Theoretical Computer Science 567 (2015) 87–104

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

From black-box complexity to designing new genetic algorithms [☆]

Benjamin Doerr [a], Carola Doerr [b,c,*], Franziska Ebel [d]

[a] École Polytechnique, Palaiseau, France
[b] Sorbonne Universités, UPMC Univ Paris 06, UMR 7606, LIP6, Paris, France
[c] CNRS, UMR 7606, LIP6, Paris, France
[d] Saarland University, Saarbrücken, Germany

ARTICLE INFO

Article history:
Received 28 June 2014
Received in revised form 16 November 2014
Accepted 27 November 2014
Available online 4 December 2014
Communicated by L. Kari

Keywords:
Heuristic search
Theory of randomized search heuristics
Genetic algorithms
Runtime analysis
Black-box complexity

ABSTRACT

Black-box complexity theory recently produced several surprisingly fast black-box optimization algorithms. In this work, we exhibit one possible reason: These black-box algorithms often profit from solutions inferior to the previous-best. In contrast, evolutionary approaches guided by the "survival of the fittest" paradigm often ignore such solutions. We use this insight to design a new crossover-based genetic algorithm. It uses mutation with a higher-than-usual mutation probability to increase the exploration speed and crossover with the parent to repair losses incurred by the more aggressive mutation. A rigorous runtime analysis proves that our algorithm for many parameter settings is asymptotically faster on the OneMax test function class than all what is known for classic evolutionary algorithms. A fitness-dependent choice of the offspring population size provably reduces the expected runtime further to linear in the dimension. Our experimental analysis on several test function classes shows advantages already for small problem sizes and broad parameter ranges. Also, a simple self-adaptive choice of these parameters gives surprisingly good results.

1. Introduction

The area of black-box complexity theory recently produced a number of black-box optimization algorithms for classic test problems that have a much better runtime than all known evolutionary algorithms for these problems. In this work, we observe that these algorithms greatly profit from search points with fitness inferior to the current-best solution. We use this finding as inspiration to design a new class of genetic algorithms (GAs), which use an untypically high mutation rate to speed-up exploration and then employ crossover in a novel fashion to repair the defects caused by the more aggressive mutation. Both via rigorous runtime analysis and experiments we demonstrate that our algorithm for a broad range of parameter settings is faster than previous evolutionary algorithms. We also observe that a fitness-dependent choice of the offspring population size as well as a self-adjusting choice of this parameter work well and give further speed-ups.

[☆] An extended abstract containing parts of our results appeared in [12].
[*] Corresponding author.

http://dx.doi.org/10.1016/j.tcs.2014.11.028
0304-3975/© 2014 Elsevier B.V. All rights reserved.

1.1. Black-box complexity theory: unexpectedly fast black-box optimization algorithms

Black-box complexity, introduced in the seminal paper by Droste, Jansen, Tinnefeld, and Wegener [20] more than ten years ago and recently revived starting with the works by Anil and Wiegand [2] as well as Lehre and Witt [31],[1] is a theory-guided notion to describe how difficult a problem is when to be solved via algorithms that do not have access to an explicit description of the problem instance (such algorithms are typically referred to in the literature as *black-box optimization algorithms*; the class of black-box algorithms in particular includes most genetic algorithms as well as many other randomized search heuristics). In simple words, black-box complexity asks for how many evaluations of solution candidates are necessary to solve a problem. In such, it yields lower bounds for the performance of any genetic algorithm for this problem.

In particular the last few years produced a number of surprising black-box complexity results. They show that for many problems there are black-box algorithms that are significantly faster than the best known genetic algorithms. For example, Anil and Wiegand [2] studied the generalized OneMax problem which contains all functions that have a fitness landscape that is equivalent to the one of the classic OneMax function. The well-studied OneMax function assigns to each bit string x of length n the number of ones in x, i.e., OneMax $(x) := \sum_{i=1}^{n} x_i$. It is not difficult to see that for any bit string $z \in \{0,1\}^n$ the function $f_z : \{0,1\}^n \to \mathbb{R}; x \mapsto |\{i \in \{1,\dots,n\} \mid x_i = z_i\}|$ generates the same fitness landscape than OneMax. Note that OneMax$= f_{(1,\dots,1)}$. We therefore call the problem of optimizing an unknown function from the class $\{f_z \mid z \in \{0,1\}^n\}$ the *generalized* OneMax *problem*. Anil and Wiegand [2] presented a black-box algorithm that finds the optimum of any such generalized OneMax function in only $O(n/\log n)$ function evaluations. This bound is best-possible as has been shown in [19].[2]

In contrast, all genetic algorithms for which the runtime on OneMax is known need $\Omega(n \log n)$ function evaluations (see [18] for randomized local search (RLS) and the $(1+1)$ EA, [28] for the $(1+\lambda)$ EA, see [41,46] for the $(\mu+1)$ EA, [27,38] for the $(1,\lambda)$ EA, and [42] for the $(2+1)$ GA). In fact, it is also known that every purely mutation-based black-box algorithm needs $\Omega(n \log n)$ function evaluations to optimize OneMax, as has been shown by Lehre and Witt in their studies of so-called unary unbiased black-box algorithms [32].

Similarly, the generalized LeadingOnes problem in dimension n, consisting of all functions with fitness landscape equivalent to the one of the classic LeadingOnes test function, i.e., the function class consisting of all functions arising from concatenating an automorphism of the n-dimensional hypercube with the classic LeadingOnes test function LeadingOnes : $\{0,1\}^n \to \{0,1,\dots,n\}, x \mapsto \max\{i \in [0..n] \mid \forall j \le i : x_j = 1\}$, has a black-box complexity of $\Theta(n \log\log n)$, see [1]. Here, all known genetic algorithms need at least quadratic time, see, e.g., [18,28]. For the two combinatorial optimization problems *Minimum Spanning Tree* and *Single-Source Shortest Paths* the situation is no better, see [17] for known discrepancies between the black-box complexities of these two problems and the respective runtimes of classic evolutionary algorithms.

The reason for the surprising performances of black-box algorithms does not necessarily lie in an exploitation of resources not available to classic evolutionary approaches. For example, the $\Theta(n/\log n)$ complexity for all OneMax functions remains unchanged if we restrict ourselves to black-box algorithms storing only 2 solution candidates at all times [13] or black-box algorithms not having access to the precise function values, but only to a relative comparison of the fitness of the visited search points [14].

An explanation closer to the truth is that most evolutionary algorithms were developed to cope with a broad range of applications. In such, it is no surprise that they are outperformed on a particular problem by the best black-box algorithm for this particular problem.

In this work, we take the view that it still makes sense to have a look at the various black-box algorithms developed in the past few years, to try to distill general reasons for their superiority (apart from being more problem-specific), and to use these insights as an inspiration to develop novel genetic algorithms.

1.2. Profiting from inferior solutions

A closer inspection of the above-mentioned surprisingly fast black-box algorithms reveals that all of them greatly profit from search points that are inferior to the current-best solution.

For example, the known asymptotically optimal black-box algorithm for the generalized OneMax problem consists of sampling $\Theta(n/\log n)$ random bit strings (independently and uniformly at random) and then computing from these and their objective values the optimal solution. This algorithm clearly does not perform hill-climbing of any kind. In contrast, it benefits from sampling search points that (with high probability) all have a fitness close to $n/2$.[3]

Similarly, the known asymptotically optimal black-box algorithm for the generalized LeadingOnes problem profits from solutions with fitness inferior to the best-so-far solution (which at first is surprising given that the fitness landscape of

[1] See [19] and [32] for the journal versions of [20] and [31], respectively.

[2] As noted in [13] both results had appeared earlier in [23], in the context of information theory.

[3] In fact, as the analysis in [14] on the ranking-based black-box complexity of OneMax shows, it suffices to consider only those samples that have a function value between $n/2 - \kappa\sqrt{n}$ and $n/2 + \kappa\sqrt{n}$, for some small constant κ.

each LeadingOnes function is such that no information about higher-order bits can be derived from flipping lower-order bits). The reason for inferior solutions to be useful is the following. As the analysis in [1] shows, it can be easier to find a solution with larger fitness than to determine which bit causes the fitness increase. For this reason, it is more efficient to first increase the fitness to a certain extent and then learn from offspring with reduced fitness which bits caused the previous fitness increases.

In contrast to these two examples of known optimal black-box algorithms, many genetic algorithms do not profit significantly from inferior solutions. For the $(1+\lambda)$ EA, it is obvious that any solution worse than the current best individual is simply discarded. But also for other genetic algorithms one has the feeling that often inferior offspring when accepted into the population do not stay there for long, but are quickly removed in favor of better solutions, and that when inferior solutions are accepted, then mostly with the hope that after a while they can improve and lead to an optimal solution in a different area of attraction (this is in particular true for Simulated Annealing and the Metropolis algorithm). For this reason, GAs with low selection pressure are often not very efficient for unimodal functions. For example, optimizing the easy OneMax function takes an exponential time with the *simple genetic algorithm* (SGA) when the population size is less than $n^{1/4-\varepsilon}$ [36] or with RLS and the $(1+1)$ EA when using fitness-proportional selection [25].

As a consequence of inferior solutions being not very useful in evolutionary computation, an often recommended choice for the mutation probability is $1/n$, that is, the mutation operator creates a new solution candidate from a parent solution by flipping each bit independently with probability $1/n$. This choice necessarily leads to a slow exploration of the search space. For example, it is well known from the coupon collectors problem and variants thereof that it already takes an expected number of $\Theta(n \log n)$ mutation steps to ensure that each bit of an n-bit string was flipped at least once.

1.3. Designing new genetic algorithms

In this work, inspired by the above discussion, we experiment with a simple way of letting the algorithms exploit inferior solutions. This will allow us to use larger mutation probabilities, and thus a faster exploration of the search space.

We propose a genetic algorithm that works with a parent population of size one. From this parent we generate λ offspring independently by standard bit mutation. However, to speed up exploration, we use the increased mutation probability $p = k/n$ for some $k > 1$. This larger mutation probability will frequently lead to a situation in which even the best of these offspring has a worse fitness than the parent individual, simply because among the in average k bits that flip, some will give away good parts of the current best solution. On the positive side, however, the increased mutation probability will also speed up finding yet unfound elements of the optimal solution. To avoid losing the good parts already present in the parent individual, but at the same time keeping the new good parts found by the more explorative mutation operator, we use crossover between the parent and the best among the offspring (randomly breaking ties).

We use a uniform crossover that takes bits from the parent with probability $1-c$ and from the winning offspring with probability c for some not too large crossover probability c. The outcome of such a crossover step will be close enough to the parent to give us a good chance of keeping the positive aspects of the parent. To give newly found positive genes of the winning offspring a reasonable chance to survive, we create again λ offspring by this crossover. We call this algorithm the $(1+(\lambda,\lambda))$ GA. We will discuss variants of this algorithm using a fitness-dependent parameter choice and a self-adjusting parameter choice in Sections 3.2 and 4.

1.4. Our results

We analyze the $(1+(\lambda,\lambda))$ GA both by theoretical means and through experiments. For the theoretical analysis, as many preceding works, we restrict ourselves to the simple OneMax function. Note that our algorithm is *unbiased* in the sense of Lehre and Witt [32], so all results we prove for OneMax actually hold for all generalized OneMax functions having an equivalent fitness landscape (see page 2 for the definitions).

We show that the expected runtime (that is, the expected number of fitness evaluations until an optimal solution is found, also referred to as the *optimization time*) of our algorithm is $O((\frac{1}{k}+\frac{1}{\lambda})n\log n+(k+\lambda)n)$, when the crossover probability is taken as $c = 1/k$, which is what both the proofs and the intuition given in Section 2 suggest. Consequently, quite a broad selection of choices of k and λ leads to expected optimization times better than the classic $\Theta(n \log n)$. This runtime bound suggests to take $k = \Theta(\lambda)$, which yields an expected optimization time of $O(\frac{1}{\lambda}n\log n + \lambda n)$. For $\lambda = \sqrt{\log n}$, we obtain an expected optimization time of $O(n\sqrt{\log n})$. Note that all other choices of $\lambda \in [\omega(1), o(\log n)]$ give an asymptotically better runtime as well, so there is some indication that this approach is useful also for problems for which analyzing the optimal parameter choices is not possible.

Note that our result solves a longstanding problem in the theory of evolutionary algorithms community as it gives a positive answer to the question whether crossover can provably have a significant benefit also for easy optimization problems like the OneMax test function class. The previously best known crossover-based genetic algorithm for OneMax was given by Sudholt [42]. It is by a (constant) factor of 2 faster than the $(1+1)$ Evolutionary Algorithm when the standard mutation probability of $1/n$ is used, and by a factor of 2.3 for the optimal mutation probability $(1+\sqrt{5})/(2n)$. Note that in this statistics, we disregard the algorithm *shuffle GA* investigated in [29], since it only gives an improvement for the two particular OneMax functions $x \mapsto \sum_i x_i$ and $x \mapsto n - \sum_i x_i$. For all other OneMax functions, the runtime seems to be larger than the one of the $(1+1)$ EA; moreover, the average runtime over all OneMax functions is even exponential.

The insight into the working principles of the $(1+(\lambda,\lambda))$ GA gained in the theoretical analysis can be used to design a fitness-dependent choice of λ giving an even better expected runtime. If in each iteration we chose λ to be of order $\sqrt{n/d}$, where d is the fitness-distance to the optimum, the resulting algorithm has a linear expected optimization time only. Interestingly, this binary unbiased algorithm has the same asymptotic performance as the best known (and slightly artificial) 2-ary unbiased black-box algorithms for the OneMax function presented in [16]. It seems that generally the concept of fitness-dependent parameter choices is not too well-understood from the theoretical point of view. The only other theoretical result we are aware of in the domain of evolutionary algorithms for discrete search spaces that yields a provable advantage over the respective algorithm with constant mutation rates is a 20% runtime improvement via a fitness-dependent choice of the mutation probability when optimizing the LeadingOnes test function via the $(1+1)$ EA [9]. A couple of other theoretical investigations of bio-inspired search heuristics with fitness-dependent mutation rate exist [35,48,49], however without proving an advantage over the respective constant mutation rate algorithms.

We complement our theoretical results with an experimental evaluation. In a nutshell, the experiments show that the asymptotic advantages of our $(1+(\lambda,\lambda))$ GA over the $(1+1)$ EA are visible already for small instance sizes. Our $(1+(\lambda,\lambda))$ GA remains superior when optimizing random linear functions with coefficients in $[1,2]$ and royal road functions with block size five. For the latter, however, we need to slightly alter the selection rules so that, when possible, the parent individual is not selected. All our results indicate that the particular choice of λ is not very critical. Also, for reasonable problem sizes, a constant λ is sufficient (hence guessing a functional relation like $\lambda = \sqrt{\log n}$ is not necessary). The experiments also confirm a good performance of the fitness-dependent variant of the $(1+(\lambda,\lambda))$ GA.

Since the fitness-dependent parameter choice was very successful (giving provably a linear expected runtime and also performing well in experiment), but possibly hard to find without theoretical analyses, we also investigate a simple self-adjusting[4] choice of λ. To this aim, we imitate the one-fifth success rule from evolution strategies (ES), which was independently discovered in [37,10,39]. For a suitable constant $F > 1$, we multiply λ by $F^{1/4}$ after each unsuccessful iteration and we divide it by F after each iteration that found a superior solution. As our experiments show, this simple mechanism seems to find good (variable) values for λ, so that the resulting optimization times are among the best ones seen in all our experiments. This is quite surprising. While in continuous domains self-adjusting parameter choices (for example, the step size of an ES) are a common technique that is understood also from a theoretical perspective [5,26,24], this approach seems less used in discrete domains. In particular, we are not aware of a comparably simple self-adjustment of the population size. We are aware of a number of early works deriving self-adjusting populations sizes, e.g., from schema theory [21] or via giving individuals a lifetime [3]. An approach possibly closest resembling ours was given in [30]. Here, a parallel EA was investigated in which the number of parallel instances is doubled after each unsuccessful run. This approach, however, only led to a reduction of the parallel runtime, not the total optimization time. In this light, we are optimistic that our new results on (simple) self-adjustment in discrete domains may revive this research direction.

2. The algorithm

We now explain in detail our $(1+(\lambda,\lambda))$ GA. We assume that we have a search space with bit string representation, but there is no reason why the algorithm should not be extensible to other discrete search spaces. We denote by n the length of the bit strings. For $x \in \{0,1\}^n$, we write $x = x_1 \dots x_n$. For all $k \in \mathbb{Z}_{>0}$ we abbreviate $[k] := \{1, 2, \dots, k\}$ and $[0..k] := [k] \cup \{0\}$. For $0 < p < 1$, $\mathcal{B}(n,p)$ denotes the binomial distribution with n trials and success probability p. That is, for every $\ell \in [0..n]$ we have $\mathcal{B}(n,p)(\ell) = \binom{n}{\ell} p^\ell (1-p)^{n-\ell}$.

Our $(1+(\lambda,\lambda))$ GA will use the following mutation and crossover operators.

Definition 1 *(Mutation operator* $\mathrm{mut}_\ell(\cdot)$*).* For $x \in \{0,1\}^n$ and $\ell \in [0..n]$, let $\mathrm{mut}_\ell(x)$ be a bit string that is obtained from x by a random ℓ-bit mutation. That is, we choose a set of ℓ different positions in $[n]$ uniformly at random and we create $\mathrm{mut}_\ell(x)$ from x by flipping the bit-values in these ℓ positions.

As for the standard $(1+1)$ EA, in our $(1+(\lambda,\lambda))$ GA the *step size* ℓ is a random variable itself. In each iteration of the algorithm, it is sampled from $\mathcal{B}(n,p)$, where p denotes the *mutation probability*. In the standard $(1+1)$ EA we typically have $p = 1/n$. Since we aim at a faster exploration, we consider a general mutation probability p, typically larger than $1/n$.

Unlike the $(1+1)$ EA we create in each iteration of the mutation phase λ offspring. To allow a comparison between these offspring, we need to ensure that they all have the same distance ℓ from the parent x. Otherwise, as will become clear in the discussion of the algorithm, once we are close to the optimum, the offspring which is closest to the parent will typically have the best fitness while it is likely to have little or no information that helps us improving the current best solution. We therefore correlate the step size ℓ of the λ mutants. As we shall see below, the step sizes between two iterations are, of course, independent of each other.

[4] We follow the standard classification of parameter control mechanisms as presented in [22]. Both the fitness-dependent and the self-adjusting algorithm classify as *adaptive* mechanisms. Other authors would call the first variant adaptive and the second self-adaptive.

Algorithm 1: The $(1 + (\lambda, \lambda))$ GA with offspring population size λ, mutation probability p, and crossover probability c.

```
1  Initialization: Sample $x \in \{0,1\}^n$ uniformly at random and query $f(x)$;
2  Optimization: for $t = 1, 2, 3, \ldots$ do
3      Mutation phase: Sample $\ell$ from $\mathcal{B}(n, p)$;
4      for $i = 1, \ldots, \lambda$ do
5          Sample $x^{(i)} \leftarrow \mathrm{mut}_\ell(x)$ and query $f(x^{(i)})$;
6      Choose $x' \in \{x^{(1)}, \ldots, x^{(\lambda)}\}$ with $f(x') = \max\{f(x^{(1)}), \ldots, f(x^{(\lambda)})\}$ u.a.r.;
7      Crossover phase: for $i = 1, \ldots, \lambda$ do
8          Sample $y^{(i)} \leftarrow \mathrm{cross}_c(x, x')$ and query $f(y^{(i)})$;
9      If exists, choose $y \in \{y^{(1)}, \ldots, y^{(\lambda)}\} \setminus \{x\}$ with $f(y) = \max\{f(y^{(1)}), \ldots, f(y^{(\lambda)})\}$ u.a.r.; otherwise, set $y := x$;
10     Selection step: if $f(y) \geq f(x)$ then $x \leftarrow y$;
```

Definition 2 *(Crossover operator* $\mathrm{cross}_c(\cdot,\cdot)$*).* For two bit strings $x, x' \in \{0,1\}^n$, let $y = \mathrm{cross}_c(x, x') \in \{0,1\}^n$ be obtained by taking for every position $i \in [n]$ the i^{th} entry in y from x' with probability c, and taking it from x otherwise. Here $c \in [0,1]$ denotes the *crossover probability*, which is a parameter of the algorithm.

The crossover operator $\mathrm{cross}_c(\cdot,\cdot)$ is well known in the literature as a *biased* or *parameterized* uniform crossover, see, e.g., [44,40].

The pseudo-code of the $(1 + (\lambda, \lambda))$ GA is given in Algorithm 1. The algorithm starts with a random initial bit string x. Each iteration of the algorithm consists of a mutation phase, a crossover phase, and a selection step.

In the *mutation phase*, after sampling the random step size ℓ from $\mathcal{B}(n, p)$, we create λ offspring from x using the mutation operator $\mathrm{mut}_\ell(\cdot)$. From these λ individuals, one with largest fitness is selected to take part in the crossover phase. If there are more than one offspring with the maximal fitness, we choose one of them uniformly at random (u.a.r.). We denote this offspring by x'.

In the *crossover phase* we use the operator $\mathrm{cross}_c(\cdot,\cdot)$ to create λ new offspring from the parent individual x and the winner x' of the mutation phase. From these λ offspring we choose the one with largest fitness. If there are several such offspring, we choose uniformly at random one of those which do not equal the parent solution x. This tie-braking rule ensures that we do not waste a whole iteration just for reconstructing the parent solution. As we shall see in the experimental analyses in Section 4, this will be crucial for functions whose fitness landscape has plateaus of equal fitness. Our selection rule ensures that we perform a random walk on such a plateau, without staying at the same point for too long. Due to the symmetry properties of OneMax, it is clear that all results proven below also hold for the slightly simpler algorithm in which an arbitrary offspring of maximal fitness is taken in line 9.

In the *selection step* the parent x is replaced by the winning individual y of the crossover phase if the fitness of y is at least as large as the fitness of x.

In Algorithm 1 we do not specify a *termination criterion*. This is justified by the fact that in this scientific work we are mainly interested in the first point in time in which we evaluate a search point of optimal fitness. In a practical application, of course, one needs to specify a termination criterion.

2.1. Elementary properties and parameter choice

Algorithm 1 is well-defined for all offspring populations sized $\lambda \in \mathbb{Z}_{>0}$ and all mutation and crossover probabilities $p, c \in [0,1]$. Without proof, we note that the algorithm does not converge to an optimal solution when $p = 0$ or $c = 0$, or $p = c = 1$. In all other cases, it finds (and keeps) the optimum eventually.

For $c = 1$ we have $\mathrm{cross}_c(x, x') = x'$ with probability one, so the crossover phase has no effect and the $(1 + (\lambda, \lambda))$ GA reduces to a variant of the $(1 + \lambda)$ EA. Compared to the standard $(1 + \lambda)$ EA, here all offspring in one mutation phase have the same Hamming distance ℓ from the parent. However, each offspring individually has the same distribution as when sampled with standard bit mutation with mutation probability p. In particular, for $c = 1$, $p = 1/n$, and $\lambda = 1$ we regain the classic $(1 + 1)$ EA. For all values $0 < c < 1$, the $(1 + (\lambda, \lambda))$ GA is a truly crossover-based one.

It is not difficult to see that the $(1 + (\lambda, \lambda))$ GA is an unbiased algorithm of arity two (see [32] for a definition of unbiasedness). Consequently, all runtime bounds we shall prove later and all experimental results we obtain for the OneMax test function are true as well for any other objective function with fitness landscape equivalent to the one of OneMax, that is, any generalized OneMax function $f_z : \{0,1\}^n \to \mathbb{Z}; x \mapsto |\{i \in [n] \mid x_i = z_i\}|$. We point out this seemingly natural fact to avoid a confusion with [29], which presents an algorithm for the OneMax problem that is not unbiased.

A single application of the mutation and the crossover operator gives an offspring $\mathrm{cross}_c(x, \mathrm{mut}_p(x))$ that has the distribution of a search point generated from x by standard-bit mutation with mutation probability pc. Since for many mutation-based evolutionary algorithm a mutation probability of $1/n$ is the recommended choice [6] or is even provably optimal [47], we would recommend to use our $(1 + (\lambda, \lambda))$ GA with p, c satisfying $pc = 1/n$. In this approach, starting with a high-rate mutation and then using crossover with the parent to reduce its possibly malicious effect, the role of crossover can be seen as a *genetic repair* mechanism. We are not aware of any previous approaches in discrete optimization, whereas in evolution strategies this seems to be well-known [7,8].

In the remainder of this paper, we shall often parameterize the mutation probability by $p = k/n$, so that k is the expected number of bits that the mutation operator changes. With this, the recommended crossover probability c becomes $1/k$. Our mathematical analysis for the OneMax function in Section 3.1 will show that indeed in many situations $p = k/n$ and $c = 1/k$ are the optimal choices.

3. Runtime analysis

In this section, we conduct a rigorous runtime analysis for several variants of our algorithm when optimizing the test function OneMax, which traditionally is the first test case regarded in runtime analysis.

For all runtime analysis results, recall that the standard performance measure is the *optimization time* (also "*runtime*") defined as follows.

Definition 3 *(Optimization time).* The *optimization time* of an algorithm $\mathcal{A}$ on a function f is the random variable $T = T(\mathcal{A}, f)$ that denotes the number of fitness evaluations performed until for the first time an optimal solution is evaluated.

Observe that one iteration of the $(1+(\lambda,\lambda))$ GA requires 2λ function evaluations. Assume that we are working with a static value for λ. If t^* is the first iteration in which the $(1+(\lambda,\lambda))$ GA evaluates an individual with $f(x) = \max_{z\in\{0,1\}^n} f(z)$, then the optimization time T of this run is between $2(t^*-1)\lambda + 2$ and $2t^*\lambda + 1$; recall that also the initial search point has to be evaluated. Since hence the optimization time and the first iteration t^* to find an optimum deviate basically by a factor of 2λ, we shall argue with either of the two notions, but state the main results in terms of the optimization time. This will be different in Section 3.2, where a varying λ forbids this simplification.

3.1. Static parameter settings

In this section, we conduct a runtime analysis for the $(1+(\lambda,\lambda))$ GA on the classic test function $f = $ OneMax. Among others, this will give further evidence for our previous suggestion to take the crossover probability p as $1/k$, where k is the parameter defining the mutation probability k/n. The main result is the following runtime bound, which in particular shows that our $(1+(\lambda,\lambda))$ GA for all $k, \lambda \in [\omega(1), o(\log n)]$ is faster than all evolutionary approaches for which the runtime on OneMax is known. As discussed in the introduction, this is the first time that a crossover-based evolutionary algorithm provably beats the $\Omega(n \log n)$ optimization time barrier for OneMax.

Theorem 4. *Let $k, \lambda \geq 2$, possibly depending on n. The expected optimization time of the $(1+(\lambda,\lambda))$ GA with mutation probability $p = k/n$ and crossover probability $c = 1/k$ on the* OneMax *function is*

$$O\left(\left(\frac{1}{k} + \frac{1}{\lambda}\right) n \log n + (k+\lambda) n\right).$$

In particular, for both k and λ in $\Theta(\sqrt{\log n})$, the expected optimization time is of order at most $n\sqrt{\log n}$.

To analyze the expected optimization time of Algorithm 1 on OneMax, we first bound the probability that the mutation phase (lines 4 to 6) creates an individual that is not dominated by the parent, since such an individual cannot produce a better offspring in our crossover phase (and, in fact, in any geometric crossover [34]). A search point x' is said to be *dominated* by x if $x'_i \leq x_i$ for all $i \in [n]$.

We thus say that the mutation phase is *successful* if at least one of the λ offspring $x^{(i)}$ of x is not dominated by x. Since all $x^{(i)}$ have the same Hamming distance ℓ from x, this is equivalent to saying that $f(x^{(i)}) > f(x) - \ell$. Consequently, also the winner individual x' is not dominated by x.

Lemma 5. *In the notation of Algorithm 1, for all ℓ and x, the success probability of the mutation phase is at least $1 - (\frac{f(x)}{n})^{\lambda\ell}$.*

Proof. For $z = \mathrm{mut}_\ell(x)$ we have $f(z) > f(x) - \ell$ if and only if for at least one of the ℓ positions $j \in \{i \in [n] \mid x_i \neq z_i\}$ we have $x_j = 0$. Thus, the probability of this event equals one minus the probability that we flip only positions $j \in [n]$ with $x_j = 1$. Initially, there are $f(x)$ such positions. Thus, $\Pr[f(z) = f(x) - \ell] = \prod_{j=0}^{\ell-1} \frac{f(x)-j}{n-j}$.

Finally, the probability that for all λ offspring $x^{(1)}, \ldots, x^{(\lambda)}$ we have flipped only 1-bits is $\Pr[\forall i \in [\lambda] : f(x^{(i)}) = f(x) - \ell] = (\prod_{j=0}^{\ell-1} \frac{f(x)-j}{n-j})^\lambda \leq (\prod_{j=0}^{\ell-1} \frac{f(x)}{n})^\lambda = (\frac{f(x)}{n})^{\lambda\ell}$. □

Let us use the above lemma to derive some first insight into the influence of the parameters. Note first that a simple union bound argument shows that the success probability of the mutation phase has an upper bound of $\lambda\ell(1 - \frac{f(x)}{n})$. Write $f(x) = n - d$, so d is the distance (both Hamming distance and fitness distance) to the optimum. When d is not excessively large (and this is the difficult and thus more interesting part of the optimization process for the OneMax function), say

$d = o(n/\lambda\ell)$, then both bounds together show that the success probability of the mutation phase is $(1 \pm o(1))\lambda\ell d/n$. Since one mutation phase takes λ fitness evaluations, it is the parameter ℓ that can possibly lead to an improvement over existing algorithms. Recall that, e.g., the classic $(1+1)$ EA finds an improvement in one iteration with probability $\Theta(d/n)$. Consequently, an asymptotic performance gain can only be achieved when $\ell = \omega(1)$, which requires $k = \omega(1)$.

For future discussions, let us also note that when the mutation phase is successful, this typically stems from only a single 0-bit in x being flipped to 1: The probability that x' has at least i of the 0-bits of x flipped to 1, can easily be seen to be of order at most $\lambda(\ell d/n)^i$ for all constant i.

We now turn to the analysis of the crossover phase (lines 7 to 9) of the $(1 + (\lambda, \lambda))$ GA. We call one run of this phase successful if it leads to the creation of a solution y with $f(y) > f(x)$. As discussed above, the crossover phase can only be successful when the winning individual x' satisfies $f(x') > f(x) - \ell$. In the following lemma, we thus analyze the probability that the crossover phase is successful given that the mutation phase was successful.

Lemma 6. *In the notation of Algorithm 1, consider fixed outcomes of ℓ, x, and x'. Then the random outcome y of the crossover phase satisfies*

$$\Pr\bigl[f(y) > f(x) \mid f\bigl(x'\bigr) > f(x) - \ell\bigr] \geq 1 - \bigl(1 - c(1-c)^{\ell-1}\bigr)^{\lambda}.$$

Proof. To prove Lemma 6 one adopts a worst-case view and assumes that $f(x') = f(x) - \ell + 2$ (note that this is the smallest possible fitness value of x' in case $f(x') > f(x) - \ell$: since at least one of the zeros in x must have been flipped to one and since we have flipped ℓ bits in total, we have $f(x') \geq f(x) + 1 - (\ell - 1)$). In this case, the crossover can be successful only if we copy from x' exactly the one bit in x' that is zero in x and one in x'. Thus, for $y^* = \text{cross}_c(x, x')$ we have $\Pr[f(y^*) > f(x) \mid f(x') > f(x) - \ell] \geq c(1-c)^{\ell-1}$.

Therefore, $\Pr[\forall i \in [\lambda] : f(y^{(i)}) \leq f(x) \mid f(x') > f(x) - \ell] \leq (1 - c(1-c)^{\ell-1})^{\lambda}$. $\square$

Let us again comment on how this analysis gives us information on the choice of the parameters, this time the parameter c. We first observe that the estimate of Lemma 6 is relatively sharp. It is exact when $f(x') = f(x) - \ell + 2$, that is, x' is obtained from x by flipping a single 0-bit and $\ell - 1$ bits that are 1. As argued earlier, this is the most likely outcome of a successful mutation phase when $d = n - f(x)$ is small, more precisely, $d = o(n/\lambda\ell)$. Hence when $d = o(n/\lambda\ell)$,

$$\begin{aligned}\Pr\bigl[f(y) > f(x) \mid f\bigl(x'\bigr) > f(x) - \ell\bigr] &\leq \bigl(1 - o(1)\bigr)\bigl(1 - \bigl(1 - c(1-c)^{\ell-1}\bigr)^{\lambda}\bigr) \\ &\leq \bigl(1 - o(1)\bigr)\lambda c(1-c)^{\ell-1}.\end{aligned}$$

Both this upper bound and the lower bound of Lemma 6 are maximized when $c = \Theta(1/\ell)$. Assuming $\ell = \omega(1)$ as argued for earlier, we even have that $c = (1 \pm o(1))(1/\ell)$ is the optimal choice for the crossover probability. Recall that ℓ is binomially distributed with expectation k, so when having $\ell = \omega(1)$, we also know that ℓ is strongly concentrated around its mean k. This shows that taking $c = 1/k$, as argued intuitively in the description of the $(1 + (\lambda, \lambda))$ GA, is indeed a good choice.

For this reason, to ease the calculations, we treat from now on only the case $c = 1/k$. We call an iteration of the main loop of Algorithm 1 successful if it produced a solution y with $f(y) > f(x)$.

Lemma 7. *For $p = k/n$ and $c = 1/k$, the probability that one iteration of the main loop of Algorithm 1, started with an individual x, is successful is at least*

$$C\left(1 - \left(\frac{f(x)}{n}\right)^{\lambda k/2}\right)\bigl(1 - e^{-\lambda/(8k)}\bigr)$$

for some constant $C > 0$.

Proof. Observe that in the case of $k = 1$, y has the same distribution as the offspring generated by a $(1+1)$ EA. Since such an offspring has a probability of at least $(1/e)(n - f(x))/n$ of being better than its parent x, our claim holds in this case. We may thus assume $k \geq 2$ in the remainder of this proof.

Let L denote the step size, i.e., let L be the random variable sampled in line 3 of Algorithm 1. By the law of total probability,

$$\Pr\bigl[f(y) > f(x)\bigr] \geq \sum_{\ell=\lceil k/2\rceil}^{\lfloor 3k/2\rfloor} \Pr\bigl[f(y) > f(x) \mid L = \ell\bigr]\Pr[L = \ell].$$

Using Lemmas 5 and 6, we estimate

$$\begin{aligned}\Pr\bigl[f(y) > f(x) \mid L = \ell\bigr] &= \Pr\bigl[f\bigl(x'\bigr) > f(x) - L \mid L = \ell\bigr] \cdot \Pr\bigl[f(y) > f(x) \mid \bigl(f\bigl(x'\bigr) > f(x) - L\bigr) \wedge (L = \ell)\bigr] \\ &\geq \left(1 - \left(\frac{f(x)}{n}\right)^{\lambda\ell}\right)\bigl(1 - \bigl(1 - c(1-c)^{\ell-1}\bigr)^{\lambda}\bigr).\end{aligned}$$

Using the facts that $k \geq 2$, $c = 1/k$, and that we are only interested in values $\ell \in [k/2, 3k/2]$ we compute

$$\left(1 - c(1-c)^{\ell-1}\right)^{\lambda} \leq \left(1 - \frac{1}{k}\left(1 - \frac{1}{k}\right)^{3k/2}\right)^{\lambda} \leq \left(1 - 1/(8k)\right)^{\lambda} \leq e^{-\lambda/(8k)},$$

where we use in the second step the fact that for all $m \geq 2$ it holds that $(1-1/m)^m \geq 1/4$, and in the third step that for all $m \geq 2$, $1/e \geq (1-1/m)^m \geq 1/(2e)$ (in the following, we shall use these inequalities without explicit mention).

Since we are interested only in $\ell \geq k/2$, we may estimate $(\frac{f(x)}{n})^{\lambda\ell} \leq (\frac{f(x)}{n})^{\lambda k/2}$. Thus, in total we obtain

$$\Pr\left[f(y) > f(x) \mid L = \ell\right] \geq \left(1 - \left(\frac{f(x)}{n}\right)^{\lambda k/2}\right)\left(1 - e^{-\frac{\lambda}{8k}}\right),$$

which is independent of $\ell \in [k/2, 3k/2]$.

Finally, it is not hard to see that $\sum_{\ell=\lceil k/2\rceil}^{\lfloor 3k/2\rfloor} \Pr[L = \ell]$ is constant. For $k = \omega(1)$ this follows easily from Chernoff's bound (see [11] for an introduction to basic tail bounds in probability theory). For constant k we trivially have $\Pr[L = k] = \Theta(1)$. □

Proof of Theorem 4. We use the classic fitness level approach [43,45] and bound the number of iterations needed by the sum (over all fitness levels) of the expected times to leave this level. The latter is the reciprocal of the probability of generating a superior solution as estimated in Lemma 7.

The expected total number of iterations needed by Algorithm 1 to find the all-ones bit string is thus bounded from above by

$$C^{-1}\left(1 - e^{-\lambda/(8k)}\right)^{-1} \sum_{j=0}^{n-1} \left(1 - \left(\frac{j}{n}\right)^{\lambda k/2}\right)^{-1}, \tag{1}$$

where C is the constant from Lemma 7.

Changing the order of summation, we have

$$\sum_{j=0}^{n-1} \left(1 - \left(\frac{j}{n}\right)^{\lambda k/2}\right)^{-1} = \sum_{j=1}^{n} \left(1 - \left(1 - \frac{j}{n}\right)^{\lambda k/2}\right)^{-1}.$$

For $\lambda kj > 2n$, we estimate $(1 - \frac{j}{n})^{\lambda k/2} \leq \exp(-\frac{\lambda kj}{2n}) \leq e^{-1}$. By Bernoulli's inequality it holds that $(1-x)^m \leq (1+mx)^{-1}$ for $m \in \mathbb{Z}_{>0}$ and $x \in [0,1]$. Thus, for $\lambda kj \leq 2n$ we get $(1 - \frac{j}{n})^{\lambda k/2} \leq (1 + \frac{\lambda kj}{2n})^{-1} = 1 - \frac{\lambda kj}{2n + \lambda kj} \leq 1 - \frac{\lambda kj}{4n}$. Overall, this shows that

$$\begin{aligned}\sum_{j=0}^{n} \left(1 - \left(\frac{j}{n}\right)^{\lambda k/2}\right)^{-1} &\leq \sum_{j=0}^{n} \left(1 - \max\left\{1 - \frac{\lambda kj}{4n}, e^{-1}\right\}\right)^{-1} \\ &= \sum_{j=0}^{n} \max\left\{\frac{4n}{\lambda kj}, \left(1 - e^{-1}\right)^{-1}\right\} = O\left(\frac{n \log n}{\lambda k} + n\right).\end{aligned} \tag{2}$$

Next we bound the second term in (1), i.e., the factor $(1 - e^{-\lambda/(8k)})^{-1}$. Clearly, if $\lambda/(8k) \geq 1$, then $(1 - e^{-\lambda/(8k)})^{-1} = O(1)$. For $\lambda/(8k) < 1$ we use the fact that for all $x > 0$ it holds that $\exp(-x) < 1 - x + \frac{x^2}{2}$ and bound

$$e^{-\lambda/(8k)} < 1 - \frac{\lambda}{8k} + \frac{1}{2}\left(\frac{\lambda}{8k}\right)^2 \leq 1 - \frac{\lambda}{8k} + \frac{1}{2}\frac{\lambda}{8k} = 1 - \frac{\lambda}{16k}.$$

This shows that for $\lambda/(8k) < 1$ we have

$$\left(1 - e^{-\lambda/(8k)}\right)^{-1} < \left(\lambda/(16k)\right)^{-1} = O(k/\lambda).$$

Thus, overall, it holds that

$$\left(1 - e^{-\lambda/(8k)}\right)^{-1} = O\left(\max\{1, k/\lambda\}\right). \tag{3}$$

Replacing the terms in (1) by (2) and (3) it is thus easy to see that the overall number of iterations needed is

$$O\left(\max\left\{1, \frac{k}{\lambda}\right\}\left(\frac{n \log n}{\lambda k} + n\right)\right).$$

Since one iteration of Algorithm 1 requires 2λ fitness evaluations, the expected optimization time of the algorithm is

$$O\left(\lambda\left(\frac{n \log n}{\lambda k} + n\right) + k\left(\frac{n \log n}{\lambda k} + n\right)\right) = O\left(\left(\frac{1}{k} + \frac{1}{\lambda}\right) n \log n + (k + \lambda) n\right). \quad \square$$

Table 1
Standard deviations in % of the mean optimization time observed in 1000 runs of three of our algorithms, the $(1+1)$ EA, and the $(2+1)$ GA of [42]. The problem size is $n = 1000$.

	OneMax	Linear functions	RR_5
$(1+(8,8))$ GA	9.9	10.2	14.9
fitness-dependent	6.7		
self-adjusting	6.6	12	12.9
$(1+1)$ EA	21.1	19.4	25.3
$(2+1)$ GA [42]	16.8	20.5	21.6

3.2. Fitness-dependent parameter settings

In this section we prove that a suitable fitness-dependent choice of λ, ensuring larger λ-values towards the more difficult end of the optimization process, provably yields an asymptotic improvement. In fact, this reduces the expected optimization time to $O(n)$. We are not aware of any previous results showing more than a constant-factor gain through an adaptive parameter choice for a discrete search problem. Note that $O(n)$ is also the best known upper bound for the 2-ary unbiased black-box complexity of OneMax, cf. [15,16], that is, no better binary crossover-based unbiased algorithms (including arbitrarily problem-specific ones) are known for the OneMax problem.

Theorem 8. *Consider the $(1+(\lambda,\lambda))$ GA with standard parameters $p = \lambda/n$ and $c = 1/\lambda$ together with a fitness-dependent choice of λ such that in the beginning of each iteration λ is set to $\lambda = \lceil\sqrt{n/(n-f(x))}\rceil$. Then expected optimization time on* OneMax *is $O(n)$.*

Proof. We first show that the fitness-dependent choice of λ ensures that each iteration of the main loop has a constant success probability. Fix a value for $x \in \{0,1\}^n$ and $\lambda = \lceil\sqrt{n/(n-f(x))}\rceil$ at the beginning of the main loop. By Lemma 7—recall that we have $k = \lambda$ in the notation there—the probability that y at the end of the main loop satisfies $f(y) > f(x)$ is at least

$$C\left(1-\left(\frac{f(x)}{n}\right)^{\lambda^2/2}\right)\left(1-e^{-1/8}\right)$$

for some constant $C > 0$. Since

$$\left(\frac{f(x)}{n}\right)^{\lambda^2/2} \le \left(1-\frac{n-f(x)}{n}\right)^{\frac{n}{2(n-f(x))}} \le e^{-1/2},$$

this success probability is bounded from below by a constant.

Consequently, the expected number of iterations performed in each fitness level is also constant, and thus a pessimistic estimate for the total expected optimization time is

$$O\left(\sum_{f=0}^{n-1}\sqrt{n/(n-f)}\right) = O\left(\sqrt{n}\int_1^n \sqrt{1/i}\,\mathrm{d}i\right) = O(n). \qquad \square$$

4. Experimental results

In this section, we conduct an experimental analysis for our GAs, aiming at results that seem difficult to obtain via theoretical means: (i) Precise runtime results for concrete problem sizes, (ii) precise information on the optimal choice of the parameters, and (iii) runtime results for other test function classes (linear functions, royal road functions).

The experimental setup is as follows. All algorithms were implemented in C++ with random numbers sampled using the random number generator in the GNU scientific library. All reported numbers are derived from 1000 independent runs of the respective algorithms. In all experiments the mutation probability is k/n and the crossover probability $1/k$. Except for Fig. 3 we have $k = \lambda$.

For reasons of clarity and brevity, we mostly report average optimization times. Standard deviations are given in Table 1 and in Fig. 2. The interested reader can find some statistics (precisely, the 2, 25, 50, 75, and 98 percentiles, mean values, and standard deviations in percentage of the mean value) of the experiments in Appendix A.

4.1. Influence of the parameters

In Fig. 1 we compare the average optimization time of the $(1+(\lambda,\lambda))$ GA for different values of λ and n with the average optimization time of the $(1+1)$ EA. We use the $(1+1)$ EA as reference as this is among the best possible mutation-based algorithms, as independently shown in [32,43,47]. We observe that already a small constant λ gives a stable advantage of the $(1+(\lambda,\lambda))$ GA over the $(1+1)$ EA. Also, the particular choice of the λ value seems not very delicate. This is further

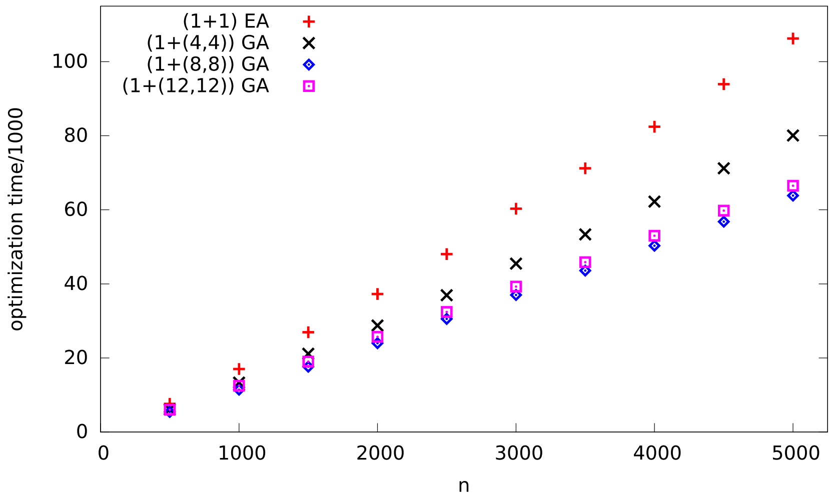

Fig. 1. Average optimization time of the $(1+(\lambda,\lambda))$ GA on OneMax for different values of λ and n.

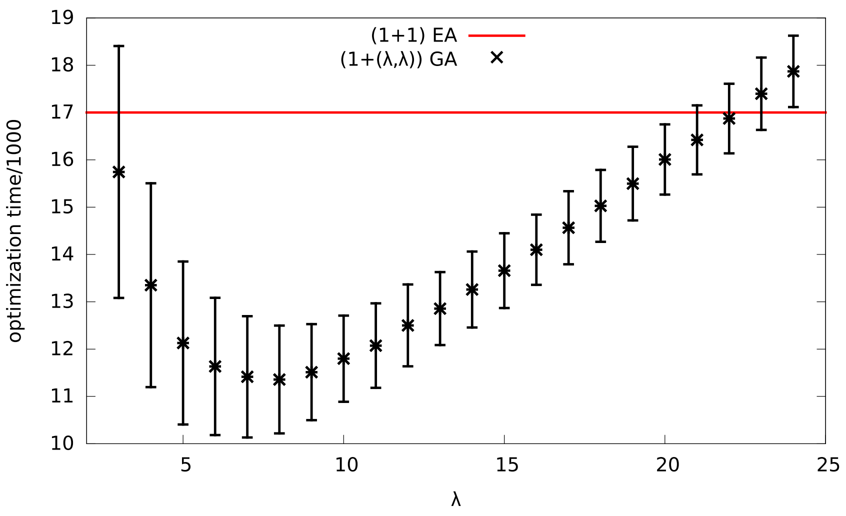

Fig. 2. Average optimization time and standard deviation of the $(1+(\lambda,\lambda))$ GA on OneMax ($n = 1000$) for different values of λ. For reasons of space, the average optimization time for $\lambda = 2$, which is 21 074, is not depicted.

detailed in Fig. 2, where we compare the average optimization times for fixed $n = 1000$ and varying λ. We also see (Fig. 3) that the exact relation of k and λ is not too critical. For a broad range of combinations, the $(1+(\lambda,\lambda))$ GA performs much better than the $(1+1)$ EA, which for $n = 1000$ has an average optimization time of 17 001.

We also see that for small and large λ the runtime increases. For large λ this is due to the fact that the increased probability of a successful iteration comes at the cost of too many fitness evaluations. That is, the additional benefit from exploring larger areas of the search space cannot be exploited by the algorithm. For too small λ, the algorithm does not explore sufficiently large portions of the search space, thus staying in the same fitness level for too long.

4.2. Fitness-dependent and self-adjusting parameter choices

While in the previous subsection we saw that the choice of the parameter λ is not too delicate, we now show that we can make our lives even easier and let the $(1+(\lambda,\lambda))$ GA find a suitable value for λ self-adaptively.

When designing this self-adaptive GA, we imitate the classic 1/5th rule from evolution strategies. To this end, see also Algorithm 2 for the pseudo-code of this self-adaptive $(1+(\lambda,\lambda))$ GA, we initialize $\lambda = 1$ and we update λ as follows. If at the end of one iteration we have $f(y) > f(x)$, we replace λ by λ/F, and we replace it by $\lambda F^{1/4}$ otherwise. Here F is a (constant) parameter of the algorithm. Where an integer is required (e.g., lines 5 and 9 of Algorithm 2) we round λ to its closest integer, i.e., instead of λ we regard $\lfloor\lambda\rfloor$ if the fractional part $\{\lambda\} := \lambda - \lfloor\lambda\rfloor$ of λ is less then $1/2$ and we regard $\lceil\lambda\rceil$ otherwise.

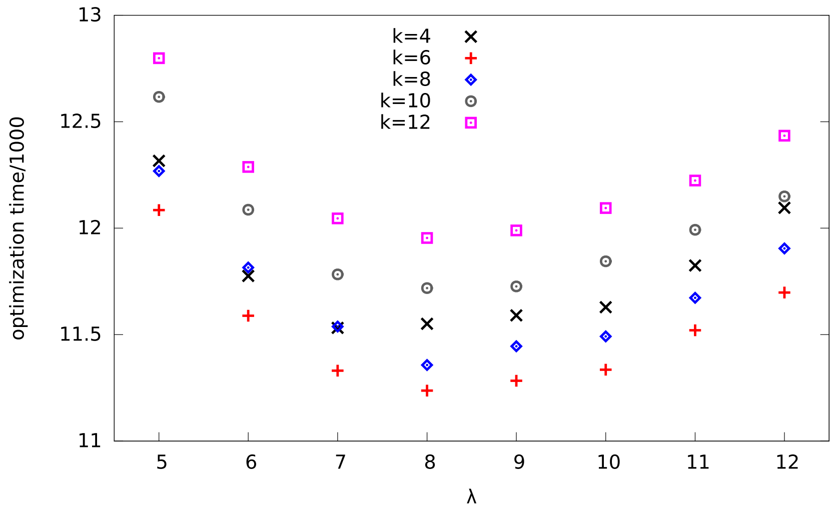

Fig. 3. Optimization time of the $(1+(\lambda,\lambda))$ GA for different combinations of λ and k on OneMax for $n = 1000$. Note that all values are between 66% and 75% of the runtime of the $(1+1)$ EA.

Algorithm 2: The self-adjusting $(1+(\lambda,\lambda))$ GA with mutation probability p, crossover probability c, and update strength F.

```
1  Initialization: Sample $x \in \{0,1\}^n$ uniformly at random and query $f(x)$;
2  Initialize $\lambda \leftarrow 1$;
3  Optimization: for $t = 1, 2, 3, \ldots$ do
4      Mutation phase: Sample $\ell$ from $\mathcal{B}(n,p)$;
5      for $i = 1, \ldots, \lambda$ do
6          Sample $x^{(i)} \leftarrow \text{mut}_\ell(x)$ and query $f(x^{(i)})$;
7      Choose $x' \in \{x^{(1)}, \ldots, x^{(\lambda)}\}$ with $f(x') = \max\{f(x^{(1)}), \ldots, f(x^{(\lambda)})\}$ u.a.r.;
8      Crossover phase:
9      for $i = 1, \ldots, \lambda$ do
10         Sample $y^{(i)} \leftarrow \text{cross}_c(x, x')$ and query $f(y^{(i)})$;
11     Choose $y \in \{y^{(1)}, \ldots, y^{(\lambda)}\}$ with $f(y) = \max\{f(y^{(1)}), \ldots, f(y^{(\lambda)})\}$ u.a.r.;
12     Selection step:
13     if $f(y) > f(x)$ then $x \leftarrow y$; $\lambda \leftarrow \max\{\lambda/F, 1\}$;
14     if $f(y) = f(x)$ then $x \leftarrow y$; $\lambda \leftarrow \min\{\lambda F^{1/4}, n\}$;
15     if $f(y) < f(x)$ then $\lambda \leftarrow \min\{\lambda F^{1/4}, n\}$;
```

That the strategy described above is a reasonable implementation of the 1/5th rule (at least in the context of evolution strategies) was argued in [4]. In the 1/5th rule, the motivation for multiplying a parameter by $F^{1/4}$ in an unsuccessful iteration and dividing it by F otherwise is that if we have a success in every fifth iteration then the parameter stays roughly constant. Having a success every fifth iteration is considered to be a fair trade-off between exploration and exploitation.

In Fig. 4 we present the average optimization times of the GA with the fitness-dependent parameter setting $\lambda = \sqrt{n/(n-f(x))}$ as in Theorem 8, of the self-adjusting $(1+(\lambda,\lambda))$ GA with update strength $F = 1.5$, the same choice as in [4], and as comparison the $(1+1)$ EA and the $(1+(8,8))$ GA. We observe that the self-adjusting GA performs basically as good the fitness-dependent GA with the asymptotically optimal choice of λ.

The results for one typical run with $n = 1000$ are depicted in Fig. 5. Interestingly, the self-adjusting choice of λ computes values for λ that are surprisingly close to the asymptotically optimal choice of λ. Note that, naturally, in Fig. 5 on the x-axis we plot the iterations, each of which has effort proportional to the current λ-value. Rescaling by this would show that, as expected, the progress towards the end of the process is much slower than in the beginning.

4.3. Other test functions

We analyze our GAs on two other classic test functions, linear functions with random weights and royal road functions.

In Fig. 6—ignore for the moment the data points for the $(2+1)$ GA, which will be discussed in the next section—we present runtimes for optimizing linear functions $f : \{0,1\}^n \to \mathbb{R}; x \mapsto \sum_{i=1}^n w_i x_i$ with weights w_i chosen independently and uniformly at random from the interval $[1,2]$. The experiments show that our GA, both for a fixed value of $\lambda = 8$ and for the self-adjusting choice of λ, outperforms the $(1+1)$ EA also for these linear fitness functions.

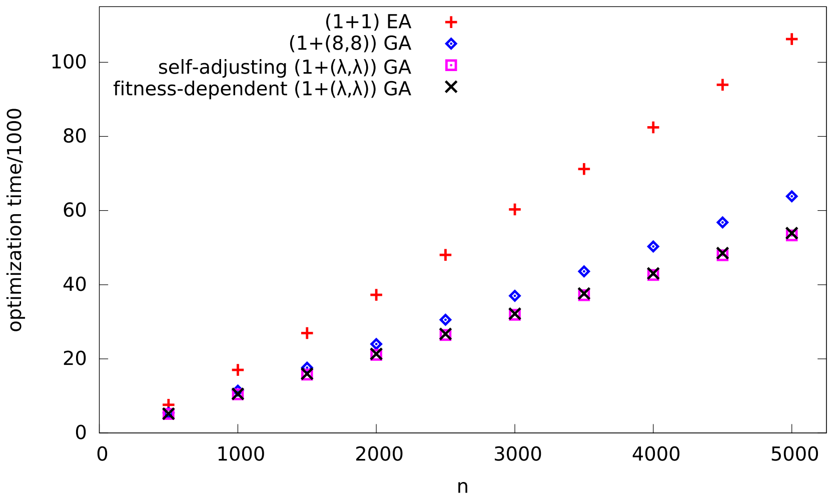

Fig. 4. Average optimization times for OneMax of the $(1 + (\lambda, \lambda))$ GA with fitness-dependent and self-adjusting λ (with $F = 1.5$).

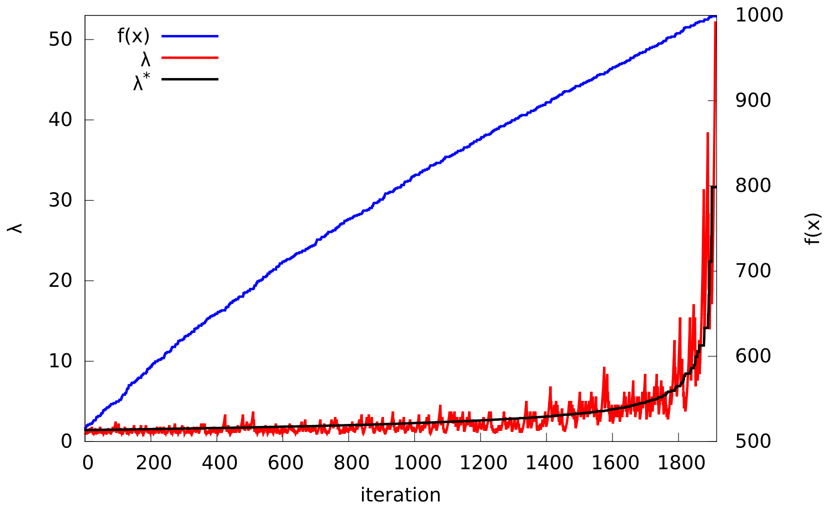

Fig. 5. Evolution of $f(x)$ and λ in one representative run of the $(1 + (\lambda, \lambda))$ GA with self-adjusting λ and update strength $F = 1.5$ on OneMax with $n = 1000$. The bottom non-rugged curve plots $\lambda^* = \sqrt{n/(n - f(x))}$.

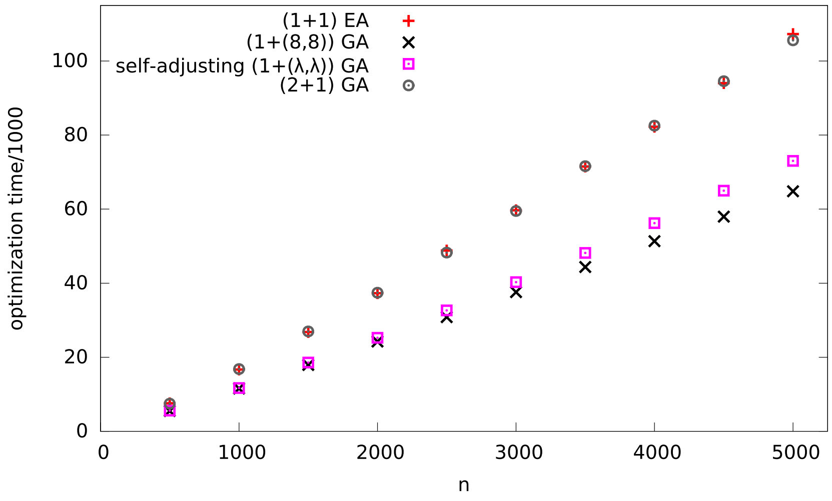

Fig. 6. Average optimization time of the $(1 + (\lambda, \lambda))$ GA on linear functions with random weights $w_i \in [1, 2]$.

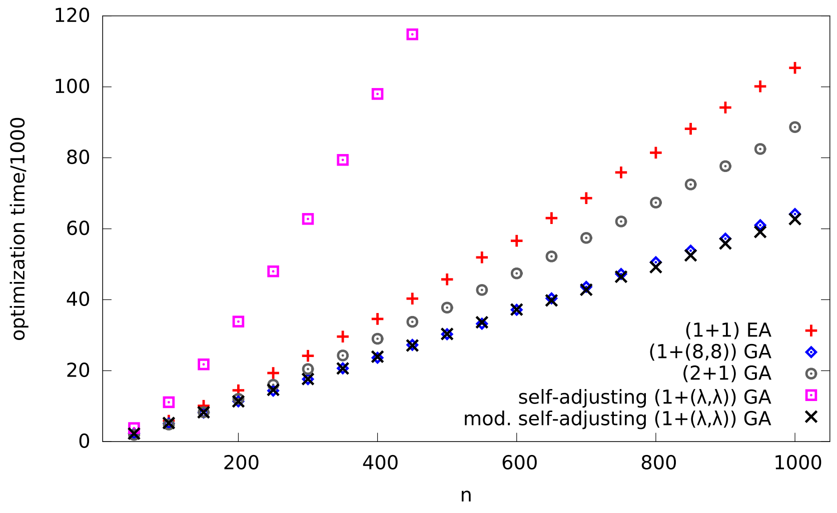

Fig. 7. Average optimization time of the $(1+(\lambda,\lambda))$ GA with fixed and self-adjusting choices of λ on RR_5.

Royal road functions were originally proposed in [33]. A *royal road function of block size k* assigns to every bit string $x \in \{0,1\}^n$ the fitness $RR_k(x) := |\{i \mid x_{(i-1)k+1} = \ldots = x_{\min\{ik,n\}} = 1\}|$, that is, it partitions the string x into n/k consecutive blocks of size k and counts the number of blocks in which all bits are set to one. Like Sudholt [42] we consider in our experiments royal road functions of block size five.

One key difference between OneMax (which is a royal road function of block size one) and royal road functions with larger block sizes is the fact that in the latter large plateaus of equal fitness exist. The typical way to leave such plateaus is via a random walk on the plateau. This requires accepting offspring of equal fitness different from the parent. It is for this reason that we have requested in the crossover phase to select the parent individual only if among the offspring no other one of maximal fitness exists.

Experimental results for the average optimization times of the $(1+(\lambda,\lambda))$ GA and the $(1+1)$ EA on RR_5 are reported in Fig. 7 (ignore again for a moment the data points for the $(2+1)$ GA). While for fixed λ we observe a similar improvement as for OneMax, the self-adjusting rule produces dissatisfactory results. Again, we suspect that this is caused by walks on fitness plateaus, now leading to an increase of the λ-value, and consequently, very expensive iterations. We therefore also use a modified self-adjusting rule. It is identical to the previous except that the λ-value remains unchanged if the winner offspring y has a fitness equal to the one of the parent x. As the figure shows, this seems to solve the problem: The resulting GA has a performance comparable to the one for fixed $\lambda = 8$.

4.4. Comparison with Sudholt's GA

In Figs. 6, 7, and 8 we compare our GAs with Sudholt's $(2+1)$ GA from [42]. The $(2+1)$ GA maintains a population $\mathcal{P}$ of size two. If both search points in $\mathcal{P}$ have the same fitness, a new search point y' is created from them by an unbiased uniform crossover ($\text{cross}_{1/2}(\cdot,\cdot)$) and y' is initialized as the better of the two individuals otherwise. In the mutation phase, standard bit mutation with bit flip probability p is applied to y'. The resulting individual y' replaces the worse of the two individuals z in $\mathcal{P}$ if and only if (i) it is at least as good as z (i.e., if $f(y') \geq f(z)$ holds) and (ii) y' in not yet contained in $\mathcal{P}$.

In Fig. 8, we compare the $(2+1)$ GA using the for OneMax optimal mutation probability $(1+\sqrt{5})/(2n)$ with our algorithms (note that we did not optimize constant factors in most parameters of our algorithms while the $(2+1)$ GA with optimal mutation rate is 14% better on average than the $(2+1)$ GA with standard mutation rate $1/n$, cf. Theorems 2 and 4 as well as Corollary 3 in [42]). For moderate problem sizes, the $(2+1)$ GA with optimized mutation rate outperforms our algorithms moderately. For example, it is 14% faster than the self-adjusting GA for $n = 1000$. As is to be expected from our theoretical findings, this advantage reduces with increasing problem size and eventually reverts. Indeed, for $n = 5000$ we find that the self-adjusting $(1+(\lambda,\lambda))$ GA is, on average, 2% faster than the $(2+1)$ GA. This still small difference seems mostly be caused by the fact that the $(2+1)$ GA shows larger runtime deviations above the expectation. While the median runtime of the $(2+1)$ GA is 0.2% smaller than for our self-adjusting GA, its 75-percentile runtime is 6% larger and its 98-percentile runtime is 30% larger.

For random linear functions (Fig. 6), no advantage of the $(2+1)$ GA over the $(1+1)$ EA is visible, consequently the self-adjusting GA is significantly faster. For royal road functions (Fig. 7), we confirm an advantage of the $(2+1)$ GA over the $(1+1)$ EA (16% for $n = 1000$), but observe a larger advantage of the $(1+(8,8))$ GA (39%) and the modified self-adjusting GA (40%). In these experiments, we used the standard mutation probability of $1/n$ for the $(2+1)$ GA, which seems a reasonable choice given the experimental results in [42].

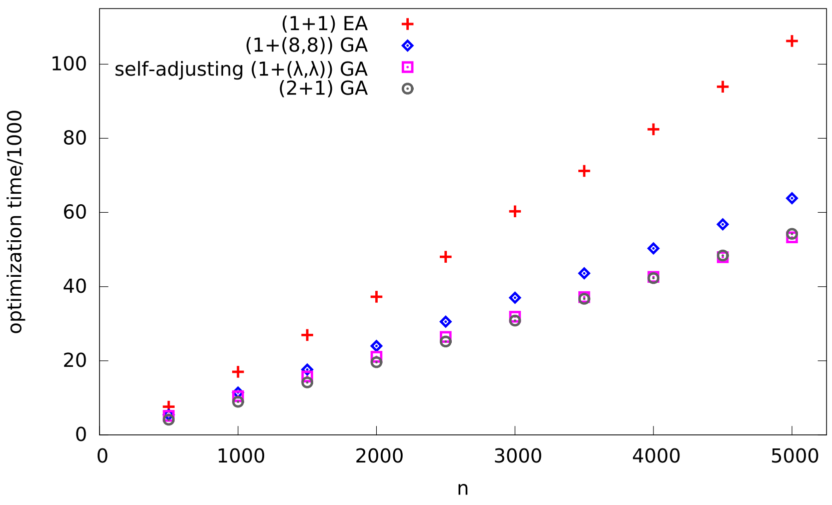

Fig. 8. Comparison of the average optimization times of the $(2+1)$ GA, some of our algorithms, and the $(1+1)$ EA on OneMax.

5. Conclusion

Inspired by a series of recent results in black-box complexity, we developed a class of natural population-based genetic algorithms (GAs) that compare favorably with previous GAs on several classic test function classes. Both the use of crossover as a repair mechanism (known to biologists, but seemingly new to evolutionary discrete optimization) and the self-adjusting parameter setting are promising features that seem to be worth further research. Indeed, one of the most interesting research question arising from our work is for what other problems these two building blocks can be successfully applied, either leading to provable runtime gains for theoretically defined problems or to a better performance in an experimental analysis for real-world problems.

The results in this work were triggered by the observation that many optimal black-box algorithms gain from exploiting inferior solutions. A second interesting question is if there are other properties of these algorithms that we can learn from.

In this work we rediscovered (in the discrete domain) the genetic repair principle from evolution strategies and we gave a useful implementation of a one-fifth rule for a discrete search problem. Hence from a very broad perspective, these results raise the question what other methodology from continuous evolutionary algorithmics could be successfully transfered to discrete search and optimization.

Acknowledgements

The authors would like to thank Anne Auger, Nikolaus Hansen, Thomas Jansen, and Christine Zarges for valuable discussions on various aspect of this project.

Carola Doerr gratefully acknowledges support from the Alexander von Humboldt Foundation and the Agence Nationale de la Recherche (project ANR-09-JCJC-0067-01).

Appendix A. Statistical results for some plots in Section 4

A.1. Statistics for experimental results of Figs. 1, 4, and 8 (runtimes for OneMax)

n	Algorithm	Percentile					Mean	StdDev/ Mean
		2	25	50	75	98		
500	$(1+1)$ EA	4949	6285	7218	8459	12 346	7585	24.1%
500	$(1+(4,4))$ GA	4456	5328	5944	6720	8712	6116	17.9%
500	$(1+(8,8))$ GA	4464	4992	5344	5712	6928	5411	11.2%
500	$(1+(12,12))$ GA	5928	6192	5544	6168	6144	6045	8.1%
500	fitness-dep. $(1+(\lambda,\lambda))$ GA	4222	4848	5176	5512	6252	5192	9.2%
500	self-adj. $(1+(\lambda,\lambda))$ GA	4372	4860	5108	5376	6162	5143	8.4%
500	$(2+1)$ GA	2780	3539	3984	4493	5968	4074	19.0%

(*continued*)

n	Algorithm	Percentile					Mean	StdDev/ Mean
		2	25	50	75	98		
1000	$(1+1)$ EA	11 916	14 419	16 490	18 854	26 678	17 001	21.1%
1000	$(1+(4,4))$ GA	10 088	11 784	13 016	14 560	18 952	13 351	16.1%
1000	$(1+(8,8))$ GA	9616	10 640	11 248	12 032	14 320	11 418	9.9%
1000	$(1+(12,12))$ GA	10 920	11 928	12 408	13 032	14 472	12 500	6.9%
1000	fitness-dep. $(1+(\lambda,\lambda))$ GA	9170	10 086	10 518	10 984	12 218	10 563	6.7%
1000	self-adj. $(1+(\lambda,\lambda))$ GA	9286	9966	10 350	10 808	11 994	10 414	6.6%
1000	$(2+1)$ GA	6668	7838	8691	9754	12 773	8911	16.8%
1500	$(1+1)$ EA	19 784	23 314	26 269	29 355	40 426	26 950	18.8%
1500	$(1+(4,4))$ GA	16 136	18 760	20 528	22 896	29 920	21 128	15.6%
1500	$(1+(8,8))$ GA	14 976	16 400	17 408	18 512	21 888	17 623	9.5%
1500	$(1+(12,12))$ GA	16 992	18 144	18 840	19 680	21 912	19 006	6.4%
1500	fitness-dep. $(1+(\lambda,\lambda))$ GA	14 272	15 328	15 926	16 546	17 956	15 963	5.6%
1500	self-adj. $(1+(\lambda,\lambda))$ GA	14 268	15 184	15 636	16 224	17 944	15 753	5.6%
1500	$(2+1)$ GA	10 682	12 624	13 753	15 311	19 255	14 140	15.6%
2000	$(1+1)$ EA	27 218	32 460	35 887	40 964	52 643	37 256	17.5%
2000	$(1+(4,4))$ GA	22 464	25 784	28 088	30 976	39 224	28 752	14.4%
2000	$(1+(8,8))$ GA	20 480	22 464	23 664	25 168	29 632	23 973	9.3%
2000	$(1+(12,12))$ GA	22 896	24 600	25 464	26 664	29 544	25 715	6.3%
2000	fitness-dep. $(1+(\lambda,\lambda))$ GA	19 332	20 656	21 340	22 036	23 398	21 336	4.7%
2000	self-adj. $(1+(\lambda,\lambda))$ GA	19 272	20 376	21 008	21 660	23 448	21 079	4.8%
2000	$(2+1)$ GA	14 977	17 443	19 025	21 141	27 473	19 603	15.7%
2500	$(1+1)$ EA	35 443	42 453	46 370	52 487	69 910	48 043	17.3%
2500	$(1+(4,4))$ GA	29 240	33 432	36 280	39 496	50 880	36 952	13.7%
2500	$(1+(8,8))$ GA	26 368	28 672	30 208	31 856	37 248	30 550	8.8%
2500	$(1+(12,12))$ GA	29 160	31 032	32 184	33 624	36 840	32 419	6.0%
2500	fitness-dep. $(1+(\lambda,\lambda))$ GA	24 440	25 830	26 680	27 494	29 294	26 703	4.5%
2500	self-adj. $(1+(\lambda,\lambda))$ GA	24 426	25 588	26 304	27 044	29 346	26 433	4.8%
2500	$(2+1)$ GA	19 357	22 491	24 590	27 130	35 422	25 188	15.2%
3000	$(1+1)$ EA	43 842	52 749	58 550	66 102	87 939	60 311	17.9%
3000	$(1+(4,4))$ GA	35 696	40 816	44 456	48 760	62 392	45 488	14.2%
3000	$(1+(8,8))$ GA	31 952	34 720	36 624	38 784	44 192	36 991	8.3%
3000	$(1+(12,12))$ GA	35 568	37 728	39 024	40 536	44 472	39 263	5.6%
3000	fitness-dep. $(1+(\lambda,\lambda))$ GA	29 652	31 230	32 082	33 002	35 022	32 150	4.0%
3000	self-adj. $(1+(\lambda,\lambda))$ GA	29 364	30 992	31 746	32 606	35 066	31 875	4.3%
3000	$(2+1)$ GA	23 692	27 570	30 273	33 136	42 772	30 838	15.1%
3500	$(1+1)$ EA	52 505	62 684	69 069	77 536	102 197	71 218	17.0%
3500	$(1+(4,4))$ GA	42 512	48 376	52 280	57 096	71 112	53 374	13.0%
3500	$(1+(8,8))$ GA	37 232	40 816	42 944	45 792	53 664	43 586	9.0%
3500	$(1+(12,12))$ GA	41 592	43 992	45 504	47 376	52 224	45 857	5.6%
3500	fitness-dep. $(1+(\lambda,\lambda))$ GA	34 826	36 610	37 534	38 616	40 794	37 621	3.9%
3500	self-adj. $(1+(\lambda,\lambda))$ GA	34 662	36 198	37 106	37 938	40 314	37 164	3.8%
3500	$(2+1)$ GA	28 623	32 721	35 730	39 727	50 031	36 713	14.4%
4000	$(1+1)$ EA	60 290	72 714	80 560	89 588	116 824	82 434	16.8%
4000	$(1+(4,4))$ GA	49 456	56 528	60 784	66 888	81 480	62 232	13.0%
4000	$(1+(8,8))$ GA	43 392	47 360	49 552	52 640	60 976	50 321	8.5%
4000	$(1+(12,12))$ GA	48 264	50 712	52 608	54 672	61 032	53 018	5.9%
4000	fitness-dep. $(1+(\lambda,\lambda))$ GA	40 128	41 938	42 956	44 036	46 420	43 030	3.6%
4000	self-adj. $(1+(\lambda,\lambda))$ GA	39 758	41 508	42 410	43 608	46 676	42 612	4.0%
4000	$(2+1)$ GA	33 256	37 926	41 151	45 512	58 675	42 279	14.5%
4500	$(1+1)$ EA	70 279	82 890	91 523	101 253	137 282	93 920	16.8%
4500	$(1+(4,4))$ GA	56 816	64 456	69 856	76 240	94 848	71 218	13.2%
4500	$(1+(8,8))$ GA	49 488	53 344	56 224	59 520	68 352	56 816	8.3%
4500	$(1+(12,12))$ GA	54 456	57 336	59 304	61 704	68 520	59 777	5.7%
4500	fitness-dep. $(1+(\lambda,\lambda))$ GA	45 324	47 352	48 524	49 620	51 906	48 521	3.4%
4500	self-adj. $(1+(\lambda,\lambda))$ GA	45 068	46 734	47 752	48 928	52 044	47 951	3.6%
4500	$(2+1)$ GA	37 832	43 407	47 467	51 782	65 401	48 393	14.2%
5000	$(1+1)$ EA	80 659	94 663	103 259	115 173	148 450	106 261	15.7%
5000	$(1+(4,4))$ GA	63 552	72 456	78 288	85 392	110 584	80 069	13.8%
5000	$(1+(8,8))$ GA	55 440	59 824	62 800	66 912	77 440	63 847	8.5%
5000	$(1+(12,12))$ GA	60 504	63 984	66 192	68 472	75 312	66 517	5.4%
5000	fitness-dep. $(1+(\lambda,\lambda))$ GA	50 568	52 730	53 876	55 068	57 538	53 923	3.2%
5000	self-adj. $(1+(\lambda,\lambda))$ GA	50 096	52 074	53 140	54 330	57 790	53 297	3.4%
5000	$(2+1)$ GA	43 259	48 840	53 027	57 664	75 281	54 245	13.9%

A.2. Statistics for experimental results of Fig. 6 (runtimes for linear functions with random weights $w_i \in [1, 2]$)

n	Algorithm	Percentile					Mean	StdDev/Mean
		2	25	50	75	98		
500	$(1+1)$ EA	4919	6300	7321	8403	12096	7562	23.7%
500	$(2+1)$ GA	4892	6252	7135	8311	11 941	7457	23.6%
500	$(1+(8,8))$ GA	4480	5040	5424	5824	7104	5501	11.5%
500	self-adj. $(1+(\lambda,\lambda))$ GA	4640	5190	5492	5874	7354	5596	11.2%
1000	$(1+1)$ EA	11 744	14 388	16 196	18 456	24 500	16 694	19.4%
1000	$(2+1)$ GA	11 691	14 371	16 009	18 688	25 842	16 809	20.5%
1000	$(1+(8,8))$ GA	9648	10 736	11 376	12 224	14 416	11 567	10.2%
1000	self-adj. $(1+(\lambda,\lambda))$ GA	9830	10 772	11 346	12 200	15 978	11 676	12.0%
1500	$(1+1)$ EA	19 052	23 086	25 776	29 403	40 283	26 800	20.2%
1500	$(2+1)$ GA	19 110	23 284	26 165	29 588	40 058	26 958	18.9%
1500	$(1+(8,8))$ GA	15 184	16 768	17 616	18 768	22 848	17 927	10.0%
1500	self-adj. $(1+(\lambda,\lambda))$ GA	15 342	16 782	17 932	19 540	26 214	18 553	13.9%
2000	$(1+1)$ EA	26 759	32 361	36 265	41 128	57 350	37 304	18.9%
2000	$(2+1)$ GA	27 379	32 523	36 382	40 938	56 453	37 438	18.3%
2000	$(1+(8,8))$ GA	20 800	22 704	23 936	25 424	29 568	24 246	9.0%
2000	self-adj. $(1+(\lambda,\lambda))$ GA	20 790	22 720	24 200	26 934	35 338	25 243	14.4%
2500	$(1+1)$ EA	36 308	42 541	47 488	53 306	72 234	48 803	17.4%
2500	$(2+1)$ GA	35 348	42 105	47 079	52 983	70 734	48 256	18.1%
2500	$(1+(8,8))$ GA	26 592	28 880	30 432	32 416	38 416	30 842	9.1%
2500	self-adj. $(1+(\lambda,\lambda))$ GA	26 256	29 096	31 276	34 832	47 526	32 665	15.8%
3000	$(1+1)$ EA	43 941	51 965	58 186	65 236	84 757	59 706	17.2%
3000	$(2+1)$ GA	43 278	52 562	57 826	64 593	86 755	59 506	17.5%
3000	$(1+(8,8))$ GA	32 432	35 216	37 136	39 248	45 936	37 576	8.6%
3000	self-adj. $(1+(\lambda,\lambda))$ GA	32 374	35 740	38 938	43 464	55 436	40 284	14.7%
3500	$(1+1)$ EA	53 153	62 182	69 645	78 432	104 304	71 497	17.4%
3500	$(2+1)$ GA	53 032	62 958	69 329	77 579	104 063	71 608	17.2%
3500	$(1+(8,8))$ GA	38 304	41 568	43 680	46 352	54 592	44 363	9.0%
3500	self-adj. $(1+(\lambda,\lambda))$ GA	38 240	42 390	46 180	51 664	72 530	48 158	16.9%
4000	$(1+1)$ EA	62 022	72 789	80 026	89 070	118 043	82 256	16.4%
4000	$(2+1)$ GA	61 755	73 379	80 548	89 153	117 184	82 540	16.2%
4000	$(1+(8,8))$ GA	44 400	48 048	50 752	53 808	62 688	51 351	8.9%
4000	self-adj. $(1+(\lambda,\lambda))$ GA	44 180	49 272	53 954	61 036	79 788	56 202	16.5%
4500	$(1+1)$ EA	70 170	82 545	91 141	102 239	133 060	94 019	16.9%
4500	$(2+1)$ GA	70 209	83 486	91 954	102 497	135 558	94 536	16.8%
4500	$(1+(8,8))$ GA	50 016	54 400	57 184	60 640	71 536	57 978	9.0%
4500	self-adj. $(1+(\lambda,\lambda))$ GA	50 262	56 994	62 986	70 360	94 408	64 997	16.6%
5000	$(1+1)$ EA	80 319	95 001	104 421	116 720	154 306	107 268	16.2%
5000	$(2+1)$ GA	79 558	92 935	102 760	115 637	149 748	105 590	16.3%
5000	$(1+(8,8))$ GA	56 144	60 784	64 000	67 920	78 288	64 834	8.8%
5000	self-adj. $(1+(\lambda,\lambda))$ GA	56 772	63 826	70 616	79 352	105 718	73 020	16.7%

A.3. Statistics for experimental results of Fig. 7 (runtimes for royal road functions of block size five, RR_5)

n	Algorithm	Percentile					Mean	StdDev/Mean
		2	25	50	75	98		
200	self-adj.	39 116	38 344	29 824	21 764	50 598	33,841	27.5%
200	$(1+1)$ EA	7549	10 825	13 379	16 900	27 742	14,461	34.8%
200	$(2+1)$ GA	6654	9398	11 288	14 029	22 070	12 064	31.1%
200	$(1+(8,8))$ GA	6960	9376	10 976	12 816	18 400	11 348	24.8%
200	modified self-adj.	6974	9404	10 990	12 662	18 680	11 354	24.7%
400	self-adj.	76 976	121 368	102 808	145 540	92 400	97,997	21.3%
400	$(1+1)$ EA	19 486	27 679	32 745	39 660	58 900	34,566	29.0%
400	$(2+1)$ GA	17 740	23 730	27 759	32 886	48 707	28,985	25.9%
400	$(1+(8,8))$ GA	16 512	20 464	23 136	25 824	34 560	23,636	19.0%
400	modified self-adj.	16 084	20 536	23 266	26 596	35 858	23,955	19.9%
600	self-adj.	137 376	163 778	223 252	191 248	190 248	174,231	16.4%
600	$(1+1)$ EA	32 821	45 235	53 337	65 003	98 796	56,600	28.6%
600	$(2+1)$ GA	30 103	39 830	45 591	53 556	76 298	47,429	23.3%
600	$(1+(8,8))$ GA	26 192	32 784	36 208	40 560	53 728	37,141	17.8%
600	modified self-adj.	26 940	32 612	36 386	40 696	52 626	37,225	17.1%

(*continued*)

n	Algorithm	Percentile					Mean	StdDev/ Mean
		2	25	50	75	98		
800	self-adj.	250 512	210 674	329 532	255 710	221 372	267,545	14.7%
800	$(1+1)$ EA	50 739	66 262	77 128	92 248	132 898	81 415	26.3%
800	$(2+1)$ GA	42 445	55 856	65 311	75 330	103 993	67 370	23.0%
800	$(1+(8,8))$ GA	36 960	44 816	49 536	55 008	71 040	50 577	16.1%
800	modified self-adj.	36 332	43 588	47 760	53 210	70 878	49 194	16.5%
1000	self-adj.	265 894	416 770	360 584	356 522	382 530	369,405	12.9%
1000	$(1+1)$ EA	144 710	124 298	159 904	134 603	122 441	105,318	25.3%
1000	$(2+1)$ GA	58 778	75 349	85 911	97 620	138 769	88 642	21.6%
1000	$(1+(8,8))$ GA	48 608	57 536	62 784	68 848	89 664	64 147	14.9%
1000	modified self-adj.	48 070	56 044	61 516	67 596	85 514	62 719	15.2%

References

[1] Peyman Afshani, Manindra Agrawal, Benjamin Doerr, Carola Doerr, Kasper Green Larsen, Kurt Mehlhorn, The query complexity of finding a hidden permutation, in: Space-Efficient Data Structures, Streams, and Algorithms – Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday, in: Lecture Notes in Computer Science, vol. 8066, Springer, 2013, pp. 1–11.
[2] Gautham Anil, R. Paul Wiegand, Black-box search by elimination of fitness functions, in: Proc. of the 10th ACM Workshop on Foundations of Genetic Algorithms, FOGA'09, ACM, 2009, pp. 67–78.
[3] Jaroslaw Arabas, Zbigniew Michalewicz, Jan Mulawka, GAVaPS – a genetic algorithm with varying population size, in: Proceedings of the First IEEE Conference on Evolutionary Computation, CEC'94, IEEE, 1994, pp. 73–78.
[4] Anne Auger, Benchmarking the $(1+1)$ evolution strategy with one-fifth success rule on the BBOB-2009 function testbed, in: Proc. of the 9th Annual Genetic and Evolutionary Computation Conference (GECCO'09) (Companion), ACM, 2009, pp. 2447–2452.
[5] Anne Auger, Nikolaus Hansen, Linear convergence on positively homogeneous functions of a comparison based step-size adaptive randomized search: the $(1+1)$ ES with generalized one-fifth success rule, CoRR, abs/1310.8397. Available online at http://arxiv.org/abs/1310.8397.
[6] Thomas Bäck, Optimal mutation rates in genetic search, in: Proceedings of the 5th International Conference on Genetic Algorithms, ICGA'93, Morgan Kaufmann, 1993, pp. 2–8.
[7] Hans-Georg Beyer, Toward a theory of evolution strategies: on the benefit of sex – the $(\mu/\mu,\lambda)$-theory, Evol. Comput. 3 (1995) 81–111.
[8] Hans-Georg Beyer, The Theory of Evolution Strategies, Springer Verlag, Heidelberg, 2002.
[9] Süntje Böttcher, Benjamin Doerr, Frank Neumann, Optimal fixed and adaptive mutation rates for the LeadingOnes problem, in: Proc. of the 11th International Conference on Parallel Problem Solving from Nature, PPSN'10, in: Lecture Notes in Computer Science, vol. 6238, Springer, 2010, pp. 1–10.
[10] Luc Devroye, The compound random search, Ph.D. dissertation, Purdue Univ., West Lafayette, IN, 1972.
[11] Benjamin Doerr, Analyzing randomized search heuristics: tools from probability theory, in: Anne Auger, Benjamin Doerr (Eds.), Theory of Randomized Search Heuristics, World Scientific Publishing, 2011, pp. 1–20.
[12] Benjamin Doerr, Carola Doerr, Franziska Ebel, Lessons from the black-box: fast crossover-based genetic algorithms, in: Proc. of the Annual Genetic and Evolutionary Computation Conference, GECCO'13, ACM, 2013, pp. 781–788.
[13] Benjamin Doerr, Carola Winzen, Playing Mastermind with constant-size memory, Theory Comput. Syst. 55 (2014) 658–684.
[14] Benjamin Doerr, Carola Winzen, Ranking-based black-box complexity, Algorithmica 68 (2014) 571–609.
[15] Benjamin Doerr, Carola Winzen, Reducing the arity in unbiased black-box complexity, Theoret. Comput. Sci. 545 (2014) 108–121.
[16] Benjamin Doerr, Daniel Johannsen, Timo Kötzing, Per Kristian Lehre, Markus Wagner, Carola Winzen, Faster black-box algorithms through higher arity operators, in: Proc. of the 11th ACM Workshop on Foundations of Genetic Algorithms, FOGA'11, ACM, 2011, pp. 163–172.
[17] Benjamin Doerr, Timo Kötzing, Johannes Lengler, Carola Winzen, Black-box complexities of combinatorial problems, Theoret. Comput. Sci. 471 (2013) 84–106.
[18] Stefan Droste, Thomas Jansen, Ingo Wegener, On the analysis of the $(1+1)$ evolutionary algorithm, Theoret. Comput. Sci. 276 (2002) 51–81.
[19] Stefan Droste, Thomas Jansen, Ingo Wegener, Upper and lower bounds for randomized search heuristics in black-box optimization, Theory Comput. Syst. 39 (2006) 525–544.
[20] Stefan Droste, Thomas Jansen, Karsten Tinnefeld, Ingo Wegener, A new framework for the valuation of algorithms for black-box optimization, in: Proc. of the 7th Workshop on Foundations of Genetic Algorithms, FOGA'03, Morgan Kaufmann, 2003, pp. 253–270.
[21] Agoston Endre Eiben, Jim E. Smith, Introduction to Evolutionary Computing, Springer Verlag, Heidelberg, 2003.
[22] Agoston Endre Eiben, Robert Hinterding, Zbigniew Michalewicz, Parameter control in evolutionary algorithms, IEEE Trans. Evol. Comput. 3 (1999) 124–141.
[23] Paul Erdős, Alfréd Rényi, On two problems of information theory, Magy. Tud. Akad. Mat. Kut. Intéz. Közl. 8 (1963) 229–243.
[24] Nikolaus Hansen, Andreas Gawelczyk, Andreas Ostermeier, Sizing the population with respect to the local progress in $(1,\lambda)$-evolution strategies – a theoretical analysis, in: Proc. of the IEEE Congress on Evolutionary Computation, CEC'95, IEEE, 1995, pp. 80–85.
[25] Edda Happ, Daniel Johannsen, Christian Klein, Frank Neumann, Rigorous analyses of fitness-proportional selection for optimizing linear functions, in: Proc. of the Annual Genetic and Evolutionary Computation Conference, GECCO'08, ACM, 2008, pp. 953–960.
[26] Jens Jägersküpper, Rigorous runtime analysis of the $(1+1)$ ES: 1/5-rule and ellipsoidal fitness landscapes, in: Proc. of the 8th ACM Workshop on Foundations of Genetic Algorithms, FOGA'05, in: Lecture Notes in Computer Science, vol. 3469, Springer, 2005, pp. 260–281.
[27] Jens Jägersküpper, Tobias Storch, When the plus strategy outperforms the comma strategy and when not, in: Proc. of IEEE Symposium on Foundations of Computational Intelligence, FOCI'07, IEEE, 2007, pp. 25–32.
[28] Thomas Jansen, Kenneth A. De Jong, Ingo Wegener, On the choice of the offspring population size in evolutionary algorithms, Evol. Comput. 13 (2005) 413–440.
[29] Timo Kötzing, Dirk Sudholt, Madeleine Theile, How crossover helps in pseudo-Boolean optimization, in: Proc. of the 13th Annual Genetic and Evolutionary Computation Conference, GECCO'11, ACM, 2011, pp. 989–996.
[30] Jörg Lässig, Dirk Sudholt, Adaptive population models for offspring populations and parallel evolutionary algorithms, in: Proc. of the 11th ACM Workshop on Foundations of Genetic Algorithms, FOGA'11, ACM, 2011, pp. 181–192.
[31] Per Kristian Lehre, Carsten Witt, Black-box search by unbiased variation, in: Proc. of the 12th Annual Genetic and Evolutionary Computation Conference, GECCO'10, ACM, 2010, pp. 1441–1448.
[32] Per Kristian Lehre, Carsten Witt, Black-box search by unbiased variation, Algorithmica 64 (2012) 623–642.
[33] Melanie Mitchell, Stephanie Forrest, John H. Holland, The royal road for genetic algorithms: fitness landscapes and GA performance, in: Proc. of the First European Conference on Artificial Life, MIT Press, 1992, pp. 245–254.

[34] Alberto Moraglio, Riccardo Poli, Topological interpretation of crossover, in: Proc. of the Annual Genetic and Evolutionary Computation Conference, GECCO'04, ACM, 2004, pp. 1377–1388.
[35] Pietro Simone Oliveto, Per Kristian Lehre, Frank Neumann, Theoretical analysis of rank-based mutation – combining exploration and exploitation, in: Proc. of the IEEE Congress on Evolutionary Computation, CEC'09, 2009, pp. 1455–1462.
[36] Pietro Simone Oliveto, Carsten Witt, Improved runtime analysis of the simple genetic algorithm, in: Proc. of the Annual Genetic and Evolutionary Computation Conference, GECCO'13, ACM, 2013, pp. 1621–1628.
[37] Ingo Rechenberg, Evolutionsstrategie, Friedrich Fromman Verlag (Günther Holzboog KG), Stuttgart, 1973.
[38] Jonathan E. Rowe, Dirk Sudholt, The choice of the offspring population size in the $(1, \lambda)$ EA, in: Proc. of the Annual Genetic and Evolutionary Computation Conference, GECCO'12, ACM, 2012, pp. 1349–1356.
[39] Michael A. Schumer, Kenneth Steiglitz, Adaptive step size random search, IEEE Trans. Automat. Control 13 (1968) 270–276.
[40] William M. Spears, Kenneth A. De Jong, An analysis of multi-point crossover, in: Proc. of the 1st Workshop on Foundations of Genetic Algorithms, FOGA'90, Morgan Kaufmann, 1990, pp. 301–315.
[41] Tobias Storch, On the choice of the parent population size, Evol. Comput. 16 (2008) 557–578.
[42] Dirk Sudholt, Crossover speeds up building-block assembly, in: Proc. of the 14th Annual Genetic and Evolutionary Computation Conference, GECCO'12, ACM, 2012, pp. 689–702.
[43] Dirk Sudholt, A new method for lower bounds on the running time of evolutionary algorithms, IEEE Trans. Evol. Comput. 17 (2013) 418–435.
[44] Gilbert Syswerda, Uniform crossover in genetic algorithms, in: Proc. of the 3rd International Conference on Genetic Algorithms, ICGA'89, Morgan Kaufmann Publishers Inc., 1989, pp. 2–9.
[45] Ingo Wegener, Methods for the analysis of evolutionary algorithms on pseudo-Boolean functions, in: S. Sarker, X. Yao, M. Mohammadian (Eds.), Evolutionary Optimization, Kluwer, Dordrecht, 2002, pp. 349–369.
[46] Carsten Witt, Runtime analysis of the $(\mu + 1)$ EA on simple pseudo-Boolean functions, Evol. Comput. 14 (2006) 65–86.
[47] Carsten Witt, Tight bounds on the optimization time of a randomized search heuristic on linear functions, Comb. Probab. Comput. 22 (2013) 294–318.
[48] Christine Zarges, Rigorous runtime analysis of inversely fitness proportional mutation rates, in: Proc. of the 10th International Conference on Parallel Problem Solving from Nature, PPSN'08, in: Lecture Notes in Computer Science, vol. 5199, Springer, 2008, pp. 112–122.
[49] Christine Zarges, On the utility of the population size for inversely fitness proportional mutation rates, in: Proc. of the 10th ACM Workshop on Foundations of Genetic Algorithms, FOGA'09, ACM, 2009, pp. 39–46.