On the Choice of the Parameter Control Mechanism in the (1+(λ, λ)) Genetic Algorithm

Mario Alejandro Hevia Fajardo
Department of Computer Science
University of Sheffield, Sheffield, UK

Dirk Sudholt
Department of Computer Science
University of Sheffield, Sheffield, UK

ABSTRACT

The self-adjusting $(1 + (\lambda, \lambda))$ GA is the best known genetic algorithm for problems with a good fitness-distance correlation as in OneMax. It uses a parameter control mechanism for the parameter λ that governs the mutation strength and the number of offspring. However, on multimodal problems, the parameter control mechanism tends to increase λ uncontrollably.

We study this problem and possible solutions to it using rigorous runtime analysis for the standard Jump_k benchmark problem class. The original algorithm behaves like a (1+n) EA whenever the maximum value $\lambda = n$ is reached. This is ineffective for problems where large jumps are required. Capping λ at smaller values is beneficial for such problems. Finally, resetting λ to 1 allows the parameter to cycle through the parameter space. We show that this strategy is effective for all Jump_k problems: the $(1 + (\lambda, \lambda))$ GA performs as well as the $(1 + 1)$ EA with the optimal mutation rate and fast evolutionary algorithms, apart from a small polynomial overhead.

Along the way, we present new general methods for bounding the runtime of the $(1 + (\lambda, \lambda))$ GA that allows to translate existing runtime bounds from the $(1 + 1)$ EA to the self-adjusting $(1 + (\lambda, \lambda))$ GA. Our methods are easy to use and give upper bounds for novel classes of functions.

CCS CONCEPTS

• **Theory of computation** → *Theory of randomized search heuristics*;

KEYWORDS

Parameter control, runtime analysis, theory

ACM Reference Format:
Mario Alejandro Hevia Fajardo and Dirk Sudholt. 2020. On the Choice of the Parameter Control Mechanism in the (1+(λ, λ)) Genetic Algorithm. In *Genetic and Evolutionary Computation Conference (GECCO '20), July 8–12, 2020, Cancún, Mexico.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3377930.3390200

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

ACM ISBN 978-1-4503-7128-5/20/07…$15.00
https://doi.org/10.1145/3377930.3390200

1 INTRODUCTION

Parameter control mechanisms are non-static parameter choices that aim to identify parameter values that are optimal for the current state of the optimisation process. In continuous optimisation, parameter control such as step-size adaptation is vital to ensure convergence to the optimum. In the discrete domain, parameter control is much less common and we are just beginning to understand the benefits that parameter control can provide. There have been several examples where parameter control mechanisms were proposed, along with proven performance guarantees.

Böttcher et al. [5] showed that fitness-dependant mutation rates can improve performance of the $(1 + 1)$ EA on LeadingOnes by a constant factor. Badkobeh *et al.* [2] presented an adaptive strategy for the mutation rate in the (1+λ) EA that, for all values of λ, leads to provably optimal performance on OneMax. Lässig and Sudholt [26] presented adaptive schemes for choosing the offspring population size in (1+λ) EAs and the number of islands in an island model. Doerr *et al.* [16] showed that a success-based parameter control mechanism is able to identify and track the optimal mutation rate in the (1+λ) EA on OneMax, matching the performance of the best known fitness-dependent parameter [2]. Similarly Doerr *et al.* [19] presented that a self-adaptive mechanism for the mutation rate in the (1,λ) EA with a sufficiently large λ has the same asymptotic expected runtime on OneMax as in [2]. Mambrini and Sudholt [32] adapted the migration interval in island models and showed that adaptation can reduce the communication effort beyond the best possible fixed parameter. Doerr et al. [15] proved that a success-based parameter control mechanism based on the 1/5 rule is able to achieve an asymptotically optimal runtime on LeadingOnes. Lissovoi et al. [31] propose a Generalised Random Gradient Hyper-Heuristic that can learn to adapt the neighbourhood size of Random Local Search optimally during the run on LeadingOnes, proving that it has the best possible runtime achievable by any algorithm that uses the same low level heuristics. Doerr and Doerr give a comprehensive survey of theoretical results [13].

One of the most successful implementations of parameter control mechanisms is the self-adjusting $(1 + (\lambda, \lambda))$ GA [14], which is the fastest known unbiased genetic algorithm on OneMax and has shown excellent performance on NP-hard problems like MaxSat in both empirical [21] and theoretical studies [6]. The $(1 + (\lambda, \lambda))$ GA first creates λ mutants by a process similar to a standard bit mutation with mutation rate λ/n (the only difference to standard GAs being that all mutants flip *the same* number of bits). Then it picks the best mutant and performs λ crossovers with the original parent. A biased uniform crossover is used that independently picks each bit from the mutant with probability $1/\lambda$. If the best search point created by crossover is at least as good as the parent, the former replaces the latter. The parameter λ is key to the performance of the

$(1 + (\lambda, \lambda))$ GA as it governs the number of offspring, the mutation rate and the crossover bias.

The self-adjusting $(1 + (\lambda, \lambda))$ GA uses the 1/5-th success rule to adjust λ. Doerr and Doerr [12] proved that the algorithm only needs $O(n)$ expected function evaluations on OneMax, breaking the $\Theta(n \log n)$ barrier that applies to all mutation-only algorithms [29]. The time of $O(n)$ is asymptotically the best runtime possible for any static or dynamic parameter setting on the $(1 + (\lambda, \lambda))$ GA. Doerr and Doerr [12] point out that this was the first success-based parameter control mechanism proven to reduce the optimisation time of an algorithm by more than a constant factor, compared to optimal static parameter settings.

Goldman and Punch [21] reported excellent performance for the self-adjusting $(1 + (\lambda, \lambda))$ GA on random instances of the maximum satisfiability problem, and a similar setting was studied by Buzdalov and Doerr [6]. In the latter analysis it was shown that the algorithm is effective on instances with good fitness-distance correlation, however on instances with low fitness-distance correlation the algorithm's performance decays. Antipov et al. [1] analysed the algorithm on LeadingOnes to understand better the behaviour on functions with low fitness-distance correlation. They showed that the self-adjusting $(1 + (\lambda, \lambda))$ GA has the same asymptotic runtime as the $(1 + 1)$ EA, but the hidden constants seem to be very large. Other empirical studies [20, 23] corroborated that the self-adjusting mechanism does not behave well on problems with low fitness-distance correlation or local optima.

Several of the above works identified that the issue lies in the parameter control mechanism used. On functions with low fitness-distance correlation, the algorithm can get stuck in situations where λ diverges to its maximum value $\lambda = n$, and then performance deteriorates.

Goldman and Punch [21] suggested to restart the parameter λ to 1 when $\lambda = n$ but also restart the search from a random individual. In Buzdalov and Doerr [6] the authors proposed capping the value of λ depending on the fitness-distance correlation of the problem at hand. Lastly, in [3, 4] the authors proposed a modification where the growth of λ is slowed down for long unsuccessful runs, while still letting the algorithm increase the parameter indefinitely. It achieves this by resetting λ to the parameters of its last successful generation after a certain number of unsuccessful generations and letting the algorithm increase λ a bit more every time it is reset.

At the moment, it is not clear which of these modifications is the best choice. Previous research is fragmented and most of the modifications proposed have only been studied empirically. We seek to provide more clarity by providing a comprehensive analysis of different approaches for parameter control in the self-adjusting $(1 + (\lambda, \lambda))$ GA. We consider the $\textsc{Jump}_k$ benchmark problem class, a class of multimodal problems on which evolutionary algorithms typically have to make a jump to the optimum at a Hamming distance of k. The parameter k means that $\textsc{Jump}_k$ has an adjustable difficulty and thus represents a whole range from easy to difficult multimodal and even deceptive problems. It was also the first problem for which a drastic advantage from using crossover could be proven with mathematical rigour [24]. More recent analyses have shown that crossover can reduce the runtime to $O(n^{k-1} \log n)$ [11] and $O(n \log n + 4^k)$ using additional diversity mechanisms [10].

We first present a general method for obtaining upper bounds on the expected optimisation time of the original $(1 + (\lambda, \lambda))$ GA, based on the fitness-level method, in Section 3. Despite its simplicity, we show that it gives tight bounds, up to lower-order terms, on $\textsc{Jump}_k$ functions. Our lower bounds in Section 4 show that the original $(1 + (\lambda, \lambda))$ GA does not benefit from crossover on $\textsc{Jump}_k$ functions. Subsequently in Section 5 we analyse the performance change when λ is capped to a value less than n, providing a method to analyse any parameter choice $\lambda_{\max}$. We also show that capping λ can improve the performance of the algorithm, but its behaviour is highly dependant on the choice of $\lambda_{\max}$ defying the point of using a parameter control mechanism. Finally in Section 6, we analyse the benefits of resetting λ. With this strategy the algorithm is able to traverse the parameter space, instead of getting stuck with a certain parameter, benefiting the algorithm's behaviour when encountering local optima. In particular with a clever selection of the update factor F we show that for $\textsc{Jump}_k$ the runtime of the algorithm is only $O(n^2/k)$ slower than the runtime of the $(1 + 1)$ EA with optimal parameter choice.

2 PRELIMINARIES

We use runtime analysis to analyse the performance of the self-adjusting $(1 + (\lambda, \lambda))$ GA on n-dimensional pseudo-Boolean functions $f\colon \{0,1\}^n \to \mathbb{R}$. We consider both the random number of fitness evaluations T^{eval} until a global optimum is found (also called optimisation time or runtime) as well as the random number of generations T^{gen}. The former reflects the total computational effort, whereas the latter is more relevant when the execution and the generation of offspring can be parallelised efficiently.

In the following, we write $H(x, x^*)$ to denote the Hamming distance between bit strings x and x^*. By $\mathcal{B}(n, p)$ we denote the binomial distribution with parameters $n \in \mathbb{N}$ and $p \in [0, 1]$.

The self-adjusting $(1 + (\lambda, \lambda))$ GA is a crossover-based evolutionary algorithm that uses a mutation phase with a mutation rate higher than usual to assist exploration and a crossover phase as a repair mechanism. During the mutation phase the parent is mutated λ times, using a mutation operator called $\text{flip}_\ell(x)$. It chooses ℓ different bit positions in x uniformly at random and then it flips the values in those bits to create the mutated bit string. Each mutation phase the variable ℓ is sampled only once from a binomial distribution $\mathcal{B}(n, \frac{\lambda}{n})$, causing all mutation offspring have the same distance to the parent. Afterwards, during the crossover phase the algorithm creates λ offspring by applying a biased uniform crossover called $\text{cross}_c(x, x')$ between the parent x and the best offspring from the mutation phase x'. The crossover operator works as follows: for each bit position, it selects the bit value from x' with probability $c = \frac{1}{\lambda}$ and from x otherwise. After the crossover phase, the algorithm performs an elitist selection using only the offspring from the crossover phase and the parent.

The algorithm adjusts λ every generation with a multiplicative update rule, where λ is multiplied by a factor $F^{1/4}$ if there is no improvement in fitness and divided by F otherwise. The parameter λ has an upper limit $\lambda_{\max}$ which is commonly set to n.

In this work we use a small variation of the algorithm, shown in Algorithm 1, where during the selection step the best offspring from the mutation phase is also considered. This modification has been

suggested before in [7, 21] as a way to improve the performance of the $(1 + (\lambda, \lambda))$ GA. We believe (and tacitly take for granted) that this change does not invalidate previous theoretical runtime guarantees on problems such as OneMax [12] and LeadingOnes [1][1].

Carvalho Pinto and Doerr [7] presented refinements of the $(1 + (\lambda, \lambda))$ GA that they call *implementation-aware*; these can save unnecessary evaluations and decrease some runtime results by constant factors. We consider the original $(1 + (\lambda, \lambda))$ GA for simplicity, and since we are interested in larger performance differences.

Considering the best offspring from the mutation phase is particularly helpful when the algorithm needs to make large jumps, as when encountering with local optima. In fact, in Section 4 we show that for large jumps the crossover phase is not very helpful for reaching a higher fitness level, because the crossover phase tend to search near the current parent while the large mutations during the mutation phase can more easily jump out of local optima.

The $\textsc{Jump}_k$ benchmark problem class is a class of unitation functions, that is, functions that only depend on the number of 1-bits in a bit string, denoted as $|x|$:

$$\textsc{Jump}_k(x) := \begin{cases} n - |x| & \text{if } n - k < |x| < n \\ k + |x| & \text{otherwise} \end{cases}$$

The $\textsc{Jump}_k$ function increases its fitness value with the first $n - k$ 1-bits in the bit string, reaching a plateau of local optima that all have $n - k$ 1-bits. Larger numbers of 1-bits leads to a *valley* that contains the lowest fitness values in all the function. The fitness value in the *valley* decreases when adding 1-bits, with the minimum fitness neighboring the global optimum. The optimum is the bit string 1^n.

3 FITNESS-LEVEL UPPER BOUNDS FOR THE SELF-ADJUSTING $(1 + (\lambda, \lambda))$ GA

The self-adjusting $(1 + (\lambda, \lambda))$ GA has only been analysed theoretically on easy unimodal functions like OneMax [12] and LeadingOnes [1] as well as random satisfiability instances [6]. Despite these analyses we do not have a clear understanding of its behaviour on other problem settings, especially when it encounters local optima.

In this section we give a new general method to find upper bounds for the runtime of the self-adjusting $(1 + (\lambda, \lambda))$ GA using previously known runtime bounds from the $(1 + 1)$ EA. It is based on the observation that, when λ hits its maximum value of $\lambda = n$, the algorithm temporarily performs n standard bit mutations and thus simulates a generation of a (1+n) EA.

In a nutshell, the fitness-level method uses so-called f-based partitions, which is a partition of $\{0, 1\}^n$ into sets $A_1, \ldots, A_{m+1}$ where all search points in A_i are strictly worse than all search points in A_{i+1} and A_{m+1} contains all global optima. For the $(1 + 1)$ EA, we derive an upper bound as follows. Suppose that we know that for every search point $x \in A_i$, the probability of finding a search point in a higher fitness-level set is at least s_i, for some expression $s_i > 0$. Then the expected time for leaving set A_i is at most $1/s_i$.

[1]For OneMax, Doerr and Doerr [12] confirm this fact without proof. Analyses on OneMax and LeadingOnes were based on drift analysis, and the drift can only increase if additional opportunities for improvements are considered. It is less clear how this affects the self-adjusting mechanism; however, previous analyses have shown that the $(1 + (\lambda, \lambda))$ GA is very robust in tracking the best parameter setting for these easy unimodal functions and we believe this will still be the case with the modification.

Algorithm 1: The self-adjusting $(1 + (\lambda, \lambda))$ GA [14]

Initialization: Sample $x \in \{0, 1\}^n$ uniformly at random;
Initialize $\lambda \leftarrow 1$, $p \leftarrow \lambda/n$, $c \leftarrow 1/\lambda$;
Optimization: for $t = 1, 2, \ldots$ **do**
  Mutation phase:
  Sample ℓ from $\mathcal{B}(n, p)$;
  for $i = 1, \ldots, \lambda$ **do**
    Sample $x^{(i)} \leftarrow \text{flip}_\ell(x)$ and query $f(x^{(i)})$;
  Choose $x' \in \{x^{(1)}, \ldots, x^{(\lambda)}\}$ with $f(x') = \max\{f(x^{(1)}), \ldots, f(x^{(\lambda)})\}$ u.a.r.;
  Crossover phase:
  for $i = 1, \ldots, \lambda$ **do**
    Sample $y^{(i)} \leftarrow \text{cross}_c(x, x')$ and query $f(y^{(i)})$;
  If exists, choose $y \in \{x', y^{(1)}, \ldots, y^{(\lambda)}\} \setminus \{x\}$ with $f(y) = \max\{f(x'), f(y^{(1)}), \ldots, f(y^{(\lambda)})\}$ u.a.r.;
  otherwise, set $y := x$;
  Selection and update step:
  if $f(y) > f(x)$ **then** $x \leftarrow y$; $\lambda \leftarrow \max\{\lambda/F, 1\}$;
  if $f(y) = f(x)$ **then** $x \leftarrow y$; $\lambda \leftarrow \min\{\lambda F^{1/4}, \lambda_{\max}\}$;
  if $f(y) < f(x)$ **then** $\lambda \leftarrow \min\{\lambda F^{1/4}, \lambda_{\max}\}$;

Since every fitness level has to be left at most once, the expected optimisation time of the $(1 + 1)$ EA is at most $\sum_{i=1}^{m} 1/s_i$.

The fitness-level method is a simple and versatile method in its own right, and it allows researchers to translate bounds on the runtime of the simple $(1 + 1)$ EA to other elitist algorithms. This has been achieved for parallel evolutionary algorithms [28], ant colony optimisation [22, 33], and particle swarm optimisation [35]. It further gives rise to tail bounds [37] and lower bounds [34], and the principles extend to non-elitist algorithms as well [9, 17].

Fitness levels may contain search points of different fitness. In the special case where each set A_i contains search points with only one fitness value the partition is called a *canonical* partition.

The following theorem gives a fitness-level upper bound tailored to the $(1 + (\lambda, \lambda))$ GA.

Theorem 3.1. *Given an arbitrary f-based partition $A_1, \ldots, A_{m+1}$. Let d be the number of non-optimal fitness values, $F > 1$ constant, and s_i a lower bound for the $(1 + 1)$ EA finding an improvement from any search point in fitness level A_i. Then for the self-adjusting $(1 + (\lambda, \lambda))$ GA we have*

$$E\left(T^{\text{eval}}\right) \leq O(dn) + 2\sum_{i=1}^{m} \frac{1}{s_i} \quad \text{and}$$

$$E\left(T^{\text{gen}}\right) \leq 4\log_F(n) + 6d + \frac{1}{n}\sum_{i=1}^{m} \frac{1}{s_i}.$$

To prove Theorem 3.1, we analyse the time the algorithm spends in generations with $\lambda < n$ and those in generations with $\lambda = n$.

In the following, we refer to a generation that improves the current best fitness as *successful* and otherwise as *unsuccessful*. We show that a logarithmic number of unsuccessful generations is sufficient to reach the maximum λ value.

Lemma 3.2. *Let $F > 1$, $F = O(1)$, $\lambda_{\text{init}} \in [1, n)$ and $\lambda_{\text{init}} < \lambda_{\text{new}} \leq n$. If in every generation $f(y) \leq f(x)$, the self-adjusting $(1 + (\lambda, \lambda))$ GA needs at most $4 \log_F \left(\frac{\lambda_{\text{new}}}{\lambda_{\text{init}}}\right)$ unsuccessful generations to grow λ from λ_{init} to λ_{new}. During these generations the algorithm makes $O\left(\frac{\lambda_{\text{new}}}{F^{1/4}-1}\right)$ evaluations.*

If $F = 1 + \Omega(1)$ then the number of generations is $O(\log n)$ and the number of evaluations is $O(n)$.

Proof. After i unsuccessful generations the offspring population size is updated by $\lambda_{\text{new}} = \lambda_{\text{init}} \cdot F^{i/4}$. The number of unsuccessful generations needed is thus

$$4 \log_F \left(\frac{\lambda_{\text{new}}}{\lambda_{\text{init}}}\right) \leq 4 \log_F(\lambda_{\text{new}}) \leq 4 \log_F(n).$$

During these generations, the number of evaluations is at most:

$$2 \sum_{i=4\log_F(\lambda_{\text{init}})}^{4\log_F(\lambda_{\text{new}})} \left(F^{1/4}\right)^i = 2 \frac{\left(F^{1/4}\right)^{4\log_F(\lambda_{\text{new}})+1} - \left(F^{1/4}\right)^{4\log_F(\lambda_{\text{init}})}}{F^{1/4}-1}$$

$$= \frac{2\left(F^{1/4}\lambda_{\text{new}} - \lambda_{\text{init}}\right)}{F^{1/4}-1} = O\left(\frac{\lambda_{\text{new}}}{F^{1/4}-1}\right)$$

For $F = 1 + \Omega(1)$, $4 \log_F(n) = 4 \log(n)/\log(F) = O(\log n)$ and $O\left(\frac{\lambda_{\text{new}}}{F^{1/4}-1}\right) = O(n)$. □

Now we bound the number of generations in which the self-adjusting $(1 + (\lambda, \lambda))$ GA operates with $\lambda < n$. To do so, we take into account that the algorithm will not need to increase from λ_{init} every time since we only decrease λ by a factor F each time we find an improvement.

Lemma 3.3. *Let $F > 1$, $\lambda_{\max} \leq n$, and d be the number of non-optimal fitness values of an arbitrary fitness function. The maximum number of generations in which the self-adjusting $(1 + (\lambda, \lambda))$ GA uses $\lambda < \lambda_{\max}$ is at most $4 \log_F(\lambda_{\max}) + 5d$. These generations lead to $O\left(dn + \frac{n}{F^{1/4}-1}\right)$ evaluations.*

Proof. In every successful generation, λ is decreased to $\max\{1, \lambda/F\}$ and otherwise it is increased to $\min\{n, \lambda \cdot F^{1/4}\}$.

We use the *accounting method* [8, Chapter 17] to account for all generations with $\lambda < \lambda_{\max}$. The basic idea is to create a fictional bank account to which operations are being charged. Some operations are allowed to pay excess amounts, while others can take money from such accounts to pay for their costs. Provided that no fictional account gets overdrawn the total amount of money paid bounds the total cost of all operations.

We start with a fictional bank account and pay costs of $4 \log_F(\lambda_{\max})$, since that is the maximum number of consecutive unsuccessful generations before reaching $\lambda = \lambda_{\max}$ (Lemma 3.2).

In a successful generation, we pay costs of 1 to cover the cost of the generation, and deposit an additional amount of 4 to the fictional bank account, which will be used to pay for 4 unsuccessful generations needed to increase λ to its original value. Unsuccessful generations that increase λ may withdraw 1 from the fictional account and pay for the cost of this generation. Unsuccessful generations where $\lambda = \lambda_{\max}$ are not charged since they are not counted.

We now need to prove that the fictional bank account is never overdrawn. For any point in time, the number of generations where λ increases is bounded by $T^{\text{inc}} \leq 4 \log_F(n) + 4T^{\text{dec}}$ where T^{dec} is the number of generations decreasing λ. This holds by Lemma 3.2 and the fact that one successful generation that decreases λ compensates for 4 unsuccessful generations that may increase λ. Considering the initial payment of $4 \log_F(n)$ and transactions for each generation, the current balance is

$$4 \log_F(n) - T^{\text{inc}} + 4T^{\text{dec}} \geq 0,$$

that is, the account is never overdrawn. The number of generations with $\lambda < \lambda_{\max}$ is thus bounded by the sum of all payments. There can only be d successful generations, hence the sum of payments is at most $4 \log_F(n) + 5d$.

It remains to bound the number of evaluations. Since we initialise with $\lambda = 1$, there must be $4 \log_F(n)$ generations $t_1, t_2, \ldots$ such that during generation t_i, we have $\lambda \leq F^{i/4}$. From Lemma 3.2, we know that during these $4 \log_F(\lambda_{\max})$ generations, the algorithm will use $O\left(\frac{n}{F^{1/4}-1}\right)$ evaluations. Adding to this, in the other at most $5d$ possible generations, the maximum number of evaluations per generation is bounded by $2\lambda_{\max}$. Therefore the algorithm will use $O\left(d\lambda_{\max} + \frac{n}{F^{1/4}-1}\right) = O\left(dn + \frac{n}{F^{1/4}-1}\right)$ evaluations with an offspring population size $\lambda < \lambda_{\max}$. □

With Lemma 3.3 now we have the tools necessary to analyse the runtime of Algorithm 1.

Proof of Theorem 3.1. Owing to Lemma 3.3, we can focus on bounding the time spent in generations with $\lambda = n$. In these generations, the mutation rate is $p = \lambda/n = 1$ and thus all bits are flipped during mutation. When the current search point is x, mutation thus produces its binary complement, $\overline{x}$. The crossover phase uses a crossover bias of $1/n$, which means that each bit is independently taken from the mutant $\overline{x}$ with probability $1/n$ and otherwise it is taken from x. This is equivalent to a standard bit mutation with the default mutation rate of $1/n$. Given that $\lambda = n$, during the crossover phase the algorithm creates n independent offspring using standard bit mutation. The crossover phase is then equivalent to the output of a (1+n) EA.

For each fitness level we calculate the number of generations the $(1 + (\lambda, \lambda))$ GA spends on this level while $\lambda = n$ and the $(1 + (\lambda, \lambda))$ GA essentially simulates a (1+n) EA. (We pessimistically ignore the fact that such a situation may not be reached at all; especially on easy problems, λ may not hit the maximum value before the optimum is found.) We argue as in Lässig and Sudholt [27, Theorem 1] to derive a fitness-level bound for the (1+n) EA. The probability that there is one of n offspring that finds a better fitness level is at least (using $s_i^{(n)}$ to denote an amplified success probability with n offspring)

$$s_i^{(n)} = 1 - (1 - s_i)^n \geq 1 - \left(\frac{1}{1 + s_i n}\right) = \frac{s_i n}{1 + s_i n}$$

and the expected number of generations to leave A_i using $\lambda = n$ is at most $1/s_i^{(n)}$. Adding the generations with $\lambda = n$ over all fitness

levels and the generations spent with $\lambda < n$, we get

$$\mathrm{E}\left(T^{\mathrm{gen}}\right) \leq 4\log_F(n) + 5d + \sum_{i=1}^{m}\left(1 + \frac{1}{s_i n}\right) \leq 4\log_F(n) + 6d + \frac{1}{n}\sum_{i=1}^{m}\frac{1}{s_i}$$

where the last step used $m \leq d$, that is, the number of fitness levels is bounded by the number of fitness values.

By Lemma 3.3, the number of evaluations used with $\lambda < n$ is $O(dn)$ when $F = 1 + \Omega(1)$. Since with $\lambda = n$ each generation leads to $2n$ evaluations[2], multiplying the above bound yields the claimed bound on the number of evaluations. □

We show how to apply Theorem 3.1 to obtain novel bounds on the expected optimisation time of the $(1 + (\lambda, \lambda))$ GA, including the $\textsc{Jump}_k$ function class.

Theorem 3.4. *The expected optimisation time $E\left(T^{\mathrm{eval}}\right)$ of the self-adjusting $(1 + (\lambda, \lambda))$ GA with constant $F > 1$ is at most*

(a) $E\left(T^{\mathrm{eval}}\right) = O(n^n/|\mathrm{OPT}|)$ *and* $E(T^{\mathrm{gen}}) = O(n^{n-1}/|\mathrm{OPT}|)$ *on any function with d non-optimal function values, and a set* OPT *of global optima*

(b) $E\left(T^{\mathrm{eval}}\right) = O(dn)$ *and* $E(T^{\mathrm{gen}}) = O(d + \log n)$ *on unimodal functions with $d + 1$ fitness values*

(c) $E\left(T^{\mathrm{eval}}\right) = (1 + o(1)) \cdot 2n^k (1 - 1/n)^{-n+k}$ *and* $E(T^{\mathrm{gen}}) = (1 + o(1)) \cdot 2n^{k-1}(1 - 1/n)^{-n+k}$ *on $\textsc{Jump}_k$ with $k \geq 3$.*

Proof. For the general upper bound, we use a fitness level partition with A_1 containing all non-optimal fitness values and A_2 containing the set OPT. We use the corresponding success probability $s_i \geq |\mathrm{OPT}|/n^n$. With this we bound $\sum_{i=1}^{m}\frac{1}{s_i} = O(n^n/|\mathrm{OPT}|)$. The term $O(dn)$ can be absorbed since $d \leq 2^n$ and $n^n/|\mathrm{OPT}| \geq (n/2)^n$.

For unimodal functions, we use a canonical f-based partition and success probabilities of $s_i \geq 1/n \cdot (1-1/n)^{n-1} \geq 1/(en)$. This yields $\mathrm{E}\left(T^{\mathrm{eval}}\right) \leq O(dn) + 2\sum_{i=1}^{d} en = O(dn)$. For the expected number of generations, we get $\mathrm{E}(T^{\mathrm{gen}}) \leq 4\log_F(n) + 6d + \frac{1}{n}\sum_{i=1}^{d} en = O(d + \log n)$.

For $\textsc{Jump}_k$ functions with $k \geq 2$ any individual that is not a local or global optimum can find an improvement by increasing or decreasing the number of 1-bits. This yields success probabilities of at least $(n - i)/(en)$ for all search points with $0 \leq i < n - k$ ones and of at least $i/(en)$ for all search points with $n - k < i < n$ ones. For search points with $n - k$ ones, a standard bit mutation can jump to the optimum by flipping the correct k 0-bits and not flipping any other bit. This has a probability $s_{n-k} = (1/n)^k (1 - 1/n)^{n-k}$. Hence,

$$\mathrm{E}\left(T^{\mathrm{eval}}\right) \leq O(n^2) + 2n^k\left(1 - \frac{1}{n}\right)^{-n+k} = (1 + o(1)) \cdot 2n^k\left(1 - \frac{1}{n}\right)^{-n+k}$$ □

[2]This value can be reduced to $n + 1$ if identical mutants are not evaluated, following ideas similar to Carvalho Pinto and Doerr [7]

4 THE $(1 + (\lambda, \lambda))$ GA IS INEFFICIENT ON JUMP

We now show that the bound for the $(1 + (\lambda, \lambda))$ GA on $\textsc{Jump}_k$ from Theorem 3.4 (c) is asymptotically tight. This implies that the $(1 + (\lambda, \lambda))$ GA is no more efficient on $\textsc{Jump}_k$ than the $(1 + 1)$ EA and less efficient than other GAs using crossover [10, 11, 24].

Theorem 4.1. *Let $F > 1$ be a constant. The expected optimisation of the self-adjusting $(1 + (\lambda, \lambda))$ GA on the $\textsc{Jump}_k$ function with $4 \leq k \leq (1 - \varepsilon)n/2$, for any constant $\varepsilon > 0$, is*

$$(1 - o(1)) \cdot 2n^k\left(1 - \frac{1}{n}\right)^{-n+k}.$$

We first show the following upper and lower bounds on the probability that the $(1 + (\lambda, \lambda))$ GA will find any particular target search point x^* during one mutation phase. Even though we only need the upper bounds in this section, the lower bounds on transition probabilities will be useful later on.

Lemma 4.2. *For every current search point x, every target search point x^* and every current parameter λ, let $p^{\lambda}_{\mathrm{mut}}(x, x^*)$ be the probability that the $(1 + (\lambda, \lambda))$ GA creates x^* during the mutation phase of one generation.*

If $x^* \in \{x, \overline{x}\}$, $p^{\lambda}_{\mathrm{mut}}(x, x^*) = (\lambda/n)^{H(x,x^*)}(1 - \lambda/n)^{n-H(x,x^*)}$. Otherwise, $\frac{\lambda}{2}(\lambda/n)^{H(x,x^*)}(1 - \lambda/n)^{n-H(x,x^*)} \leq p^{\lambda}_{\mathrm{mut}}(x, x^*)$ and $p^{\lambda}_{\mathrm{mut}}(x, x^*) \leq \lambda(\lambda/n)^{H(x,x^*)}(1 - \lambda/n)^{n-H(x,x^*)}$.

The term $(\lambda/n)^{H(x,x^*)}(1 - \lambda/n)^{n-H(x,x^*)}$ equals the probability of a standard bit mutation with mutation probability λ/n creating x^* from x. If $x^* \notin \{x, \overline{x}\}$, the offspring population of λ amplifies this probability by a factor within $[\lambda/2, \lambda]$. If $x^* \in \{x, \overline{x}\}$, the $(1 + (\lambda, \lambda))$ GA does not benefit from its offspring population at all.

Proof of Lemma 4.2. The algorithm needs to sample $\ell = H(x, x^*)$, in order to find x^* during the mutation phase. The probability of this happening is

$$\Pr\left(\ell = H(x, x^*)\right) = \binom{n}{H(x, x^*)}(\lambda/n)^{H(x,x^*)}(1 - \lambda/n)^{n-H(x,x^*)} \tag{1}$$

If $x^* \in \{x, \overline{x}\}$, $\binom{n}{H(x,x^*)} = 1$ and the claim for this case follows as all λ mutants will create x^* for the right choice of ℓ.

Otherwise, the $(1 + (\lambda, \lambda))$ GA also needs to flip the correct bits during the mutation phase. Since there are $\binom{n}{H(x,x^*)}$ possible ways to flip the bits the probability that one offspring flips the correct bits is $\binom{n}{H(x,x^*)}^{-1}$. This gives us the following probability of finding x^* during λ mutations, conditional on $\ell = H(x, x^*)$:

$$\Pr\left(x^* \mid \ell = H(x, x^*)\right) = 1 - \left(1 - 1/\binom{n}{H(x, x^*)}\right)^{\lambda}.$$

This is bounded from above by

$$\lambda/\binom{n}{H(x, x^*)}$$

and bounded from below using $(1 + x)^r \leq \frac{1}{1-rx}$ as

$$\frac{\lambda/\binom{n}{H(x,x^*)}}{1 + \lambda/\binom{n}{H(x,x^*)}} \geq \frac{\lambda/\binom{n}{H(x,x^*)}}{2}$$

where the last inequality follows from $\binom{n}{H(x,x^*)} \geq n$ and thus $1 + \lambda/n \leq 2$. Since

$$p^{\lambda}_{\text{mut}}(x, x^*) = \Pr\left(x^* \mid \ell = H(x, x^*)\right) \Pr\left(\ell = H(x, x^*)\right),$$

multiplying (1) with the above bounds on $\Pr(x^* \mid \ell = H(x, x^*))$ and observing that the binomial coefficients cancel completes the proof. □

The following lemma gives an upper bound on the probability of hitting any specific target search point x^* during one crossover phase of the $(1 + (\lambda, \lambda))$ GA. Note that, for the original $(1 + (\lambda, \lambda))$ GA that does not consider mutants for selection, Lemma 4.3 gives an upper bound for hitting x^* in one generation.

Lemma 4.3. *For every current search point x, every target search point x^* and every current parameter λ, the probability that the $(1 + (\lambda, \lambda))$ GA creates x^* during the crossover phase of one generation is at most*

$$\lambda^2 \left(\frac{1}{n}\right)^{H(x,x^*)} \left(1 - \frac{1}{n}\right)^{n-H(x,x^*)}.$$

Before diving into the proof, we give the main idea here. Recall that in every generation, the $(1 + (\lambda, \lambda))$ GA performs λ mutations with a radius of ℓ (drawn from a binomial distribution with parameters λ/n and n) and λ crossover operations with the best mutant. All mutants are chosen uniformly at random from the Hamming ball of radius ℓ around x. However, the following selection of the best mutant does not preserve uniformity as some offspring on said Hamming ball may have a higher fitness than others. Hence the crossover operations will affect particular regions of the search space more than others. While this is a helpful algorithmic concept (in a sense that this makes the $(1 + (\lambda, \lambda))$ GA solve OneMax in expected time $O(n)$ [14]), it makes it hard to analyse what search points will be generated during crossover as it depends on the fitness function in hand.

As a solution, we borrow an idea similar to *non-selective family trees* by Witt [36] and Lehre and Yao [30] that was also used in previous analyses of the $(1 + (\lambda, \lambda))$ GA [12, Proof of Proposition 1]. We consider a variant of the $(1 + (\lambda, \lambda))$ GA that we call *non-selective* $(1 + (\lambda, \lambda))$ *GA*: instead of performing λ crossovers with the best mutant, it performs λ crossovers for *all of the λ mutants*. This results in λ^2 offspring generated from crossover, in addition to λ mutants. Since the offspring created by the original $(1 + (\lambda, \lambda))$ GA form a subset of the offspring generated by the non-selective $(1 + (\lambda, \lambda))$ GA, the probability of the original $(1 + (\lambda, \lambda))$ GA creating x^* in one crossover phase is bounded by the probability of the non-selective $(1 + (\lambda, \lambda))$ GA creating x^* during crossover. Owing to the absence of selection, the output of the λ^2 crossover operations is independent of the fitness and we obtain a probability bound that only depends on the Hamming distance $H(x, x^*)$.

Proof of Lemma 4.3. We argue similarly as the proof of Proposition 1 in [12]. Fix an offspring y created by the non-selective $(1 + (\lambda, \lambda))$ GA. The process for creating y can be described as follows. The algorithm first picks a random value of ℓ according a Binomial distribution with parameters n and λ/n, and then flips ℓ bits chosen at random to create a mutant x'. The creation of x' can alternatively be regarded as a standard bit mutation with a mutation rate of λ/n. To create y, each bit is independently taken from x' with probability $1/\lambda$. Hence, each bit y_i in y attains the value $1 - x_i$ with probability $1/n$, and it attains value x_i with probability $1 - 1/n$, independently from all other bits. Hence the creation of y can be described as a standard bit mutation with mutation rate $1/n$.

The probability of $y = x^*$ is thus $(1/n)^{H(x,x^*)}(1 - 1/n)^{n-H(x,x^*)}$. Note that different offspring are not independent as they use the same random value of ℓ, and every batch of λ crossover operations is derived from the same mutant. Taking a union bound over all λ^2 offspring allows us to conclude, despite these dependencies, that the probability of one offspring generating x^* is at most

$$\lambda^2 \left(\frac{1}{n}\right)^{H(x,x^*)} \left(1 - \frac{1}{n}\right)^{n-H(x,x^*)}.$$ □

Now we are in a position to prove Theorem 4.1.

Proof of Theorem 4.1. By standard Chernoff bounds, the probability that the initial search point will have at most $(1 + \varepsilon)n/2$ ones is $1 - 2^{-\Omega(n)}$. We assume this to happen and note that then the algorithm will never accept a search point in the fitness valley of $n - k < i < n$ ones.

Let T^{plateau} be the random number of generations until the plateau of search points with $n - k$ ones or the global optimum is reached for the first time. We bound $\mathrm{E}\left(T^{\text{plateau}}\right)$ from above as follows[3]. This can be done using Theorem 3.1 as in Theorem 3.4 (c), but with a non-canonical f-based partition where the best fitness level includes the plateau and the global optimum. This yields $\mathrm{E}\left(T^{\text{plateau}}\right) = O(n)$.

By Lemmas 4.3 and 4.2, a generation with parameter λ reaches the optimum with probability at most

$$\lambda(\lambda/n)^k(1 - \lambda/n)^{n-k} + \lambda^2 n^{-k} = O(1/n^2) := p$$

since the current search point has Hamming distance at least k from the optimum and $k \geq 4$. Then the probability that the global optimum is found during the first T^{plateau} steps is at most

$$\sum_{t=0}^{\infty} \Pr\left(T^{\text{plateau}} = t\right) \cdot tp = p \cdot \mathrm{E}\left(T^{\text{plateau}}\right) = O((\log n)/n).$$

Now assume that the plateau has been reached and $\lambda < \lambda_{\max} = n$. Since the optimum is the only search point with a strictly larger fitness, λ will be increased in every generation unless the optimum is found. By Lemma 3.2, there are at most $4 \log_F n$ generations before λ has increased to $\lambda_{\max}$. By the same arguments as above, the probability that the optimum is found during this time is $O((\log n)/n^2)$.

Once $\lambda = \lambda_{\max} = n$ has been reached, mutation always creates search points with k ones (i. e. mutation will never find the optimum) and crossover boils down to a standard bit mutation with mutation rate $1/n$. Then the probability of one crossover creating the optimum is $(1/n)^k(1 - 1/n)^{n-k}$ and the expected number of

[3] The $(1 + (\lambda, \lambda))$ GA optimises OneMax in expected time $O(n)$, but it is not immediately obvious how to translate the analysis to the OneMax-like parts of Jump_k. Note that the $(1 + (\lambda, \lambda))$ GA may overshoot the plateau before the plateau is reached and then the analysis on OneMax breaks down. We suspect this can be fixed with small modifications, but for now we show a more obvious bound as this is sufficient for our purposes.

crossover operations for hitting the optimum is thus

$$n^k \cdot \left(1 - \frac{1}{n}\right)^{-n+k}.$$

Since every batch of λ crossover operations is preceded by λ (useless) fitness evaluations during mutation, this adds a factor of 2 to the above lower bound. The proof is completed by noting that, after adding up all failure probabilities, this case is reached with probability at least $1 - O((\log n)/n)$. □

5 CAPPING λ

A simple solution to prevent λ from growing to large values is to constrain it. Buzdalov and Doerr [6] first suggested this strategy, showing that it benefits the algorithm when optimising instances of the maximum satisfiability problem with weak fitness-distance correlation. Similarly in [4] the authors showed empirically that capping λ at $\log n$ can improve the performance on linear functions with random weights. In [15] it was used, arguing that mutation probabilities larger than $1/2$ are considered ill-natured. Here we explore its benefits on $\textsc{Jump}_k$ functions. In the following theorem, we assume for simplicity to start on the plateau.

Theorem 5.1. *After reaching the plateau, the expected number of function evaluations for the* $(1 + (\lambda, \lambda))$ *GA with* $F > 1$ *constant and* λ *capped at* $\lambda_{\max} < n$ *is at most*

$$O(n) + 4\left(\frac{n}{\lambda_{\max}}\right)^k \left(1 - \frac{\lambda_{\max}}{n}\right)^{-n+k}.$$

Proof. By Lemma 3.2, λ will reach $\lambda_{\max}$ or find the global optimum within $O(n)$ evaluations. Then the probability of jumping to the optimum in one generation is at least $\lambda_{\max}/2 \cdot (\lambda_{\max}/n)^k (1 - \lambda_{\max}/n)^{n-k}$ by Lemma 4.2. Taking the reciprocal and multiplying by 2λ yields the claim. □

Note that for $\lambda_{\max} := k$, Theorem 5.1 yields an upper bound of

$$O(n) + 4\left(\frac{n}{k}\right)^k \left(1 - \frac{k}{n}\right)^{-n+k}.$$

For $k \geq 3$, this matches the expected time for the $(1 + 1)$ EA with the optimal mutation rate of k/n up to constant factors. However, we would need to know k in advance, which defies the goal of parameter control.

If k is not known, an alternative strategy is to set $\lambda_{\max} := n/2$. Then the $(1 + (\lambda, \lambda))$ GA is able to simulate random search whenever $\lambda_{\max}$ is reached. This is a potential advantage on very hard and deceptive functions where random search is a viable technique (e. g. $\textsc{Jump}_k$ with very large k such as $k > n/\log n$). Note that the $(1 + (\lambda, \lambda))$ GA still retains its exploitation capability and is still able to optimise OneMax efficiently; the cap on λ only kicks in when regular exploitation fails.

Theorem 5.2. *Let* f *be any function with* d *non-optimal fitness values and a set* OPT *of global optima such that either* $|\,\mathrm{OPT}\,| \geq 2$ *or* $\mathrm{OPT} = \{x^*\}$ *and its complement* $\overline{x^*}$ *does not have the second-best fitness value. Then for the self-adjusting* $(1 + (\lambda, \lambda))$ *GA with* $\lambda_{\max} := n/2$ *and* $F > 1$ *constant we have*

$$E\left(T^{\text{eval}}\right) \leq O(dn) + \frac{2^{n+4}}{|\mathrm{OPT}|}$$

$$E\left(T^{\text{gen}}\right) \leq O(d + \log n) + \frac{2^{n+4}}{n|\mathrm{OPT}|}$$

Proof. By Lemma 3.3, the algorithm spends $O(dn)$ evaluations and $O(d + \log n)$ generations in settings with $\lambda < \lambda_{\max}$. Hence we can focus on improvement probabilities when $\lambda = \lambda_{\max}$.

If $|\,\mathrm{OPT}\,| \geq 2$, by Lemma 4.2, the probability of one generation hitting any search point in OPT that is not the binary complement of the current search point is at least $\lambda_{\max} \cdot 2^{-n-1} = n \cdot 2^{-n-2}$. Since there are at least $|\,\mathrm{OPT}\,| - 1 \geq |\,\mathrm{OPT}\,|/2$ such search points and the probabilities for hitting these are disjoint events, the probability for finding the optimum is at least $n \cdot 2^{-n-3} \cdot |\,\mathrm{OPT}\,|$. Taking the reciprocal gives an upper bound on $\mathrm{E}\,(T^{\text{gen}})$, and multiplying by $2\lambda_{\max} = n$ yields a bound on $\mathrm{E}\left(T^{\text{eval}}\right)$.

If $\mathrm{OPT} = \{x^*\}$, we use the same argument to show that within $\frac{2^{n+3}}{n|\mathrm{OPT}|}$ generations we either hit an optimum or a second-best search point. From the latter, the probability of hitting the optimum is bounded in the same way, since by assumption the current search point is different from $\overline{x^*}$. Then we proceed as before. □

Theorem 5.2 yields good results for $\lambda_{\max} = n/2$ if k is very large.

Corollary 5.3. *For the self-adjusting* $(1 + (\lambda, \lambda))$ *GA with* $\lambda_{\max} := n/2$ *on* $\textsc{Jump}_k$ *we have* $E\left(T^{\text{eval}}\right) \leq O(n^2) + 2^{n+1}$ *and* $E(T^{\text{gen}}) \leq O(n) + 2^{n+1}/n$, *for all* k.

For $k > n/\log n$, this is faster than the $(1 + 1)$ EA with the default mutation rate $1/n$ as the latter needs expected time $\Theta(n^k)$.

6 RESETTING λ

Any generic choice of a maximum $\lambda_{\max}$ bears the risk that the $(1 + (\lambda, \lambda))$ GA might get stuck with sub-optimal parameters. A solution to avoid this is to reset λ to 1 if $\lambda = \lambda_{\max}$ and there is another unsuccessful generation. This makes the algorithm cycle through the parameter space in unsuccessful generations. A similar modification was made in [21], where the authors restart the parameter to $\lambda = 1$, but also restart the search from a random individual. We do not restart the search because for functions like $\textsc{Jump}_k$, restarts would run into the same set of local optima with overwhelming probability. In [3, 4], the authors reset λ, but instead of resetting to 1, they reset to the last successful parameter. We argue that, if the next step of the optimisation needs a lower value of λ than this, the algorithm will never use the correct parameter.

In the following, we will analyse the simple strategy of resetting λ to 1 after an unsuccessful generation at $\lambda_{\max} = n$. This strategy takes advantage of two different behaviours. When hill-climbing the algorithm uses self-adjustment to regulate λ and maintain its value in a *good* parameter range, because of this, its optimisation time is not affected for problems like OneMax. However, when the algorithm encounters a local optimum its behaviour is similar to the dynamic $(1 + 1)$ EA [25], cycling through different parameter regions, like $\lambda \sim n/2, \lambda = n$, helping the algorithm simulate random search and the $(1 + n)$ EA in one cycle. In addition, during every

generation the crossover phase is still focusing on exploitation, generating offspring concentrated around the parent.

We show that the fitness-level method can be applied here as well. In contrast to Theorem 3.1, here improvement probabilities refer to the transitions of the $(1 + (\lambda, \lambda))$ GA, and we consider improvement probabilities across a whole cycle of parameter values.

THEOREM 6.1. *Given a canonical f-based partition $A_1, \dots, A_{m+1}$, and s_i^{cycle} a lower bound on the probability of finding an improvement on level i during a cycle. Then for the self-adjusting $(1 + (\lambda, \lambda))$ GA resetting λ to 1, using $F > 1$, we have*

$$E\left(T^{\text{eval}}\right) \leq O\left(\frac{n}{F^{1/4}-1}\right) \sum_{i=1}^{m} \frac{1}{s_i^{\text{cycle}}}$$

$$E\left(T^{\text{gen}}\right) \leq O(\log_F(n)) \sum_{i=1}^{m} \frac{1}{s_i^{\text{cycle}}}$$

PROOF. Since the f-based partition is canonical, the current fitness level is left as soon as we encounter a successful generation. Until this happens, the $(1 + (\lambda, \lambda))$ GA cycles through all parameters for λ. We use Lemma 3.2 to show that for every time the algorithm cycles through all the parameters once, it will use $O(\log_F(n))$ generations and $O\left(\frac{n}{F^{1/4}-1}\right)$ evaluations. The probability of leaving the current fitness level during a cycle is at least s_i^{cycle} by assumption. Hence the expected number of cycles is at most $1/s_i^{\text{cycle}}$. Together, this proves the claimed bounds. □

To showcase how this bound can be used we show an upper bound for $\textsc{Jump}_k$.

THEOREM 6.2. *Let $F > 1$, $k \geq 2$. The expected optimisation time of the self-adjusting $(1 + (\lambda, \lambda))$ GA resetting λ to 1 on $\textsc{Jump}_k$ is*

$$\min\left\{O\left(\frac{n^k}{F^{1/4}-1}\right), O\left(\left(\frac{F^{(k+1)/4}}{F^{1/4}-1}\right)\left(\frac{n}{k}\right)^{k+1}\left(\frac{n}{n-k}\right)^{n-k}\right)\right\}$$

PROOF. For the fitness levels $A_1 \dots A_{m-1}$ any individual can leave the current fitness level by increasing or decreasing the number of 1-bits. For these fitness levels we bound s_i^{cycle} by only considering generations with $\lambda = n$, therefore similar to Theorem 3.1 $s_i^{\text{cycle}} \geq \frac{s_i n}{1+s_i n}$. Using the crude estimate $s_i \geq 1/(en)$, gives us an expected time of $1/s_i^{\text{cycle}} \leq e + 1$ to leave any of these fitness levels.

For the plateau in fitness level A_m, we use $s_m^{\text{cycle}} \geq \max\{s_m^{k^*}, s_m^n\}$ with $s_m^{k^*}$ and s_m^n being the probability of leaving A_m with $\lambda \in \left[\frac{k}{F^{1/4}}, k\right]$ and $\lambda = n$ respectively and bound them separately.

$$s_m^n \geq \frac{s_m n}{1 + s_m n} \geq \frac{1/(en^{k-1})}{1 + 1/(en^{k-1})} \geq \frac{1}{en^{k-1} + 1}$$

For $s_m^{k^*}$ we use Lemma 4.2 and the range $\lambda \in \left[\frac{k}{F^{1/4}}, k\right]$ as follows,

$$s_m^{k^*} \geq \frac{\lambda}{2}(\lambda/n)^k(1-\lambda/n)^{n-k} \geq \frac{k}{2F^{1/4}}\left(\frac{k}{F^{1/4}n}\right)^k\left(1-\frac{k}{n}\right)^{n-k}$$

Applying Theorem 6.1 with $s_m^{\text{cycle}} \geq \max\{s_m^{k^*}, s_m^n\}$ and absorbing the expected times for fitness levels $i < m$ in the asymptotic notation proves the claimed bounds. □

Similar to Bassin and Buzdalov [4], we can slow down the growth of λ. We accomplish this by cleverly choosing F in such a way that the algorithm is able to use every $\lambda \leq n$, ensuring that the algorithm uses the best parameter value. Choosing $F = (1+1/n)^4$ in Theorem 6.2 implies that $F^{(k+1)/4} = (1 + 1/n)^{k+1} \leq (1 + 1/n)^{n+1} = O(1)$, hence this factor can be dropped.

COROLLARY 6.3. *Let $F = (1 + 1/n)^4$, $k \geq 2$. The expected optimisation time of the modified self-adjusting $(1 + (\lambda, \lambda))$ GA on the $\textsc{Jump}_k$ function is*

$$\min\left\{O\left(n^{k+1}\right), O\left(\frac{n^2}{k}\left(\frac{n}{k}\right)^k\left(\frac{n}{n-k}\right)^{n-k}\right)\right\}$$

Comparing the bound of Corollary 6.3 against the expected optimisation time of the $(1 + 1)$ EA with optimal mutation rate of k/n, which is $T_{\text{opt}} = \Theta\left(\left(\frac{n}{k}\right)^k\left(\frac{n}{n-k}\right)^{n-k}\right)$, [18], our bound is larger than T_{opt} by a factor of $O\left(n^2/k\right)$. Our bound is only by a factor of $O(n^2/k^2)$ larger than the bound for the $(1 + 1)$ EA with the heavy-tailed ("fast") mutation operators from [18] with the recommended parameter $\beta = 1.5$.

7 CONCLUSIONS

We have provided a rigorous runtime analysis of the $(1 + (\lambda, \lambda))$ GA for general function classes by presenting a fitness-level theorem for the $(1 + (\lambda, \lambda))$ GA that is easy to use and enables a transfer of runtime bounds for the $(1 + 1)$ EA to the $(1 + (\lambda, \lambda))$ GA.

The parameter control mechanism in the original $(1 + (\lambda, \lambda))$ GA tends to diverge λ to its maximum on multimodal problems. Then the algorithm effectively simulates a $(1+n)$ EA with the default mutation rate of $1/n$. For the multimodal benchmark problem class $\textsc{Jump}_k$, we proved upper and lower runtime bounds that are tight up to lower-order terms, showing that despite using crossover the $(1 + (\lambda, \lambda))$ GA is not as efficient as other crossover-based algorithms.

Imposing a maximum value $\lambda_{\max}$ can improve performance, however then the problem remains of how to set $\lambda_{\max}$ if no problem-specific knowledge is available. The generic choice $\lambda_{\max} = n/2$ makes the $(1 + (\lambda, \lambda))$ GA perform random search steps in case the algorithm gets stuck. This guards against deceptive problems and the algorithm still retains its original exploitation capabilities.

Finally, we investigated resetting λ to 1 after an unsuccessful generation at the maximum value. This makes the $(1 + (\lambda, \lambda))$ GA cycle through the parameter space, approaching optimal or near-optimal parameter values in every cycle. We recommend to choose $F = (1 + 1/n)^4$ if a slow growth of λ is desired. For $\textsc{Jump}_k$, this strategy gives the same expected runtime as that of the $(1 + 1)$ EA with the optimal mutation rate and fast mutation operators, up to small polynomial factors.

ACKNOWLEDGMENTS

We would like to thank an anonymous reviewer for suggesting a simplification to the proof of Lemma 4.3. This research has been supported by CONACYT under the grant no. 739621 and registration no. 843375.

REFERENCES

[1] Denis Antipov, Benjamin Doerr, and Vitalii Karavaev. 2019. A Tight Runtime Analysis for the (1 + (λ, λ)) GA on LeadingOnes. In *Proceedings of the 15th ACM/SIGEVO Conference on Foundations of Genetic Algorithms (FOGA '19)*. ACM, 169–182.

[2] Golnaz Badkobeh, Per Kristian Lehre, and Dirk Sudholt. 2014. Unbiased Black-Box Complexity of Parallel Search. In *Proc. of Parallel Problem Solving from Nature – PPSN XIII*. Springer, 892–901.

[3] Anton Bassin and Maxim Buzdalov. 2019. The 1/5-th Rule with Rollbacks: On Self-Adjustment of the Population Size in the (1 + (λ, λ)) GA. *CoRR* abs/1904.07284 (2019). http://arxiv.org/abs/1904.07284

[4] Anton Bassin and Maxim Buzdalov. 2019. The 1/5-th Rule with Rollbacks: On Self-Adjustment of the Population Size in the (1 + (λ, λ)) GA. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '19)*. ACM, 277–278.

[5] Süntje Böttcher, Benjamin Doerr, and Frank Neumann. 2010. Optimal Fixed and Adaptive Mutation Rates for the LeadingOnes Problem. In *Proc. of Parallel Problem Solving from Nature – PPSN XI*, Vol. 6238. Springer, 1–10.

[6] Maxim Buzdalov and Benjamin Doerr. 2017. Runtime Analysis of the (1 + (λ, λ)) Genetic Algorithm on Random Satisfiable 3-CNF Formulas. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '17)*. ACM, 1343–1350.

[7] Eduardo Carvalho Pinto and Carola Doerr. 2018. Towards a More Practice-Aware Runtime Analysis of Evolutionary Algorithms. *CoRR* abs/1812.00493 (2018). http://arxiv.org/abs/1812.00493

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3rd ed.). The MIT Press.

[9] D. Corus, D. Dang, A. V. Eremeev, and P. K. Lehre. 2018. Level-Based Analysis of Genetic Algorithms and Other Search Processes. *IEEE Transactions on Evolutionary Computation* 22, 5 (2018), 707–719.

[10] Duc-Cuong Dang, Tobias Friedrich, Timo Kötzing, Martin S. Krejca, Per Kristian Lehre, Pietro S. Oliveto, Dirk Sudholt, and Andrew M. Sutton. 2016. Escaping Local Optima with Diversity Mechanisms and Crossover. In *Proc. Genetic and Evolutionary Computation Conference (GECCO'16)*. ACM, 645–652.

[11] Duc-Cuong Dang, Tobias Friedrich, Timo Kötzing, Martin S. Krejca, Per Kristian Lehre, Pietro S. Oliveto, Dirk Sudholt, and Andrew M. Sutton. 2018. Escaping Local Optima Using Crossover With Emergent Diversity. *IEEE Transactions on Evolutionary Computation* 22, 3 (2018), 484–497.

[12] Benjamin Doerr and Carola Doerr. 2018. Optimal Static and Self-Adjusting Parameter Choices for the (1+(λ,λ)) Genetic Algorithm. *Algorithmica* 80, 5 (01 May 2018), 1658–1709.

[13] Benjamin Doerr and Carola Doerr. 2020. *Theory of Parameter Control for Discrete Black-Box Optimization: Provable Performance Gains Through Dynamic Parameter Choices*. Springer, 271–321.

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

[15] Benjamin Doerr, Carola Doerr, and Johannes Lengler. 2019. Self-adjusting mutation rates with provably optimal success rules. *Proc. Genetic and Evolutionary Computation Conference (GECCO'19)* (2019).

[16] Benjamin Doerr, Christian Gießen, Carsten Witt, and Jing Yang. 2017. The (1+λ) Evolutionary Algorithm with Self-Adjusting Mutation Rate. In *Proc. Genetic and Evolutionary Computation Conference (GECCO '17)*. ACM, 1351–1358.

[17] Benjamin Doerr and Timo Kötzing. 2019. Multiplicative Up-Drift. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '19)*. ACM, New York, NY, USA, 1470–1478.

[18] Benjamin Doerr, Huu Phuoc Le, Régis Makhmara, and Ta Duy Nguyen. 2017. Fast Genetic Algorithms. In *Proc. Genetic and Evolutionary Computation Conference (GECCO'17)*. ACM, 777–784.

[19] Benjamin Doerr, Carsten Witt, and Jing Yang. 2018. Runtime analysis for self-adaptive mutation rates. *Proc. Genetic and Evolutionary Computation Conference (GECCO'18)* (2018).

[20] Carola Doerr, Furong Ye, Naama Horesh, Hao Wang, Ofer M. Shir, and Thomas Bäck. 2019. Benchmarking Discrete Optimization Heuristics with IOHprofiler. In *Proc. Genetic and Evolutionary Computation Conference (GECCO'19)*. ACM, 1798–1806.

[21] Brian W. Goldman and William F. Punch. 2014. Parameter-Less Population Pyramid. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation (GECCO '14)*. 785–792.

[22] Walter J. Gutjahr. 2008. First steps to the runtime complexity analysis of ant colony optimization. *Computers and Operations Research* 35, 9 (2008), 2711–2727.

[23] Mario A. Hevia Fajardo. 2019. An Empirical Evaluation of Success-based Parameter Control Mechanisms for Evolutionary Algorithms. In *Proc. Genetic and Evolutionary Computation Conference (GECCO'19)*. ACM, 787–795.

[24] Thomas Jansen and Ingo Wegener. 2002. On the Analysis of Evolutionary Algorithms—A Proof That Crossover Really Can Help. *Algorithmica* 34, 1 (2002), 47–66.

[25] Thomas Jansen and Ingo Wegener. 2006. On the analysis of a dynamic evolutionary algorithm. *Journal of Discrete Algorithms* 4, 1 (2006), 181 – 199.

[26] Jörg Lässig and Dirk Sudholt. 2011. Adaptive Population Models for Offspring Populations and Parallel Evolutionary Algorithms. In *Proceedings of the 11th Workshop Proceedings on Foundations of Genetic Algorithms (FOGA '11)*. ACM, 181–192.

[27] Jörg Lässig and Dirk Sudholt. 2014. Analysis of speedups in parallel evolutionary algorithms and (1+λ) EAs for combinatorial optimization. *Theoretical Computer Science* 551 (2014), 66–83.

[28] Jörg Lässig and Dirk Sudholt. 2014. General Upper Bounds on the Running Time of Parallel Evolutionary Algorithms. *Evolutionary Computation* 22, 3 (2014), 405–437.

[29] Per Kristian Lehre and Carsten Witt. 2012. Black-Box Search by Unbiased Variation. *Algorithmica* 64, 4 (2012), 623–642.

[30] Per Kristian Lehre and Xin Yao. 2009. On the Impact of the Mutation-Selection Balance on the Runtime of Evolutionary Algorithms. In *Proceedings of the Tenth ACM SIGEVO Workshop on Foundations of Genetic Algorithms (FOGA '09)*. ACM, 47–58.

[31] Andrei Lissovoi, Pietro S. Oliveto, and John Alasdair Warwicker. 2019. Simple Hyper-heuristics Control the Neighbourhood Size of Randomised Local Search Optimally for LeadingOnes*. *Evolutionary Computation* (2019), 1–25.

[32] Andrea Mambrini and Dirk Sudholt. 2015. Design and Analysis of Schemes for Adapting Migration Intervals in Parallel Evolutionary Algorithms. *Evolutionary Computation* 23, 4 (2015), 559–582.

[33] Frank Neumann, Dirk Sudholt, and Carsten Witt. 2009. Analysis of Different MMAS ACO Algorithms on Unimodal Functions and Plateaus. *Swarm Intelligence* 3, 1 (2009), 35–68.

[34] Dirk Sudholt. 2013. A New Method for Lower Bounds on the Running Time of Evolutionary Algorithms. *IEEE Transactions on Evolutionary Computation* 17, 3 (2013), 418–435.

[35] Dirk Sudholt and Carsten Witt. 2010. Runtime Analysis of a Binary Particle Swarm Optimizer. *Theoretical Computer Science* 411, 21 (2010), 2084–2100.

[36] Carsten Witt. 2006. Runtime Analysis of the (μ+1) EA on Simple Pseudo-Boolean Functions. *Evolutionary Computation* 14, 1 (2006), 65–86.

[37] Carsten Witt. 2014. Fitness levels with tail bounds for the analysis of randomized search heuristics. *Inform. Process. Lett.* 114, 1–2 (2014), 38–41.