Mastermind. *Discrete Applied Mathematics* 260 (2019), 28–50. https://doi.org/10.1016/j.dam.2019.01.007

[2] Aldeida Aleti and Irene Moser. 2016. A Systematic Literature Review of Adaptive Parameter Control Methods for Evolutionary Algorithms. *Comput. Surveys* 49 (2016), 56:1–56:35.

[3] Thomas Bäck. 1998. An Overview of Parameter Control Methods by Self-Adaption in Evolutionary Algorithms. *Fundam. Informaticae* 35, 1-4 (1998), 51–66. https://doi.org/10.3233/FI-1998-35123404

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

[5] Roberto Battiti and Paolo Campigotto. 2012. An Investigation of Reinforcement Learning for Reactive Search Optimization. In *Autonomous Search*, Y. Hamadi, E. Monfroy, and F. Saubion (Eds.). Springer, 131–160.

[6] André Biedenkapp, H. Furkan Bozkurt, Theresa Eimer, Frank Hutter, and Marius Lindauer. 2020. Dynamic Algorithm Configuration: Foundation of a New Meta-Algorithmic Framework. In *Proc. of European Conference on Artificial Intelligence (ECAI'20) (Frontiers in Artificial Intelligence and Applications, Vol. 325)*. IOS Press, 427–434. https://doi.org/10.3233/FAIA200122

[7] André Biedenkapp, Nguyen Dang, Martin S. Krejca, Frank Hutter, and Carola Doerr. 2022. Code and data repository of this paper. https://github.com/ndangtt/LeadingOnesDAC.

[8] Mauro Birattari. 2009. *Tuning Metaheuristics - A Machine Learning Perspective*. Studies in Computational Intelligence, Vol. 197. Springer. https://doi.org/10.1007/978-3-642-00483-4

[9] Süntje Böttcher, Benjamin Doerr, and Frank Neumann. 2010. Optimal Fixed and Adaptive Mutation Rates for the LeadingOnes Problem. In *Proc. of Parallel Problem Solving from Nature (PPSN'10) (LNCS, Vol. 6238)*. Springer, 1–10.

[10] Edmund K. Burke, Michel Gendreau, Matthew R. Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and Rong Qu. 2013. Hyper-heuristics: a survey of the state of the art. *J. Oper. Res. Soc.* 64, 12 (2013), 1695–1724. https://doi.org/10.1057/jors.2013.71

[11] Nathan Buskulic and Carola Doerr. 2021. Maximizing Drift Is Not Optimal for Solving OneMax. *Evol. Comput.* 29, 4 (2021), 521–541. https://doi.org/10.1162/evco_a_00290

[12] Maxim Buzdalov and Arina Buzdalova. 2015. Can OneMax help optimizing LeadingOnes using the EA+RL method?. In *Proc. of Congress on Evolutionary Computation (CEC'15)*. IEEE, 1762–1768. https://doi.org/10.1109/CEC.2015.7257100

[13] Maxim Buzdalov and Carola Doerr. 2020. Optimal Mutation Rates for the $(1 + \lambda)$ EA on OneMax. In *Proc. of Parallel Problem Solving from Nature (PPSN'20) (LNCS, Vol. 12270)*. Springer, 574–587. https://doi.org/10.1007/978-3-030-58115-2_40

[14] Maxim Buzdalov and Carola Doerr. 2021. Optimal static mutation strength distributions for the $(1 + \lambda)$ evolutionary algorithm on OneMax. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'21)*. ACM, 660–668. https://doi.org/10.1145/3449639.3459389

[15] Luís Da Costa, Álvaro Fialho, Marc Schoenauer, and Michèle Sebag. 2008. Adaptive operator selection with dynamic multi-armed bandits. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'08)*. ACM, 913–920.

[16] Christian Daniel, Jonathan Taylor, and Sebastian Nowozin. 2016. Learning Step Size Controllers for Robust Neural Network Training, See [55].

[17] Benjamin Doerr. 2019. Analyzing randomized search heuristics via stochastic domination. *Theoretical Computer Science* 773 (2019), 115–137. https://doi.org/10.1016/j.tcs.2018.09.024

[18] Benjamin Doerr and Carola Doerr. 2018. Optimal Static and Self-Adjusting Parameter Choices for the $(1+(\lambda,\lambda))$ Genetic Algorithm. *Algorithmica* 80 (2018), 1658–1709. https://doi.org/10.1007/s00453-017-0354-9

[19] Benjamin Doerr and Carola Doerr. 2020. Theory of Parameter Control Mechanisms for Discrete Black-Box Optimization: Provable Performance Gains Through Dynamic Parameter Choices. In *Theory of Evolutionary Computation: Recent Developments in Discrete Optimization*. Springer, 271–321.

[20] Benjamin Doerr, Carola Doerr, and Johannes Lengler. 2021. Self-Adjusting Mutation Rates with Provably Optimal Success Rules. *Algorithmica* 83, 10 (2021), 3108–3147. https://doi.org/10.1007/s00453-021-00854-3 Available at https://arxiv.org/abs/1902.02588.

[21] Benjamin Doerr, Carola Doerr, and Jing Yang. 2016. k-Bit Mutation with Self-Adjusting k Outperforms Standard Bit Mutation. In *Proc. of Parallel Problem Solving from Nature (PPSN'16) (LNCS, Vol. 9921)*. Springer, 824–834. https://doi.org/10.1007/978-3-319-45823-6_77

[22] Benjamin Doerr, Carola Doerr, and Jing Yang. 2020. Optimal parameter choices via precise black-box analysis. *Theoretical Computer Science* 801 (2020), 1–34. https://doi.org/10.1016/j.tcs.2019.06.014

[23] Benjamin Doerr, Andrei Lissovoi, Pietro S. Oliveto, and John Alasdair Warwicker. 2018. On the runtime analysis of selection hyper-heuristics with adaptive learning periods. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'18)*. ACM, 1015–1022. https://doi.org/10.1145/3205455.3205611

[24] Carola Doerr and Johannes Lengler. 2018. The (1+1) Elitist Black-Box Complexity of LeadingOnes. *Algorithmica* 80, 5 (2018), 1579–1603. https://doi.org/10.1007/s00453-017-0304-6 Also available at https://arxiv.org/abs/1604.02355.

[25] Carola Doerr and Markus Wagner. 2018. Simple on-the-fly parameter selection mechanisms for two classical discrete black-box optimization benchmark problems. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'18)*. ACM, 943–950. https://doi.org/10.1145/3205455.3205560

[26] Agoston Endre Eiben, Robert Hinterding, and Zbigniew Michalewicz. 1999. Parameter control in evolutionary algorithms. *IEEE Transactions on Evolutionary Computation* 3 (1999), 124–141.

[27] Theresa Eimer, André Biedenkapp, Maximilian Reimer, Steven Adriaensen, Frank Hutter, and Marius Lindauer. 2021. DACBench: A Benchmark Library for Dynamic Algorithm Configuration. In *Proc. of International Joint Conference on Artificial Intelligence (IJCAI'21)*. ijcai.org, 1668–1674. https://doi.org/10.24963/ijcai.2021/230

[28] Álvaro Fialho, Luís Da Costa, Marc Schoenauer, and Michèle Sebag. 2008. Extreme Value Based Adaptive Operator Selection. In *Proc. of Parallel Problem Solving from Nature (PPSN'08) (LNCS, Vol. 5199)*. Springer, 175–184.

[29] Álvaro Fialho, Luís Da Costa, Marc Schoenauer, and Michèle Sebag. 2010. Analyzing bandit-based adaptive operator selection mechanisms. *Annals of Mathematics and Artificial Intelligence* 60 (2010), 25–64. https://doi.org/10.1007/s10472-010-9213-y

[30] George T. Hall, Pietro S. Oliveto, and Dirk Sudholt. 2022. On the impact of the performance metric on efficient algorithm configuration. *Artif. Intell.* 303 (2022), 103629. https://doi.org/10.1016/j.artint.2021.103629

[31] Assaf Hallak, Dotan Di Castro, and Shie Mannor. 2015. Contextual Markov Decision Processes. *CoRR* abs/1502.02259 (2015). http://arxiv.org/abs/1502.02259

[32] Nikolaus Hansen and Andreas Ostermeier. 2001. Completely Derandomized Self-Adaptation in Evolution Strategies. *Evolutionary Computation* 9, 2 (2001), 159–195. https://doi.org/10.1162/106365601750190398

[33] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. 2018. Deep reinforcement learning that matters. In *Proceedings of the Thirty-Second Conference on Artificial Intelligence (AAAI'18)*, Sheila A. McIlraith and Kilian Q. Weinberger (Eds.). AAAI Press, 3207–3214.

[34] Holger H. Hoos. 2012. Automated Algorithm Configuration and Parameter Tuning. In *Autonomous Search*, Youssef Hamadi, Éric Monfroy, and Frédéric Saubion (Eds.). Springer, 37–71. https://doi.org/10.1007/978-3-642-21434-9_3

[35] Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. 2009. ParamILS: An Automatic Algorithm Configuration Framework. *Journal of Artificial Intelligence Research* 36 (2009), 267–306.

[36] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren (Eds.). 2019. *Automated Machine Learning - Methods, Systems, Challenges*. Springer. https://doi.org/10.1007/978-3-030-05318-5

[37] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. 2017. Population Based Training of Neural Networks. *arXiv:1711.09846 [cs.LG]* (2017).

[38] Giorgos Karafotias, Mark Hoogendoorn, and A.E. Eiben. 2015. Parameter Control in Evolutionary Algorithms: Trends and Challenges. *IEEE Transactions on Evolutionary Computation* 19 (2015), 167–187.

[39] Giorgos Karafotias, Selmar K. Smit, and A. E. Eiben. 2012. A Generic Approach to Parameter Control. In *Proc. of Applications of Evolutionary Computation (EvoApplications'12) (LNCS, Vol. 7248)*. Springer, 366–375. https://doi.org/10.1007/978-3-642-29178-4_37

[40] Eric Kee, Sarah Airey, and Walling Cyre. 2001. An Adaptive Genetic Algorithm. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'01)*. Morgan Kaufmann, 391–397. https://doi.org/10.5555/2955239.2955303

[41] Robert Kirk, Amy Zhang, Edward Grefenstette, and Tim Rocktäschel. 2021. A Survey of Generalisation in Deep Reinforcement Learning. *arXiv:2111.09794 [cs.LG]* (2021).

[42] Scott Kirkpatrick, C. D. Gelatt, and Mario P. Vecchi. 1983. Optimization by Simulated Annealing. *Science* 220 (1983), 671–680.

[43] Michail G. Lagoudakis and Michael L. Littman. 2000. Algorithm Selection using Reinforcement Learning. In *Proceedings of the Seventeenth International Conference on Machine Learning (ICML'00)*, Pat Langley (Ed.). Morgan Kaufmann Publishers, 511–518.

[44] Michail G. Lagoudakis and Michael L. Littman. 2001. Learning to Select Branching Rules in the DPLL Procedure for Satisfiability. *Electronic Notes in Discrete Mathematics* 9 (2001), 344–359.

[45] Per Kristian Lehre and Carsten Witt. 2012. Black-Box Search by Unbiased Variation. *Algorithmica* 64 (2012), 623–642.

[46] Andrei Lissovoi, Pietro S. Oliveto, and John Alasdair Warwicker. 2020. Simple Hyper-Heuristics Control the Neighbourhood Size of Randomised Local Search Optimally for LeadingOnes. *Evol. Comput.* 28, 3 (2020), 437–461. https://doi.org/10.1162/evco_a_00258

[47] Ilya Loshchilov and Frank Hutter. 2017. SGDR: Stochastic Gradient Descent with Warm Restarts. In *Proceedings of the International Conference on Learning Representations (ICLR'17)*. Published online: iclr.cc.

[48] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529–533.

[49] Jack Parker-Holder, Vu Nguyen, and Stephen J. Roberts. 2020. Provably Efficient Online Hyperparameter Optimization with Population-Based Bandits. In *Proceedings of the 33rd International Conference on Advances in Neural Information Processing Systems (NeurIPS'20)*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). Curran Associates.

[50] Jack Parker-Holder, Raghu Rajan, Xingyou Song, André Biedenkapp, Yingjie Miao, Theresa Eimer, Baohe Zhang, Vu Nguyen, Roberto Calandra, Aleksandra Faust, Frank Hutter, and Marius Lindauer. 2022. Automated Reinforcement Learning (AutoRL): A Survey and Open Problems. *CoRR* abs/2201.03916 (2022). arXiv:2201.03916 https://arxiv.org/abs/2201.03916

[51] James E. Pettinger and Richard M. Everson. 2002. Controlling Genetic Algorithms with Reinforcement Learning. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'02)*, W. Langdon, E. Cantu-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. Potter, A. Schultz, J. Miller, E. Burke, and N. Jonoska (Eds.). Morgan Kaufmann Publishers, 692.

[52] Ingo Rechenberg. 1973. *Evolutionsstrategie*. Friedrich Fromman Verlag (Günther Holzboog KG), Stuttgart.

[53] John R. Rice. 1976. The Algorithm Selection Problem. *Advances in Computers* 15 (1976), 65–118.

[54] Yoshitaka Sakurai, Kouhei Takada, Takashi Kawabe, and Setsuo Tsuruta. 2010. A Method to Control Parameters of Evolutionary Algorithms by Using Reinforcement Learning. In *Proceedings of Sixth International Conference on Signal-Image Technology and Internet-Based Systems (SITIS)*, K. Yétongnon, A. Dipanda, and R. Chbeir (Eds.). IEEE Computer Society, 74–79.

[55] D. Schuurmans and M. Wellman (Eds.). 2016. *Proceedings of the Thirtieth National Conference on Artificial Intelligence (AAAI'16)*. AAAI Press.

[56] Gresa Shala, André Biedenkapp, Noor Awad, Steven Adriaensen, Marius Lindauer, and Frank Hutter. 2020. Learning Step-Size Adaptation in CMA-ES. In *Proceedings of the Sixteenth International Conference on Parallel Problem Solving from Nature (PPSN'20) (Lecture Notes in Computer Science)*. Springer, 691–706.

[57] Mudita Sharma, Alexandros Komninos, Manuel López-Ibáñez, and Dimitar Kazakov. 2019. Deep Reinforcement Learning-Based Parameter Control in Differential Evolution. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'19)*. ACM, 709–717. https://doi.org/10.1145/3321707.3321813

[58] Selmar K. Smit and A. E. Eiben. 2009. Comparing parameter tuning methods for evolutionary algorithms. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC'09)*. IEEE, 399–406. https://doi.org/10.1109/CEC.2009.4982974

[59] David Speck, André Biedenkapp, Frank Hutter, Robert Mattmüller, and Marius Lindauer. 2021. Learning Heuristic Selection with Dynamic Algorithm Configuration. In *Proceedings of the 31st International Conference on Automated Planning and Scheduling (ICAPS'21)*, H. H. Zhuo, Q. Yang, M. Do, R. Goldman, S. Biundo, and M. Katz (Eds.). AAAI, 597–605.

[60] Dirk Sudholt. 2013. A New Method for Lower Bounds on the Running Time of Evolutionary Algorithms. *IEEE Transactions on Evolutionary Computation* 17 (2013), 418–435.

[61] Richard S. Sutton and Andrew G. Barto. 1998. *Reinforcement learning - an introduction*. MIT Press. https://www.worldcat.org/oclc/37293240

[62] Hado van Hasselt, Arthur Guez, and David Silver. 2016. Deep Reinforcement Learning with Double Q-Learning, See [55], 2094–2100.

[63] Diederick Vermetten, Sander van Rijn, Thomas Bäck, and Carola Doerr. 2019. Online selection of CMA-ES variants. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'19)*. ACM, 951–959. https://doi.org/10.1145/3321707.3321803

[64] Christopher. J. C. H. Watkins. 1989. *Learning from Delayed Rewards*. Ph. D. Dissertation. King's College, Cambridge, United Kingdom.

[65] Thomas Weise, Yan Chen, Xinlu Li, and Zhize Wu. 2020. Selecting a diverse set of benchmark instances from a tunable model problem for black-box discrete optimization algorithms. *Applied Soft Computing* 92 (2020), 106269. https://doi.org/10.1016/j.asoc.2020.106269