Benchmarking discrete optimization heuristics: from building a sound experimental environment to algorithm configuration

Ye, F.

Chapter 4

Problem Specific Benchmarking: Study on OneMax and LeadingOnes

In this chapter, we focus on the two classic problems OneMax and LeadingOnes, to show how benchmarking can benefit theoreticians and practitioners for discussing new research directions. We start our investigation on the $(1 + \lambda)$ EAs comparing the theoretical results and empirical performance. A *normalized standard bit mutation* is proposed based on benchmarking an existing self-adaption of mutation rate for the $(1+\lambda)$ EAs. In addition, we study the impact of the population size and the mutation rate for the $(1 + \lambda)$ EAs and the crossover probability for the $(\mu + \lambda)$ GA.

4.1 Profiling $(1 + \lambda)$ EA

4.1.1 Background

Two fundamental building blocks of evolutionary algorithms are global variation operators and populations. *Global variation operators* are sampling strategies that are characterized by the property that every possible solution candidate has a positive probability of being sampled within a short time window, regardless of the current state of the algorithm. *Standard bit mutation* is an example of a global mutation operator. From a given input string $x \in \{0, 1\}^n$, standard bit mutation creates an offspring by flipping each bit in x with some positive probability $0 < p < 1$, with independent decisions for each bit. For any x, y the probability to sample y from x is thus $p^{H(x,y)}(1-p)^{n-H(x,y)}$, where $H(x,y)$ is the number of bits in which x and y differ (*Hamming distance*). This probability is positive even for search points that are very far apart. The motivation to use global sampling strategies is to overcome local optima by eventually performing a sufficiently large jump.

Storing information about the optimization process, maintaining a diverse set of reasonably good solutions, and gathering a more complete picture about the structure of the problem at hand are among the most important reasons to employ *population-based EAs*. The first two objectives are served by the *parent population*, which is the subset of previously evaluated search points that are kept in the memory of the algorithm. The parent population is updated after each generation. New solution candidates are sampled from it through the use of variation operators. These points form the *offspring population* of the generation. Non-trivial offspring population sizes address the desire to gather more information about the fitness landscape before making any decision about which of the points from the parent and offspring population to keep in the memory for the next iteration.

It is very well understood that both the size of the parent population as well as the size of the offspring population can have a significant impact on the performance. Finding suitable parameter values for these two quantities remains to be a challenging problem in practical applications of EAs. From an analytical point of view, populations increase the complexity of the optimization process considerably, as they introduce a lot of dependencies that need to be taken care of in the mathematical analysis. It is therefore not surprising that only few theoretical works on population-based EAs exist, cf. [105] and references mentioned therein. Most existing theoretical works regard algorithms with non-trivial *offspring population* sizes, while the impact of the *parent population* size has received much less interest.

We present below empirical and theoretical results for the $(1+\lambda)$ EA, the arguably simplest EA that combines a global sampling technique with a non-trivial offspring population size.

4.1.2 Algorithms

As noted in [128] there exists an important discrepancy between the algorithms classically regarded in the theory of evolutionary computation literature and their common implementations in practice. For mutation-based algorithms like $(\mu + \lambda)$ and

Algorithm 1: flip_ℓ chooses ℓ different positions and flips the entries in these positions.

```
Input: $x = (x_1 \dots x_n) \in \{0,1\}^n$, $\ell \in \mathbb{N}$;
$y \leftarrow x$;
Select $\ell$ pairwise different positions $i_1, \dots, i_\ell \in [n]$ u.a.r.;
for $j = 1, \dots, \ell$ do $y_{i_j} \leftarrow 1 - x_{i_j}$;
```

(μ, λ) EAs, this discrepancy concerns the way new solution candidates are sampled from previously evaluated ones, and how the function evaluations are counted. Both algorithms use the above-described standard bit mutation as only variation operator. An often recommended value for the mutation rate p is $1/n$, which corresponds to flipping exactly one bit on average, an often desirable behavior when the search converges.

When implementing standard bit mutation, it would be rather inefficient to decide for each $i \in [n]$ whether or not the i-th bit of x should be flipped. Luckily, this is not needed, as we can simply observe that standard bit mutation can be equally expressed as drawing a random number ℓ from the binomial distribution $\text{Bin}(n,p)$ with n trials and success probability p and then flipping ℓ bits that are sampled from $[n]$ uniformly at random (u.a.r.) and without replacement. This latter operation is formalized by the flip_ℓ operator in Algorithm 1. We refer to ℓ as the *mutation strength* or the *step size*, while we call p the *mutation rate*.

Analyzing standard bit mutation, we easily observe that the probability to not flip any bit at all equals $(1-p)^n$, which for $p = 1/n$ converges to $1/\exp(1)$. That is, in about 36.8% of calls to this operator, a copy of the input is returned. For the $(1+\lambda)$ EA there is no benefit of evaluating such a copy (unless facing a dynamic or noisy optimization setting), since it applies *plus selection*, where both the parent as well as the offspring can be selected to "survive" for the next generation. It is therefore advisable to change the probability distribution from which the mutation strength ℓ is sampled. A straightforward (and commonly used) idea is to simply re-sample ℓ from $\text{Bin}(n,p)$ until a non-zero value is returned. This approach corresponds to distributing the probability mass $(1-p)^n$ of sampling a zero proportionally to all step sizes $\ell > 0$. This gives the conditional binomial distribution $\text{Bin}_{>0}(n,p)$, which assigns to each $\ell \in \mathbb{N}$ a probability of $\binom{n}{\ell} p^\ell (1-p)^{n-\ell} / (1-(1-p)^n)$. All our empirical results use this conditional sampling strategy. The results can therefore differ significantly from figures previously published in the theory of EA literature [91, 128].

Algorithm 2: The $(1 + \lambda)$ $\text{EA}_{>0}$ with mutation rate $p \in (0, 1)$ for the maximization of $f : \{0, 1\}^n \to \mathbb{R}$

```
Initialization: Sample $x \in \{0,1\}^n$ u.a.r.;
Optimization: for $t = 1, 2, 3, \ldots$ do
    for $i = 1, \ldots, \lambda$ do
        Sample $\ell^{(i)}$ from $\text{Bin}_{>0}(n, p)$;
        $y^{(i)} \leftarrow \text{flip}_{\ell^{(i)}}(x)$;
    Sample $x$ from $\arg\max\{f(x), f(y^{(1)}), \ldots, f(y^{(\lambda)})\}$ u.a.r.;
```

The Basic $(1+\lambda)$ $\text{EA}_{>0}$ The $(1+\lambda)$ EA samples λ offspring in every iteration, from which only the best one survives (ties broken uniformly at random). Each offspring is created by standard bit mutation. Following our discussion above, we make use of the re-sampling strategy described above, and obtain the $(1 + \lambda)$ $\text{EA}_{>0}$, which we summarize in Algorithm 2.

Adaptive $(1+\lambda)$ $\text{EA}_{>0}$ The $(1+\lambda)$ $\text{EA}_{>0}$ has two parameters: the offspring population size λ and the mutation rate p. Common implementations of the $(1+\lambda)$ $\text{EA}_{>0}$ use the same population size λ and the same mutation rate p throughout the whole optimization process (*static parameter choice*), while the use of *dynamic parameter values* is much less established. A few works exist, nevertheless, that propose to *control* the parameters of the $(1 + \lambda)$ EA online [92]. We focus in our empirical comparison on algorithms that have a mathematical support. These are summarized in the following two subsections.

Adaptive Mutation Rates One of the few works that experiments with a *non-static mutation rate* for the $(1 + \lambda)$ EA was presented in [46]. The there-suggested algorithm stores a parameter r that is adjusted online. In each iteration, the $(1 + \lambda)$ $\text{EA}_{r/2,2r}$ creates $\lambda/2$ offspring by standard bit mutation with mutation rate $r/(2n)$, and it creates $\lambda/2$ offspring with mutation rate $2r/n$. The value of r is updated after each iteration. With probability $1/2$ it is set to the value that the best offspring individual of the last iteration has been created with (ties broken at random), and it is replaced by either $r/2$ or $2r$ otherwise (unbiased random decision). Finally, the value r is capped at 2 if smaller, and at $n/4$, if it exceeds this value. In our experiments, we use $r = 2$ as initial value.

In [46] it is shown analytically that the $(1 + \lambda)$ $\text{EA}_{r/2,2r}$ yields an asymptotically optimal runtime on OneMax. This performance is strictly better than what any static mutation rate can achieve, cf. Section 4.1.3. How well the adaptive scheme works for other benchmark problems is left as an open question in [46].

Adaptive Population Sizes Apart from the mutation rate, one can also consider to *adjust the offspring population size* λ. This is a much more prominent problem, because λ is an explicit parameter, while the mutation rate is often not specified (and thus by default assumed to be $1/n$).

In the theory of EC literature, the following three success-based update rules have been studied. In [87] the offspring population size λ is initialized as one. After each iteration, we count the number s of offspring that are at least as good as the parent. When $s = 0$, we double the population size, and we replace it by $\lfloor \lambda/s \rfloor$ otherwise. For brevity, we call this algorithm the $(1 + \{2\lambda, \lambda/s\})$ EA, and its resampling variant the $(1 + \{2\lambda, \lambda/s\})$ EA$_{>0}$.

Two similar schemes were studied in [104] where λ is doubled if no strictly better search point has been identified and either set to one or to $\max\{1, \lfloor \lambda/2 \rfloor\}$ otherwise. We regard here the resampling variants of these algorithms, which we call the $(1 + \{2\lambda, 1\})$ EA$_{>0}$ and the $(1 + \{2\lambda, \lambda/2\})$ EA$_{>0}$, respectively.

4.1.3 Profiling on OneMax

We recall that the class of OneMax functions is the generalization of the function OM that assigns to each bit string x the number $|\{i \in [n] \mid x_i = 1\}|$ of ones in it. For this generalization, OM is composed with all possible XOR operations on the hypercube. More precisely, for any bit string $z \in \{0,1\}^n$ we define the function $\mathrm{OM}_z : \{0,1\}^n \to [0..n], x \mapsto |\{i \in [n] \mid x_i = z_i\}|$, the number of bits in which x and z agree. The OneMax problem is the collection of all functions OM_z, $z \in \{0,1\}^n$.

Theoretical Bounds OneMax is often referred to as the *drosophila of EC*. It is therefore not surprising that among all benchmark functions, OneMax is the problem for which most runtime results are available. We summarize in this section a few selected results.

Concerning the $(1 + \lambda)$ EA, the first question that one might ask is whether or not it can be beneficial to generate more than one offspring per iteration. When using the number of function evaluations (and not the number of generations) as performance measure, intuitively, it should always be better to create the offspring sequentially, to profit from intermediate fitness gains. This intuition has been formally proven in [87], where it is shown that for all $\lambda, k \in \mathbb{N}$ the expected optimization time (i.e., the number of function evaluations until an optimal solution is queried for the first time) of the $(1+k\lambda)$ EA cannot be better than that of the $(1+\lambda)$ EA. This result implies that $k=1$ is an optimal choice. Note, however, that the number of *generations* needed to find an optimal solution can significantly decrease with increasing λ, so that it can be beneficial even for OneMax to run the $(1+\lambda)$ EA with $\lambda > 1$ when parallel function evaluations are possible.

For $\lambda = 1$ the runtime of the $(1+1)$ EA with **static mutation rate** $p > 0$ is quite well understood, cf. [163] for a detailed discussion. For general λ and static mutation rate $p = c/n$ (where here and henceforth $c > 0$ is assumed to be constant), the expected optimization time is $(1 \pm o(1))\big(\frac{n\lambda \ln\ln\lambda}{2\ln\lambda} + \frac{e^c}{c} n \ln n\big)$ [68]. An interesting observation made in [46] reveals that the parametrization $p = c/n$ is suboptimal: with $p = \ln(\lambda)/(2n)$ the $(1+\lambda)$ EA needs only an expected number of $O(n\lambda/\log(\lambda) + \sqrt{\lambda} n \log n)$ function evaluations to optimize OneMax [46] (the proof requires $\lambda \geq 45$ and $\lambda = n^{O(1)}$). We call this algorithm the $(1+\lambda)$ $\text{EA}_{p=\ln(\lambda)/(2n)}$.

When using **non-static mutation rates**, the best expected optimization time that a $(1+\lambda)$ EA can achieve on OneMax is bounded from below by $\Omega(n\lambda/\log(\lambda) + n \log n)$ [8]. This bound is attained by a $(1+\lambda)$ EA variant with fitness-dependent mutation rates [8]. Interestingly, it is also achieved by the self-adjusting $(1+\lambda)$ $\text{EA}_{r/2,2r}$ described in Section 5.1.2 (the proof requires again $\lambda \geq 45$ and $\lambda = n^{O(1)}$).

Concerning the algorithms using **non-static offspring population sizes** (see Section 5.1.2), we do not have an explicit theoretical analysis for the $(1+\{2\lambda, \lambda/s\})$ EA, but it is known that the expected optimization time of both the $(1+\{2\lambda, 1\})$ EA and the $(1+\{2\lambda, \lambda/2\})$ EA is $O(n \log n)$ [104].

Disclaimer. It is important to note that all the bounds reported above (and those mentioned in Section 4.1.4) hold, a priori, only for the classical $(1+\lambda)$ EA variants, not the resampling versions regarded here in this thesis. For most bounds, and in particular the ones with static parameter choices, it is, however, not difficult to prove that the modifications do not change the asymptotic order of the expected optimization times. What does change, however, is the leading constant. As a rule of thumb, runtime bounds for the $(1+\lambda)$ EA decrease by a multiplicative factor of about $1-(1-p)^n$ when the mutation strengths are sampled from the conditional binomial distribution $\text{Bin}_{>0}(n,p)$.

Note also that in our summary we collect only statements about the total expected *optimization time*, i.e., AHT. Here again the $(1+1)$ EA and the $(1+\lambda)$ EA with static parameters form an exception as for these two algorithms a few theoretical fixed-budget results exist, cf. [45, 108, 122] and references therein.

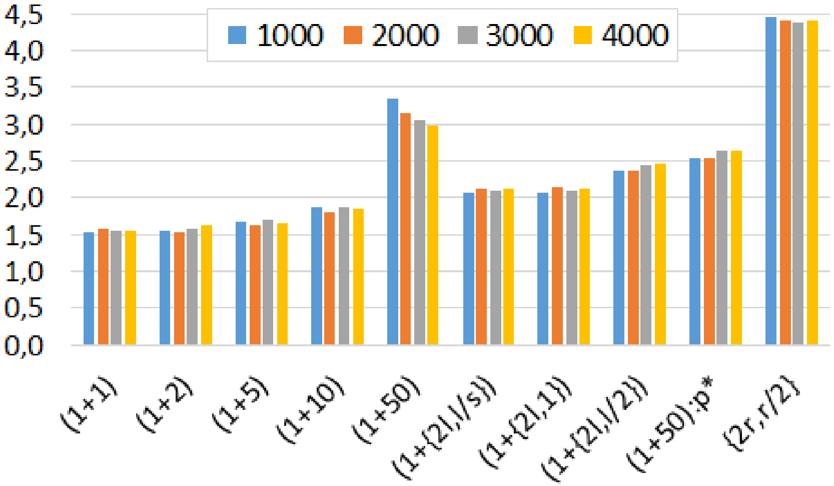


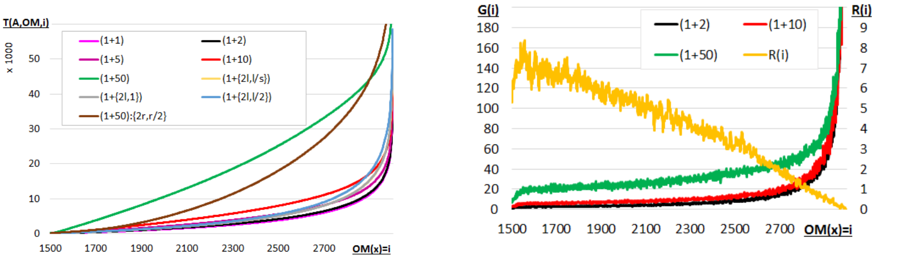
Figure 4.1: Average optimization times for 100 independent runs of the $(1+\lambda)$ EA$_{>0}$ variants for OneMax, normalized by $n \ln n$. (Initial) population size for the adaptive variants is 50. $p^* = \ln(\lambda)/(2n)$

Empirical Evaluation We now come to the results of our empirical investigation. All figures presented in this section are averages over 100 independent runs. This might look like a small number, but we recall that we track the whole optimization process, to present the fixed-target results below.

We have seen above that, for reasonable parameter settings, the average optimization times of the $(1+\lambda)$ EA variants are all of order $n \log n$. We therefore normalize the empirical averages in Figure 4.1 by this factor.

We observe that the normalized averages are quite stable across the dimensions, with an exception of the $(1+50)$ EA, whose relative performance improves with increasing problem dimension. We also see that the $(1+50)$ EA$_{>0}$ variant with $p^* = \ln(\lambda)/(2n)$ achieves a better optimization time than the $(1+50)$ EA$_{>0}$ with $p = 1/n$. The variants with adaptive offspring population size perform significantly worse than the $(1+1)$ EA$_{>0}$, which is not surprising given the performance hierarchy of the $(1+\lambda)$ EAs mentioned in the beginning of Section 4.1.3.

For the tested problem dimensions, the worst-performing algorithm in our comparison is the $(1+\lambda)$ EA$_{r/2,2r}$. This may come as a surprise since this algorithm is the one with the best theoretical support. The advantage of this algorithm seems to require much larger problem dimensions, different values of λ, and/or different settings of the hyper-parameters that determined its update mechanism.



(a) Average fixed-target runtimes $T(A, \mathrm{OM}, i)$.

(b) Gradients $G(A, \mathrm{OM}, i)$ and relative disadvantage $R(i)$ of the (1+50) $\mathrm{EA}_{>0}$ over the (1+2) $\mathrm{EA}_{>0}$.

Figure 4.2: Fixed target data for the 3,000-dimensional OneMax problem (averages over 100 runs)

A general question raised by the data in Figure 4.1 concerns the sensitivity of the adaptive $(1 + \lambda)$ EA variants with respect to the initialization of their parameters and with respect to their hyper-parameters, which are the update strengths, but also the initial parameter values of λ and r, respectively.

It has been discussed that for OneMax the performance of the $(1+\lambda)$ EA can only be worse than that of the $(1 + 1)$ EA. In fact, the data in Figure 4.1 demonstrates a quite significant discrepancy between the performance of the $(1+1)$ $\mathrm{EA}_{>0}$ and the $(1+\lambda)$ $\mathrm{EA}_{>0}$ variants with $\lambda \geq 10$. The expected optimization time of the $(1 + 50)$ $\mathrm{EA}_{>0}$, for example, is about twice as large as that of the $(1 + 1)$ $\mathrm{EA}_{>0}$. Intuitively, this can be explained as follows. At the beginning of the OneMax optimization process the probability that a random offspring created by the $(1+\lambda)$ EA improves upon its parent is constant. In this phase the $(1 + \lambda)$ EA variants with small λ have an advantage as they can (almost) instantly make use of this progress, while the $(1 + \lambda)$ EAs with large λ first need to wait for all λ offspring to be evaluated. Since the expected fitness gain of the best of these λ offspring is not much larger than that of a random individual, large offspring population sizes are detrimental in this first phase of the optimization process. We note, however, that the relative disadvantage of large λ is much smaller towards the end of the optimization process. When, say, the parent individual x satisfies $\mathrm{OM}(x) = n - \Theta(1)$, the probability that a random offspring has a better function value is of order $\Theta(1/n)$ only. We therefore have to create $\Theta(n)$ offspring, in expectation, before we see any progress. The relative disadvantage of creating several offspring in one generation is therefore almost negligible in the later parts of the optimization process (provided that $\lambda = O(n)$). This informal explanation is confirmed by the plots in Figure 4.2, which display for $n = 3,000$

1. in Figure 4.2a: the empirical average fixed-target times $T(A, \mathrm{OM}, i)$; i.e., the average number of function evaluations that the $(1+\lambda)$ $\mathrm{EA}_{>0}$ variant A needs to identify a solution of OM-value at least i. We cap this plot at $60,000$ evaluations.

2. in Figure 4.2b: the gradients $G(A,i) := T(A, \mathrm{OM}, i) - T(A, \mathrm{OM}, i-1)$ (three lowermost curves), and the relative difference $R_i := (\mathrm{avg}_{j=i,i+1,\dots,i+5}(G((1+50)\mathrm{EA}_{>0}, j) - \mathrm{avg}_{j=i,i+1,\dots,i+5}G((1+2)\mathrm{EA}_{>0}, j)/G((1+2)\mathrm{EA}_{>0}, j))$ of the rolling average of the gradients of the $(1+50)$ and the $(1+2)$ $\mathrm{EA}_{>0}$.

Note that the gradient $G(A,i)$ measures the average time needed by algorithm A to make a progress of one when starting in a point x of OneMax-value $\mathrm{OM}(x) = i$. Small gradients are therefore desirable. We see that, for example, the $(1+50)$ $\mathrm{EA}_{>0}$ (green curve) needs, on average, about 20 fitness evaluations to generate a strictly better search point when starting in a solution of OM-value around $1,700$. The $(1+2)$ $\mathrm{EA}_{>0}$, in contrast, needs only about 2.6 function evaluations, on average. The relative disadvantage of the $(1+50)$ $\mathrm{EA}_{>0}$ over the $(1+2)$ $\mathrm{EA}_{>0}$ decreases with increasing function values from around 7 to zero, cf. the uppermost (yellow) curve in Figure 4.2b. We use the rolling average of 5 consecutive values here to obtain a smoother curve for $R(i)$.

Another important insight from Figure 4.2a is that the dominance of the $(1+1)$ $\mathrm{EA}_{>0}$ over all $(1+\lambda)$ $\mathrm{EA}_{>0}$ variants does not only apply to the total expected optimization time, but also to all intermediate target values. This can be shown with mathematical rigor by adjusting the proofs in [87, Section 3] to suboptimal target values.

4.1.4 Profiling on LeadingOnes

We recall that LeadingOnes is the generalization of the function LO, which counts the number of initial ones in the string, i.e., $\mathrm{LO}(x) = \max\{i \in [0..n] \mid \forall j \leq i : x_j = 1\}$. The generalization is by composing with an XOR-shift and a permutation of the positions. This way, we obtain for every $z \in \{0,1\}^n$ and for every permutation (one-to-one map) σ of the set $[n]$ the function $\mathrm{LO}_{z,\sigma} : \{0,1\}^n \to \mathbb{N}$, which assigns to each x the function value $\max\{i \in [0..n] \mid \forall j \in [i] : x_{\sigma(j)} = z_{\sigma(j)}\}$. The LeadingOnes problem is the collection of all these functions.

Theoretical Bounds Also for LeadingOnes it has been proven that the optimal value of the offspring population size λ in the $(1+\lambda)$ EA is one when using function evaluations and not generations as performance indicator [87]. In contrast to

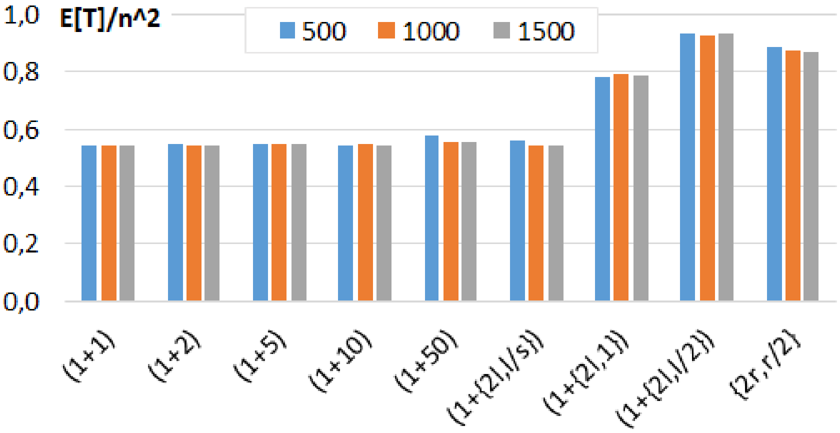


Figure 4.3: Average optimization times for 100 independent runs of the $(1 + \lambda)$ EA$_{>0}$ variants for LeadingOnes, normalized by n^2. (Initial) population size for the adaptive variants is 50.

OneMax, however, we will observe, by empirical and mathematical means, that the disadvantage of non-trivial population sizes is much less pronounced for this problem.

The $(1 + 1)$ EA with fixed mutation probability p has an expected optimization time of $\frac{1}{2p^2}((1-p)^{-n+1} - (1-p)) + 1$ on LeadingOnes [21], which is minimized for $p \approx 1.59/n$. This choice gives an expected runtime of about $0.77n^2$. A fitness-dependent mutation rate can decrease this runtime further to around $0.68n^2$ [21]. For the $(1+1)$ EA$_{>0}$, it has been observed in [91] that its expected optimization time decreases with decreasing p. More precisely, it equals $\frac{1-(1-p)^n}{2p^2}((1-p)^{-n+1} - (1-p)) + 1$, which converges to $n^2/2 + 1$ for $p \to 0$.

For $\lambda = n^{O(1)}$, the expected optimization time of the $(1+\lambda)$ EA with mutation rate $p = 1/n$ is $O(n^2 + n\lambda)$ [87]. With this mutation rate, the adaptive $(1+\{2\lambda, \lfloor\lambda/2\rfloor\})$ EA and the $(1 + \{2\lambda, 1\})$ EA achieve an expected optimization time of $O(n^2)$ [104]. Any $(1+\lambda)$ EA variant with fixed offspring population size λ but possibly adaptive mutation rate p needs at least $\Omega(\frac{\lambda n}{\ln(\lambda/n)} + n^2)$ function evaluations, on average, to optimize LeadingOnes [8].

Recall that the bounds reported above are for the classic $(1 + \lambda)$ EA variants, not the resampling versions.

This section presents a refining of the bound for the $(1+\lambda)$ EA, and we first present the empirical results that have motivated this analysis.

Empirical Results Similarly to the data plotted in Figure 4.1, we show in Figure 4.3 the normalized average optimization times of the different algorithms; the normalization factor is n^2.

For all $(1 + \lambda)$ EA variants, a fairly stable performance across the three tested dimensions $n = 500$, $n = 1,000$, and $n = 1,500$ can be observed. We also see that the adaptive algorithms seem to perform worse than the ones with static parameter values; we will address this point in more detail below.

Another interesting observation is that the value of λ does not seem to have a significant impact on the expected performance. This is in sharp contrast to the situation for OneMax, cf. our discussion in the second half of Section 4.1.3. Building on our discussion there, we can explain this phenomenon as follows. Unlike for OneMax, the situation for LeadingOnes is that the expected fitness gain of a random offspring created by a $(1 + \lambda)$ EA variant with static mutation rate $p = c/n$ is very small throughout the whole optimization process. More precisely, it decreases only mildly from around $2c/n$ when $\text{LO}(x) = 0$ to $2c(1-c/n)^{n-1}/n \approx 2c/(e^c n)$ for $\text{LO}(x) = n-1$. Thus, intuitively, the whole optimization process of LeadingOnes is very similar to the last steps of the OneMax optimization.

Figure 4.4 presents the average fixed-target runtimes for selected $(1 + \lambda)$ $\text{EA}_{>0}$ variants on the $1,500$-dimensional LeadingOnes problem. We add to this figure the fixed target runtime of Randomized Local Search (RLS), the greedy (1+1)-type hill climber that always flips one random bit per iteration. RLS has a constant expected fitness gain of $2/n$ on LeadingOnes and thus a total expected optimization time of $n^2/2 + 1$.

We observe that the performance of the $(1+1)$ $\text{EA}_{>0}$ and that of the $(1+50)$ $\text{EA}_{>0}$ are indeed very similar throughout the optimization process. The curves for the $(1 + \lambda)$ $\text{EA}_{>0}$ with $\lambda = 2, 5, 10$ were indistinguishable in this plot and are therefore not shown in the figure. We also see that their fixed-target performance is better than that of RLS for all LO-values up to around $1,250 \approx 0.42n$ (exact empirical values are $1,227$ for the $(1 + 50)$ $\text{EA}_{>0}$ and $1,283$ for the $(1 + 1)$ $\text{EA}_{>0}$, but we recall that such numbers should be taken with care as they represent an average of 100 runs only. It should not be very difficult to compute the cutting point precisely, by mathematical means).

Since the only difference between the $(1 + 1)$ $\text{EA}_{>0}$ and RLS is the distribution from which new offspring are sampled, we see that the $(1 + \lambda)$ EA variants profit from iterations in which more than one bit are flipped in the beginning of the optimization process, while they suffer from this same effect in the later parts. This situation is

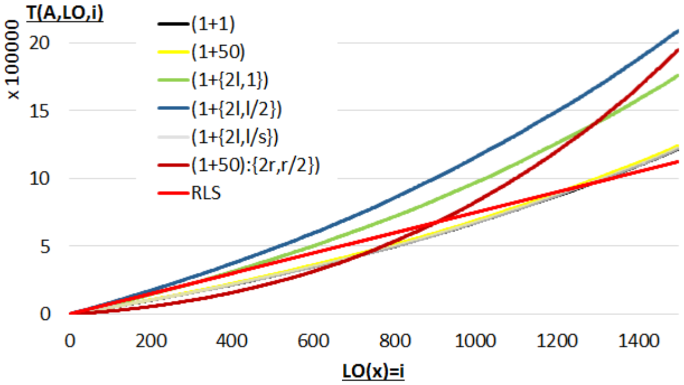


Figure 4.4: Fixed-target runtimes of the $(1 + \lambda)$ EA$_{>0}$ variants on the $1,500$-dimensional LeadingOnes problem

even more pronounced in the $(1 + \lambda)$ EA$_{r/2,2r}$, which outperforms all other tested algorithms for target values up to 709. Its performance then suffers from creating at least half of its offspring with a too large mutation rate. Recall that even if the algorithm has correctly identified the optimal mutation rate $p(\text{LO}(x))$, it still creates half of its offspring with mutation rate $2p(\text{LO}(x))$. This results in a mediocre overall performance. This observation certainly raises the question of how to adjust the structure of the $(1 + \lambda)$ EA$_{r/2,2r}$ to benefit from its good initial performance. From the viewpoint of hyper-heuristics, or algorithm selection, an adaptive selection between the $(1 + \lambda)$ EA$_{r/2,2r}$ and the $(1 + \lambda)$ EA$_{>0}$ would be desirable.

If we had looked only at the total optimization times, we would have classified the $(1 + \lambda)$ EA$_{r/2,2r}$ as being inefficient. The fixed-target results, however, nicely demonstrate that despite the poor overall performance, there is something to be learned from this algorithm. This emphasizes the need for a fine-grained benchmarking environment, similar to what is done in continuous black-box optimization (where fixed-target and fixed-budget considerations are a necessary standard since the algorithms cannot identify an optimal solution but only get arbitrarily close to it).

Precise Bounds for LeadingOnes We have observed that, for LeadingOnes, the runtimes of the $(1 + \lambda)$ EA$_{>0}$ variants with static parameter choices are very close to that of the $(1 + 1)$ EA$_{>0}$. Theorem 1 shows that for every constant λ, the expected optimization time of the $(1 + \lambda)$ EA$_{>0}$ converges from above against that of the $(1+1)$ EA$_{>0}$ (and the same holds for the $(1+\lambda)$ EA and $(1+1)$ EA, respectively). Theorem 1 can be proven by adjusting the proofs in [21] to the $(1+\lambda)$ EA. An important ingredient in this analysis is the observation that the probability of making progress in one *generation* with parent individual x equals $1-(1-p(1-p)^{\mathrm{LO}(x)})^{\lambda}$, i.e., 1 minus the probability that none of the λ offspring is better. Recall that in order to create a better offspring, none of the first $\mathrm{LO}(x)$ bits should flip, while the $(\mathrm{LO}(x)+1)$-st bit *does* need to be flipped. The results for the $(1+\lambda)$ EA$_{>0}$ can be obtained from that for the $(1+\lambda)$ EA by taking into account that the re-sampling strategy increases the expected fitness gain by a multiplicative factor of $1/(1-(1-p)^n)$. Therefore, we can derive the following:

Theorem 1. *For all $n, \lambda \in \mathbb{N}$ the expected optimization time of the $(1+\lambda)$ EA with static mutation rate $0 < p < 1$ on the n-dimensional* LeadingOnes *function is at most*

$$1+\frac{\lambda}{2}\sum_{j=0}^{n-1}\frac{1}{1-(1-p(1-p)^{j})^{\lambda}} \tag{4.1}$$

and the expected optimization time of the $(1+\lambda)$ EA$_{>0}$ is at most

$$1+\frac{(1-(1-p)^{n})\lambda}{2}\sum_{j=0}^{n-1}\frac{1}{1-(1-p(1-p)^{j})^{\lambda}}. \tag{4.2}$$

To judge the precision of the bound stated in Theorem 1, we first note that for $\lambda = 1$ expression (4.1) is tight by the result presented in [21] (note though that the additive +1 term is suppressed there as they regard the number of iterations, not function evaluations). Similarly, for the $(1+1)$ EA$_{>0}$ expression (4.2) is tight by the bound proven in [91].

Apart from this case with trivial offspring population size $\lambda = 1$, it might be tedious to compute the expected optimization time of the $(1+\lambda)$ EA on LeadingOnes exactly, since in our proof for Theorem 1 we would have to take into account that in one generation more than one search point that improves upon the current best search point can be generated. Since the $(1+\lambda)$ EA chooses the best one of these, the distribution of this offspring would have to be computed. Note, however, that this effect can only have a very mild impact on the bounds stated above, as it occurs relatively rarely and does, in general, not result in a much larger fitness gain. Put differently, the bounds in Theorem 1 are close to tight for reasonable (i.e., not too large) values of λ.

As the expressions in Theorem 1 are not easy to interpret, we provide in Table 4.1 a numerical evaluation of the upper bound (4.2) for different values of λ and n. We add to this table (second row) the empirically observed averages, which show a good match to the theoretical bound.

	500	1,000	1,500	10,000	100,000	500,000
(1+1)	54.317%	54.313%	54.311%	54.309%	54.308%	54.308%
emp.	54.0%	54.1%	54.2%	-	-	-
(1+2)	54.349%	54.328%	54.322%	54.310%	54.308%	54.308%
emp.	54.8%	54.4%	54.2%	-	-	-
(1+5)	54.444%	54.376%	54.353%	54.315%	54.309%	54.308%
emp.	54.5%	54.8%	54.6%	-	-	-
(1+50)	55.883%	55.091%	54.829%	54.386%	54.316%	54.310%
emp.	57.6%	55.3%	55.2%	-	-	-

Table 4.1: Theoretical upper bounds from Theorem 1 and empirical optimization times for the $(1+\lambda)$ EA$_{>0}$

4.1.5 Summary

The work profiling $(1+\lambda)$ EA provids an example of how benchmarking studies associate theoretical analysis and empirical studies. In practice, the results shows that the value of λ significantly affects the performance of (1+1) EA for OneMax. However, we did not observe such effect for LeadingOnes. Moreover, the result inspired a refined analysis of the expected optimization time of the $(1+\lambda)$ EA on LeadingOnes [56].

4.2 Interpolating Local and Global Search by Controlling the Variance of Standard Bit Mutation

Recall that the probability of using the standard bit mutation to sample a specific offspring y at distance $0 \leq d \leq n$ from x thus equals $p^{H(x,y)}(1-p)^{H(x,y)}$, where $H(x,y) = |\{1 \leq i \leq n \mid x_i \neq y_i\}|$ denotes the Hamming distance of x and y. This probability is strictly positive for all y, thus showing that the probability that an EA using standard bit mutation will have sampled a global optimum of f converges to one as the number of iterations increases. In contrast to pure random search, however, the distance at which the offspring y is sampled follows a binomial distribution, $\text{Bin}(n,p)$, and is thus concentrated around its mean np.

The ability to escape local optima comes at the price of frequent uses of non-optimal search radii even in those regimes in which the latter are stable for a long time. The incapability of standard bit mutation to adjust to such situations results in important performance losses on almost all classical benchmark functions, which often exhibit large parts of the optimization process in which flipping a certain number of bits is required. A convenient way to control the degree of randomness in the choice of the search radius would therefore be highly desirable.

In this section we introduce such an interpolation. It allows to calibrate between deterministic and pure random search, while encompassing standard bit mutation as one specification. More precisely, we investigate *normalized standard bit mutation*, in which the mutation strength (i.e., the search radius) is sampled from a normal distribution $N(\mu, \sigma^2)$. By choosing $\sigma = 0$ one obtains a deterministic choice, and the "degree of randomness" increases with increasing σ. By the central limit theorem, we recover a distribution that is very similar to that of standard bit mutation by setting $\mu = np$ and $\sigma^2 = np(1-p)$.

Apart from conceptual advantages, normalized standard bit mutation offers the advantage of separating the variance from the mean, which makes it easy to control both parameters independently during the optimization process. While multi-dimensional parameter control for discrete EAs is still in its infancy, cf. comments in [92, 42], we demonstrate in this work a simple, yet efficient way to control mean and variance of normalized standard bit mutation.

4.2.1 Background

In Section 4.1, we observed that the EA with success-based self-adjusting mutation rate proposed in [46] outperforms the $(1+\lambda)$ EA for a large range of sub-optimal targets. It then drastically looses performance in the later parts of the optimization process, which results in an overall poor optimization time on OneMax and LeadingOnes functions of moderate problem dimensions $n \leq 10,000$. The proven optimal asymptotic behavior on OneMax in [46] can thus not be observed for these dimensions.

The algorithm from [46], which we named $(1+\lambda)$ EA$_{r/2,2r}$, has been mentioned in Section 4.1.2. The details are presented in Algorithm 3. It is a $(1+\lambda)$ EA which applies in each iteration two different mutation rates. Half of the offspring population is generated with mutation rate $r/(2n)$, the other half with mutation rate $2r/n$. The parameter r is the current best mutation strength, which is updated after each iteration, with a bias towards the rate by which the best of the λ offspring has been sampled.

Recall that we apply the standard bit mutation by first sampling a radius ℓ from the binomial distribution $\text{Bin}(n,p)$ and then applying the flip$_\ell$ operator, which flips

Algorithm 3: The 2-rate $(1 + \lambda)$ $\text{EA}_{r/2,2r}$ with adaptive mutation rates proposed in [46]

```
Initialization: Sample $x \in \{0,1\}^n$ uniformly at random and evaluate $f(x)$;
Initialize $r \leftarrow r^{\text{init}}$; // Following [46] we use $r^{\text{init}} = 2$;
Optimization: for $t = 1, 2, 3, \ldots$ do
    for $i = 1, \ldots, \lambda/2$ do
        Sample $\ell^{(i)} \sim \text{Bin}_{>0}(n, r/(2n))$, create $y^{(i)} \leftarrow \text{flip}_{\ell^{(i)}}(x)$, and evaluate $f(y^{(i)})$;
    for $i = \lambda/2 + 1, \ldots, \lambda$ do
        Sample $\ell^{(i)} \sim \text{Bin}_{>0}(n, 2r/n)$, create $y^{(i)} \leftarrow \text{flip}_{\ell^{(i)}}(x)$, and evaluate $f(y^{(i)})$;
    $x^* \leftarrow \arg\max\{f(y^{(1)}), \ldots, f(y^{(\lambda)})\}$ (ties broken u.a.r.);
    if $f(x^*) \geq f(x)$ then $x \leftarrow x^*$;
    if $x^*$ has been created with mutation rate $r/2$ then $s \leftarrow 3/4$ else $s \leftarrow 1/4$;
    Sample $q \in [0,1]$ u.a.r.;
    if $q \leq s$ then $r \leftarrow \max\{r/2, 2\}$ else $r \leftarrow \min\{2r, n/4\}$;
```

ℓ pairwise different bits that are chosen from the index set $[n]$ uniformly at random. Note that we still apply the *re-sampling strategy* enforcing that all offspring differ from their parents by at least one bit, which is achieved by sampling ℓ from the conditional binomial distribution $\text{Bin}_{>0}(n, p)$.

In Section 4.1.2, we compared the fixed-target performance of the $(1 + 50)$ $\text{EA}_{>0}$ (i.e., the $(1 + \lambda)$ EA using the conditional sampling rule introduced above) and the $(1 + 50)$ $\text{EA}_{r/2,2r}$ on OneMax and LeadingOnes. In Figure 4.5 we report similar empirical results for $n = 10,000$ (OneMax) and $n = 2,000$ (LeadingOnes) (the other results in the two figures will be addressed below). We also observed in Section 4.1 that for both functions the $(1+50)$ $\text{EA}_{r/2,2r}$ from [46] performs well for small target values, but drastically looses performance in the later stages of the optimization process.

Properties of OneMax and LeadingOnes

Both OneMax and LeadingOnes have a long period during the optimization run in which flipping one bit is optimal.

For OneMax flipping one bit is widely assumed to be optimal as soon as $f(x) \geq 2n/3$. Quite interestingly, however, this conjecture has not not been rigorously proven to date. It is only known that drift-maximizing (i.e., maximizing the expected fitness gain over the best-so-far individual) mutation strengths are *almost* optimal [45], in

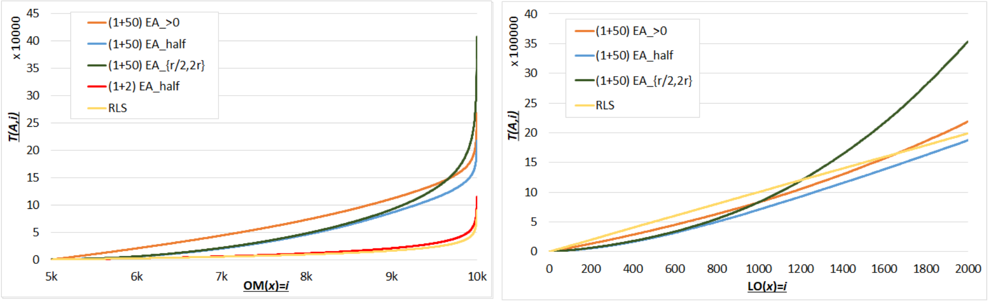


Figure 4.5: Average fixed-target running times for variants of the 2-rate $(1+50)$ EA for $10,000$-dimensional OneMax and $2,000$-dimensional LeadingOnes.

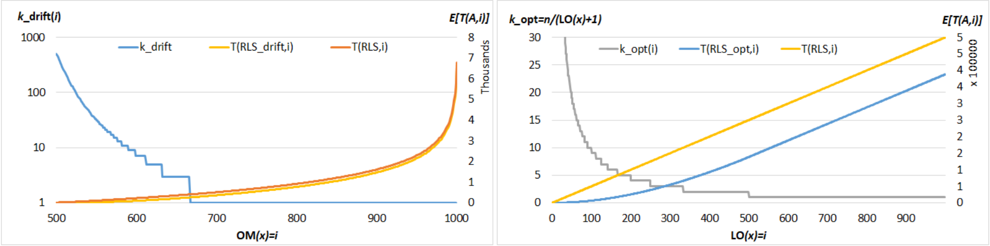


Figure 4.6: Drift maximizing and optimal mutation strength for $1,000$-dimensional OneMax and LeadingOnes functions, respectively. Note the logarithmic scale for k_{drift} for OneMax. For OneMax, RLS spends around 94% of the total optimization time in the regime in which $k_{\text{drift}} = 1$, for LeadingOnes this fraction is still 50%. For the drift-maximizing/optimal RLS-variants flipping in each iteration k_{drift} and k_{opt} bits, respectively, these fractions are around 96% for OneMax and 64% for LeadingOnes.

the sense that the overall expected optimization time of the elitist (1+1) algorithm using these rates in each step cannot be worse than the best-possible unary unbiased algorithm for OneMax by more than an additive $o(n)$ lower order term [45]. But even for the drift maximizer the statement that flipping one bit is optimal when $f(x) \geq 2n/3$ has only be shown for an approximation, not the actual drift maximizer. Numerical evaluations for problem dimensions up to $10,000$ nevertheless confirm that 1-bit flips are optimal when the OneMax-value exceeds $2n/3$.

For LeadingOnes, on the other hand, it is well known that flipping one bit is optimal as soon as $f(x) \geq n/2$ [38].

We display in Figure 4.6, which is adjusted from [53], the optimal and drift-maximizing mutation strength for LeadingOnes and OneMax, respectively. We also display in the same figure the expected time needed by RLS_{opt} and $\text{RLS}_{\text{drift}}$, the elitist (1+1) algorithm using in each step these mutation rates. We see that these algorithms spend around 96% (for OneMax) and 64% (for LeadingOnes), respectively, of their time in the regime where flipping one bit is (almost) optimal. These numbers are based on an exact computation for LeadingOnes and on an empirical evaluation of 500 independent runs for OneMax.

Implications for the $(1+50)$ $\mathbf{EA}_{r/2,2r}$

Assume that in the regime of optimal one-bit flips the $(1+50)$ $\text{EA}_{r/2,2r}$ has correctly identified that flipping one bit is optimal. It will hence use the smallest possible value for r, which is 2. In this case, half the offspring are sampled with the (for this algorithm optimal) mutation rate $1/n$, while the other half of the offspring population is sampled with mutation rate $4/n$, thus flipping on average more than four times the optimal number of bits. It is therefore non-surprising that in this regime (and already before) the gradient of the average fixed-target running time curves in Figures 4.5 are much worse for the $(1+50)$ $\text{EA}_{r/2,2r}$ than for the $(1+50)$ $\text{EA}_{>0}$.

4.2.2 Creating Half the Offspring with Optimal Mutation Rate

The observations made in the last section inspire the design of the $(1+\lambda)$ $\text{EA}_{r,U(0,\sigma r/n)}$ defined in Algorithm 4. This algorithm samples half the offspring using as deterministic mutation strength the best mutation strength of the last iteration. The other offspring are sampled with a mutation rate that is sampled uniformly at random from the interval $(0, \sigma r/n)$.

As we can see in Figure 4.5 this algorithm significantly improves the performance in those later parts of the optimization process. Normalized total optimization times for various problem dimensions are provided in Figures 4.7 and 4.8, respectively. We display data for $\sigma = 2$ only, and call this $(1+\lambda)$ $\text{EA}_{r,U(0,\sigma r/n)}$ variant $(1+\lambda)$ EA_{half}. We note that smaller values of σ, e.g., $\sigma = 1.5$ would give better results. The same effect would be observable when replacing the factor two in the $(1+\lambda)$ $\text{EA}_{r/(2n),2r}$, i.e., when using a $(1+\lambda)$ $\text{EA}_{r/(\sigma n),\sigma r}$ rule with $\sigma \neq 2$.

It is remarkable that on LeadingOnes the $(1+\lambda)$ EA_{half} performs better than RLS, the elitist (1+1) algorithm flipping in each iteration exactly one uniformly chosen bit. The slightly worse gradients for target values $v > n/2$ (which are a consequence of randomly sampling the mutation rate instead of using mutation strength one deterministically) are compensated for by the gains made in the initial phase of the

Algorithm 4: The $(1+\lambda)$ EA$_{r,U(0,\sigma r/n)}$. In line 6 we denote by $U(a,b)$ the uniform distribution in the interval (a,b). For $\sigma = 2$ we call this algorithm the $(1+\lambda)$ EA$_{\text{half}}$.

```
1  Initialization: Sample $x \in \{0,1\}^n$ uniformly at random and evaluate $f(x)$;
2  Initialize $r \leftarrow r^{\text{init}}$; // we use $r^{\text{init}} = 2$;
3  Optimization: for $t = 1, 2, 3, \ldots$ do
4      for $i = 1, \ldots, \lambda/2$ do
5          Set $\ell^{(i)} \leftarrow r$, create $y^{(i)} \leftarrow \text{flip}_{\ell^{(i)}}(x)$, and evaluate $f(y^{(i)})$;
6      for $i = \lambda/2 + 1, \ldots, \lambda$ do
7          Sample $p^{(i)} \sim \min\{U(0, \sigma r/n), 1\}$, $\ell^{(i)} \sim \text{Bin}_{>0}(n, p^{(i)})$, create
           $y^{(i)} \leftarrow \text{flip}_{\ell^{(i)}}(x)$, and evaluate $f(y^{(i)})$;
8      $i \leftarrow \min\{j \mid f(y^{(j)}) = \max\{f(y^{(k)}) \mid k \in [n]\}\}$;
9      $r \leftarrow \ell^{(i)}$;
10     if $f(y^{(i)}) \geq f(x)$ then $x \leftarrow y^{(i)}$;
```

optimization process, where the EA variants benefit from larger mutation rates.

On OneMax the performance of the $(1+\lambda)$ EA$_{\text{half}}$ is better than that of the plain $(1+\lambda)$ EA$_{>0}$ for both tested values $\lambda = 50$ and $\lambda = 2$.

We recall that it is well known that, both for OneMax and LeadingOnes, the optimal offspring population size in the regular $(1+\lambda)$ EA is $\lambda = 1$ [87]. A monotonic dependence of the average optimization time on λ is conjectured (and empirically observed) but not formally proven. While for OneMax the impact of λ is significant, the dependency on λ is much less pronounced for LeadingOnes. Empirical results for both functions and a theoretical running time analysis for LeadingOnes can be found in [56]. For OneMax [68] offers a precise running time analysis of the $(1+\lambda)$ EA for broad ranges of offspring population sizes λ and mutation rates $p = c/n$. In light of the fact that the theoretical considerations in [46] required $\lambda = \omega(1)$, it is worthwhile to note that for all tested problem dimensions the $(1+2)$ EA$_{r/2,2r}$ performs better on OneMax than the $(1+50)$ EA$_{r/2,2r}$.

4.2.3 Normalized Standard Bit Mutation

In light of the results presented in the previous section, one may wonder if splitting the population into two halves is needed after all. We investigate this question by introducing the $(1+\lambda)$ EA$_{\text{norm.}}$ which in each iteration and for each $i \in [\lambda]$ samples the mutation strength $\ell^{(i)}$ from the normal distribution $N(r, r(1-r/n))$ around the best mutation strength r of the previous iteration and rounding the sampled value to the closest integer. The reasons to replace the uniform distribution $U(r/n-\sigma, r/n+\sigma)$ will be addressed below. As before we enforce $\ell^{(i)} \geq 1$ by re-sampling if needed, thus effectively sampling the mutation strength from the conditional distribution $N_{>0}(r, r(1-r/n))$. Algorithm 5 summarizes this algorithm.

Algorithm 5: The $(1+\lambda)$ $\mathrm{EA}_{\mathrm{norm.}}$ with normalized standard bit mutation

```
1 **Initialization:** Sample $x \in \{0,1\}^n$ uniformly at random and evaluate $f(x)$;
2 Initialize $r \leftarrow r^{\text{init}}$; // we use $r^{\text{init}} = 2$;
3 **Optimization:** **for** $t = 1, 2, 3, \ldots$ **do**
4     **for** $i = 1, \ldots, \lambda$ **do**
5         Sample $\ell^{(i)} \sim \min\{N_{>0}(r, r(1-r/n)), n\}$, create $y^{(i)} \leftarrow \mathrm{flip}_{\ell^{(i)}}(x)$, and
           evaluate $f(y^{(i)})$;
6     $i \leftarrow \min\{j \mid f(y^{(j)}) = \max\{f(y^{(k)}) \mid k \in [n]\}\}$;
7     $r \leftarrow \ell^{(i)}$;
8     **if** $f(y^{(i)}) \geq f(x)$ **then** $x \leftarrow y^{(i)}$;
```

Note that the variance $r(1-r/n)$ of the unconditional normal distribution $N(r, r(1-r/n))$ is identical to that of the unconditional binomial distribution $\mathrm{Bin}(n, r/n)$. We use the normal distribution here for reasons that will be explained in the next section. Note, however, that very similar results would be obtained when replacing in line 4 of Algorithm 5 the normal distribution $N_{>0}(r, r(1-r/n))$ by the binomial one $\mathrm{Bin}_{>0}(n, r/n)$. We briefly recall that, by the central limit theorem, the (unconditional) binomial distribution converges to the (unconditional) normal distribution.

The empirical performance of the $(1+50)$ $\mathrm{EA}_{\mathrm{norm.}}$ is comparable to that of the $(1+50)$ $\mathrm{EA}_{\mathrm{half}}$ for both problems and all tested problem dimensions, cf. Figures 4.7 and 4.8. Note, however, that for $\lambda = 2$ the $(1+2)$ $\mathrm{EA}_{\mathrm{norm.}}$ performs worse than the $(1+2)$ $\mathrm{EA}_{\mathrm{half}}$.

4.2.4 Interpolating Local and Global Search

As discussed above, all EA variants mentioned so far suffer from the variance of the random selection of the mutation rate, in particular in the long final part of the optimization process in which the optimal mutation strength is one. We therefore analyze a simple way to reduce this variance on the fly. To this end, we build upon the $(1+\lambda)$ $\mathrm{EA}_{\mathrm{norm.}}$ and introduce a counter c, which is initialized at zero. In each iteration, we check if the value of r changes. If so, the counter is re-set to zero. It

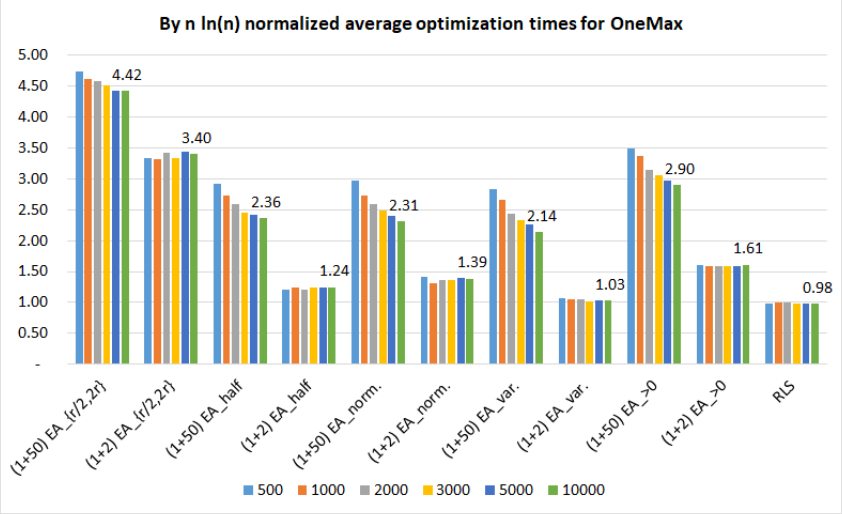


Figure 4.7: By $n \ln(n)$ normalized average optimization times for OneMax, for n between 500 and 10,000. Displayed numbers are for $n = 10,000$.

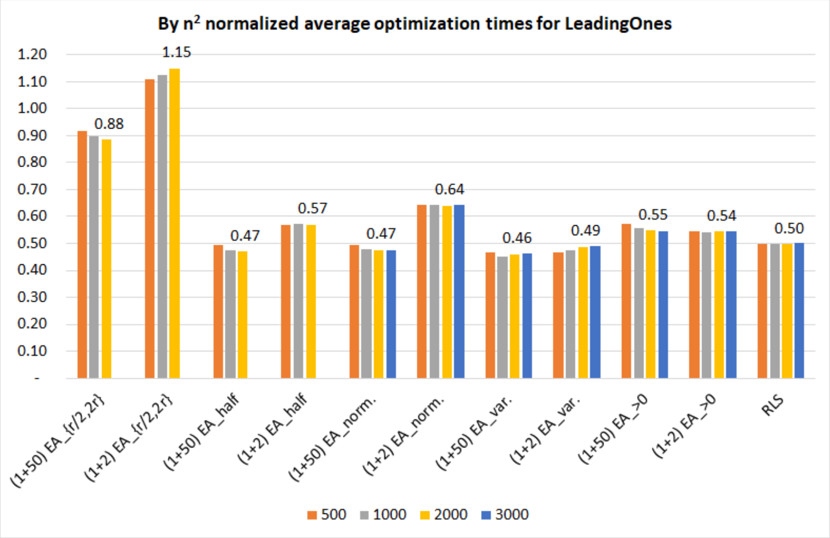


Figure 4.8: By n^2 normalized average optimization times for LeadingOnes, for n between 500 and 3,000. Displayed numbers are for $n = 2,000$.

is increased by one otherwise, i.e., if the value of r remains the same. We use this counter to self-adjust the variance of the normal distribution. To this end, we replace in line 4 of Algorithm 5 the conditional normal distribution $N_{>0}(r, r(1 - r/n))$ by the conditional normal distribution $N_{>0}(r, F^c r(1 - r/n))$, where $F < 1$ is a constant discount factor. Algorithm 6 summarizes this $(1 + \lambda)$ EA variant with normalized standard bit mutation and a self-adjusting choice of mean and variance.

Algorithm 6: The $(1+\lambda)$ $\text{EA}_{\text{var.}}$ with normalized standard bit mutation and a self-adjusting choice of mean and variance

```
1  Initialization: Sample $x \in \{0,1\}^n$ uniformly at random and evaluate $f(x)$;
2  Initialize $r \leftarrow r^{\text{init}}$; // we use $r^{\text{init}} = 2$;
3  Initialize $c \leftarrow 0$;
4  Optimization: for $t = 1, 2, 3, \ldots$ do
5      for $i = 1, \ldots, \lambda$ do
6          Sample $\ell^{(i)} \sim \min\{N_{>0}(r, F^c r(1 - r/n)), n\}$, create $y^{(i)} \leftarrow \text{flip}_{\ell^{(i)}}(x)$,
             and evaluate $f(y^{(i)})$;
7      $i \leftarrow \min\{j \mid f(y^{(j)}) = \max\{f(y^{(k)}) \mid k \in [n]\}\}$;
8      if $r = \ell^{(i)}$ then $c \leftarrow c + 1$; else $c \leftarrow 0$;
9      $r \leftarrow \ell^{(i)}$;
10     if $f(y^{(i)}) \geq f(x)$ then $x \leftarrow y^{(i)}$;
```

Choice of F: We use $F = 0.98$ in all reported experiments. Preliminary tests suggest that values $F < 0.95$ are not advisable, since the algorithm may get stuck with sub-optimal mutation rates. This could be avoided by introducing a lower bound for the variance and/or by mechanisms taking into account whether or not an iteration has been *successful*, i.e., whether it has produced a strictly better offspring.

The empirical comparison suggests that the self-adjusting choice of the variance in the $(1 + \lambda)$ $\text{EA}_{\text{var.}}$ improves the performance on OneMax further, cf. also Figure 4.7 for average fixed-target results for $n = 10,000$. For $\lambda = 2$ the average performance is comparable to, but slightly worse than that of RLS. For LeadingOnes, the $(1 + 50)$ $\text{EA}_{\text{var.}}$ is comparable in performance to the $(1 + 50)$ $\text{EA}_{\text{norm.}}$, but we observe that for $\lambda = 2$ the $(1 + \lambda)$ $\text{EA}_{\text{var.}}$ performs better. It is the only one among all tested EAs for which decreasing λ from 50 to 2 does not result in a significantly increased running time.

4.2.5 A Meta-Algorithm with Normalized Standard Bit Mutation

In the $(1 + \lambda)$ $\text{EA}_{\text{var.}}$ we make use of the fact that a small variance in line 6 of Algorithm 6 results in a more concentrated distribution. The variance adjustment is thus an efficient way to steer the *degree of randomness* in the selection of the mutation

Algorithm 7: The $(1+\lambda)$ Meta-Algorithm with (static) normalized standard bit mutation. The RLS variant with deterministic search radius r and $(1+\lambda)$ EA using standard bit mutation with mutation rate r/n are identical to this algorithm with $\sigma^2 = 0$ and $\sigma^2 = r(1-r/n)$, respectively.

```
1 Initialization: Sample $x \in \{0,1\}^n$ uniformly at random and evaluate $f(x)$;
2 Optimization: for $t = 1, 2, 3, \dots$ do
3     for $i = 1, \dots, \lambda$ do
4         Sample $\ell^{(i)} \sim \min\{N_{>0}(r, \sigma^2), n\}$, create $y^{(i)} \leftarrow \text{flip}_{\ell^{(i)}}(x)$, and evaluate
            $f(y^{(i)})$; // $r$ and $\sigma$ are two parameters;
5     $y \leftarrow \arg\max\{f(y^{(k)}) \mid k \in [n]\}$;
6     if $f(y) \geq f(x)$ then $x \leftarrow y$;
```

rate. It allows to interpolate between deterministic and random mutation rates. In our experimentation we do not go beyond the variance of the binomial distribution, but in principle there is no reason to not regard larger variance as well. The question of how to best determine the degree of randomness in the choice of the mutation rate has, to the best of our knowledge, not previously been addressed in the EC literature. We believe that this idea carries good potential, since it demonstrates that local search with its deterministic search radius and evolutionary algorithms with their global search radii are merely two different configurations of the same meta-algorithm, and not two different algorithms as the general perception might indicate. To make this point very explicit, we introduce with Algorithm 7 a general meta-algorithm, of which local search with deterministic mutation strengths and EAs are special instantiations.

Note that in this meta-model we use static parameter values, variants with adaptive mutation rates can be obtained by applying the usual parameter control techniques, as demonstrated above. Of course, the same normalization can be done for similar EAs, the technique is not restricted to elitist $(1+\lambda)$-type algorithms. Likewise, the condition to flip at least one bit can be omitted, i.e., one can replace the conditional normal distribution $N_{>0}(r, \sigma^2)$ in line 3 by the unconditional $N(r, \sigma^2)$.

4.2.6 Summary

In this section, we introduced the *normalized standard bit mutation*, which replaces the binomial choice of the mutation strength in standard bit mutation by a normal distribution [165]. This normalization allows a straightforward way to control the variance of the distribution, which can now be adjusted independently of the mean. We have demonstrated that such an approach can be beneficial when optimizing classic benchmark problems such as OneMax and LeadingOnes.

We also note that the parameter control technique which we applied to adjust the mean of the sampling distribution for the mutation strength has an extremely short learning period, since we simply use the best mutation strength of the last iteration as mean for the sampling distribution of the next iteration. For more rugged fitness landscapes a proper learning, which takes into account several iterations, should be preferable. We recall that multi-dimensional parameter control has not received much attention inthe EC literature for discrete optimization problems [42, 92]. Our work falls into this category, and we have demonstrated a simple way to separate the control of the mean from that of the variance of the mutation strength distribution. In Chapter 6, we will introduce more work on *hyperparameter optimization*.

Finally, the meta-algorithm presented in Section 4.2.5 demonstrates that Randomized Local Search and evolutionary algorithms can be seen as two configurations of the meta-algorithm. Parameter control, or, in this context possibly more suitably referred to as online algorithm configuration, offers the possibility to interpolate between these algorithms (and even more drastically, randomized heuristics). Given the significant advances in the context of algorithm configuration witnessed by the EC and machine learning communities, we believe that such meta-models carry significant potential to exploit and profit from advantages of different heuristics. Note here that the configuration of meta-algorithms offers much more flexibility than the algorithm selection approach classically taken in EC, e.g., in most works on hyper-heuristics. Related work *algorithm selection* will also be discussed in Chapter 7.

4.3 The Impact of Crossover Probability for GA

4.3.1 Background

In this section, we look closely into the performance of a $(\mu + \lambda)$ GA on LeadingOnes. The work of this section is motivated by the investigation of the effectiveness of mutation and crossover, more details will be introduced in Section 5.2.

We observe some very interesting effects, that we believe may motivate the theory community to look at the question of usefulness of crossover from a different angle. More precisely, we find that, against our intuition that uniform crossover cannot be beneficial on LeadingOnes, the performance of the $(\mu + \lambda)$ GA on LeadingOnes improves when p_c takes values greater than 0 (and smaller than 1), see Figure 4.9. The performances are quite consistent, and we can observe clear patterns, such as a tendency for the optimal value of p_c (displayed in Table 4.2) to increase with increasing μ, and to decrease with increasing problem dimension. The latter effect may explain why it is so difficult to observe benefits of crossover in theoretical work: they disappear with the asymptotic view that is generally adopted in runtime analysis.

We have also performed similar experiments on OneMax (see our data [171]), but the good performance of the $(\mu+\lambda)$ GA configurations using crossover is less surprising for this problem, since this benefit has previously been observed for genetic algorithms that are very similar to the $(\mu + \lambda)$ GA; see [25, 29, 30, 143] for examples and further references. In contrast to a large body of literature on the benefit of crossover for solving OneMax, we are not aware of the existence of such results for LeadingOnes, apart from the highly problem-specific algorithms that were developed and analyzed in [1, 52].

4.3.2 A Family of $(\mu + \lambda)$ Genetic Algorithms

We investigate a meta-model, which allows us to easily transition from a mutation-only to a crossover-only algorithm. Algorithm 8 presents this framework, which, for ease of notation, we refer to as the family of the $(\mu + \lambda)$ GA in the following.

The $(\mu + \lambda)$ GA initializes its population uniformly at random (u.a.r., lines 1-2). In each iteration, it creates λ offspring (lines 6–16). For each offspring, we first decide whether to apply crossover (with probability p_c, lines 8–11) or whether to apply mutation (otherwise, lines 12–15). Offspring that differ from their parents are evaluated, whereas offspring identical to one of their parents inherit this fitness value without function evaluation (see [25] for a discussion). The best μ of parent and offspring individuals form the new parent population of the next generation (line 17).

Note the unconventional use of *either* crossover *or* mutation. As mentioned, we consider this variant to allow for a better attribution of the effects to each of the operators. Moreover, note that in Algorithm 8 we decide for each offspring individually which operator to apply. We call this scheme the $(\mu + \lambda)$ **GA with offspring-based variator choice**. We also study the performance of the $(\mu+\lambda)$ **GA with population-based variator choice**, which is the algorithm that we obtain from Algorithm 8 by swapping lines 7 and 6.

4.3.3 Experimental Results

Before we go into the details of the experimental setup and our results, we recall that for the optimization of LeadingOnes, the fitness values only depend on the first bits,

Algorithm 8: A Family of $(\mu + \lambda)$ Genetic Algorithms

```
Input: Population sizes $\mu$, $\lambda$, crossover probability $p_c$, mutation rate $p$;
Initialization: for $i = 1, \ldots, \mu$ do sample $x^{(i)} \in \{0,1\}^n$ uniformly at random
  (u.a.r.), and evaluate $f(x^{(i)})$;
Set $P = \{x^{(1)}, x^{(2)}, \ldots, x^{(\mu)}\}$;
Optimization: for $t = 1, 2, 3, \ldots$ do
    $P' \leftarrow \emptyset$;
    for $i = 1, \ldots, \lambda$ do
        Sample $r \in [0,1]$ u.a.r.;
        if $r \leq p_c$ then
            select two individuals $x, y$ from $P$ u.a.r. (w/ replacement);
            $z^{(i)} \leftarrow$ Crossover$(x, y)$;
            if $z^{(i)} \notin \{x, y\}$ then evaluate $f(z^{(i)})$ else infer $f(z^{(i)})$ from parent;
        else
            select an individual $x$ from $P$ u.a.r.;
            $z^{(i)} \leftarrow$ Mutation$(x)$;
            if $z^{(i)} \neq x$ then evaluate $f(z^{(i)})$ else infer $f(z^{(i)})$ from parent;
        $P' \leftarrow P' \cup \{z^{(i)}\}$;
    $P$ is updated by the best $\mu$ points in $P \cup P'$ (ties broken u.a.r.);
```

whereas the tail is randomly distributed and has no influence on the selection. More precisely, a search point x with LeadingOnes-value $f(x)$ has the following structure: the first $f(x)$ bits are all 1, the $f(x)+1$st bit equals 0, and the entries in the tail (i.e., in positions $[f(x)+2..n]$) did not have any influence on the optimization process so far. For many algorithms, it can be shown that these tail bits are uniformly distributed, see [39] for an extended discussion.

Experimental Setup. We fix in this section the variator choice to the *offspring-based setting*. We test $(\mu + \lambda)$ GA with following parameter combinations on 100-dimensional LeadingOnes: $\mu \in \{2, 3, 5, 8, 10, 20, 30, \ldots, 100\}$ (14 values), $\lambda \in \{1, \mu\}$ (3 values), $p_c \in \{0.1k \mid k \in [0..9]\} \cup \{0.95\}$ (11 values), and mutation and crossover operators are fixed standard bit mutation with $p_m = 1/n$ and *uniform crossover*. For each of the settings listed there, we perform 100 independent runs, with a maximal budget of $5n^2$ each.

Overall Running Time. We first investigate the impact of the crossover probability on the average running time, i.e., on the average number of function evaluations that the algorithm performs until it evaluates the optimal solution for the first time. The results for the $(\mu + 1)$ and the $(\mu + \mu)$ GA using uniform crossover and standard

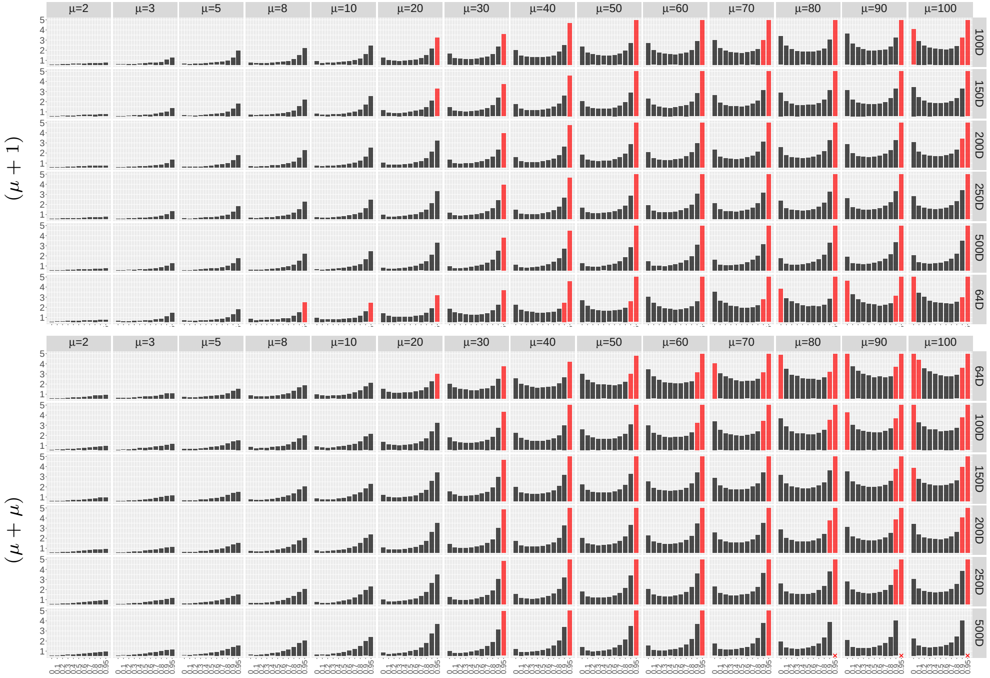


Figure 4.9: By n^2 normalized ERT values for the $(\mu+\lambda)$ GA using standard bit mutation and uniform crossover on LeadingOnes, for different values of μ and for $\lambda = 1$ (top) and for $\lambda = \mu$ (bottom). Results are grouped by the value of μ (main columns), by the crossover probability p_c (minor columns), and by the dimension (rows). The ERTs are computed from 100 independent runs for each setting, with a maximal budget of $5n^2$ fitness evaluations. ERTs for algorithms which successfully find the optimum in all 100 runs are depicted as black bars, whereas ERTs for algorithms with success rates in $(0, 1)$ are depicted as red bars. All bars are capped at 5.

bit mutation are summarized in Figure 4.9. Since not all algorithms managed to find the optimum within the given time budget, we plot as red bars the ERT values for such algorithms with success ratio strictly smaller than 1, whereas the black bars are reserved for algorithms with 100 successful runs. All values are normalized by n^2, to allow for a better comparison.

As a first observation, we note that the pattern of the results are quite regular. As can be expected, the dispersion of the running times is rather small. To give an impression for the concentration of the running times, we report that the standard deviation of the $(50 + 1)$ GA on the 100-dimensional LeadingOnes function is approximately 14% of the average running time across all values of p_c. As can be expected for a genetic algorithm on LeadingOnes, the average running time increases with increasing population size μ, see [142] for a proof of this statement when $p_c = 0$.

Next, we compare the sub-plots in each row, i.e., fixing the dimension. We see that the $(\mu + \lambda)$ GA suffers drastically from large p_c values when μ is smaller, suggesting that the crossover operator hinders performance. But as μ gets larger, the average running time at moderate crossover probabilities (p_c around 0.5) is significantly smaller than that in two extreme cases, $p_c = 0$ (mutation-only GAs), and $p_c = 0.95$. This observation holds for all dimensions and for both algorithm families, the $(\mu + 1)$ and the $(\mu + \mu)$ GA.

Looking at the sub-plots in each column (i.e., fixing the population size), we identify another trend: for those values of μ for which an advantage of $p_c > 0$ is visible for the smallest tested dimension, $n = 64$, the relative advantage of this rate decreases and eventually disappears as the dimension increases.

Finally, we compare the results of the $(\mu + 1)$ GA with those of the $(\mu + \mu)$ GA. Following [87], it is not surprising that for $p_c = 0$, the results of the $(\mu + 1)$ GA are better than those of the $(\mu + \mu)$ GA (very few exceptions to this rule exist in our data, but in all these cases the differences in average runtime are negligibly small), and following the theoretical analysis [56, Theorem 1], it is not surprising that the differences between these two algorithmic families are rather small: the typical disadvantage of the $(\mu + \lceil \mu/2 \rceil)$ GA over the $(\mu + 1)$ GA is around 5% and it is around 10% for the $(\mu + \mu)$ GA, but these relative values differ between the different configurations and dimensions.

Optimal Crossover Probabilities. To make our observations on the crossover probability clearer, we present in Table 4.2 a heatmap of the values p_c^* for which we observed the best average running time (with respect to all tested p_c values). We see the same trends here as mentioned above: as μ increases, the value of p_c^* increases, while, for fixed μ its value decreases with increasing problem dimension n. Here again we omit details for the $(\mu + \lceil \mu/2 \rceil)$ GA and for the fast mutation scheme, but the patterns are identical, with very similar absolute values.

Fixed-Target Running Times. We now study where the advantage of the crossover-based algorithms stems from. We demonstrate this using the example of the $(50 + 50)$ GA in 200 dimensions. We recall from Table 4.2 that the optimal crossover probability for this setting is $p_c^* = 0.3$. The left plot in Fig. 4.10 is a fixed-target plot, in which we display for each tested crossover probability p_c (different lines) and each fitness value $i \in [0..200]$ (x-axis) the average time needed until the respective algorithm evaluates for the first time a search point of fitness at least i. The mutation-only configuration ($p_c = 0$) performs on par with the best configurations for the first

	n \ μ	2	3	5	8	10	20	30	40	50	60	70	80	90	100
$(\mu+1)$	64	0.0	0.1	0.1	0.1	0.2	0.3	0.5	0.4	0.5	0.5	0.6	0.7	0.6	0.7
	100	0.0	0.1	0.1	0.1	0.1	0.3	0.4	0.4	0.4	0.5	0.5	0.5	0.4	0.6
	150	0.0	0.1	0.1	0.1	0.1	0.2	0.3	0.3	0.4	0.4	0.5	0.4	0.4	0.5
	200	0.0	0.0	0.1	0.1	0.1	0.2	0.2	0.3	0.3	0.3	0.4	0.4	0.4	0.4
	250	0.0	0.0	0.1	0.1	0.1	0.2	0.2	0.3	0.3	0.3	0.4	0.4	0.3	0.4
	500	0.0	0.0	0.0	0.1	0.1	0.1	0.1	0.2	0.2	0.3	0.3	0.3	0.3	0.3
$(\mu+\mu)$	64	0.0	0.2	0.1	0.1	0.2	0.2	0.4	0.4	0.6	0.5	0.5	0.7	0.5	0.7
	100	0.0	0.0	0.1	0.1	0.2	0.3	0.3	0.3	0.5	0.4	0.5	0.5	0.6	0.5
	150	0.0	0.0	0.1	0.1	0.2	0.2	0.3	0.3	0.5	0.4	0.5	0.5	0.5	0.5
	200	0.0	0.0	0.1	0.1	0.1	0.1	0.3	0.3	0.3	0.3	0.4	0.5	0.4	0.5
	250	0.0	0.0	0.1	0.1	0.1	0.2	0.3	0.3	0.3	0.3	0.3	0.3	0.4	0.4
	500	0.0	0.0	0.0	0.1	0.1	0.1	0.1	0.2	0.2	0.3	0.3	0.3	0.3	0.3

Table 4.2: On LeadingOnes, the optimal value of p_c for the $(\mu+1)$ and the $(\mu+\mu)$ GA with uniform crossover and standard bit mutation, for various combinations of dimension n (rows) and μ (columns). Values are approximated from 100 independent runs each, probing $p_c \in \{0.1k \mid k \in [0..9]\} \cup \{0.95\}$.

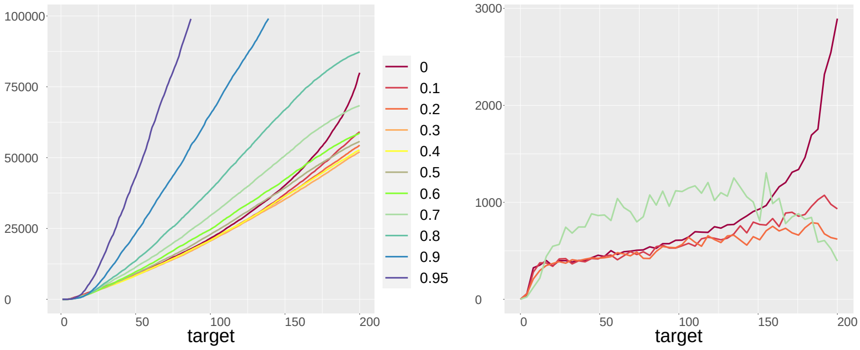


Figure 4.10: **Left:** Average fixed-target running times of the $(50+50)$ GA with uniform crossover and standard bit mutation on LeadingOnes in 200 dimensions, for different crossover probabilities p_c. Results are averages of 100 independent runs. **Right:** Gradient of selected fixed-target curves.

part of the optimization process, but then loses in performance as the optimization progresses. The plot on the right shows the gradients of the fixed-target curves. The gradient can be used to analyze which configuration performs best at a given target value. We observe an interesting behavior here, namely that the gradient of the configuration $p_c = 0.8$, which has a very bad fixed-target performance on all targets (left plot), is among the best in the final parts of the optimization. Inspired by the plot on the right therefore, we investigate the $(\mu+\lambda)$ GA using adaptive an choice of p_c in Section 7.2.

4.3.4 Summary

In this section, by varying the value of the crossover probability p_c of the $(\mu + \lambda)$ GA, we discovered on LeadingOnes that its optimal value p_c (with respect to the average running time) increases with the population size μ, whereas for fixed μ it decreases with increasing dimension n.

Our results raise the interesting question of whether a non-asymptotic runtime analysis (i.e., bounds that hold for a fixed dimension rather than in big-Oh notation) could shed new light on our understanding of evolutionary algorithms. We note that a few examples of such analyses can already be found in the literature, e.g., in [23, 27]. The regular patterns observed in Figure 4.9 suggest the presence of trends that could be turned into formal knowledge. In addition, the result in this section inspires the work on dynamic p_c for the the $(\mu + \lambda)$ GA in Section 7.2.