Using Entropy for Parameter Analysis of Evolutionary Algorithms

S.K. Smit   A.E. Eiben

Abstract— **Evolutionary Algorithms (EA) form a rich class of stochastic search methods that share the basic principles of incrementally improving the quality of a set of candidate solutions by means of variation and selection [10], [8]. Such variation and selection operators often require parameters to be specified. Finding a good set of parameter values is a non-trivial problem in itself, furthermore some EA parameters are more relevant than others in the sense that choosing different values for them affects EA performance more than for the other parameters. In this chapter we explain the notion of entropy and discuss how entropy can disclose important information on EA parameters, in particular, about their relevance. We describe an algorithm that is able to estimate the entropy of EA parameters and we present a showcase, based on extensive experimentation, to demonstrate the usefulness of this approach and some interesting insights that are gained.**

I. Introduction and Background

Evolutionary algorithms form a rich class of stochastic search methods that share the basic principles of incrementally improving the quality of a set of candidate solutions by means of variation and selection [10], [8]. Algorithms in this class are all based on the same generic framework (explained in the next section) and for obtaining a concrete algorithm one needs to fill in many details, that is to say, specify the parameters of the algorithm. Over the history of EAs it has became clear that good parameter values are essential for good performance. However, as of today, not much is known about the effect of parameters on performance. Setting parameter values is commonly done in a very ad hoc manner, based on conventions, intuition, and experimental comparisons on a limited scale. Collective wisdom in evolutionary computing (EC) acknowledges that some parameters have more impact on performance than others. Obviously, more influential parameters need more care when setting their values, but at the moment there are no widely used techniques to establish the (relative) importance of different parameters. Using screening methods [28] is one of the few techniques currently used to indicate importance, however the information that can be extracted is limited and the results of different algorithms cannot be compared.

In this chapter we show how entropy can be used to indicate how influential a particular parameter is. We use the term *parameter relevance* to reflect the level of influence on EA performance and argue that entropy is a good measure of relevance. The main contributions of this chapter are as follows:

1) We explain the notion of entropy and discuss how entropy can disclose important information on EA parameters, in particular, about their relevance.
2) Describe an algorithm, REVAC, that is able to estimate entropy of EA parameters.
3) Present a showcase, based on extensive experimentation, to demonstrate the usefulness of this approach and some interesting insights gained.

Vrije Universiteit Amsterdam, The Netherlands, {sksmit, gusz}@cs.vu.nl

The rest of this chapter is organized as follows. In Section II we briefly introduce evolutionary algorithms, followed by a discussion on their parameters and issues in parameter tuning in Section III. We elaborate on the notion of entropy in Section IV, including a discussion on the use of entropy for parameter analysis of EAs. The REVAC method is described in Section V. Section VI contains the case study and we conclude the paper in Section VII by summarizing the main issues.

II. Evolutionary Algorithms

Evolutionary Algorithms are all based on the same generic framework, inspired by biological evolution. The fundamental metaphor of evolutionary computing relates natural evolution to problem solving in a trial-and-error (a.k.a. generate-and-test) fashion as illustrated in Table I.

TABLE I
THE BASIC EVOLUTIONARY COMPUTING METAPHOR LINKING NATURAL EVOLUTION TO PROBLEM SOLVING

EVOLUTION		PROBLEM SOLVING
environment	$\longleftrightarrow$	problem
individual	$\longleftrightarrow$	candidate solution
fitness	$\longleftrightarrow$	quality

In natural evolution, a given environment is filled with a population of individuals that strive for survival and reproduction. Their fitness – determined by the environment – tells how well they succeed in achieving these goals, i.e., it reflects their chances to live and multiply. In the context of problem solving we have a collection of candidate solutions. Their quality – defined by the given problem – determines the chance that they will be kept and used as seeds for constructing further candidate solutions.

Surprisingly enough, this idea of applying Darwinian principles to automated problem solving dates back to the forties, long before the breakthrough of computers [12]. As early as in 1948 Turing proposed “genetical or evolutionary search” and already in 1962 Bremermann actually executed computer experiments on “optimization through evolution and recombination". During the sixties three different implementations of the basic idea have been developed at three different places. In the USA Fogel *et al.* introduced evolutionary programming, [13], [11], while Holland called his method a genetic algorithm [15], [16], [20]. In Germany Rechenberg and Schwefel invented evolution strategies [27], [31]. For about 15 years these areas developed separately; it is since the early nineties that they are envisioned as different representatives of one technology that was termed evolutionary computing [1], [2], [3], [20]. It was also in the early nineties that a fourth stream following the general ideas has emerged: Koza's genetic programming [4], [17]. The contemporary terminology denotes the whole field by evolutionary computing and the methods herein evolutionary algorithms. The historical versions evolutionary programming, evolution strategies, genetic algorithms, and genetic programming are seen as sub-types or dialects within the family of EAs.

As the history of the field suggests there are many different variants of evolutionary algorithms. The common underlying idea behind all these techniques is the same. Given an objective function to be maximized we can randomly create a set of candidate solutions, i.e., elements of the objective function's domain that forms the search space, and apply the objective function as an abstract fitness measure – the higher the better. Based on this fitness, some of the better candidates are chosen to seed the next generation by applying so-called variation operators, recombination and mutation, to them. Recombination is a binary variation operator applied to two selected candidates (the so-called parents) and results one or two new candidates (the children). Mutation is a unary variation operator, it is applied to one candidate and results in one new candidate. Executing recombination and mutation leads to a set of new candidates (the offspring) that compete – based on their fitness – with the old ones for a place in the next generation. This cycle can be iterated until a solution is found or a previously set computational limit is reached.

In this process there are two fundamental forces that form the basis of all evolutionary systems:

- *Variation* (implemented through recombination and mutation operators) creates the necessary diversity within the population, thus it facilitates novelty.
- *Selection* (implemented through parent selection and survivor selection operators) acts as a force towards increasing the quality of solutions in the population.

The combined application of variation and selection generally leads to improving fitness values in consecutive populations. It is easy, although somewhat misleading, to view this process as if evolution is optimizing (or at least "approximizing") the fitness function, by approaching the optimal values closer and closer over time.

It should be noted that many components of such an evolutionary process are stochastic. Thus, although during selection fitter individuals have a higher chance of being selected than less fit ones, typically even the weak individuals have a chance of becoming a parent or of surviving. During the recombination process, the choice of which pieces from the parents will be recombined is made at random. Similarly for mutation, the choice of which pieces will be changed within a candidate solution, and of the new pieces to replace them, is made randomly.

It is easy to see that EAs fall into the category of generate-and-test algorithms. The fitness function represents a heuristic estimation of solution quality, and the search process is driven by the variation and selection operators. Evolutionary algorithms possess a number of features that can help to position them among generate-and-test methods:

- EAs are population based, i.e., they process a whole collection of candidate solutions simultaneously.
- EAs mostly use recombination, mixing information from two or more candidate solutions to create a new one.
- EAs are stochastic.

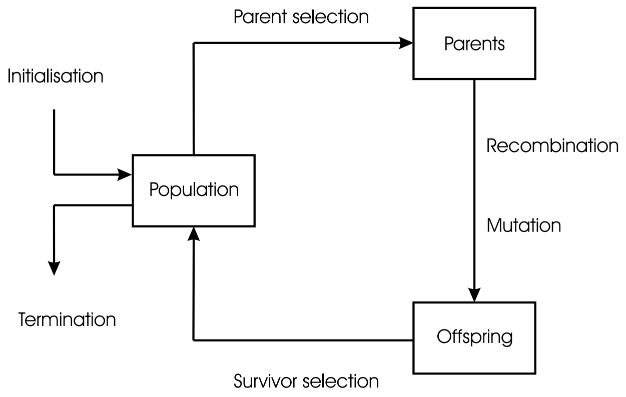

Fig. 1. The general scheme of an evolutionary algorithm as a flow-chart

The various dialects of evolutionary computing that we have mentioned previously all follow the general EA outlines, differing only in technical details. In particular, the representation of a candidate solution is often used to characterize different streams. Typically the representation (i.e., the data structure encoding a candidate solution) has the form of strings over a finite alphabet in genetic algorithms (GAs), real-valued vectors in evolution strategies (ESs), finite state machines in classical evolutionary programming (EP), and trees in genetic programming (GP). The origin of these differences is mainly historical. Technically, one representation might be preferable to others if it matches the given problem better; that is, if it makes the encoding of candidate solutions easier or more natural. For instance, when solving a satisfiability problem with n logical variables, the straightforward choice is to use bit-strings of length n, hence the appropriate EA would be a genetic algorithm. To evolve a computer program that can play checkers, trees are well-suited (namely, the parse trees of the syntactic expressions forming the programs), thus a GP approach is likely. It is important to note that the recombination and mutation operators working on candidates must match the given representation. Thus, for instance, in GP the recombination operator works on trees, while in GAs it operates on strings. In contrast to variation operators, the selection process only takes fitness information into account, and so it works independently from the choice of representation. Therefore differences between the selection mechanisms commonly applied in each stream are a matter of tradition rather than of technical necessity.

It is worth to note that the borders between the four main EC streams are diminishing in the last decade. Approaching EAs from a "unionist" perspective the distinguishing features of different EAs are the algorithmic components, representation, recombination operator, mutation operator, parent selection operator, and survivor selection operator. Reviewing the details of the commonly used operators for these components exceeds the scope of this chapter. For those details we refer to a modern text book, such as [10] or [8], and in the sequel we will use (the names of) such operators without further explanation. Here we restrict ourselves to providing an illustration in Table II showing how particular choices can lead to a typical genetic algorithm or evolution strategy.

TABLE II
A TYPICAL GA AND ES AS AN INSTANTIATION OF THE GENERIC EA SCHEME BY PARTICULAR REPRESENTATION AND OPERATORS

	GA	ES
Representation	bit-strings	real-valued vectors
Recombination	1-point crossover	intermediary
Mutation	bit-flip	Gaussian noise by $N(0, \sigma)$
Parent selection	2-tournament	uniform random
Survivor selection	generational	(μ, λ)
Extra	none	self-adaptation of σ

III. EA Design, EA Parameters

Given a particular problem, designing an EA for solving it requires filling in the details of the generic EA framework appropriately. For a solid basis of this paper we first elaborate on suitable naming conventions regarding these details.

One possibility is to call these details *EA parameters*. In this case, designing an EA for a given application amounts to selecting good values for these parameters. For instance, the definition of an EA might include setting the parameter `crossoveroperator` to `1-point`, 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 and choosing a good value for the related crossover rate $p_c \in [0, 1]$. One feels that the parameters `crossoveroperator` and `crossoverrate` are different.

This difference can be formalized if we distinguish parameters by their domains. The parameter `crossoveroperator` has a finite domain with no sensible distance metric, e.g., $\{$`1-point`, `uniform`, `averaging`$\}$, whereas the domain of the parameter p_c is a subset of $\mathbb{R}$ with the natural metric for real numbers. This difference is essential for searchability. For parameters with a domain that has a distance metric, 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. For a clear distinction between these cases we can use the terms *symbolic parameter* or *qualitative parameter*, e.g., `crossoveroperator`, and *numeric parameter* or *quantitative parameter*, e.g., crossover rate. 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.

An alternative naming convention, (used in [26] for instance) is to call symbolic parameters *components* and the elements of their domains *operators*. In the corresponding terminology a parameter is instantiated by a value, while a component is instantiated by allocating an operator to it. Using this naming convention for the example in the beginning of this section, `crossoveroperator` is a component instantiated by the operator `1-point`, while `crossoverrate` is a parameter instantiated by the value 0.5.

In this paper we adhere to the second terminology distinguishing components and parameters. Further to this, we distinguish two levels in designing a particular EA instance for a given problem by saying that the operators (the high-level, symbolic details) define the EA, while the parameters (the low-level, numerical details) define a variant of this EA. Table III illustrates this matter.

TABLE III
THREE EA INSTANCES SPECIFIED BY THE COMPONENTS RECOMBINATION, MUTATION, PARENT SELECTION, SURVIVOR SELECTION AND THE PARAMETERS MUTATION RATE (p_c), MUTATION STEP SIZE (σ), CROSSOVER RATE (p_c), POPULATION SIZE (μ), OFFSPRING SIZE (λ), AND TOURNAMENT SIZE. THE EA INSTANCES IN COLUMNS A AND B ARE JUST VARIANTS OF THE SAME EA. THE EA INSTANCE IN COLUMN C BELONGS TO A DIFFERENT EA.

	A	B	C
Recombin.	1-point	1-point	averaging
Mutation	bit-flip	bit-flip	Gaussian $N(0, \sigma)$
Parent sel.	tournament	tournament	uniform random
Survivor sel.	generational	generational	(μ, λ)
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
tourn. size	2	4	n.a

This terminology enables precise formulations, meanwhile it enforces care with phrasing. From now on the phrase *an EA for problem X* means a partially specified algorithm where the operators to instantiate EA components are defined, but the parameter values are not. After specifying all details, including the values for all parameters, we obtain *an EA instance for problem X*.

It has long been noticed that EA parameters have a strong influence on EA performance. The problem of setting EA parameters correctly is therefore highly relevant. Setting EA parameters is commonly divided into two cases, parameter tuning and parameter control [9]. 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 there is a large number of options, but only little knowledge about the effect of EA parameters on EA performance. EA users mostly rely on conventions (mutation rate should be low), ad hoc choices (why not use population size 100), and experimental comparisons on a limited scale (testing combinations of three different crossover rates and three different mutation rates).

In these terms we can express the primary focus of this chapter as being parameter tuning. Entropy is proposed as a generic measure of parameter relevance that shows how difficult it is to find parameter values that induce good EA performance. The practical use of this information is obvious. Given an EA (thus, all operators specified), if the relevance levels of the parameters are known then it is possible to allocate tuning efforts such that more relevant parameters are tuned more extensively than less relevant ones. It is important to note that relevance information of EA parameters depends on two other factors: the EA itself (that is, the chosen operators to instantiate EA components) and the problem at hand. The aspect of problem dependence belongs to the issue of scoping, that is, establishing the scope of validity of experimental work. A thorough treatment of this issue exceeds the limitations of this chapter; our case study attempts to cope with this problem by using many problem instances produced by a parameterized random problem instance generator (see Section VI for details). Concerning the dependence on the EA itself, the case study will illustrate that the approach we advocate here is also helpful for deciding about good operators for EA components, thus for designing EAs in general. The basis of such aggregation is the hierarchy between EA components and EA parameters. This hierarchy is visible in Table VIII that arranges parameters by the operators they belong to (with population size as the only exception). Relying on this hierarchy, it is possible to aggregate results concerning parameters to results at the level of operators and thus at the level of EAs.

IV. Shannon and Differential Entropy

As we have seen, an EA can be composed from a wide variety of operators, each with its own numeric parameters that need properly chosen values for satisfactory EA performance. However, choosing proper values, i.e., tuning, requires effort both in terms of time and computing facilities and both resources are limited in practice. Hence, tuning efforts should be carefully allocated to different parameters such that the most relevant parameters receive the most attention and only little effort is spent on finding good values for parameter with limited relevance . The problem is, how to quantify the relevance of parameters.

A. Using Success Ranges for Relevance Estimation

In order to objectively quantify a parameter's relevance, thus the amount of tuning it needs, one can look at how accurately its value needs to be specified for achieving a given performance level. A straightforward approach is to measure which part of the parameter's range leads to the desired performance. For example, let us assume that an EA has two parameters $X_1 \in [0, 1]$ and $X_2 \in [0, 1]$ and that the algorithm reaches some desired performance if X_1 is in the range [0, 0.5] **and** X_2 is in the range of [0, 0.1]. One can argue that X_2 is more relevant, because X_1 has a success range of 50% of its full range [0, 1] and X_2 has a success range of 10%. Furthermore, one can assume that the success range within the full 2D parameter space $[0, 1] \times [0, 1]$ is $50\% \cdot 10\% = 5\%$. However, this kind of reasoning can lead to misleading conclusions if the parameters are not independent. For example, if we have an algorithm that achieves the desired performance if X_1 is in the range [0, 0.5] **or** X_2 is in the range of [0, 0.1]. The success range of X_1 is in that case equal to [0, 1], because the EA instances with parameter values of $\langle 0, 0.05\rangle$ and $\langle 1, 0.05\rangle$ both terminate with success. Similarly, the success range of X_2 is equal to [0, 1] too.

B. Shannon Entropy

Entropy is commonly used to measure the amount of disorder of a system and this concept has been extended in information theory to quantify the uncertainty associated with a random variable. To be precise, the (Shannon) entropy $H(X)$ of a random variable X with probability mass function $p(x)$ can be used to measure the average information content that is missing when the value of X is unknown [32]. Shannon defined the entropy for discrete variables as:

$$H(X) = -\sum^{i} p_i \cdot \log_2 p_i \qquad (1)$$

where p_i is equal to the chance of observing value i and $\log_2$ is the logarithm with base 2.

Notice that a random variable with a large range of different values will have a higher entropy than a random variable with just a few specific values. For example, a fair coin has an entropy of 1 bit. A biased coin has an entropy that is lower, because it will return one side more often than the other. Predicting the next value for a biased coin is easier, lowering the uncertainty. So, entropy can be seen a measure of the extent of bias towards a certain value, or range of values.

C. Using the Shannon Entropy for Relevance Estimation

Let us consider an evolutionary algorithm with two parameters, population size $P \in \{10, 100\}$ and tournament size $T \in \{5, 10\}$. We can now execute the EA 100 times with all possible parameter value combinations and thus experimentally establish whether a given combination is successful. Success here can mean that the EA finds the optimal fitness value in all runs, or that the mean best fitness (MBF) over all runs is above a certain threshold. Table IV shows a possible outcome.

TABLE IV
SUCCESS OR FAILURE FOR DIFFERENT PARAMETER VALUE COMBINATIONS, 1 = SUCCESS, 0 = FAILURE

		Population Size	
		10	**100**
Tournament Size	**5**	1	1
	10	0	1

We can observe that a high population size (100), a low tournament size (5), or a combination of both, leads to success. The list of population sizes that lead to success will therefore be $\{10, 100, 100\}$ with p values of $\frac{1}{3}$ and $\frac{2}{3}$. The entropy of this distribution is therefore $\frac{1}{3} \cdot log_2(\frac{1}{3}) + \frac{2}{3} \cdot log_2(\frac{2}{3}) = 0.92$ bits. Unlike the success range-based measure from Section IV-A, the entropy identifies correctly that there is a bias for one of those two values. Thereby it indicates that it is beneficial to choose the parameter value from a specific area, rather than selecting an arbitrary value. The size of the entropy indicates the size of the area. The lower the entropy, the smaller the area that leads to success.

Furthermore, we can use this approach to show how the desired performance is related to the required tuning effort. To this end we need a fine graded overview of the experimental outcomes that exhibits the mean best fitness over the 100 EA runs belonging to the parameter values used in those runs. Table V shows a possible outcome for five different population sizes and a fixed tournament size (not shown in the table).

TABLE V
MEAN BEST FITNESS FOR DIFFERENT POPULATION SIZES

Population Size	Performance (MBF)
10	0.80
20	0.85
30	0.90
40	0.95
50	1.00

Based on these results, we can calculate the entropy not only for single parameter values, but for a whole range of values. This results in a table containing the desired performance, and the corresponding entropy (Table VI).

We can use such a table or graph to determine how big the set of all possible parameter values is that lead to the desired performance. Furthermore, this can indicate how relevant it is to tune a certain parameter with a specific minimal performance in mind. In this case, each of the 5 population sizes lead to a performance of at least 0.8. The entropy, using a minimal performance of 0.8, is therefore the highest. If we define success as reaching a performance of at least 1.0, then only one setup (population size = 50) results in success. The corresponding entropy is therefore the lowest. In terms of

TABLE VI
THE MINIMAL PERFORMANCE REQUIRED FOR SUCCESS AND THE CORRESPONDING ENTROPY

Performance (MBF) required for Success	Entropy
0.80	2.32
0.85	2.00
0.90	1.59
0.95	1.00
1.00	0.00

(un)certainty, if we observe success in this case, then we know for sure that the EA used population size 50. While observing success in the first case we do not know anything, because all possible population sizes could have caused it.

D. Differential Entropy

The differential entropy is an extension of the Shannon entropy to the domain of continuous probability distributions. This is required for parameters that are real-valued, for example mutation rate. It is clear that calculating the entropy of such parameters requires a somewhat different approach than enumerating on all possible combinations of parameter values.

One approach is to divide the continuous domain in a certain number of bins. Because this makes the domain discrete, we can use the Shannon entropy as described in the previous section. However, the number of bins highly influences the outcomes. One way of dealing with this problem is always using the same amount of bins. This makes the results comparable, but could lead to problems if the number of bins is to small. The best number of bins, would therefore be infinity, which is exactly the approach that is used with the differential entropy.

In order to calculate the differential entropy, it is required that the probability distribution of such a parameter is known. Just as with the Shannon entropy, this can be any distribution. With probability density function $f(x)$, the entropy is defined as:

$$h(X) = -\int_{\mathbb{X}} f(x) \log_2 f(x)\, dx \tag{2}$$

Unlike the Shannon entropy, the differential entropy can get negative, for example, a uniform distribution over the range [0, 0.1] results in a differential entropy of:

$$\begin{aligned} f(x) &= \frac{1}{0.1-0} && (3)\\ h(X) &= -\int_0^{0.1} f(x)\log_2 f(x)\, dx && (4)\\ &= \log_2(0.1) && (5)\\ &= -3.3 && (6)\end{aligned}$$

In order to compare the entropy of distributions that are defined over different parameter ranges in a meaningful way, we normalize the range of all parameters to the unit interval $[0, 1]$ before calculating the entropy. In this way the uniform distribution has a Shannon entropy of zero, and any other distribution has a negative Shannon entropy.

E. Joint Entropy

The notion of entropy as introduced above can be calculated for each specific parameter. However, sometimes we are interested in the entropy of constructs that depend on more then one parameters. For instance, the Gaussian (p, σ) mutation operator is regulated by the mutation probability p and the mutation stepsize σ. The amount of tuning required by this mutation operator will thus depend on the amount of tuning required by two parameters. This idea can be also extended to the level of the algorithm, namely the set of all instantiated operators. For an illustration recall Table III and observe that EA with the instances in columns A and B depends on four parameters and the EA whose instance is shown in column C depends on five. In such situations we need to handle the combination of more parameters, that is to calculate joint entropies. If we assume independence of the parameters in question, then the joint entropy is equal to the sum of the individual entropies. If the parameters are not independent then one should calculate the combined probability density function and use this to calculate the entropy. If this is not possible, one can use lower and upper bounds for the joint entropy that are easy to calculate, because the sum of the individual entropies forms the lower bound and the maximum individual entropy is the upperbound of the joint entropy.

$$h(X \cap Y) \leq \quad h(X) + h(Y) \tag{7}$$

$$h(X \cap Y) \geq \quad max(h(X), h(Y)) \tag{8}$$

To illustrate the usage of such bounds assume that we need information on the relevance of the uniform crossover operator (one parameter, p_c) and the Gaussian (p_m, σ) mutation operator. Assume furthermore that the entropies belonging to p_c, p_m, and σ are known. Then the sum of entropies of the parameters p_m and σ is an upper bound for entropy of the mutation operator (that would correspond to the joint entropy of p_m and σ). Thus, if the sum of entropies of the parameters p_m and σ is lower than the entropy of p_c, then we know that the entropy of this mutation operator is lower than the entropy of this crossover operator. Application to complete EAs with more parameters is similar.

V. Estimating Entropy

Calculating the entropy as proposed in the previous section is a computationally intensive task. Even if one performs a full parameter sweep over thousands of different parameters settings, the resulting entropy is still just an estimation. Although such a sweep can be distributed over multiple machines [30], it is still a very time consuming task. Especially because much time is spent on evaluating parameter settings that are not interesting, because their performance is far from optimal.

There are three different approaches to estimate the entropy more efficiently. Ranking and Selection of parameters can be used to estimate entropy [7] with less effort. The principle is not very different from a full parameter sweep, however, instead of assigning each parameter setting the same computational effort, Ranking and Selection focuses on the areas with a high utility. This results in a better estimated entropy, and expectedly, a higher level of utility with the same computational effort.

Secondly, one could build models of the utility landscape and calculate the entropy through the model. There are several approaches to create such models, for example, Sequential Parameter Optimization [5], [6] and Response Surface Models[29]. Some of those models can directly be translated into a probability density function, for which the entropy is given in Equation 2. In other cases, a sweep over all possible parameter values can be used to calculate the entropy of the model. Because utility is estimated by the model, rather than tested, the entropy can be estimated efficiently.

Finally, one could use heuristic search methods that iteratively generate parameter vectors to be tested and used to calculate entropy. The search heuristic should represent a bias towards better parameter vectors thus allocating more computational efforts to interesting areas of the search space. Because of this bias, the estimations of entropy will be better in high utility regions, quite similarly to Ranking and Selection methods. At this moment we only know of one method in this category: REVAC (Relevance Estimation and VAlue Calibration) [23], [24]. REVAC has been developed to aid the design of evolutionary mechanisms for simulation and optimization in application areas without much knowledge on successful EA designs [25]. The main activities of REVAC can be summarized as follows. Given a problem to be solved and an EA to solve it with

- REVAC finds parameter vectors with high utility,
- REVAC collects values of entropy for different utility levels,
- REVAC creates a distribution for each parameter that indicates the expected utility of parameter values.

It is important to note that REVAC does not handle parameter interactions (no joint distributions for multiple parameters) and that it can be used for tuning numeric parameters only.

The case study, described in Section VI, is based on estimating entropy values. In principle, the experiments could have been conducted using any of the methods mentioned above, but actually we have used REVAC [26]. Therefore, we describe it in detail in the sequel.

A. REVAC: the Algorithm

Technically, REVAC is a heuristic generate-and-test method that is iteratively adapting a set of parameter vectors of a given EA. Testing a parameter vector is done by executing the EA with the given parameters and measuring the EA performance. EA performance can be defined by any appropriate performance measure, or combination of performance measures, and the results will reflect the utility of the parameter vector in question. Because of the stochastic nature of EAs, in general a number of runs is advisable to obtain better statistics.

For a good understanding of the REVAC method it is helpful to distinguish two views on a given set of parameter vectors as shown in Table VII. Taking a *horizontal* view on

TABLE VII
TWO VIEWS ON A TABLE OF PARAMETER VECTORS.

	$\mathcal{D}(x_1)$	$\cdots$	$\mathcal{D}(x_i)$	$\cdots$	$\mathcal{D}(x_k)$	Utility
$\vec{x}^1$	$\{x_1^1$	$\cdots$	x_i^1	$\cdots$	$x_k^1\}$	u^1
$\vdots$		$\ddots$				$\vdots$
$\vec{x}^n$	$\{x_1^n$	$\cdots$	x_i^n	$\cdots$	$x_k^n\}$	u^n
$\vdots$				$\ddots$		$\vdots$
$\vec{x}^m$	$\{x_1^m$	$\cdots$	x_i^m	$\cdots$	$x_k^m\}$	u^m

the table, each row shows the name of a vector (first column), the k parameter values of this vector, and the utility of this vector (last column), defined through the performance of the EA in question. However, taking a *vertical* view on the table, the i^{th} column in the inner box shows m values from the domain of parameter i and this can be seen as a distribution over the range of that parameter.

To understand how REVAC is generating parameter vectors the horizontal view is more helpful. From this perspective, REVAC can be described as an evolutionary algorithm, in the style of EDAs [21], working on a population of m parameter vectors. This population is updated by selecting parent vectors, which are then recombined and mutated to produce one child vector that is then inserted into the population. The exact details are as follows.

- **Parent selection** is deterministic in REVAC as the best n ($n < m$) vectors of the population, i.e., those with the highest utility, are selected to become the parents of the new child vector. For further discussion we denote the set of parents by $\{\vec{y}^1, \ldots, \vec{y}^n\} \subset \{\vec{x}^1, \ldots, \vec{x}^m\}$.
- **Recombination** is performed by a multi-parent crossover operator, uniform scanning. In general, this operator can be applied to any number of parent vectors and the ith value in the child $\langle c_1, \ldots, c_k\rangle$ is selected uniformly random from the $i$the values, $y_i^1, \ldots, y_i^n$, of the parents. Here we create one child from the selected n parents.
- **Mutation**, applied to the offspring $\langle c_1, \ldots, c_k\rangle$ created by recombination, works independently on each parameter $i \in \{1, \ldots, k\}$ in two steps. First, a mutation interval $[a_i, b_i]$ is calculated, then a random value is chosen uniformly from this interval. The mutation interval for a given c_i is determined by all values $y_i^1, \ldots, y_i^n$ for this parameter in the selected parents as follows. First, the parental values are sorted in increasing order such that $y_i^1 \leq \cdots \leq y_i^n$. (Note, that for the sake of readability, we do not introduce new indices corresponding to this ordering.) Recall that the child $\langle c_1, \ldots, c_k\rangle$ is created by uniform scanning crossover, hence the value c_i comes from one of the parents. That is, $c_i = y_i^j$ for some $j \in \{1, \ldots, n\}$ and we can define the neighbors of c_i as follows. The first neighbors of c_i are y_i^{j-1} and y_i^{j+1}, the second neighbors are y_i^{j-2} and y_i^{j+2}, the third neighbors are y_i^{j-3} and y_i^{j+3}, etc. Now, the begin point a_i of the mutation interval is defined as the h-th lower neighbor of c_i, while the end point of the interval b_i is the h-th upper neighbor of c_i, where h is a parameter of the REVAC method (as there are no neighbors beyond the upper and lower limits of the domain, we extend it by mirroring the parent values as well as the mutated values at the limits). The mutated value c'_i is drawn from this mutation interval $[a_i, b_i]$ with a uniform distribution and the child $\langle c'_1, \ldots, c'_k\rangle$ is composed from these mutated values.
- **Survivor selection** is also deterministic in REVAC as the newly generated vector always replaces the oldest vector in the population.
- **Evaluation** The newly generated vector is tested by running the EA in question with the values it contains.

The above list describes one REVAC cycle that is iterated until the maximum number of vectors tested is reached.

B. REVAC: the Data Generated

In each REVAC cycle several data records are saved to allow analysis after termination. This happens directly after the n parent vectors are selected from the population. First, the lowest utility in the set of parents is identified as $u = min\{u^1, \ldots, u^n\}$. Then for each parameter $i \in \{1, \ldots, k\}$ we calculate the entropy e_i and store the pair $\langle e_i, u\rangle$. The calculation of e_i is based on the set $\{y_i^1, \cdots, y_i^n\}$ of parental values for parameter i that we consider to be a representative sample of good parameter values – 'good' defined as leading to a utility higher than u.[1]

For the calculation of e_i we use the formula for differential entropy (Equation 2) applied to the probability density function

$$f^i(z) = \frac{1}{(n+1)\cdot(b(z)-a(z))} \tag{9}$$

where $a(z)$ and $b(z)$ are the h-th lower and upper neighbor of z, respectively.

For determining the (h-th) neighbors of any given z, we use a method similar to the procedure for defining the neighbors of c_i in the description of the mutation operator. However, in general, there need not be a $j \in \{1, \ldots, n\}$ such that $z = y_i^j$. Hence, the index j is now defined as the one satisfying $y_i^j \leq z < y_i^{j+1}$ and we call y_i^j and y_i^{j+1} the first neighbors of z, y_i^{j-1} and y_i^{j+2} its second neighbors, y_i^{j-2} and y_i^{j+3} its third neighbors, etc. Now, $a(z)$ and $b(z)$ in Equation 9 are the h-th lower neighbor and the h-th upper neighbor of z, respectively .

Calculating the entropy e_i for all $i = 1, \ldots, k$ we get k pairs, $\langle e_1, u\rangle$ through $\langle e_k, u\rangle$, one for each parameter. These pairs can be used for making plots of performance levels (parameter vector utilities) and entropy values.

[1]In a previous study [26] we associated the entropy e_i with the expected utility $v = avg\{u^1, \ldots, u^n\}$ among the parents. In other words, we defined 'good' as leading to an expected utility v.

TABLE VIII

EA COMPONENTS, OPERATORS, AND PARAMETERS USED IN THIS STUDY

Component	**Operator**	**Parameter(s)**
		population size μ
parent selection	tournament	(parent) tournament size
	random uniform	-
	fitness proportional	-
	best selection	number n of best
survivor selection	generational	-
	tournament	(survivor) tournament size
	random uniform	-
	(μ, λ)	λ
	$(\mu + \lambda)$	λ
recombination	none	-
	one-point	crossover probability
	uniform	crossover probability
mutation	reset	mutation probability
	Gaussian$(\sigma, 1)$	step size
	Gaussian(σ, p)	step size, mutation probability

VI. CASE STUDY

In this section we present a case study, based on data generated by a large experimental investigation [22], [26]. Our case study will present entropy data that is inherently produced by every REVAC run. Strictly speaking, the use of REVAC generated data implies that we are not showing results on entropy, but results on the estimation of entropy as done by REVAC. Our discussion, however, will be in general terms since it could be presented along the same lines with any other similar method for entropy estimations.

A. Experimental Setup

For a clear discussion we separate three different levels that can be distinguished in the context of algorithm design.

1) The problem/application (here: fitness landscapes created by a problem generator).
2) The problem solver (here: an Evolutionary Algorithm).
3) The design method for calibrating the problem solver (here: REVAC).

To obtain concrete problem instances to be solved by the EAs we use a parameterized random problem instance generator that produces real-valued fitness landscapes or objective functions to be maximized. This generator [14] defines a class of landscapes formed by the Max-Set of Gaussian curves in high dimensional Cartesian spaces. Where a Gaussian mixture model takes the average of several Gaussians, a max-set takes their enveloping maximum. In this way, the complexity of maximizing a Gaussian mixture can be combined with full control over the location and height of global and local maxima. For this study we selected problem set 4 from [14] with peaks that get higher the closer they get to the origin. Using 10 dimensions, 100 Gaussians and the same distributions over height, location, and rotation of these Gaussians we generated 10 test landscapes by different random seeds.

For the EA we use the open source Evolutionary Computation toolkit in Java (ECJ) [19]. ECJ allows the specification of a full EA through a simple parameter file. Obviously, we do not use all possibilities ECJ offers, but select a number of operators for the EA components and run REVAC for all those EAs that can be obtained by the combinations of these operators. To be specific, we base our study on the components parent selection, survivor selection, recombination, and mutation, with three to five commonly used operators for each as shown in Table VIII. We follow the naming convention of ECJ. For any given EA, the population size parameter is always present, other parameters depend on the actual chosen operators. Due to technical details in ECJ, only 10 different combinations of parent and survivor selection operators are possible[2]. Together with 3 choices for the recombination operator and 3 choices for the mutation operator, this yields 90 different EAs to be tuned, of which 6 EAs with 2, 27 with 3, 38 with 4, 17 with 5, and 2 with 6 free parameters. Most operators have one or no parameter to calibrate. One operator has 2 parameters—Gaussian(σ, p) with free parameters σ for step size and p for mutation probability, the latter set to one in the case of Gaussian$(\sigma, 1)$.

The basic data nuggets in this case study are produced by REVAC runs with a given EA on one of the 10 test landscapes. In one run REVAC is allowed to generate and test 1000 parameter vectors. Generating parameter vectors is done through the main REVAC loop using $m = 100$ vectors to form the population and selecting the best $n = 50$ of them as parents to create one child vector by uniform scanning crossover. REVAC's smoothing parameter used in the mutation operator is set at $h = 5$. Testing parameter vectors happens by executing 10 independent runs of the

[2]Arguably, (μ, λ) and $(\mu + \lambda)$ define both parent *and* survivor selection. Here we classify them under survivor selection because that is what the parameter λ influences.

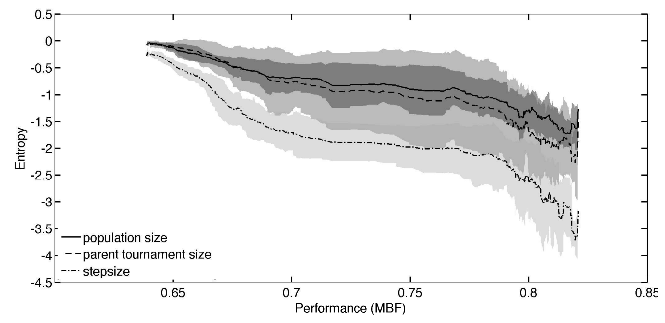

Fig. 2. Parameter entropy plot for EA-1: {Tournament Parent Selection, Generational Survivor Selection, No Crossover and Gaussian(σ)}

given EA using the given vector. The utility of a vector is measured by the performance of the EA using that vector, which is in turn measured by the mean best fitness. That is, the best fitness value after each run, averaged over the 10 independent runs.

The experimental data used in this section are generated by carrying out 10 REVAC runs with all of the 90 EAs on all of the 10 test landscapes. The basic data points are pairs of estimated entropy values and corresponding performance levels, as explained in section V-B. All together we have 901 of such pairs per REVAC run (because after initialization 900 generations have passed before REVAC terminates), hence the plots shown in the sequel are based on $10 \times 901 = 9010$ pairs.

B. Entropy of Parameters

Each of the 90 EAs can be individually analyzed to get insight into the relevance of its parameters. For example, let EA-1 be the evolutionary algorithm defined by the operators {Tournament Parent Selection, Generational Survivor Selection, No Crossover and Gaussian(σ)} for its components. EA-1 has three parameters, namely population size, tournament size, and mutation stepsize (σ). For each of these parameters we can plot the entropy for reaching a specific performance using our REVAC generated data.

Figure 2 exhibits the mean and standard deviation of the estimated entropy for a given performance. From this figure it is clear that mutation stepsize is the most relevant parameter, and the other two parameters do not differ significantly w.r.t. their relevance. For example, if the desired performance is equal to 0.75, the entropy of stepsize is equal to -2, while the population size and tournament size have an entropy higher than -1. This means that the mutation stepsize has the smallest range of values that lead to a performance of at least 0.75. Thus, for this EA it is advisable to dedicate the most tuning effort to the stepsize parameter.

Furthermore, we can observe a rather straight line of the average entropy of stepsize between approximately 0.7 and 0.78. This indicates that the size of the area that leads to values of at least 0.7 is equal to the size of the area that leads to a performance of 0.78 or higher. Therefore, we can conclude that it is equally hard to tune the EA for reaching a performance of 0.7 or to a performance of 0.78. However, for a higher performance, not only stepsize, but also the other parameters need to be carefully set.

C. Entropy of Operators

In general, operators can have zero, one, or more parameters. For the following example we use an EA that has an operator with two parameters, EA-2, defined by {Tournament Parent Selection, Generational Survivor Selection, No Crossover and Gaussian(p, σ)}. The Gaussian(p, σ) mutation operator is an extension of the Gaussian(σ) operator, where the probability p of applying the mutation operator is a parameter. (The usual option of always applying mutation is now a special case belonging to $p = 1$). As we will show, a parameter entropy plot similar to the previous figure could lead to incorrect conclusions in this case.

Looking at Figure 3 we can observe that the entropy levels of the parameters do not differ very much, all somewhere between -1.5 and -2.5 for a range of performance levels between 0.7 and 0.8. This implies that these **parameters** require roughly equal tuning effort. However, one can not infer from this figure that the mutation **operator** requires roughly the same tuning effort as the other details of the algorithm. Because the Gaussian(p,σ) operator has two parameters, both parameters have to be taken into account. As explained in section IV-E, the joint entropy of two parameters can be estimated by the sum of both entropies. This leads to Figure 4, where entropy data is elevated from parameter level to operator level, except for population size, of course.[3] Note that for tournament selection (and in general for all operators with just one parameter) the parameter level and the operator level plots are identical. This figure clearly shows that Gaussian(p,σ) is by far the most relevant operator of EA-2, requiring the most tuning effort.

D. Entropy of EAs

The previous examples illustrated matters within one given EA. In particular, we showed that the comparison of entropies can provide valuable information on which design detail (parameter or operator) is the most relevant, and thus requires the most tuning effort. Because this all happened in the context of one EA, the operators were only meant to instantiate different EA components (within the given EA). In this section we consider information on different operators for the same component (thus leading to different EAs). We can distinguish these two cases by the main question that can be answered by looking at the data plots. In the previous subsections the question was "Given an EA, which parameter, respectively operator, of this EA needs to be tuned most carefully for a given level of desired performance?", whereas here we address the following question: "Given all operators for instantiating an EA component, which of these operators implies the most effort for tuning?"

Posing this question concerning mutation operators, we might try to get an answer from the previous figures on the

[3]Recall that EA-2 does not use an operator for the crossover component and that the Survivor selection component is instantiated through "generational" that has no parameters.

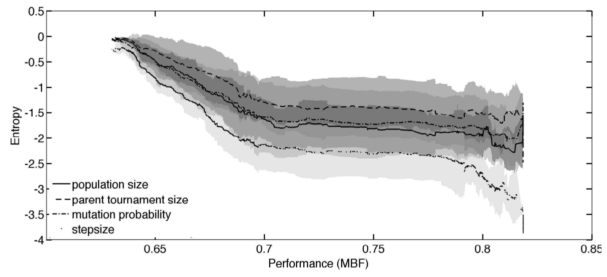

Fig. 3. Parameter entropy plot for EA-2: {Tournament Parent Selection, Generational Survivor Selection, No Crossover and Gaussian(p, σ)}

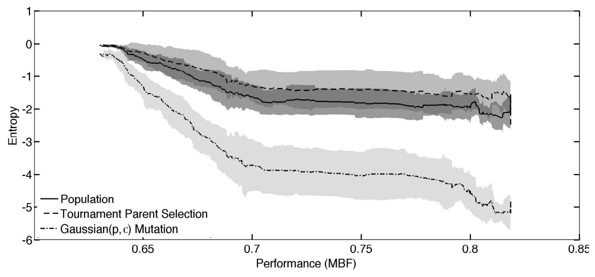

Fig. 4. Operator entropy plot for EA-2: {Tournament Parent Selection, Generational Survivor Selection, No Crossover and Gaussian(p, σ)} (Mind the different scale w.r.t. Figure 3

average entropies of the Gaussian(σ) and Gaussian(p,σ) operators. The entropy levels are around -3 and -5 for reaching a performance of 0.8, respectively. This indicates that the Gaussian(p,σ) mutation operator requires more tuning effort to reach the same performance than the Gaussian(σ) operator. However, this would be a too hasty conclusion because it ignores the influence of other operators and parameters. It is therefore important to compare the total entropy of an algorithm design.

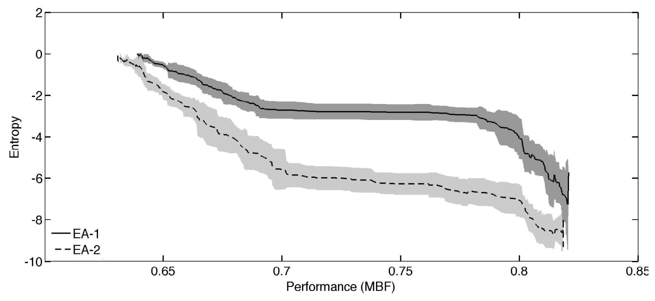

Fig. 5. Algorithm entropy plot for EA-1 (with the Gaussian(σ) mutation operator) and EA-2 (with the Gaussian(p,σ) mutation operator)

Figure 5 shows these algorithm entropies for EA-1 and EA-2. The curves show that the Gaussian(σ) operator indeed causes a higher algorithm entropy, and requires therefore probably less tuning. The difference on the algorithm level is even bigger than we observed in Figure 4 and Figure 2 regarding the operator level.

A similar comparison can be done for the crossover component. To this end, we define EA-3 as {Tournament Parent Selection, Generational Survivor Selection, One Point Crossover, and Gaussian(σ) Mutation} and compare it with EA-1 that used no crossover at all. The plots are shown in Figure 7. Obviously, EA-1 has a crossover operator entropy of 0 at all performance levels, while the crossover operator entropy of EA-3 is most likely lower than 0. Therefore, one would expect that the algorithm entropy of EA-3 is lower than the algorithm entropy of EA-1. However, the absence of crossover makes that variation is only regulated by a single parameter. This causes the range of possible σ values that lead to success highly decreases if the required performance increases. In this case, the entropy of σ decreases significantly (Figure 7). The total algorithm entropy of the algorithm without crossover is therefore lower than the algorithm entropy of the algorithm with one-point crossover (Figure 6). This indicates that adding crossover to the algorithm decreases tuning effort.

VII. Conclusions

In this chapter we illustrated how entropy can be used to obtain useful information on evolutionary algorithm parameters. The main problem we consider here is the tuning of EA parameters, that is, the process of searching for good parameter values in the space of all possibilities. The related challenges are rooted in the facts that 1) EA parameters need to be instantiated with good values for good EA performance, 2) some EA parameters are more relevant than others in the sense that choosing different values for them affects EA performance more than for those other parameters, 3) to maximize the effectivity of parameter tuning, tuning efforts should be allocated such that more relevant parameters are given more time/capacity to find good values than less relevant parameters.

As we explained, entropy (two versions discussed) is a good indicator of parameter relevance. Furthermore, we described the REVAC method [22] that can be used to collect data on specific performance levels and corresponding entropy values. Using a large data set generated by many REVAC runs we created plots concerning the (estimated) entropy of EA parameters, EA operators, and complete EAs as well. For the sake of correctness, let us recall that the entropy values in the source data set are calculated from a pool of parameter vectors as known to REVAC at a given moment, the performance level associated with an entropy value is the utility of the worst vector in the pool, that the combined entropies in our plots are estimated by the sum of the underlying entropies, and that our data were based on just one problem (be it more random instances of it). With these caveats in mind we presented a number of case studies to illustrate the kind of knowledge that can be gained through the entropy-based approach. These case studies showed that it is indeed possible to distinguish parameters and operators that need more tuning than others and that differences in relevance can be quantified. Our entropy-based analysis also disclosed a formerly unknown advantage of using crossover in an EA: It reduces the tuning

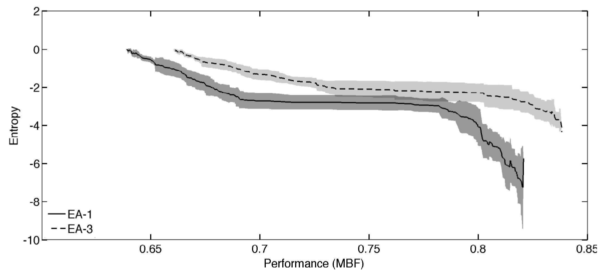

Fig. 6. Algorithm entropy plot for EA-1 (without crossover) and EA-3 (with crossover)

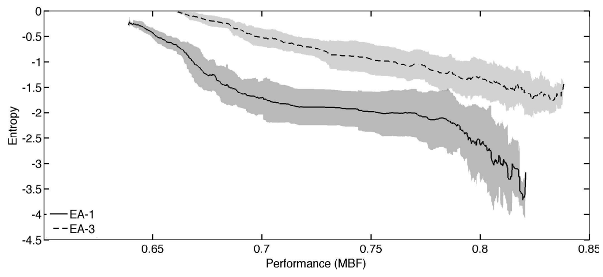

Fig. 7. Mutation stepsize entropy plot for EA-1 (without crossover) and EA-3 (with crossover)

effort associated with mutation. This kind of knowledge has immediate practical relevance in guiding users when tuning their EAs and it has more theoretical advantages for it can disclose relationships between parameters in general. The scope of validity of such insights can be small (the set of problem instances used in the experiments), medium range (more problems of a certain type), or rather generic ("all" EAs and "all" problems). Although the case studies here serve mainly as illustration, not as crisp claims, we do believe that many of our findings hold in more cases than just the 10 landscapes and the 90 EAs we used here and that more of such findings are possible. This, however, requires more experimental research in the future.

ACKNOWLEDGEMENT

All experimental data and results are due to Volker Nannen.

REFERENCES

[1] T Bäck. *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, Oxford, UK, 1996.

[2] T Bäck, D.B Fogel, and Z Michalewicz, editors. *Evolutionary Computation 1: Basic Algorithms and Operators*. Institute of Physics Publishing, Bristol, 2000.

[3] T Bäck, D.B Fogel, and Z Michalewicz, editors. *Evolutionary Computation 2: Advanced Algorithms and Operators*. Institute of Physics Publishing, Bristol, 2000.

[4] W Banzhaf, P Nordin, R.E Keller, and F.D Francone. *Genetic Programming: An Introduction*. Morgan Kaufmann, San Francisco, 1998.

[5] 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.

[6] Thomas Bartz-Beielstein. *Experimental Research in Evolutionary Computation—The New Experimentalism*. Natural Computing Series. Springer, 2006.

[7] 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.

[8] K.A. De Jong. *Evolutionary Computation: A Unified Approach*. The MIT Press, 2006.

[9] A.E. Eiben, R. Hinterding, and Z. Michalewicz. Parameter Control in Evolutionary Algorithms. *IEEE Transactions on Evolutionary Computation*, 3(2):124–141, 1999.

[10] A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computation*. Natural Computing Series. Springer, 2003.

[11] D.B Fogel. *Evolutionary Computation*. IEEE Press, 1995.

[12] D.B Fogel, editor. *Evolutionary Computation: the Fossil Record*. IEEE Press, Piscataway, NJ, 1998.

[13] L.J Fogel, A.J Owens, and M.J Walsh. *Artificial Intelligence through Simulated Evolution*. Wiley, Chichester, UK, 1966.

[14] M. Gallagher and B. Yuan. A General-Purpose Tunable Landscape Editor. *IEEE Transactions on Evolutionary Computation*, 10(5):590–603, 2006.

[15] D.E Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

[16] J.H Holland. *Adaption in Natural and Artificial Systems*. MIT Press, Cambridge, MA, 1992. 1st edition: 1975, The University of Michigan Press, Ann Arbor.

[17] J.R Koza. *Genetic Programming*. MIT Press, Cambridge, MA, 1992.

[18] Hod Lipson, editor. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2007)*. ACM, 2007.

[19] S. Luke et al. A java-based evolutionary computation research system. http://www.cs.gmu.edu/∼eclab/projects/ecj/.

[20] M Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, 1996.

[21] H. Mühlenbein and R. Höns. The Estimation of Distributions and the Minimum Relative Entropy Principle. *Evolutionary Computation*, 13(1):1–27, 2005.

[22] V. Nannen. *Evolutionary Agent-Based Policy Analysis in Dynamic Environments*. PhD thesis, Vrije Universiteit Amsterdam, April, 2009.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] I Rechenberg. *Evolutionstrategie: Optimierung Technisher Systeme nach Prinzipien des Biologischen Evolution*. Fromman-Hozlboog Verlag, Stuttgart, 1973.

[28] E. Ridge and D. Kudenko. Screening the parameters affecting heuristic performance. In Lipson [18], pages 180–180.

[29] Enda Ridge and Daniel Kudenko. Analyzing heuristic performance with response surface models: prediction, optimization and robustness. In Lipson [18], pages 150–157.

[30] M.E. Samples, M.J. Byom, and J.M. Daida. Parameter sweeps for exploring parameter spaces of genetic and evolutionary algorithms. In F.G. Lobo, C.F. Lima, and Z. Michalewicz, editors, *Parameter Setting in Evolutionary Algorithms*, pages 161–184. Springer, 2007.

[31] H.-P Schwefel. *Evolution and Optimum Seeking*. Wiley, New York, 1995.

[32] C.E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 623–656, 1948.