An Empirical Evaluation of Success-Based Parameter Control Mechanisms for Evolutionary Algorithms

Mario Alejandro Hevia Fajardo
University of Sheffield
Sheffield, United Kingdom

ABSTRACT

Success-based parameter control mechanisms for Evolutionary Algorithms (EA) change the parameters every generation based on the success of the previous generation and the current parameter value. In the last years there have been proposed several mechanisms of success-based parameter control in the literature. The purpose of this paper is to evaluate and compare their sequential optimisation time and parallelisation on different types of problems. The geometric mean of the sequential and parallel optimisation times is used as a new metric to evaluate the parallelisation of the EAs capturing the trade off between both optimisation times. We perform an empirical study comprising of 9 different algorithms on four benchmark functions. From the 9 algorithms eight algorithms were taken from the literature and one is a modification proposed here.

We show that the modified algorithms has a 20% faster sequential optimisation time than the fastest known GA on OneMax. Additionally we show the benefits of success-based parameter control mechanisms for NP-hard problems and using the proposed metric we also show that success-based offspring population size mechanisms are outperformed by static choices in parallel EAs.

CCS CONCEPTS

• **Computing methodologies** → **Genetic algorithms**; • **Theory of computation** → Evolutionary algorithms;

KEYWORDS

Parameter Selection, Parameter Control, Empirical Study, Success-based, Genetic Algorithms

ACM Reference Format:
Mario Alejandro Hevia Fajardo. 2019. An Empirical Evaluation of Success-Based Parameter Control Mechanisms for Evolutionary Algorithms. In *Genetic and Evolutionary Computation Conference (GECCO '19), July 13–17, 2019, Prague, Czech Republic.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3321707.3321858

1 INTRODUCTION

Evolutionary algorithms (EAs) are population-based optimisation algorithms, most of them have a set of parameters that need to be tuned or have parameters set to default values by design. Theoretical research has demonstrated that the correct (or incorrect) selection of these parameters impact on the performance of the algorithms [9], therefore parameter tuning and parameter control are key themes in the study of EAs.

Parameter tuning aims to select static parameter settings that are globally optimal for a given algorithm on a given problem. Thanks to theoretical studies in parameter tuning we have a better understanding on the correct selection of parameters for EAs. One of the first results of this research area proved that the standard mutation rate of $p = 1/n$ minimises the expectation time on linear functions [27]. Even though parameter tuning has been essential on our understanding of EAs, these parameter settings generally are suboptimal in some states of the optimisation process in order to find a good compromise for the whole process [9].

In contrast, parameter control addresses this problem by using dynamic parameter settings that search for the best parameters *on the fly*. There are several types of parameter control, in particular fitness-dependant parameter control has proven that dynamic parameter choices can decrease the expected optimisation time of an algorithm in particular problems [1, 2]. These choices are highly problem-tailored, therefore a good understanding of the problem is needed to create a suitable dependence of the parameters. This is often not feasible, for this reason some researchers have focused on success-based parameter control mechanisms, which do not depend on the problem at hand to improve the performance.

Examples of EAs with success-based parameter control are the following. The $(1 + \lambda)$ EA with Two-rate Standard Bit Mutation [11] which uses a self-adjusting mechanism for the mutation rate. This EA was proven to achieve asymptotically the same expected runtime on OneMax as the highly tailored self-adaptive parameter control in [1]. Another group of algorithms are multiplicative success-based rules for the offspring population size λ, i.e. the value is multiplied or divided depending on the success of the previous generation. These group of algorithms have shown to adjust λ well [18, 21]. The last group of algorithms includes the $(1 + (\lambda, \lambda))$ GA [10] and the self-adjusting $(1 + (\lambda, \lambda))$ GA. The self-adjusting $(1 + (\lambda, \lambda))$ GA was proven to optimise OneMax in linear time [8] being the fastest known GA on that problem. We also propose a new modification of the self-adjusting $(1+(\lambda, \lambda))$ GA explained in Section 4. A broader review on $(1 + \lambda)$ EA and success-based mechanisms is done in Section 3.

In this paper we aim to complement the theoretical studies on EAs with success-based parameter control mechanisms with an empirical study that evaluates both non-parallel and parallel computing. With this aim we also want to contribute in reducing the gap between mathematical and empirical research on EAs as done by other studies such as Doerr et al. [14] where the authors study some of the same algorithms studied here. In order to achieve our

GECCO '19, July 13–17, 2019, Prague, Czech Republic

ACM ISBN 978-1-4503-6111-8/19/07…$15.00
https://doi.org/10.1145/3321707.3321858

aim we use the sequential optimisation time and the geometric mean of the sequential and parallel optimisation times, which is a new performance measure for parallel computing. A discussion on the benefits of this metric is made in Section 5.

We focus on evaluating EAs with success-based parameter control mechanisms that were proposed and study in theoretical papers on four different functions; OneMax, LeadingOnes, SufSamp and Makespan Scheduling which are explained in Section 2.

We found that for complex problems such as Makespan Scheduling, the (1+λ) EA with Two-rate Standard Bit Mutation outperforms all other algorithms that use the static mutation probability $p = 1/n$. We confirmed that all the $(1 + \lambda)$ EAs with Success-Based Offspring Population Size have similar sequential optimisation time than the best static values. Also the modified Self-Adjusting $(1 + (\lambda, \lambda))$ GA improves by 20% the sequential optimisation time of the original Self-Adjusting $(1 + (\lambda, \lambda))$ GA on OneMax for problem sizes $n = \{100, 500, 5000\}$.

The paper is structured as follows. The four benchmark functions are defined in Section 2. Section 3 gives a background on parameter control and introduces the EAs with success-based parameter control mechanisms that are evaluated here. The proposed modification to the Self-Adjusting $(1 + (\lambda, \lambda))$ GA is shown in Section 4. Section 5 explains the motivation behind the use of the geometric mean and its benefits. Section 6 shows the experiment results and discussion. Finally we finish with the conclusions in Section 7.

2 ARTIFICIAL LANDSCAPES

The artificial landscapes used in this study were selected for different reasons. OneMax and LeadingOnes are used as a baseline because they are popular benchmark functions for theoretical results based on runtime analyses [2, 5, 10, 11, 13, 14, 16–18, 20, 26]. Makespan Scheduling is included to consider an NP-hard problem from combinatorial optimisation. Finally SufSamp [18] was chosen because it is designed to trap small offspring population sizes making it interesting for self-adjusting population sizes.

2.1 OneMax

The OneMax function was designed as a simple benchmark function. $\textsc{OneMax} : \{0,1\} \to \mathbb{N}$ is defined by

$$\textsc{OneMax}(x) := \sum_{i=1}^{n} x_i \text{ for all } n \in \mathbb{N} \text{ and all } x \in \{0,1\}^n$$

Since it is one of the simplest functions, it has been studied profoundly and in particular is known as the easiest problem [5, 12] for the (1 + 1) EA, which is a special case of (1 + λ) EA where $\lambda = 1$. The optimum of the function is a bit-string with only ones.

2.2 LeadingOnes

The function $\textsc{LeadingOnes} : \{0,1\} \to \mathbb{N}$ is defined by

$$\textsc{LeadingOnes}(x) := \sum_{i=1}^{n} \prod_{j=1}^{i} x_j \text{ for all } n \in \mathbb{N} \text{ and all } x \in \{0,1\}^n$$

The LeadingOnes function has the same optimum as OneMax but the fitness function is different. To calculate the fitness of an individual, the number of consecutive ones are counted starting from the left and stop whenever a zero is found.

2.3 SufSamp

This function was created by Jansen et al. [18] as a problem where an EA with $\lambda > 1$ would have a faster expected optimisation time than using $\lambda = 1$. Its fitness landscape has a single narrow steep path to the global optimum and several branches guiding to a local optimum with a more gradual uphill path. With these characteristics an EA would need to have a sufficient large offspring population size in order to increase the probability to find the right path at every branch point B_n or jump out of a local optimum. In the function there are about $\sqrt{n}$ number of B_n that can lead to a local optimum.

For $n \in \mathbb{N}$, $k := \left\lfloor \sqrt{n} \right\rfloor$ and $|x| = \textsc{OneMax}(x)$. The function $f : \{0,1\}^n \to \mathbb{N}$ for all $x \in \{0,1\}^n$ is:

$$f(x) := \begin{cases} (i+3)n + |x| & \text{if } (x = 0^{n-i}1^i \text{ with } 0 \le i \le n) \text{ or} \\ & (x = y0^{n-i-k}1^i \text{ with} \\ & i \in \{k, 2k, \ldots, (k-2)k\},\ y \in \{0,1\}^k) \\ 0 & \text{otherwise.} \end{cases}$$

With the previous definition of f a start point of 0^n is needed, in order to avoid this, f is extended into SufSamp as follows:

For $n \in \mathbb{N}$, $m' := \lfloor n/2 \rfloor$, $m'' := \lceil n/2 \rceil$. Then $\textsc{SufSamp}: \{0,1\}^n \to \mathbb{N}$ is defined as:

$$\textsc{SufSamp}(x) := \begin{cases} n - \textsc{OneMax}(x'') & \text{if } x' \neq 0^{m'} \wedge x'' \neq 0^{m''} \\ 2n - \textsc{OneMax}(x') & \text{if } x' \neq 0^{m'} \wedge x'' = 0^{m''} \\ f(x'') & \text{if } x' = 0^{m'} \end{cases}$$

for all $x = x_1 x_2 \cdots x_n \in \{0,1\}^n$ with $x' := x_1 x_2 \cdots x_{m'} \in \{0,1\}^{m'}$ and $x'' := x_{m'+1} x_{m'+2} \cdots x_n \in \{0,1\}^{m''}$

2.4 Makespan Scheduling

The makespan scheduling problem is a combinatorial optimisation problem, where the scheduling of n jobs in m machines takes place. Within this problem there are different sets of constraints that can be applied, in this work the description of Makespan Scheduling on 2 machines is used as given by Neumann and Witt [22].

Makespan scheduling on 2 machines. There are n jobs with positive processing times $p_1, p_2, \ldots, p_n$ and need to be scheduled on two identical machines in order to minimise the makespan. To encode the machines a bit-string $x \in \{0,1\}^n$ is used where the job i is scheduled on machine 1 if the $x_i = 0$ and on machine 2 otherwise. The fitness function is described by:

$$f_{p_1,\ldots,p_n}(x) := \max\left\{ \sum_{i=1}^{n} p_i x_i, \sum_{i=1}^{n} p_i (1 - x_i) \right\} \tag{1}$$

For this problem a common practice is to initialise the processing times randomly. Because of this, the solution of each problem instance is not known beforehand.

An approximation of the solution on the makespan scheduling problem is used as stopping criterion, in order to avoid the complexity of calculating an exact solution. The algorithm used as approximation is called Longest Processing Time (LPT), it consists in sorting the jobs by processing time decreasingly, and using the sorted jobs to put every job in the currently emptier machine [22].

3 BACKGROUND

In this section, we introduce the success-based EAs that are evaluated. We start with the basic $(1 + \lambda)$ EA and later we explain the success-based parameter control mechanisms implemented in each algorithm.

3.1 The $(1 + \lambda)$ EA

The $(1 + \lambda)$ EA is one of the simplest EAs. It uses a parent which is mutated into λ offspring with a mutation operator that flips each bit independently with mutation probability p, afterwards an elitist selection step decides the parent of the next generation. All of the EAs shown here are based on this algorithm.

In the literature there have been several runtime analysis of the $(1 + \lambda)$ EA on OneMax [5, 13, 16–18] and LeadingOnes [2, 14, 18]. From these studies we have a better understanding of how to select the best parameter settings for each problem.

3.1.1 Mutation rate in $(1 + \lambda)$ EA. The mutation probability p is described as the independent probability of flipping each bit, and its formula is $p = c/n$ with mutation rate c and number of bits n. Traditionally the mutation probability of $p = 1/n$ has been used in (1 + 1) EA as a rule of thumb, but this was not formally proven to be the best choice at first. Later a tight (up to lower order terms) bound of $(1 \pm o(1))\frac{e^c}{c} n \ln n$ for all linear functions was proven by Witt [27]; given that this bound has the factor $\frac{e^c}{c}$ that depends on the mutation rate, it can be derived that the mutation probability $p = c/n$ with $c = 1$ is optimal to minimise the specified bound.

This optimal probability was later generalised for the $(1 + \lambda)$ EA on the OneMax function by Gießen and Witt [16] were they found that the expected parallel optimisation time (i.e. number of generations) is equal to

$$(1 \pm o(1)) \left(\frac{e^c}{c} \cdot \frac{n \ln n}{\lambda} + \frac{1}{2} \cdot \frac{n \ln \ln \lambda}{\ln \lambda} \right)$$

From this we can infer that if the value of λ is smaller than a certain threshold, the first part of the function is predominant and the optimal mutation rate is still $c = 1$, and after that threshold the value of c does not change the expected runtime, up to lower order terms; the threshold value or cut-off point of λ is $\ln(n) \ln(\ln(n)) / \ln(\ln(\ln(n)))$.

Even though these results help us choosing parameters, they do not imply that for every instance, problem and algorithm the best choice of mutation rate is $c = 1$, for example small values of n in OneMax result in a small increase on the optimal mutation rate [4] and also the optimal value for the mutation probability on LeadingOnes is $p \approx 1.59/n$ [2] with $\lambda = 1$ and can vary for different values of λ [14].

Furthermore it has been demonstrated that the implementation of fitness-dependent adaptive schemes for the mutation rate is faster compared to the mutation rate $c = 1$ for the $(1 + \lambda)$ EA on OneMax [1] and LeadingOnes [2].

Later a success-based parameter control mechanism for the mutation rate in the $(1 + \lambda)$ EA was proven to achieve asymptotically the same expected number of fitness evaluations as the fitness-dependent mutation rate for the OneMax problem [11].

3.1.2 Offspring population in $(1 + \lambda)$ EA. There have been a wide range of studies on the behaviour of the offspring population size, especially in the $(1 + \lambda)$ EA. In Jansen et al. [18] the authors made a broad study of the impact of λ; they found that for simple landscapes setting λ in the range of $1 \leq \lambda \leq \log n$ grant asymptotically equivalent sequential optimisation times, and specifically for OneMax and LeadingOnes there is no benefit in increasing λ beyond $\lambda = 1$. In contrast with these landscapes the authors also showed that for more complex problems there is a benefit on using large values for λ; the tailored function SufSamp (explained in Section 2.3) was presented with such characteristics.

In the study above the focus was made on sequential processing time but for parallel processing time it has been shown that there is a linear improvement in parallel optimisation time on OneMax [16] and LeadingOnes [18] by increasing λ. This is true until a cut-off value of $\lambda = o(\ln(n) \ln(\ln(n)) / \ln(\ln(\ln(n))))$ and $\lambda = O(n)$ respectively.

Given the knowledge that an improvement can be made by parallelisation several models to exploit this have been studied [19, 20]. Lässig and Sudholt [21] also described a parallel model that can be used with success-based offspring population, they used two parameter control mechanisms that are explained in Section 3.3.

3.1.3 Crossover and Genetic Algorithms (GA). One of the main operators of EAs is the crossover (also called recombination). This operator is inspired by sexual reproduction in nature, and essentially it combines two parents into one offspring; there are several ways to implement crossover, and empirical research has been made showing their performance on different problems [23]. From a range of seven crossover operators uniform crossover performs better in almost all problems tested.

Crossover is not used in the basic $(1 + \lambda)$ EA, but it has been believed to improve the optimisation time compared against EAs with only mutation. This was empirically proved by Picek and Golub [23], but only on artificially constructed functions, later Sudholt [26] proved that crossover is at least twice as fast for building-block functions like OneMax, as long as diversity is enforced. Following this Corus and Oliveto [6] proved that GAs with $\mu \geq 3$ are at least 25% faster than all unbiased standard bit mutation-based EAs with static mutation rate for OneMax even if no diversity is enforced.

3.2 The $(1 + \lambda)$ EA with Two-rate Standard Bit Mutation

This algorithm was proposed in Doerr et al. [11]. The general idea of the mutation scheme is to adjust the mutation strength according to its success in the population, in order to do this it creates $\lambda/2$ offspring with a mutation probability $p = c/(2n)$ and the other $\lambda/2$ offspring with $p = 2c/n$. The mutation rate c is initialised as $c^{init} \geq F$ (F being the mutation update factor) and adjusted at every iteration with 50% probability to $c_{new} = c/F$ or $c_{new} = Fc$ depending on the mutation probability of the fittest offspring, and 50% probability to a random value in $\{c/F, Fc\}$. If at any time the mutation rate goes outside of the boundaries $[F, n/(2F)]$, the mutation rate is replaced by the boundary exceeded. The pseudocode can be found in the original paper [11].

3.3 The $(1 + \lambda)$ EA with Success-Based Offspring Population Size

All of the success-based mechanisms described in this section are multiplicative updates, where the offspring population size is multiplied if there is no improvement in fitness and divided otherwise.

A simple success-based offspring population size was proposed by Lässig and Sudholt [21] where the authors double the offspring population size in an unsuccessful generation (i.e. no fitness improvement) to help finding improvements. In order to reduce the offspring population size on a successful generation and maintain a "healthy" size they used two approaches. The first approach is to set the offspring population size to 1, this might help if it becomes easier to find an improvement after a success, but if the landscape does not change, it also makes sense to have a similar offspring population size, therefore in the second approach they only halve the offspring population size after a successful generation. From their behaviour they have been called $(1+\{2\lambda,1\})$ EA and $(1+\{2\lambda,\lambda/2\})$ EA [9, 14]. We generalise the algorithms with an offspring population update factor G and end up with the $(1+\{G\lambda,1\})$ EA and $(1+\{G\lambda,\lambda/G\})$ EA where G can be any value larger or equal to 1.

Another update rule was proposed by Jansen et al. [18] where λ is multiplied by 2 in an unsuccessful generation and divided by s otherwise, with s being the number of successful offspring. The reasoning given in their work was to set λ to roughly the reciprocal of the success probability, minimising the total expected optimisation time. Again we generalise this idea as the $(1+\{G\lambda,\lambda/s\})$ EA.

The last update rule is based on the idea of using the 1/5th rule to update the offspring population size [8]. In this case if a generation is not successful the offspring is multiplied by $G^{1/4}$ and divided by G otherwise. Following the same name convention this EA is called $(1+\{G^{1/4}\lambda,\lambda/G\})$ EA.

3.4 The $(1+(\lambda,\lambda))$ GA

The $(1+(\lambda,\lambda))$ GA was first proposed by Doerr et al. [10]. It works by first mutating the parent λ times, then applying a uniform crossover λ times between the parent and the fittest mutated offspring, in the end it performs an elitist selection. This GA has a variant of the standard mutation operator, that is equivalent to it. The mutation operator is called $\mathrm{mut}_\ell(.)$. First it chooses ℓ different positions in $[n]$ uniformly at random and then it flips the values in those bits in the original bit-string to create the mutated bit-string. The variable ℓ is sampled from a binomial distribution $\mathrm{B}(n,p)$, where p denotes the mutation probability. In this algorithm ℓ is only sampled once per generation, making all mutation offspring have the same distance to the parent.

A benefit of the operator, as shown in [24] is that you can also use the conditional distribution $\mathrm{B}_{>0}(n,p)$ which resamples ℓ when $\ell=0$ to avoid not flipping bits, or simply use 1 when ℓ is sampled as 0.

3.5 The Self-Adjusting $(1+(\lambda,\lambda))$ GA

In the same paper where the $(1+(\lambda,\lambda))$ GA was proposed [10], the authors used a success-based parameter control based on the 1/5th rule. The algorithm updates λ every generation with a multiplicative update where λ is multiplied by a factor $G^{1/4}$ if there is no improvement in fitness and divided by G otherwise. The mutation rate and the crossover probability are parametrised by $p=\lambda/n$ and $c=1/\lambda$. Since all parameters are a function of λ, the only parameter to adjust by the user is the update factor G. This algorithm was proven to optimise OneMax in linear time[8], showing for the first time that self-adjusting parameter choices can be beneficial in discrete optimization problems.

4 THE SELF-ADJUSTING $(1+(\lambda,\lambda))$ GA WITH OPTIMAL STOPPING THEORY

In this section we explain the modifications made to the original $(1+(\lambda,\lambda))$ GA and the motivation behind them.

We take inspiration from the field of optimal stopping theory. There is a wide variety of problems that are considered part of the optimal stopping theory [15], we consider the results found in the variation called secretary problem or marriage problem. The simplest form of this problem is described by Ferguson [15], where there is a secretary position and there are n number of applicants. The applicants are interviewed sequentially in random order, you can rank all applicants without ties and you must decide to reject or accept each applicant based on the relative ranks of the previous interviews; the catch is that you cannot recall a rejected applicant. The optimal solution to this problem is to wait until you have interviewed approximately 36.78% $(1/e)$ of the applicants and then select the next relatively best one, this gives you approximately $1/e$ probability of choosing the best one.

We use these results to create a new selection mechanism and use this mechanism to select the second parent used in crossover from the first λ offspring. The selection of a parent does not have the same rules as the secretary problem, therefore the results of this algorithm might be improved by the use of different percentages.

In the selection mechanism first we sample and evaluate $x^{(i)} \leftarrow \mathrm{mut}_\ell(x)$ for $i \in \{1,\dots,\lfloor 0.37\lambda\rfloor\}$, choose the fittest x' breaking ties uniformly at random. Then we sample $x^{(i)} \leftarrow \mathrm{mut}_\ell(x)$ for $i \in \{\lceil 0.37\lambda\rceil,\dots,\lambda\}$, evaluate and compare them against x' one by one, if we find an individual $x^{(i)}$ that is better than x' we stop and do not evaluate the remaining individuals. If we do not find such individual all the λ offspring will be evaluated and x' is selected. We also use a new update rule for the parameter λ. The pseudocode of the complete algorithm is shown in Algorithm 1

5 EVALUATING THE PERFORMANCE OF A PARALLEL EA

In the EA research in order to evaluate the performance of an EA a common approach is to use the number of evaluations that takes to solve a problem, this is because it is similar to the wall-clock time. Following this idea for a parallel EA if we consider that the execution of a generation dominates the computational effort it may sound reasonable to use the number of generations. We argue against this performance metric because for the most common benchmark problems for EAs such as OneMax and LeadingOnes it has been proven that an increase of the offspring population size creates a speed-up on the parallel optimisation time but this also increases the sequential optimisation time [18].

Other performance measures used for parallel EAs can be found in [25], where the author define a *speedup* comparing the parallel run time against a baseline. This baseline can be the sequential run time of the best known sequential algorithm (*Strong speedup*), a panmictic version of the parallel EA, or the parallel EA running in a single computer. Later the author use this *speedup* to calculate the *efficiency*, normalising the *speedup* with the number of computers or parallel threads used. Even though the *efficiency* could be considered a good measure of performance in special cases, it does not take into account the total computer resources used.

Algorithm 1: The self-adjusting $(1 + (\lambda, \lambda))$ GA with mutation probability p, crossover probability c, update strength G, and optimal stopping theory for parent selection.

Initialisation: Sample $x \in 0, 1^n$ uniformly at random and query $f(x)$;
Initialise $\lambda \leftarrow 3$;
Optimisation: **for** $t = 1, 2, \ldots$ **do**
  Initialise $h \leftarrow 0$;
  Mutation phase:
  Sample ℓ from $\mathcal{B}(n, p)$;
  for $i = 1, \ldots, \lfloor 0.37\lambda \rfloor$ **do**
    Sample $x^{(i)} \leftarrow \text{mut}_\ell(x)$ and query $f(x^{(i)})$;
  Choose $x' \in \{x^{(1)}, \ldots, x^{(\lfloor 0.37\lambda \rfloor)}\}$ with $f(x') = \max\{f(x^{(1)}), \ldots, f(x^{(\lfloor 0.37\lambda \rfloor)})\}$ u.a.r.;
  for $i = \lceil 0.37\lambda \rceil, \ldots, \lambda$ **do**
    Sample $x^{(i)} \leftarrow \text{mut}_\ell(x)$ and query $f(x^{(i)})$;
    if $f(x^{(i)}) > f(x')$ **then**
      $x' \leftarrow x^{(i)}$;
      $h \leftarrow 1$;
      break
  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 $y \in \{x', y^{(1)}, \ldots, y^{(\lambda)}\} \backslash \{x\}$ 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 $f(y) > f(x)$ **and** $h = 0$ **then** $x \leftarrow y$; $\lambda \leftarrow \max\{\lambda/G, 3\}$;
  if $f(y) > f(x)$ **and** $h = 1$ **then** $x \leftarrow y$;
  if $f(y) = f(x)$ **then** $x \leftarrow y$; $\lambda \leftarrow \min\{\lambda G^{1/4}, n\}$;
  if $f(y) < f(x)$ **then** $\lambda \leftarrow \min\{\lambda G^{1/4}, n\}$;

We argue that in order to compare a parallel EA, we need to take into account both the parallel and sequential optimisation times. For simple problems such as OneMax, a reasonable way to do this is by calculating the contribution that every extra offspring makes to the increase of evaluations (C_t) and the decrease of generations (C_g) (2). To calculate the contributions first we calculate the increase in evaluations subtracting the number of evaluations with $\lambda > 1$ (t_λ) to the number of evaluations with $\lambda = 1$ (t_1), afterwards we normalise this increase dividing it by t_1. At last we divide it by $\lambda - 1$ to know the contribution that every extra offspring is making to the normalised increase of evaluations. For the generations we use the same calculation, but since we expect a decrease of generations, we subtract the number of generations with $\lambda = 1$ (g_1) to the number of generations with $\lambda > 1$ (g_λ).

$$C_t = \frac{t_\lambda - t_1}{t_1 \cdot (\lambda - 1)} \qquad C_g = \frac{g_1 - g_\lambda}{g_1 \cdot (\lambda - 1)} \tag{2}$$

In Figure 1 we show these contributions on OneMax with $n = 100$. The intersection of C_t with C_g ($\lambda = 43$) point to what we argue is the best trade off between the number of generations and the number of evaluations, because each extra offspring contributes the same proportion to the increase of evaluations as to the decrease of generations. This is true on OneMax because this is the only point where the contributions intersect and an increase of λ will always decrease C_g and increase C_t.

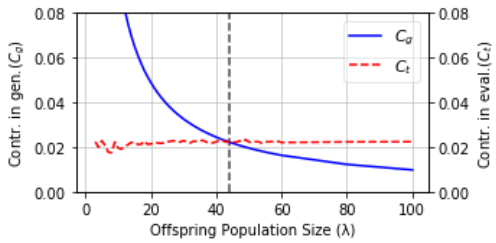

Figure 1: Contributions in evaluations and generations per offspring of $(1 + \lambda)$ EA on OneMax

More complex problems might have several intersections, making the use of the contributions to evaluate parallel EAs not feasible. In order to use a similar approach of considering the contributions, we propose a metric that aggregates the parallel and sequential optimisation times in a meaningful single quantity that is able to rank all algorithms and all problems. We propose the geometric mean (GM (3)) of the number of evaluations t_λ and generations g_λ for this purpose.

$$\text{GM} = \sqrt{t_\lambda \cdot g_\lambda} \tag{3}$$

The GM has several benefits, firstly it gives the same weight to t_λ and g_λ, for example if we double t_λ and halve g_λ the GM would stay the same. Secondly it makes an easy comparison between two algorithms because there is no baseline needed. Lastly it take into account the importance of the wall-clock time of parallel EAs in form of the parallel optimisation time and also use the sequential optimisation time to avoid the waste of computer resources in unnecessary evaluations.

To illustrate these benefits we compare the GM of the $(1 + \lambda)$ EA on OneMax with $n = 100$ using different values of λ and 500 runs per instance. We can see in Figure 2 that the GM decreases when increasing the value of λ up until $\lambda = 43$ where the number of evaluation starts to increase more rapidly than the decrease in generations, which is reflected in an increase of GM.

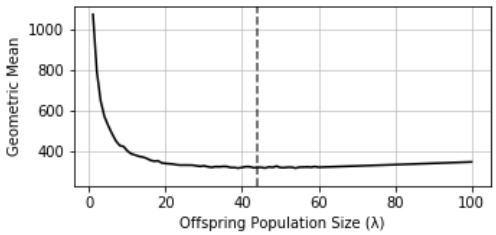

Figure 2: Geometric mean of the $(1+\lambda)$ EA on OneMax with $n = 100$

From the empirical results the three best GM in the tested λ are achieved at $\lambda = 39, 43, 46$. In theory for OneMax we would expect a smooth curve with only one minimum, but due to the randomness in the results there where 3 local minima. If we compare these results with the previous calculation of contributions, it

shows the effectiveness of the GM to select the best λ capturing the relationship between sequential and parallel optimisation times.

6 EXPERIMENTAL RESULTS

In this section we show the detailed setup of the experiment followed by the comparison of the algorithms with the $(1 + \lambda)$ EA as a baseline.

6.1 Experiment Design

We compare the sequential optimisation time and the GM of the 9 algorithms described in Sections 3 and 4. Each algorithm is tested 500 runs on the four landscapes, with the following problem settings. The dimensionality n of the fitness landscape is set to 100 for all the problems; in Makespan Scheduling the bit strings were initialised to $x \in \{0\}^n$ and the weights were sampled from a uniform distribution over the half-open interval $[0, 1)$, each algorithm was tested using the same 500 sets of weights.

An important issue was the choice of appropriate values for the non self-adjusted parameters in each algorithm. For all $(1 + \lambda)$ EA variants, we chose small values of offspring population size $\lambda \in \{1, 2, 8, 10, 16, 40\}$[1] because we are interested in the linear improvement in parallel optimisation time below the λ cut-off point. For the $(1 + (\lambda, \lambda))$ GA, we used small offspring population sizes of $\lambda \in \{8, 10, 16, 40\}$ guiding us in previous analyses [8, 10]. Additionally bigger values of offspring population size $\lambda \in \{100, 400, 600\}$ were chosen for all algorithms to visualise the improvement in sequential optimisation time for more complex landscapes.

The mutation probability was set to $p = 1/n$ for all $(1 + \lambda)$ EA variants on OneMax, SufSamp and Makespan Scheduling. On LeadingOnes the static value of $p = 1.59/n$ was used. For the $(1 + (\lambda, \lambda))$ GA there were two different settings for each value of λ , in the first one the mutation probability was set to $p = \lambda/n$ and the crossover probability of $c = 1/\lambda$, in the second parameter setting the mutation and crossover probability were $p = 6/n$ and $c = 1/6$.

At last the update factor in the algorithms was set between 1 and 2 following the recommendation of the authors of such algorithms. The values used were $F \in \{1.2, 2\}$ and $G \in \{1.1, 1.2, 1.5, 2\}$[2]. Given the large number of parameter settings we only show the performance of each algorithm with the best combination of settings found and interesting findings. The code used and raw optimisation times are given as supplementary material.

6.2 Results and Discussion

6.2.1 OneMax. From the experiments we made several observations that confirm previous theoretical and empirical studies. We observed that smaller offspring population sizes in the family of $(1 + \lambda)$ EAs yield faster sequential optimisation times as previously known by theoretical analyses [18]. In addition we found that for small problem sizes the $(1 + (\lambda, \lambda))$ GA can be better than its self-adjusting counterpart when using $\lambda = 8$, $p = 6/n$ and $c = 1/6$ as shown in Figure 3. Furthermore we corroborate that all the $(1 + \lambda)$ EA with success-based offspring population size tested can regulate with great precision the value of λ having similar sequential optimisation times as the $(1 + 1)$ EA.

We also made new findings with the test of the modified algorithm. The Self-Adjusting $(1 + (\lambda, \lambda))$ GA with optimal stopping theory and $G = 1.5$ achieved the best results in the experiments using 20% less evaluations than the original Self-Adjusting $(1 + (\lambda, \lambda))$ GA, which is the fastest known GA. We also tested on larger problem sizes of $n = \{500, 5000\}$ with a 19-20% speedup. Analysing the tests we found that the modified version used less generations than the original which suggests two things; the selection mechanism is not affecting the improvements, and for small n the update rule used combined with the limit of $\lambda \geq 3$ decreases the number of generations.

To test this further we did 500 runs using the modified algorithm with the original update rule and the original algorithm with $\lambda \geq 3$. We found that on average both the original and the modified algorithms use the same number of generations and there is still a 15% speedup in sequential optimisation time for $n = 100$. We also tested another variation that only creates one offspring in the mutation phase that we call Self-Adjusting $(1 + (1, \lambda))$ GA and it performed badly, which indicates that the selection mechanism proposed is choosing a good parent for crossover.

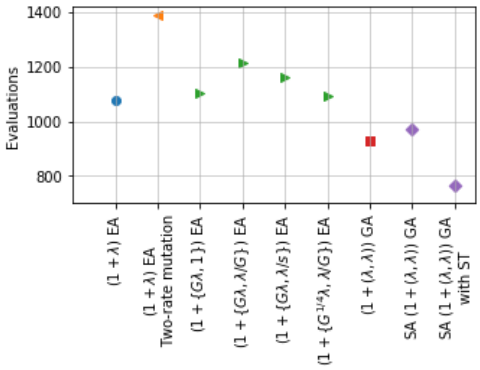

Figure 3: Sequential optimisation time on OneMax with $n = 100$

The results found for the Geometric Mean of the algorithms on OneMax show that the success-based algorithms are worse than static parameter choices, even taking into account the benefit of easier choice of parameter settings. For example the $(1 + \lambda)$ EA outperformed all other success-based algorithms (including Self-Adjusting $(1 + (\lambda, \lambda))$ GAs) with any offspring population size in the interval $\lambda = [20, 80]$.

To calculate the GM for the family of $(1 + (\lambda, \lambda))$ GAs the number of generations was multiplied by 2 to take into account that the evaluations needed in the mutation and crossover steps cannot be made in parallel. Even with this consideration, the best algorithm was the $(1 + (\lambda, \lambda))$ GA with $\lambda = 16$, $p = 6/n$ and $c = 1/6$ followed by the $(1 + \lambda)$ EA with $\lambda = 40$ and the $(1 + \lambda)$ EA with Two-rate Standard Bit Mutation with $\lambda = 100, F = 1.2$. These results are shown in Figure 4.

Given the nature of the Self-Adjusting $(1 + (\lambda, \lambda))$ GA with optimal stopping theory it is not shown in Figure 4, because most of its evaluations need to be done sequentially.

6.2.2 LeadingOnes. On LeadingOnes, the family of $(1 + \lambda)$ EAs had similar results as on OneMax as shown in Figure 5. All the $(1+\lambda)$ EAs with success-based offspring population size tested were able to select good values for λ *on the fly* to reduce the sequential optimisation time.

[1] $\lambda = 1$ was not used in $(1 + \lambda)$ EA with Two-rate Standard Bit Mutation
[2] Not all parameters where tested in all algorithms

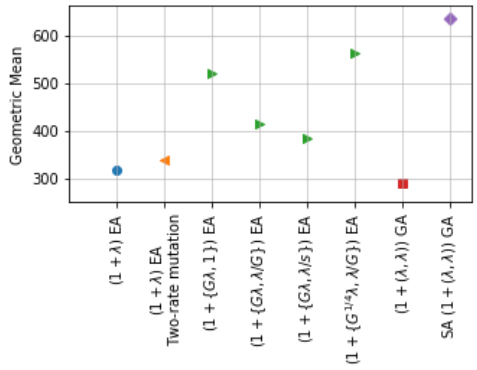

Figure 4: Geometric mean on OneMax with $n = 100$

An important result is that all the algorithms in the family of $(1 + (\lambda, \lambda))$ GAs performed worse than any of the $(1 + \lambda)$ EAs variations. To the best of our knowledge there are no empirical or theoretical results of $(1 + (\lambda, \lambda))$ GAs on LeadingOnes making this the first study to show that they do not perform well.

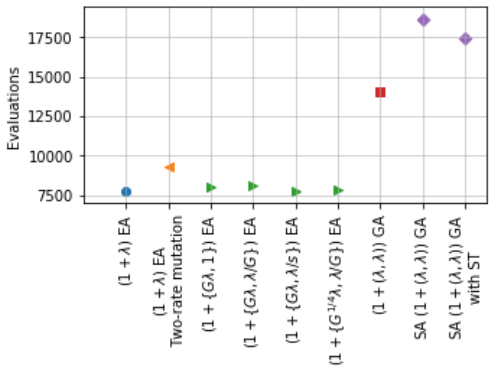

Figure 5: Sequential optimisation time on LeadingOnes with $n = 100$

From Figure 6 we can see that unlike with the sequential optimisation time, the variations of $(1 + (\lambda, \lambda))$ GA had a smaller GM than the $(1 + \lambda)$ EAs with success-based offspring population size due to greatly reducing the parallel optimisation time. Still their GM is at least the double of the simple $(1 + \lambda)$ EA with $\lambda = 100$ and worse than any value of λ in the interval $[16, 600]$. The only success-based EA with good GM was the $(1 + \lambda)$ EA with two-rate standard bit mutation having similar results than the $(1 + \lambda)$ EA with all values of λ.

6.2.3 Makespan Scheduling. On Makespan Scheduling there was a trend in almost all algorithms to solve most of the instances with a relatively small number of evaluations (10 - 10,000 evaluations) but for difficult instances they took hundreds of thousands or even millions of evaluations to get to the solution. In general the static choices for the offspring population size with less evaluations, were between $\lambda = 10$ and $\lambda = 16$.

The $(1 + \lambda)$ EA with Two-rate Standard Bit Mutation had by far the best sequential optimisation time on Makespan Scheduling with $\lambda = 16, F = 2$ as shown in Figure 7. It was better than all other algorithms with all parameter settings tested; the only exception was $\lambda = 100, F = 1.2$ which was worse than the $(1 + \lambda)$ EA with $\lambda =$

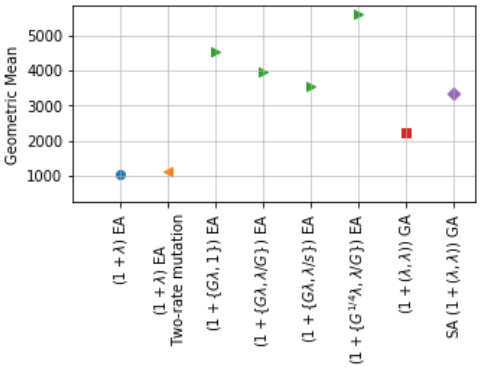

Figure 6: Geometric mean on LeadingOnes with $n = 100$

10. From these results we can imply that the mutation probability of $p = 1/n$ is far from optimal on this problem, and studying the values taken by the $(1+\lambda)$ EA with Two-rate Standard Bit Mutation during the optimisation, we notice that the average mutation probability per instance ranges from $2/n$ to $1/4$ (all possible values) indicating that it is instance dependant and the average over all instances is around $p = 6/n$ which is much higher than the standard $p = 1/n$.

These high mutation rates may allow the EAs escape local optima by finding better searching points far away from the current ones. A similar result can be seen in [7] where it was proved that Artificial Immune Systems with hypermutations can find a $(1 + \epsilon)$ approximation in expected time polynomial in n.

All of the Self-Adjusting $(1+(\lambda, \lambda))$ GAs had a poor performance. We attribute this to the relationship between the offspring population size and the other parameters (i.e. $p = \lambda/n$ and $c = 1/\lambda$). We base this assumption on the tests made with the $(1 + (\lambda, \lambda))$ GA; when using $p = 6/n$ and $c = 1/6$ it had a performance similar to other algorithms but with $p = \lambda/n$ and $c = 1/\lambda$ it needed up to 5 times more evaluations to get to the optimum.

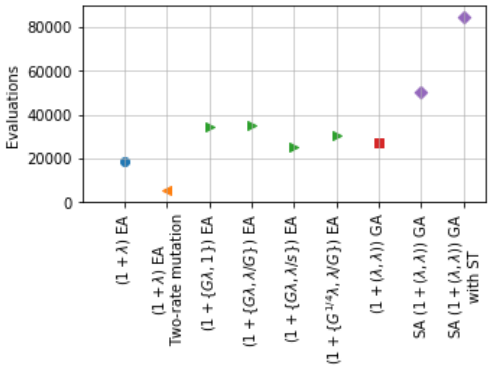

Figure 7: Sequential optimisation time on Makespan Scheduling with $n = 100$

On Figure 8 we see that the GM values of the $(1 + \lambda)$ EAs with success-based offspring population size were similar to the best $(1 + \lambda)$ EA, showing that these algorithms can be used in parallel computing for more complex problems. These might be explained by the big offspring population sizes (400, 600) that minimised the GM for static choices. Another possible explanation is that there is only a small increase in evaluations with larger offspring population size and a substantial decrease of generations.

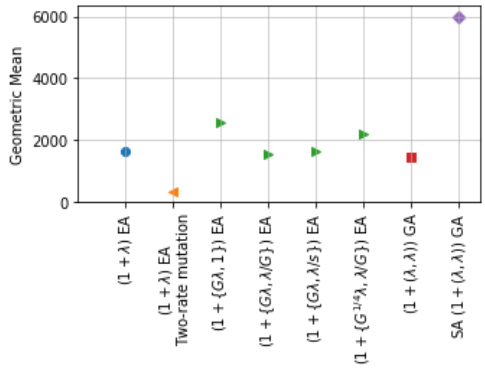

Figure 8: Geometric mean on Makespan Scheduling with $n = 100$

6.2.4 SufSamp. For SufSamp we implemented an early stop on the optimisation process because of constraints in computation time. The optimisation was stopped if the algorithm was in a local optimum and more than one million evaluations were made without improvement. This might have caused a lower success rate and lower average fitness, affecting the most algorithms with unconstrained self-adjusting offspring population size, but comparing our results with the ones obtained by Jansen et al. [18] we consider that the effects were minimal.

Unlike the other problems in SufSamp we evaluate the average fitness of the algorithms (Figure 9) instead of the sequential optimisation time and the GM. This choice was made because of the low success rate of every algorithm (0% - 35%) which makes impossible to compare the algorithms as with previous problems.

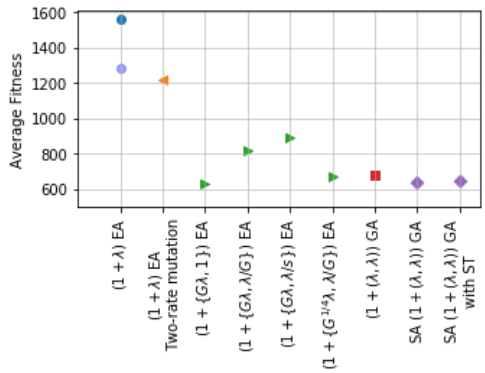

Figure 9: Average Fitness on SufSamp with $n = 100$

From Figure 9 we can observe that the EAs with Success-Based Offspring Population Size had a poor performance, we hypothesise that this is caused because on the branch points in SufSamp the algorithms have a high probability of having a small offspring population size causing them to take the wrong path, this makes the population size even smaller and the probability of correcting their path smaller. Later they reach a local optimum where they start to increase the population size but the probability to get out of the local optimum is almost null.

For the non self-adjusting population size algorithms, all their best performances where with the highest offspring population size $\lambda = 600$, which was expected. At last we can appreciate that the $(1 + \lambda)$ EA outperformed every other algorithm, even with $\lambda = 400$ (showed translucent in Figure 9). This shows the importance of having the necessary conditions on the branch points. If the mutation rate or the offspring population size is not the correct, it can lead to the wrong path and ultimately to a local optimum.

7 CONCLUSIONS

In this paper we compared well known EAs and modified versions with success-based parameter control using four artificial landscapes. The landscapes include two simple and well studied benchmark problems (OneMax, LeadingOnes), one NP-hard problem from combinatorial optimisation (Makespan Scheduling) and a function designed to trap small offspring population sizes (SufSamp).

We proposed a new metric to evaluate the performance of EAs that are used in parallel computing. The metric is the geometric mean of the evaluations and generations, which takes into account the proportion of improvement in the number of generations without neglecting the proportion of increased evaluations.

From the empirical study we found the following insights from each type of success-based parameter control mechanism.

The $(1 + \lambda)$ EA with Two-rate Standard Bit Mutation showed a good performance overall, it never was too far away from the best algorithms and outperformed all other EAs on Makespan Scheduling. It also tends to work better than other algorithms when using large offspring population sizes.

All the $(1 + \lambda)$ EAs with Success-Based Offspring Population Size had a small increase in sequential optimisation time with the added advantage of not needing to select the appropriate offspring population size. The GM of these algorithms was worse than several static choices on OneMax and LeadingOnes indicating that these algorithms might not be a good choice for parallel computing. Even though these algorithms had a good performance in sequential optimisation time and GM on Makespan Scheduling, there was a substantial difference in performance with different update factors. Also the performance of these algorithms on SufSamp show that they cannot update their parameters sufficiently well on problems with several local optima that trap small offspring population sizes. From the four EAs tested, the $(1 + \{G\lambda, \lambda/s\})$ EA performed better in general but it had the most variability with different selections of the update factor G.

The $(1 + (\lambda, \lambda))$ GA performed badly in 3 of the 4 problems tested, we need to note that it also has been shown to work well for several types of problems [3, 10]. On OneMax we showed that the selection mechanism based on the Optimal Stopping Theory helped improve the results of this algorithm in problem sizes of $n = \{100, 500, 5000\}$ compared to the original Self-Adjusting $(1 + (\lambda, \lambda))$ GA which is the fastest known GA on that problem.

Going forward, we intend to theoretically study the new proposed algorithm to understand better why it uses fewer evaluations and to try to improve its performance. We also intend to study further the success-based algorithms showed here to understand their benefits for complex problems.

ACKNOWLEDGMENTS

I would like to thank Dirk Sudholt for valuable discussions and many useful suggestions.

REFERENCES

[1] Golnaz Badkobeh, Per Kristian Lehre, and Dirk Sudholt. 2014. Unbiased black-box complexity of parallel search. In *Lecture Notes in Computer Science (PPSN 2014)*, Vol. 8672. Springer.

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

[3] 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, New York, NY, USA, 1343–1350. https://doi.org/10.1145/3071178.3071297

[4] Francisco Chicano, Andrew M. Sutton, L. Darrell Whitley, and Enrique Alba. 2015. Fitness Probability Distribution of Bit-flip Mutation. *Evol. Comput.* 23, 2 (June 2015), 217–248. https://doi.org/10.1162/EVCO_a_00130

[5] Dogan Corus, Jun He, Thomas Jansen, Pietro S. Oliveto, Dirk Sudholt, and Christine Zarges. 2017. On Easiest Functions for Mutation Operators in Bio-Inspired Optimisation. In *Algorithmica*, Vol. 78. Springer, 714–740.

[6] D. Corus and P. S. Oliveto. 2018. Standard Steady State Genetic Algorithms Can Hillclimb Faster Than Mutation-Only Evolutionary Algorithms. *IEEE Transactions on Evolutionary Computation* 22, 5 (Oct 2018), 720–732. https://doi.org/10.1109/TEVC.2017.2745715

[7] Dogan Corus, Pietro Simone Oliveto, and Donya Yazdani. 2018. Artificial Immune Systems Can Find Arbitrarily Good Approximations for the NP-Hard Partition Problem. *CoRR* abs/1806.00300 (2018). arXiv:1806.00300 http://arxiv.org/abs/1806.00300

[8] Benjamin Doerr and Carola Doerr. 2015. Optimal Parameter Choices Through Self-Adjustment: Applying the 1/5-th Rule in Discrete Settings. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. 1335–1342.

[9] Benjamin Doerr and Carola Doerr. 2018. Theory of Parameter Control for Discrete Black-Box Optimization: Provable Performance Gains Through Dynamic Parameter Choices. *CoRR* abs/1804.05650 (2018). arXiv:1804.05650 http://arxiv.org/abs/1804.05650

[10] 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. https://doi.org/10.1016/j.tcs.2014.11.028

[11] Benjamin Doerr, Christian Gießen, Carsten Witt, and Jing Yang. 2017. The (1+λ) Evolutionary Algorithm with Self-Adjusting Mutation Rate. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '17)*. ACM, 1351–1358. https://doi.org/10.1145/3071178.3071279

[12] Benjamin Doerr, Daniel Johannsen, and Carola Winzen. 2012. Multiplicative Drift Analysis. *Algorithmica* 64, 4 (01 Dec 2012), 673–697. https://doi.org/10.1007/s00453-012-9622-x

[13] Benjamin Doerr and Marvin Künnemann. 2015. Optimizing linear functions with the (1+λ) evolutionary algorithm - Different asymptotic runtimes for different instances. *Theoretical Computer Science* 561 (2015), 3 – 23. https://doi.org/10.1016/j.tcs.2014.03.015 Genetic and Evolutionary Computation.

[14] Carola Doerr, Furong Ye, Sander van Rijn, Hao Wang, and Thomas Bäck. 2018. Towards a Theory-guided Benchmarking Suite for Discrete Black-box Optimization Heuristics: Profiling (1 + λ) EA Variants on Onemax and Leadingones. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '18)*. ACM, New York, NY, USA, 951–958. https://doi.org/10.1145/3205455.3205621

[15] Thomas S. Ferguson. 1989. Who Solved the Secretary Problem? *Statist. Sci.* 4, 3 (08 1989), 282–289. https://doi.org/10.1214/ss/1177012493

[16] Christian Gießen and Carsten Witt. 2015. The Interplay of Population Size and Mutation Probability in the (1 + λ) EA on OneMax. *Algorithmica* 78, 2 (June 2015), 587–609. https://doi.org/10.1007/s00453-016-0214-z

[17] Christian Gießen and Carsten Witt. 2018. Optimal Mutation Rates for the (1+λ) EA on OneMax Through Asymptotically Tight Drift Analysis. *Algorithmica* 80, 5 (2018), 1710–1731. https://doi.org/10.1007/s00453-017-0360-y

[18] Thomas Jansen, Kenneth A. De Jong, and Ingo Wegener. 2005. On the Choice of the Offspring Population Size in Evolutionary Algorithms. *Evolutionary Computation* 13, 4 (2005), 413–440.

[19] Jörg Lässig and Dirk Sudholt. 2010. The benefit of migration in parallel evolutionary algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2010)*. 1105–1112.

[20] Jörg Lässig and Dirk Sudholt. 2010. General Scheme for Analyzing Running Times of Parallel Evolutionary Algorithms. In *Parallel Problem Solving from Nature (PPSN XI)*. Springer, 234–243.

[21] 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.

[22] Frank Neumann and Carsten Witt. 2010. *Bioinspired Computation in Combinatorial Optimization*. Springer.

[23] Stjepan Picek and Marin Golub. 2010. Comparison of a Crossover Operator in Binary-coded Genetic Algorithms. *WSEAS Transactions on Computers* 9, 9 (September 2010), 1064–1073.

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

[25] Dirk Sudholt. 2015. Parallel Evolutionary Algorithms. In *Handbook of Computational Intelligence*. Springer, 929–959.

[26] Dirk Sudholt. 2017. How Crossover Speeds Up Building-Block Assembly in Genetic Algorithms. In *Evolutionary Computation*, Vol. 25. 237–274. https://doi.org/10.1162/EVCO_a_00171

[27] Carsten Witt. 2013. Tight bounds on the optimization time of a randomized search heuristic on linear functions. *Combinatorics, Probability and Computing* 22, 2 (2013), 294–318.