From understanding genetic drift to a smart-restart parameter-less compact genetic algorithm

Benjamin Doerr, Weijie Zheng

From Understanding Genetic Drift to a Smart-Restart Parameter-less Compact Genetic Algorithm

Benjamin Doerr*
Laboratoire d'Informatique (LIX)
École Polytechnique, CNRS
Institut Polytechnique de Paris
Palaiseau, France

Weijie Zheng*†
Guangdong Provincial Key Laboratory of Brain-inspired Intelligent Computation
Department of Computer Science and Engineering
Southern University of Science and Technology
Shenzhen, China

ABSTRACT

One of the key difficulties in using estimation-of-distribution algorithms is choosing the population sizes appropriately: Too small values lead to genetic drift, which can cause enormous difficulties. In the regime with no genetic drift, however, often the runtime is roughly proportional to the population size, which renders large population sizes inefficient.

Based on a recent quantitative analysis which population sizes lead to genetic drift, we propose a parameter-less version of the compact genetic algorithm that automatically finds a suitable population size without spending too much time in situations unfavorable due to genetic drift.

We prove an easy mathematical runtime guarantee for this algorithm and conduct an extensive experimental analysis on four classic benchmark problems. The former shows that under a natural assumption, our algorithm has a performance similar to the one obtainable from the best population size. The latter confirms that missing the right population size can be highly detrimental and shows that our algorithm as well as a previously proposed parameter-less one based on parallel runs avoids such pitfalls. Comparing the two approaches, ours profits from its ability to abort runs which are likely to be stuck in a genetic drift situation.

CCS CONCEPTS

• **Theory of computation → Theory and algorithms for application domains**; *Theory of randomized search heuristics*;

KEYWORDS

Estimation-of-distribution algorithms; parameter-less algorithm; empirical study; theory

*Both authors contributed equally to this work and both act as corresponding authors.
†Also with School of Computer Science and Technology, University of Science and Technology of China, Hefei, China.

GECCO '20, July 8–12, 2020, Cancún, Mexico

ACM ISBN 978-1-4503-7128-5/20/07.
https://doi.org/10.1145/3377930.3390163

1 INTRODUCTION

Estimation-of-distribution algorithms (EDAs) [26, 32] are a branch of evolutionary algorithms (EAs) that evolve a probabilistic model instead of a population. The update of the probabilistic model is based on the current model and the fitness of a population sampled according to the model. The size of this population is crucial for the performance of the EDA. Taking the UMDA [31] with artificial frequency margins $\{1/n, 1-1/n\}$ optimizing the n-dimensional DeceptiveLeadingBlocks problem as an example, Lehre and Nguyen [27, Theorem 4.9] showed that if the population size is small ($\lambda = \Omega(\log n) \cap o(n)$) and the selective pressure is standard ($\mu/\lambda \geq 14/1000$), then the expected runtime is $\exp(\Omega(\lambda))$. The essential reason for this weak performance, quantified in Doerr and Zheng [16] but observed also in many previous works, is that the small population size leads to strong genetic drift, that is, the random fluctuations of frequencies caused by the random sampling of search points eventually move some sampling frequencies towards a boundary of the frequency range which is not justified by the fitness. Doerr and Krejca's recent work [12] showed that when the population size is large enough, that is, $\lambda = \Omega(n \log n)$ and $\mu = \Theta(\lambda)$, the genetic drift effect is weak and with high probability, the UMDA finds the optimum in $\lambda(n/2 + 2e \ln n)$ fitness evaluations. This runtime bound is roughly proportional to the population size λ. Assuming that this bound describes the true runtime behavior (no lower bound was shown in [12]), we see that a too large population size will again reduce the efficiency of the algorithm.

We refer to the recent survey paper of Krejca and Witt [25] for more runtime analyses of EDAs. For most of the results presented there, a minimum population size is necessary and then the runtime is roughly proportional to the population size. In a word, for many EDAs a too small population size leads to genetic drift, while a too large size results in inefficiency. Choosing the appropriate population size is one of the key difficulties in the practical usage of EDAs.

We note that there have been attempts to define EDAs that are not prone to genetic drift [11, 19], also with promising results, but from the few existing results (essentially only for the OneMax, BinVal, and LeadingOnes benchmarks) it is hard to estimate how promising these ideas are for real-world optimization problems. For this reason, in this works we rather discuss how to set the parameters for the established EDAs.

Parameter tuning and parameter control have successfully been used to find suitable parameter values. However, both approaches

will usually only design problem-specific strategies and often require sophisticated expertise to become successful.

In order to free the practitioner from the task of choosing parameters, researchers have tried to remove the parameters from the algorithm while trying to maintain a good performance, ideally comparable to the one with best parameter choice for the problem to be solved. Such algorithms are called parameter-less[1]. This paper will address the problem of designing a parameter-less cGA. In an early work, Harik and Lobo [21] proposed two strategies to remove the population size of crossover-based genetic algorithms. One basic strategy is doubling the population size and restarting when all individuals' genotypes have become the same. The drawback of this strategy is long runtime when genetic drift becomes detrimental, which is hard to detect. Harik and Lobo proposed a second strategy in which multiple populations with different sizes run simultaneously, smaller population sizes may use more function evaluations, but are removed once their fitness value falls behind the one of larger populations. Their experimental results showed that the algorithm with this second strategy only had a small performance loss over the regular genetic algorithm with optimal parameter settings. Many extensions of this strategy and applications with other optimization algorithms have followed, giving rise to the extended compact genetic algorithm [29], the hierarchical Bayesian optimization algorithm [33], and many others.

Goldman and Punch [20] proposed the parameter-less population pyramid, called P3, to iteratively construct a collection of populations. In P3, the population in the pyramid expands iteratively by first adding a currently not existing solution obtained by some local search strategy into the lowest population, and then utilizing some model-building methods to expand the population in all hierarchies of the pyramid. Since initially no population exists in the pyramid, this algorithm frees the practitioner from specifying a population size. For EDAs, Doerr [6] recently proposed another strategy building a parallel EDA running with exponentially growing population size. With a careful strategy to assign the computational resources, he obtained that under a suitable assumption this parallel EDA only had a logarithmic factor performance loss over the corresponding original EDA using the optimal population size.

Our contribution: The above parameter-less strategies use clever but indirect ways to handle the possibly long wasted time caused by genetic drift. In this work, we aim at a more direct approach by exploiting a recent mathematical analysis which predicts when genetic drift arises. Doerr and Zheng [16] have theoretically analyzed the boundary hitting time caused by the genetic drift. In very simple words, their result indicates that genetic drift in a bit position of the cGA occurs when the runtime of the algorithm exceeds $4\mu^2$, where μ is the hypothetical population size of the cGA. We use this insight to design the following parameter-less version of the cGA. Our parameter-less cGA, called *smart-restart cGA*, is a simple restart process with exponentially growing population size. It stops a run once the risk of genetic drift is deemed too high, based on the analysis in [16].

[1]Not surprisingly, many mechanisms to remove parameters have themselves some parameters. The name *parameter-less* might still be justified when these meta-parameters have a less critical influence on the performance of the algorithm.

Since Doerr and Zheng [16] proved that a neutral frequency reaches the boundaries of the frequency range in an expected number of $4\mu^2$ generations, via Markov's inequality we know that with probability at least $1/2$ a boundary is reached in $8\mu^2$ generations. Since genetic drift affects neutral bits stronger than those subject to a clear fitness signal, we can pessimistically take $8\mu^2$ generations as the termination budget for a cGA run with population size μ.

This heuristic builds only a single frequency. Since there are n frequencies, one may speculate that the first of these reaches a boundary already in $\Theta(\mu^2/\ln n)$ generations. We do not have a fully rigorous analysis showing that the first of the frequencies reaches a boundary in $O(\mu^2/\ln n)$ iterations, but the tail bound in [16] shows that this does not happen earlier and our experiments suggest that taking this smaller budget is indeed often profitable. For this reason, we work with both termination criteria, where for the latter we choose the implicit constant as $c = 0.5$, based on our experimental investigation how the cGA with varying population sizes optimizes various benchmark problems.

For our algorithm, we prove a mathematical runtime guarantee. For this we assume that there are numbers $\tilde{\mu}$ and T such that the cGA with all population size $\mu \geq \tilde{\mu}$ solves the given problem in time μT with sufficiently high probability. Such a runtime behavior is indeed often observed, see, e.g., [25]. We theoretically prove that under this assumption, our smart-restart cGA solves the problem in expected time $\max\{O(\tilde{\mu}^2), O(T^2)\}$ when a termination budget of $8\mu^2$ is used, and in expected time $\max\{O(\tilde{\mu}^2/\ln n), O(T^2 \ln n)\}$ when a termination budget of $\Theta(\mu^2/\ln n)$ is used. Together with the known results that the cGA with all $\mu = \Omega(\sqrt{n}\log n) \cap O(poly(n))$ optimizes the OneMax function and the Jump with jump size $k \leq \frac{1}{20}\ln n - 1$ in time $O(\sqrt{n}\mu)$ with probability $1-o(1)$ [6, 35], this shows that our algorithm with the second termination rule optimizes the Jump and OneMax benchmark in time $O(n \log n)$, which is the asymptotically best performance the cGA can have with an optimal choice of μ.

For an algorithm with the two generation budgets as well as the original cGA and the parallel-run cGA as comparison, we conduct an extensive experimental analysis on the OneMax, LeadingOnes, Jump, and DeceptiveLeadingBlocks functions. Our experimental results indicate that the better runtime of our smart-restart cGAs against parallel-run cGA appears on OneMax and LeadingOnes functions no matter using $8\mu^2$ or $0.5\mu^2/\ln n$ generation budgets, and the better runtime on Jump and DeceptiveLeadingBlocks function when using $0.5\mu^2/\ln n$ generation budget. Our experimental results also show that indeed missing the right population size can be detrimental.

The remainder of this paper is structured as follows. Section 2 introduces the preliminaries including a detailed description of the cGA and the parallel-run cGA. Our proposed smart-restart cGA will be stated in Section 3. Sections 4 and 5 show the theoretical result and experimental analysis respectively. Section 6 concludes our paper.

2 PRELIMINARIES

In this paper, we consider algorithms maximizing pseudo-boolean functions $f : \{0,1\}^n \to \mathbb{R}$. Since our smart-restart cGA is based on the original cGA of Harik, Lobo, and Goldberg [22] and since

we will compare our algorithm with the parallel-run cGA [6], this section will give a brief introduction to these algorithms.

2.1 The Compact Genetic Algorithm

The compact genetic algorithm (cGA) with hypothetical population size μ samples two individuals in each generation and moves the sampling frequencies by an absolute value of $1/\mu$ towards the bit values of the better individual. Usually, in order to avoid frequencies reaching the absorbing boundaries 0 or 1, the artificial margins $1/n$ and $1-1/n$ are utilized, that is, we restrict the frequency values to be in the interval $[1/n, 1-1/n]$. The following Algorithm 1 shows the details. As common in runtime analysis, we do not specify a termination criterion. When talking about the runtime of an algorithm, we mean the first time (measured by the number of fitness evaluations) an optimum was sampled.

Algorithm 1 The cGA to maximize a function $f : \{0,1\}^n \to \mathbb{R}$ with hypothetical population size μ

$p^0 = (\frac{1}{2}, \frac{1}{2}, \dots, \frac{1}{2}) \in [0,1]^n$
for $g = 1, 2, \dots$ **do**
  %%Sample two individuals X_1^g, X_2^g
  for $i = 1, 2$ **do**
    for $j = 1, 2, \dots, n$ **do**
      $X_{i,j}^g \leftarrow 1$ with probability p_j^{g-1} and $X_{i,j}^g \leftarrow 0$ with probability $1 - p_j^{g-1}$.
    end for
  end for
  %%Update of the frequency vector
  if $f(X_1^g) \geq f(X_2^g)$ **then**
    $p' = p^{g-1} + \frac{1}{\mu}(X_1^g - X_2^g)$;
  else
    $p' = p^{g-1} + \frac{1}{\mu}(X_2^g - X_1^g)$;
  end if
  $p^g = \min\{\max\{\frac{1}{n}, p'\}, 1 - \frac{1}{n}\}$;
end for

2.2 The Parallel-run cGA

The parallel EDA framework was proposed by Doerr [6] as a side result when discussing the connection between runtime bounds that hold with high probability and the expected runtime. For the cGA, this framework yields the following *parallel-run cGA*. In the initial round $\ell = 1$, we start process $\ell = 1$ to run the cGA with population size $\mu = 2^{\ell-1}$ for 1 generation. In round $\ell = 2, 3, \dots$, all running processes $j = 1, \dots, \ell - 1$ run $2^{\ell-1}$ generations and then we start process ℓ to run the cGA with population size $\mu = 2^{\ell-1}$ for $\sum_{i=0}^{\ell-1} 2^i$ generations. The algorithm terminates once any process has found the optimum. Algorithm 2 shows the details of the parallel-run cGA.

Based on the following assumption, Doerr [6] proved that the expected runtime for this parallel-run cGA is at most $6\tilde{\mu}T(\log_2(\tilde{\mu}T) + 3)$.

Assumption [6]: Consider using the cGA with population size μ to maximize a given function f. Assume that there are unknown $\tilde{\mu}$ and T such that the cGA for all population sizes $\mu \geq \tilde{\mu}$ optimizes this function f in μT fitness evaluations with probability at least $\frac{3}{4}$.

Algorithm 2 The parallel-run cGA to maximize a function $f : \{0,1\}^n \to \mathbb{R}$

Process 1 runs cGA (Algorithm 1) with population size $\mu = 1$ for 1 generation.
for Round $\ell = 2, \dots$ **do**
  Processes $1, \dots, \ell - 1$ continue to run for another $2^{\ell-1}$ generations, one process after the other one.
  Start process ℓ to run cGA (Algorithm 1) with population size $\mu = 2^{\ell-1}$ and run it for $\sum_{i=0}^{\ell-1} 2^i$ generations.
end for

3 THE SMART-RESTART CGA

In this section, we introduce our parameter-less cGA, called *smart-restart cGA*. In contrast to the parallel-run cGA it does not run processes in parallel, which is an advantage from the implementation point of view. The main advantage we aim for is that by predicting when runs become hopeless, we can abort these runs and save runtime.

To detect such a hopeless situation, we use the first tight quantification of the genetic drift effect of the EDAs by Doerr and Zheng [16]. Detailedly, they proved that in a run of the cGA with hypothetical population size μ a frequency of a neutral bit will reach the boundaries of the frequency range in expected number of at most $4\mu^2$ generations, which is asymptotically tight. By Markov's inequality the probability that a boundary is reached in $8\mu^2$ generations is at least $1/2$.

This suggests the restart scheme described in Algorithm 3. We start with a small population size of $\mu = 2$. We then repeat running the cGA with population size μ for $8\mu^2$ generations and doubling the population size. As before, we do not specify a termination criterion since for our analysis we just count the number of fitness evaluations until a desired solution is found.

We consider two variations of this process. As discussed in the introduction already, we also regard a second criterion for stopping a run of a cGA, namely when a smaller *generation budget* of $B = 0.5\mu^2/\ln(n)$ iterations is used. Second, since the runtimes of the runs with either termination criterion are $\Theta(\mu^2)$, doubling μ after each run implies that the costs of the iterations increase by a factor of 4. To have the possibly more desired property that the costs only double, we also use the update factor $\sqrt{2}$ instead of 2.

Algorithm 3 The smart-restart cGA to maximize a function $f : \{0,1\}^n \to \mathbb{R}$ with update factor U and generation budget B. For the update factor, we propose to use the value $U = 2$ or $U = \sqrt{2}$. For the generation budget, we propose to use $B = 8\mu^2$ or $B = 0.5\mu^2/\ln(n)$.

for Round $\ell = 1, 2, \dots$ **do**
  Run the cGA (Algorithm 1) with population size $\mu = 2U^{\ell-1}$ for B iterations.
end for

4 THEORETICAL ANALYSIS

We follow the approach of [6] of building a theoretical analysis based on a runtime behavior often observed. Our Assumption (L) is the same as the one in [6] except that we ask for a slightly higher success probability of $7/8$ instead of $3/4$. Since most existing

runtime analyses give bounds with success probability $1 - o(1)$, this change is not too important.

Assumption (L): Consider using the cGA with population size μ to maximize a given function f. Assume that there are unknown $\tilde{\mu}$ and T such that the cGA for all population sizes $\mu \geq \tilde{\mu}$ optimizes this function f in μT fitness evaluations with probability at least $\frac{7}{8}$.

Under this assumption, we obtain the following theoretical result. It is easy to see that the asymptotic order of magnitude of the runtime is independent of U (as long as it is a constant), so we exemplarily make the constants explicit for $U = 2$. One may refer to [15, Appendix] for the details of the proof.

Theorem 4.1. *Consider using the smart-restart cGA with update factor $U = 2$ optimizing a function f satisfying Assumption (L). With a generation budget of $B = 8\mu^2$, the expected runtime is at most $\max\{\frac{112}{3}\tilde{\mu}^2, \frac{7}{12}T^2\}$ fitness evaluations. With $B = 0.5\mu^2/\ln n$, the expected runtime is at most $\max\{\frac{7}{3}\tilde{\mu}^2/\ln n, \frac{28}{3}T^2 \ln n\}$ fitness evaluations.*

We recall that the complexity of the parallel-run cGA under the original assumption [6] and also our Assumption (L) is $O(\tilde{\mu}T \log(\tilde{\mu}T))$. Hence, the asymptotic relationships among T, $\tilde{\mu}$, and n as well as the constants in these asymptotic notations matter in the actual performance comparison between these algorithms.

To apply our result, we recall that the cGA with all population sizes $\mu \geq K\sqrt{n}\ln(n)$ for a suitable constant K optimizes the OneMax function and the Jump function with jump size $k < \frac{1}{20}\ln n$ in time $O(\mu\sqrt{n})$ with probability $1 - o(1)$. Hence we have Assumption (L) satisfied with $\tilde{\mu} = K\sqrt{n}\ln(n)$ and $T = \sqrt{n}$. Consequently, by our result above, our smart-restart cGA with $B = 0.5\mu^2/\log n$ finds the optimum of Jump and OneMax in time $O(n \log n)$, which is also the runtime the classic cGA has with optimal parameter choice. With $B = 8\mu^2$, we obtain a slightly inferior runtime of $O(n \log^2 n)$, which is also the runtime of the parallel-run cGA of [6].

5 EXPERIMENTAL RESULTS

In this section, we experimentally analyze the smart-restart cGA proposed in this work. Since such data is not available from previous works, we start with an investigation how the runtime of the classic cGA depends on the population size μ. This will in particular support the basic assumption underlying the smart-restart cGA (and the parallel-run cGA from [6]) that the runtime is excessively large when μ is below some threshold, and moderate and linearly increasing with μ when μ is larger than this threshold.

Since the choice of the right population size is indeed critical for a good performance of the cGA, we then analyze the performance of the two existing approaches to automatically deal with the problem of choosing μ. Our focus is on understanding how one can relieve the user of an EDA from the difficult task of setting this parameter, not on finding the most efficient algorithm for the benchmark problems we regard. For this reason, we do not include other algorithms in this investigation.

5.1 Test Problems

Based on the above goals, we selected the four benchmark functions OneMax, LeadingOnes, Jump, and DeceptiveLeadingBlocks as optimization problems. For most of them also some mathematical runtime analyses exist, which help to understand and interpret the experimental results.

All four problems are defined on binary representations (bit strings) and we use n to denote their length. The **OneMax** problem is one of the easiest benchmark problems. The OneMax fitness of a bit string is simply the number of ones in the bit string. Having the perfect fitness-distance correlation, most EAs find it easy to optimize OneMax, a common runtime is $\Theta(n \log n)$. Also, mathematical runtime analyses are aided by its simple structure (see, e.g., [1, 13, 24, 30, 36]), though apparently for EDAs the runtime of OneMax is highly non-trivial. The known results for EDAs are the following. The first mathematical runtime analysis for EDAs by Droste [17] together with a recent refinement [35] shows that the cGA can efficiently optimize OneMax in time $\Theta(\mu\sqrt{n})$ when $\mu \geq K\sqrt{n}\ln(n)$ for some sufficiently large constant K. As the proofs of this result show (and the same could be concluded from the general result [16]), in this parameter regime there is little genetic drift. Throughout the runtime, with high probability, all bit frequencies stay above $\frac{1}{4}$. For hypothetical population sizes below the $\sqrt{n}\log n$ threshold, the situation is less understood. However, the lower bound of $\Omega(\mu^{1/3}n)$ valid for all $\mu = O\left(\frac{\sqrt{n}}{\ln(n)\ln\ln(n)}\right)$ proven in [28] together with its proof shows that in this regime the cGA suffers from genetic drift, leading to (mildly) higher runtimes.

The **LeadingOnes** benchmark is still an easy unimodular problem, however, typically harder than OneMax. The LeadingOnes value of a bit string is the number of ones in it, counted from left to right, until the first zero. How simple EAs optimize LeadingOnes is extremely well understood [2, 4, 18, 24, 34, 36], many EAs optimize this benchmark in time $\Theta(n^2)$. Surprisingly, no theoretical results are known on how the cGA optimizes LeadingOnes. However, the runtime of another EDA, the UMDA, with population sizes $\mu = \Theta(\lambda)$ with suitable implicit constants and $\lambda = \Omega(\log n)$ was shown to be $O(n\lambda\log(\lambda) + n^2)$ [3] and, recently, $\Theta(n\lambda)$ for $\lambda = \Omega(n \log n)$ [9]. Without going into details on this EDA not discussed so far in this work, we remark that [16] for this situation shows that genetic drift occurs when λ is below a threshold of $\Theta(n)$. Consequently, these results show a roughly linear influence of λ on the runtime when λ is (roughly) at least linear in n, but below this value, there is apparently no big penalty for running the EDA in the genetic drift regime. For the cGA, we will observe a different behavior, which also indicates that translating general behaviors from one EDA to another, even within the class of univariate EDAs, has to be done with caution.

The **Jump** benchmark is a class of multimodal fitness landscapes of scalable difficulty. For a difficulty parameter k, the fitness landscape is isomorphic to the one of OneMax except that there is a valley of low fitness of width k around the optimum. More precisely, all search points in distance 1 to $k - 1$ from the optimum have a fitness lower than all other search points. Recent results [5, 6, 23] show that when μ is large enough (so that the genetic drift is low, that is, all bit frequencies stay above $\frac{1}{4}$), then the cGA can optimize Jump functions quite efficiently and significantly more efficient than

many classic EAs. We omit some details and only mention that for k not too small, a runtime exponential in k results from a population size μ that is also exponential in k. This is much better than the $\Omega(n^k)$ runtime of many classic EAs [7, 14, 18]. It was not known whether the runtime of the cGA becomes worse in the regime with genetic drift, but our experimental results now show this.

The **DeceptiveLeadingBlocks** benchmark was introduced in [27]. It can be seen as a deceptive version of the LeadingOnes benchmark. In DeceptiveLeadingBlocks, the bits are partitioned into blocks of length two in a left-to-right fashion. The fitness is computed as follows. Counting from left to right, each block that consists of two ones contributes two to the fitness, until the first block is reached that does not consist of two ones. This block contributes one to the fitness if it consists of two zeros, otherwise it contributes zero. All further blocks do not contribute to the fitness. The main result in [27] is that when $\mu = \Theta(\lambda)$ and $\lambda = o(n)$, the runtime of the UMDA on DeceptiveLeadingBlocks is exponential in λ. With λ as small as $o(n)$ and a runtime that is at least quadratic, this result holds in a regime with strong genetic drift according to [16]. When $\lambda = \Omega(n \log n)$, a runtime of approximately $\frac{1}{2}\lambda n$ was shown in [12]. Hence for this function and the UMDA as optimizer, the choice of the population size is again very important. This was the reason for including this function into our set of test problems and the results indicate that indeed the cGA shows a behavior similar to what the mathematical results showed for the UMDA.

5.2 Experimental Settings

We ran the original cGA (with varying population sizes), the parallel-run cGA, and our smart-restart cGA (with two generation budgets and two update factors) on each of the above-described four problems. For each experiment we conducted 20 independent trials expect that for reasons of extremely large runtimes in the regime with genetic drift only 10 independent trials were conducted for the original cGA on the Jump and DeceptiveLeadingBlocks functions. The detailed settings for our experiments were as follows.

- Benchmark functions: OneMax (problem size $n = 500$), LeadingOnes ($n = 50$), Jump ($n = 50$ and the jump size $k = 10$), and DeceptiveLeadingBlocks ($n = 30$).
- Maximum number of generations of the original cGA: n^5 for OneMax and LeadingOnes, $n^{k/2}$ for Jump, and $10n^5$ for DeceptiveLeadingBlocks.
- Population size of the original cGA: $\mu = 2^{[2..10]}$ for OneMax and LeadingOnes, $\mu = 2^{[2..18]}$ for Jump, and $\mu = 2^{[1..14]}$ for DeceptiveLeadingBlocks. Since for all $\mu = 2^{[2..8]}$, none of 10 trials of the original cGA in the Jump experiments reached the optimum within $n^{k/2}$ generations, we do not report these values below.
- Generation budget B for the smart-restart cGA: $8\mu^2$ and $0.5\mu^2/\ln n$. As explained in the introduction, $B = 8\mu^2$ and $\Theta(\mu^2/\ln n)$ are two proper choices. We chose the constant 0.5 based on the experimental results on Jump and DeceptiveLeadingBlocks in Figure 1. We ignored the results for OneMax and LeadingOnes since for these functions larger ranges of population sizes all gave a good performance.
- Update factor U for the smart-restart cGA: 2 and $\sqrt{2}$. Doubling the parameter value after each unsuccessful run ($U = 2$) is a natural choice for a sequential parameter search. Since with the above generation budget the runtime of a cGA run depends quadratically on μ, we were wondering if a doubling scheme is not too aggressive (as it would be a factor-4 increase scheme in terms of runtime). Hence, we also experimented with $U = \sqrt{2}$.

5.3 Experimental Results and Analysis I: The cGA with Different Population Sizes

Figure 1 shows the runtime (measured by the number of fitness evaluations) of the classic cGA with different population sizes when optimizing our four test functions. To allow an estimate in which iteration the smart-restart cGA would have found the optimum, we also plotted the two generation budgets $8\mu^2$ and $0.5\mu^2/\ln n$ ($2 * 8\mu^2$ and $2 * 0.5\mu^2/\ln n$ for the fitness evaluation numbers). We make the precise runtimes explicit in Table 1 for the value of μ which gave the best average runtime.

As a side result, this data confirms that the cGA has a good performance on Jump functions, not only in asymptotic terms as proven in [6, 23], but also in terms of actual runtimes for concrete problem sizes. On a Jump function with parameters $n = 50$ and $k = 10$, a classic mutation-based algorithm would run into the local optimum and from there would need to generate the global optimum via one mutation. For standard bit mutation with mutation rate $\frac{1}{n}$, this last step would take an expected time of $n^k(\frac{n}{n-1})^{n-k}$, which for our values of n and k is approximately $2.2 \cdot 10^{17}$. With the asymptotically optimal mutation rate of $\frac{k}{n}$ determined in [14], this time would still be approximately $7.3 \cdot 10^{10}$. In contrast, the median optimization time of the cGA with $\mu \in 2^{[15..18]}$ is always below $4 \cdot 10^6$.

The results displayed in Figure 1 generally show that indeed the runtime of the cGA is large both for small values of μ and for large values. The efficient middle regime is relatively wide for OneMax. On the small end only a population size of $\mu = 4$ led to larger (but then truly huge) runtimes. For $\mu \geq 2^7$, the runtime increases roughly linearly with μ.

For all other functions, there is one clear optimal population size. Above this value, the runtime increases in a regular manner and the runtimes are strongly concentrated.

Below this value, the runtimes quickly raise (much steeper than on the large side) and are less concentrated. We note that for runs that were stopped because the maximum number of generations was reached, we simply and bluntly counted this maximum number of generations as runtime. Clearly, there are better ways to handle such incomplete runs, but since a fair computation for these inefficient parameter ranges is not too important, we did not start a more elaborate evaluation.

Let us regard the increase of the runtime for smaller population sizes in more detail. For OneMax, it appears only for $\mu = 4$, which clearly is a population size too small to give any significant information. For the remaining values before the start of the linear increase of the runtime, the runtime is always very small. Since it is clear that for small values like $\mu = 8$ there will be genetic drift, that is, frequencies that reach the lower boundary, this shows that the cGA can optimize OneMax also in the presence of genetic drift. For LeadingOnes, there is already a unique value for μ, namely

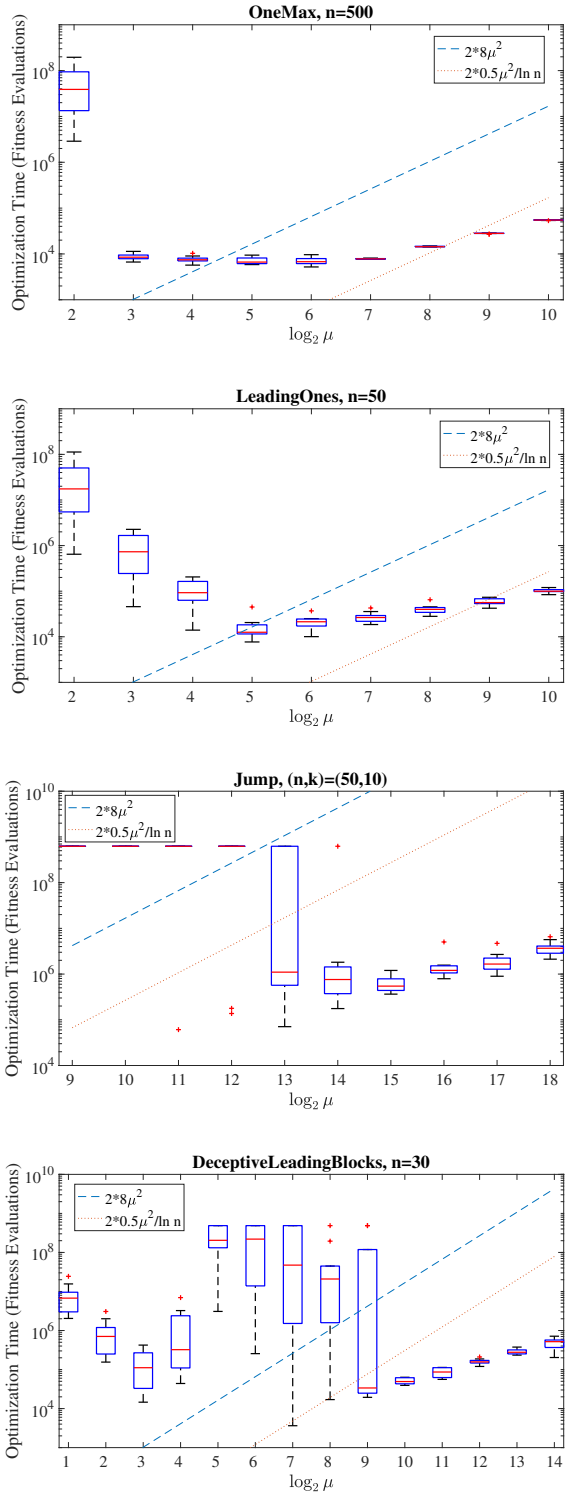

Figure 1: Runtimes of the classic cGA with different population sizes.

Table 1: Runtime details for the classic cGA. Best-μ is the value of μ leading to the smallest mean runtime. For this μ, the mean, median, minimum, and maximum runtimes are given.

	OneMax	LeadingOnes	Jump	DLB
Best-μ	64	32	32,768	1,024
Mean	6,970	17,061	620,658	51,219
Median	6,736	14,706	545,285	49,279
Min	5,202	7,698	364,110	39,346
Max	9,632	44,958	1,201,142	63,410

Note: DLB for DeceptiveLeadingBlocks.

$\mu = 2^5$ that gives a clearly visibly unique minimum median runtime. Reducing μ further leads to a clear increase of the runtime, roughly by a factor of 6 for each halving of μ. For Jump and DeceptiveLeadingBlocks, reducing the population size below the efficient values leads to a catastrophic increase of the runtime by factors of more than 100 just by having the last reasonable runtime (a further increase from reducing the runtime could not be observed because these experiments took so long that they had to be stopped). We also observe a drastic increase of the variance of the runtime when leaving the efficient regime. This indicates that some runs were very lucky to not suffer from genetic drift and then finished early (at a runtime as if the linear regimes was continued), whereas others suffered from genetic drift and thus took very long. We note that when some frequencies reach the lower boundary (genetic drift), then it takes longer to move them back into the middle regime. During this longer runtime, of course, the remaining frequencies are still are prone to genetic drift (recall that the quantitative analysis [16] shows that genetic drift is more likely the longer the run takes). These mathematical considerations and the experimental results indicate that for objective functions which could suffer from genetic drift, there are two very distinct extremal regimes: either no frequency reaches the wrong boundary and the optimization is efficient, or many frequencies reach the wrong boundary and the optimization is highly inefficient.

We note that for DeceptiveLeadingBlocks, the runtime decreases again when further decreasing the population size. We have no explanation for this. Apart from this single observation, our results indicate that all four test problems show a runtime behavior as described in Assumption (L). To make this more visible, we computed the ratio of the median runtimes for the highest and second-highest population size in the data displayed in Figure 1. Since the highest population size is twice the second-highest one, a ratio of two would indicate a perfect linear behavior (note that we regard the two highest population sizes to be as much as possible in the linear regime). The ratios we observed are 1.95, 1.82, 1.93, and 1.87 for OneMax, LeadingOnes, Jump, and DeceptiveLeadingBlocks respectively, which supports our theory that Assumption (L) is a runtime profile often observed for the cGA.

Table 2: Runtime comparison between the parallel-run cGA and the four variants of the smart-restart cGA

		OneMax	LeadingOnes	Jump	DeceptiveLeadingBlocks
Parallel-run cGA	Mean(Std-dev)	62,705(2,241)	183,992(81,499)	6,964,249(8,647,270)	365,757(317,466)
	Median	63,150	147,456	3,888,671	169,445
	Min	59,380	64,466	384,826	65,416
	Max	68,440	319,074	37,466,078	1,159,150
Smart-restart cGA	Mean(Std-dev)	**12,559**(1,264)+	**28,359**(12,983)+	2,506,117,742(5,170,078,482)-	2,798,567(2,363,710)-
with $B = 8\mu^2$	Median	12,376	31,661	268,907,524	1,434,478
and $U = 2$	Min	10,568	11,668	16,785,876	7,408
	Max	15,528	47,564	17,180,765,184	5,656,524
Smart-restart cGA	Mean(Std-dev)	18,297(4,683)+	42,202(18,221)+	368,933,280(530,770,847)-	1,766,550(1,927,353)-
with $B = 8\mu^2$	Median	15,690	46,248	134,412,883	1,070,238
and $U = \sqrt{2}$	Min	13,820	14,028	528,240	1,832
	Max	27,168	80,620	2,148,950,634	8,453,288
Smart-restart cGA	Mean(Std-dev)	41,966(431)+	90,777(36,745)+	**6,870,179**(8,709,131)=	**83,694**(61,514)+
with $B = 0.5\mu^2/\ln n$	Median	42,082	77,842	1,997,966	54,070
and $U = 2$	Min	40,978	52,544	14,026	14,730
	Max	42,658	176,806	23,530,502	167,804
Smart-restart cGA	Mean(Std-dev)	41,247(298)+	132,115(36,457)+	7,872,212(15,051,482)=	120,973(77,923)+
with $B = 0.5\mu^2/\ln n$	Median	41,193	123,155	4,454,652	103,718
and $U = \sqrt{2}$	Min	40,768	66,848	160,766	30,384
	Max	41,648	224,416	69,217,910	360,656

Note: A Wilcoxon rank sum test with significance level 0.05 is conducted between parallel cGA and fours variants of the smart-restart cGA, and "=", "-", and "+" represent that the variant has similar, worse, and better performance than the parallel-run cGA. A Wilcoxon rank sum test (not displayed in the table) between the variant of the smart-restart cGA with smallest mean runtime and the other variants of the smart-restart cGA in all cases showed the other variants to be significantly inferior apart from the case $B = 0.5\mu^2/\ln n$ and $U = \sqrt{2}$ for Jump and DeceptiveLeadingBlocks.

5.4 Experimental Results and Analysis II: Comparison Between Parallel-run cGA and Smart-restart cGA

5.4.1 Runtimes. Table 2 shows the runtimes of the parallel-run cGA and smart-restart cGA (with two generation budgets and two update factors). We see that for the easy functions OneMax and LeadingOnes, the smart-restart cGA in any setting has a smaller runtime than the parallel-run cGA. This can be explained from the data in Figure 1: Since the runtimes are similar for several population sizes, the parallel-run cGA with its strategy to assign a similar budget to different population sizes just wastes computational power, which the smart-restart cGA saves by aborting some processes early. For both functions, the larger generation budget is superior. This fits again to the plots in Figure 1 and to our interpretation that genetic drift here is not so detrimental. Consequently, it is better to let the current run continue than to abort it and start a new one.

More interesting are the results for Jump and DeceptiveLeadingBlocks. We recall that here a wrong choice of the population size can be catastrophic, so these are the two functions where not having to choose the population size is a big advantage for the user. What is clearly visible from the data is that here the smaller generation budget is preferable for the smart-restart cGA. This fits to our previously gained intuition that for these two functions, genetic drift is detrimental. Hence there is no gain from continuing a run that is suffering from genetic drift (we note that there is no way to detect genetic drift on the fly – a frequency can be at a (wrong) boundary value due to genetic drift or at a (correct) boundary value because of a sufficiently strong fitness signal). Concerning the update factor, there is no clear picture.

What is clear as a general rule is that both algorithms, the parallel-run cGA and the smart-restart cGA with the small generation budget, clearly do a good job in running the cGA with a reasonable population size – recall that for both of the difficult functions, a wrong choice of the population size can easily imply that the cGA does not find the optimum in 10^8 iterations.

5.4.2 Population Sizes. Table 3 collects the population size μ which first finds the optimum (*stopping population size*) for the parallel-run cGA and smart-restart cGA (with two generation budgets and two update factors). For the parallel-run cGA, we also give the round ℓ in which the optimum is found. Before we look at the data in Table 3, we go back to Figure 1 to figure out what our experiments on the cGA with different population sizes could indicate on the stopping population size. Focusing on the intersection points of the two generation budget lines with the boxes and whiskers and the positional relations between two lines and the outliers, we obtain the following prior guesses on the ranges of the stopping population sizes.

- OneMax: $[16, 32]$ for $B = 8\mu^2$ and $[256, 512]$ for $B = 0.5\mu^2/\ln n$.
- LeadingOnes: $[16, 64]$ for $B = 8\mu^2$ and $[256, 1024]$ for $B = 0.5\mu^2/\ln n$.
- Jump: $[2048, 16384]$ for $B = 8\mu^2$ and $[1024, 32768]$ for $B = 0.5\mu^2/\ln n$.

Table 3: Population size μ that found the optimum ("stopping population size") in runs of the parallel-run cGA and smart-restart cGA

Parallel-run cGA		OneMax	LO	Jump	DLB
Round ℓ	Mean	12	13	17	14
	Median	12	13	17	13
	Min	12	12	14	12
	Max	12	14	20	16
μ	Mean	14	41	12,288	236
	Median	16	32	6,144	128
	Min	8	16	1,024	4
	Max	32	128	65,536	1024
Smart-restart cGA		**OneMax**	**LO**	**Jump**	**DLB**
$B = 8\mu^2$ and $U = 2$	Mean	32	50	8,550	477
	Median	32	64	4,096	512
	Min	32	32	1,024	64
	Max	32	64	32,768	1,024
$B = 8\mu^2$ and $U = \sqrt{2}$	Mean	26	39	3,834	278
	Median	23	45	2,896	256
	Min	23	23	181	8
	Max	32	64	11,585	724
$B = 0.5\mu^2/\ln n$ and $U = 2$	Mean	512	589	7,027	640
	Median	512	512	4,096	512
	Min	512	512	256	256
	Max	512	1,024	16,384	1,024
$B = 0.5\mu^2/\ln n$ and $U = \sqrt{2}$	Mean	362	536	4,330	529
	Median	362	512	4,096	512
	Min	362	362	724	256
	Max	362	724	16,384	1,024

Note: LO for LeadingOnes and DLB for DeceptiveLeadingBlocks.

- DeceptiveLeadingBlocks: $[64, 1024]$ for both $B = 8\mu^2$ and $0.5\mu^2/\ln n$.

Now together with Table 3, we find that the experimental stopping population sizes almost perfectly match our guesses. The most striking exceptions are the minimum values of 181 and 8 for the Jump and DeceptiveLeadingBlocks function in the case $(B, U) = (8\mu^2, \sqrt{2})$. Here apparently some run with very small population size was very lucky to not suffer from genetic drift and thus reach the optimum quickly.

We note that the maximum value for the smart-restart cGA with $B = 8\mu^2$ and $U = 2$ on the Jump function is not an exception since we took $n^{k/2}$ as the maximum generation budget for a cGA run. The outlier of $\log_2 \mu = 14$ has the runtime of $2 \cdot 50^{10/2}$ in Figure 1, which means that the real runtime is larger than the current one, thus the intersection should happen at a larger population size.

Together with the "optimal" population size of the cGA from Table 1, we can see that except LeadingOnes function the stopping population size of the smart-restart cGA is smaller than the optimal population size of the original cGA. The reason could be that the optimal population size in Table 1 is based on the mean runtime, which does not rule out that there is a good probability that the cGA with a smaller population can reach the optimum within a reasonable time with fair probability.

In Table 3, we also collect the round number in which the parallel-run cGA has found the optimum. Note that in round ℓ, the maximum population size of all processes changes from $2^{\ell-2}$ to $2^{\ell-1}$. However, the successful run almost always used a much smaller population.

6 CONCLUSION

Choosing the right population size for EDAs is one of the key difficulties for their practical usage. In order to remove the population size as a parameter and thus make EDAs easier to use, this paper proposed a parameter-less framework for EDAs, using the cGA as example. This framework is a simple restart strategy with exponentially growing population size, but different from previous works it sets a prior generation budget for each population size based on a recent quantitative analysis estimating when genetic drift is likely to occur and render the EDA inefficient.

Under a reasonable assumption on how the runtime depends on the population size, we theoretically analyzed our scheme and observed that it can lead to asymptotically optimal runtimes for the cGA.

Via extensive experiments on OneMax, LeadingOnes, Jump, and DeceptiveLeadingBlocks, we showed the efficiency of the parameter-less cGA, also when compared with the parallel-run cGA. The results for the original cGA with different population sizes experimentally show that the population size is crucial for the performance of the cGA.

We positively believe that our parameter-less framework for the cGA can be also applied to other univariate EDAs, again building on the quantitative analysis of genetic drift in [16]. The problem of how to cope with genetic drift, naturally, is equally interesting for multivariate EDAs. For these, however, our theoretical understanding is limited to very few results such as [10, 27, 37]. In particular, a quantitative understanding of genetic drift comparable to [16] is completely missing. Another interesting question is if dynamic choices of the population size in EDAs can be fruitful. In classic EAs, dynamic parameter choices have recently been used very successfully to overcome the difficulty of finding a suitable static parameter value, see, e.g., the survey [8]. How to use such ideas for EDAs is currently not at all clear.

ACKNOWLEDGMENTS

This work was supported by a public grant as part of the Investissement d'avenir project, reference ANR-11-LABX-0056-LMH, LabEx LMH, in a joint call with Gaspard Monge Program for optimization, operations research and their interactions with data sciences.

This work was also supported by Guangdong Provincial Key Laboratory (Grant No. 2020B121201001), the Program for Guangdong Introducing Innovative and Enterpreneurial Teams (Grant No. 2017ZT07X386); Guangdong Basic and Applied Basic Research Foundation (Grant No. 2019A1515110177); Shenzhen Peacock Plan (Grant No. KQTD2016112514355531); and the Program for University Key Laboratory of Guangdong Province (Grant No. 2017KSYS008).

The authors are thankful to an anonymous reviewer for proposing the name *smart-restart cGA*.

REFERENCES

[1] Denis Antipov, Benjamin Doerr, Jiefeng Fang, and Tangi Hetet. 2018. Runtime analysis for the $(\mu + \lambda)$ EA optimizing OneMax. In *Genetic and Evolutionary Computation Conference, GECCO 2018*. ACM, 1459–1466.

[2] Süntje Böttcher, Benjamin Doerr, and Frank Neumann. 2010. Optimal fixed and adaptive mutation rates for the LeadingOnes problem. In *Parallel Problem Solving from Nature, PPSN 2010*. Springer, 1–10.

[3] Duc-Cuong Dang and Per Kristian Lehre. 2015. Simplified runtime analysis of estimation of distribution algorithms. In *Genetic and Evolutionary Computation Conference, GECCO 2015*. ACM, 513–518.

[4] Benjamin Doerr. 2019. Analyzing randomized search heuristics via stochastic domination. *Theoretical Computer Science* 773 (2019), 115–137.

[5] Benjamin Doerr. 2019. An exponential lower bound for the runtime of the compact genetic algorithm on jump functions. In *Foundations of Genetic Algorithms, FOGA 2019*. ACM, 25–33.

[6] Benjamin Doerr. 2019. A tight runtime analysis for the cGA on jump functions: EDAs can cross fitness valleys at no extra cost. In *Genetic and Evolutionary Computation Conference, GECCO 2019*. ACM, 1488–1496.

[7] Benjamin Doerr. 2020. Does comma selection help to cope with local optima?. In *Genetic and Evolutionary Computation Conference, GECCO 2020*. ACM. To appear.

[8] Benjamin Doerr and Carola Doerr. 2020. Theory of parameter control for discrete black-box optimization: provable performance gains through dynamic parameter choices. In *Theory of Evolutionary Computation: Recent Developments in Discrete Optimization*, Benjamin Doerr and Frank Neumann (Eds.). Springer, 271–321. Also available at https://arxiv.org/abs/1804.05650.

[9] Benjamin Doerr and Martin Krejca. 2020. A simplified run time analysis of the univariate marginal distribution algorithm on LeadingOnes. *CoRR* abs/2004.04978 (2020). arXiv:2004.04978

[10] Benjamin Doerr and Martin S. Krejca. 2020. Bivariate estimation-of-distribution algorithms can find an exponential number of optima. In *Genetic and Evolutionary Computation Conference, GECCO 2020*. ACM. To appear.

[11] Benjamin Doerr and Martin S. Krejca. 2020. Significance-based estimation-of-distribution algorithms. *IEEE Transactions on Evolutionary Computation* (2020), To appear. https://doi.org/10.1109/TEVC.2019.2956633

[12] Benjamin Doerr and Martin S. Krejca. 2020. The univariate marginal distribution algorithm copes well with deception and epistasis. In *Evolutionary Computation in Combinatorial Optimization, EvoCOP 2020*. Springer, 51–66.

[13] Benjamin Doerr and Marvin Künnemann. 2015. Optimizing linear functions with the $(1 + \lambda)$ evolutionary algorithm—Different asymptotic runtimes for different instances. *Theoretical Computer Science* 561 (2015), 3–23.

[14] Benjamin Doerr, Huu Phuoc Le, Régis Makhmara, and Ta Duy Nguyen. 2017. Fast genetic algorithms. In *Genetic and Evolutionary Computation Conference, GECCO 2017*. ACM, 777–784.

[15] Benjamin Doerr and Weijie Zheng. 2020. From understanding genetic drift to a smart-restart parameter-less compact genetic algorithm. *CoRR* abs/2004.07141 (2020). arXiv:2004.07141

[16] Benjamin Doerr and Weijie Zheng. 2020. Sharp bounds for genetic drift in estimation of distribution algorithms. *IEEE Transactions on Evolutionary Computation* (2020), Accepted. https://doi.org/10.1109/TEVC.2020.2987361

[17] Stefan Droste. 2006. A rigorous analysis of the compact genetic algorithm for linear functions. *Natural Computing* 5 (2006), 257–283.

[18] Stefan Droste, Thomas Jansen, and Ingo Wegener. 2002. On the analysis of the (1+1) evolutionary algorithm. *Theoretical Computer Science* 276 (2002), 51–81.

[19] Tobias Friedrich, Timo Kötzing, and Martin S. Krejca. 2016. EDAs cannot be balanced and stable. In *Genetic and Evolutionary Computation Conference, GECCO 2016*. ACM, 1139–1146.

[20] Brian W. Goldman and William F. Punch. 2014. Parameter-less population pyramid. In *Genetic and Evolutionary Computation Conference, GECCO 2014*. ACM, 785–792.

[21] Georges R. Harik and Fernando G. Lobo. 1999. A parameter-less genetic algorithm. In *Genetic and Evolutionary Computation Conference, GECCO 1999*. 258–265.

[22] Georges R. Harik, Fernando G. Lobo, and David E. Goldberg. 1999. The compact genetic algorithm. *IEEE Transactions on Evolutionary Computation* 3 (1999), 287–297.

[23] Václav Hasenöhrl and Andrew M. Sutton. 2018. On the runtime dynamics of the compact genetic algorithm on jump functions. In *Genetic and Evolutionary Computation Conference, GECCO 2018*. ACM, 967–974.

[24] Thomas Jansen, Kenneth A. De Jong, and Ingo Wegener. 2005. On the choice of the offspring population size in evolutionary algorithms. *Evolutionary Computation* 13 (2005), 413–440.

[25] Martin Krejca and Carsten Witt. 2020. Theory of estimation-of-distribution algorithms. In *Theory of Evolutionary Computation: Recent Developments in Discrete Optimization*, Benjamin Doerr and Frank Neumann (Eds.). Springer, 405–442. Also available at https://arxiv.org/abs/1806.05392.

[26] Pedro Larrañaga and José Antonio Lozano (Eds.). 2002. *Estimation of Distribution Algorithms*. Springer.

[27] Per Kristian Lehre and Phan Trung Hai Nguyen. 2019. On the limitations of the univariate marginal distribution algorithm to deception and where bivariate EDAs might help. In *Foundations of Genetic Algorithms, FOGA 2019*. 154–168.

[28] Johannes Lengler, Dirk Sudholt, and Carsten Witt. 2018. Medium step sizes are harmful for the compact genetic algorithm. In *Genetic and Evolutionary Computation Conference, GECCO 2018*. ACM, 1499–1506.

[29] Cláudio F. Lima and Fernando G. Lobo. 2004. Parameter-less optimization with the extended compact genetic algorithm and iterated local search. In *Genetic and Evolutionary Computation Conference, GECCO 2004*. Springer, 1328–1339.

[30] Heinz Mühlenbein. 1992. How genetic algorithms really work: mutation and hillclimbing. In *Parallel Problem Solving from Nature, PPSN 1992*. Elsevier, 15–26.

[31] Heinz Mühlenbein and Gerhard Paass. 1996. From recombination of genes to the estimation of distributions I. Binary parameters. In *Parallel Problem Solving from Nature, PPSN 1996*. Springer, 178–187.

[32] Martin Pelikan, Mark Hauschild, and Fernando G. Lobo. 2015. Estimation of distribution algorithms. In *Springer Handbook of Computational Intelligence*, Janusz Kacprzyk and Witold Pedrycz (Eds.). Springer, 899–928.

[33] Martin Pelikan and Tz-Kai Lin. 2004. Parameter-less hierarchical BOA. In *Genetic and Evolutionary Computation Conference, GECCO 2004*. Springer, 24–35.

[34] Dirk Sudholt. 2013. A new method for lower bounds on the running time of evolutionary algorithms. *IEEE Transactions on Evolutionary Computation* 17 (2013), 418–435.

[35] Dirk Sudholt and Carsten Witt. 2019. On the choice of the update strength in estimation-of-distribution algorithms and ant colony optimization. *Algorithmica* 81 (2019), 1450–1489.

[36] Carsten Witt. 2006. Runtime analysis of the $(\mu + 1)$ EA on simple pseudo-Boolean functions. *Evolutionary Computation* 14 (2006), 65–86.

[37] Qingfu Zhang and Heinz Mühlenbein. 2004. On the convergence of a class of estimation of distribution algorithms. *IEEE Transactions on Evolutionary Computation* 8 (2004), 127–136.