Parameter Control in Evolutionary Algorithms

A.E. Eiben[1], Z. Michalewicz[2], M. Schoenauer[3], and J.E. Smith[4]

[1] Free University Amsterdam, The Netherlands gusz@cs.vu.nl
[2] University of Adelaide, Australia zbyszek@cs.adelaide.edu.au
[3] INRIA Futurs, France marc.schoenauer@inria.fr
[4] UWE, United Kingdom james.smith@uwe.ac.uk

1 Abstract

The issue of setting the values of various parameters of an evolutionary algorithm is crucial for good performance. In this paper we discuss how to do this, beginning with the issue of whether these values are best set in advance or are best changed during evolution. We provide a classification of different approaches based on a number of complementary features, and pay special attention to setting parameters on-the-fly. This has the potential of adjusting the algorithm to the problem while solving the problem.

This paper is intended to present a survey rather than a set of prescriptive details for implementing an EA for a particular type of problem. For this reason we have chosen to interleave a number of examples throughout the text. Thus we hope to both clarify the points we wish to raise as we present them, and also to give the reader a feel for some of the many possibilities available for controlling different parameters.

2 Introduction

Finding the appropriate setup for an evolutionary algorithm is a long standing grand challenge of the field [23, 27]. The main problem is that the description of a specific EA contains its components, such as the choice of representation,selection, recombination, and mutation operators, thereby setting a framework while still leaving quite a few items undefined. For instance, a simple GA might be given by stating it will use binary representation, uniform crossover, bit-flip mutation, tournament selection, and generational replacement. For a full specification, however, further details have to be given, for instance, the population size, the probability of mutation p_m and crossover p_c, and the tournament size. These data – called the **algorithm parameters** or **strategy parameters** – complete the definition of the EA and are necessary to produce an executable version. The values of these parameters

greatly determine whether the algorithm will find an optimal or near-optimal solution, and whether it will find such a solution efficiently. Choosing the right parameter values is, however, a hard task.

Globally, we distinguish two major forms of setting parameter values: **parameter tuning** and **parameter control**. By parameter tuning we mean the commonly practised approach that amounts to finding good values for the parameters *before* the run of the algorithm and then running the algorithm using these values, which remain fixed during the run. Later on in this section we give arguments that any static set of parameters having the values fixed during an EA run seems to be inappropriate. Parameter control forms an alternative, as it amounts to starting a run with initial parameter values that are changed *during* the run.

Parameter tuning is a typical approach to algorithm design. Such tuning is done by experimenting with different values and selecting the ones that give the best results on the test problems at hand. However, the number of possible parameters and their different values means that this is a very time-consuming activity. Considering four parameters and five values for each of them, one has to test $5^4 = 625$ different setups. Performing 100 independent runs with each setup, this implies 62,500 runs just to establish a good algorithm design.

The technical drawbacks to parameter tuning based on experimentation can be summarised as follows:

- Parameters are not independent, but trying all different combinations systematically is practically impossible.
- The process of parameter tuning is time consuming, even if parameters are optimised one by one, regardless of their interactions.
- For a given problem the selected parameter values are not necessarily optimal, even if the effort made for setting them was significant.

This picture becomes even more discouraging if one is after a "generally good" setup that would perform well on a range of problems or problem instances. During the history of EAs considerable effort has been spent on finding parameter values (for a given type of EA, such as GAs), that were good for a number of test problems. A well-known early example is that of [20], determining recommended values for the probabilities of single-point crossover and bit mutation on what is now called the DeJong test suite of five functions. About this and similar attempts [35, 66], it should be noted that genetic algorithms used to be seen as robust problem solvers that exhibit approximately the same performance over a wide range of problems [34, page 6]. The contemporary view on EAs, however, acknowledges that specific problems (problem types) require specific EA setups for satisfactory performance [12]. Thus, the scope of "optimal" parameter settings is necessarily narrow. There are also theoretical arguments that any quest for generally good EA, thus generally good parameter settings, is lost a priori, such as the No Free Lunch theorem [90].

To elucidate another drawback of the parameter tuning approach recall how we defined it: finding good values for the parameters before the run of the algorithm and then running the algorithm using these values, *which remain fixed during the run*. However, a run of an EA is an intrinsically dynamic, adaptive process. The use of rigid parameters that do not change their values is thus in contrast to this spirit. Additionally, it is intuitively obvious, and has been empirically and theoretically demonstrated, that different values of parameters might be optimal at different stages of the evolutionary process [6, 7, 8, 17, 40, 46, 67, 70, 72, 76, 77, 82, 83].

To give an example, large mutation steps can be good in the early generations, helping the exploration of the search space, and small mutation steps might be needed in the late generations to help fine-tune the suboptimal chromosomes. This implies that the use of static parameters itself can lead to inferior algorithm performance.

A straightforward way to overcome the limitations of static parameters is by replacing a parameter p by a function $p(t)$, where t is the generation counter (or any other measure of elapsed time). However, as indicated earlier, the problem of finding optimal static parameters for a particular problem is already hard. Designing optimal dynamic parameters (that is, functions for $p(t)$) may be even more difficult. Another possible drawback to this approach is that the parameter value $p(t)$ changes are caused by a "blind" deterministic rule triggered by the progress of time t, without taking any notion of the actual progress in solving the problem, i.e., without taking into account the current state of the search. A well-known instance of this problem occurs in simulated annealing [50] where a so-called cooling schedule has to be set before the execution of the algorithm.

Mechanisms for modifying parameters during a run in an "informed" way were realised quite early in EC history. For instance, evolution strategies changed mutation parameters on-the-fly by Rechenberg's 1/5 success rule using information on the ratio of successful mutations. Davis experimented within GAs with changing the crossover rate based on the progress realised by particular crossover operators [17]. The common feature of these and similar approaches is the presence of a human-designed feedback mechanism that utilises actual information about the search process for determining new parameter values.

Yet another approach is based on the observation that finding good parameter values for an evolutionary algorithm is a poorly structured, ill-defined, complex problem. This is exactly the kind of problem on which EAs are often considered to perform better than other methods. It is thus a natural idea to use an EA for tuning an EA to a particular problem. This could be done using two EAs: one for problem solving and another one – the so-called meta-EA – to tune the first one [33, 35, 49]. It could also be done by using only one EA that tunes itself to a given problem, while solving that problem. Self-adaptation, as introduced in Evolution Strategies for varying the mutation

parameters, falls within this category. In the next section we discuss various options for changing parameters, illustrated by an example.

3 Case study: Evolution Strategies

The history of Evolution Strategies (ES) is a typical case study for parameter tuning, as it went through several successive steps pertaining to many of the different approaches listed so far.

Typical ES work in a real-valued search space (typically $\mathbb{R}^n$, or a subset of $\mathbb{R}^n$, for some integer n).

3.1 Gaussian mutations

The main operator (and almost the trademark) of ES is the Gaussian mutation, that adds centred normally distributed noise to the variables of the individuals. The most general Gaussian distribution in $\mathbb{R}^n$ is the multivariate normal distribution $\mathcal{N}(m, C)$, with mean m and **covariance matrix** C, a $n \times n$ positive definite matrix, that has the following Probability Distribution Function

$$\Phi(X) = \frac{\exp(-\frac{1}{2}(X-m)^t C^{-1}(X-m))}{\sqrt{(2\pi)^n |C|}}$$

where $|C|$ is the determinant of C.

It is then convenient to write the mutation of a vector $X \in \mathbb{R}^n$ as

$$X \to X + \sigma N(0, C)$$

i.e. to distinguish a scaling factor σ, also called the **step-size**, from the directions of the Gaussian distribution given by the covariance matrix C.

For example, the simplest case of Gaussian mutation assumes that C is the identity matrix in $\mathbb{R}^n$ (the diagonal matrix that has only 1s on the diagonal). In this case, all coordinates will be mutated independently, and will have added to them some Gaussian noise with variance σ^2.

Tuning an ES algorithm therefore amounts to tuning the step-size and the covariance matrix – or simply tuning the step-size in the simple case mentioned above.

3.2 Adapting the step-size

The step-size of the Gaussian mutation gives the scale of the search. To make things clear, suppose that you are minimising x^2 in one dimension, running a (1+1)-ES (one parent gives birth to one offspring, and the best of both is the next parent) with a fixed step-size σ. Then the average distance between parent and successful offspring is proportional to σ. This has two consequences:

first, starting from distance d_0 from the solution, it will take an average of d_0/σ steps to reach a region close to the optimum. On the other hand, when hovering around the optimum, the precision you can hope is again proportional to σ. Those arguments naturally lead to the optimal **adaptive** setting of the step-size for the sphere function: σ should be proportional to the distance to the optimum. Details can be found in the studies of the so-called *progress rate*: early work was done by Schwefel [70], completed and extended by Beyer and recent work by Auger [5] gave a formal global convergence proof of this ... impractical algorithm: indeed, the distance to the optimum is not known in real situations!

But another piece of information is always available to the algorithm: the *success rate* (the proportion of successful mutations, where the offspring is better than the parent). This can indirectly give information about the step-size: this was Rechenberg's main idea to propose the first practical method for an **adaptive** step-size, the so-called *one-fifth rule*: if the success rate over some time window is larger than the success rate when the step-size is optimal (0.2, or one-fifth!), the the step-size should be increased (the algorithm is making too many small steps); on the other hand, if the success rate is smaller than 0.2, then the step-size should be decreased (the algorithm is constantly missing the target, because it shoots too far). Though formally derived from studies on the sphere function and the corridor function (a half-bounded linear function), the one-fifth rule was generalised to any function.

However, there are many situations where the one-fifth rule can be mislead. Moreover, it does not in any way handle the case of non-isotropic functions, where a non-diagonal covariance matrix is mandatory. Hence, it is no longer used today.

3.3 Self-Adaptive ES

The next big step in ESs was the invention of the **self-adaptive** mutation: the parameters of the mutation (both the step-size and the covariance matrix) are attached to each individual, and are subject to mutation, too. Those personal mutation parameters range from a single step-size, leading to *isotropic* mutation, where all coordinates are mutated independently with the same variance, to the *non-isotropic* mutation, that use a vector of n "standard deviations" σ_i, equivalent to a diagonal matrix C with σ_i on the diagonal, and to the *correlated* mutations, where a full covariance matrix is attached to each individual. Mutating an individual then amounts to first mutating the mutation parameters themselves, and then mutating the variables using the new mutation parameters. Details can be found in [70, 13].

The rationale for SA-ES are that the algorithm relies on the selection step to keep in the population not only the best fit individuals, but also the individuals with the best mutation parameters – according to the region of the search space they are in. Indeed, although the selection acts based on the fitness, the underlying idea beneath Self-Adaptive ES (SA-ES) is that if two

individuals starts with the same fitness, the offspring of one that has "better" mutation parameters will reach regions of higher fitness faster than the offspring of the other: selection will hence keep the ones with the good mutation parameters. This is what has often been stated as *"mutation parameters are optimised for free"* by the evolution itself. And indeed, SA-ES have long been the state-of-the-art in parametric optimisation [9].

But what are "good" mutation parameters? The issue has already been discussed for the step-size in previous section, and similar arguments can be given for the covariance matrix itself. Replace the sphere model ($\min \sum x_i^2 \equiv X^t X$) with a *quadratic* function[5] ($\min \frac{1}{2} X^t H X$ for some positive definite matrix H). Then it is clear that the mutation should progress slower along the directions of steepest descent of H: the covariance matrix should be proportional to H_{-1}. And whereas the step-size actually self-adapts to quasi-optimal values [9, 21], the covariance matrix that is learned by the *correlated* SA-ES is not the actual inverse of the Hessian [4].

3.4 CMA-ES: a clever adaptation

Another defect of SA-ES is the relative slowness of adaptation of the mutation parameters: even for the simple case of the step-size, if the initial value is not the optimal one (proportional to the distance to the optimum in the case of the sphere function), it takes some time to reach that optimal value and to start being efficient.

This observation led Hansen and Ostermeier to propose deterministic schedules to adapt the mutation parameters in ES, hence heading back to an **adaptive** method for parameter tuning. Their method was first limited to the step-size [39], and later addressed the adaptation of the full covariance matrix [37]. The complete *Covariance Matrix Adaptation* (CMA-ES) algorithm was finally detailed (and its parameters carefully tuned) in [38] and an improvement for the update of the covariance matrix was proposed in [36].

The basic idea in CMA-ES is to use the path followed by the algorithm to deterministically update the different mutation parameters, and a simplified view is given by the following: suppose that the algorithm has made a series of steps in co-linear directions; then the step-size should be increased, to allow larger steps and increase speed. Similar ideas undermine the covariance matrix update(s).

Indeed, using such clever learning method, CMA-ES proved to outperform most other stochastic algorithms for parametric optimisation, as witnessed by its success in the 2005 contest that took place at CEC'2005.

[5] The Evolutionary Computation community usually calls this type of function *elliptic* – as iso-fitness surfaces are ellipsoids. However, elliptic functions have a completely different meaning in Mathematics, and we will hence come back here to the more standard terminology, using the word *quadratic*.

3.5 Lessons learnt

This brief summary of ES history witnesses that

- Static parameters are not only hard but can be impossible to tune: there doesn't exist any good static value for the step-size in Gaussian mutation
- Adaptive methods use some information about the current state of the search, and are as good as the information they get: the *success rate* is a very raw information, and lead to the "easy-to-defeat" one-fifth rule, while CMA-ES uses high-level information to cleverly update all the parameters of the most general Gaussian mutation
- Self-adaptive methods are efficient methods when applicable, i.e. when the only available selection (based on the fitness) can prevent bad parameters from proceeding to future generations. They outperform basic static and adaptive methods, but are outperformed by clever adaptive methods.

4 Case Study: Changing the Penalty Coefficients

Let us assume we deal with a numerical optimisation problem to minimise

$$f(\overline{x}) = f(x_1, \ldots, x_n),$$

subject to some inequality and equality constraints

$$g_i(\overline{x}) \leq 0, i = 1, \ldots, q,$$

and

$$h_j(\overline{x}) = 0, j = q + 1, \ldots, m,$$

where the domains of the variables are given by lower and upper bounds $l_i \leq x_i \leq u_i$ for $1 \leq i \leq n$.

For such a numerical optimisation problem we may consider an evolutionary algorithm based on a floating-point representation, where each individual $\overline{x}$ in the population is represented as a vector of floating-point numbers $\overline{x} = \langle x_1, \ldots, x_n \rangle$.

In the previous section we described different ways to modify a parameter controlling mutation. Several other components of an EA have natural parameters, and these parameters are traditionally tuned in one or another way. Here we show that other components, such as the evaluation function (and consequently the fitness function) can also be parameterised and thus varied. While this is a less common option than tuning mutation (although it is practised in the evolution of variable-length structures for parsimony pressure [91]), it may provide a useful mechanism for increasing the performance of an evolutionary algorithm.

When dealing with constrained optimisation problems, penalty functions are often used. A common technique is the method of static penalties [63],

which requires fixed user-supplied penalty parameters. The main reason for its widespread use is that it is the simplest technique to implement: it requires only the straightforward modification of the evaluation function as follows:

$$eval(\overline{x}) = f(\overline{x}) + W \cdot penalty(\overline{x}),$$

where f is the objective function, and $penalty(\overline{x})$ is zero if no violation occurs, and is positive,[6] otherwise. Usually, the *penalty* function is based on the distance of a solution from the feasible region, or on the effort to "repair" the solution, i.e., to force it into the feasible region. In many methods a set of functions f_j $(1 \leq j \leq m)$ is used to construct the penalty, where the function f_j measures the violation of the jth constraint in the following way:

$$f_j(\overline{x}) = \begin{cases} \max\{0, g_j(\overline{x})\} & \text{if} \quad 1 \leq j \leq q, \\ |h_j(\overline{x})| & \text{if} \quad q+1 \leq j \leq m. \end{cases} \tag{1}$$

W is a user-defined weight, prescribing how severely constraint violations are weighted. In the most traditional penalty approach the weight W does not change during the evolution process. We sketch three possible methods of changing the value of W.

First, we can replace the static parameter W by a dynamic parameter, e.g., a function $W(t)$. Just as for the mutation parameter σ, we can develop a heuristic that modifies the weight W over time. For example, in the method proposed by Joines and Houck [47], the individuals are evaluated (at the iteration t) by a formula, where

$$eval(\overline{x}) = f(\overline{x}) + (C \cdot t)^{\alpha} \cdot penalty(\overline{x}),$$

where C and α are constants. Since

$$W(t) = (C \cdot t)^{\alpha},$$

the penalty pressure grows with the evolution time provided $1 \leq C, \alpha$.

Second, let us consider another option, which utilises feedback from the search process. One example of such an approach was developed by Bean and Hadj-Alouane [15], where each individual is evaluated by the same formula as before, but $W(t)$ is updated in every generation t in the following way:

$$W(t+1) = \begin{cases} (1/\beta_1) \cdot W(t) & \text{if } \overline{b}^i \in \mathcal{F} \quad \text{for all } t-k+1 \leq i \leq t, \\ \beta_2 \cdot W(t) & \text{if } \overline{b}^i \in \mathcal{S} - \mathcal{F} \text{ for all } t-k+1 \leq i \leq t, \\ W(t) & \text{otherwise.} \end{cases}$$

In this formula, $\mathcal{S}$ is the set of all search points (solutions), $\mathcal{F} \subseteq \mathcal{S}$ is a set of all *feasible* solutions, $\overline{b}^i$ denotes the best individual in terms of the function *eval* in generation i, $\beta_1, \beta_2 > 1$, and $\beta_1 \neq \beta_2$ (to avoid cycling). In other words,

[6] For minimisation problems.

the method decreases the penalty component $W(t+1)$ for the generation $t+1$ if all best individuals in the last k generations were feasible (i.e., in $\mathcal{F}$), and increases penalties if all best individuals in the last k generations were infeasible. If there are some feasible and infeasible individuals as best individuals in the last k generations, $W(t+1)$ remains without change.

Third, we could allow self-adaptation of the weight parameter, similarly to the mutation step sizes in the previous section. For example, it is possible to extend the representation of individuals into

$$\langle x_1, \ldots, x_n, W \rangle,$$

where W is the weight. The weight component W undergoes the same changes as any other variable x_i (e.g., Gaussian mutation and arithmetic recombination).

To illustrate this method,which is analogous to using a separate σ_i for each x_i, we need to redefine the evaluation function. Let us first introduce penalty functions for each constraint as per Eq. (1). Clearly, these penalties are all non-negative and are at zero if no constraints are violated. Then consider a vector of weights $\overline{w} = (w_1, \ldots, w_m)$, and define

$$eval(\overline{x}) = f(\overline{x}) + \sum_{j=1}^{m} w_j f_j(\overline{x}),$$

as the function to be minimised and also extend the representation of individuals into

$$\langle x_1, \ldots, x_n, w_1, \ldots, w_m \rangle.$$

Variation operators can then be applied to both the $\overline{x}$ and the $\overline{w}$ part of these chromosomes, realising a self-adaptation of the constraint weights, and thereby the fitness function.

It is important to note the crucial difference between self-adapting mutation step sizes and constraint weights. Even if the mutation step sizes are encoded in the chromosomes, the evaluation of a chromosome is *independent* from the actual σ values. That is,

$$eval(\langle \overline{x}, \overline{\sigma} \rangle) = f(\overline{x}),$$

for any chromosome $\langle \overline{x}, \overline{\sigma} \rangle$. In contrast, if constraint weights are encoded in the chromosomes, then we have

$$eval(\langle \overline{x}, \overline{w} \rangle) = f_{\overline{w}}(\overline{x}),$$

for any chromosome $\langle \overline{x}, W \rangle$. This could enable the evolution to "cheat" in the sense of making improvements by minimising the weights instead of optimising f and satisfying the constraints. Eiben et al. investigated this issue in [24] and found that using a specific tournament selection mechanism neatly solves this problem and enables the EA to solve constraints.

4.1 Summary

In the previous sections we illustrated how the mutation operator and the evaluation function can be controlled (adapted) during the evolutionary process. The latter case demonstrates that not only can the traditionally adjusted components, such as mutation, recombination, selection, etc., be controlled by parameters, but so can other components of an evolutionary algorithm. Obviously, there are many components and parameters that can be changed and tuned for optimal algorithm performance. In general, the three options we sketched for the mutation operator and the evaluation function are valid for any parameter of an evolutionary algorithm, whether it is population size, mutation step, the penalty coefficient, selection pressure, and so forth.

The mutation example above also illustrates the phenomenon of the **scope** of a parameter. Namely, the mutation step size parameter can have different domains of influence, which we call scope. Using the $\langle x_1, \ldots, x_n, \sigma_1, \ldots, \sigma_n \rangle$ model, a particular mutation step size applies only to one variable of a single individual. Thus, the parameter σ_i acts on a subindividual, or component, level. In the $\langle x_1, \ldots, x_n, \sigma \rangle$ representation, the scope of σ is one individual, whereas the dynamic parameter $\sigma(t)$ was defined to affect all individuals and thus has the whole population as its scope.

These remarks conclude the introductory examples of this section. We are now ready to attempt a classification of parameter control techniques for parameters of an evolutionary algorithm.

5 Classification of Control Techniques

In classifying parameter control techniques of an evolutionary algorithm, many aspects can be taken into account. For example:

1. *What* is changed? (e.g., representation, evaluation function, operators, selection process, mutation rate, population size, and so on)
2. *How* the change is made (i.e., deterministic heuristic, feedback-based heuristic, or self-adaptive)
3. *The evidence* upon which the change is carried out (e.g., monitoring performance of operators, diversity of the population, and so on)
4. *The scope/level* of change (e.g., population-level, individual-level, and so forth).

In the following we discuss these items in more detail.

5.1 *What* is Changed?

To classify parameter control techniques from the perspective of what component or parameter is changed, it is necessary to agree on a list of all major components of an evolutionary algorithm, which is a difficult task in itself. For that purpose, let us assume the following components of an EA:

- Representation of individuals
- Evaluation function
- Variation operators and their probabilities
- Selection operator (parent selection or mating selection)
- Replacement operator (survival selection or environmental selection)
- Population (size, topology, etc.)

Note that each component can be parameterised, and that the number of parameters is not clearly defined. For example, an offspring $\overline{v}$ produced by an arithmetical crossover of k parents $\overline{x}_1, \ldots, \overline{x}_k$ can be defined by the following formula:

$$\overline{v} = a_1\overline{x}_1 + \ldots + a_k\overline{x}_k,$$

where $a_1, \ldots, a_k$, and k can be considered as parameters of this crossover. Parameters for a population can include the number and sizes of subpopulations, migration rates, and so on for a general case, when more then one population is involved. Despite the somewhat arbitrary character of this list of components and of the list of parameters of each component, we will maintain the "what-aspect" as one of the main classification features, since this allows us to locate where a specific mechanism has its effect.

5.2 *How* are Changes Made?

As discussed and illustrated in the two earlier case studies, methods for changing the value of a parameter (i.e., the "how-aspect") can be classified into one of three categories.

- **Deterministic parameter control**
  This takes place when the value of a strategy parameter is altered by some deterministic rule. This rule modifies the strategy parameter in a fixed, predetermined (i.e., user-specified) way without using any feedback from the search. Usually, a time-varying schedule is used, i.e., the rule is used when a set number of generations have elapsed since the last time the rule was activated.
- **Adaptive parameter control**
  This takes place when there is some form of feedback from the search that serves as inputs to a mechanism used to determine the direction or magnitude of the change to the strategy parameter. The assignment of the value of the strategy parameter may involve credit assignment, based on the quality of solutions discovered by different operators/parameters, so that the updating mechanism can distinguish between the merits of competing strategies. Although the subsequent action of the EA may determine whether or not the new value persists or propagates throughout the population, the important point to note is that the updating mechanism used to control parameter values is externally supplied, rather than being part of the "standard" evolutionary cycle.

- **Self-adaptive parameter control**
  The idea of the evolution of evolution can be used to implement the self-adaptation of parameters (see [10] for a good review). Here the parameters to be adapted are encoded into the chromosomes and undergo mutation and recombination. The better values of these encoded parameters lead to better individuals, which in turn are more likely to survive and produce offspring and hence propagate these better parameter values. This is an important distinction between adaptive and self-adaptive schemes: in the latter the mechanisms for the credit assignment and updating of different strategy parameters are entirely implicit, i.e., they are the selection and variation operators of the evolutionary cycle itself.

  This terminology leads to the taxonomy illustrated in Fig. 1.

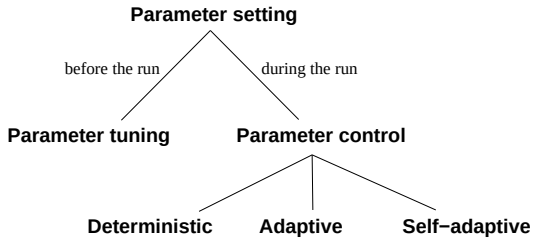

Fig. 1. Global taxonomy of parameter setting in EAs

Some authors have introduced a different terminology. Angeline [2] distinguished "absolute" and "empirical" rules, which correspond to the "uncoupled" and "tightly-coupled" mechanisms of Spears [80]. Let us note that the uncoupled/absolute category encompasses deterministic and adaptive control, whereas the tightly-coupled/empirical category corresponds to self-adaptation. We feel that the distinction between deterministic and adaptive parameter control is essential, as the first one does not use any feedback from the search process. However, we acknowledge that the terminology proposed here is not perfect either. The term "deterministic" control might not be the most appropriate, as it is not determinism that matters, but the fact that the parameter-altering transformations take no input variables related to the progress of the search process. For example, one might *randomly* change the mutation probability after every 100 generations, which is not a deterministic process. The name "fixed" parameter control might provide an alternative that also covers this latter example. Also, the terms "adaptive" and "self-adaptive" could be replaced by the equally meaningful "explicitly adaptive" and "implicitly adaptive" controls, respectively. We have chosen to use "adaptive" and "self-adaptive" for the widely accepted usage of the latter term.

5.3 What *Evidence* Informs the Change?

The third criterion for classification concerns the evidence used for determining the change of parameter value [71, 78]. Most commonly, the progress of the search is monitored, e.g., by looking at the performance of operators, the diversity of the population, and so on. The information gathered by such a monitoring process is used as feedback for adjusting the parameters. From this perspective, we can make further distinction between the following two cases:

- **Absolute evidence**
  We speak of absolute evidence when the value of a strategy parameter is altered by some rule that is applied when a predefined event occurs. The difference from deterministic parameter control lies in the fact that in deterministic parameter control a rule fires by a deterministic trigger (e.g., time elapsed), whereas here feedback from the search is used. For instance, the rule can be applied when the measure being monitored hits a previously set threshold – this is the event that forms the evidence. Examples of this type of parameter adjustment include increasing the mutation rate when the population diversity drops under a given value [56], changing the probability of applying mutation or crossover according to a fuzzy rule set using a variety of population statistics [54], and methods for resizing populations based on estimates of schemata fitness and variance [79]. Such mechanisms require that the user has a clear intuition about how to steer the given parameter into a certain direction in cases that can be specified in advance (e.g., they determine the threshold values for triggering rule activation). This intuition may be based on the encapsulation of practical experience, data-mining and empirical analysis of previous runs, or theoretical considerations (in the order of the three examples above), but all rely on the implicit assumption that changes that were appropriate to make on *another* search of *another* problem are applicable to *this* run of the EA on *this* problem.
- **Relative evidence**
  In the case of using relative evidence, parameter values are compared according to the fitness of the offspring that they produce, and the better values get rewarded. The direction and/or magnitude of the change of the strategy parameter is not specified deterministically, but relative to the performance of other values, i.e., it is necessary to have more than one value present at any given time. Here, the assignment of the value of the strategy parameter involves credit assignment, and the action of the EA may determine whether or not the new value persists or propagates throughout the population. As an example, consider an EA using more crossovers with crossover rates adding up to 1.0 and being reset based on the crossovers performance measured by the quality of offspring they create. Such methods may be controlled adaptively, typically using "bookkeeping" to monitor performance and a user-supplied update procedure

[17, 48, 68], or self-adaptively [7, 30, 53, 69, 75, 80] with the selection operator acting indirectly on operator or parameter frequencies via their association with "fit" solutions.

5.4 What is the *Scope* of the Change?

As discussed earlier, any change within any component of an EA may affect a gene (parameter), whole chromosomes (individuals), the entire population, another component (e.g., selection), or even the evaluation function. This is the aspect of the scope or level of adaptation [2, 41, 78]. Note, however, that the scope or level is not an independent dimension, as it usually depends on the component of the EA where the change takes place. For example, a change of the mutation step size may affect a gene, a chromosome, or the entire population, depending on the particular implementation (i.e., scheme used), but a change in the penalty coefficients typically affects the whole population. In this respect the scope feature is a secondary one, usually depending on the given component and its actual implementation.

It should be noted that the issue of the scope of the parameter might be more complicated than indicated in Sect. 4.1. First of all, the scope depends on the interpretation mechanism of the given parameters. For example, in a non-isotropic SA-ES (see section 3.3), an individual might be represented as

$$\langle x_1, \ldots, x_n, \sigma_1, \ldots, \sigma_n, \alpha_1, \ldots, \alpha_{n(n-1)/2} \rangle,$$

where the vector $\overline{\alpha}$ denotes the covariances between the variables $x_1, \ldots, x_n$. In this case the scope of the strategy parameters in $\overline{\alpha}$ is the whole individual, although the notation might suggest that they act on a sub-individual level.

The next example illustrates that the same parameter (encoded in the chromosomes) can be interpreted in different ways, leading to different algorithm variants with different scopes of this parameter. Spears [80], following [31], experimented with individuals containing an extra bit to determine whether one-point crossover or uniform crossover is to be used (bit 1/0 standing for one-point/uniform crossover, respectively). Two interpretations were considered. The first interpretation was based on a pairwise operator choice: If both parental bits are the same, the corresponding operator is used; otherwise, a random choice is made. Thus, this parameter in this interpretation acts at an individual level. The second interpretation was based on the bit distribution over the whole population: If, for example, 73% of the population had bit 1, then the probability of one-point crossover was 0.73. Thus this parameter under this interpretation acts on the population level. Spears noted that there was a definite impact on performance, with better results arising from the individual level scheme, and more recently Smith [73] compared three versions of a self-adaptive recombination operator, concluding that the component-level version significantly outperformed the individual or population-level versions.

However, the two interpretations of Spears' scheme can be easily combined. For instance, similar to the first interpretation, if both parental bits are the

same, the corresponding operator is used, but if they differ, the operator is selected according to the bit distribution, just as in the second interpretation. The scope/level of this parameter in this interpretation is neither individual nor population, but rather both. This example shows that the notion of scope can be ill-defined and very complex. This, combined with the arguments that the scope or level entity is primarily a feature of the given parameter and only secondarily a feature of adaptation itself, motivates our decision to exclude it as a major classification criterion.

5.5 Summary

In conclusion, the main criteria for classifying methods that change the values of the strategy parameters of an algorithm during its execution are:

1. What component/parameter is changed?
2. How is the change made?
3. Which evidence is used to make the change?

Our classification is thus three-dimensional. The *component* dimension consists of six categories: representation, evaluation function, variation operators (mutation and recombination), selection, replacement, and population. The other dimensions have respectively three (deterministic, adaptive, self-adaptive) and two categories (absolute, relative). Their possible combinations are given in Table 1. As the table indicates, deterministic parameter control with relative evidence is impossible by definition, and so is self-adaptive parameter control with absolute evidence. Within the adaptive scheme both options are possible and are indeed used in practise.

	Deterministic	Adaptive	Self-adaptive
Absolute	+	+	–
Relative	–	+	+

Table 1. Refined taxonomy of parameter setting in EAs: types of parameter control along the type and evidence dimensions. The – entries represent meaningless (nonexistent) combinations

6 Examples of Varying EA Parameters

Here we review some illustrative examples from the literature concerning all major components. For a more comprehensive overview the reader is referred to [23].

6.1 Representation

The choice of representation forms an important distinguishing feature between different streams of evolutionary computing. From this perspective GAs and ES can be distinguished from (historical) EP and GP according to the data structure used to represent individuals. In the first group this data structure is linear, and its length is fixed, that is, it does not change during a run of the algorithm. For (historical) EP and GP this does not hold: finite state machines and parse trees are nonlinear structures, and their size (the number of states, respectively nodes) and shape can change during a run. It could be argued that this implies an intrinsically adaptive representation in traditional EP and GP. On the other hand, the main structure of the finite state machines does not change during the search in traditional EP, nor do the function and terminal sets in GP (without automatically defined functions, ADFs). If one identifies "representation" with the basic syntax (plus the encoding mechanism), then the differently sized and shaped finite state machines, respectively trees, are only different expressions in this unchanging syntax. Based on this view we do not consider the representations in traditional EP and GP intrinsically adaptive.

We illustrate variable representations with the delta coding algorithm of Mathias and Whitley [89], which effectively modifies the encoding of the function parameters. The motivation behind this algorithm is to maintain a good balance between fast search and sustaining diversity. In our taxonomy it can be categorised as an adaptive adjustment of the representation based on absolute evidence.

The GA is used with multiple restarts; the first run is used to find an *interim solution*, and subsequent runs decode the genes as distances (*delta values*) from the last interim solution. This way each restart forms a new hypercube with the interim solution at its origin. The resolution of the delta values can also be altered at the restarts to expand or contract the search space. The restarts are triggered when population diversity (measured by the Hamming distance between the best and worst strings of the current population) is not greater than one. The sketch of the algorithm showing the main idea is given in Fig. 2.

Note that the number of bits for δ can be increased if the same solution `INTERIM` is found. This technique was further refined in [60, 61] to cope with deceptive problems.

Another similar idea to refine the grain of the representation for real numbers was also proposed in [64]: the *Real-Coded Adaptive Range Genetic Algorithm* gradually refines the range of the variables around the best-so-far values.

6.2 Evaluation function

Evaluation functions are typically not varied in an EA because they are often considered as part of the problem to be solved and not as part of the problem-

```
BEGIN
  /* given a starting population and genotype-phenotype encoding */
  WHILE (  HD > 1 ) DO
    RUN_GA with k bits per object variable;
  OD
  REPEAT UNTIL (  global termination is satisfied ) DO
    save best solution as INTERIM;
    reinitialise population with new coding;
    /*  k-1 bits as the distance δ to the object value in  */
    /*  INTERIM and one sign bit */
    WHILE (  HD > 1 ) DO
      RUN_GA with this encoding;
    OD
  OD
END
```

Fig. 2. Outline of the delta coding algorithm

solving algorithm. In fact, an evaluation function forms the bridge between the two, so both views are at least partially true. In many EAs the evaluation function is derived from the (optimisation) problem at hand with a simple transformation of the objective function. In the class of constraint satisfaction problems, however, there is no objective function in the problem definition [22]. Rather, these are normally posed as decision problems with an Boolean outcome ϕ denoting whether a given assignment of variables represents a valid solution. One possible approach using EAs is to treat these as minimisation problems where the evaluation function is defined as the amount of constraint violation by a given candidate solution. This approach, commonly known as the penalty approach, can be formalised as follows. Let us assume that we have constraints c_i ($i = \{1, \ldots, m\}$) and variables v_j ($j = \{1, \ldots, n\}$) with the same domain S. The task is to find one variable assignment $\bar{s} \in S$ satisfying all constraints. Then the penalties can be defined as follows:

$$f(\bar{s}) = \sum_{i=1}^{m} w_i \times \chi(\bar{s}, c_i),$$

where

$$\chi(\bar{s}, c_i) = \begin{cases} 1 \text{ if } \bar{s} \text{ violates } c_i, \\ 0 \text{ otherwise.} \end{cases}$$

Obviously, for each $\bar{s} \in S$ we have that $\phi(\bar{s}) = true$ if and only if $f(\bar{s}) = 0$, and the weights specify how severely the violation of a certain constraint is

penalised. The setting of these weights has a large impact on the EA performance, and ideally w_i should reflect how hard c_i is to satisfy. The problem is that finding the appropriate weights requires much insight into the given problem instance, and therefore it might not be practicable.

The stepwise adaptation of weights (SAW) mechanism, introduced by Eiben and van der Hauw [28] as an improved version of the weight adaptation mechanism of Eiben, Raué, and Ruttkay [25, 26], provides a simple and effective way to set these weights. The basic idea behind the SAW mechanism is that constraints that are not satisfied after a certain number of steps (fitness evaluations) must be difficult, and thus must be given a high weight (penalty). SAW-ing changes the evaluation function adaptively in an EA by periodically checking the best individual in the population and raising the weights of those constraints this individual violates. Then the run continues with the new evaluation function. A nice feature of SAW-ing is that it liberates the user from seeking good weight settings, thereby eliminating a possible source of error. Furthermore, the used weights reflect the difficulty of constraints for *the given algorithm* on the *given problem instance* in *the given stage of the search* [29]. This property is also valuable since, in principle, different weights could be appropriate for different algorithms.

6.3 Mutation

A large majority of work on adapting or self-adapting EA parameters concerns variation operators: mutation and recombination (crossover). As we discussed in section 3, the 1/5 rule of Rechenberg constitutes a first historical example for adaptive mutation step size control. It was followed by the traditional self-adaptive control of mutation step sizes, and more recently by the adaptive CMA method.

Hesser and Männer [40] derived theoretically optimal schedules within GAs for deterministically changing p_m for the counting-ones function. They suggest:

$$p_m(t) = \sqrt{\frac{\alpha}{\beta}} \times \frac{\exp\left(\frac{-\gamma t}{2}\right)}{\lambda\sqrt{L}},$$

where α, β, γ are constants, L is the chromosome length, λ is the population size, and t is the time (generation counter). This is a purely deterministic parameter control mechanism.

A self-adaptive mechanism for controlling mutation in a bit-string GA is given by Bäck [6]. This technique works by extending the chromosomes by an additional 20 bits that together encode the individuals' own p_m. Mutation then works by:

1. Decoding these bits first to p_m
2. Mutating the bits that encode p_m with mutation probability p_m
3. Decoding these (changed) bits to p'_m
4. Mutating the bits that encode the solution with mutation probability p'_m

This approach is highly self-adaptive since even the rate of variation of the search parameters is given by the encoded value, as opposed to the use of an external parameters such as learning rates for step-sizes. More recently Smith [74] showed theoretical predictions, verified experimentally, that this scheme gets "stuck" in suboptimal regions of the search space with a low, or zero, mutation rate attached to each member of the population. He showed that a more robust problem-solving mechanism can simply be achieved by ignoring the first step of the algorithm above, and instead using a fixed learning rate as the probability of applying bitwise mutation to the encoding of the strategy parameters in the second step.

6.4 Crossover

The classical example for adapting crossover rates in GAs is Davis's adaptive operator fitness. The method adapts the rates of crossover operators by rewarding those that are successful in creating better offspring. This reward is diminishingly propagated back to operators of a few generations back, who helped setting it all up; the reward is a shift up in probability at the cost of other operators [18]. This, actually, is very close in spirit to the "implicit bucket brigade" credit assignment principle used in classifier systems [34].

The GA using this method applies several crossover operators simultaneously within the same generation, each having its own crossover rate $p_c(op_i)$. Additionally, each operator has its "local delta" value d_i that represents the strength of the operator measured by the advantage of a child created by using that operator with respect to the best individual in the population. The local deltas are updated after every use of operator i. The adaptation mechanism recalculates the crossover rates after K generations. The main idea is to redistribute 15% of the probabilities biased by the accumulated operator strengths, that is, the local deltas. To this end, these d_i values are normalised so that their sum equals 15, yielding d_i^{norm} for each i. Then the new value for each $p_c(op_i)$ is 85% of its old value and its normalised strength:

$$p_c(op_i) = 0.85 \cdot p_c(op_i) + d_i^{norm}.$$

Clearly, this method is adaptive based on relative evidence.

6.5 Selection

It is interesting to note that neither the parent selection nor the survivor selection (replacement) component of an EA has been commonly used in an adaptive manner, even though there are selection methods whose parameters can be easily adapted. For example, in linear ranking there is a parameter s representing the expected number of offspring to be allocated to the best individual. By changing this parameter within the range of $[1 \ldots 2]$ the selective pressure of the algorithm can be varied easily. Similar possibilities exist for

tournament selection, where the tournament size provides a natural parameter.

Most existing mechanisms for varying the selection pressure are based on the so-called **Boltzmann** selection mechanism, which changes the selection pressure during evolution according to a predefined "cooling schedule" [58]. The name originates from the Boltzmann trial from condensed matter physics, where a minimal energy level is sought by state transitions. Being in a state i the chance of accepting state j is

$$P[\text{accept } j] = \begin{cases} 1 & \text{if} \quad E_i \geq E_j, \\ \exp\left(\frac{E_i - E_j}{K_b \cdot T}\right) & \text{if} \quad E_i < E_j, \end{cases}$$

where E_i, E_j are the energy levels, K_b is a parameter called the Boltzmann constant, and T is the temperature. This acceptance rule is called the Metropolis criterion.

We illustrate variable selection pressure in the survivor selection (replacement) step by **simulated annealing** (SA). SA is a generate-and-test search technique based on a physical, rather than a biological analogy [1]. Formally, however, SA can be envisioned as an evolutionary process with population size of 1, undefined (problem-dependent) representation and mutation, and a specific survivor selection mechanism. The selective pressure changes during the course of the algorithm in the Boltzmann style. The main cycle in SA is given in Fig. 3.

```
BEGIN
  /* given a current solution i ∈ S */
  /* given a function to generate the set of neighbours N_i of i */
  generate j ∈ N_i;
  IF (f(i) < f(j)) THEN
    set i = j;
    ELSE
      IF ( exp((f(i)−f(j))/c_k) > random[0,1)) THEN
        set i = j;
      FI
    ESLE
  FI
END
```

Fig. 3. Outline of the simulated annealing algorithm

In this mechanism the parameter c_k, the temperature, decreases according to a predefined scheme as a function of time, making the probability of

accepting inferior solutions smaller and smaller (for minimisation problems). From an evolutionary point of view, we have here a (1+1) EA with increasing selection pressure.

A successful example of applying Boltzmann acceptance is that of Smith and Krasnogor [52], who used it in the local search part of a memetic algorithm (MA), with the temperature inversely related to the fitness diversity of the population. If the population contains a wide spread of fitness values, the "temperature" is low, so only fitter solutions found by local search are likely to be accepted, concentrating the search on good solutions. However, when the spread of fitness values is low, indicating a converged population which is a common problem in MAs, the "temperature" is higher, making it more likely that an inferior solution will be accepted, thus reintroducing diversity and offering a potential means of escaping from local optima.

6.6 Population

An innovative way to control the population size is offered by Arabas et al. [3, 62] in their GA with variable population size (GAVaPS). In fact, the population size parameter is removed entirely from GAVaPS, rather than adjusted on-the-fly. Certainly, in an evolutionary algorithm the population always has a size, but in GAVaPS this size is a derived measure, not a controllable parameter. The main idea is to assign a lifetime to each individual when it is created, and then to reduce its remaining lifetime by one in each consecutive generation. When the remaining lifetime becomes zero, the individual is removed from the population. Two things must be noted here. First, the lifetime allocated to a newborn individual is biased by its fitness: fitter individuals are allowed to live longer. Second, the expected number of offspring of an individual is proportional to the number of generations it survives. Consequently, the resulting system favours the propagation of good genes.

Fitting this algorithm into our general classification scheme is not straightforward because it has no explicit mechanism that sets the value of the population size parameter. However, the procedure that implicitly determines how many individuals are alive works in an adaptive fashion using information about the status of the search. In particular, the fitness of a newborn individual is related to the fitness of the present generation, and its lifetime is allocated accordingly. This amounts to using relative evidence.

6.7 Varying Several Parameters Simultaneously

One of the studies explicitly devoted to adjusting more parameters (and also on more than one level) is that of Hinterding et al. on a "self-adaptive GA" [42]. This GA uses self-adaptation for mutation rate control, plus relative-based adaptive control for the population size.[7] The mechanism for controlling

[7] Strictly speaking, the authors' term "self-adaptive GA" is only partially correct. However, this paper is from 1996, and the contemporary terminology distinguishing

mutation is similar to that from Bäck [6], (Sect. 6.3), except that mutating the bits encoding the mutation strength is not based on the bits in question, but is done by a universal mechanism fixed for all individuals and all generations. In other words, the self-adaptive mutation parameter is only used for the genes encoding a solution. As for the population size, the GA works with three subpopulations: a small, a medium, and a large one, P1, P2, and P3, respectively (the initial sizes respectively being 50, 100, and 200). These populations are evolved in parallel for a given number of fitness evaluations (an epoch) independently by the same GA setup. After each epoch, the subpopulations are resized based on some heuristic rules, maintaining a lower and an upper bound (10 and 1000) and keeping P2 always the medium-sized subpopulation. There are two categories of rules. Rules in the first category are activated when the fitnesses in the subpopulations converge and try to move the populations apart. For instance, if P2 and P3 have the same fitness, the size of P3 is doubled. Rules from another set are activated when the fitness values are distinct at the end of an epoch. These rules aim at maximising the performance of P2. An example of one such rule is: if the performance of the subpopulations ranks them as P2 < P3 < P1 then size(P3) = (size(P2) + size(P3))/2. In our taxonomy, this population size control mechanism is adaptive, based on relative evidence.

Lis and Lis [57] also offer a parallel GA setup to control the mutation rate, the crossover rate, and the population size during a run. The idea here is that for each parameter a few possible values are defined in advance, say *lo, med, hi*, and only these values are allowed in any of the GAs, that is, in the subpopulations evolved in parallel. After each epoch the performances of the applied parameter values are compared by averaging the fitnesses of the best individuals of those GAs that use a given value. If the winning parameter value is:

1. *hi*, then all GAs shift one level up concerning this parameter in the next epoch;
2. *med*, then all GAs use the same value concerning this parameter in the next epoch;
3. *lo*, then all GAs shift one level down concerning this parameter in the next epoch.

Clearly, the adjustment mechanism for all parameters here is adaptive, based on relative evidence.

Mutation, crossover, and population size are all controlled on-the-fly in the GA "without parameters" of Bäck et al. in [11]. Here, the self-adaptive mutation from [6] (Sect. 6.3) is adopted without changes, a new self-adaptive technique is invented for regulating the crossover rates of the individuals, and the GAVaPS lifetime idea (Sect. 6.6) is adjusted for a steady-state GA

dynamic, adaptive, and self-adaptive schemes as we do it here was only published in 1999 [23].

model. The crossover rates are included in the chromosomes, much like the mutation rates. If a pair of individuals is selected for reproduction, then their individual crossover rates are compared with a random number $r \in [0, 1]$ and an individual is seen as ready to mate if its $p_c > r$. Then there are three possibilities:

1. If both individuals are ready to mate then uniform crossover is applied, and the resulting offspring is mutated.
2. If neither is ready to mate then both create a child by mutation only.
3. If exactly one of them is ready to mate, then the one not ready creates a child by mutation only (which is inserted into the population immediately through the steady-state replacement), the other is put on the hold, and the next parent selection round picks only one other parent.

This study differs from those discussed before in that it explicitly compares GA variants using only one of the (self-)adaptive mechanisms and the GA applying them all. The experiments show remarkable outcomes: the completely (self-)adaptive GA wins, closely followed by the one using only the adaptive population size control, and the GAs with self-adaptive mutation and crossover are significantly worse. These results suggest that putting effort into adapting the population size could be more effective than trying to adjust the variation operators. This is truly surprising considering that traditionally the on-line adjustment of the variation operators has been pursued and the adjustment of the population size received relatively little attention. The subject certainly requires more research.

7 Parameter Control in DyFor GP

Several studies have applied genetic programming (GP) to the task of forecasting with favourable results. However, these studies, like those applying other heuristic and statistical techniques, have assumed a static environment, making them unsuitable for many real-world time series which are generated by varying processes. The study of Wagner et al. [86] investigated the development of a new "dynamic" GP model (DyFor GP) that is specifically tailored for forecasting in non-static environments. Note that if the underlying data generating process shifts, the forecasting methods must be reevaluated in order to accommodate the new process. Additionally, these forecasting methods require that the number of historical time series data used for analysis be designated *shape a priori*. This presents a problem in non-static environments because different segments of the time series may have different underlying data generating processes.

Usually some degree of human judgement is necessary to assign the number of historical data to be used for analysis. If the time series is not well-understood, then the assignment may contain segments with disparate underlying processes. This situation highlights the need for forecasting methods that

can automatically determine the correct analysis "window" (i.e., the correct number of historical data to be analysed). Research reported in [86] attempted to develop a dynamic forecasting model (DyFor GP) that can do just that. Furthermore, this study explores methods that can retain knowledge learnt from previously encountered environments. Such past-learnt knowledge may prove useful when current environmental conditions resemble those of a prior setting. Specifically, this knowledge allows for faster convergence to current conditions by giving the model searching process a "head-start" (i.e., by narrowing the model search space).

The DyFor GP model were tested for forecasting efficacy on real-world economic time series, namely the U.S. Gross Domestic Product and Consumer Price Index Inflation. Results showed that the DyFor GP model outperformed benchmark models from leading studies for both experiments. These findings affirmed the DyFor GP's potential as an adaptive, non-linear model for real-world forecasting applications.

In addition to automatic time-horizon window adjustment, the DyFor GP also contains features that automatically adjust population size and diversity. The following subsections describe all these features.

7.1 Adapting the Analysis Window

As indicated earlier, designating the correct size for the analysis window is critical to the success of any forecasting model. Automatic discovery of this windowsize is indispensable when the forecasting concern is not well-understood. With each slide of the window, the DyFor GP adjusts its windowsize dynamically. This is accomplished in the following way.

1. Select two initial windowsizes, one of size n and one of size $n+i$ where n and i are positive integers, $i < n$.
2. Run dynamic generations at the beginning of the historical data with windowsizes n and $n+i$, use the best solution for each of these two independent runs to predict a number of future data points, and measure their predictive accuracy.
3. Select another two windowsizes based on which windowsize had better accuracy. For example if the smaller of the 2 windowsizes (size n) predicted more accurately, then choose 2 new windowsizes, one of size n and one of size $n-i$. If the larger of the 2 windowsizes (size $n+i$) predicted more accurately, then choose windowsizes $n+i$ and $n+2i$.
4. Slide the analysis window to include the next time series observation. Use the two selected windowsizes to run another two dynamic generations, predict future data, and measure their prediction accuracy.
5. Repeat the previous two steps until the the analysis window reaches the end of historical data.

Thus, at each slide of the analysis window, predictive accuracy is used to determine the direction in which to adjust the windowsize.

Consider the following example. Suppose the time series given in figure 4 is to be analysed and forecast. As depicted in the figure, this time series consists of two segments each with a different underlying data generating process. The

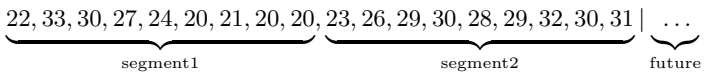

Fig. 4. Time series containing segments with differing underlying processes.

second segment's underlying process represents the current environment and is valid for forecasting future data. The first segment's process represents an older environment that no longer exists but may contain patterns that can be learnt and exploited when forecasting the current environment. If there is no knowledge available concerning these segments, automatic techniques are required to discover the correct windowsize needed to forecast the current setting. The DyFor GP starts by selecting two initial windowsizes, one larger than the other. Then, two separate dynamic generations are run at the beginning of the historical data, each with its own windowsize. After each dynamic generation, the best solution is used to predict some number of future data and the accuracy of this prediction is measured. Figure 5 illustrates these steps. In the figure **win1** and **win2** represent data analysis windows of size 3 and 4, respectively, and **pred** represents the future data predicted.

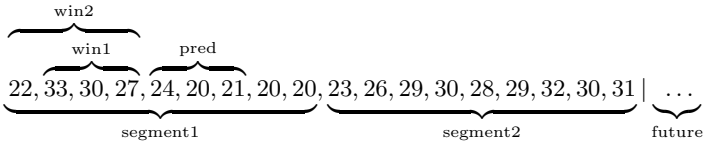

Fig. 5. Initial steps of window adaptation.

The data predicted in these initial steps lies inside the first segment's process and, because the dynamic generation involving analysis window **win2** makes use of a greater number of appropriate data than that of **win1**, it is likely that **win2**'s prediction accuracy is better. If this is true, two new windowsizes for **win1** and **win2** are selected with sizes of 4 and 5, respectively. The analysis window then slides to include the next time series value, two new dynamic generations are run, and the best solutions for each used to predict future data. Figure 6 depicts these steps. In the figure data analysis windows **win1** and **win2** now include the next time series value, 24, and **pred** has shifted one value to the right.

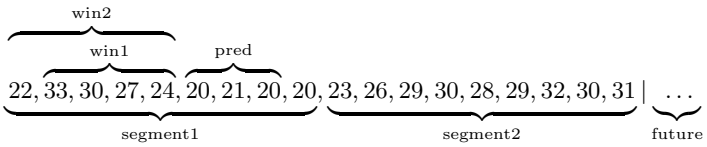

Fig. 6. Window adaptation after the first window slide. Note: **win1** and **win2** have size 4 and 5, respectively.

This process of selecting two new windowsizes, sliding the analysis window, running two new dynamic generations, and predicting future data is repeated until the analysis window reaches the end of historical data. It may be noted that while the prediction data, **pred**, lies entirely inside the first segment, the data analysis windows, **win1** and **win2**, are likely to expand to encompass a greater number of appropriate data. However, after several window slides, when the data analysis window spans data from both the first and second segments, it is likely that the window adjustment reverses direction.

The DyFor GP uses predictive accuracy to adapt the size of its analysis window automatically. When the underlying process is stable (i.e., the analysis window is contained inside a single segment), the windowsize is likely to expand. When the underlying process shifts (i.e., the analysis window spans more than one segment), the windowsize is likely to contract.

7.2 The Problem of Bloat

Bloat in GP is the tendency for solution trees to grow large as they approach the optimal. Solutions may become so large that they exhaust computer resources. Additionally, bloat hinders a GP model's adaptability as solutions become too specialised to adjust to changing conditions. Bloat is a problem for any GP model regardless of the application. In order to allow the DyFor GP to search efficiently for an appropriate forecasting model, bloat must be minimised without sacrificing the quality of solutions. Two methods are proposed to overcome this obstacle:

1. natural non-static population control and
2. dynamic increase of diversity.

GP models evolve a population of solutions for a given problem. Thus, a GP model must contain some method to control the number and size of solutions in any population. The standard method of GP population control is due to Koza [51] and uses a static population cardinality and a maximum tree depth for solutions.However, this method does not protect a GP model from bloat. If numerous solutions in a population have full or nearly full trees of depth close to the maximum, available resources may be exhausted. Additionally, the artificial limit for tree depth prohibits the search process

from exploring solutions of greater complexity, which, especially for many real-world problems, may be solutions of higher quality.

An alternative method for GP population control is presented to allow natural growth of complex solutions in a setting that more closely emulates the one found in nature. In nature the number of organisms in a population is not static. Instead, the population cardinality varies as fitter organisms occupy more available resources and weaker organisms make do with less. Thus, from generation to generation, the population cardinality changes depending on the quality and type of individual organisms present. The proposed natural non-static population control (NNPC) is based on a variable population cardinality with a limit on the total number of tree nodes present in a population and no limit for solution tree depth. This method addresses the following issues:

1. allowing natural growth of complex solutions of greater quality,
2. keeping resource consumption within some specified limit, and
3. allowing the population cardinality to vary naturally based on the make-up of individual solutions present.

By not limiting the tree depth of individual solutions, natural evolution of complex solutions is permitted. By restricting the total number of tree nodes in a population, available resources are conserved. Thus, for a GP model that employs NNPC, the number of solutions in a population grows or declines naturally as the individual solutions in the population vary. This method is described in more detail below.

NNPC works in the following way. Two node limits for a population are specified as parameters: the soft node limit and the hard node limit. The soft node limit is defined as the limit for adding new solutions to a population. This means that if adding a new solution to a population causes the total nodes present to exceed the soft node limit, then that solution is the last one added. The hard node limit is defined as the absolute limit for total nodes in a population. This means that if adding a new solution to a population causes the total nodes present to exceed the hard node limit, then that solution may be added only after it is repaired (the tree has been trimmed) so that the total nodes present no longer exceeds this limit. During the selection process of the DyFor GP, a count of the total nodes present in a population is maintained. Before adding a new solution to a population, a check is made to determine if adding the solution will increase the total nodes present beyond either of the specified limits.

Wagner and Michalewicz [85] provide a study comparing a GP forecasting model with NNPC to one with the standard population control (SPC) method introduced by Koza [51]. Observed results indicate that the model with NNPC was significantly more efficient in its consumption of computer resources than the model with SPC while the quality of forecasts produced by both models remained equivalent.

An important issue in GP is that of how diversity of GP populations can be achieved and maintained. Diversity refers to non-homogeneity of solutions in

a population [55]. A population that is spread out over the search space has a greater chance of finding an optimal solution than one that is concentrated in a small area of the search space. The significance of this concern is recognised in [14, 19, 87].

A method that dynamically increases the diversity of a DyFor GP population is proposed to accomplish these objectives. The dynamic increase of diversity (DIOD) method increases diversity by building a new population before each dynamic generation using evolved components from the previous dynamic generation. The following steps outline this procedure.

1. An initial population is constructed (using randomly generated trees in the usual way) for the first dynamic generation.
2. After the dynamic generation is completed, a new initial population is constructed for the next dynamic generation that consists of two types of solution trees:
   a) randomly generated solution trees and
   b) solution trees that are subtrees of fitter solutions from the last population of the previous dynamic generation.
3. The previous step is repeated after each successive dynamic generation.

Thus, each new dynamic generation after the first starts with a new initial population whose solution trees are smaller than those of the last population of the previous dynamic generation but have not lost the adaptations gained from past dynamic generations. In this way, solution tree bloat is reduced without harming the quality of solutions. Additionally, because randomly generated trees make up a portion of the new population, diversity is increased.

8 Discussion

Summarising this paper a number of things can be noted. First, parameter control in an EA can have two purposes. It can be done to avoid suboptimal algorithm performance resulting from suboptimal parameter values set by the user. The basic assumption here is that the applied control mechanisms are intelligent enough to do this job better than the user could, or that they can do it approximately as good, but they liberate the user from doing it. Either way, they are beneficial. The other motivation for controlling parameters on-the-fly is the assumption that the given parameter can have a different "optimal" value in different phases of the search. If this holds, then there is simply no optimal static parameter value; for good EA performance one must vary this parameter.

The second thing we want to note is that making a parameter (self-)adaptive does not necessarily mean that we have an EA with fewer parameters. For instance, in GAVaPS the population size parameter is eliminated at the cost of introducing two new ones: the minimum and maximum lifetime of

newborn individuals. If the EA performance is sensitive to these new parameters then such a parameter replacement can make things worse. This problem also occurs on another level. One could say that the procedure that allocates lifetimes in GAVaPS, the probability redistribution mechanism for adaptive crossover rates (Sect. 6.4), or the function specifying how the σ values are mutated in ES are also (meta) parameters. It is in fact an assumption that these are intelligently designed and their effect is positive. In many cases there are more possibilities, that is, possibly well-working procedures one can design. Comparing these possibilities implies experimental (or theoretical) studies very much like comparing different parameter values in a classical setting. Here again, it can be the case that algorithm performance is not so sensitive to details of this (meta) parameter, which fully justifies this approach.

Finally, let us place the issue of parameter control in a larger perspective. Over the last 20 years the EC community shifted from believing that EA performance is to a large extent independent from the given problem instance to realising that it is. In other words, it is now acknowledged that EAs need more or less fine-tuning to specific problems and problem instances. Ideally, it should be the algorithm that performs the necessary problem-specific adjustments. Parameter control as discussed here is a step towards this.

References

1. E.H.L. Aarts and J. Korst. *Simulated Annealing and Boltzmann Machines*. Wiley, Chichester, UK, 1989.
2. P.J. Angeline. Adaptive and self-adaptive evolutionary computations. In *Computational Intelligence*, pages 152–161. IEEE Press, 1995.
3. J. Arabas, Z. Michalewicz, and J. Mulawka. GAVaPS – a genetic algorithm with varying population size. In ICEC-94 [43], pages 73–78.
4. A. Auger. *Contributions thoriques et numriques l'optimisation continue par algorithmes volutionnaires*. PhD thesis, Universit Paris 6, December 2004. in French.
5. A. Auger, C. Le Bris, and M. Schoenauer. Dimension-independent convergence rate for non-isotropic $(1, \lambda) - es$. In E. Cantu-Paz et al., editor, *Proceedings of the Genetic and Evolutionary Conference 2003*, pages 512–524. LNCS 2723 and 2724, Springer Verlag, 2003.
6. T. Bäck. The interaction of mutation rate, selection and self-adaptation within a genetic algorithm. In Männer and Manderick [59], pages 85–94.
7. T. Bäck. Self adaptation in genetic algorithms. In F.J. Varela and P. Bourgine, editors, *Toward a Practice of Autonomous Systems: Proceedings of the 1st European Conference on Artificial Life*, pages 263–271. MIT Press, Cambridge, MA, 1992.
8. T. Bäck. Optimal mutation rates in genetic search. In Forrest [32], pages 2–8.
9. T. Bäck. *Evolutionary Algorithms in Theory and Practice*. New-York:Oxford University Press, 1995.
10. T. Bäck. Self-adaptation. In T. Bäck, D.B. Fogel, and Z. Michalewicz, editors, *Evolutionary Computation 2: Advanced Algorithms and Operators*, chapter 21, pages 188–211. Institute of Physics Publishing, Bristol, 2000.

11. T. Bäck, A.E. Eiben, and N.A.L. van der Vaart. An empirical study on GAs "without parameters". In M. Schoenauer, K. Deb, G. Rudolph, X. Yao, E. Lutton, J.J. Merelo, and H.-P. Schwefel, editors, *Proceedings of the 6th Conference on Parallel Problem Solving from Nature*, number 1917 in Lecture Notes in Computer Science, pages 315–324. Springer, Berlin, Heidelberg, New York, 2000.
12. T. Bäck, D.B. Fogel, and Z. Michalewicz, editors. *Handbook of Evolutionary Computation*. Institute of Physics Publishing, Bristol, and Oxford University Press, New York, 1997.
13. T. Bäck, M. Schütz, and S. Khuri. A comparative study of a penalty function, a repair heuristic and stochastic operators with set covering problem. In J.-M. Alliot, E. Lutton, E. Ronald, M. Schoenauer, and D. Snyers, editors, *Artificial Evolution*, number 1063 in LNCS. Springer Verlag, 1995.
14. W. Banzhaf. *Genetic Programming: An Introduction*. Morgan Kaufmann, 1998.
15. J.C. Bean and A.B. Hadj-Alouane. A dual genetic algorithm for bounded integer problems. Technical Report 92-53, University of Michigan, 1992.
16. R.K. Belew and L.B. Booker, editors. *Proceedings of the 4th International Conference on Genetic Algorithms*. Morgan Kaufmann, San Francisco, 1991.
17. L. Davis. Adapting operator probabilities in genetic algorithms. In Schaffer [65], pages 61–69.
18. L. Davis, editor. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, 1991.
19. E. de Jong, R. Watson, and J. Pollack. Reducing bloat and promoting diversity using multi-objective methods. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2001)*, pages 11–18, 2001.
20. K.A. De Jong. *An Analysis of the Behaviour of a Class of Genetic Adaptive Systems*. PhD thesis, University of Michigan, 1975.
21. K. Deb and H.-G. Beyer. Self-adaptive genetic algorithms with simulated binary crossover. *Evolutionary Computation*, 9(2):197 – 221, 2001.
22. A.E. Eiben. Evolutionary algorithms and constraint satisfaction: Definitions, survey, methodology, and research directions. In L. Kallel, B. Naudts, and A. Rogers, editors, *Theoretical Aspects of Evolutionary Computing*, pages 13–58. Springer, Berlin, Heidelberg, New York, 2001.
23. A.E. Eiben, R. Hinterding, and Z. Michalewicz. Parameter control in evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 3(2):124–141, 1999.
24. A.E. Eiben, B. Jansen, Z. Michalewicz, and B. Paechter. Solving CSPs using self-adaptive constraint weights: how to prevent EAs from cheating. In Whitley et al. [88], pages 128–134.
25. A.E. Eiben, P.-E. Raué, and Z. Ruttkay. GA-easy and GA-hard constraint satisfaction problems. In M. Meyer, editor, *Proceedings of the ECAI-94 Workshop on Constraint Processing*, number 923 in LNCS, pages 267–284. Springer, Berlin, Heidelberg, New York, 1995.
26. A.E. Eiben and Z. Ruttkay. Self-adaptivity for constraint satisfaction: Learning penalty functions. In ICEC-96 [44], pages 258–261.
27. A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computation*. Springer, 2003.
28. A.E. Eiben and J.K. van der Hauw. Solving 3-SAT with adaptive genetic algorithms. In ICEC-97 [45], pages 81–86.

29. A.E. Eiben and J.I. van Hemert. SAW-ing EAs: adapting the fitness function for solving constrained problems. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, chapter 26, pages 389–402. McGraw Hill, London, 1999.
30. D.B. Fogel. *Evolutionary Computation*. IEEE Press, 1995.
31. D.B. Fogel and J.W. Atmar. Comparing genetic operators with Gaussian mutations in simulated evolutionary processes using linear systems. *Biological Cybernetics*, 63(2):111–114, 1990.
32. S. Forrest, editor. *Proceedings of the 5th International Conference on Genetic Algorithms*. Morgan Kaufmann, San Francisco, 1993.
33. B. Friesleben and M. Hartfelder. Optimisation of genetic algorithms by genetic algorithms. In R.F. Albrecht, C.R. Reeves, and N.C. Steele, editors, *Artifical Neural Networks and Genetic Algorithms*, pages 392–399. Springer, Berlin, Heidelberg, New York, 1993.
34. D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.
35. J.J. Grefenstette. Optimisation of control parameters for genetic algorithms. *IEEE Transaction on Systems, Man and Cybernetics*, 16(1):122–128, 1986.
36. N. Hansen, S. Mller, and P. Koumoutsakos. Reducing the Time Complexity of the Derandomized Evolution Strategy with Covariance Matrix Adaptation (CMA-ES). *Evolution Computation*, 11(1), 2003.
37. N. Hansen and A. Ostermeier. Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaption. In *Proceedings of the Third IEEE International Conference on Evolutionary Computation*, pages 312–317. IEEE Press, 1996.
38. N. Hansen and A. Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9(2):159–195, 2001.
39. N. Hansen, A. Ostermeier, and A. Gawelczyk. On the adaptation of arbitrary normal mutation distributions in evolution strategies: The generating set adaptation. In L. J. Eshelman, editor, *Proceedings of the* 6^{th} *International Conference on Genetic Algorithms*, pages 57–64. Morgan Kaufmann, 1995.
40. J. Hesser and R. Manner. Towards an optimal mutation probablity in genetic algorithms. In H.-P. Schwefel and R. Männer, editors, *Proceedings of the 1st Conference on Parallel Problem Solving from Nature*, number 496 in Lecture Notes in Computer Science, pages 23–32. Springer, Berlin, Heidelberg, New York, 1991.
41. R. Hinterding, Z. Michalewicz, and A.E. Eiben. Adaptation in evolutionary computation: A survey. In ICEC-97 [45].
42. R. Hinterding, Z. Michalewicz, and T.C. Peachey. Self-adaptive genetic algorithm for numeric functions. In Voigt et al. [84], pages 420–429.
43. *Proceedings of the First IEEE Conference on Evolutionary Computation*. IEEE Press, Piscataway, NJ, 1994.
44. *Proceedings of the 1996 IEEE Conference on Evolutionary Computation*. IEEE Press, Piscataway, NJ, 1996.
45. *Proceedings of the 1997 IEEE Conference on Evolutionary Computation*. IEEE Press, Piscataway, NJ, 1997.
46. A. Jain and D.B. Fogel. Case studies in applying fitness distributions in evolutionary algorithms. II. comparing the improvements from crossover and gaussian mutation on simple neural networks. In X. Yao and D.B. Fogel, editors, *Proc. of*

the 2000 IEEE Symposium on Combinations of Evolutionary Computation and Neural Networks, pages 91–97, 2000.
47. J.A. Joines and C.R. Houck. On the use of non-stationary penalty functions to solve nonlinear constrained optimisation problems with ga's. In ICEC-94 [43], pages 579–584.
48. B.A. Julstrom. What have you done for me lately?: Adapting operator probabilities in a steady-state genetic algorithm. In L.J. Eshelman, editor, *Proceedings of the 6th International Conference on Genetic Algorithms*, pages 81–87. Morgan Kaufmann, San Francisco, 1995.
49. Y. Kakuza, H. Sakanashi, and K. Suzuki. Adaptive search strategy for genetic algorithms with additional genetic algorithms. In Männer and Manderick [59], pages 311–320.
50. S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by simulated anealing. *Science*, 220:671–680, 1983.
51. J. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.
52. N. Krasnogor and J.E. Smith. A memetic algorithm with self-adaptive local search: TSP as a case study. In Whitley et al. [88], pages 987–994.
53. N. Krasnogor and J.E. Smith. Emergence of profitable search strategies based on a simple inheritance mechanism. In Spector et al. [81], pages 432–439.
54. M. Lee and H. Takagi. Dynamic control of genetic algorithms using fuzzy logic techniques. In Forrest [32], pages 76–83.
55. J. Levenick. Swappers: introns promote flexibility, diversity, and invention. In *Proceedings of the Genetic and Evolutionary Computation Conference GECCO '99*, pages 361–368, 1999.
56. J. Lis. Parallel genetic algorithm with dynamic control parameter. In ICEC-96 [44], pages 324–329.
57. J. Lis and M. Lis. Self-adapting parallel genetic algorithm with the dynamic mutation probability, crossover rate, and population size. In J. Arabas, editor, *Proceedings of the First Polish Evolutionary Algorithms Conference*, pages 79–86. Politechnika Warszawska, Warsaw, 1996.
58. S.W. Mahfoud. Boltzmann selection. In Bäck et al. [12], pages C2.5:1–4.
59. R. Männer and B. Manderick, editors. *Proceedings of the 2nd Conference on Parallel Problem Solving from Nature*. North-Holland, Amsterdam, 1992.
60. K.E. Mathias and L.D. Whitley. Remapping hyperspace during genetic search: Canonical delta folding. In L.D. Whitley, editor, *Foundations of Genetic Algorithms 2*, pages 167–186. Morgan Kaufmann, San Francisco, 1993.
61. K.E. Mathias and L.D. Whitley. Changing representations during search: A comparative study of delta coding. *Evolutionary Computation*, 2(3):249–278, 1995.
62. Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer, Berlin, Heidelberg, New York, 3rd edition, 1996.
63. Z. Michalewicz and M. Schoenauer. Evolutionary algorithms for constrained parameter optimisation problems. *Evolutionary Computation*, 4(1):1–32, 1996.
64. Akira Oyama, Shigeru Obayashi, and Takashi Nakamura. Real-coded adaptive range genetic algorithm applied to transonic wing optimization. In M. Schoenauer et al., editor, *Proceedings of the* 6^{th} *Conference on Parallel Problems Solving from Nature*, pages 712–721. LNCS 1917, Springer Verlag, 2000.
65. J.D. Schaffer, editor. *Proceedings of the 3rd International Conference on Genetic Algorithms*. Morgan Kaufmann, San Francisco, 1989.

66. 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 optimisation. In Schaffer [65], pages 51–60.
67. J.D. Schaffer and L.J. Eshelman. On crossover as an evolutionarily viable strategy. In Belew and Booker [16], pages 61–68.
68. D. Schlierkamp-Voosen and H. Mühlenbein. Strategy adaptation by competing subpopulations. In Y. Davidor, H.-P. Schwefel, and R. Männer, editors, *Proceedings of the 3rd Conference on Parallel Problem Solving from Nature*, number 866 in Lecture Notes in Computer Science, pages 199–209. Springer, Berlin, Heidelberg, New York, 1994.
69. H.-P. Schwefel. *Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie*, volume 26 of *ISR*. Birkhaeuser, Basel/Stuttgart, 1977.
70. H.-P. Schwefel. *Numerical Optimisation of Computer Models*. Wiley, New York, 1981.
71. J.E. Smith. *Self Adaptation in Evolutionary Algorithms*. PhD thesis, University of the West of England, Bristol, UK, 1998.
72. J.E. Smith. Modelling GAs with self-adaptive mutation rates. In Spector et al. [81], pages 599–606.
73. J.E. Smith. On appropriate adaptation levels for the learning of gene linkage. *J. Genetic Programming and Evolvable Machines*, 3(2):129–155, 2002.
74. J.E. Smith. Parameter perturbation mechanisms in binary coded gas with self-adaptive mutation. In Rowe, Poli, DeJong, and Cotta, editors, *Foundations of Genetic Algorithms 7*, pages 329–346. Morgan Kaufmann, San Francisco, 2003.
75. J.E. Smith and T.C. Fogarty. Adaptively parameterised evolutionary systems: Self adaptive recombination and mutation in a genetic algorithm. In Voigt et al. [84], pages 441–450.
76. J.E. Smith and T.C. Fogarty. Recombination strategy adaptation via evolution of gene linkage. In ICEC-96 [44], pages 826–831.
77. J.E. Smith and T.C. Fogarty. Self adaptation of mutation rates in a steady state genetic algorithm. In ICEC-96 [44], pages 318–323.
78. J.E. Smith and T.C. Fogarty. Operator and parameter adaptation in genetic algorithms. *Soft Computing*, 1(2):81–87, 1997.
79. R.E. Smith and E. Smuda. Adaptively resizing populations: Algorithm, analysis and first results. *Complex Systems*, 9(1):47–72, 1995.
80. W.M. Spears. Adapting crossover in evolutionary algorithms. In J.R. McDonnell, R.G. Reynolds, and D.B. Fogel, editors, *Proceedings of the 4th Annual Conference on Evolutionary Programming*, pages 367–384. MIT Press, Cambridge, MA, 1995.
81. L. Spector, E. Goodman, A. Wu, W.B. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. Garzon, and E. Burke, editors. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*. Morgan Kaufmann, San Francisco, 2001.
82. C.R. Stephens, I. Garcia Olmedo, J. Moro Vargas, and H. Waelbroeck. Self-adaptation in evolving systems. *Artificial life*, 4:183–201, 1998.
83. G. Syswerda. A study of reproduction in generational and steady state genetic algorithms. In G. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 94–101. Morgan Kaufmann, San Francisco, 1991.
84. H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel, editors. *Proceedings of the 4th Conference on Parallel Problem Solving from Nature*, number 1141

in Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, New York, 1996.
85. N. Wagner and Z. Michalewicz. Genetic programming with efficient population control for financial times series prediction. In *Genetic and Evolutionary Computation Conference Late Breaking Papers*, pages 458–462, 2001.
86. N. Wagner, Z. Michalewicz, M. Khouja, and R.R. McGregor. Forecasting with dynamic window of time: the dyfor genetic program model. In *Proceedings of the International Workshop on Intelligent Media Technology for Communicative Intelligence, Warsaw, Poland, September 13 14, 2004*. Springer-Verlag, 2005.
87. D. Whitley. The genitor algorithm and selection pressure: why rank-based allocation of reproductive trials is best. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 116–121, 1989.
88. D. Whitley, D. Goldberg, E. Cantu-Paz, L. Spector, I. Parmee, and H.-G. Beyer, editors. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*. Morgan Kaufmann, San Francisco, 2000.
89. L.D. Whitley, K.E. Mathias, and P. Fitzhorn. Delta coding: An iterative search strategy for genetic algorithms,. In Belew and Booker [16], pages 77–84.
90. D.H. Wolpert and W.G. Macready. No Free Lunch theorems for optimisation. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.
91. B. Zhang and H. Mühlenbeim. Balancing accuracy and parsimony in genetic programming. *Evolutionary Computing*, 3(3):17–38, 1995.