Evolutionary Algorithm Parameters and Methods to Tune them

A.E. Eiben and S.K. Smit

Abstract I
n this chapter we discuss the notion of Evolutionary Algorithm (EA) parameters and propose a distinction between EAs and EA instances, based on the type of parameters used to specify their details. Furthermore, we consider the most important aspects of the parameter tuning problem and give an overview of existing parameter tuning methods. Finally, we elaborate on the methodological issues involved here and provide recommendations for further development.

1 Background and Objectives

Finding appropriate parameter values for evolutionary algorithms (EA) is one of the persisting grand challenges of the evolutionary computing (EC) field. In general, EC researchers and practitioners all acknowledge that good parameter values are essential for good EA performance. However, very little effort is spent on studying the effect of EA parameters on EA performance and on tuning them. In practice, parameter values are mostly selected by conventions (mutation rate should be low), ad hoc choices (why not use uniform crossover), and experimental comparisons on a limited scale (testing combinations of three different crossover rates and three different mutation rates). Hence, there is a striking gap between the widely acknowledged importance of good parameter values and the widely exhibited ignorance concerning principled approaches to tune EA parameters.

To this end, it is important to recall that the problem of setting EA parameters is commonly divided into two cases, parameter tuning and parameter control [14].

A.E. Eiben
Vrije Universiteit Amsterdam, e-mail: `gusz@cs.vu.nl`

S.K. Smit
Vrije Universiteit Amsterdam e-mail: `sksmit@cs.vu.nl`

In case of parameter control, the parameter values are changing during an EA run. In this case one needs initial parameter values and suitable control strategies, that in turn can be deterministic, adaptive, or self-adaptive. Parameter tuning is easier in the sense that the parameter values are not changing during a run, hence only a single value per parameter is required. Nevertheless, even the problem of tuning an EA for a given application is hard because of the large number of options and the limited knowledge about the effect of EA parameters on EA performance.

Given this background, we can express the primary focus of this chapter as being parameter tuning. Our main message is that the technical conditions to choose good parameter values are given and the technology is easily available. As it happens, there exist various algorithms that can be used for tuning EA parameters. Using such tuning algorithms (tuners, for short, in the sequel) implies significant performance improvements and the computational costs are moderate. Hence, changing the present practice and using tuning algorithms widely would lead to improvements on a massive scale: large performance gains for a large group of researchers and practitioners.

The overall aim of this chapter is to offer a thorough treatment of EA parameters and algorithms to tune them. This aim can be broken down in a number of technical objectives:

1. To discuss the very notion of EA parameters and its relationship with the concepts of EAs and EA instances.
2. To consider the most important aspects of the parameter tuning problem.
3. To give an overview of existing parameter tuning methods.
4. To elaborate on the methodological issues involved here and provide recommendations for further research.

2 Evolutionary Algorithms, Parameters, Algorithm Instances

Evolutionary algorithms form a class of heuristic search methods based on a particular algorithmic framework whose main components are the variation operators (mutation and recombination – a.k.a. crossover) and the selection operators (parent selection and survivor selection), cf. [17]. The general evolutionary algorithm framework is shown in Figure1.

A decision to use an evolutionary algorithm to solve some problem implies that the user, respectively algorithm designer, adopts the main design decisions that led to the general evolutionary algorithm framework and only needs to specify "a few" details. In the sequel we use the term *parameters* to denote these details.

Using this terminology, designing an EA for a given application amounts to selecting good values for the parameters. For instance, the definition of an EA might include setting the parameter `crossoveroperator` to `onepoint`, the parameter `crossoverrate` to 0.5, and the parameter `populationsize` to 100. In principle, this is a sound naming convention, but intuitively, there is a difference between choosing a good crossover operator from a given list of three operators

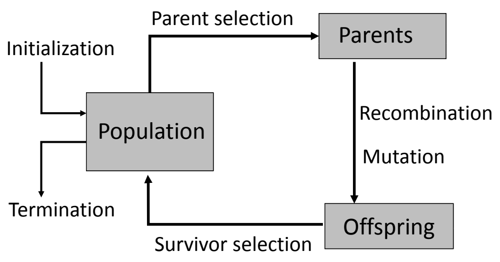

Fig. 1: General framework of an evolutionary algorithm.

and choosing a good value for the related crossover rate $p_c \in [0,1]$. This difference can be formalized if we distinguish parameters by their domains. The parameter `crossoveroperator` has a finite domain with no sensible distance metric or ordering, e.g., $\{$`onepoint`, `uniform`, `averaging`$\}$, whereas the domain of the parameter p_c is a subset of $\mathbb{R}$ with the natural structure for real numbers. This difference is essential for searchability. For parameters with a domain that has a distance metric, or is at least partially ordered, one can use heuristic search and optimization methods to find optimal values. For the first type of parameters this is not possible because the domain has no exploitable structure. The only option in this case is sampling.

The difference between two types of parameters has already been noted in evolutionary computing, but various authors use various naming conventions. For instance, [5] uses the names *qualitative*, respectively *quantitative* parameters, [36] distinguishes *symbolic* and *numeric* parameters, [10] calls them *categorical* and *numerical*, while [**?**] refers to *structural* and *behavioral* parameters. Furthermore, [29] calls unstructured parameters *components* and the elements of their domains operators and in the corresponding terminology a parameter is instantiated by a value, while a component is instantiated by allocating an operator to it. In the context of statistics and data mining one distinguishes two types of variables (rather then parameters) depending on the presence of an ordered structure, but a universal terminology is lacking here too. Commonly used names are *nominal* vs. *ordinal* and *categorical* vs. *ordered* variables. Table 1 provides a quick overview of these options in general; Table 2 shows an EA specific illustration with commonly used parameters in both categories.

From now on we will use the terms *qualitative parameter* and *quantitative parameter*. For both types of parameters the elements of the parameter's domain are called *parameter values* and we *instantiate* a parameter by allocating a value to it. In practice, quantitative parameters are mostly numerical values, e.g., the parameter crossover rate uses values from the interval $[0,1]$, and qualitative parameters are often symbolic, e.g., `crossoveroperator`. However, in general, quantitative parameters and numerical parameters are not the same, because it is possible to

Parameter with an unordered domain	**Parameter with an ordered domain**
qualitative	quantitative
symbolic	numeric
categorical	numerical
structural	behavioral
component	parameter
nominal	ordinal
categorical	ordered

Table 1: Pairs of terms used in the literature to distinguish two types of parameters (variables).

have an ordering on a set of symbolic values. (For instance, one could impose an alphabetical ordering on a set of colors $\{blue, green, red, yellow\}$.)

It is important to note that the number of parameters of EAs is not specified in general. Depending on particular design choices one might obtain different numbers of parameters. For instance, instantiating the qualitative parameter `parentselection` by `tournament` implies a new quantitative parameter `tournamentsize`. However, choosing for `roulettewheel` does not add any parameters. This example also shows that there can be a hierarchy among parameters. Namely, qualitative parameters may have quantitative parameters “under them”. If an unambiguous treatment requires we can call such parameters *sub-parameters*, always belonging to a qualitative parameter.

Distinguishing qualitative and quantitative parameters naturally leads to distinguishing two levels in designing a specific EA for a given problem. In the sequel, we perceive qualitative parameters as high-level ones that define and evolutionary algorithm, and look at quantitative parameters as low-level ones that define a variant of this EA. Table 2 illustrates this matter.

Following this naming convention an evolutionary algorithm is a partially specified algorithm where the values to instantiate qualitative parameters are defined, but the quantitative parameters are not. Hence, we consider two EAs to be different if they differ in one of their qualitative parameters, e.g., use different mutation operators. If the values for all parameters are specified then we obtain *an EA instance*. This terminology enables precise formulations, meanwhile it enforces care with phrasing. Clearly, this distinction between EAs and EA instances is similar to distinguishing problems and problem instances. For example, the abbreviation TSP represents the set of all possible problem configurations of the traveling salesman problem, whereas a TSP instance is one specific problem, e.g., 10 specific cities with a given distance matrix D. If rigorous terminology is required then the right phrasing is “to apply an EA instance to a problem instance”. However, such rigor is not always required, and formally inaccurate but understandable phrases like “to apply an EA to a problem” can be fully acceptable in the practical sense.

	EA_1	EA_2	EA_3
qualitative parameters			
Representation	bitstring	bitstring	real-valued
Recombination	1-point	1-point	averaging
Mutation	bit-flip	bit-flip	Gaussian $N(0,\sigma)$
Parent selection	tournament	tournament	uniform random
Survivor selection	generational	generational	(μ,λ)
quantitative parameters			
p_m	0.01	0.1	0.05
σ	n.a.	n.a	0.1
p_c	0.5	0.7	0.7
μ	100	100	10
λ	n.a.	n.a	70
tournament size	2	4	n.a

Table 2: Three EA instances specified by the qualitative parameters representation, recombination, mutation, parent selection, survivor selection, and the quantitative parameters mutation rate (p_m), mutation step size (σ), crossover rate (p_c), population size (μ), offspring size (λ), and tournament size. In our terminology, the EA instances in columns EA_1 and EA_2 are just variants of the same EA. The EA instance in column EA_3 belongs to a different EA.

3 Algorithm Design and Parameter Tuning

In the broad sense, algorithm design includes all decisions needed to specify an algorithm (instance) for solving a given problem (instance). Throughout this paper we perceive parameter tuning as a special case of algorithm design. The principal challenge for evolutionary algorithm designers is caused by the fact that the design details, i.e., parameter values, largely influence the performance of the algorithm. An EA with good parameter values can be orders of magnitude better than one with poorly chosen parameter values. Hence, algorithm design in general, and EA design in particular, is an optimization problem itself.

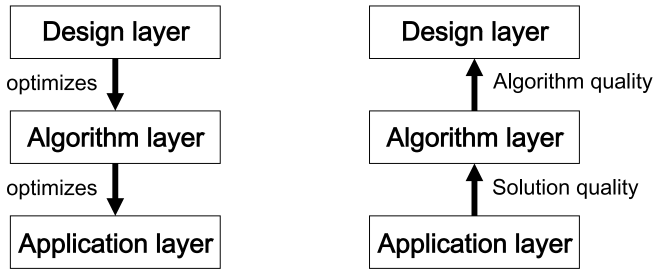

Fig. 2: The left diagram depicts the control flow through the three main layers in the hierarchy of parameter tuning, while the right diagram shows the information flow.

To obtain a detailed view on this issue we distinguish three layers: application layer, algorithm layer, and design layer, see Figure 2. As this figure indicates, the whole scheme can be divided into two optimization problems. The lower part of this three-tier hierarchy consists of a problem on the application layer (e.g., the traveling salesman problem) and an EA (e.g., a genetic algorithm) on the algorithm layer trying to find an optimal solution for this problem. Simply put, the EA is iteratively generating candidate solutions (e.g., permutations of city names) seeking one with maximal quality. The upper part of the hierarchy contains a tuning method that is trying to find optimal parameter values for the EA on the algorithm layer. Similarly to the lower part, the tuning method is iteratively generating parameter vectors seeking one with maximal quality, where the quality of a given parameter vector is based on the performance of the EA using the values of it. To avoid confusion we use distinct terms to designate the quality function of these optimization problems. Conform the usual EC terminology we use the term fitness for the quality of candidate solutions on the lower level, and the term *utility* to denote the quality of parameter vectors. Table 3 provides a quick overview of the related vocabulary.

	Problem solving	**Parameter tuning**
Method at work	evolutionary algorithm	tuning procedure
Search space	solution vectors	parameter vectors
Quality	fitness	utility
Assessment	evaluation	testing

Table 3: Vocabulary distinguishing the main entities in the context of problem solving (lower two blocks in Figure 2) and algorithm design, respectively parameter tuning (upper two blocks in Figure 2).

With this nomenclature we can formalize the problem to be solved by the algorithm designer if we denote the qualitative parameters and their domains by $q_1,\ldots,q_m$ and $Q_1,\ldots,Q_m$, likewise using the notation $r_1,\ldots,r_n$ and $R_1,\ldots,R_n$ for the quantitative parameters.[1] The problem of parameter tuning can then be seen as a search problem $\langle S,u\rangle$ in the parameter space

$$S = Q_1 \times \cdots \times Q_m \times R_1 \times \cdots \times R_n \tag{1}$$

using a utility function u, where the utility $u(\bar{p})$ of a given parameter vector $\bar{p} \in S$ reflects the performance of $EA(\bar{p})$, i.e., the evolutionary algorithm instance using the values of $\bar{p}$, on the given problem instance(s). Solutions of the parameter tuning problem can then be defined as parameter vectors with maximum utility. Given this conceptual framework it is easy to distinguish the so-called "structural" and "parametric" tuning [9] in a formal way: structural tuning takes place in

[1] Observe that by the possible presence of sub-parameters the number of quantitative parameters n depends on the instantiations of $q_1,\ldots q_m$. This makes the notation somewhat inaccurate, but we use it for sake of simplicity.

the space $S = Q_1 \times \cdots \times Q_m$, while parametric tuning refers to searching through $S = R_1 \times \cdots \times R_n$.

Now we can define the *utility landscape* as an abstract landscape where the locations are the parameter vectors of an EA and the height reflects utility. It is obvious that fitness landscapes –commonly used in EC– have a lot in common with utility landscapes as introduced here. To be specific, in both cases we have a search space (candidate solutions vs. parameter vectors), a quality measure (fitness vs. utility) that is conceptualized as 'height', and a method to assess the quality of a point in the search space (fitness evaluation vs. utility testing). Finally, we have a search method (an evolutionary algorithm vs. a tuning procedure) that is seeking for a point with maximum height.

Despite the obvious analogies between the upper and the lower halves of Figure 2), there are some differences we want to note here. First of all, fitness values are most often deterministic – depending, of course, on the problem instance to be solved. However, the utility values are always stochastic, because they reflect the performance of an EA which is a stochastic search method. The inherently stochastic nature of utility values implies particular algorithmic and methodological challenges that will be discussed later on. Second, the notion of fitness is usually strongly related to the objective function of the problem on the application layer and differences between suitable fitness functions mostly concern arithmetic details. The notion of utility, however, depends on the problem instance(s) to be solved and the performance metrics used to define how good an EA is. In the next secion we will have a closer look on these apsects.

4 Utility, Algorithm Performance, Test Functions

As stated above, solutions of the parameter tuning problem are parameter vectors with maximum utility, where utility is based on some definition of EA performance and some objective functions, respectively problem instances.

In general, there are two atomic performance measures for EAs: one regarding solution quality and one regarding algorithm speed or search effort. The latter is interpreted here in the broad sense, referring to any sensible measure of computational effort for evolutionary algorithms, e.g., the number of fitness evaluations, CPU time, wall-clock time, whatever.[2] There are different combinations of fitness and time that can be used to define algorithm performance in one single run. For instance:

- Given a maximum running time (computational effort), algorithm performance is defined as the best fitness at termination.
- Given a minimum fitness level, algorithm performance is defined as the running time (computational effort) needed to reach it.
- Given a maximum running time (computational effort) and a minimum fitness level, algorithm performance is defined through the Boolean notion of success:

[2] Please refer to [15] for more details on measuring EA search effort.

a run succeeds if the given fitness is reached within the given time, otherwise it fails.

Obviously, by the stochastic nature of EAs multiple runs on the same problem are necessary to get a good estimation of performance. Aggregating the measures mentioned above over a number of runs we obtain the performance metrics commonly used in evolutionary computing, cf. [17, Chapter 14]:

- MBF (mean best fitness),
- AES (average number of evaluations to solution),
- SR (success rate),

respectively. When designing a good EA, one may tune it on either of these performance measures, or a combination of them.

Further to the given performance metrics, utility is also determined by the problem instance(s), respectively objective functions for which the EA is being tuned. In the simplest case, we are tuning our EA on one function f. Then the utility of a parameter vector $\bar{p}$ is measured by the EA performance on f. In this case, tuning delivers a *specialist*, that is, an EA that is very good in solving one problem instance, the function f, with no claims or indications regarding its performance on other problem instances. This can be a satisfactory result if one is only interested in solving f. However, algorithm designers in general, and evolutionary computing experts in particular, are often interested in so called robust EA instances, that is, in EA instances that work well on many objective functions. In terms of parameters, this requires "robust parameter settings". To this end, test suites consisting of many test functions are used to test and evaluate algorithms. For instance, a specific set $\{f_1,\ldots,f_n\}$ is used to support claims that the given algorithm is good on a "wide range of problems". Tuning an EA on a set of functions delivers a *generalist*, that is, an EA that is good in solving various problem instances. Obviously, a true generalist would perform well on all possible functions. However, this is impossible by the no-free-lunch theorem [35]. Therefore, the quest for generalist EAs is practically limited to less general claims that still raise serious methodology issues as discussed in [15].

Technically, tuning an EA on collection of functions $\{f_1,\ldots,f_n\}$ means that the utility is not a single number, but a vector of utilities corresponding to each of the test functions. Hence, finding a good generalist is a multi-objective problem, for which each test function is one objective. The current parameter tuning algorithms can only deal with tuning problems for which the utilities can be compared directly, therefore a method is needed to aggregate the utility-vectors into one scalar number. A straightforward solution to this can be obtained by averaging over the given test suite. For a precise definition we need to extend the notation of the utility function such that it shows the given objective function f. Then $u_f(\bar{p})$ reflects the performance of the evolutionary algorithm instance $EA(\bar{p})$ on f, and the utility of $\bar{p}$ on $F=\{f_1,\ldots,f_z\}$ can be defined as the average of the utilities $u_{f_1}(\bar{p}),\ldots,u_{f_z}(\bar{p})$:

$$u_F(\bar{p}) = \frac{1}{z}\cdot\sum_{i=1}^{z} u_{f_i}(\bar{p}).$$

Obviously, instead of a simple arithmetic average, one could use weighted averages or more advanced multi-objective aggregation mechanisms, for instance, based on lexicographic orderings.

Summarizing, the core of our terminology is as follows:

1. Solution vectors have fitness values, based on the objective function related to the given problem instance to be solved.
2. EA instances have performance values, based on information regarding fitness and running time on one or more problem instances (objective functions).
3. Parameter vectors have utility values, defined by the performance of the corresponding EA instance and the problem instances (objective functions) used for tuning.

Figure 3 illustrates the matter in a graphical form. It shows that the solutions of a tuning problem depend on the problem(s) to be solved, the EA used, and the utility function. Adding the tuner to the equation we obtain this picture showing how a set of good parameter values obtained through tuning depends on four factors.

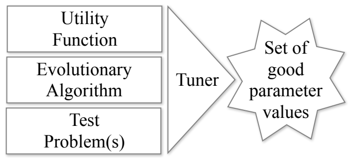

Fig. 3: Generic scheme of parameter tuning showing how good parameter values depend on four factors: the problem(s) to be solved, the EA used, the utility function, and the tuner itself.

5 Algorithmic Approaches to Parameter Tuning

In this section we review algorithmic approaches to solving the parameter tuning problem. In essence, all of these methods work by the generate and test principle, i.e., through generating parameter vectors and testing them to establish their utility.

5.1 Iterative and Non-iterative Tuners

The tuners we are to review in the following are variants of this generic scheme that can be divided into two main categories:

1. *iterative* and
2. *non-iterative* tuners.

All non-iterative tuners execute the GENERATE step only once, during initialization, thus creating a fixed set of vectors. Each of those vectors is then tested during the TEST phase to find the best vector in the given set. Hence, one could say that non-iterative tuners follow the INITIALIZE–and–TEST template. Initialization can be done by random sampling, generating a systematic grid, or some space filling set of vectors.

The second category of tuners is formed by iterative methods that do no not fix the set of vectors during initialization, but start with a small initial set and create new vectors iteratively during execution.

5.2 Single and Multi-stage Procedures

Given the stochastic nature of EAs, a number of tests is necessary for a reliable estimate of utility. Following [8] and [5], we distinguish

1. *single-stage* and
2. *multi-stage procedures*.

Single-stage procedures perform the same number of tests for each given vector, while multi-stage procedures use a more sophisticated strategy. In general, they augment the TEST step by adding a SELECT step, where only promising vectors are selected for further testing, deliberately ignoring those with a low performance.

As mentioned in Section 2, some methods are only applicable to quantitative parameters. Sophisticated tuners, such as SPOT [3], however, can be used for quantitative, qualitative or even mixed parameter spaces.

5.3 Measuring Search Effort

In the sequel, we present an overview of parameter tuning algorithms. The arranging principle we use is based on the search effort perspective. Obviously, good tuning algorithms try to find a good parameter vector with the least possible effort. In general, the total search effort can be expressed as $A \times B \times C$, where

- A is the number of parameter vectors to be tested by the tuner.
- B is the number of tests, e.g., EA runs, per parameter vector to establish its utility. The product $A \times B$ represents the total number of algorithm runs used by the tuners. Note, $B = 1$ for deterministic algorithms which optimize deterministic problems.
- C is the number of function evaluations performed in one run of the EA. This is related to the estimation of the performance, because we are trying to estimate the performance of an algorithm based on a few number of function evaluations only, and not after the complete run is finished.

Based on this perspective, we summarize existing methods divided into four categories: those that try to allocate search efforts optimally by saving on A, B, C, respectively. In addition, we consider tuners that try to allocate search efforts optimally by saving on A and B.

5.4 Classification of Tuning Methods

5.4.1 Using a Small Number of Parameter Vectors (A)

In this group we find tuners that are trying to allocate search efforts efficiently by cleverly generating parameter vectors. Strictly speaking they might not always minimize A, but try to "optimize the spending", that is, get the most out of testing A parameter vectors. Such tuners are usually iterative methods, although there are exceptions, e.g., Latin-Square [24] and Taguchi orthogonal arrays [34]. The idea behind most tuners in this category is to start with a relatively small set of vectors and iteratively generate new sets in a clever way, i.e., such that new vectors are likely to be good. These tuners are only appropriate for quantitative parameters.

Finding a good set of parameter values for EAs is the kind of search problem that EAs are good at. It is therefore possible to use an EA (on the design layer) for optimizing the parameters of an EA (on the algorithm layer). The EA on the design layer is called a *meta-EA* and in general any kind of evolutionary algorithm, e.g., *genetic algorithms* (GA), *evolution strategies* (ES), *evolutionary programming* (EP), *differential evolution* (DE), *particle swarm optimization* (PSO), *estimation of distribution algorithms* (EDA), etc., could be used for this purpose. However, existing literature only reports on using GA, ES, and EDA as meta-EA. We assume that readers of this paper are familiar with the basics of evolutionary algorithms and need no introduction to GAs or ES. Therefore, here we only discuss the generic template of meta-EA applications, which is the following. The individuals used in the meta-EA encode a parameter vector $\bar{p}$ of (quantitative) parameter values of the baseline EA. This encoding depends on the type of EA on the design layer, and may differ for a meta-GA, meta-ES, or meta-EDA. To evaluate the utility of such a vector $\bar{p}$, the baseline EA is ran B times using the given parameter values in $\bar{p}$, thereby performing B tests. The utility of $\bar{p}$ is determined based on these runs/tests, and the actual definition of EA performance. Using this utility information, the meta-EA can perform selection, recombination, and mutation of parameter vectors in the usual way. As early as 1978, Mercer and Sampson [23] elaborated on this idea by their meta-GA (called meta-plan), but due to the large computational costs, their research was very limited. Greffenstette [18] was the first to conduct more extensive experiments with a meta-EAs and showed the effectiveness of the approach. The use of meta-ES has been reported by Greffenstette[18] and Herdy[20] with mixed results.

The meta-GA and the meta-ES approach do not differ significantly, the GA or ES is used as an optimizer and any preference between them is solely based on their optimization power and/or speed. The approach based on ideas from EDAs is different in this respect. While EAs only try to find good points in a search space belonging to a certain problem, EDAs try to estimate the distribution of promising values over the search space. Consequently, they provide more information than a GA or an ES, because the evolved distributions disclose a lot about the parameters, in particular about the utility landscape and can be used to get insight into the sensitivity and relevance of the different parameters. Hence, the choice for using a meta-EDA can be motivated by the additional information it provides, w.r.t, a meta-GA or meta-ES. The *Relevance Estimation and VAlue Calibration* method (REVAC) offered by Nannen and Eiben is based on this idea. REVAC has been introduced in [28] and further polished in [26], [27] and [25]. In [29] and [33] it is demonstrated how REVAC can be used to indicate the costs and benefits of tuning each parameter—something a meta-GA or a meta-ES cannot do. In the process of searching for good parameter values REVAC implicitly creates probability distributions regarding the parameters (one probability distribution per parameter) in such a way that parameter values that proved to be good in former trials have a higher probability then poor ones. Initially, all distributions represent a uniform random variable and after each new test they are updated based on the new information. After terminating the tuning process, i.e., stopping REVAC, these distributions can be retrieved and analyzed, showing not only the range of promising parameter values, but also disclosing information about the relevance of each parameter. For a discussion of REVAC's probability distributions, entropy, and parameter relevance we refer to [33].

5.4.2 Using a Small Number of Tests (B)

The methods in this group are trying to reduce B, i.e., the number of tests per parameter vector. The essential dilemma here is that fewer tests yield less reliable estimates of utility and if the number of tests is too low, then the utilities of two parameter vectors might not be statistically distinguishable. More tests can improve (sharpen) the stability of the expected utility to a level such that the superiority of one vector over the other can be safely concluded, but more tests come with a price in the form of longer runtimes. The methods in this group use the same trick to deal with this problem: performing only a few tests per parameter vector initially and increasing this number to the minimum level that is enough to obtain statistically sound comparisons between the given parameter vectors. Such methods known as *statistical screening, ranking, and selection*. Some of these methods are designed to select a single best solution, others are designed to screen a set of solutions by choosing a (random size) subset of the solutions containing the best one. Several of these approaches are based on multi-stage procedures such as sequential sampling [8, 11].

Maron et al. [22] introduced the term *racing* for a basic ranking and selection scheme and applied several methods, based on this scheme, to EAs. These non-iterative methods assume that a set $P = \{\bar{p}_1, \ldots, \bar{p}_A\}$ of parameter vectors is given and a maximum number of tests, B, per parameter vector is specified. Then they work by the following procedure.

1. Test all $\bar{p} \in P$ once
2. Determine $\bar{p}_{best}$ with highest (average) utility
3. Determine P' as the set of parameters vectors whose utility is not significantly worse than that of $\bar{p}_{best}$
4. $P = P'$
5. If $size(P) > 1$ and the number of tests for each $\bar{p} \in P$ is smaller than B, goto 1

The main difference between the existing racing methods can be found in step 3 as there are several statistical tests[3] that can indicate significant differences in utility:

- Analysis of Variance (ANOVA), cf. [31]
- Kruskal-Wallis (Rank Based Analysis of Variance)
- Hoeffding's bound, cf. [22]
- Unpaired Student T-Test, cf. [22]

All these methods are multi-stage procedures and they can be used for both quantitative and qualitative parameters, see also the discussion in [9]. Recently, Balaprakash et al [1] introduced iterative racing procedures. Birattari et al [10] present an overview of state-of-the-art racing procedures.

5.4.3 Using a Small Number of Parameter Vectors and Tests (*A* and *B*)

There are currently four approaches that try to combine the main ideas from both categories. Yuan et al. [36] showed how *racing* [22] can be combined with meta-EAs in order to reduce both A and B. They propose two different approaches. In both methods, they use the standard meta-EA approach, but introduce a racing technique into the evaluation of parameter vectors.

In the first method, the quantitative (numerical) parameters are represented by a vector and are evolved using a meta-(1+λ) ES. At each generation, a set of λ new vectors is created using a Gaussian distribution centered at the current best vector. Racing is then used to determine which vector has the highest utility. By using racing, not all of the λ individuals have to be evaluated B times to determine the current best vector. It is expected that many of the vectors are eliminated after just a few test saving a large portion of computational resources.

Their second method uses a population of parameter vectors containing the quantitative (numerical) parameters, which is evolved using selection, recombination, and mutation. However, the utility of a vector of numerical parameters is not only evaluated within one single EA instance, but on all possible combinations of qualitative parameters. The utility of the vector is equal to the performance of the best combination. Racing is then used to determine which combination is most successfully using the vector of numerical parameters. By employing racing, not every combination has to be evaluated B times. This saves computational resources, while still

[3] Not to be confused with tests of parameter vectors, cf. Table 3.

being able to explore the search space of both quantitative and qualitative parameters.

Introducing the *sequential parameter optimization toolbox* (SPOT), Bartz-Beielstein et al. [3] outlined another approach which uses a small number of parameter vectors A and tests B by means of an iterative procedure. SPOT starts with an initial set of parameter vectors usually laid out as a space filling design, e.g., *Latin hybercube design* (LHD). These are tested B_0 times to estimate their expected utility. Based on the results, a prediction model is fitted to represent an approximation of the utility landscape (additionally, the often used kriging component enables estimating the error at any point of the model). Then l new parameter vectors are generated and their expected utility is estimated using the model, i.e., without additional runs of the EA. This step employs predictions and error estimations within the expected improvement heuristic to find points with good potential to improve over the currently existing ones. The most promising points are then tested B times. If none of those points results in a better expected utility, B is increased. To establish fairness, the number of re-evaluations of the current best and new design points is the same. As the estimates of the current vectors are sharpened, utility prediction gets more accurate over time and "lucky samples" from the start are dropped. Note that the set of parameter vectors employed after the initial step is usually very small, often in the range of 1 to 4 vectors, and is held constant in size.

In order to explore the search space relatively quickly, B is initialized at a small value. If a promising area is found, the method focuses on improving the model and the best vector using more accurate utilities estimates by increasing B. Although in [3] a kriging enhanced regression model is used to predict utilities, it is in principle a general framework suited for a large range of modeling techniques, see, e.g. [4].

Note that SPOT may also be combined with racing-like statistical techniques for keeping B low where possible. In [7], random permutation tests have been employed to decide if a newly tested configuration shall be repeated as often as the current best one or if it can safely be regarded as inferior. Lasarczyk [21] implemented Chen's *optimal computing budget allocation* (OCBA) [12] into SPOT.

The fourth approach that explicitly aims at using a small number of parameter vectors and tests is REVAC++, the most recent variant of REVAC. The name REVAC++ is motivated by the fact that the method can be simply described as REVAC + racing + sharpening, where racing and sharpening are separate methods that serve as possible add-ons to REVAC, or in fact any search-based tuner. This version of REVAC has been introduced in [32], where the advantageous effects of the racing and sharpening extensions have been also shown.

5.4.4 Using a Small Number of Function Evaluations (C)

Optimizing A and/or B are the most commonly used methods to optimally allocate search efforts, but in principle, the number of fitness evaluations per EA run (C) could also be optimized. Each decrease in EA run-time is multiplied by both A and B, hence such runtime reduction methods can have a high impact on the total search effort. However, as of summer 2009 we are not aware of any methods proposed for this purpose.

In theory, methods to reduce the number of evaluations per EA run could be based on the idea to terminate an EA run if there is an indication that the parameter-vector used by the given EA will not reach some utility threshold, e.g., the average utility, or the utility of another, known parameter-vector like the current best or current worst. Such an indication can be statistically or theoretically grounded, but also based on heuristics. This concept is very similar to racing, but instead of decreasing the number of tests (B), it is to decrease the number of evaluations (C). In general, such a method could deal with both qualitative and quantitative parameters.

As mentioned above, we could not find any general parameter tuning methods of this type in the literature. However, we found one paper whose main concept would fit under the umbrella of saving on C, whenever possible. This paper of Harik *et al.* is concerned with population sizes only as all other EA parameters are fixed and only population sizes are varied and compared [19]. The essence of the method is to run multiple EAs in parallel, each with a different population size, compare the running EAs after a fixed number of fitness evaluations, and terminate those of them that use a population size with an expectedly poor performance (i.e., utility). The expectations regarding the utility of different population sizes are based on a simple heuristic observing that small populations always have a head-start, because of the fast convergence to a (local) optimum. This fact motivates the assumption that once a large population EA overtakes a small population EA (using the same number of fitness evaluations), the small population EA will always remain behind. Hence, when an EA with a small population size starts performing worse than EAs with a large population size it can be terminated in an early stage, thus reducing search effort. The method in [19] can only use a fitness-based notion of utility, i.e., no speed of success rate information, and the heuristic is only applicable to population sizes. Further research is needed to develop heuristics for parameters regarding variation and selection operators.

6 Successful Case-studies on Tuning Evolutionary Algorithms

In recent years, the number of well documented case studies on parameter tuning is increasing. Depending on the type of tuning method, the focus varies from increasing algorithm performance to decreasing effort. Examples of the later one are for instance the experiments of Clune et al. who are tuning a traditional GA on the counting ones and a 4-bit deceptive trap problem [13]. Using the DAGA2 algorithm, they found parameter values that performed only slightly worse than the parameters found using extensive experimentation by experts. They concluded that ‘In situations in which the human investment required to set up runs is more precious than performance, the meta-GA could be preferred.

The experiments of Yuan and Gallagher in [36] are similar, but with a slightly different focus. They compared the tuning effort of their Racing-GA with the effort needed to test all possible combinations op parameter values, rather than manual labor. They tuned a range of algorithms to the 100-bit One-Max problem. They reported a 90% decrease in effort related to the brute-force method with a similar precision.

Nannen et al. illustrate the relation between tuning costs and performance in [29]. In contrast to [13], tuning effort is not compared with human effort, but measured as the computational effort to reach a certain performance. The study reports on tuning a large collection of EAs on the Rastrigin problem. The results show that the amount of tuning needed to reach a given performance differs greatly per EA. Additionally, the best EA depends on the amount of tuning effort that is allowed. Another interesting outcome indicates that the tuning costs mainly depend on the overall setup of the Evolutionary Algorithm, rather than the number of free parameters.

There are two studies we know of reporting on a significant performance improvement compared to the 'common wisdom' parameter values using an automated tuning algorithm. Bartz-Beielstein and Markon [6] showed how an Evolution Strategy and Simulated Annealing algorithm can be tuned using DACE and regression analysis on a real world problem concerning elevator group control. They reported that Standard parameterizations of search algorithms might be improved significantly. Smit and Eiben [33] compare 3 different tuning algorithms with -and without extra add-ons to enhance performance. The problem at the application layer was a the Rastrigin function, which was solved using a simple GA. They concluded that tuning parameters pays off in terms of EA performance. All 9 tuner instances managed to find a parameter vector that greatly outperformed the best guess of a human user. As the authors summarize: "no matter what tuner algorithm you use, you will likely get a much better EA than relying on your intuition and the usual parameter setting conventions".

Tuning evolutionary algorithms can be considered as a special case of tuning metaheuristics. Birattari's book, cf. [9], gives a good treatment from a machine learning perspective including several case studies, for instance, on tuning iterated local search and ant colony optimization by the F-race tuner.

7 Considerations for Tuning EAs

In this section we discuss some general issues forthcoming from EA tuning as we advocate here and provide some guidelines for tuning EAs.

To begin with, let us address concerns questioning the benefits of the whole tuning approach for real-world applications due to the computational overhead costs. For a detailed answer it is helpful to distinguish two types of applications, on-off problems and repetitive problems, cf. [17, Chapter 14]. In case of one-off problems, one has to solve a given problem instance only once and typically the solution quality is much more important than the total computational effort. Optimizing the road network around a new airport is an example of such problems. In such cases it is a sound strategy to spend the available time on running the EA several times

–with possibly long run times and different settings– and keep the best solution ever found without willing to infer anything about good parameters. After all, this problem instance will not be faced again, so information about the best EA parameters for solving it is not really relevant. It is these cases, i.e., one-off problems, where the added value of tuning is questionable. Applying some parameter control technique to adjust the parameter values on-the-fly, while solving the problem, is a better approach here. In case of repetitive problems, one has to solve many different instances of a given problem, where it is assumed that the different instances are not too different. Think, for example, on routing vehicles of a parcel delivery service in a city. In this case it is beneficial to tune the EA to the given problem, because the computational overhead costs of tuning only occur once, while the advantages of using the tuned EA are enjoyed repeatedly. Given these considerations, it it interesting to note that academic research into heuristic algorithms is more akin to solving repetitive problems, than to one-off applications. This is especially true for research communities that use benchmark test suites and/or problem instance generators.

Given the above considerations, we can argue that algorithmic tuning of EA parameters is beneficial for at least two types of users, practitioners facing real-world applications with a repetitive character and academics willing to advance the development of evolutionary algorithms in general. To these groups our first and foremost recommendation is:

> Do tune your EA with a tuner algorithm (and report the tuning efforts alongside the performance results).

This suggestion may sound trivial, but considering the current practice in evolutionary computing it it far from being superfluous. As we argue in this chapter, there are enough good tuners available, the efforts of tuning are limited, and the gains can be substantial. A typical use-case is the publication of a research paper showing that a newly invented EA (NI-EA) is better than some carefully chosen benchmark EA (BM-EA). In this case, tuning NI-EA can make the difference between comparable-and-sometimes-better and convincingly-better in favor of NI-EA.

Let us note that using parameter tuners in performance comparisons leads to the methodological question whether the benchmark algorithm should be tuned too. One argument for not tuning it is to interpret a benchmark as being a fixed and invariable algorithm to provide the same challenge for every comparison using it. This may sound unfair, giving an indecent advantage to the new algorithm in the comparison. However, the current EC research and publication practice ignores the tuning issue completely, hiding the tuning efforts, thus hiding the possible unfairness of a comparison between a tuned NI-EA and an untuned BM-EA. In this respect, using a tuner for NI-EA and reporting the tuning efforts alongside the performance results is more fair than the huge majority of publications at present. A typical claim in a paper following the usual practice sounds like

> "NI-EA is better than BM-EA."

with possible refinements concerning the number or type of test functions where this holds. Using tuners and reporting the tuning effort would improve the present practice by making things public and transparent. A typical claim of the new style would sound like

> "Spending effort X on tuning NI-EA, we obtain an instance that is better than (the untuned) BM-EA."

To this end, the tuning effort could be measured by the number of parameter vectors tested (A), the number of utility tests executed (A $\times$ B)[4], the computing time spent on running the tuner, etc.

Alternatively, one could reinterpret the concept of a benchmark EA from as-is (including the values of its parameters) to as-it-could-be (if its parameters were tuned). This stance would imply that both NI-EA and BM-EA need to be tuned for a fair comparison. A typical claim of this style would sound like

> "The best instance of NI-EA is better than the best instance of BM-EA, where for both EAs the best instance is the one obtained through spending effort X on tuning."

Note, that this new philosophy eliminates the rather accidental parameter values as a factor when comparing EAs and focuses on the type of EA operators (recombination, mutation, etc.) instead. In other words, we would obtain statements about EAs, rather than EA instances.

An important caveat with respect to tuning EAs is the fact that the set of important parameters is often less clear than it may seem. At the first sight, it is easy to identify the parameters of a given EA has, but a closer look *at its code* may disclose that the working of the algorithm depends on several constants hidden in the code. These "magic constants" are determined by the algorithm designer/programmer, but are not made visible as independent parameters. For instance, the conceptual description of the well-known G-CMA-ES will indicate the population size μ and the offspring size λ as parameters with corresponding default values, such as $\lambda = 4 + 3 \cdot log_2(n)$ and $\mu = \frac{\lambda}{2}$, including the magic constants 4, 3, and 2. These constants can be designated as independent parameters of the given EA, thereby making them subject to tuning. The advantage of this practice is clear, it provides additional knobs to tune, potentially increasing the maximum achievable performance. If your game is to obtain a very well performing EA for some application, then you should definitely consider this option. For academic comparisons, it may raise questions regarding the fairness of comparison, similar to those discussed above regarding benchmarking. The main question here is whether it is correct to compare the given EA using some good values[5] for its original parameters with a new variant of it using tuned values for those parameters and tuned values for some of its magic constants.

Finally, let us make a note that the adaptability of EAs or related algorithms to a specific problem is different. We have seen this in the documented results from [29] as explained in section 6: The default parameters of algorithm A already lead to very good performance, whether the ones of algorithm B do not. Tuning is thus

[4] If the number of tests per parameter vector is not constant then A $\times$ B is not the total number of utility tests executed, but for the present argument this is not relevant.

[5] Where the definition of 'good" is irrelevant here. It could be hand-tuned, algorithmically tuned, commonly used, whatever.

much more important for B. An attempt to measure the adaptablity by means of the *empirical tuning potential* (ETP) is given in [30]. However, a lower adaptability may stem from two sources. Either the algorithm is harder to tune and requires more effort while tuning, or the obtainable best parameter vectors are not much better than the default values. Running tuners with increased number of tests T may pay off to gain insight here.

8 Conclusions and Outlook

In this chapter we discussed the notions of EA parameters, elaborated on the issue of tuning EA parameters and reviewed several algorithmic approaches to solve it. Our main message can be summarized as follows: As opposed to what contemporary practice would suggest, *there are* good tuning methods that enable EA users to boost EA performance at moderate costs through finding good parameter values tailored to the given problem(s).

Regarding the future, we are moderately optimistic. We foresee a change in attitude on a large scale, fueled by the easily accessible and easy-to-use tuners. The most prominent change we expect is the wide-spread use of tuners in scientific publications as well as in applications. To be specific, we expect that evolutionary computing researchers and practitioners will spend an extra day on tuning their EA before creating the figures for an experimental comparison or deploying their EA-based optimizer module within a decision support system. Given that the costs of this extra effort are limited, but the gains can be substantial, it is hard to imagine a reason for not doing it. To support this development, let us point to freely available software from `http://sourceforge.net/projects/tuning/`. This web site contains the Matlab codes of the tuning algorithms we have been using over the last couple of years. Additionally, `http://sourceforge.net/projects/mobat/` provides the Meta-Heuristic Optimizer Benchmark and Analysis Toolbox. It can be used to configure, benchmark and analyze Evolutionary Algorithms and other Meta-Heuristic Optimizers. MOBAT is written in Java and optimized for parallel processing. Alternatively, SPOT can be obtained from `http://www.gm.fh-koeln.de/campus/personen/lehrende/thomas.bartz-beielstein/00489/`.

A change of practice as foreseen here could also imply a change in methodology on the long term, illustrated in Figure 4. In today's research practice comparisons of EAs are typically based on ad hoc parameter values. Hence, the experiments show a comparison between two rather arbitrary *EA instances* and tell nothing about the full potential of the given EAs. Using tuner algorithms could fix this deficiency and eliminate much of the noise introduced by ad hoc parameter values, thus supporting comparisons of evolutionary algorithms (i.e., configurations of the qualitative parameters), rather than evolutionary algorithm instances.

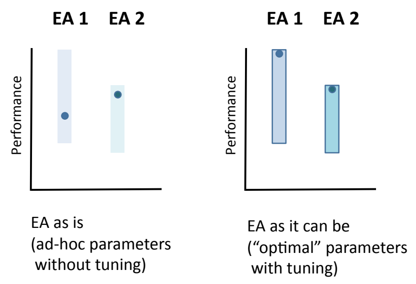

Fig. 4: The effect of parameter tuning on comparing EAs. Left: the traditional situation, where the reported EA performance is an "accidental" point on the scale ranging between worst and best performance (as determined by the used parameter values). Right: the improved situation, where the reported EA performance is a near-optimal point on this scale, belonging to the tuned instance. This indicates the full potential of the given EA, i.e., how good it can be with using the right parameter values.

References

1. Prasanna Balaprakash, Mauro Birattari, and Thomas Stützle. Improvement strategies for the f-race algorithm: Sampling design and iterative refinement. In *Hybrid Metaheuristics*, pages 108–122, 2007.
2. T. Bartz-Beielstein, M. Chiarandini, Luis L. Paquete, and M. Preuss, editors. *Empirical Methods for the Analysis of Optimization Algorithms*. Springer, 2009. In print.
3. T. Bartz-Beielstein, K.E. Parsopoulos, and M.N. Vrahatis. Analysis of Particle Swarm Optimization Using Computational Statistics. In Chalkis, editor, *Proceedings of the International Conference of Numerical Analysis and Applied Mathematics (ICNAAM 2004)*, pages 34–37, 2004.
4. Thomas Bartz-Beielstein. Experimental Analysis of Evolution Strategies: Overview and Comprehensive Introduction. Technical Report Reihe CI 157/03, SFB 531, Universität Dortmund, Dortmund, Germany, 2003.
5. Thomas Bartz-Beielstein. *Experimental Research in Evolutionary Computation—The New Experimentalism*. Natural Computing Series. Springer, Berlin, Heidelberg, New York, 2006.
6. Thomas Bartz-Beielstein and Sandor Markon. Tuning search algorithms for real-world applications: A regression tree based approach. Technical Report of the Collaborative Research Centre 531 Computational Intelligence CI-172/04, University of Dortmund, March 2004.
7. Thomas Bartz-Beielstein and Mike Preuss. Considerations of budget allocation for sequential parameter optimization (SPO). In L. Paquete et al., editors, *Workshop on Empirical Methods for the Analysis of Algorithms, Proceedings*, pages 35–40, Reykjavik, Iceland, 2006.
8. R. E. Bechhofer, C. W. Dunnett, D. M. Goldsman, and M. Hartmann. A comparison of the performances of procedures for selecting the normal population having the largest mean when populations have a common unknown variance. *Communications in Statistics*, B19:971–1006, 1990.
9. Mauro Birattari. *Tuning Metaheuristics*. Springer, Berlin, Heidelberg, New York, 2005.
10. Mauro Birattari, Zhi Yuan, Prasanna Balaprakash, and Thomas Stützle. F-frace and iterated F-frace: An overview. In Bartz-Beielstein et al. [2], pages X–Y. To appear.

11. Jürgen Branke, E. Chick, Stephen, and Christian Schmidt. New developments in ranking and selection: an empirical comparison of the three main approaches. In *WSC '05: Proceedings of the 37th conference on Winter simulation*, pages 708–717. Winter Simulation Conference, 2005.
12. J.E. Chen, C.H. Chen, and D.W. Kelton. Optimal computing budget allocation of indifference-zone-selection procedures. Working paper, taken from `http://www.cba.uc.edu/faculty/keltonwd`. Cited 6 January 2005, 2003.
13. J. Clune, S. Goings, B. Punch, and E. Goodman. Investigations in meta-gas: panaceas or pipe dreams? In *GECCO '05: Proceedings of the 2005 workshops on Genetic and evolutionary computation*, pages 235–241, New York, NY, USA, 2005. ACM.
14. A.E. Eiben, R. Hinterding, and Z. Michalewicz. Parameter Control in Evolutionary Algorithms. *IEEE Transactions on Evolutionary Computation*, 3(2):124–141, 1999.
15. A.E. Eiben and M. Jelasity. A Critical Note on Experimental Research Methodology in EC. In *Proceedings of the 2002 Congress on Evolutionary Computation (CEC'2002)*, pages 582–587. IEEE Press, Piscataway, NJ, 2002.
16. A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computation*. Natural Computing Series. Springer, 2003.
17. A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computing*. Springer, Berlin, Heidelberg, New York, 2003.
18. J.J Greffenstette. Optimisation of Control Parameters for Genetic Algorithms. In *IEEE Transactions on Systems, Man and Cybernetics*, volume 16, pages 122–128, 1986.
19. G. R. Harik and F. G. Lobo. A parameter-less genetic algorithm. In *In W. Banzhaf et al., editors, Proceedings of the Genetic and Evolutionary Computation Conference GECCO-99, San Francisco, CA. Morgan Kaufmann*, pages 258–265, 1999.
20. Michael Herdy. Reproductive isolation as strategy parameter in hierarichally organized evolution strategies. In R Männer and B Manderick, editors, *Proceedings of the 2nd Conference on Parallel Problem Solving from Nature*, pages 209–. North-Holland, Amsterdam, 1992.
21. Christian W. G. Lasarczyk. *Genetische Programmierung einer algorithmischen Chemie*. PhD thesis, Technische Universitt Dortmund, 2007.
22. O. Maron and A. Moore. The racing algorithm: Model selection for lazy learners. In *Artificial Intelligence Review*, volume 11, pages 193–225, April 1997.
23. R.E. Mercer and J.R. Sampson. Adaptive search using a reproductive metaplan. *Kybernetes*, 7:215–228, 1978.
24. Richard Myers and Edwin R. Hancock. Empirical modelling of genetic algorithms. *Evolutionary Computation*, 9(4):461–493, 2001.
25. V. Nannen. *Evolutionary Agent-Based Policy Analysis in Dynamic Environments*. PhD thesis, Free University Amsterdam, 2009.
26. V. Nannen and A. E. Eiben. Efficient Relevance Estimation and Value Calibration of Evolutionary Algorithm Parameters. In *IEEE Congress on Evolutionary Computation*, pages 103–110. IEEE, 2007.
27. V. Nannen and A. E. Eiben. Relevance Estimation and Value Calibration of Evolutionary Algorithm Parameters. In Manuela M. Veloso, editor, *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 1034–1039, 2007.
28. V. Nannen and A.E. Eiben. A Method for Parameter Calibration and Relevance Estimation in Evolutionary Algorithms. In M. Keijzer, editor, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2006)*, pages 183–190. Morgan Kaufmann, San Francisco, 2006.
29. V. Nannen, S.K. Smit, and A.E. Eiben. Costs and benefits of tuning parameters of evolutionary algorithms. In Günter Rudolph, Thomas Jansen, Simon M. Lucas, Carlo Poloni, and Nicola Beume, editors, *PPSN*, volume 5199 of *Lecture Notes in Computer Science*, pages 528–538. Springer, 2008.
30. Mike Preuss. Adaptability of algorithms for real-valued optimization. In Mario Giacobini et al., editors, *Applications of Evolutionary Computing, EvoWorkshops 2009. Proceedings*, volume 5484 of *Lecture Notes in Computer Science*, pages 665–674, Berlin, 2009. Springer.

31. J.D. Schaffer, R.A. Caruana, L.J. Eshelman, and R. Das. A study of control parameters affecting online performance of genetic algorithms for function optimization. In *Proceedings of the third international conference on Genetic algorithms*, pages 51–60, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.
32. S.K. Smit and A.E. Eiben. Comparing Parameter Tuning Methods for Evolutionary Algorithms. In *Proceedings of the 2009 IEEE Congress on Evolutionary Computation*, pages 399–406. IEEE Computational Intelligence Society, IEEE Press, 2009.
33. S.K. Smit and A.E. Eiben. Using Entropy for Parameter Analysis of Evolutionary Algorithms. In Bartz-Beielstein et al. [2], pages X–Y. To appear.
34. G. Taguchi and T. Yokoyama. *Taguchi Methods: Design of Experiments*. ASI Press, 1993.
35. David H. Wolpert and William G. Macready. No free lunch theorems for optimization. *IEEE Transaction on Evolutionary Computation*, 1(1):67–82, 1997.
36. B. Yuan and M. Gallagher. Combining Meta-EAs and Racing for Difficult EA Parameter Tuning Tasks. In F.G. Lobo, C.F. Lima, and Z. Michalewicz, editors, *Parameter Setting in Evolutionary Algorithms*, pages 121–142. Springer, 2007.