Natural Computing (2024) 23:115–129
https://doi.org/10.1007/s11047-022-09915-0

Large population sizes and crossover help in dynamic environments

Johannes Lengler[1] · **Jonas Meier**[1]

Accepted: 27 July 2022 / Published online: 11 August 2022
© The Author(s) 2022

Abstract

Dynamic linear functions on the boolean hypercube are functions which assign to each bit a positive weight, but the weights change over time. Throughout optimization, these functions maintain the same global optimum, and never have defecting local optima. Nevertheless, it was recently shown [Lengler, Schaller, FOCI 2019] that the $(1+1)$-Evolutionary Algorithm needs exponential time to find or approximate the optimum for some algorithm configurations. In this experimental paper, we study the effect of larger population sizes for *dynamic binval*, the extreme form of dynamic linear functions. We find that moderately increased population sizes extend the range of efficient algorithm configurations, and that crossover boosts this positive effect substantially. Remarkably, similar to the static setting of monotone functions in [Lengler, Zou, FOGA 2019], the hardest region of optimization for $(\mu+1)$-EA is not close the optimum, but far away from it. In contrast, for the $(\mu+1)$-GA, the region around the optimum is the hardest region in all studied cases.Kindly check and confirm the inserted city name is correctly identified.Correct.

Keywords Evolutionary algorithms · Genetic algorithms · Crossover · Population size · Dynamic linear functions · Dynamic binval · Degenerate population drift · Simulations

Mathematics Subject Classification 68Q07

1 Introduction

The $(\mu+1)$ Evolutionary Algorithm and the $(\mu+1)$ Genetic Algorithm, $(\mu+1)$-EA and $(\mu+1)$-GA for short, are heuristic algorithms that aim to optimize an objective or fitness function $f:\{0,1\}^n \rightarrow \mathbb{R}$. Both maintain a population of μ search points, and in each round they create an offspring from the population and discard one of the $\mu+1$ search points, based on their objective values. They differ in how the offspring is created: the $(\mu+1)$-EA chooses a parent from the population and *mutates* it, the $(\mu+1)$-GA may use *crossover* of two parent solutions instead of mutation, see Algorithm 1 in Sect. 2 for the precise definition.[1]

Two classical theoretical questions for these algorithms have always been:

- What is the effect of the population size? In which optimization landscapes and regimes are larger (smaller) populations beneficial?
- In which situations does crossover improve performance?

Although these questions have always been central for studies of the $(\mu+1)$-EA and the $(\mu+1)$-GA , there is still vivid ongoing research on these questions, see (Antipov et al. 2018; Dang et al. 2017; Doerr et al. 2012; Jansen and Wegener 2002; Pinto and Doerr 2018; Sudholt 2017; Witt 2006, 2013) for a selection of theoretical works. (Also, the book chapter (Sudholt 2020) treats related

✉ Johannes Lengler
johannes.lengler@inf.ethz.ch

[1] Department of Computer Science, ETH Zürich, Zürich, Switzerland

[1] This is the version of the $(\mu+1)$-EA and $(\mu+1)$-GA that we use in this paper, but there are also alternative versions and terminologies. Firstly, the term *Evolutionary Algorithms* is an umbrella term which is not restricted to mutation-based algorithms in general. However, we follow the convention common in the theory community that the $(\mu+1)$-EA only uses mutation. For the $(\mu+1)$-GA , there are many different versions; for example, alternative versions of the algorithm may in each generation use crossover *and* mutation, or may use *only* mutation.

topics.) More generally, the research question is: which algorithm configurations perform well in which optimization landscapes? Such landscapes are given by a specific benchmark functions or by a class of functions.

Recently, a new type of dynamic landscapes was introduced by Lengler and Schaller (Lengler and Schaller 2018). It was called *noisy linear functions* in Lengler and Schaller (2018), but we prefer the term *dynamic linear functions*. In this setting, the objective function is of the form $f:\{0,1\}^n \rightarrow \mathbb{R}; f(x)=\sum_{i=1}^n W_i x_i$ with positive coefficients $W_i > 0$. However, the twist is that the weights W_i are redrawn for each generation. I.e., we have a distribution $\mathcal{D}$, and for the tth generation we draw i.i.d. weights $W_i^{(t)}$ from that distribution, which define a function $f^{(t)}$. Then the $\mu+1$ competing individuals are compared with respect to the fitness function $f^{(t)}$.

To motivate this setting, let us give a grotesquely oversimplified example. Imagine a chess engine has 100 bits as parameters. Each bit switches on/off a database tailored to one specific opening ($1=$ access, $0=$ no access), and this improves performance massively for this opening. E.g., the first bit determines whether the engine plays well in a French Opening, but has no influence whatsoever on the performance of an Italian Opening (the database is ignored since it does not produce matches to that situation). Let us go even further and assume that the engine will always win an opening if the corresponding database is active, and always lose with inactive database. Then we have removed even the slightest ambiguity, and this setting has an obvious optimal solution, which is the all-one string (activate all databases). This situation may seem completely trivial, but crucially, *it is not solved by some standard optimization algorithms*.

To complete the analogy, assume that the engine is trained by playing against different players, where player t has probability $W_i^{(t)}$ to choose the ith opening. Then the reward is precisely the dynamic linear function introduced above, and it was shown in Lengler and Schaller (2018) that *the* $(1+1)$-EA *needs exponential time to approximate the optimum within a constant factor* when configured with bad parameters. These bad parameter settings look quite innocent. With standard bit mutation (i.e., for mutation we flip each bit independently with probability $p=c/n$), any choice $c>c_0\approx 1.59$ leads asymptotically to an exponential time for finding or approximating the optimum, if the distribution $\mathcal{D}$ is too skewed. On the other hand, for any $c<c_0$ the $(1+1)$-EA finds the optimum in time $O(n\log n)$ for any $\mathcal{D}$. This lack of stability motivates our paper: we ask whether larger population sizes and/or crossover can push the threshold c_0 of failure.

Optimization of dynamic functions may occur in various contexts. The chess engine with varying opponents is one such example. Similar examples arise in the context of co-evolution, e.g., a chess-engine trained against itself, or a team of DOTA agents in which some abilities of an agent (good aim, good exploration strategy, good path planning, ...) are always helpful (positive weight), but may be more or less important depending on her current co-agents. A rather different example is planning the timetable of a transportation company: to be efficient in the exploration phase of the optimization algorithm, schedules may be compared only for some partial data, and not for the whole data set. Similarly, consider an optimization process in which the function evaluation involves an offline test, as in drug development or robotic training. Then each test may involve subtly varying outer conditions (e.g., different temperatures or humidity, different lighting), which effectively gives a slightly different fitness function for each test.

Our Results in a Nutshell Instead of the full range of dynamic linear functions as in Lengler and Schaller (2018), we only study the limiting case of these functions, which we call *dynamic binval*. We perform experiments to study the performance of the $(\mu+1)$-EA and the $(\mu+1)$-GA for small values of μ. Similarly as for the $(1+1)$-EA , we find that for each algorithm there is a threshold c_0 such that the algorithm is efficient for every mutation parameter $c<c_0$, and inefficient for $c>c_0$. This threshold c_0 is our main object of study, and we investigate how it depends on the algorithmic choices. We find that an increased population size helps to push c_0, but that the benefits are much larger when crossover is used. As a baseline, we recover the theoretical result from Lengler and Schaller (2018) that for the $(1+1)$-EA the threshold is at $c_0\approx 1.59$, though experimentally, for $n=3000$ it seems closer to 1.7. Thus $n=3000$ seems sufficient to observe the theoretical results that have been proven for $n\rightarrow\infty$, and all following experimental results were also obtained for $n=3000$. For the $(2+1)$-EA the threshold increases to $c_0\approx 2.2$, and further to $c_0\approx 3.1$ for the $(2+1)$-GA . If we explicitly forbid that the two parents in crossover are identical then the threshold even shifts to $c_0\approx 4.2$. We call the resulting algorithm $(2+1)$-GA-NoCopy . For larger population sizes we get a threshold of $c_0\approx 2.6$ for the $(3+1)$-EA , $c_0\approx 3.4$ for the $(5+1)$-EA , $c_0\approx 6.1$ for the $(3+1)$-GA , and $c_0>20$ for the $(5+1)$-GA . Thus we find that moderately increased population sizes extend the range of efficient algorithm configurations, and that crossover boosts this positive effect substantially.

The theoretical results for the $(1+1)$-EA predict that the runtime jumps from quasi-linear to exponential. Indeed, we can experimentally confirm huge jumps in the runtime even for slight changes of the mutation parameter c. For example, we obtain a significant p-value for the a posteriori hypothesis that the $(2+1)$-EA with $c=2.5$ is more than 60 times slower than the $(2+1)$-EA with $c=2.0$. In fact, this is a highly conservative estimate since we needed to

cut off the runs for $c = 2.5$. We systematically list these factors in our result sections.

To get a better understanding of the hardness of the optimization landscape, we compute the *drift of degenerate populations*, inspired by Lengler (2019). We call a population *degenerate* if it consists entirely of multiple copies of the same individual. If X_i is the number of zero-bits in the ith degenerate population, then we estimate the drift $\mathbb{E}[X_i - X_{i+1} \mid X_i = y]$ by Monte-Carlo simulations. Moreover, for y close to 0 we derive precise asymptotic formulas for the degenerate population drift for the $(2+1)$-EA and the $(2+1)$-GA . In Lengler (2019) the degenerate population drift was studied theoretically for the $(\mu+1)$-EA on monotone functions, which is a related, but not identical setup (see below). Still, parts of the analysis carry over: if the population drift is negative for some y then the runtime is exponential, while it is $O(n \log n)$ if the population drift is positive everywhere.As per the information provided by the publisher, Figure [5] will be black and white in print; hence, please confirm whether we can add "colour figure online" to the caption Yes, please do that.

Perhaps surprisingly, the $(\mu+1)$-EA and the $(\mu+1)$-GA are not just quantitatively different, but we also find a strong qualitative difference in the hardness landscape. For the $(\mu+1)$-GA , the "hardest" part of the optimization process is close to optimum, in all cases that we have experimentally explored. Formally, we found that if the degenerate population drift is negative *somewhere*, then it is also negative close to the optimum. For the $(\mu+1)$-EA , we found the opposite: the degenerate population drift can be negative for some intermediate ranges, although it is positive around the optimum. This implies that the hard part of optimization (taking exponential time) is getting in the vicinity of the optimum. But once the algorithm is somewhat near the optimum, it will efficiently finish optimization. This behavior is rather counter-intuitive, since common wisdom says that optimization gets harder close to the optimum.[2] Notably, a similar phenomenon has recently been proven for certain monotone functions by Lengler (2019); Lengler and Zou (2019), see below.

Related Work The only previous work on dynamic linear functions is by Lengler and Schaller (2018). As mentioned before, they proved that for every $c > c_0 \approx 1.59$ there is $\varepsilon > 0$ and a distribution $\mathcal{D}$ such that the $(1+1)$-EA with mutation rate c/n needs exponential time to find a search point with at least $(1-\varepsilon)n$ one-bits for dynamic linear functions with weight distribution $\mathcal{D}$. For $c < c_0$ the optimization time is $O(n \log n)$ for all distributions $\mathcal{D}$. Moreover, for any $c > c_0$, they gave a complete characterization of all distributions for which the $(1+1)$-EA with mutation rate c/n is efficient/inefficient.

Other previous work on population-based algorithms in dynamic environments had a slightly different flavor. One goal was to understand how well various algorithms can track a slowly moving optimum in a dynamic environment, e.g. Dang et al. (2017), Lissovoi and Witt (2016). Another was to show that dynamic environments can prevent algorithms from falling into traps (Rohlfshagen et al. 2009). For more details on these topics we refer to the survey by Sudholt (Sudholt 2020, Section 8.5), with emphasis on the role of diversity.

An important strand of work that is similar to our setting in spirit, though not in detail, is the study of *monotone functions*. A function is *monotone* if for every bit-string, flipping any zero-bit into a one-bit increases the fitness. Doerr et al. (2013) and Lengler and Steger (2018) showed that there are monotone functions for which the $(1+1)$-EA needs exponential time to find or approximate the optimum if the mutation parameter c is too large ($c > c_0 \approx 2.1$), while it is efficient for all monotone functions if $c \leq 1 + \varepsilon$ for some small $\varepsilon > 0$ (Lengler et al. 2019). The construction of hard (static) instances from Lengler and Steger (2018) was named HotTopic in Lengler (2019), and it resembles dynamic linear functions: the HotTopic function is locally given by linear functions with certain positive weights, but as the algorithm proceeds from one part of the search space ("level") to the next, the weights change. This analogy inspired the introduction of dynamic linear functions in Lengler and Schaller (2018). Other than the rather technical HotTopic functions, dynamic linear functions have the advantage that they are extremely simple and arguably more natural.

For HotTopic functions, there is a plethora of results. In Lengler (2019), the dichotomy between exponential and quasi-linear time from the $(1+1)$-EA was extended to a large number of other algorithms, including the $(1+\lambda)$-EA , the $(\mu+1)$-EA , their so-called "fast" counterparts, and the $(1+(\lambda,\lambda))$-GA . On the other hand, it was shown that the $(\mu+1)$-GA is always efficient for HotTopic functions if the population size is sufficiently large, while for the $(\mu+1)$-EA the population size does not change the threshold c_0 at all. Notably, for the population-based algorithms $(\mu+1)$-EA and $(\mu+1)$-GA , the efficiency result was only obtained for parameterizations of the HotTopic functions in which the weight changes occur close to the optimum. This seemed like a technical detail at first, but in an extremely surprising result, Lengler and Zou (2019) showed that this detail was hiding an unexpected core: if the weights are changed far away from the optimum, then increasing the population size has a devastating effect on the performance of the $(\mu+1)$-EA . For any $c > 0$ (also values much smaller than 1), there is a μ_0 such that the $(\mu+1)$-EA with $\mu \geq \mu_0$ and mutation rate c/n needs exponential time on some monotone functions.

[2] Also known as Pareto principle.

Together with Lengler (2019), this shows three things for monotone functions:

1. For optimization close to the optimum, the population size has no strong impact on the performance of the $(\mu+1)$-EA .
2. Close to the optimum, the $(\mu+1)$-GA outperforms the $(\mu+1)$-EA massively (quasi-linear instead of exponential) if the population size is large enough. It can cope with any constant mutation parameter c.
3. Far away from the optimum, a larger population size decreases the performance of $(\mu+1)$-EA massively. There is no safe choice of c if μ is too large.

It would be extremely interesting to understand the $(\mu+1)$-GA far away from the optimum. Unfortunately, such results are unknown. Theoretical analysis is hard (though perhaps not impossible), and function evaluations of HotTopic are extremely expensive, so experiments are only possible for very small problem sizes. Our paper can be seen as the first work which studies the behavior of the $(\mu+1)$-GA in a related, though not identical setting.

We conclude this section with a word of caution. HotTopic functions and dynamic linear functions are similar in spirit, but not in actual detail. For example, the analysis of the $(\mu+1)$-GA in Lengler (2019) or of the $(\mu+1)$-EA in Lengler and Zou (2019) rely heavily on the fact that their weights are locally stable in HotTopic functions. Thus it is unclear how far the analogy carries. Some of our experimental findings for the $(\mu+1)$-EA for dynamic linear functions differ from the theoretical (asymptotic) results for HotTopic in Lengler (2019), Lengler and Zou (2019). For us, a larger μ is beneficial, as it shifts the theshold c_0 to the right. For HotTopic functions, it does not shift the threshold at all if the algorithm operates close to the optimum, and it shifts the threshold *to the left* (i.e., makes things worse) far away from the optimum. This could either be because the theoretical effects only kick in for very large μ, or because HotTopic and dynamic linear functions are genuinely different. On the other hand, both settings agree in the surprising effect that the hardest part for the algorithm is not close to the optimum, but rather far away from it.

2 Preliminaries

2.1 The algorithms

All our considered algorithms maintain a population P of search points of size μ. In each round (or *generation*), they create an offspring from the population, and from the $\mu+1$ search points they remove the one with lowest fitness, breaking ties randomly. They only differ in the offspring creation. The $(\mu+1)$-EA uses *standard bit mutation*: a random parent is picked from the population, and each bit in the parent is flipped with probability c/n. The genetic algorithms flip a coin in each round whether to use mutation (as above), or whether to use *bitwise uniform crossover*: for the latter, it picks two random parents from the population, and for each bit it randomly chooses the bit of either parent. For the $(\mu+1)$-GA , the two parents are chosen independently. For the $(\mu+1)$-GA-NoCopy , they are chosen without repetition. The parameters are thus the mutation parameter $c>0$, which we will assume to be independent of n, and the population size μ. In our theoretical (asymptotic) results, we will only consider $\mu=2$. The pseudocode description is given in Algorithm 1.

Algorithm 1: $(\mu+1)$-EA, $(\mu+1)$-GA, and $(\mu+1)$-GA-NoCopy with mutation parameter c for maximizing an unknown function $f:\{0,1\}^n\to\mathbb{R}$. P is a multiset, i.e., it may contain search points several times.

```
1  Initialize P with μ independent x ∈ {0,1}^n uniformly at random;
2  Optimization: for t = 1, 2, 3, ... do
3      Creation of Offspring: For GAs, flip a fair coin to do either mutation or
        crossover; for EA, always do mutation and no crossover.
4      if mutation then
5          Choose x ∈ P uniformly at random;
6          Create y by flipping each bit in x independently with probability c/n;
7      if crossover then
8          Choose x, x' ∈ X uniformly at random: independently for (μ+1)-GA;
            without repetition for (μ+1)-GA-NoCopy;
9          Create y by setting y_i to either x_i or x'_i, each with probability 1/2,
            independently for all bits;
10     Set P ← P ∪ {y};
11     Selection: Select z ∈ arg min{f(x) | x ∈ P} (break ties randomly);
12     P ← P \ {z};
```

2.2 Dynamic linear functions and the dynamic binval function

We have described the algorithms for optimizing a static fitness function f. However, throughout the paper, we will consider dynamic functions that change in every round. We denote the function in the tth iteration by $f^{(t)}$. That means that in the selection step (Line 11), we select the worst individual as $z \in \arg\min\{f^{(t)}(x) \mid x \in P^{(t)}\}$, where $P^{(t)}$ is the tth population. Crucially, we never mix different fitness functions, i.e., we never compare $f^{(t_1)}(x)$ with $f^{(t_2)}(x')$ for different $t_1 \neq t_2$. In other words, the fitness of all individuals changes in each round. Hence, this requires $\mu + 1$ function evaluations per generation. However, as the examples in the introduction indicate, the dynamic setting is more natural in situations in which the fitness can not be explicitly computed, but where only comparisons are available. Therefore, we define *the runtime as the number of generations* until the algorithm finds the optimum. This deviates from the more standard convention to count the number of function evaluations (essentially by a factor $\mu + 1$), but it makes the performance easier to compare with previous work on static linear functions. Also, note that the runtime equals the number of search points that are sampled, up to an additive $-(\mu - 1)$ from initialization.

The dynamic function we consider is *dynamic binval*. In order to explain and motivate this function, we first introduce a closely related type of dynamic functions: a *dynamic linear function* is described by a distribution $\mathcal{D}$ on $\mathbb{R}^+$. For the tth round, we draw n independent samples $W_1^{(t)}, \ldots, W_n^{(t)}$ and set

$$f^{(t)}(x) := \sum_{i=1}^{n} W_i^{(t)} \cdot x_i. \tag{1}$$

So $f^{(t)}(x)$ is a non-negative value as the weights are positive. Thus all $f^{(t)}$ share the same global optimum $1\ldots1$, have no other local optima, and they are monotone, i.e., flipping a zero-bit into a one-bit always increases the fitness.

For *dynamic binval*, DYNBV , in the tth round we draw a permutation $\pi_t : \{1..n\} \to \{1..n\}$ uniformly at random, and define

$$f^{(t)}(x) = \sum_{i=1}^{n} 2^i \cdot x_{\pi_t(i)}. \tag{2}$$

So, we randomly permute the bits of the string, and take the binary value of the permuted string. As for dynamic linear functions, all $f^{(t)}$ share the same global optimum $1\ldots1$, have no other local optima, and prefer one-bits to zero-bits.

We claim that in a certain sense, DYNBV is a limit case of dynamic linear functions in which the tail of the distribution $\mathcal{D}$ becomes infinitely heavy. Let us make this precise. Consider a noisy linear function f and two strings x, x' that differ in k bits. To ease notation, assume they differ in the first $k \geq 2$ bits. The *order statistics* $W_{(1)}^{(t)} \leq \ldots \leq W_{(k)}^{(t)}$ are obtained from $W_1^{(t)}, \ldots, W_k^{(t)}$ by sorting, i.e., the first order statistics $W_{(1)}^{(t)}$ is the smallest value among $W_1^{(t)}, \ldots, W_k^{(t)}$, and so on. If the distribution $\mathcal{D}$ is sufficiently skewed, then the probability

$$p_k := \mathbb{P}\left(W_{(k)}^{(t)} \sum_{i=1}^{k-1} W_{(i)}^{(t)}\right) \tag{3}$$

comes arbitrarily close to one. However, conditioned on the event in (3), comparison with respect to the dynamic linear function is *equivalent* to comparison with respect to DYNBV . If we compute the difference $f^{(t)}(x) - f^{(t)}(x') = \sum_{i=1}^{k} W_i^{(t)}(x_i - x_i')$, then the sign of this difference will only depend on which of the k bits has the highest weight, since this summand dominates the whole remaining sum. The position of the highest weight bit is uniformly at random, so effectively, conditioned on the event in (3), the dynamic linear function picks randomly one of the bits in which x and x' differ, and bases its comparison only on that bit. This is exactly what DYNBV also does.

In fact, this reasoning was implicitly used in (Lengler and Schaller 2018, Theorem 7). There, to construct a hard example for the $(1+1)$-EA with $c > c_0 \approx 1.59$, the authors used $\mathcal{D}$ as a Pareto distribution $\mathrm{Par}(\beta)$, showed that for this distribution $p_k \geq (k-1)^{-\beta}$, and observed that this term comes arbitrarily close to 1 as $\beta \to 0$. Then they picked β so close to zero that the difference was negligible. Thus, in effect, they used for their hard function that DYNBV can be arbitrarily well approximated by dynamic linear functions, and their proof implies that the $(1+1)$-EA also needs exponential time for DYNBV if $c > c_0$, and time $O(n \log n)$ for $c < c_0$. Note that in general one needs to be a bit careful when two limits $n \to \infty$ and $\beta \to \infty$ are involved. However, this is no problem for the $(1+1)$-EA since there are strong tail bounds for the number of bits in which x and x' differ Lengler and Schaller (2018). The same argument also extends to populations in situations in which the population tends to degenerate to copies of a single point, as it is the case for $(\mu + 1)$ algorithms if it is hard to find improvements (Lengler 2019).

2.3 Runtime simulations

Recall that we count the runtime as the number of generations until the optimum is sampled. We run the different algorithms to observe the distribution of the runtime. A run

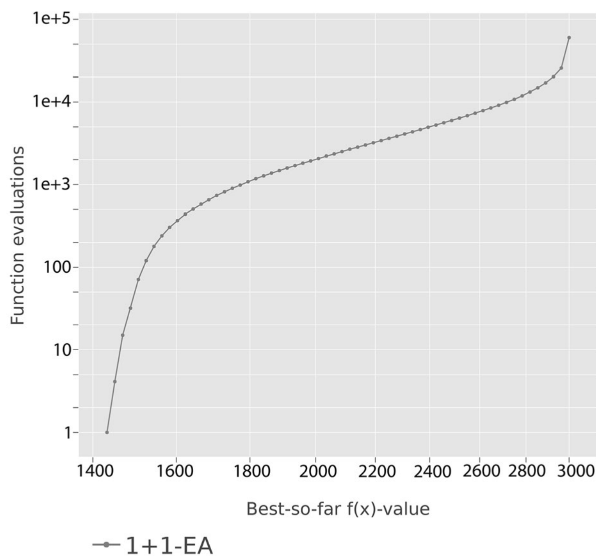

Fig. 1 Runtime of the $(1+1)$-EA on ONEMAX

terminates if either the optimum is found, or an upper limit of generations is reached. Unless otherwise noted, the upper limit is set to be $100e^c/c \cdot n \ln n$, which is 100 times larger than the expected runtime of the $(1+1)$-EA Witt (2013). The python code of our runtime simulation can be found in our GitHub repository (Lengler and Meier 2020). Unless otherwise noted, each data point is obtained by 30 independent runs.

To verify the correctness of our simulation we first measure mean and variance of the runtime of the (1+1)-EA with mutation parameter $c = 1$ on the function ONEMAX (the linear function where all weights are 1), and compare them with the highly accurate values derived in Hwang et al. (2018). We compute mean and variance of 3000 runs and find that our observed mean deviates 0.16% of the predicted mean, while the observed variance of is within 2.3% of the predicted variance.

To visualize runtimes, we use plots provided by the IOHprofiler (Doerr et al. 2018). As an example, the runtime of the $(1+1)$-EA with mutation parameter $c = 1.0$ on ONEMAX is visualized in Fig. 1. Note that time (i.e., number of generations) is displayed *on the y-axis*, while the x-axis corresponds to the number of one-bits. Thus, a steep part of the curve corresponds to slow progress, while a flat part of a curve corresponds to fast progress. Also, mind that the y-axis uses log scale.

Due to the exponential runtimes, we frequently encounter the problem that runs are terminated due to the iteration limit of $100e^c/c \cdot n \ln n$ generations. In this case we will plot two values, which are a lower and upper estimates for the expected runtime. The lower bound is the *mean runtime*, i.e., the average number of generations among the *successful* runs. By definition, this number never exceeds the upper limit. For the upper bound, we use the *expected runtime* (ERT) as defined by the IOH profiler. The ERT is calculated by drawing random runs from our pool of runs, until a successful run is drawn. Then the runtime of all drawn runs is added up, and the expectation of this process is defined as the ERT. This estimates the expected runtime if we start over the algorithm every time the iteration limit is reached. The ERT overestimates the runtime if the state at hitting the iteration limit is better than the starting state of a restart, which is the case in all benchmarks we consider. (It may not be the case for deceptive functions.) If *all* runs hit the iteration limit, then we define the mean runtime (our lower bound) to be the iteration limit, while the ERT (the upper bound) is infinity. A more detailed explanation of the ERT can be found in Doerr et al. (2018).

Comparison of Runtimes. We want to compare runtimes for different algorithms and values of c. We denote by R_c^{Alg} the random variable describing the runtime of Algorithm Alg for a specific c. Because our sample size is fairly small (10–30), we compare runtimes using the Wilcoxon-Mann-Whitney test. The Wilcoxon-Mann-Whitney test is a test of the null hypothesis that with probability at least 1/2 a randomly selected value from one runtime distribution will be at most (at least) a randomly selected value from a second runtime distribution. A small p-value would then indicate that the runtime of one algorithm, treated as random variable, is larger (smaller) than the runtime of the other algorithm in significantly more than half of the cases.

We will also be interested in quantifying *by how much* an algorithm is slower than another algorithm. To this end, we will determine the largest factor $d \geq 1$ by which we can multiply one runtime distribution such that the Wilcoxon-Mann-Whitney test still yields a statistically significant p-value. For example, we will find that even if we multiply the runtime $R_{2.0}^{(2+1)-\mathrm{EA}}$ with $d = 63.15$ then this is still significantly faster than $R_{2.5}^{(2+1)-\mathrm{EA}}$ according to the Wilcoxon-Mann-Whitney test. Note that this is a posteriori hypothesis since the factor d is chosen in hindsight. Therefore, it must not be treated as an actually significant result. Still, it gives useful information, and shows that $R_{2.0}^{(2+1)-\mathrm{EA}}$ is *very much* smaller than $R_{2.5}^{(2+1)-\mathrm{EA}}$. All tests are done with R R Core Team (2020).

2.4 Analysis of population drift

As defined in the introduction, X_i is the number of zero-bits in an individual of the ith degenerated population, where degenerate means that all individuals are copies of the same search point. To be precise, if $P^{(i)}$ is the ith degenerate population, then $P^{(i+1)}$ is the first degenerate

population after $P^{(i)}$ has changed at least once. That does not exclude the possibility $P^{(i)} = P^{(i+1)}$, but we require at least one intermediate step in which an offspring is accepted that is not a copy of the parent(s) in $P^{(i)}$. Then the *degenerate population drift* (population drift for short) is $\mathbf{E}[X_i - X_{i+1} \mid X_i = y]$. We will estimate this drift with Monte-Carlo simulations. Moreover, for $y = o(n)$ we will derive a Markov chain state diagram in which all transition probabilities coincide with the transition probabilities in the real process up to $(1 + o(1))$ factors. By analyzing this state diagram, we are able to compute the population drift up to minor order error terms.

To motivate the use of population drift, consider the following example for $\mu = 2$. Take a population $\{x_1, x_2\}$, and assume that x_1 has at least as many one-bits than x_2. Then in every iteration, there is a chance to simply copy x_1 (mutate but flip none of the bits), and accept it. For this to happen, we need to first mutate x_1, flip no bits at all and accept the new offspring. The probability of mutating x_1 and flipping no bits is given by $\frac{1}{2} \cdot (1 - \frac{c}{n})^n \approx e^{-c}/2$ for the $(2+1)$-EA and $\frac{1}{4} \cdot (1 - \frac{c}{n})^n \approx e^{-c}/4$ for the $(2+1)$-GA and the $(2+1)$-GA-NoCopy . The probability of accepting x_1 is at least 1/2, as x_1 has at least as many 1s as x_2. Hence, we have a constant probability to degenerate to a population $\{x_1, x_1\}$ in every iteration. This implies that any population degenerates within an expected constant number of rounds. The argument can be generalized to larger μ, see Lengler (2019): for every constant μ and c, the expected time until the population degenerates from any starting population is $O(1)$. It is easy to see Lengler (2019) that for every constant μ and c, the expected time for degeneration is $O(1)$. Moreover, it was shown in Lengler (2019) that if the population drift is negative for $y = \alpha n$ for some $\alpha \in (1/2, 1)$ then asymptotically the runtime is exponential in n. On the other hand, if the population drift is positive for all α then the runtime is $O(n \log n)$. Hence, we are trying to identify parameter regimes for which areas of negative population drift occur.

3 Results

3.1 Runtimes

The results of our runtime simulations for different algorithms and values of c can be found in Fig. 2. As expected, we find a threshold behavior, i.e., there is a value c_0 such that the runtime increases dramatically as c crosses this threshold.

We will estimate threshold ranges for the different algorithms by visual inspection. We chose the estimates in a conservative manner, such that the ranges will be too big rather than too small. For the $(1+1)$-EA , we observe a threshold in $c_0 \in [1.5, 1.8]$ (in agreement with the theoretically derived threshold $c_0 \approx 1.59$ from Lengler and Schaller (2018)). For the $(2+1)$-EA , it seems to be within $c_0 \in [2.2, 2.3]$, for the $(2+1)$-GA in [3.0, 3.2] and for the $(2+1)$-GA-NoCopy in [4.1, 4.3]. Thus we obtain a clear ranking of the algorithms for $\mu \leq 2$, which is $(1+1)$-EA (worst), $(2+1)$-EA , $(2+1)$-GA , and $(2+1)$-GA-NoCopy . For the $(3+1)$-EA , the threshold appears to lie in the interval [2.5, 2.7], for the $(3+1)$-GA in [6.0, 6.3] and for the $(5+1)$-EA in [3.3, 3.45]. We were not able to find a threshold behavior for the $(5+1)$-GA for any $c < 20$. These results further confirm that the GA variants are performing massively better than its EA counterparts. Moreover, both for EAs and GAs, a larger population size shifts the threshold to the right. For GAs, this is analogous to theoretical results for monotone functions, but for EAs the effect goes in the opposite direction than for monotone functions, see the discussion in Sect. 1.

To validate the ranking for the algorithms with $\mu \leq 2$ statistically, we use the following comparisons. If we compare $R_{2.0}^{(1+1)-\mathrm{EA}}$ to $d \cdot R_{2.0}^{(2+1)-\mathrm{EA}}$ the Wilcoxon-Mann-Whitney test yields significant p-values ≤ 0.05 for every $d \leq 57.88$. We conclude that for mutation parameter $c = 2.0$, the $(1+1)$-EA is much slower than the $(2+1)$-EA . In the same manner, $d \cdot R_{2.5}^{(2+1)-\mathrm{GA}}$ is significantly smaller than $R_{2.5}^{(2+1)-\mathrm{EA}}$ for $d \leq 39.09$, and $d \cdot R_{3.5}^{(2+1)-\mathrm{GA-NoCopy}}$ is significantly smaller than $R_{3.5}^{(2+1)-\mathrm{GA}}$ for $d \leq 63.36$. This confirms the aforementioned ranking of the algorithms.

To establish intervals for the critical value c_0 of the algorithms with $\mu \leq 2$, we compare $R_{2.0}^{(1+1)-\mathrm{EA}}$ to $d \cdot R_{1.5}^{(1+1)-\mathrm{EA}}$, and find that the latter is significantly smaller for all $d \leq 38.84$. We interpret this huge drop in performance as strong indication that the threshold lies in the interval $c_0 \in [1.5, 2]$. Likewise, $R_{2.5}^{(2+1)-\mathrm{EA}}$ is larger than $d \cdot R_{2.0}^{(2+1)-\mathrm{EA}}$ for $d \leq 63.15$, $R_{3.5}^{(2+1)-\mathrm{GA}}$ is larger than $d \cdot R_{3.0}^{(2+1)-\mathrm{GA}}$ for $d \leq 29.00$, and $R_{4.5}^{(2+1)-\mathrm{GA-NoCopy}}$ is larger than $d \cdot R_{4.0}^{(2+1)-\mathrm{GA-NoCopy}}$ for $d \leq 29.59$, all with $p < 0.05$.

3.1.1 Degenerate population drift

We estimate the degenerate population drift by Monte-Carlo simulation on the $(2+1)$-EA with $c = 2.3$, which is slightly above the threshold. The results are visualized in Fig. 3. We can clearly see that the conditional population drift is negative in the area between 300 and 50 one-bits away from the optimum, but then becomes positive again when being less than 50 one-bits away from the optimum. We conclude that the hardest part for the $(2+1)$-EA is not

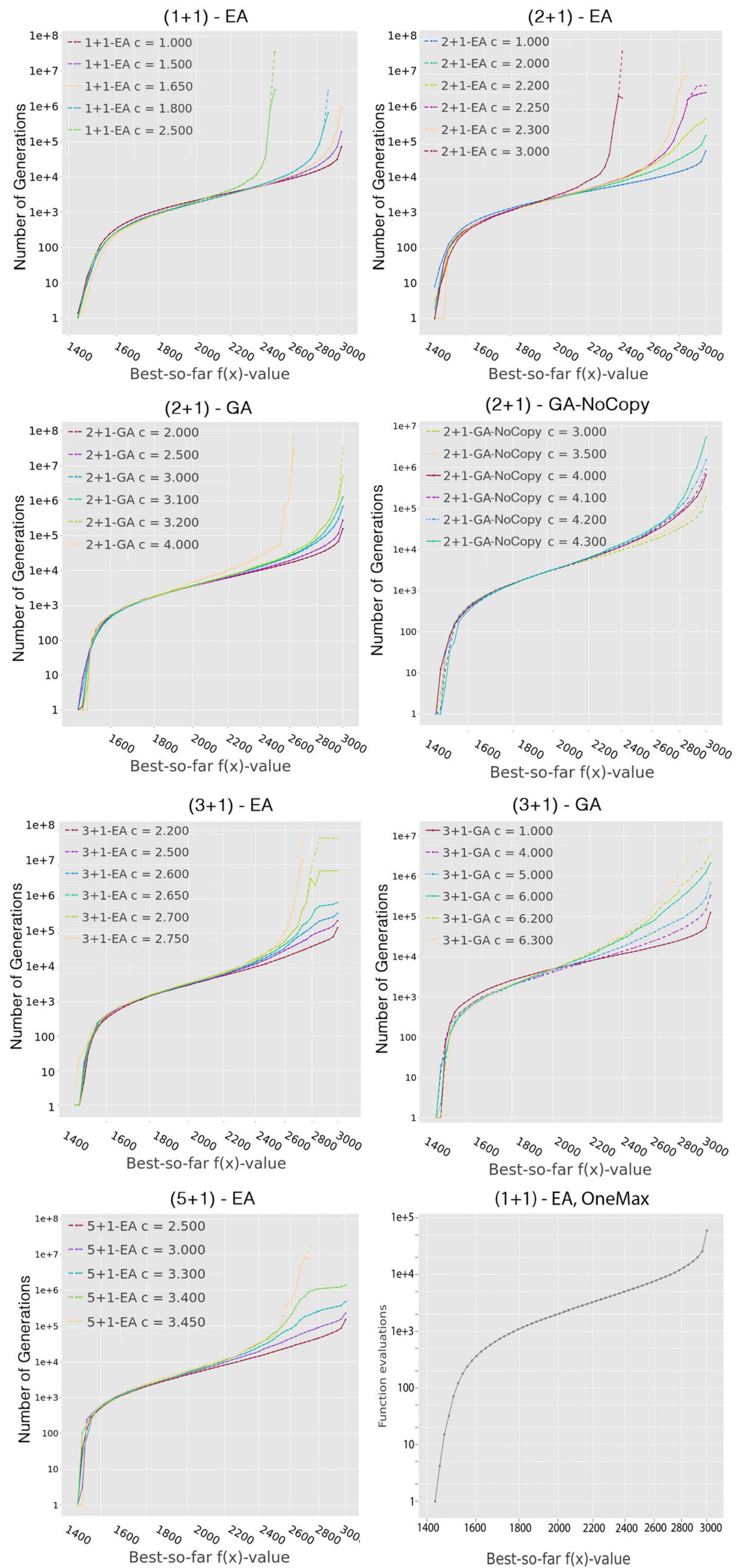

Fig. 2 Comparison of runtimes for different algorithms and values of c. We choose values of c that lie around the threshold for the respective algorithm. We plot both ERT and mean (both in the same color) if they differ significantly (mean is always the lower curve, see Sect. 3.1 for a more detailed explanation)

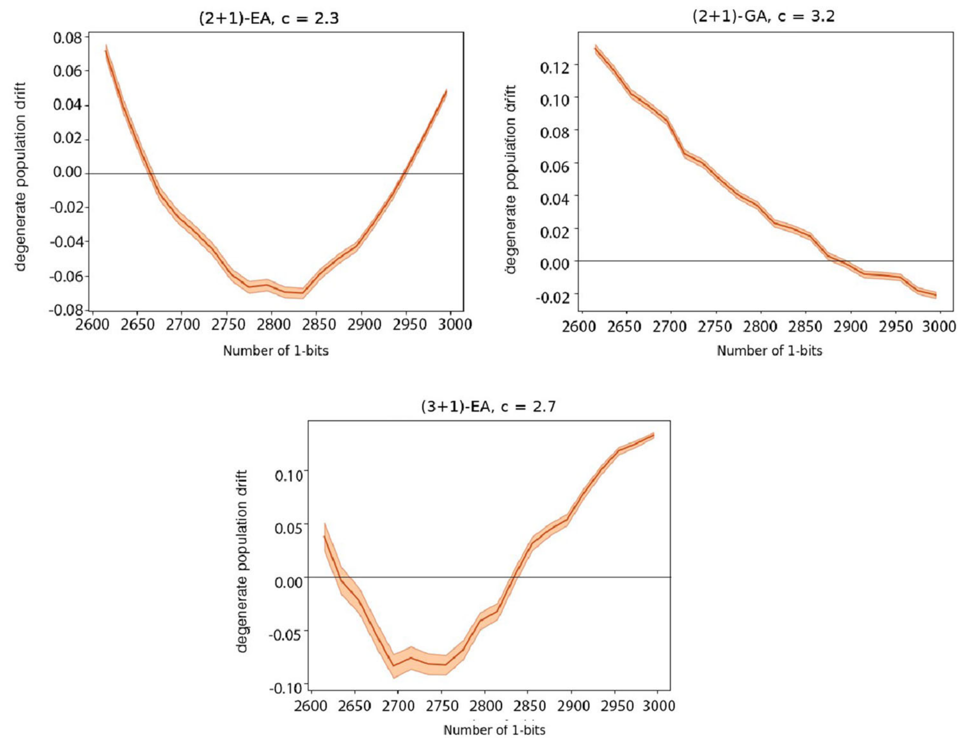

Fig. 3 Degenerate population drift for different algorithms and values of c just above the respective thresholds. The shaded area shows standard deviation

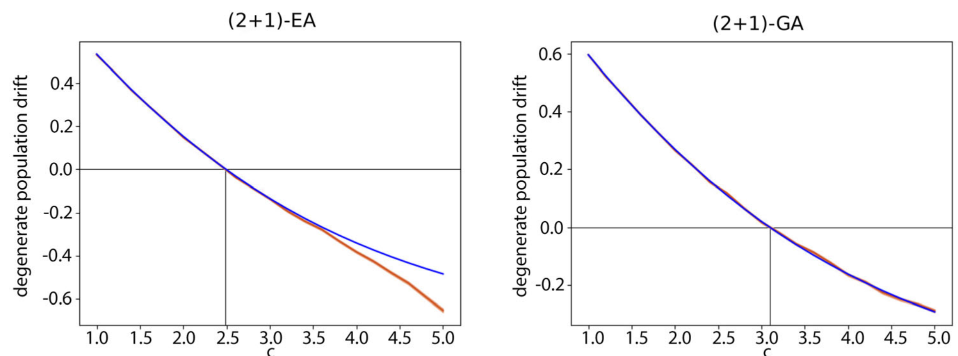

Fig. 4 Degenerate Population Drift for the $(2+1)$-EA and $(2+1)$-GA at the optimum. Blue: asymptotic formula. Orange: Monte Carlo simulation

around the optimum. We obtained similar results for the $(3+1)$-EA , also visualized in Fig. 3. This surprising result is similar to monotone functions (Lengler 2019; Lengler and Zou 2019), see the discussion in Sect. 1.

For the $(2+1)$-GA , the picture looks entirely different, as the drift is now strictly decreasing. We can only observe a negative drift area right at the optimum, starting from about 2900 one-bits. This behavior is similar to the $(1+1)$-EA and the $(\mu+1)$-GA-NoCopy , where the most difficult part is also close to the optimum (data not shown). Unfortunately, due to the large value of c_0, we were not able to obtain a conclusive result for the $(3+1)$-GA within reasonable computation time.

For the $(2+1)$-EA and $(2+1)$-GA , we also derive exact asymptotic formulas for the population drift close to the optimum. Since the derivation is rather tedious, we postpone it to the appendix. Define y to be the number of zero-bits. We compare the formula with the estimates via Monte Carlo simulation for different values of c, using $n=3000, y=1$, see Fig. 4. We can see that the curves match closely for small c, and that we get a moderate fit for larger c. We suspect that the deviations for the $(2+1)$-EA come from the expectation of the population drift being influenced by large but rare values of $X_i - X_{i+1}$ for large c. So the Monte Carlo simulations might be missing parts of this heavy tail. This tail is less heavy for the $(2+1)$-GA , since the probability to produce duplicates is always high, and thus degeneration happens quickly even for large c. In both cases, the curves agree perfectly in the *sign* of the population drift, which is our main interest. Negative drift at the optimum occurs at $c>3.1$ for the $(2+1)$-GA , which matches well the threshold obtained from runtime simulations. However, as expected, there is *no such match* for the $(2+1)$-EA , where the threshold for the population

drift at the optimum is 2.5 while the threshold for the runtime is below 2.3. I.e., the $(2+1)$-EA already struggles for values of c for which optimization around the optimum is easy. This confirms that the hardest region for the $(2+1)$-EA (but not the $(2+1)$-GA) is not at the optimum, but a bit away from it.

4 Conclusions

We have studied the effect of population size and crossover for the dynamic DYNBV benchmark. We have found that the algorithms generally profited from larger population size. Moreover, they profited strongly from crossover, even more so if we forbid crossovers between identical parents.

We have studied the case $\mu \leq 2$ in more depth. Remarkably, there is a strong qualitative difference between the $(2+1)$-EA and the $(2+1)$-GA . While for the latter one, the hardest region for optimization is close to the optimum (as one would expect), the same is not true for the $(2+1)$-EA . We believe that this is an interesting discovery. The only hint at such an effect on ONEMAX -like functions that we are aware of is for monotone functions (Lengler and Zou 2019). However, the results in Lengler and Zou (2019) predict that large population sizes hurt the $(\mu+1)$-EA , in opposition to our findings. Currently we are lacking any understanding of whether this comes from the small values of μ that we considered here, or whether it is due to the differences between monotone and dynamic linear functions.

For future work, there are many natural questions. We have chosen the $(\mu+1)$-GA to decide randomly between a mutation and a crossover step, but other choices are possible. Even with our formulation, it might be that the probability 1/2 for choosing crossover has a strong impact. And it is open whether and how the results transfer to other variants of the algorithm, for example a version that uses both crossover and mutation in each generation, or an algorithm that uses k-point crossover. Also, we have exclusively focused on the limiting case DYNBV , but dynamic linear functions are also interesting for less extreme case weight distributions. Finally, an interesting variant of dynamic linear functions or DYNBV might not change the objective every round, but only every s rounds (our runtime simulation already supports this feature and is publicly available).

Appendix

Degenerate population drift of the $(2+1)$-EA

Here, we derive an expression for the conditional drift $\mathbb{E}[X_i - X_{i+1} \mid X_i = y]$ for all $y = o(n)$, based on a state diagram for the $(2+1)$-EA . Recall that we have defined X_i to be the number of zero-bits in the ith degenerate population. We define $\mathcal{E}_{\text{progress}}$ as the event that an offspring is accepted into the degenerate population that is not identical with the old search point in the population. We will always assume that we are working close to the optimum, which means the number y of zero-bits is $o(n)$. This allows us to ignore cases where more than one zero-bit was flipped when going from one degenerated population to the next. Additionally, we are interested only in the case where $n \to \infty$. We will encounter several $o(1)$ terms in our calculations that will go to 0 as n becomes large.

We claim that the population follows the state diagram in Fig. 5. The transition probabilities are written below the arrows. Before we justify Fig. 5 in detail, note crucially that an arrow may summarize several generations. For example, assume that the algorithm generates a population $\{x, y\}$ in which x strictly dominates y, i.e., every one-bit in y is also a one-bit in x (and the converse is not true). In particular, x is strictly fitter than y. We could model this situation by a state $\tilde{S}$. However, recall that it only takes $O(1)$ steps until a copy of x is generated. Since by our assumption $y = o(n)$, the probability that any zero-bit is flipped into a one-bit in this time is $o(1)$. Hence, whp (with high probability, i.e., with probability $1 - o(1)$) all offspring are strictly dominated by x until a copy of x is created, and then the population degenerates. Hence, we would obtain an arrow from $\tilde{S}$ to the state $\{x, x\}$ with probability $1 - o(1)$, and the $o(1)$ will only affect the minor order terms of the final result. In cases like this, we will not draw the state $\tilde{S}$ in the first place. Instead, we may omit it, and replace any arrow to $\tilde{S}$ with an arrow (of the same probability) to the state $\{x, x\}$. This will allow us to keep the state diagram simple.

The top vertex represents a degenerate starting state $\{x, x\}$. Let y be the number of zero-bits in an individual x of our initial degenerated population $\{x, x\}$. For the other individual, we will use a superscript to denote the number of additional one-bits of an individual compared to x. This number can also be negative. By $S(k)$ we denote a state where the next degenerate population has $y - k$ zero-bits. So the initial top state is identical with $S(0)$.

Starting with a population $\{x, x\}$, three things can happen. First, no bits at all, or only one-bits are flipped. In this case, we will surely reject the offspring $\bar{x}$, as it is dominated (there is no position where $\bar{x}$ has a one-bit and x a zero-bit) by x. When being close to the optimum, this is what happens most of the time, as there are only few zero-bits left and they will rarely be flipped.

Secondly, we could also just flip a zero-bit, but no one-bits. Then we are in a state such that the offspring $\bar{x}^1$ is dominating x^0. Since whp no further one-bits are flipped,

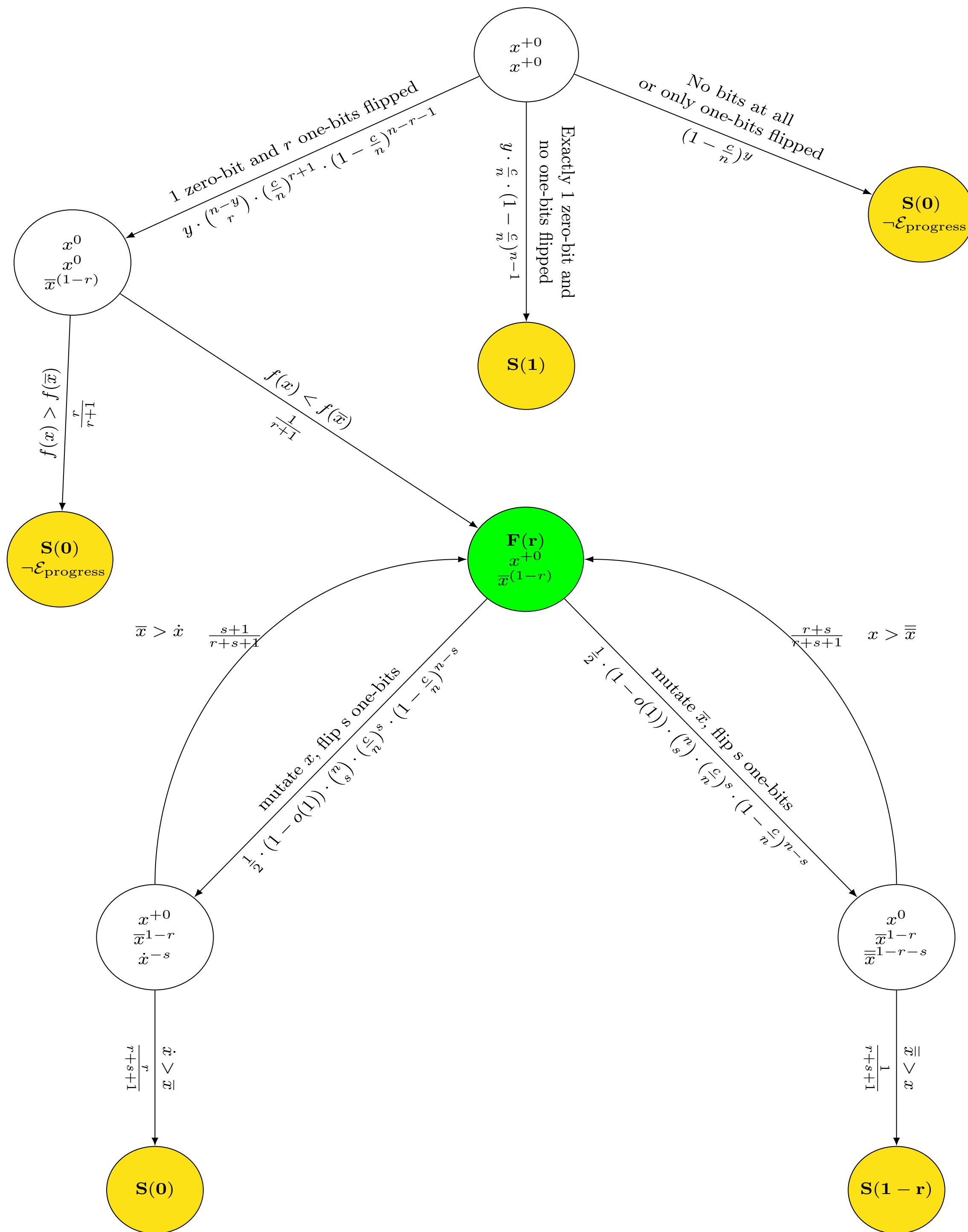

Fig. 5 Transition diagram of a degenerated population in the $(2+1)$-EA. (Colour figure online)

the population will whp degenerate to $\{\overline{x}, \overline{x}\}$, as no offspring will be accepted over $\overline{x}$. At some point, $\overline{x}$ will just be copied and accepted.

The third and most interesting case occurs when a zero-bit and $r \geq 1$ one-bits are flipped. We then have an offspring $\overline{x}^{(1-r)}$ such that $\overline{x}$ and x differ at exactly $r+1$ positions. In these $r+1$ positions, $\overline{x}$ has exactly one one-bit, and zero-bits everywhere else. Now $\overline{x}$ will either be rejected, which results in the same population $\{x, x\}$ we started with, or will be accepted. If it is accepted, we land in a state which we called $F(r)$ (green in Fig. 5).

Starting from $F(r)$ we can either mutate x or $\bar{x}$. Assume we mutate x, and flip s one-bits to create $\dot{x}^{-s}$. Notice that $\dot{x}$ is dominated by x. We can then either accept $\dot{x}$ to land in $S(0)$ or reject it to return to $F(r)$ once again.

If we mutate $\bar{x}$, we create an offspring $\bar{\bar{x}}^{1-r-s}$, which is dominated by $\bar{x}$. If we accept $\bar{\bar{x}}$, we will conclude in $S(1-r)$, otherwise we will go back to $F(r)$.

Putting it all together, we can get an explicit formula for the drift. In the diagram, note that we need to compute the drift conditioned on $\mathcal{E}_{\text{progress}}$, i.e., assuming we visit either $S(1)$ or $F(r)$.

First, we compute the expected number of zero-bits when starting in state $F(r)$. Let us slightly abuse notation and also call this $F(r) := \mathbb{E}[X_{i+1} - X_i \mid X_i = y \in o(n) \wedge$ we are in state $F(r)]$. Recall that we land in state $F(r)$ if we flip one zero-bit and r one-bits to create an offspring $\bar{x}$ and accept it. Simply writing out the transition probabilities yields

$$F(r) = (1 \pm o(1))\frac{1}{2}\sum_{s=0}^{n-y}((1-o(1))\binom{n}{s}\left(\frac{c}{n}\right)^s \left(1-\frac{c}{n}\right)^{n-s} \cdot \frac{1}{r+s+1} \cdot ((s+1)\cdot F(r) + r\cdot 0 + (r+s)\cdot F(r) + 1\cdot(1-r)).$$

Approximating the sums by letting them run up to infinity only gives another factor $(1 \pm o(1))$. The dominant terms have small s, and so we may approximate $\binom{n}{s}(c/n)^s(1-c/n)^{n-s} = (1 \pm o(1))c^s e^{-c}/s!$ in this case. Hence, we obtain

$$2F(r) = (1 \pm o(1))\sum_{s=0}^{\infty}\frac{c^s}{s!}\cdot e^{-c}\cdot\frac{r+2s+1}{r+s+1} \cdot F(r) + \sum_{s=0}^{\infty}\frac{c^s}{s!}\cdot e^{-c}\cdot\frac{1-r}{r+s+1}.$$

For the left hand side, we artificially write $F(r) = \sum_{s=0}^{\infty} c^s e^{-c}/s! \cdot F(r)$, which is true since the sum evaluates to 1. Solving for $F(r)$ yields

$$F(r) = (1 \pm o(1))\frac{\sum_{s=0}^{\infty}\frac{c^s}{s!}\cdot e^{-c}\cdot\frac{1}{r+s+1}\cdot(1-r)}{\sum_{s=0}^{\infty}\frac{c^s}{s!}\cdot e^{-c}\cdot\frac{1}{r+s+1}\cdot(r+1)} = (1 \pm o(1))\frac{1-r}{r+1}.$$

Before we can use $F(r)$ to compute the population drift, we need some elementary probabilities. Let $\mathcal{E}_0^r$ be the event that exactly r zero-bits are flipped and $\mathcal{E}_1^r$ the event that exactly r one-bits are flipped. Also, define $\mathcal{E}_{acc}$ to be the event that the offspring is accepted. We can compute

$$\mathbb{P}\left[\mathcal{E}_0^0 \wedge \mathcal{E}_1^0\right] = \left(1-\frac{c}{n}\right)^y,$$

$$\mathbb{P}\left[\mathcal{E}_0^1 \wedge \mathcal{E}_1^0\right] = y\cdot\frac{c}{n}\cdot\left(1-\frac{c}{n}\right)^{n-1},$$

$$\mathbb{P}\left[\mathcal{E}_0^1 \wedge \mathcal{E}_1^r\right] = y\cdot\binom{n-y}{r}\cdot\left(\frac{c}{n}\right)^{r+1}\cdot\left(1-\frac{c}{n}\right)^{n-r-1},$$

$$\mathbb{P}\left[\mathcal{E}_0^1 \wedge \neg\mathcal{E}_1^0 \wedge \neg\mathcal{E}_{acc}\right] = \sum_{r=1}^{r_{\max}}\mathbb{P}\left[\mathcal{E}_0^1 \wedge \mathcal{E}_1^r\right]\cdot\frac{r}{r+1},$$

$$\mathbb{P}\left[\mathcal{E}_0^1 \wedge \mathcal{E}_1^r \wedge \mathcal{E}_{acc}\right] = \mathbb{P}\left[\mathcal{E}_0^1 \wedge \mathcal{E}_1^r\right]\cdot\frac{1}{r+1},$$

$$\mathbb{P}\left[\mathcal{E}_{\text{progress}}\right] = 1 - \mathbb{P}\left[\mathcal{E}_0^0 \wedge \mathcal{E}_1^0\right] + \mathbb{P}\left[\mathcal{E}_0^1 \wedge \neg\mathcal{E}_1^0 \wedge \neg\mathcal{E}_{acc}\right].$$

Here, $r_{\max}$ could be as large as $n-y$, but for numerical evaluations of the formula we will cut off at a value such that the difference is negligible, e.g. 50. With these values, we can finally give a formula for the population drift:

$$\mathbb{E}[X_i - X_{i+1} \mid X_i = y] = \frac{\mathbb{P}\left[\mathcal{E}_0^1 \wedge \mathcal{E}_1^0\right] + \sum_{r=1}^{r_{\max}}\mathbb{P}\left[\mathcal{E}_0^1 \wedge \mathcal{E}_1^r \wedge \mathcal{E}_{acc}\right]\cdot F(r)}{\mathbb{P}\left[\mathcal{E}_{\text{progress}}\right]}.$$

Degenerate population drift of the (2 + 1)-GA

We compute the degenerate population drift at the optimum using a state diagram as we did with the $(2+1)$-EA , see Fig. 6. We only show the part that has changed significantly, which is the state that we now call $\overline{F}(r)$ (before $F(r)$). The upper part of the state diagram would be the same as in Fig. 5, except that now we can also do a crossover in the initial state, which will not change our population. Notice that the population drift will also exclude this case, since we only consider cases where at least one new offspring is accepted in the process.

Starting from $\overline{F}(r)$, doing a mutation gives exactly the same as in the $(2+1)$-EA . The other possibility is a crossover between two strings. If we do a crossover between x and x (x will just be copied in this case), we can either accept x over $\bar{x}$, to get to a state $S(0)$, or reject it to return to $\overline{F}(r)$. Similarly, we can do a crossover between $\bar{x}$ and $\bar{x}$ and either accept to end in $S(1-r)$ or reject to land back in $\overline{F}(r)$.

Finally, there may be a crossover between x and $\bar{x}$. To simplify notation, let us remove the bits on which x and $\bar{x}$ agree. Moreover, we may assume that the first position is the one at which $\bar{x}$ has a one-bit, but not x. So we are left with $x' = (0, 1, 1, \ldots, 1)$ and $\bar{x}' = (1, 0, 0, \ldots, 0)$. We can now do a case distinction depending on the shape of our crossover result $\dot{x}$. If $\dot{x}$ gets every one-bit, we land in a state $S(1)$, as $\dot{x}$ dominates both x and $\bar{x}$. If $\dot{x}$ has a one-bit at the first position, plus inherits $s < r$ one-bits from x, we can either remove $\bar{x}$ to go into a state $\overline{F}(r-s)$ or remove x, and

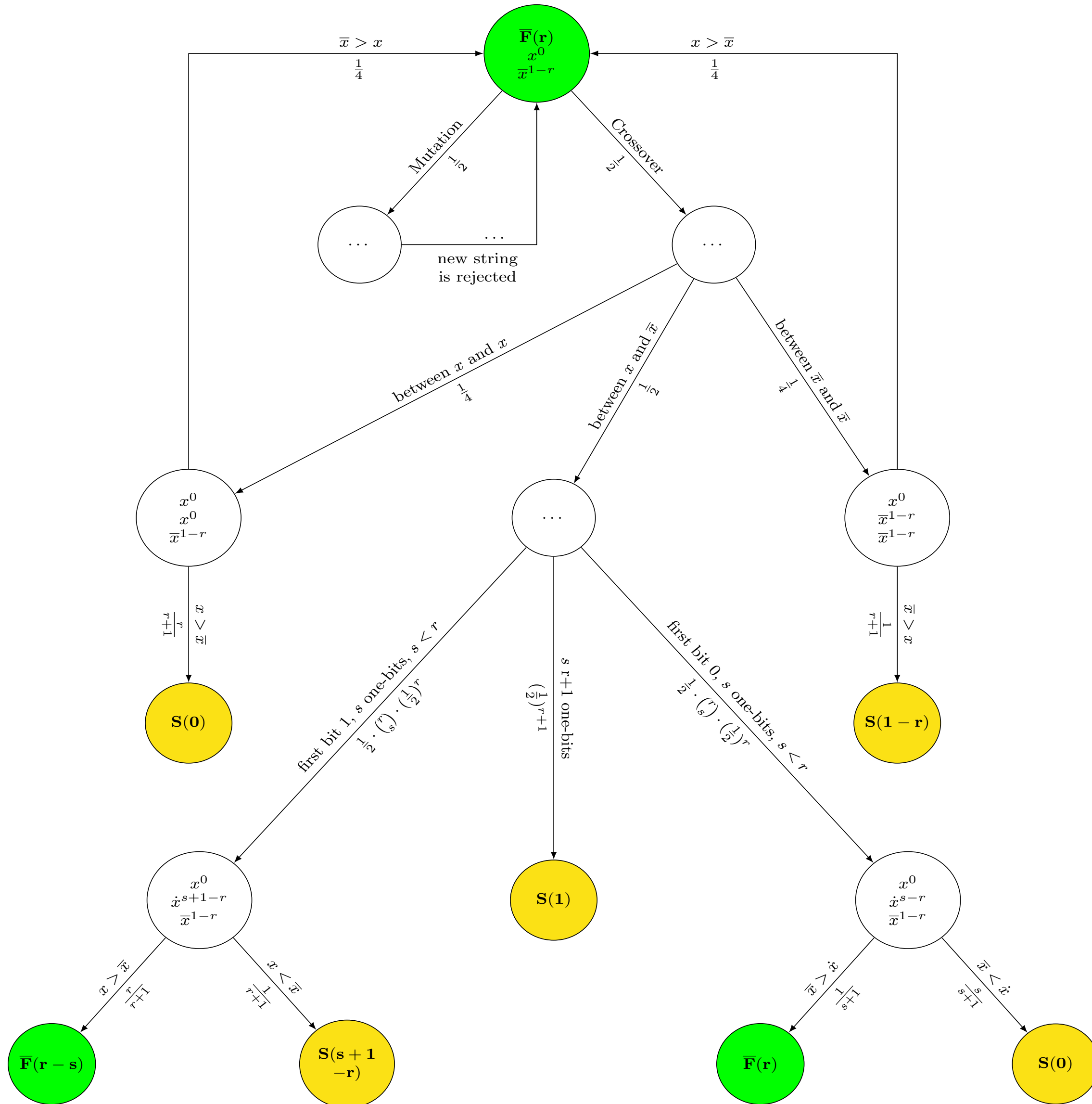

Fig. 6 Transition diagram of a degenerated population in the $(2+1)$-GA

land in $S(s+1-r)$, as $\dot{x}$ dominates $\overline{x}$. Finally, $\dot{x}$ could have a zero-bit at the first position and inherit s one-bits from x. Then we could reject $\dot{x}$ to return to $\overline{F}(r)$ or accept it over $\overline{x}$ to conclude in $S(0)$.

We can also derive the conditional drift for the (2+1)-GA in exactly the same fashion as before. We start again by computing $\overline{F}(r) := \mathbb{E}[X_{i+1} \mid X_i = y \wedge \text{we are in state } \overline{F}(r)]$ for $y \in o(n)$. Notice that now we obtain a recursive formula as we can go from a state $F(r)$ to a state $F(r-s)$.

$$
\begin{aligned}
\overline{F}(r) = {} & \frac{1}{4}\sum_{s=0}^{n-y}((1 \pm o(1))\binom{n}{s}\left(\frac{c}{n}\right)^s\left(1-\frac{c}{n}\right)^{n-s}\frac{1}{r+s+1} \\
& \cdot \left((s+1)\cdot\overline{F}(r) + r\cdot 0 + (r+s)\cdot\overline{F}(r) + 1\cdot(1-r)\right) \\
& + \frac{1}{8}\cdot\frac{1}{r+1}\cdot\left(r\cdot 0 + 1\cdot\overline{F}(r) + r\cdot(1-r) + r\cdot\overline{F}(r)\right) \\
& + \frac{1}{4}\cdot\sum_{s=0}^{r-1}\binom{r}{s}\left(\frac{1}{2}\right)^{r+1}\cdot\frac{1}{r+1}\cdot\left(1\cdot(s+1-r) + r\cdot\overline{F}(r-s)\right) \\
& + \frac{1}{4}\cdot\left(\frac{1}{2}\right)^{r+1} + \frac{1}{4}\cdot\sum_{s=0}^{r}\binom{r}{s}\left(\frac{1}{2}\right)^{r+1}\frac{1}{s+1}\cdot\left(s\cdot 0 + 1\cdot\overline{F}(r)\right).
\end{aligned}
$$

As for the $(2+1)$-EA , we may approximate $\binom{n}{s}(c/n)^s(1-c/n)^{n-s} = (1 \pm o(1))c^s e^{-c}/s!$ for small s, which are dominating. After simplification, we obtain

$$4\overline{F}(r) = \sum_{s=0}^{\infty} \frac{c^s}{s!} e^{-c} \cdot \left(\frac{r+2s+1}{r+s+1} \cdot \overline{F}(r) + \frac{1-r}{r+s+1} \right)$$
$$+ \frac{1}{2} \cdot \overline{F}(r) + \frac{1}{2} \cdot \frac{1-r}{r+1} + \sum_{s=0}^{r-1} \binom{r}{s} \left(\frac{1}{2}\right)^{r+1}$$
$$\cdot \frac{s+1-r}{r+1}$$
$$+ \sum_{s=1}^{r-1} \binom{r}{s} \left(\frac{1}{2}\right)^{r+1} \cdot \frac{r}{r+1} \cdot \overline{F}(r-s)$$
$$+ \left(\frac{1}{2}\right)^{r+1} \cdot \frac{r}{r+1} \cdot \overline{F}(r) +$$
$$+ \left(\frac{1}{2}\right)^{r+1} + \sum_{s=0}^{r} \binom{r}{s} \cdot \left(\frac{1}{2}\right)^{r+1} \cdot \frac{\overline{F}(r)}{s+1}.$$

Unfortunately, the formula does not simplify as much as for the $(2+1)$-EA . Solving for $\overline{F}(r)$ yields a recursive formula, which can still be evaluated numerically:

$$\overline{F}(r) = \frac{\sum_{s=0}^{\infty} \frac{c^s}{s!} e^{-c} \frac{1-r}{r+s+1} + \frac{1-r}{2r+2} + \sum_{s=1}^{r-1} \binom{r}{s} \left(\frac{1}{2}\right)^{r+1} \frac{r\overline{F}(r-s)}{r+1} + \left(\frac{1}{2}\right)^{r+1} + \sum_{s=0}^{r-1} \binom{r}{s} \left(\frac{1}{2}\right)^{r+1} \frac{s+1-r}{r+1}}{2 - \sum_{s=0}^{\infty} \frac{c^s}{s!} e^{-c} \cdot \frac{s}{r+s+1} - \left(\frac{1}{2}\right)^{r+1} \frac{r}{r+1} - \sum_{s=0}^{r} \binom{r}{s} \left(\frac{1}{2}\right)^{r+1} \cdot \frac{1}{s+1}}$$

Now computing the population drift is exactly the same exercise as for the $(2+1)$-EA , except that we have to account for crossovers in the initial states. We only have to adjust the probability $\mathbf{P}[\mathcal{E}_{\text{progress}}]$ and beware that mutations now have an additional $\frac{1}{2}$ factor. Then we can write down the expression for the conditional drift of the $(2+1)$-GA .

$$\mathbf{P}[\mathcal{E}_{\text{progress}}] = \frac{1}{2} - \frac{1}{2}\left(\mathbb{P}\left[\mathcal{E}_0^0 \wedge \mathcal{E}_1^0\right] + \mathbb{P}\left[\mathcal{E}_0^1 \wedge \neg\mathcal{E}_1^0 \wedge \neg\mathcal{E}_{acc}\right]\right)$$

$$\mathbb{E}[X_i - X_{i+1} \mid X_i = y \wedge \mathcal{E}_{\text{progress}}] = \frac{\mathbb{P}\left[\mathcal{E}_0^1 \wedge \mathcal{E}_1^0\right] + \sum_{r=1}^{r_{max}} \mathbb{P}\left[\mathcal{E}_0^1 \wedge \mathcal{E}_1^r \wedge \mathcal{E}_{acc}\right] \cdot \overline{F}(r)}{2\mathbb{P}\left[\mathcal{E}_{\text{progress}}\right]}$$

Funding Open access funding provided by Swiss Federal Institute of Technology Zurich.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/ .

References

Antipov D, Doerr B, Fang J, Hetet T (2018) A tight runtime analysis for the (μ+ λ) EA. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO). pp 1459–1466

Dang DC, Friedrich T, Kötzing T, Krejca MS, Lehre PK, Oliveto PS, Sudholt D, Sutton AM (2017) Escaping local optima using crossover with emergent diversity. IEEE Trans Evol Comput 22(3):484–497

Dang DC, Jansen T, Lehre PK (2017) Populations can be essential in tracking dynamic optima. Algorithmica 78(2):660–680

Doerr B, Happ E, Klein C (2012) Crossover can provably be useful in evolutionary computation. Theo Comput Sci 425:17–33

Doerr B, Jansen T, Sudholt D, Winzen C, Zarges C (2013) Mutation rate matters even when optimizing monotonic functions. Evol Comput 21(1):1–27

Doerr C, Wang H, Ye F, van Rijn S, Bäck T (2018) IOHprofiler: A benchmarking and profiling tool for iterative optimization heuristics. arXiv preprint arXiv:1810.05281 , https://iohprofiler.github.io/

Hwang HK, Panholzer A, Rolin N, Tsai TH, Chen WM (2018) Probabilistic analysis of the (1+1)-evolutionary algorithm. Evol Comput 26(2):299–345

Jansen T, Wegener I (2002) The analysis of evolutionary algorithms-a proof that crossover really can help. Algorithmica 34(1):47–66

Lengler J (2019) A general dichotomy of evolutionary algorithms on monotone functions. IEEE Trans Evol Comput 24:995

Lengler J, Martinsson A, Steger A (2019) When does hillclimbing fail on monotone functions: an entropy compression argument. In: Proceedings of the Sixteenth Workshop on Analytic Algorithmics and Combinatorics (ANALCO). pp 94–102

Lengler J, Meier J (2020) Evolutionary algorithms in dynamic environments, source code. https://github.com/JomeierFL/BachelorThesis

Lengler J, Schaller U (2018) The (1+1)-EA on noisy linear functions with random positive weights. In: Proceedings of the Symposium Series on Computational Intelligence (SSCI). pp 712–719

Lengler J, Steger A (2018) Drift analysis and evolutionary algorithms revisited. Comb Probab Comput 27(4):643–666

Lengler J, Zou X (2019) Exponential slowdown for larger populations: the (μ+1)-EA on monotone functions. In: Proceedings of the 15th ACM/SIGEVO Conference on Foundations of Genetic Algorithms (FOGA). pp 87–101

Lissovoi A, Witt C (2016) MMAS versus population-based EA on a family of dynamic fitness functions. Algorithmica 75(3):554–576

Pinto E.C, Doerr C (2018) A simple proof for the usefulness of crossover in black-box optimization. In: Proceedings of the International Conference on Parallel Problem Solving from Nature (PPSN). pp 29–41

R Core Team (2020) R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria, https://www.R-project.org/

Rohlfshagen P, Lehre PK, Yao X (2009) Dynamic evolutionary optimisation: an analysis of frequency and magnitude of change. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO). pp 1713–1720
Sudholt D (2017) How crossover speeds up building block assembly in genetic algorithms. Evol Comput 25(2):237–274
Sudholt D (2020) The benefits of population diversity in evolutionary algorithms: a survey of rigorous runtime analyses. In: Theory of Evolutionary Computation, pp 359–404
Witt C (2006) Runtime analysis of the (μ+1)-EA on simple pseudo-Boolean functions. Evol Comput 14(1):65–86
Witt C (2013) Tight bounds on the optimization time of a randomized search heuristic on linear functions. Comb Probab Comput 22(2):294–318

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.