Hyper-parameter tuning for the (1 + (λ, λ)) GA

Nguyen Dang, Carola Doerr

Hyper-Parameter Tuning for the $(1 + (\lambda, \lambda))$ GA

Nguyen Dang
School of Computer Science, University of St-Andrews, UK

Carola Doerr
LIP6, Sorbonne Université, CNRS, Paris, France

ABSTRACT

It is known that the $(1 + (\lambda, \lambda))$ Genetic Algorithm (GA) with self-adjusting parameter choices achieves a linear expected optimization time on OneMax if its hyper-parameters are suitably chosen. However, it is not very well understood how the hyper-parameter settings influences the overall performance of the $(1 + (\lambda, \lambda))$ GA. Analyzing such multi-dimensional dependencies precisely is at the edge of what running time analysis can offer. To make a step forward on this question, we present an in-depth empirical study of the self-adjusting $(1 + (\lambda, \lambda))$ GA and its hyper-parameters. We show, among many other results, that a 15% reduction of the average running time is possible by a slightly different setup, which allows non-identical offspring population sizes of mutation and crossover phase, and more flexibility in the choice of mutation rate and crossover bias – a generalization which may be of independent interest. We also show indication that the parametrization of mutation rate and crossover bias derived by theoretical means for the static variant of the $(1 + (\lambda, \lambda))$ GA extends to the non-static case.

CCS CONCEPTS

• **Theory of computation → Random search heuristics**.

1 INTRODUCTION

The $(1 + (\lambda, \lambda))$ Genetic Algorithm (GA) is a crossover-based evolutionary algorithm that was introduced in [13] to demonstrate that the idea of recombining previously evaluated solutions can be beneficial also on very smooth functions. More precisely, it was proven in [11, 13] that the $(1 + (\lambda, \lambda))$ GA achieves an $o(n \log n)$ expected optimization time on OneMax, the problem of maximizing functions of the type $f_z : \{0, 1\}^n \to \mathbb{R}, x \mapsto |\{1 \leq i \leq n \mid x_i = z_i\}|$. All purely mutation-based algorithms, in contrast, are known to require $\Omega(n \log n)$ function evaluations, on average, to optimize these functions [14, 23].

The $(1 + (\lambda, \lambda))$ GA has three parameters, the population size λ of mutation and crossover phase, the mutation rate p, and the crossover bias c. It was shown in [13] that an asymptotically optimal linear expected running time can be achieved by the $(1 + (\lambda, \lambda))$ GA when choosing these parameters in an optimal way, which depends on the fitness of a current-best solution. This result was extended in [11] to a self-adjusting variant of the $(1 + (\lambda, \lambda))$ GA, which uses a fixed parametrization $p = \lambda/n$, $c = 1/\lambda$, and an adaptive success-based choice of λ. More precisely, in the self-adjusting $(1 + (\lambda, \lambda))$ GA the parameter λ is chosen according to a one-fifth success rule, which decreases λ to λ/F if an iteration has produced a strictly better solution, and increases λ to $F^{1/4}\lambda$ otherwise. This linear runtime result proven in [11] was the first example where a self-adjusting choice of the parameter values could be rigorously shown to outperform any possible static setting.

Despite these theoretically appealing results, the performances reported in the original work introducing this algorithm [13] are rather disappointing in that they are much worse than those of Randomized Local Search for all tested problem dimensions up to $n = 5\,000$. It was pointed out in [9] that this is partially due to a sub-optimal implementation; the average optimization times reduce drastically when enforcing that at least one bit is flipped in the mutation phase. In this case, the self-adjusting $(1 + (\lambda, \lambda))$ GA starts to outperform RLS already for dimensions around 1 000. Another possible reason lies in the fact that the hyper-parameters of the self-adjusting $(1 + (\lambda, \lambda))$ GA had not been optimized. In [13] the authors had taken some default values from the literature, and show only some very basic sensitivity analysis with respect to the update strength, but not with respect to any of the other parameters such as the success rate. In [11] some general advice on choosing the hyper-parameters is given, but their influence on the explicit running time is not discussed, mostly due to missing precision in the available results, which state the asymptotic linear order only, but not the leading constants or lower order terms. Also the update strength F for which the linear running time is obtained is only shown to exist, but not made explicit in [11].

To shed light on the question how much performance can be gained by choosing the hyper-parameters of the $(1 + (\lambda, \lambda))$ GA with more care, we present in this work a detailed empirical evaluation of this parameter tuning question. Our first finding is that the default setting studied in [13], which uses update strength $F = 3/2$ and the mentioned 1/5-th success rule is almost optimal. More precisely, we show that for all tested problem dimensions between $n = 500$ and $n = 10\,000$ only marginal gains are possible by choosing different update strengths F and/or a success rule different from 1/5.

We then introduce a more general variant of the $(1 + (\lambda, \lambda))$ GA, in which the offspring population sizes of mutation and crossover phase need not be identical, and in which more flexible choices of mutation strength and crossover bias are possible. This leaves us with a five-dimensional hyper-parameter tuning problem, which we address with the irace software [26]. We thereby find configurations whose average optimization times are around 15% better than that of the default self-adjusting $(1 + (\lambda, \lambda))$ GA, for each of the tested dimensions. The configurations achieving these advantages are quite

stable across all dimensions, so that we are able to derive configurations achieving these gains for all dimensions. We furthermore show that the relative advantage also extends to dimensions 20 000 and 30 000, for which we did not perform any hyper-parameter tuning. This five-dimensional variant of the $(1+(\lambda,\lambda))$ GA is also of independent interest, since it allows much greater flexibility than the standard versions introduced in [11, 13].

We finally study if hyper-parameter tuning of a similarly extended static $(1+(\lambda,\lambda))$ GA can give similar results, or whether the asymptotic discrepancy between non-static and static parameter settings proven in [11] also applies relatively small dimensions. We show that indeed already for the smallest tested dimension, $n = 500$, the average optimization time of the best static setting identified by our methods is around 5% worse than the standard self-adjusting $(1+(\lambda,\lambda))$ GA from [11, 13], and by 22% worse than the best found five-dimensional configuration. This disadvantage increases to 22% and 45% in dimension $n = 10\,000$, respectively, thus showing that not only the advantage of the self-adjusting $(1+(\lambda,\lambda))$ GA kicks in already for small dimensions, but also confirming that the relative advantage increases with increasing problem dimensions.

Apart from introducing the new $(1+(\lambda,\lambda))$ GA variants, which offer much greater flexibility than the standard versions, our work significantly enhance our understanding of the hyper-parameter setting in the $(1+(\lambda,\lambda))$ GA, paving the way for a precise rigorous theoretical analysis. In particular the stable performance of the tuned configurations indicates that a precise running time analysis might be possible. We furthermore learn from our work that the parametrization of the mutation rate and the crossover bias, which were suggested and proven to be asymptotically optimal for the static case in [11], seem to be optimal also in the non-static case with self-adjusting parameter choices. Finally, we also observe that for the generalized dynamic setting $1 : x$ success rules with success rates between 3 to 4 seem to be slightly better than the classic one-fifth success rule with $F = 3/2$.

Broader Context: Parameter Control and Hyper-Parameter Tuning. All iterative optimization heuristics such as EAs, GAs, local search variants, etc. are parametrized algorithms. Choosing the right parameter values is a tedious, but important task, frequently coined the "Achilles' heel of evolutionary computation" [17]. It is well known that choosing the parameter values of different parameter settings can result in much different performances. Extreme cases in which a small constant change in the mutation rate result in super-polynomial performance gaps were shown, for example, in [15, 24].

To guide the user in the parameter selection task, two main approaches have been developed: parameter tuning and parameter control. *Parameter tuning* aims at developing tools that automatize the process of identifying reasonable parameter values, cf. [2, 19, 20, 25, 26] for examples. *Parameter control*, in contrast, aims to not only identify such good values, but to also track the evolution of good configurations during the whole optimization process, thereby achieving additional performance gains over an optimally tuned static configuration, cf. [1, 12, 21] for surveys. In practice, parameter control mechanisms are parametrized themselves, thus introducing *hyper-parameters*, which again need to be chosen by the user or one of the tuning tools mentioned above. This is also the route taken in this present work: in Sections 2 and 3 we will use the iterated racing algorithm *irace* [26] to tune two different sets of hyper-parameters of the self-adjusting $(1+(\lambda,\lambda))$ GA, a two-dimensional and a five-dimensional one. In Section 4 we then tune the four parameters of a generalized static $(1+(\lambda,\lambda))$ GA variant. By comparing the results of these tuning steps, we obtain the mentioned estimates for the relative advantage of the self-adjusting over the best tuned static parameter configuration.

Reproducibility, Raw Data, and Computational Resources. Given the space limitations, we only display selected statistics. We concentrate on reporting average values to match with the available theoretical and empirical results. We recall that in theoretical works the expected optimization time dominates all other performance measures. Selected boxplots for the most relevant configurations are provided in Section 5. Source codes, additional performance statistics, summarizing plots, heatmaps with different colormaps, and raw data can be found on our GitHub repository at [10]. All experiments were run on the HPCaVe cluster [4], whose each node consists of two 12-core Intel Xeon E5 2.3GHz with 128Gb memory.

2 TUNING THE DEFAULT $(1+(\lambda,\lambda))$ GA

Our main interest is in tuning the self-adjusting variant of the $(1+(\lambda,\lambda))$ GA proposed in [13] and analyzed in [11]. As in these works, we regard the performance of this algorithm on the OneMax problem. The OneMax problem is one of the most classic benchmark problems in the evolutionary computation literature. It asks to find a secret string z via calls to the function $f_z : \{0,1\}^n \to \mathbb{R}, x \mapsto |\{1 \le i \le n \mid x_i = z_i\}|$ and is thus identical to the problem of minimizing the Hamming distance to an unknown string $z \in \{0,1\}^n$. It is referred to as "OneMax" in evolutionary computation, since the performance of most EAs (including the $(1+(\lambda,\lambda))$ GA) is identical on any of the functions f_z, and it therefore suffices to study the instance $f_{(1,\dots,1)}$.

It is known that the best possible mutation-based (i.e., formally, the best unary unbiased) black-box algorithms have an expected optimization time on OneMax of order $n \log n$ [14, 23]. The $(1+(\lambda,\lambda))$ GA, in contrast, achieves a linear expected optimization time if its parameters are suitably chosen [11, 13]. Parameter control, i.e., a non-static choice of these parameters, is essential for the linear performance, since the $(1+(\lambda,\lambda))$ GA with static parameter values cannot have an expected optimization time that is of better order than $n\sqrt{\log(n)\log\log\log(n)/\log\log(n)}$, which is super-linear.

2.1 The dynamic $(1+(\lambda,\lambda))$ GA $\mathrm{dyn}(\alpha,\beta,\gamma,A,b)$

The $(1+(\lambda,\lambda))$ GA is a binary unbiased algorithm, i.e., it applies crossover but uses only variation operators that are invariant with respect to the problem representation. We present the pseudo-code of the $(1+(\lambda,\lambda))$ GA in Algorithm 1, in which we denote by nint(.) the nearest integer function, i.e., $\mathrm{nint}(r) = \lfloor (r) \rfloor$ if $r - \lfloor r \rfloor < 1/2$ and $\mathrm{nint}(r) = \lceil (r) \rceil$ otherwise.

The $(1+(\lambda,\lambda))$ GA has two phases, a mutation phase and a crossover phase, followed by a selection step. In the *mutation phase* $\lambda_1 = \mathrm{nint}(\lambda)$ offspring are evaluated. Each of them is sampled by the operator $\mathrm{flip}_\ell(.)$ uniformly at random (u.a.r.) from all the points at a radius ℓ around the current-best solution x. The radius ℓ is sampled from the conditional binomial distribution $\mathrm{Bin}_{>0}(n,p)$,

Algorithm 1: The self-adjusting $(1 + (\lambda, \lambda))$ GA variant $\text{dyn}(\alpha, \beta, \gamma, A, b)$ with five hyper-parameters.

Initialization: Sample $x \in \{0,1\}^n$ u.a.r.;
Initialize $\lambda \leftarrow 1$;
Optimization: for $t = 1, 2, 3, \ldots$ **do**
  Mutation phase:
    Sample ℓ from $\text{Bin}_{>0}(n, p = \alpha\lambda/n)$;
    for $i = 1, \ldots, \lambda_1 = \text{nint}(\lambda)$ **do** $x^{(i)} \leftarrow \text{flip}_\ell(x)$;
    Choose $x' \in \{x^{(1)}, \ldots, x^{(\lambda_1)}\}$ with $f(x') = \max\{f(x^{(1)}), \ldots, f(x^{(\lambda_1)})\}$ u.a.r.;
  Crossover phase:
    for $i = 1, \ldots, \lambda_2 = \text{nint}(\beta\lambda)$ **do**
      $y^{(i)} \leftarrow \text{cross}_{c=\gamma/\lambda}(x, x')$;
    Choose $y \in \{x', y^{(1)}, \ldots, y^{(\lambda_2)}\}$ with $f(y) = \max\{f(x'), f(y^{(1)}), \ldots, f(y^{(\lambda_2)})\}$ u.a.r.;
  Selection and update step:
    if $f(y) > f(x)$ **then** $x \leftarrow y$; $\lambda \leftarrow \max\{b\lambda, 1\}$;
    if $f(y) = f(x)$ **then** $x \leftarrow y$; $\lambda \leftarrow \min\{A\lambda, n-1\}$;
    if $f(y) < f(x)$ **then** $\lambda \leftarrow \min\{A\lambda, n-1\}$;

which assigns to each positive integer $1 \le k \le n$ the probability $\text{Bin}_{>0}(n,p) = \binom{n}{k} p^k (1-p)^{n-k} / (1-(1-p)^n)$. Following the reasoning made in [9] we deviate here from the $(1 + (\lambda, \lambda))$ GA variants investigated in [13], to avoid useless iterations. The variants analyzed in [11, 13] allow $\ell = 0$, which is easily seen to create copies of the parent only. As it cannot advance the search, we enforce $\ell \ge 1$.

In the *crossover* phase, λ_2 offspring are evaluated. They are sampled by the crossover operator $\text{cross}_c(\cdot,\cdot)$, which creates an offspring by copying with probability c, independently for each position, the entry of the second argument, and by copying from the first argument otherwise. We refer to the parameter $0 < c < 1$ as the *crossover bias*. Again following [9], we evaluate only those offspring that differ from both their two parents; i.e., offspring that are merely copies of x or x' do not count towards the cost of the algorithm, since their function values are already known.

In the *selection step*, we replace the parent by its best offspring if the latter is at least as good. When a strict improvement has been found, the value of λ is updated to $\max\{b\lambda, 1\}$. It is increased to $\min\{A\lambda, n-1\}$ otherwise.

Note that in the description above and Algorithm 1 we have deviated from the commonly used representation of the $(1 + (\lambda, \lambda))$ GA, in that we have parametrized the mutation rate as $p = \alpha\lambda/n$, the offspring population size of the crossover phase as $\lambda_2 = \text{nint}(\beta\lambda)$, the crossover bias as $c = \gamma/\lambda$, and in that we allow more flexible update strengths A and b. We thereby obtain a more general variant of the $(1 + (\lambda, \lambda))$ GA, which we will show to outperform the standard self-adjusting one considerably. In this present section, however, we only generalize the update rule, not yet the other parameters. That is, we work in this section only with the $(1 + (\lambda, \lambda))$ GA variant $\text{dyn}(1, 1, 1, A, b)$, which uses $\lambda_1 = \lambda_2$, $p = \lambda/n$, and $c = 1/\lambda$.

In our implementation we always ensure that p and c are at least $1/n$ and at most 0.99, by capping these values if needed. Slightly better performances may be obtained by allowing even smaller p-values, but we put this question aside for this present work.

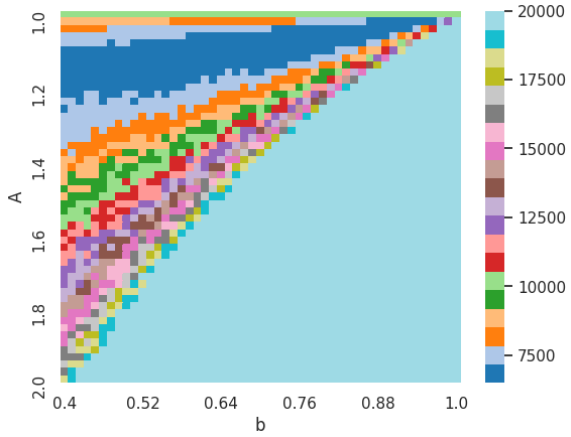

Figure 1: Heatmap for $\text{dyn}(\alpha = \beta = \gamma = 1, A \in [1.02, 2], b \in [0.4, 0.988])$**, average optimization time capped at 20 000**

2.2 Influence of the Update Strengths

As mentioned above, in our first set of experiments we focus on investigating the influence of the update strengths A and b, i.e., we fix $\alpha = \beta = \gamma = 1$ in the notation of Algorithm 1. In [13] it was suggested to set $A = (3/2)^{1/4} \approx 1.11$ and $b = 2/3$. These settings had previously been suggested in [5, 22] in a much different context, but seemed to work well enough for the purposes of [13] and was hence not questioned further in that work (apart from a simple evaluation showing that for $n = 400$ the influence of varying the update strength F within the interval $[1.1, 2]$ is not very pronounced). Note that the choices of A and b correspond to an implicit one-fifth success rule, in the sense that the value of λ is stable if one out of five iterations is successful. The *success rate* (five in this case) can be computed as $1 - \ln(b)/\ln(A)$. We emphasize that for notational convenience we prefer to speak of a success rate x instead of a $1/x$-th success rule.

The heatmap in Figure 1 shows the average running time of the self-adjusting $(1 + (\lambda, \lambda))$ GA in dependence of the update strengths A and b. We considered all combinations of 50 equally spaced values for $A \in \{1.02, 1.04, \ldots, 2\}$ and for $b \in \{0.4, 0.412, \ldots, 0.988\}$ (2 500 hyper-parameter settings). For each setting, we performed 100 independent runs of the algorithm $\text{dyn}(1, 1, 1, A, b)$. Each run has a maximum budget of 150 000 function evaluations. Our results are for problem dimension $n = 1\,000$. To show more structure, we cap in Figure 1 the values at 20 000, other versions with different color schemes and cappings are available at [10].

The best configuration is $(A = 1.06, b = 0.82)$ with an estimated average optimization time of 6 495. This configuration has a success rate of 4.4. The average optimization time of the default variant $\text{dyn}(1, 1, 1, (3/2)^{1/4}, 2/3)$ from [13], denoted by dyn(default) in the following, over 500 runs is 6, 671, and thus only 2.7% worse than $\text{dyn}(1, 1, 1, 1.06, 0.82)$. 29 of the 2 500 tested configurations have a smaller average optimization time than dyn(default), all of them

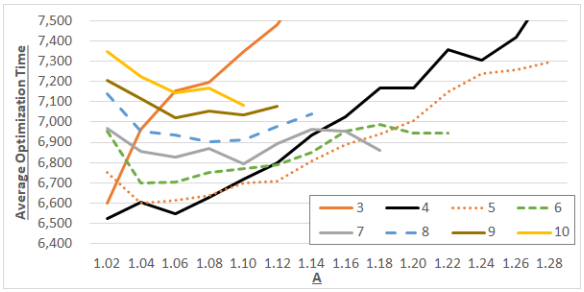

Figure 2: Average optimization time for different success rates, sorted by value of A

Figure 3: Parameter values suggested by irace for the $(1 + (\lambda, \lambda))$ GA variant $\mathrm{dyn}(1, 1, 1, A, b)$. The success rate equals $1 - \ln(b)/\ln(A)$

with A-values at most 1.12 and b-value at least 0.64. 106 configurations are worse by at most 3%, and 188 by at most 5%.

For a more stable comparison, we also ran $\mathrm{dyn}(1, 1, 1, 1.06, 0.82)$ 500 times, and its average optimization time increased to 6 573 for these 500 independent runs, reducing the relative advantage over dyn(default) to 1.5%. Boxplots with information about the runtime distributions can be found in Section 5.

In Figure 2 we plot the average optimization time for different success rates, sorted by the value A. Note that for each tested A-value we have averaged here over all configurations using the same rounded (by $\mathrm{nint}(\cdot)$) success rate. The performance of success rates 1 and 2 is much worse than 7 500 and is therefore not plotted. We plot only results for success rates at most 10, for readability purposes. We see that success rates 4 and 5 are particularly efficient, given the proper values of A. The performance curves for success rates ≥ 4 seem to be roughly U-shaped with different values of A in which the minimum is obtained. It could be worthwhile to extend the mathematical analysis of the $\mathrm{dyn}(1, 1, 1, A, b)$ presented in [11] in order to identify the precise relationship.

2.3 Tuning with irace

The computation of the heatmaps presented above is quite resource-consuming, around 286 CPU days for the full heatmap with 2 500 parameter combinations for $n = 1\,000$. Since we are interested in studying the quality of the $(1 + (\lambda, \lambda))$ GA also for other problem dimensions, we therefore investigate how well automated tuning tools approximate the best known configuration. To this end, we run the configuration tool irace [26] with *adaptive capping* [8] enabled. This new mechanism was recently added to irace to make its search procedure more efficient when optimizing running time or time-compatible performance measurement. We use irace to optimize the configuration of the $\mathrm{dyn}(1, 1, 1, A, b)$ for values of A between 1 and 2.5, and values of b between 0.4 and 1. The allocated budget is 10 and 20 hours of walltime on one 24-core cluster node for $n \leq 5\,000$ and $n > 5\,000$, respectively. This time budget is only a fraction of the ones required by heatmaps (around 3% for $n = 1\,000$).

For $n = 1\,000$ irace suggests to use configuration $(A = 1.071, b = 0.7854)$, which is similar to the one showing best performance in the heatmap. The average optimization time of this configuration is 6, 573 (this number, like all numbers for the configurations suggested by irace are simulated from 500 independent runs each), and thus identical to the best one from the heatmap computations. The suggested configuration corresponds to a 4.52 success rate.

Confident that irace is capable of identifying good parameter settings, we then run irace for various problem dimensions between 500 and 10 000. The by n normalized average optimization time of the suggested configurations are reported in Figure 4 in column $\mathrm{dyn}(1, 1, 1, A, b)$. The chosen A-values are between 1.04 and 1.12 and the b-values are between 0.63 and 0.88, with corresponding success rates between 4.41 and 6.68, cf. Figure 3. We observe a quite stable suggestion for the parameter values.

In Figure 4 we also display, in column dyn(default), the normalized average optimization times of the default setting $(1, 1, 1, (3/2)^{1/4}, 2/3)$. The relative disadvantage of the $\mathrm{dyn}(1, 1, 1, A, b)$ over the dyn(default) ranges from -1.3% to 3.3%. The negative values (in four dimensions) may be due to a suboptimal suggestion of irace, or due to the variance of the algorithms; the relative standard deviation is between 5% and 10%, cf. also the boxplots in Section 5.

We also observe that the normalized average optimization times of dyn(default) increase slightly with increasing problem dimension. Note, however, that this does not necessarily tell us something about the constant factor in the linear running time of this algorithm, although the results indicate that this factor might be larger than 7. Already for $n = 1\,000$ the dyn(default) has a smaller average optimization time than RLS, the relative advantage of dyn(default) is around 2%, and increases to around 31% for $n = 30\,000$.

3 5-DIMENSIONAL PARAMETER TUNING

Next we turn our attention to the five-dimensional $(1 + (\lambda, \lambda))$ GA variant $\mathrm{dyn}(\alpha, \beta, \gamma, A, b)$, in which not only the update strengths A and b are configurable, but also the dependence of $p = \alpha\lambda/n$, $\lambda_2 = \mathrm{nint}(\beta\lambda)$, $c = \gamma/\lambda$. The dependencies of the parameters on λ are based on a theoretical result proven in [11], where it is shown that any static configuration with $\lambda_2 = \lambda_1$ (i.e., $A = b = \beta = 1$) that achieves optimal asymptotic expected performance must necessarily satisfy $p = \Theta(\lambda/n)$ and $\gamma = \Theta(1/\lambda)$.

To investigate how much performance can be gained by this flexibility, and how reasonable parameter values look like, we run again irace, this time using the following parameter ranges: $\alpha \in (1/3, 10)$, $\beta \in (1, 10)$, $\gamma \in (1/3, 10)$, $A \in (1.01, 2.5)$ and $b \in (0.4, 0.99)$.

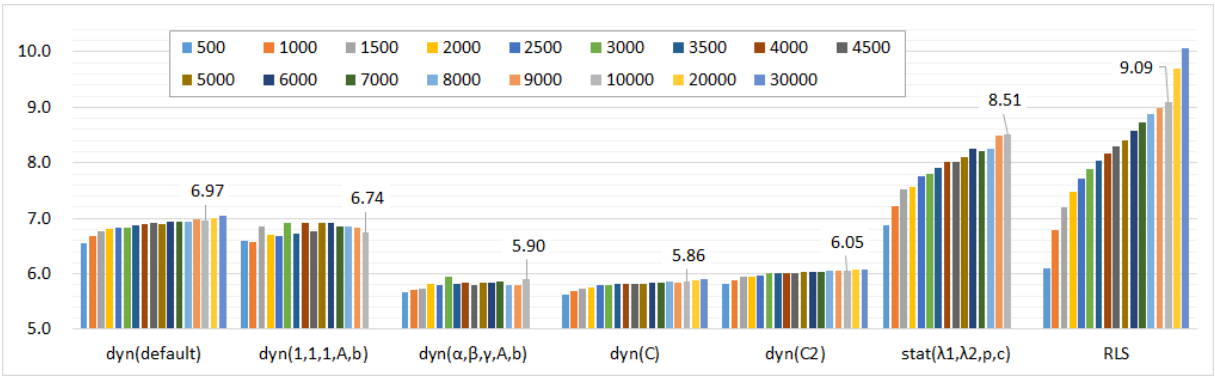

Figure 4: By n normalized average optimization times for 500 independent runs each. For data sets $\mathrm{dyn}(1,1,1,A,b)$, $\mathrm{dyn}(\alpha,\beta,\gamma,A,b)$, and $\mathrm{static}(\lambda_1,\lambda_2,p,c)$ we have taken for each dimension the configuration suggested by irace; the other results are for fixed configurations. Displayed numbers are for $n = 10\,000$.

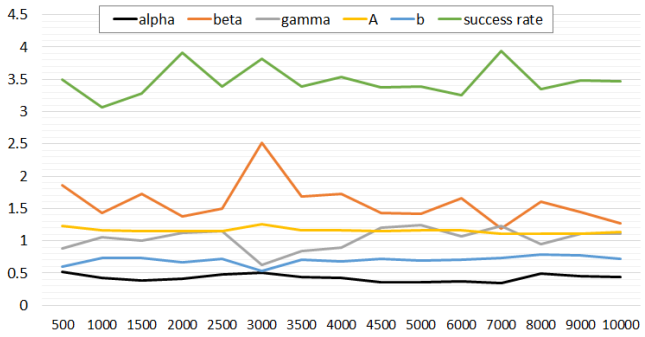

Figure 5: Hyper-parameters and success rate suggested by irace for the $\mathrm{dyn}(\alpha,\beta,\gamma,A,b)$ configuration problem.

The allocated budget is the same as for the $\mathrm{dyn}(1,1,1,A,b)$, i.e., 240 CPU hours for $n \leq 5\,000$ and 480 CPU hours for $n > 5\,000$.

The normalized average running times of the suggested configurations are presented in Column $\mathrm{dyn}(\alpha,\beta,\gamma,A,b)$ in Figure 4. We observe that the parametrization of λ_2, p, and c consistently allows to decrease the average optimization time by around 14%, when measured against the best $\mathrm{dyn}(1,1,1,A,b)$ variant.

3.1 Suggested Hyper-Parameters

The suggested parameter values are displayed in Figure 5. We observe that these are quite stable, in particular when ignoring the 3 000 and 7 000 dimensional configurations. More precisely, irace consistently suggests configurations with $\alpha \approx 0.45$, $\beta \approx 1.6$, $\gamma \approx 1$, $A \approx 1.16$, and $b \approx 0.7$, with corresponding success rates between 3 and 4. These stable values suggest that the parametrization chosen in Algorithm 1 (and originally derived in [11] for the static $(1+(\lambda,\lambda))$ GA) is indeed suitable also for the non-static setting.

In Figure 6 we plot the average optimization time of the configurations tested by irace for $n = 5\,000$ in dependence of each of the five hyper-parameters α,β,γ,A,b and in dependence of the success rate $1-\ln(b)/\ln(A)$. Note that the number of runs differs from point to point, depending on how many evaluations irace has performed for each of these configurations. It is important to note that the capping procedure may stop an algorithm before it has found an optimal solution, in order to save time for the evaluation of more promising configurations. The plotted values are the averages of the successful runs only. An exception to this rule is the chart on the lower right, which shows the whole range of all 2 212 tested configurations; these values are the average time after which the configurations had either found the optimum or were stopped by the capping procedure. We thus see that irace has indeed tested across the whole range of admitted parameter values. Around 38% of all 4 961 runs were stopped before an optimum had been found. However, we already see here that for each parameter there are configurations which use a good value for this parameter, but which shows quite poor overall performance. These results indicate that no parameter alone explains the performance, but that interaction between different parameter values is indeed highly relevant; we will discuss this aspect in more detail below.

Out of the 2 212 tested configurations only 765 configurations had at least one successful run. The averages of all successful runs are plotted in the upper right chart of Figure 6. We observe that the well-performing region of values for each parameter is quite concentrated. The charts on the left and in the middle column zoom into those configurations which had an average optimization time smaller than 35 000. These plots give a good indication where the interesting regions for each parameter are. We also plot the average optimization time in dependence of the success rate and see good performance for success rates between 3 and 4.

For 348 tested configurations only successful runs were reported; i.e., for these configurations none of the runs had been stopped before it had found an optimal solution. When restricting the zoomed plots in Figure 6 to only those 348 configurations, we obtain a very similar picture. We omit a detailed discussion but note that these plots can be found in our repository [10].

The final configuration suggested by irace, $\mathrm{dyn}(0.3594, 1.4128, 1.2379, 1.1672, 0.691)$ has an average optimization time of 29 165 in the 500 independent runs conducted for the values reported in Figure 4. During the irace optimization the estimated average was 28 876 (across 50 runs).

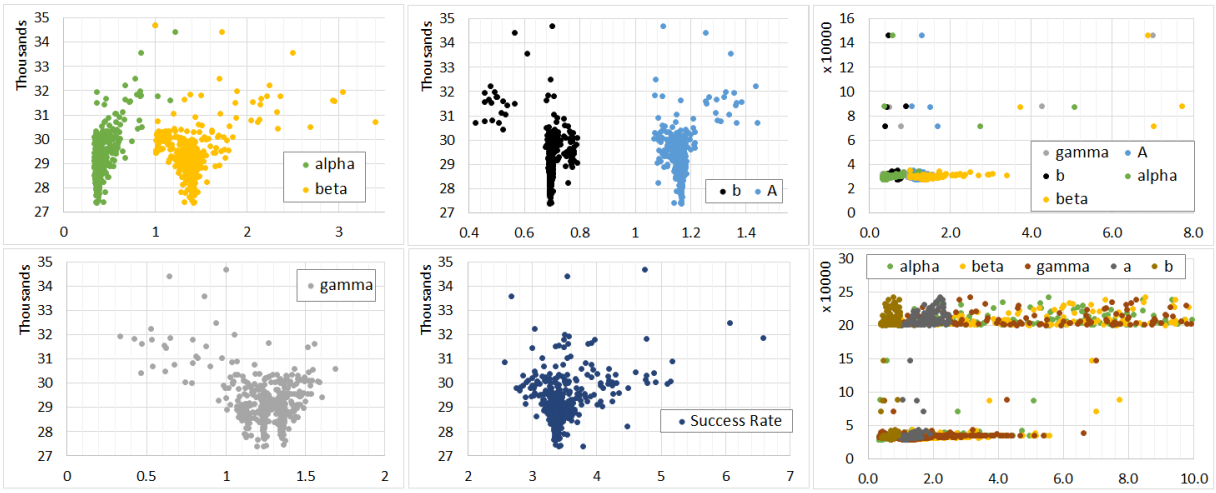

Figure 6: Average running times of different configurations on 5 000-dimensional OneMax. See text for a description.

We see that some of the configurations in Figure 6 have a smaller average optimization time than this latter value. In fact, there are 292 such configurations with at least one successful run and 62 configurations with only successful runs. As we can see from the plots in Figure 6 all these configurations have very similar parameter values. This observation nevertheless raises the question why irace has not suggested one of these presumably better configurations instead. To understand this behavior, we investigate in more detail the working principles of irace, and find two main reasons. One is that the time budget did not allow a further investigation of these configurations, so that statistical evidence that they are indeed superior to the suggested one was not sufficient. A second reason is that the capping suggested in [8] resulted in a somewhat harsh selection of "surviving" configurations. We leave the question if any of the 292 configurations would have been significantly better than the suggested one for future work. Overall, our investigation suggests that some adjustments to irace's default setting might be useful for applications similar to ours, where the performance measure may potentially suffer from high variance.

We next investigated the influence of each parameter on the overall running time. To this end, we have applied *the functional analysis of variance (fANOVA)* [18] on the performance data given by irace. fANOVA can efficiently recognize the importance of both individual algorithm parameters and their interactions through their percentage of contributions on the total performance variance. The software PyImp [3] is used for the analysis. Obtained results are quite consistent among different dimensions. The most important parameter is α, which explains on average 57% of the total variance. The second most important parameter is γ, explaining around 22% of the total variance, on average. Other important effects include pairwise interaction between α and γ or A. Individual parameters and their pairwise interaction effects are able to explain almost 100% of the total variance, so that there is no need to consider higher-order interactions.

Finally, we derive from the suggested parameter values two configurations that we investigate in more detail: dyn$(0.45, 1.6, 1, 1.16, 0.7)$ and dyn$(1/2, 2, 1/2, (3/2)^{1/4}, 2/3)$, which we abbreviate as dyn(C) and dyn(C2), respectively. While dyn(C) consistently shows better performance than dyn(C2), the latter might be easier to analyze by theoretical means. Their normalized average optimization time across all tested dimensions can be found again in Figure 4. They are considerably better than that of dyn(default) $=$ dyn$(1, 1, 1, (3/2)^{1/4}, 2/3)$, between 14% and 16% across all tested dimensions for dyn(C) and between 11% and 13% for dyn(C2). dyn(C2) is between 1% and 4% worse than the (for each dimension independently tuned) best suggested dyn$(\alpha, \beta, \gamma, A, b)$ configuration. For dyn(C) we even observe that the average running times for the 500 runs are smaller than those of dyn$(\alpha, \beta, \gamma, A, b)$ for 10 out of the 15 tested dimensions. The advantages of dyn(C) and dyn(C2) over dyn(default) also translate to larger dimensions, for which we did not perform hyper-parameter tuning. For $n = 20,000$ and $n = 30,000$ the advantage of dyn(C) over dyn(default) are 16% each, and for dyn(C2) a relative advantage of 14% is observed.

3.2 Fixed-Target Analysis

Finally, we address the question where the advantage of the self-adjusting $(1 + (\lambda, \lambda))$ GA over RLS originates from. To this end we perform an empirical fixed-target runtime analysis for two selected configurations, the default configuration dyn(default) and the configuration dyn(C) mentioned above.

The fixed-target running times have been computed with IOHprofiler [16], a recently announced tool which automates the performance analysis of iterative optimization heuristics. The average results of 100 independent runs for $n = 10\,000$ are shown in Figure 7. We observe that RLS is significantly better for almost all target values. In fact, the configuration dyn(C) has better first hitting times than RLS only for OneMax values greater than 9 978, i.e., only for the last 22 target values. We recall from Figure 4 that the average optimization time of dyn(C) is better than that of RLS by around 36% for $n = 10\,000$. To study at which point dyn(C) starts to perform better than RLS, we compute the gradient of the curves plotted in

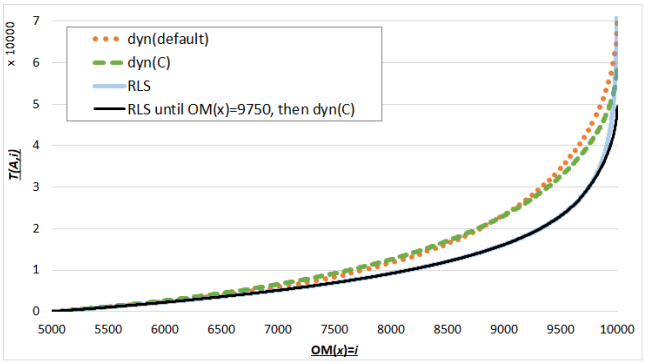

Figure 7: Average fixed-target running times for RLS and two selected dyn$(\alpha, \beta, \gamma, A, b)$ **configurations, capped at 7,100 function evaluations.**

Figure 7, showing that this happens around target value 9 750. For the default configuration dyn(default) the situation is as follows: It is has smaller first hitting time than RLS only for target values $\geq$ 9 995, although its overall average running time is smaller by around 23%. The gradient of dyn(default) is better than that of RLS starting at target value around 9 850. Finally, dyn(C) has smaller average hitting time than dyn(default) for OM-values at least 8, 934, and a better gradient starting at around 8 370. We show in Figure 7 the hypothetical running times of an algorithm that runs RLS until target value $\textsc{Om}(x) = 9\,750$ and then switches to dyn(C). Its average running time is 17% smaller than that of dyn(C), raising the interesting question how to detect such switching points on the fly.

4 TUNING THE STATIC $(1 + (\lambda, \lambda))$ GA

We had concentrated in the previous sections on optimizing dynamic versions of the $(1 + (\lambda, \lambda))$ GA, since the theoretical results guarantee configurations for which linear expected running time can be obtained. In contrast, the best possible expected running time that can be achieved with static parameters $\lambda = \lambda_1 = \lambda_2$, and arbitrary p and c is of order $n\sqrt{\log(n)\log\log\log(n)/\log\log(n)}$ [11]. While this rules out the possibility that there exists a static configuration that performs similarly well as dyn(C) across all dimensions, it is not known to date whether for concrete problem dimensions there exist static configurations that are similar in performance than the dynamic variants dyn(default), dyn(C), or even dyn$(\alpha, \beta, \gamma, A, b)$. We next show that for the tested problem dimensions between 500 and 10 000 this does not seem to be the case.

We study the four-dimensional variant static$(\lambda_1, \lambda_2, p, c)$ presented in Algorithm 2. Following [13], we enforce again that the mutation strength ℓ is strictly greater than zero, by sampling from the conditional distribution $\text{Bin}_{>0}(n, p)$ in line 4. We also allow $\lambda_1 \neq \lambda_2$, which was not the case in [13]. In line with suggestions from [11, 13] we set $p = k/n$, and optimize for integer $k \in \{1, \ldots, 100\}$. We allow the same range for λ_1 and λ_2. The crossover bias c is optimized within the range $[0.01, 1/2]$.

The normalized average running time of the best configuration that irace has been able to identify with its given budget are reported in column static$(\lambda_1, \lambda_2, p = k/n, c)$ of Figure 4. We observe that these running times are significantly larger than those of the

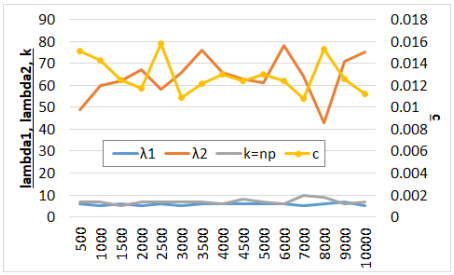

Figure 8: Suggested hyper-parameters for the static$(\lambda_1, \lambda_2, p = k/n, c)$ **by dimension.** λ_1**,** λ_2**, and** k **use the scale on the left,** c **the one on the right.**

dynamic $(1 + (\lambda, \lambda))$ GA variants. The relative disadvantage against the default dynamic variant dyn(default) monotonically increases from around 5% for $n = 500$ to around 22% for $n = 10,000$. Against the best dynamic variant dyn$(\alpha, \beta, \gamma, A, b)$ this relative disadvantage increases from around 21% to around 44%.

We also see from the results in Figure 4 that, with few exceptions, the normalized average running time increases with the problem dimension. This is in line with what the super-linear lower bound proven in [11] suggests (note, however, that the theoretical results for the static $(1 + (\lambda, \lambda))$ GA assumes $\lambda_1 = \lambda_2$). The relative increase of the normalized average running time is smaller than for RLS, again in line with the known theoretical results. The comparison with RLS also shows that the static $(1 + (\lambda, \lambda))$ GA variants start to outperform RLS at problem dimension 3 000. For $n = 10\,000$ the relative advantage of static$(\lambda_1, \lambda_2, p, c)$ over RLS is around 6%.

Finally, we study in Figure 8 the parameter values of the configurations suggested by irace. We observe that across all dimensions λ_1 is significantly smaller than λ_2, which was different for the dynamic $(1 + (\lambda, \lambda))$ GA variants. Both λ_1 and k are relatively stable, with values ranging between 5 and 7 for λ_1 and between 5 and 10 for k. The values of λ_2 fluctuates significantly more, between 43 and 78. The crossover rate is always within the range $[0.0108, 0.0158]$, and thus also quite stable. Since in the original works $c = 1/\lambda$ is assumed, we

Algorithm 2: The static $(1+(\lambda, \lambda))$ GA variant static$(\lambda_1, \lambda_2, p = k/n, c)$ with four static parameters.

1 **Initialization:** Choose $x \in \{0, 1\}^n$ u.a.r.;
2 **Optimization: for** $t = 1, 2, 3, \ldots$ **do**
3 **Mutation phase:**
4 Sample ℓ_1 from $\text{Bin}_{>0}(n, p = k/n)$;
5 **for** $i = 1, \ldots, \lambda_1$ **do** $x^{(i)} \leftarrow \text{flip}_\ell(x)$;
6 Choose $x' \in \{x^{(1)}, \ldots, x^{(\lambda_1)}\}$ with $f(x') = \max\{f(x^{(1)}), \ldots, f(x^{(\lambda_1)})\}$ u.a.r.;
7 **Crossover phase:**
8 **for** $i = 1, \ldots, \lambda_2$ **do** $y^{(i)} \leftarrow \text{cross}_c(x, x')$;
9 Choose $y \in \{y^{(1)}, \ldots, y^{(\lambda_2)}\}$ with $f(y) = \max\{f(y^{(1)}), \ldots, f(y^{(\lambda_2)})\}$ u.a.r.;
10 **Selection step: if** $f(y) \geq f(x)$ **then** $x \leftarrow y$;

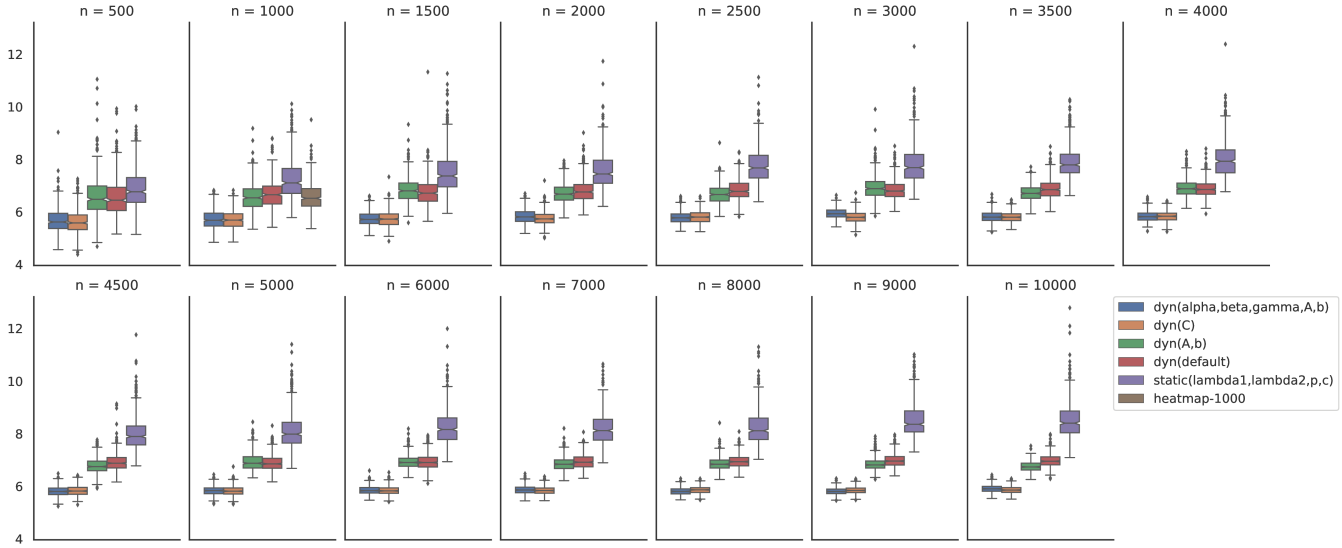

Figure 9: Distribution of the by n normalized optimization times of different $(1 + (\lambda, \lambda))$ GA variants. Heatmap-1000 refers to dyn$(1, 1, 1, 1.06, 0.82)$**, which was the best configuration identified in the heatmap from Section 2.2**

also note that for both $c\lambda_1$ and $c\lambda_2$ the factor between the minimal and maximal value is as small as 1.8 and 1.5, respectively, with no clear monotonic relationship.

5 RUNTIME DISTRIBUTION

In all figures mentioned above we have only considered average values, to obtain results that are more easily comparable with existing theoretical and empirical works. With Figure 9 we address the question how the running times are distributed. This figure provides boxplots for all tested dimensions $\leq$ 10 000. The plots confirm the performance advantages of the five-dimensional dynamic $(1+(\lambda, \lambda))$ GA variants dyn$(\alpha, \beta, \gamma, A, b)$ and dyn(C) over the 2-dimensional versions dyn$(1, 1, 1, A, b)$ and dyn(default). All adaptive versions perform consistently better than the best static version static$(\lambda_1, \lambda_2, p, c)$ in term of both median values and variance. These advantages get more visible as the problem sizes increase. We also perform two types of statistical tests - paired Student t-test and Wilcoxon signed-rank test - between those versions. Results confirm that the difference between them are statistically significant with a confidence level of 99.9%.

6 CONCLUSION

We have presented a very detailed study of the hyper-parameters of the static and the self-adjusting $(1 + (\lambda, \lambda))$ GA on the OneMax problem. Among other results, we have seen that the self-adjusting $(1 + (\lambda, \lambda))$ GA gains only around $1\% - 3\%$ in average optimization time with optimized update strengths A and b. We have then introduced a more flexible variant, the dyn$(\alpha, \beta, \gamma, A, b)$, in which the offspring population sizes of mutation and crossover phase need not be identical, and which offers more flexibility in the choice of the mutation rate and the crossover bias. This has reduced the average optimization times by another 15%. Interestingly, the parameter values by which these performance gains are achieved are quite consistent across all tested dimensions. We then analyzed a configuration in which we fixed the hyper-parameters according to the suggestions made by the tuning in lower dimensions 500 to 10 000, and show that it performs very well also on the 20 000 and 30 000 dimensional OneMax problem.

Our results suggest that the $(1+(\lambda, \lambda))$ GA can gain performance by introducing the additional hyper-parameters. We plan on investigating the gains for other problems, in particular the MaxSAT instances studied in [7]. Since all results shown in this work are quite consistent across all dimensions, we also plan on analyzing the advantages of the dyn$(\alpha, \beta, \gamma, A, b)$ by rigorous means, both in terms of optimization time, but also in terms of more general fixed-target running times. As we have demonstrated in Section 3.2, the latter reveal that the advantage of the $(1 + (\lambda, \lambda))$ GA over RLS lies in the very final phases of the OneMax optimization problem, i.e., when finding improving moves is hard. Efficiently switching between the two algorithms at the time at which the $(1 + (\lambda, \lambda))$ GA starts to outperform RLS carries the potential to reduce the optimization time further. Automating such online algorithm selection is another line of research that we plan to investigate further. Techniques from the literature on parameter control [12, 21], adaptive operator selection [17], and hyper-heuristics [6] might prove useful in this context.

Acknowledgments. This work was supported by the Paris Ile-de-France Region, by a public grant as part of the Investissement d'avenir project ANR-11-LABX-0056-LMH, LabEx LMH, by the European Cooperation in Science and Technology (COST) action CA15140, and by the UK EPSRC grant EP/P015638/1. The simulations were performed at the HPCaVe at UPMC-Sorbonne Université.

REFERENCES

[1] 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.

[2] Carlos Ansótegui, Yuri Malitsky, Horst Samulowitz, Meinolf Sellmann, and Kevin Tierney. 2015. Model-based Genetic Algorithms for Algorithm Configuration. In *Proc. of International Conference on Artificial Intelligence (IJCAI'15)*. AAAI Press, 733–739.

[3] Ml4AAD Group at Freiburg University. [n. d.]. PyImp. https://github.com/automl/ParameterImportance.

[4] HPCaVe Cluster at Sorbonne University. [n. d.]. http://hpcave.upmc.fr/index.php/resources/mesu-beta/.

[5] Anne Auger. 2009. Benchmarking the (1+1) evolution strategy with one-fifth success rule on the BBOB-2009 function testbed. In *Companion Material for Proc. of Genetic and Evolutionary Computation Conference (GECCO'09)*. ACM, 2447–2452.

[6] 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. *Journal of the Operational Research Society* 64 (2013), 1695–1724.

[7] 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.

[8] Leslie Pérez Cáceres, Manuel López-Ibáñez, Holger Hoos, and Thomas Stützle. 2017. An Experimental Study of Adaptive Capping in irace. In *Proc. of Learning and Intelligent Optimization (LION'17) (Lecture Notes in Computer Science)*, Vol. 10556. Springer, 235–250. https://doi.org/10.1007/978-3-319-69404-7_17

[9] Eduardo Carvalho Pinto and Carola Doerr. 2018. A Simple Proof for the Usefulness of Crossover in Black-Box Optimization. In *Proc. of Parallel Problem Solving from Nature (PPSN'18) (Lecture Notes in Computer Science)*, Vol. 11102. Springer, 29–41. https://doi.org/10.1007/978-3-319-99259-4_3 Full version available at http://arxiv.org/abs/1812.00493.

[10] Nguyen Dang and Carola Doerr. 2019. Hyper-Parameter Tuning for the $(1+(\lambda, \lambda))$ GA. *CoRR* (2019). https://doi.org/10.1145/3321707.3321725 arXiv:1904.04608 GitHub repository with project data and plots available at https://github.com/ndangtt/1LLGA.

[11] 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.

[12] Benjamin Doerr and Carola Doerr. 2018. Theory of Parameter Control Mechanisms for Discrete Black-Box Optimization: Provable Performance Gains Through Dynamic Parameter Choices. In *Theory of Randomized Search Heuristics in Discrete Search Spaces*, Benjamin Doerr and Frank Neumann (Eds.). Springer. To appear. Available online at https://arxiv.org/abs/1804.05650.

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

[14] Benjamin Doerr, Carola Doerr, and Jing Yang. 2016. Optimal Parameter Choices via Precise Black-Box Analysis. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'16)*. ACM, 1123–1130.

[15] Benjamin Doerr, Thomas Jansen, Dirk Sudholt, Carola Winzen, and Christine Zarges. 2013. Mutation Rate Matters Even When Optimizing Monotonic Functions. *Evolutionary Computation* 21 (2013), 1–27.

[16] Carola Doerr, Hao Wang, Furong Ye, Sander van Rijn, and Thomas Bäck. 2018. IOHprofiler: A Benchmarking and Profiling Tool for Iterative Optimization Heuristics. *CoRR* abs/1810.05281 (2018). arXiv:1810.05281 http://arxiv.org/abs/1810.05281 IOHprofiler is available at https://github.com/IOHprofiler.

[17] 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

[18] Holger Hoos and Kevin Leyton-Brown. 2014. An efficient approach for assessing hyperparameter importance. In *International Conference on Machine Learning*. 754–762.

[19] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2011. Sequential Model-Based Optimization for General Algorithm Configuration. In *Proc. of Learning and Intelligent Optimization (LION'11)*. Springer, 507–523.

[20] 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.

[21] 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.

[22] 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.

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

[24] Johannes Lengler. 2018. A General Dichotomy of Evolutionary Algorithms on Monotone Functions. In *Proc. of Parallel Problem Solving from Nature (PPSN'18) (Lecture Notes in Computer Science)*, Vol. 11102. Springer, 3–15. https://doi.org/10.1007/978-3-319-99259-4_1

[25] Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2017. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *Journal of Machine Learning Research* 18 (2017), 185:1–185:52. http://jmlr.org/papers/v18/16-558.html

[26] 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.