Using Automated Algorithm Configuration for Parameter Control

Deyao Chen, Maxim Buzdalov, Carola Doerr, Nguyen Dang

Using Automated Algorithm Configuration for Parameter Control

Deyao Chen
University of St Andrews
St Andrews, United Kingdom

Maxim Buzdalov
Aberystwyth University
Aberystwyth, United Kingdom

Carola Doerr
Sorbonne Université, CNRS, LIP6
Paris, France

Nguyen Dang
University of St Andrews
St Andrews, United Kingdom

ABSTRACT

Dynamic Algorithm Configuration (DAC) tackles the question of how to automatically learn policies to control parameters of algorithms in a data-driven fashion. This question has received considerable attention from the evolutionary community in recent years. Having a good benchmark collection to gain structural understanding on the effectiveness and limitations of different solution methods for DAC is therefore strongly desirable. Following recent work on proposing DAC benchmarks with well-understood theoretical properties and ground truth information, in this work, we suggest as a new DAC benchmark the controlling of the key parameter λ in the $(1 + (\lambda, \lambda))$ Genetic Algorithm for solving OneMax problems. We conduct a study on how to solve the DAC problem via the use of (static) automated algorithm configuration on the benchmark, and propose techniques to significantly improve the performance of the approach. Our approach is able to consistently outperform the default parameter control policy of the benchmark derived from previous theoretical work on sufficiently large problem sizes. We also present new findings on the landscape of the parameter-control search policies and propose methods to compute stronger baselines for the benchmark via numerical approximations of the true optimal policies.

CCS CONCEPTS

• **Computing methodologies** → **Randomized search**.

KEYWORDS

evolutionary computation, algorithm configuration, parameter control, genetic algorithms, benchmarking

1 INTRODUCTION

Evolutionary algorithms and similar randomized search heuristics have a number of parameters that allow to fine-tune their behavior to the problem at hand and to the current stage of the optimization process. However, despite a long series of works studying which parameters work well for which algorithms and which problem scenarios, the question how to control these parameters in an automated fashion is wide open. Concepts such as *hyper-heuristics* [12], *parameter control* [23, 29, 36], and *adaptive operator selection* [31] dominate research on this topic in evolutionary computation (EC). More recently, approaches based on machine learning (ML), and in particular based on reinforcement learning (RL), propose to explicitly *train* control policies. That is, where EC assumes that a given problem instance has to be solved instantly,

the ML-approaches focus on settings in which similar problem instances need to be solved, and an explicit training is possible. This setting was first studied in [47] and later named *dynamic algorithm configuration* (DAC) in [10]. It quickly became a very active area of research [1, 11, 30, 46, 48, 49, 51]. A parallel research direction in the hyper-heuristic community is on *automated algorithm design,* which focuses on leveraging deep RL approaches for training policies to select the best algorithm components based on the search states [52–54].

Regardless of whether adopting the EC view or whether considering the DAC setting, it is highly desirable to have access to benchmarks that support a sound investigation of the proposed methods. As in other branches of computer science, benchmarking helps to compare the efficiency of different algorithms and to investigate their strengths and weaknesses. Benchmarking has a long tradition in EC, and has undoubtedly helped our community mature [7, 33]. In difference to *competitive testing*, where the main focus is on the performance of the solvers, *benchmarking* relies on settings that are sufficiently well understood to help us gain insight into the working principles that drive this performance – knowledge that not only supports the improvement of the algorithms but also leads to a better deployment of the existing approaches in practice.

The idea of leveraging settings with well-understood theoretical properties and using them for benchmarking DAC approaches was recently promoted in [11]. The paper adopted the (1+1) Randomized Local Search for solving variants of the LeadingOnes problem with configurable problem dimension and action spaces, where the action space corresponds to the portfolio of the parameters that the DAC policies are allowed to choose from. Notably, the paper [11] extended previously known theoretical results from [21, 28] to settings with restricted portfolio. Using these provably optimal *"ground truth"* policies, their work demonstrated that an off-the-shelf RL-based learning mechanism can work very well if both the number of parameter values to choose from and the problem dimension are rather small. As soon as either of them increases beyond some relatively moderate threshold, the observed performance deteriorated quickly.

Our contributions. Following the approach in [11], in this paper we propose a new DAC benchmark derived from EC theory. An essential property of a DAC benchmark is the clear discrepancy between static configuration and dynamic control policies. To that end, we adopt the setting proposed in [24], which involves controlling the population size parameter λ in the $(1 + (\lambda, \lambda))$ Genetic Algorithm (GA) for the OneMax problem. As in the LeadingOnes benchmarks, the goal is to minimize the expected running time of

the algorithm, i.e., the total number of evaluations that the algorithm performs before it reaches an optimal solution. The setting also comes with an asymptotically optimal parameter control policy with linear expected runtime [24], which will be used as a baseline for our DAC experiments in this paper.

The new benchmark has a number of interesting properties compared to the LeadingOnes. First, the performance gain obtained by an optimally controlled parameter setting over the best possible static setting is $\omega(1)$, i.e., it grows more than a constant factor as the problem dimension increases, while the advantage of an optimally controlled policy for LeadingOnes is "only" a constant factor, i.e., $\Theta(1)$. Second, as we will show in our analysis (Section 4.2), the parameter search space has an unusual landscape with frequent abrupt changes across the whole area, making it potentially challenging for DAC approaches.

It is worth to note that deep RL is not the only solution method for DAC. An alternative approach is to formulate DAC as a (meta-)optimization problem, where we search in the policy space directly via black-box optimization methods [1]. Our second contribution is on this direction: we investigate the potential of using automated algorithm configuration techniques to solve DAC problems. More specifically, we formulate a policy of our benchmark as a static parameter setting and configure it using the automated algorithm configuration tool IRACE [41]. As will be detailed in Section 3, a naive application of IRACE does not work well, especially when the problem size is over 200, which is considered rather small for OneMax problems. This can be accounted to the large number of parameters the tuning has to deal with and the complicated landscape of the parameter search space. We then propose a *binning* approach to reduce the search space, and a *cascading* application of the tuning. A combination of those two strategies results in significant improvement in performance. The tuning is able to find policies that consistently outperform the theory-derived policy in [24].

Our last contribution involves expanding our understandings on the proposed DAC benchmark. We extend recent works on numerically computing optimal control policies for concrete dimensions via dynamic programming. Originally proposed in [13] and later refined in [6, 15], this approach allows us to approximate the optimal control policy of our new benchmark with very good precision. In this work, we compute the approximations of both the generally optimal policies and the optimal binning policies, which show that there is a consistent but small gap between the two. Apart from serving us as new baselines for our tuning-based approach and for other DAC methods in general, our work also reveals some facts about the $(1 + (\lambda, \lambda))$ GA that we expect to be of interest in the context of running time analysis, parameter setting, and algorithm design. Indeed, search landscapes associated with parameter setting problems are usually thought to be rather smooth [42, 43], but our findings suggest that it is not always true, and in particular not for parameter control settings. This finding reinforces a similar observation made in [6] for landscape of parameter control policies for the much simpler $(1+\lambda)$-type evolutionary algorithms. We consider such insights on (un)structured parameter control landscapes very valuable for further algorithm design – whether in the context of DAC or in the classic black-box optimization setting.

Availability of code and data. To adhere to reproducibility standards discussed in [40], our code and data are available at [18].

2 CONTROLLING THE POPULATION SIZE OF THE $(1 + (\lambda, \lambda))$ GA ON ONEMAX

Notation. We always denote by n the dimension of the search space. For a search point $x \in \{0, 1\}^n$ we write $x = (x_1, \ldots, x_n)$. For two real number a and b we denote by $[a..b]$ the set of all integers k that satisfy $a \le k \le b$.

OneMax. In this work, we are interested in minimizing the expected optimization time of the $(1 + (\lambda, \lambda))$ GA on OneMax. OneMax is one of the most important benchmark problems in the analysis of evolutionary algorithms (EAs). It denotes the collection of functions $\{f_z \mid z \in \{0, 1\}^n\}$ with $f_z : \{0, 1\}^n \to [0..n], x \mapsto |\{i \in [1..n] \mid x_i = z_i\}|$, the function that counts in how many positions the solution candidate x agrees with the secret target string z. As it was discussed in [26], OneMax can be seen as the Mastermind problem with two colors, 0 and 1. The name "OneMax" originates in the fact that for analyzing the performance of so-called *unbiased* algorithms, it suffices to study their behavior on the function $f_{(1,\ldots,1)}$, see [27, 39] for detailed explanations. This is also the case for the $(1 + (\lambda, \lambda))$ GA introduced below. Despite its simplicity, the OneMax problem is highly relevant for understanding the behavior of EAs in regimes in which we have a good fitness-distance correlation. That is, OneMax helps us understand how algorithms perform in environments in which they are not misled by search points that have better fitness value than another while at the same time being farther away from the optimum.

Background on the $(1+(\lambda, \lambda))$ GA. The $(1+(\lambda, \lambda))$ GA was originally introduced in [24] to formally prove that the use of crossover can be beneficial even for optimizing rather simple functions such as OneMax. It later started to play an important role in the analysis of parameter control mechanisms. Already in [24] it was shown that a fitness-dependent setting of the key parameter λ can lead to expected running times that are asymptotically better than that of the best static setting. Later works showed that a simple one-fifth success rule, as adopted to the discrete setting in [37], leads to best possible linear running time. This result inspired a number of follow-up works on parameter control [23]. More recently, the $(1 + (\lambda, \lambda))$ GA plays another important role for the study of unusual mutation operators such as the heavy-tailed "fast" mutation operators introduced in [25]; see [3] for a recent example. Finally, the $(1 + (\lambda, \lambda))$ GA has also been studied in other contexts, see [4] for work on LeadingOnes, [14, 32] for applications to satisfiability problems, and [5] for more results and further references.

The $(1 + (\lambda, \lambda))$ GA. Alg. 1 presents the pseudo-code of the $(1 + (\lambda, \lambda))$ GA version analyzed in this work. The algorithm is initialized by sampling and evaluating a search point chosen from $\{0, 1\}^n$ uniformly at random (u.a.r.). It then proceeds in rounds, consisting of one mutation phase and one crossover phase each. In the mutation phase, $\Lambda = \lfloor \lambda \rfloor$ new solution candidates ("*offspring*") are sampled, where $\lfloor \lambda \rceil := \lfloor \lambda \rfloor$ if $\lambda - \lfloor \lambda \rfloor < 0.5$ and $\lfloor \lambda \rceil := \lceil \lambda \rceil$ otherwise. Each offspring is sampled by inverting the bits in ℓ position that are chosen u.a.r.. Here, ℓ is a random number that is kept fixed throughout one iteration and that is sampled from the *resampling* binomial distribution $\text{Bin}_{>0}(n, \lambda/n)$ that performs

---

Algorithm 1: The $(1 + (\lambda, \lambda))$ GA variant analyzed in this paper.

Input: problem size n
a parameter control policy $\pi : [0..n - 1] \rightarrow [1; n]$

1 **Initialization:** Sample $x \in \{0, 1\}^n$ u.a.r. and evaluate $f(x)$;

2 **Optimization: for** $t = 1, 2, 3, \ldots$ **do**

3   Let $\lambda = \pi(f(x))$;

4   **Mutation phase:**

5   Sample ℓ from $\mathrm{Bin}_{>0}(n, \lambda/n)$;

6   Set population size Λ as $\lfloor\lambda\rfloor$;

7   **for** $i = 1, \ldots, \Lambda$ **do** $x^{(i)} \leftarrow \mathrm{flip}_\ell(x)$; Evaluate $f(x^{(i)})$;

8   Choose $x' \in \{x^{(1)}, \ldots, x^{(\Lambda)}\}$ with
        $f(x') = \max\{f(x^{(1)}), \ldots, f(x^{(\Lambda)})\}$ u.a.r.;

9   **Crossover phase:**

10   **for** $i = 1, \ldots, \Lambda$ **do**

11          $y^{(i)} \leftarrow \mathrm{cross}_{1/\lambda}(x, x')$;

12   **if** $y^{(i)} \notin \{x, x'\}$ **then** evaluate $f(y^{(i)})$;

13   Choose $y' \in \{y^{(1)}, \ldots, y^{(\Lambda)}\}$ with
        $f(y') = \max\{f(y^{(1)}), \ldots, f(y^{(\Lambda)})\}$ u.a.r.;

14   **Selection and update step:**

15   **if** $f(y') > f(x')$ **then** $y \leftarrow y'$ **else** $y \leftarrow x'$;

16   **if** $f(y) \geq f(x)$ **then** $x \leftarrow y$;

---

n trials with success rate λ/n each and resamples i.i.d. until a value greater than 0 is found. The best of these Λ offspring (ties broken u.a.r.), referred to as x' in Alg. 1, is selected to participate in the crossover phase. In the crossover phase, another Λ search points are generated, each sampled from a uniform crossover between the original parent x and the selected mutant x'. The crossover operator $\mathrm{cross}_{1/\lambda}(x, x')$ treats each position independently and sets $y_i = x'_i$ with probability $1/\lambda$ and it sets $y_i = x_i$ otherwise. The so-created *"crossover offspring"* are only evaluated if they are different from both parents, and they are not further taken into consideration otherwise. For y' denoting the best of the samples generated in the crossover phase (ties broken again u.a.r.), we then let $y = y'$ if y' is strictly better than x' and we let $y = x'$ otherwise. Finally, y replaces x as parent for the next iteration if it is at least as good, i.e., if $f(y) \geq f(x)$.

Parameter settings. In the *static* version of the $(1 + (\lambda, \lambda))$ GA, the value of λ is fixed throughout the whole run, i.e., the control policy π in Alg. 1 is constant. In the *dynamic* version, λ can take different values at different stages of the optimization process.

The default dynamic control policy that we compare our results against, namely THEORY, assigns to each fitness value i the parameter value $\pi(i) := \sqrt{n/(n - i)}$. It was used in [24, Theorem 8] to prove an asymptotic super-constant speedup over the best static choice of λ.

3 CONFIGURING CONTROL POLICIES

We represent a parameter control policy for the $(1 + (\lambda, \lambda))$ GA as a mapping from a current fitness to a specific λ value. The problem of finding the best parameter control policy therefore can be stated

as a static automated algorithm configuration, where the number of parameters is equal to the problem size n: each parameter correspond to the λ value of a fitness value in the range of $[0..n - 1]$. We use the automated algorithm configuration tool IRACE [41] to solve this task. We apply this dynamic tuning approach, namely TUNED, on various problem sizes from 10 to 2000. The tuning budget is set as 50000 runs. Since the performance metric is runtime related, the adaptive capping feature of IRACE is enabled as it has been shown to significantly improve the tuning performance in various cases [17, 20]. As we are interested in optimizing *expected* running time, we set IRACE's statistical test to be the Student t-test. Moreover, to account for the noisy nature of the benchmark, the *firstTest* parameter (the number of instances/seeds being evaluated at the beginning of each IRACE's iteration before the first statistical test is applied) is increased from 5 (default) to 10. All other parameters of IRACE are set as default. For comparison, we also apply IRACE on a *static* version of the $(1 + (\lambda, \lambda))$ GA where λ is fixed during the whole run. This version is named TUNED_STATIC, and the tuning budget is set as $\min(100 \times n, 20000)$ runs. The final configuration found by each tuning experiment is evaluated across 500 different random seeds, and their performances are presented in Figure 1.

It is clear that the dynamic tuning approach does not scale well with problem sizes: the expected runtime of TUNED is much higher than both THEORY and TUNED_STATIC, especially when $n \geq 200$. [1] The differences in performance (i.e., TUNED vs THEORY and TUNED vs TUNED_STATIC) are statistically significant [2] for $n \geq 50$ according to the Wilcoxon rank sum test with a confidence level of 99.9%. One possible explanation for the bad performance of TUNED is due to the large number of numerical parameters IRACE has to deal with. In fact, automated algorithm configuration scenarios often either: (i) involves a few dozens parameters at most, or in some cases, up to 200 parameters; or (ii) has a large number of categorical parameters with only a few possible values [35].

To reduce the parameter space, we define a *binning* approach where we partition the objective space $[0..n]$ into k consecutive bins $\{B_1, B_2, .., B_k\}$, and only tune one λ parameter for all fitness values in the same bin, i.e., the number of parameters is now reduced from n to k. In many optimization problems, the closer we get to the optimal, the harder it is to optimize. Therefore, we define the bins so that their sizes are gradually reduced as the objective values increased. More concretely, we set $B_i = [a_i..a_{i+1} - 1]$ $(i \in [1, k-1])$ where $a_i = n - \lfloor\frac{n}{2^{i-1}}\rfloor$ $\forall i \in [1, k]$, and $B_k = [a_k..n]$. Figure 3 illustrates how the bins are created with different k values. We name this approach TUNED_BIN. From this point onward, we will focus on sufficiently large problem sizes ($n \geq 500$) as those are the cases where the discrepancy between the static and the dynamic policies is obvious. The tuning budget is set as 5000, 10000, 20000, and 21000 for n equals to 500, 1000, 2000, and 3000, respectively.

A larger number of bins results in more flexibility in the tuning and better quality of the best possible parameter-control policy, but it may also lead to more difficulty in searching in such space

---

[1] for $n = 10$, the average performance of TUNED_STATIC is better than both THEORY and TUNED. But this is likely due to the problem size being too small to see the impact of dynamic parameter control.

[2] In this paper, whenever multiple statistical tests are conducted, the Bonferroni correction is applied.
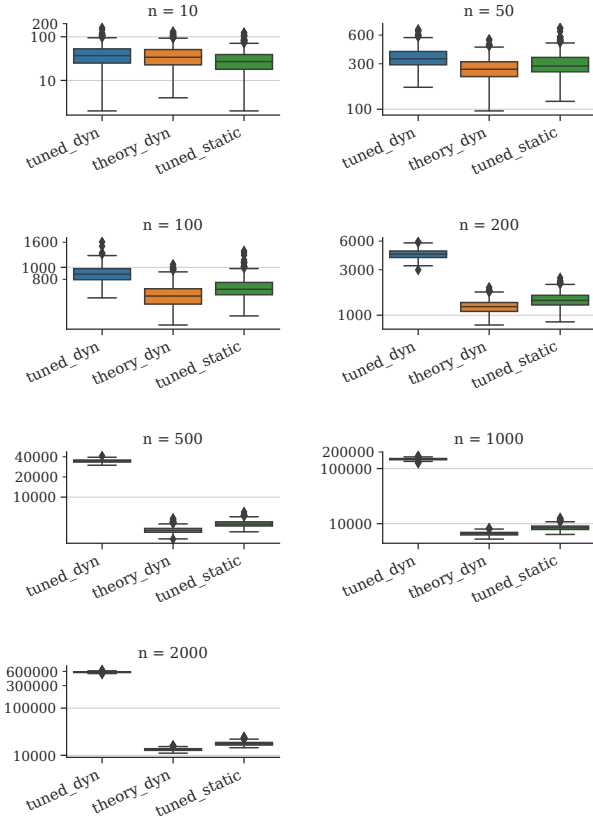
Figure 1: Performance of a naive dynamic tuning with IRACE (TUNED), a static tuning (TUNED_STATIC), and the default policy derived from theory (THEORY), evaluated across 500 random seeds.

due to its size. To study the effect of this choice, we conduct experiments with all possible numbers of bins ($k \in [1..\lceil \log_2 n \rceil]$). The performance of the final configurations are presented in Figure 2. We also conducted another baseline for the comparison: the binned versions of the THEORY, namely BINNED_THEORY, where the fitness range is split into $\lceil \log_2 n \rceil$ bins, and all λ values belonging to the same bin are derived from THEORY at the start(S), the middle (M), or the end(E) of the bin (for the middle case, if the number of elements in a bin is even, we choose the smaller point).

Results in Figure 2 clearly indicates a trade-off between the number of bins and the tuning performance. Starting from one bin, the tuning performance generally improves when the number of bins is increased until it reaches a certain limit (around 5 or 6). The tuning results after such point not only drastically degrades, but are also *unstable*. Consider, for example, the case of $n = 500$, we have both TUNED_BIN6 and TUNED_BIN7 performing quite badly compared to the cases with smaller numbers of bins, indicating degradation of the tuning performance, and we would expect TUNED_BIN7 to be worse than TUNED_BIN6 due to the larger number of parameters

IRACE has to deal with. However, we observe the opposite. To check whether the counter-intuitive performance is due to the instability of the tuning, we repeat both tuning experiments 5 times. And in fact, the average runtime of TUNED_BIN6 reduces from 6446 to 4921, while the runtime of TUNED_BIN7 increases from 5589 to 6043.

To improve the tuning further, we propose a *cascading* strategy, where the final configuration of the tuning with k bins is given as an initial configuration for the tuning with $k + 1$ bins. The cascading gives each tuning a head start by leveraging results of the previous step. The new tuning experiments are named TUNED_CAS_BIN, and their results are shown in Figure 2. The tuning performance is improved drastically. And the final configuration (found with the largest number of bins) for each problem size even statistically significantly outperforms THEORY (both original and binned versions) for $n \geq 1000$ according to the Wilcoxon rank sum test with a confidence level of 99.9%. Note that for $n = 500$, the two configurations are not statistically significantly different.

One may argue that the good performance of the cascading approach may come from the fact that the total tuning budget is larger than each individual TUNED_BIN experiment, since with cascading, the tunings are executed sequentially until we reach the largest number of bins. To investigate this point further, we re-run the last TUNED_BIN experiments for both $n \in 500, 1000$ with 10 times of their original budget (which is roughly the same as the total budget for all TUNED_CAS_BIN experiments of the same problem size). We observe some improvement in performance for $n = 500$: average runtime of the final configuration reduces from 7988 to 5818, but there is still a large gap compared to the TUNED_CAS_BIN with an average runtime of 3234. Interestingly, for $n = 1000$, the new configuration is even worse than before (original runtime: 11468, new runtime: 15109, TUNED_CAS_BIN: 6492). Again, this can be accounted to the instability of dynamic tuning without cascading. Those observation confirms the clear advantage that cascading offers to the tuning performance.

The results of this section indicates the potential of applying automated algorithm configuration for the dynamic tuning of the $(1 + (\lambda, \lambda))$ GA on ONEMAX. Our binning and cascading strategies help to find configurations that are better than the best known control policy derived from theory. This is particularly encouraging, given that the tuning does not have access to THEORY. The next question would be: has the tuning been able to find the best possible policies, or is there still room for improvement? To answer such question, in the next section, we will describe a method to compute stronger baselines for this benchmark.

4 EXACT COMPUTATIONS

In this section we describe an algorithm of computing the exact expected running time of the $(1 + (\lambda, \lambda))$ GA on ONEMAX provided we are given the mapping from the parent fitness f to the parameter value $\lambda = \lambda(f)$. Based on that, we perform some basic landscape analysis and propose a way to closely approximate the best baseline policies (the truly best policy and the best one among the binned policies) using numeric minimization.
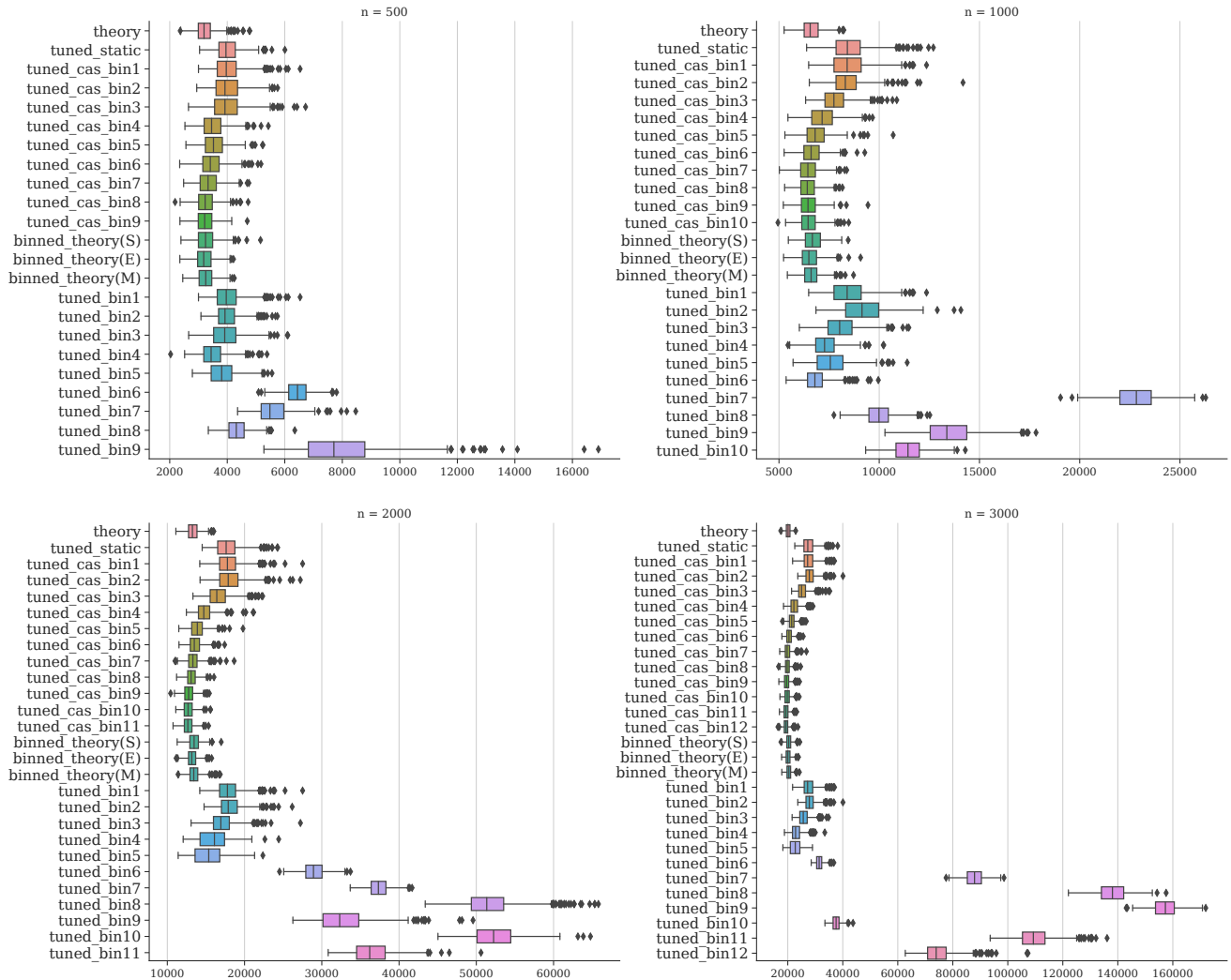
Figure 2: Performance (in terms of function evaluations) of dynamic tuning with binning (TUNED_BIN), and with binning and cascading (TUNED_CAS_BIN). The number at the end of each configuration represents the number of bins. Other baselines include: THEORY and its binned versions, and the static tuning (TUNED_STATIC). All configurations are evaluated across 500 random seeds.

4.1 Computing Runtime for Given Parameters

Let the current parent's ONEMAX fitness be f, and the current parameter be λ. Let Λ be the derived population size; the actual algorithm uses $\Lambda = \lfloor \lambda \rfloor$, but we are going to use different values during landscape analysis. As the $(1 + (\lambda, \lambda))$ GA is elitist and all parents with the same fitness produce stochastically identical behavior, we can use the dynamic programming approach similar to the one used in previous works [6, 13, 15, 16]. Basically, we compute the remaining runtimes backwards from the optimum: assuming that T_f is the expected time to reach the optimum when the parent's fitness is f, we note that $T_n = 0$ and for every f the

value of T_f depends only on T_{f+1}, T_{f+2} and other values for higher fitness.

To process the fitness value f, our algorithm essentially considers all possible event chains that the $(1 + (\lambda, \lambda))$ GA performs, computes their probabilities and saves the computational effort when possible. Since most of the probabilities are fractions formed by binomial coefficients, each of which may be small or large, we use the standard trick of computing logarithms of those quantities (which are aggressively cached to avoid calling the expensive logarithm function) and taking an exponent of them only when summing.
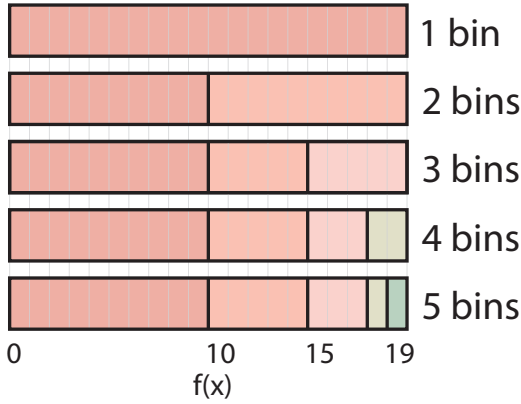
Figure 3: An illustration of the division of the bins for $n = 20$.

Figure 4: Example runtimes for $n = 500$, fixed binned policy and different values of the last λ

We start with considering how many bits are flipped in the mutation phase. The probability of flipping exactly ℓ bits follows from the fact that $\ell \sim \text{Bin}_{>0}(n, \lambda/n)$. Let g be the number of "good" bits in the best offspring, that is, how many out of the ℓ bits are flipped from 0 to 1 in x'. The probability $P[g \mid \ell]$ can be computed as follows: let $P_m(i)$ be the (easily computed) probability to have i "good" bits in one offspring, then

$$P[g \mid \ell] = \left(\sum_{i=0}^{g} P_m(i)\right)^{\Lambda} - \left(\sum_{i=0}^{g-1} P_m(i)\right)^{\Lambda}$$

by combinatorial arguments. Hence, the crossover phase starts with x' being different from the parent x in ℓ bits, g of which are "good".

Each crossover offspring is obtained from the parent x by considering the ℓ bits that are different between x and x' and flipping δ_g of those bits from 0 to 1, and δ_b bits from 1 to 0. Again, remembering that only g of these ℓ bits have the value 1 in x', it holds that $0 \le \delta_g \le g$ and $0 \le \delta_b \le \ell - g$. The probability $P[\delta_g, \delta_b | \ell, g]$ can again be computed by combinatorial arguments. The fitness of such an offspring is $f + \delta_g - \delta_b$. Since the algorithm's state changes only when the fitness improves, and because the best mutation offspring, x', can also replace the parent, we are rather interested in the truncated fitness change, e.g. $\delta = \max\{0, f(x') - f, \delta_g - \delta_b\}$. Let $P_c(i)$ be the probability of obtaining such truncated fitness change of i in one crossover offspring, then

$$P[\delta \mid \ell, g] = \left(\sum_{i=0}^{\delta} P_c(i)\right)^{\Lambda} - \left(\sum_{i=0}^{\delta-1} P_c(i)\right)^{\Lambda}$$

similarly to the above. If $\delta = 0$, we consider the iteration a failure. Otherwise the algorithm updates the fitness of the parent, for which we already computed the runtime $T_{f+\delta}$. We sum up these remaining expected times, weighed by the probabilities of all the preceding events to happen, as well as these probabilities themselves.

Getting back to the current fitness value f, we obtain the probability of improving the parent p_i and the conditioned remaining expected time t_i. Denote the expected iteration cost, in fitness evaluations, to be τ_λ. Then, $T_f = \tau_\lambda + p_i \cdot t_i + (1 - p_i) T_f$, which resolves to a well-known solution $T_f = (\tau_\lambda + t_i)/p_i$. The value of τ_λ is a sum of Λ evaluations in the mutation phase and $\Lambda \cdot (1 - (1/\lambda)^\ell - (1 - 1/\lambda)^\ell)$ evaluations in the crossover phase that do not produce a crossover
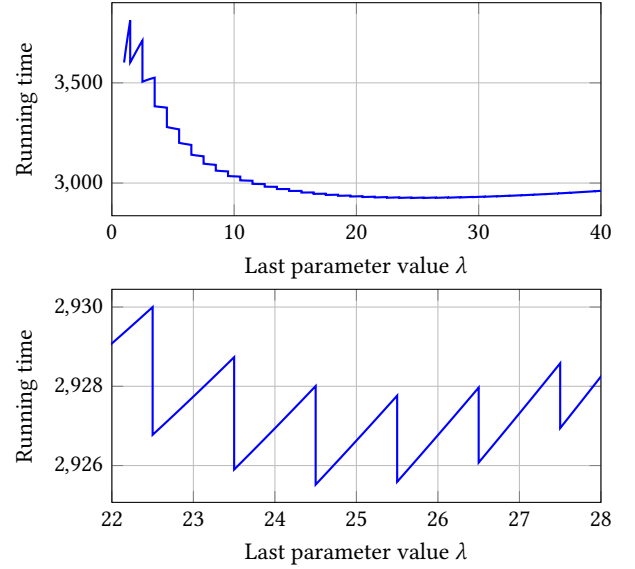
offspring equal to either of its parents. The final runtime of the policy is obtained from all T_f, given that the algorithm is initialized with fitness f with the probability $\binom{n}{f}/2^n$.

The overall runtime of the whole process can be estimated as $O(n^4)$ assuming a straightforward implementation. However, we perform certain optimizations in order to be able to work with problem sizes as large as $n = 2000$ in reasonable time.

Implementation detail 1. Note that the probabilities $P[\delta \mid \ell, g]$ do not depend on the current fitness, but do depend on both λ and Λ. For this reason, they can be computed once and stored for later re-use. Since the probabilities for a tuple $(\ell, g, \lambda, \Lambda)$ are computed in $\Theta(g^2)$ time and take only $\Theta(g)$ to store and use, such caching may improve the overall computation time by a factor up to $O(n)$. For certain scenarios the overall size of cached probabilities exceeds the amount of memory available, so we use the adaptive cache that retains only the most expensive entries w.r.t. computation time.

Implementation detail 2. To improve the computation time further, we ignore the events as long as their probability does not change the existing sum of probabilities in the machine precision when added to it, that is, whenever $p + \delta p = p$ in the 64-bit floating point type commonly known as `double`. Where possible, we reorganize computations so that the major contributions are evaluated first to further accelerate computation.

4.2 Landscape Analysis

The algorithm outlined in the previous section makes it possible to conduct a simplified form of the landscape analysis.

Our first setting uses $n = 500$ and employs binning to assign parameter values to fitness values. For all but the last bin we use the following sequence of values: [1.0, 1.0, 1.0, 1.0, 6.5, 8.5, 11.5, 16.5], which, as we show later, is the best known binned policy for this problem size. For the last bin, which corresponds to the
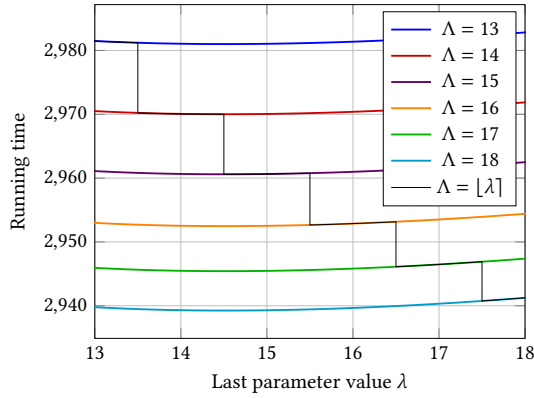
Figure 5: Example runtimes for $n = 500$, **fixed binned policy, and different values of the last λ and population size Λ**

Figure 6: Example runtimes for $n = 500$, **fixed binned policy, and different values of last two λ values**

fitness value of $n - 1 = 499$, we test all possible values in the range $[1; 40]$ with a step of 0.1. Additionally, for every half-integer value (that is, $x + 0.5$ for an integer x, which rounds to $x + 1$) we also use the largest value that is smaller than it and is representable in the machine precision (such a value rounds to x).

The plot of the resulting expected runtimes, computed as detailed in the previous section, is shown in Figure 4. One can easily observe the saw-like shape of the plot, which introduces multiple local optima and a complicated search space even when looking for one parameter. Note that the abrupt changes happen at the half-integer values, which correspond to the points when the population size, which is $\lfloor \lambda \rfloor$, changes as the parameter λ increases by a negligible amount. Note that a similar thing happens for every other bin value, and even if an arbitrary-shaped policy is sought for.

To investigate this effect further, we consider the similar setting, but this time we vary both λ and the population size Λ independently. Figure 5 shows a closeup to the region of $\lambda \in [13; 18]$ and population sizes $\Lambda \in [13..18]$. One can see that when the population size Λ is fixed, the plot is smooth with respect to λ as expected. However, when only the combinations $\Lambda = \lfloor \lambda \rfloor$ are considered, it can happen that the best λ is at either of the interval endpoints or somewhere in the middle (all these cases are covered in Figure 5).

All in all, this effect makes it complicated to compute optimal policies, either numerically or analytically. What is more, it may contribute significantly to the difficulty for the parameter tuning problem observed in Section 3 for IRACE.

Our final setting investigates the effect of interaction between parameter values. Different to the first one, we consider changing the parameter values for the two last bins. The result is represented as a heatmap in Figure 6 for parameters close to the optimal ones, which features a clear grid pattern. While within each cell one can observe gradual color change in directions that are not parallel to coordinate axes, indicating nonlinear interactions between the parameters that might be hard to capture, it is also clear that the effects from rounding — that is, multidimensional ruggedness of the landscape — are at least as large.
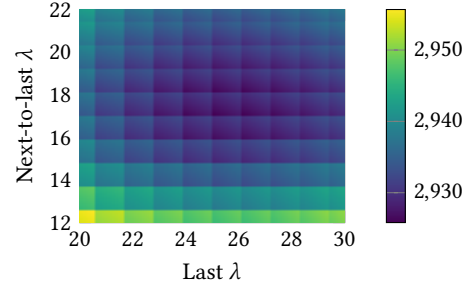
4.3 Computing the Best Policies

In order to compute the best mapping $\lambda(f)$ from fitness values to parameters values, given the problem size n, we modify the dynamic programming approach detailed in Section 4.1 to compute the best λ for a given fitness f assuming this has been done for all higher fitness values. This is done in line with the previous research [13, 15], however, due to the nature of the dependency on parameters found in Section 4.2, a more complicated optimization problem needs to be solved for each fitness value.

Based on the preliminary computations involving smaller problem sizes, we consider intervals of the form $[x - 0.5; x + 0.5)$ for integer x, with an exception of the first and last intervals $[1; 1.5)$ and $[n - 0.5; n]$. The intervals open from the above are treated as closed intervals with the upper boundary replaced by the largest 64-bit floating-point value strictly smaller than that boundary. Within each interval $[\lambda_1, \lambda_2]$, we evaluate four values $\lambda_1, \lambda_1 + \varepsilon, \lambda_2 - \varepsilon, \lambda_2$ with $\varepsilon = 10^{-8}$ to determine whether the optimum is on the boundary or within the interval. In the latter case, we run a variant of ternary search suitable for parallel evaluation to find the best value numerically. Finally, to save computational efforts further, we reduce the number of intervals checked by observing that the optimum is in one of the intervals corresponding to:

- population size 1 or 2;
- the end of a sequence of population sizes, starting from 2 upwards, such that the runtimes decrease, which is tested if the runtime for population size 3 is smaller than for population size 2 (this case is typical for large fitness values);
- the end of a sequence of population sizes, starting from n downwards, such that the runtimes decrease, which is tested if the runtimes for population sizes $n, n - n/8, n - n/4, n - n/2$ are not monotonically decreasing (this case is typical for small fitness values less than $n/3$).

Figure 7 shows the best dynamic policy for $n = 1000$. There are clearly two regions with non-trivial best parameter values. The one closer to the optimum (which is on the left) is quite close to the theoretic value, $\sqrt{n/(n - f(x))}$, which is also illustrated subsequently in Figure 8. All the best parameters that are greater than one are half-integer in this case (the ones that round up). The other region that is far from the optimum does not contribute much to the expected running time. It is clearly the result of accepting the best mutation offspring if it is better than the parent. In this region,
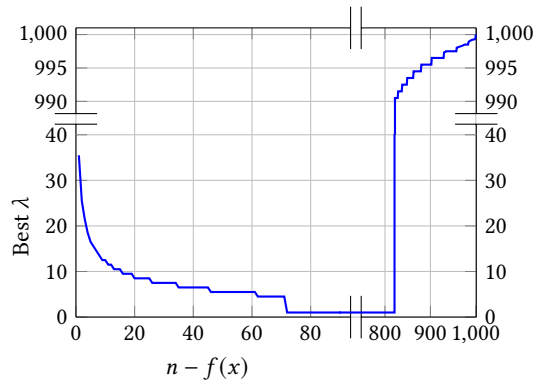
Figure 7: Best dynamic policy for $n = 1000$

Table 1: Expected running time of the numerically approximated baseline policies, rounded to two digits after the decimal comma

n	best	best binned	bin values
500	2916.94	2925.52	1, 1, 1, 1, 6.5, 8.5, 11.5, 16.5, 24.5
1000	5975.81	5994.89	1, 1, 1, 1, 6.5, 8.5, 11.5, 16.5, 23.5, 35.5
2000	12157.62	12197.66	1, 1, 1, 1, 6.5, 8.5, 11.5, 16.5, 22.5, 32.5, 49.5

Table 2: Expected runtime of the best approximated policies, the default one and the best dynamic-tuning one

n	best	best binned	default	tuned
500	2916.94	2925.52	3224.89	3249.09
1000	5975.81	5994.89	6586.67	6512.32
2000	12157.62	12197.66	13386.44	12703.88
3000	18375.48	18435.71	20128.97	19411.0

half-integer values (here, the ones that round down) are also typical, but there are also many occurrences of less trivial values, which signifies the necessity of performing proper optimization within the intervals.

It may be tempting to use dynamic programming in a similar way to compute also the best binned policies. However, this time it is not so straightforward. Indeed, if we consider the i-th bin and assume that bins $i+1$ up are solved, we can fix some λ and compute the expected runtime starting from each fitness value belonging to the i-th bin. However, the choice of parameters for the *previous* bins may affect the probability that the algorithm first hits the i-th bin at certain fitness value f, which would influence the most efficient choice of λ. As a result, we need to optimize all the parameters *simultaneously*.

For that reason, we use the ideas from [16] and employ a numeric optimizer, namely separable CMA-ES [44], to find the suitable parameter values for the bins. Based on the insights from Section 4.2, we represent each parameter λ as two variables, λ_i and λ_f, such that $\lambda = \max\{1, \min\{n, \lfloor \lambda_i \cdot n \rfloor + \lambda_f - 0.5\}\}$. These new values are box-constrained, $0 \leq \lambda_i, \lambda_f < 1$. This approach makes the optimization landscape smoother, resulting in more stable convergence.

The best results of our optimization attempts for both kinds of policies are presented in Table 1 together with the obtained parameter values for the bins. Though, strictly speaking, we have no optimality guarantees, we are still pretty confident that the presented values are not just the upper bounds, but rather represent the truly optimal policies up to available precision.

5 FINAL COMPARISON

Table 2 shows the exact expected running times of the best known policies obtained from Section 4.3 for both unrestricted (best) and binned versions (best binned), the final configuration found by the dynamic tuning with binning and cascading with the largest number of bins (tuned), and THEORY (default). The tunings outperform the default policy for $n \geq 1000$, this is consistent with our empirical evaluations in Section 3. However, there is still a rather large gap to the best policies found by our problem-specific customized search methods, which indicates room for further improvement in our tuning approach.

Figure 8 shows the policies of those four settings. We only plot λ values for $f(x) \geq n/2$ as this is the most interesting region (a random initial solution often has fitness around $n/2$). The x-axis is plotted in logarithmic scale of $n - f(x)$ to allow zoom-in effect on the more difficult regions of the objective function. A noticeable observation is that THEORY almost always underestimates the best λ values in those more difficult areas of the search, and that may be the reason for its worse performance compared to the others. The dynamic tuning, on the other hand, is able to produce policies that are generally close to the best binned policies up until a point. When the fitness is close to the optimal, the tuned policies start diverging from the best λ values, indicating the limitation of IRACE in detecting the importance of those last few bins in order to tune them properly.

6 INSIGHTS INTO FUTURE WORK

The ruggedness of the parameter search landscape highlighted in Section 4.2, and particularly the algorithmic features that cause it, made us believe that there is a rich field for future work considering the design principles of evolutionary algorithms. Rather than just outlining our thoughts, in this section we consider to perform a small series of illustrative experiments that highlight them better.

6.1 Tunability

Similarly to the concept of *testability* in engineering disciplines, we want to bring to the focus the concept of *tunability* of evolutionary algorithms. Many design choices (such as whether or not the best mutation offspring is compared to the parent, or whether or not we can sample an offspring which is equal to the parent) look rather innocent and straightforward, and are often done to improve the performance, but they can promptly turn the parameter landscape from a smooth to a complicated rugged surface. For this reason, we think that new evolutionary algorithms should be designed with care about how friendly they are to tuning procedures, that is, with *tunability* in their design. In the context of the $(1 + (\lambda, \lambda))$ GA,
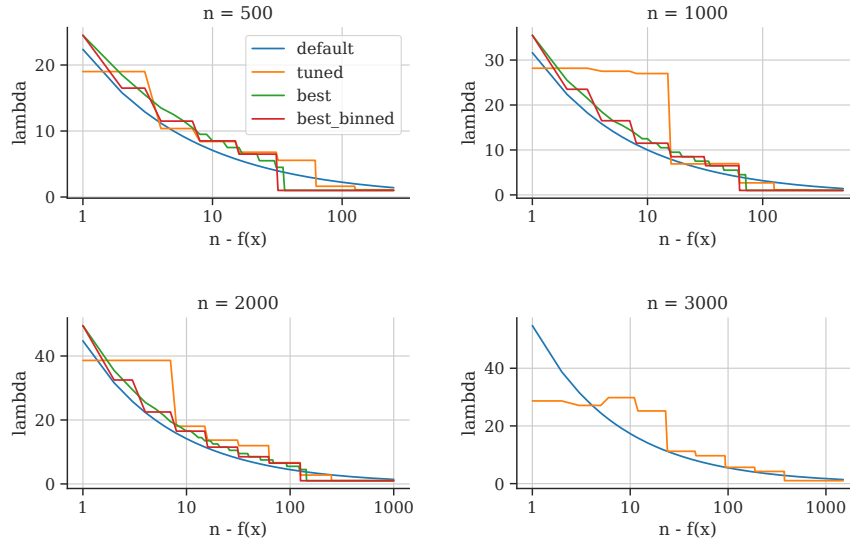
Figure 8: The best approximated policies (`best` and `best_binned`), the default one and the best dynamic one found by the tuning.

any deterministic rounding of the parameter λ to obtain the integer population size Λ is likely to be very disruptive to parameter tuning, and some other procedures may be sought for.

To illustrate this point, we compare the version of the $(1 + (\lambda, \lambda))$ GA studied in this paper with one of the versions from [8], which employs stochastic rounding. Here, $\Lambda = \lfloor \lambda \rfloor$ is taken with probability $\lceil \lambda \rceil - \lambda$ and $\Lambda = \lceil \lambda \rceil$ otherwise. We use the problem size $n = 100$, the binning policy, and optimize the parameter values for the bins using separable CMA-ES [44], but in a straightforward way unlike Section 4.3, that is, each decision variable corresponds to the parameter value. The population size of CMA-ES is 100, and we run optimization for 200 iterations, or until the internal variables of CMA-ES degenerate. Figure 9 shows the convergence plots with regards to the fitness value, whereas Figure 10 does it for the Euclidean distance to the optimum. Clearly, stochastic rounding shows a much "cleaner" convergence without getting stuck in local optima. It also seems somewhat less sensitive with respect to the parameter values around the optimum. We expect that designing algorithms with tunability in mind can bring them similar benefits.

6.2 Control of multiple parameters

De-coupling of the parameters of the $(1 + (\lambda, \lambda))$ GA is interesting from both a theoretical and an empirical point of view. Following the approaches suggested in [8, 19, 22], for example, one could explicitly ask to control the population size for the mutation phase, the mutation strength, the population size for the crossover phase, the crossover bias, etc. The method proposed in Section 4.1 can be used to model this as well. In a small experiment, we tune the parameter λ that controls the mutation strength and the crossover bias, and another parameter Λ that defines the population size for both the mutation and the crossover phases. Just as above, we use
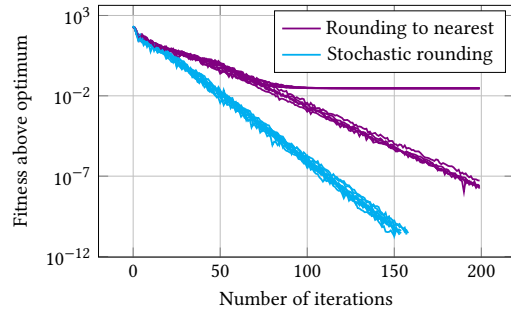

Figure 9: Convergence of parameter optimization for different roundings in $(1 + (\lambda, \lambda))$ GA in terms of fitness values. 10 runs has been executed for each rounding.
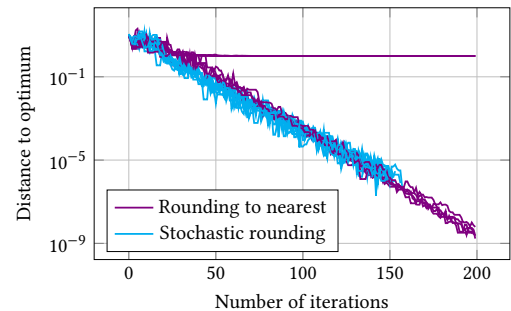

Figure 10: Convergence of parameter optimization for different roundings in $(1 + (\lambda, \lambda))$ GA in terms of distance to the optimum. 10 runs has been executed for each rounding.

Table 3: Optimal policies for $n = 100$ and three approaches to coupling parameters in the $(1+(\lambda, \lambda))$ GA. All numbers are rounded to four digits after the decimal point

Coupling	Runtime	Param	Bin						
			$[0; 50]$	$[51; 75]$	$[76; 88]$	$[89; 94]$	$[95; 97]$	$[98; 99]$	100
Rounding to nearest	534.3011	λ	1.0	1.0	1.0	3.5	5.5	7.5	10.5
		Λ	1	1	1	4	6	8	11
Stochastic rounding	540.7504	λ, Λ	1.0	1.0	1.0	4.0	6.0	8.0688	11.2628
Decoupled parameters	488.9785	λ	83.9611	1.0	1.0	1.4670	3.0433	4.8120	6.9308
		Λ	3	1	1	5	8	12	19

$n = 100$, the binning policy, and optimize the parameters using separable CMA-ES.

Table 3 shows the results and compares them with the previously discussed configuration. While rounding to the nearest integer performs slightly better than stochastic rounding in terms of the runtime of the $(1 + (\lambda, \lambda))$ GA, it appears to be that it happens because of λ and Λ being coupled, and the decoupled version wins by a large margin. Note that the decoupled version favors much larger population sizes Λ and much smaller variation strengths λ towards the optimum. However, it benefits from small population sizes and large mutation strength of almost 0.84 in the first bin, which can essentially flip all the bits at once. Note that it is impossible to benefit from this behavior with tightly coupled parameters.

7 CONCLUSION AND OUTLOOK

In this work, we have evaluated some basic properties of population size control of the $(1 + (\lambda, \lambda))$ GA solving ONEMAX as a DAC benchmark. We empirically evaluated the effectiveness of an automated algorithm configuration approach using IRACE as a DAC solving method. Our results suggest that such approach has the potential of learning well-performing parameter control policies, but it has to be implemented carefully. More concretely, a naive implementation would completely fail to get anywhere close to a good policy on our benchmark, but an improved version with binning and cascading can help to boost the performance considerably.

Our work introduces an alternative way to train control policies in DAC settings, a problem that was previously studied predominantly from the viewpoint of reinforcement learning (RL) [9, 11, 47] or (in numerical optimization) using exploratory landscape analysis [38]. We believe that each one of them has different strengths and weaknesses. Having a rich collection of DAC benchmarks with well-understood ground-truth information allows proper investigation of the potential and the limitations of each family of DAC approaches. Our work contributes to such benchmarks.

We did not compare our results with the RL-based approaches in [11, 47] as the RL algorithms used in those work (DDQN [50]) assume a discrete action space where every action value is treated as categorical, while in our setting the parameter being controlled (λ) is numerical. We did, however, conduct some preliminary experiments with PPO [45], a well-known RL approach that works with both discrete and continuous action spaces, but the results were rather disappointing. This may not be too surprising, as deep-RL algorithms are often known to be non-trivial to use and various design choices may have strong impact on their performance [2, 34]. A

thorough study on deep-RL for the benchmark is beyond the scope of this present work. We leave it as a future research question.

From the perspective of analysis and design of evolutionary computation methods, our numerical approximations of the optimal control policies raise a number of interesting questions. It appears that the design choice in the considered version of the $(1 + (\lambda, \lambda))$ GA, namely deterministic rounding of a real-valued parameter to obtain population size, led to a rugged landscape of the parameter search space, which not only complicated the numerical minimization, but probably had an effect on the application of IRACE. A rather simple change, switching to stochastic rounding, made the parameter tuning problem much simpler, but so did optimizing two parameters (the original λ and the population size Λ) separately. As a result of these exercises, we propose considering tunability of evolutionary algorithms when designing them, and note that sometimes tuning more parameters is easier. With this in hand, we can also construct new DAC benchmarks that require to control not only one, but several parameters at the same time [51] – a largely under-explored challenge [1, 36].

ACKNOWLEDGMENTS

REFERENCES

[1] Steven Adriaensen, André Biedenkapp, Gresa Shala, Noor Awad, Theresa Eimer, Marius Lindauer, and Frank Hutter. 2022. Automated Dynamic Algorithm Configuration. *Journal of Artificial Intelligence Research* 75 (2022), 1633–1699. https://doi.org/10.1613/jair.1.13922

[2] Marcin Andrychowicz, Anton Raichuk, Piotr Stańczyk, Manu Orsini, Sertan Girgin, Raphaël Marinier, Leonard Hussenot, Matthieu Geist, Olivier Pietquin, Marcin Michalski, et al. 2021. What matters for on-policy deep actor-critic methods? a large-scale study. In *International conference on learning representations*.

[3] Denis Antipov, Maxim Buzdalov, and Benjamin Doerr. 2022. Fast Mutation in Crossover-Based Algorithms. *Algorithmica* 84, 6 (2022), 1724–1761. https://doi.org/10.1007/s00453-022-00957-5

[4] Denis Antipov, Benjamin Doerr, and Vitalii Karavaev. 2019. A tight runtime analysis for the $(1+(\lambda, \lambda))$ GA on LeadingOnes. In *Proc. of Foundations of Genetic Algorithms (FOGA'19)*. ACM, 169–182. https://doi.org/10.1145/3299904.3340317

[5] Denis Antipov, Benjamin Doerr, and Vitalii Karavaev. 2022. A Rigorous Runtime Analysis of the $(1 + (\lambda, \lambda))$ GA on Jump Functions. *Algorithmica* 84, 6 (2022), 1573–1602. https://doi.org/10.1007/s00453-021-00907-7

[6] Kirill Antonov, Maxim Buzdalov, Arina Buzdalova, and Carola Doerr. 2021. Blending Dynamic Programming with Monte Carlo Simulation for Bounding the Running Time of Evolutionary Algorithms. In *Proc. of IEEE Congress on Evolutionary Computation (CEC'21)*. IEEE, 878–885. https://doi.org/10.1109/CEC45853.2021.9504775 Free version available at https://arxiv.org/abs/2102.11461.

[7] Thomas Bartz-Beielstein, Carola Doerr, Jakob Bossek, Sowmya Chandrasekaran, Tome Eftimov, Andreas Fischbach, Pascal Kerschke, Manuel López-Ibáñez, Katherine M. Malan, Jason H. Moore, Boris Naujoks, Patryk Orzechowski, Vanessa Volz, Markus Wagner, and Thomas Weise. 2020. Benchmarking in Optimization: Best Practice and Open Issues. *CoRR* abs/2007.03488 (2020). arXiv:2007.03488 https://arxiv.org/abs/2007.03488

[8] Anton Bassin and Maxim Buzdalov. 2020. An Experimental Study of Operator Choices in the $(1+(\lambda,\lambda))$ Genetic Algorithm. In *Proceedings of the International Conference on Mathematical Optimization Theory and Operations Research*. Number 1275 in Communications in Computer and Information Science. 320–335. https://doi.org/10.1007/978-3-030-58657-7_26

[9] André Biedenkapp. 2022. *Dynamic algorithm configuration by reinforcement learning*. Ph. D. Dissertation. University of Freiburg, Germany. https://freidok.uni-freiburg.de/data/230869

[10] 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

[11] André Biedenkapp, Nguyen Dang, Martin S. Krejca, Frank Hutter, and Carola Doerr. 2022. Theory-inspired parameter control benchmarks for dynamic algorithm configuration. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO)*. ACM, 766–775. https://doi.org/10.1145/3512290.3528846

[12] 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

[13] 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

[14] Maxim Buzdalov and Benjamin Doerr. 2017. Runtime Analysis of the $(1+(\lambda,\lambda))$ Genetic Algorithm on Random Satisfiable 3-CNF Formulas. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'17)*. ACM, 1343–1350.

[15] 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

[16] 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

[17] Leslie Pérez Cáceres, Manuel López-Ibáñez, Holger Hoos, and Thomas Stützle. 2017. An experimental study of adaptive capping in irace. In *Learning and Intelligent Optimization: 11th International Conference, LION 11, Nizhny Novgorod, Russia, June 19-21, 2017, Revised Selected Papers*. Springer, 235–250.

[18] Deyao Chen, Maxim Buzdalov, Carola Doerr, and Nguyen Dang. 2023. Code and data repository of this paper. https://github.com/de0ch/OLL.

[19] Nguyen Dang and Carola Doerr. 2019. Hyper-parameter tuning for the $(1 + (\lambda,\lambda))$ GA. In *Proc. of Genetic and Evolutionary Computation Conference (GECO'19)*. ACM, 889–897. https://doi.org/10.1145/3321707.3321725

[20] Marcelo de Souza, Marcus Ritt, and Manuel López-Ibáñez. 2022. Capping methods for the automatic configuration of optimization algorithms. *Comput. Oper. Res.* 139 (2022), 105615. https://doi.org/10.1016/j.cor.2021.105615

[21] 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

[22] 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

[23] 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.

[24] Benjamin Doerr, Carola Doerr, and Franziska Ebel. 2015. From black-box complexity to designing new genetic algorithms. *Theoretical Computer Science* 567 (2015), 87 – 104.

[25] Benjamin Doerr, Huu Phuoc Le, Régis Makhmara, and Ta Duy Nguyen. 2017. Fast genetic algorithms. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'17)*. ACM, 777–784. https://doi.org/10.1145/3071178.3071301

[26] Benjamin Doerr and Carola Winzen. 2014. Playing Mastermind with Constant-Size Memory. *Theory of Computing Systems* 55 (2014), 658–684.

[27] Carola Doerr. 2020. Complexity Theory for Black-Box Optimization Heuristics. In *Theory of Evolutionary Computation: Recent Developments in Discrete Optimization*. Springer, 133–212.

[28] 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

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

[30] 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

[31] 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

[32] Brian W. Goldman and William F. Punch. 2015. Fast and Efficient Black Box Optimization Using the Parameter-less Population Pyramid. *Evolutionary Computation* 23 (2015), 451–479.

[33] Nikolaus Hansen, Anne Auger, Raymond Ros, Olaf Mersmann, Tea Tušar, and Dimo Brockhoff. 2020. COCO: a platform for comparing continuous optimizers in a black-box setting. *Optimization Methods and Software* (2020), 1–31. https://doi.org/10.1080/10556788.2020.1808977 arXiv:https://doi.org/10.1080/10556788.2020.1808977

[34] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. 2018. Deep reinforcement learning that matters. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 32.

[35] Frank Hutter, Manuel López-Ibáñez, Chris Fawcett, Marius Lindauer, Holger H Hoos, Kevin Leyton-Brown, and Thomas Stützle. 2014. AClib: A benchmark library for algorithm configuration. In *Learning and Intelligent Optimization: 8th International Conference, Lion 8, Gainesville, FL, USA, February 16-21, 2014. Revised Selected Papers 8*. Springer, 36–40.

[36] 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.

[37] Stefan Kern, Sibylle D. Müller, Nikolaus Hansen, Dirk Büche, Jiri Ocenasek, and Petros Koumoutsakos. 2004. Learning probability distributions in continuous evolutionary algorithms - a comparative review. *Natural Computing* 3 (2004), 77–112.

[38] Ana Kostovska, Anja Jankovic, Diederick Vermetten, Jacob de Nobel, Hao Wang, Tome Eftimov, and Carola Doerr. 2022. Per-run Algorithm Selection with Warm-starting using Trajectory-based Features. In *Parallel Problem Solving from Nature (PPSN) (LNCS, Vol. 13398)*. Springer, 46–60. https://doi.org/10.1007/978-3-031-14714-2_4 Free version available at https://arxiv.org/abs/2204.09483.

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

[40] Manuel López-Ibáñez, Jürgen Branke, and Luís Paquete. 2021. Reproducibility in Evolutionary Computation. *ACM Trans. Evol. Learn. Optim.* 1, 4 (2021), 14:1–14:21. https://doi.org/10.1145/3466624

[41] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. 2016. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives* 3 (2016), 43–58.

[42] Yasha Pushak and Holger H. Hoos. 2018. Algorithm Configuration Landscapes: - More Benign Than Expected?. In *Proc. of Parallel Problem Solving from Nature (LNCS, Vol. 11102)*. Springer, 271–283. https://doi.org/10.1007/978-3-319-99259-4_22

[43] Yasha Pushak and Holger H. Hoos. 2022. AutoML Loss Landscapes. *ACM Trans. Evol. Learn. Optim.* 2, 3 (2022), 10:1–10:30. https://doi.org/10.1145/3558774

[44] Raymond Ros and Nikolaus Hansen. 2008. A simple modification in CMA-ES achieving linear time and space complexity. In *Parallel Problem Solving from Nature – PPSN X*. Number 5199 in Lecture Notes in Computer Science. 296–305.

[45] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. *CoRR* abs/1707.06347 (2017). http://arxiv.org/abs/1707.06347

[46] Gresa Shala, André Biedenkapp, Noor Awad, Steven Adriaensen, Marius Lindauer, and Frank Hutter. 2020. Learning Step-Size Adaptation in CMA-ES. In *Proc. of Parallel Problem Solving from Nature (PPSN'20) (LNCS, Vol. 12270)*. Springer, 691–706.

[47] 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

[48] David Speck, André Biedenkapp, Frank Hutter, Robert Mattmüller, and Marius Lindauer. 2021. Learning Heuristic Selection with Dynamic Algorithm Configuration. In *Proc. of International Conference on Automated Planning and Scheduling (ICAPS)*. AAAI Press, 597–605. https://ojs.aaai.org/index.php/ICAPS/article/view/16008

[49] Michele Tessari and Giovanni Iacca. 2022. Reinforcement learning based adaptive metaheuristics. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. 1854–1861.

[50] Hado Van Hasselt, Arthur Guez, and David Silver. 2016. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial*

intelligence, Vol. 30.

[51] Ke Xue, Jiacheng Xu, Lei Yuan, Miqing Li, Chao Qian, Zongzhang Zhang, and Yang Yu. 2022. Multi-agent Dynamic Algorithm Configuration. In *Advances in Neural Information Processing Systems 35 (NeurIPS'22)*. New Orleans, LA.

[52] Wenjie Yi, Rong Qu, and Licheng Jiao. 2023. Automated algorithm design using proximal policy optimisation with identified features. *Expert Systems with Applications* 216 (2023), 119461.

[53] Wenjie Yi, Rong Qu, Licheng Jiao, and Ben Niu. 2022. Automated Design of Metaheuristics Using Reinforcement Learning within a Novel General Search Framework. *IEEE Transactions on Evolutionary Computation* (2022).

[54] Yuchang Zhang, Ruibin Bai, Rong Qu, Chaofan Tu, and Jiahuan Jin. 2022. A deep reinforcement learning based hyper-heuristic for combinatorial optimisation with uncertainties. *European Journal of Operational Research* 300, 2 (2022), 418–427.