Ghodrat Moghadampour, PhD
Vaasa University of Applied Sciences
Vaasa, Finland
Mobile: +358407616223
Fax: +358 6 326 3112
E-Mail: mg@puv.fi

Keywords

Evolutionary algorithms, genetic algorithms, parameter control, adaptation, mutation

Abstract

Evolutionary algorithms are affected by more parameters than optimization methods typically. This is at the same time a source of their robustness as well as a source of frustration in designing them. Adaptation can be used not only for finding solutions to a given problem, but also for tuning genetic algorithms to the particular problem.

Adaptation can be applied to problems as well as to evolutionary processes. In the first case adaptation modifies some components of genetic algorithms to provide an appropriate form of the algorithm, which meets the nature of the given problem. These components could be any of representation, crossover, mutation and selection. In the second case, adaptation suggests a way to tune the parameters of the changing configuration of genetic algorithms while solving the problem.

In this paper a brief review of adaptation techniques is provided and some new techniques to implement adaptation in the mutation process are presented.

1. Introduction

Evolutionary algorithms are heuristic algorithms, which imitate the natural evolutionary process and try to build better solutions by gradually improving present solution candidates. They are mainly used to solve problems which are hard to solve in conventional ways or there is no pre-known solution for them. In an evolutionary algorithm 1) problems are described by a set of parameters, 2) parameters are interpreted as a set of artificial genes, 3) genes are considered as blueprints of individuals and 4) evolution is applied to individuals (Fogel, Owens & Walsh 1966; Rechenberg 1973; Holland 1975; Krink 2005).

It is generally accepted that any evolutionary algorithm must have five basic components: 1) a genetic representation of a number of solutions to the problem, 2) a way to create an initial population of solutions, 3) an evaluation function for rating solutions in terms of their “fitness”, 4) “genetic” operators that alter the genetic composition of offspring during reproduction, 5) values for the parameters, e.g. population size, probabilities of applying genetic operators (Michalewicz 1996).

Genetic algorithm is one kind of evolutionary algorithms, which starts the solution process by randomly generating the initial population and then refining the present solutions through natural like operators, like crossover and mutation. The behavior of the genetic algorithm can be adjusted by parameters, like the size of the initial population, the number of times genetic operators are applied and how these genetic operators are implemented. Deciding on the best possible parameter values over the genetic run is a challenging task, which requires even better and efficient techniques.

2. Genetic Algorithm

Most often genetic algorithms (GAs) have at least the following elements in common: populations of chromosomes, selection according to fitness, crossover to produce new offspring, and random mutation of new offspring. A simple GA works as follows (Mitchell 1998):

1. Start with a randomly generated population of *n l*-bit strings (chromosomes)
2. Calculate the fitness *f(x)* of each bit string *x* in the population
3. Repeat the following steps until *n* offspring have been created:
    i. Select a pair of parent bit strings from the current population, the probability of selection being an increasing function of fitness. Selection is done “with replacement”, meaning that the same chromosome can be selected more than once to become a parent.
    ii. With probability *Pc* (crossover probability or crossover rate) cross over the pair at a randomly chosen point (chosen with uniform probability) to form two offspring. If no crossover takes place, form two offspring that are exact copies of their respective parents. In multi-point crossover the crossover rate for a pair of parents is the number of points at which a crossover takes place.
    iii. Mutate the two offspring at each locus with probability *pm* (the mutation probability or mutation rate), and place the resulting bit strings in the new population
4. Replace the current population with the new population
5. Go to step 2.

3. Control Parameters

Evolutionary algorithms are affected by more parameters than optimization methods typically. This is at the same time a source of their robustness as well as a source of frustration in designing them (Michalewicz & Fogel 2004).

Adaptation can be used not only for finding solutions to a given problem, but also for tuning genetic algorithms to the particular problem (Gen et al. 2000). Adaptation can be applied to problems as well as to evolutionary processes. In the first case, adaptation modifies some components of genetic algorithms to provide an appropriate form of the algorithm, which meets the nature of the given problem. These components could be any of representation, crossover, mutation and selection.

In the second case, adaptation suggests a way to tune the parameters of the changing configuration of genetic algorithms while solving the problem (Gen et al. 2000). Some of such parameters are: population size and structure, like subpopulations, genome representation (floating point, binary, parse tree, matrix), precision and length, crossover type (arithmetic, -point, etc.), the number of crossover points and probability, mutation type (uniform, Gaussian, etc.), mutation variance and probability, selection type (tournament, proportional, etc.), tournament size.

The challenge is that optimal parameters of an EA are problem dependent and there is a large set of possible EA settings. The No-Free-Lunch theorem implies that no set of parameters for an EA is superior on all problems (Krink 2005). Finding the right parameter values is a time-consuming task and it has been the subject of many researches.

The main criteria for classifying parameter setting methods are: 1) what is changed, 2) how the change is made. The first criterion refers to the components of the evolutionary algorithm and consists of six

categories: 1) representation, 2) evaluation function, 3) variation operators (mutation and recombination), 4) selection, 5) replacement, 6) population.
The second criterion refers to the parameter setting methods, which can be divided to three main types: 1) deterministic (or fixed) parameter control (also called parameter tuning) in which the parameter-altering transformations takes no input variables related to the progress of search method, 2) adaptive (also called explicitly adaptive) parameter control in which there is some form of feedback from the search, 3) self-adaptive (also called implicitly adaptive) parameter control in which the parameters to be adapted are encoded into the chromosomes and undergo mutation and recombination (Eiben, Hinterding & Michalewicz 1999).

Since all components of the algorithms have parameters, which need to be set to reasonable values, manual tuning is impossible to avoid completely even in adaptive control techniques. Furthermore, most advanced parameter control techniques also have parameters. However, adaptive parameter control seems to be still worthy due to the following facts: 1) the performance of parameter-varying algorithms is better, 2) since the control technique is usually more robust with respect to parameter sensitivity tuning; the control method is often easier than tuning the actual parameters (Ursem 2003).

4. Mutation Operators

Mutation is a bit reversal event that occurs with small probabilities p_m per bit. Efforts to tune the mutation probability have resulted to different values and hence leaving practitioners in ambiguity. As results of tuning "optimal" mutation rate, the best rate found to be $p_m \approx 0.001$ (De Jong 1975), $p_m \approx 0.01$ (Grefenstette 1986), $p_m \in [0.005, 0.01]$ (Schaffer et al. 1989) and $p_m = 1/L$ (Mühlenbein 1992), where L is the length of the bit string (Michalewicz et al. 2004).

Different techniques for updating the mutation rate over time have also been presented by different researchers. A time dependent schedule for controlling the mutation is presented in (Laumanns, Rudolph & Schwefel 1998; Laumanns, Rudolph & Schwefel 2001). Here the mutation step sizes are discounted by a constant factor each time an offspring is produced. Fogarty presented three ideas for controlling the mutation rate in bit-flip mutation (Fogarty 1989; Ursem 2003). The first idea was to control the mutation rate $p_m(t)$ by an exponential decreasing function of the generation number t according to the following formula:

$$p_m(t) = \frac{1}{240} + \frac{0.11375}{2^t}. \qquad (1)$$

The second idea was to have different mutation rates for different bits in the chromosome. More precisely to have high mutation rates on the least significant bits and low mutation rates for the most significant bits according to the following formula:

$$p_m(i) = \frac{0.3528}{2^{i-1}}, \qquad (2)$$

where i is the bit number and $i = 1, 2, ...10$, with $i = 1$ being the least significant bit.

The third idea was to combine two previous formulas and control the mutation rates according to the following formula:

$$p_m(t,i) = \frac{28}{1905.2^{i-1}} + \frac{0.4026}{2^{t+i-1}} \, . \qquad (3)$$

Fogarty compared these three schemes with a scheme using constant mutation rate of $p_m = 0.01$ and noticed that the combination of varying across both generations and bit number was the best of all four techniques (Ursem 2003).

A theoretically optimal schedule for deterministically changing p_m for the counting-ones function is presented in (Hesser & Männer 1991; Eiben et al. 1999; Michalewicz et al. 2004). Here, the value for p_m is defined in the following way over time t:

$$p_m(t) = \sqrt{\frac{\alpha}{\beta}} \times \frac{\exp(\frac{-\gamma t}{2})}{\lambda\sqrt{L}} \, , \qquad (4)$$

where α, β and γ are constants, λ is the population size, L is the string length and t is the generation number.

The function to control the decrease of p_m is defined as a linear decreasing function from 0.5 to $1/L$ in generation T in (Bäck & Schütz 1996; Eiben et al. 1999 and Ursem 2003). The value of $p_m(t)$ is constrained so that $p_m(0) = 0.5$, $p_m(T) = \frac{1}{L}$ and otherwise:

$$p_m(t) = \left(2 + \frac{L-2}{T} t\right)^{-1} , \quad \text{if } 0 \le t \le T. \qquad (5)$$

An optimal schedule for decreasing the mutation rate as a function of the distance to the optimum is defined in (Bäck 1992 a; Eiben et al. 1999; Michalewicz et al. 2004) in the following way:

$$p_m(f(x)) \approx \frac{1}{2(f(x)+1) - L} \, . \qquad (6)$$

The idea of varying the mutation rate over both bit index number and generation was revisited and implemented in slightly different form from the traditional bit-flip mutation operation in (Janikow et al. 1991; Ursem 2003). The new mutation operator was called *non-uniform mutation* for binary encoding and was derived from the version used for real encoding. The operator mutates the k'th ($k = 1, 2, 3, ... q$) binary encoded parameter x_k of a candidate solution $\mathrm{x} = (x_1, ..., x_k, ..., x_q)$ according to:

$$x_k^{t+1} = \begin{cases} x_k^t + \Delta(t, righ(k) - x_k), & \text{if a random binary digit is 0} \\ x_k^t - \Delta(t, x_k - left(k)), & \text{if a random binary digit is 1} \end{cases} . \qquad (7)$$

The functions $left(k)$ and $right(k)$ determine the valid range $[left(k), right(k)]$ for each point x_k in the search space, where other variables x_i ($i = 1, ..., k-1, k+1, ..., q$) remain fixed. The function $\Delta(t, y)$ returns a value in the range $[0, y]$ so that the probability of $\Delta(t, y)$ being close to 0 increases as the generation number, t increases. This property causes this operator to search the space uniformly initially, when t is small, and very locally at later stages. Function $\Delta(t, y)$ is defined as:

$$\Delta(t, y) = yr(1 - \frac{t}{T})^b, \tag{8}$$

where r is a random number from [0, 1], T is the maximal generation number, and b is a system parameter determining the degree of non-uniformity.

Controlling the variance in Gaussian mutation is very critical in successful application of real-encoded EAs. The standard approach for doing this is to set the variance of the mutation according to a monotonic decreasing function depending on the generation number. These heuristic functions are usually developed by scrutinizing the experimental data and forming a hypothesis from the relationship between the performance of the algorithm and how the parameters were changed (Ursem 2003).

A classical adaptive method for changing the mutation step size is Rechenberg's 1/5 rule for Gaussian mutation in (1+1)-EAs presented in (Rechenberg 1973; Eiben et al. 1999). This rule states that the ratio of successful mutations to all mutations should be 1/5 measured over a number of generations. The standard deviation σ should increase if the ratio is above 1/5, decrease if it is below, and remain unchanged if it is 1/5 (Ursem 2003). It is assumed that maximum progress can be achieved through mutation step sizes leading to a success probability of approximately 20%.

The ratio between crossovers and mutations is regulated based on their performance in (Julstrom 1995; Eiben et al. 1999). Both operators are used separately to create an offspring, and the algorithm keeps a tree of their recent contributions to the new offspring and rewards them accordingly.

Mutation rates in a parallel GA are adapted with a farming model in (Lis 1996; Eiben et al. 1999). Probability of mutation, crossover and the population size in an algorithm of parallel farming model has been adapted in (Lis & Lis 1996; Eiben et al. 1999). They use parallel populations and each of them has one value, out of a possible three different values for p_m, p_c and the population size. The populations are compared after a certain period of time, and then the values for p_m, p_c and the population size are shifted one level towards the values of the most successful population (Eiben et al. 1999).

Gaussian mutation of a real-encoded variable x_i is usually performed according to (Ursem 2003):

$$x_i' = x_i + N(0, \sigma_i(t)). \tag{9}$$

The traditional approach to set the mutation variance is using either a linear or an exponentially decreasing function such as:

$$\sigma_i(t) = 1/\sqrt{1+t}. \tag{10}$$

Self-adaptive control of mutation step sizes has been reported in (Schwefel 1995; Bäck 1996; Bäck et al. 1997; Eiben et al. 1999). Mutating a floating-point object variable x_i happens in the following way:

$$x_i' = x_i + \sigma_i N(0,1), \tag{11}$$

where the mean step sizes are modified lognormally:

$$\sigma_i' = \sigma_i \exp(\tau' N(0,1) + \tau N_i(0,1)), \tag{12}$$

where τ and τ' are the so-called learning rates. The value of σ can also be modified normally:

$$\sigma_i' = \sigma_i + \zeta \sigma_i N(0,1), \tag{13}$$

where ζ is a scaling constant (Fogel 1995; Eiben et al. 1999). Empirical evidence suggests that lognormal perturbation of mutation rates is preferable to Gaussian perturbations on fixed-length real-valued representations (Saravanan & Fogel 1994; Saravanan, Fogel & Nelson 1995; Eiben et al. 1999). However, a slight advantage of Gaussian perturbations over lognormal updates for self-adaptively evolving finite state machines is reported in (Angeline, Fogel & Fogel 1996; Eiben et al. 1999).

Significant improvement can be achieved by controlling the mutation variance by other techniques than a strictly decreasing function (Ursem 2003). The so-called sand pile model was used to generate power-law distributed numbers for controlling the variance in Gaussian mutation (Krink, Thomsen & Rickers 2000; Ursem 2003). The sandpile model is a simple approach to study many complex phenomena found in nature and is an example of how self-organized criticality (SOC) can be generated by very simple means (Bak 1996; Ursem 2003).

Self-adaptation of the mutation step size for optimizing numeric functions in a real valued GA is applied in (Hinterding 1995; Eiben et al. 1999). In another experimentation individuals are replaced by their offspring. Probabilities of crossover and mutation for each chromosome are added to its bit string and adapted in proportion to the population maximum and mean fitness (Srinivas & Patnaik 1994; Eiben et al. 1999). In (Kursawe 1991; Laumanns et al. 2001) the selection criterion changes randomly over time. To make individuals cope with the changing environment they are supplied with a set of step sizes for each objective function through polyploidy. Polyploidy is a situation when the number of chromosomes in a cell becomes doubled. This can happen by a mutation that simply makes two copies. It can also happen when the chromosomes from two different species are mixed.

The mutation rate of GAs can also be self-adapted by adding the rate of mutating p_m, coded in bits, to every individual. Then the new p_m is used to mutate the individual's object variables. This is based on the idea that better p_m rates will produce better offspring and then hitchhike on their improved children to new generations, while bad rates will die out (Bäck 1992 a, Bäck 1992 b; Eiben et al. 1999). The same idea with an implementation of 1/5 success rule for mutation has been applied on a steady-state GA in (Smith & Fogarty 1996; Eiben et al. 1999).

Laumanns et al. (2001) discuss the problem of controlling mutation strength in multi-objective evolutionary algorithms and its implications for the convergence to the Pareto set. A Pareto set is defined to be the set of all Pareto optimal decision vectors and a Pareto optimal vector is defined to be a vector, which makes the optimization function converge the most. Convergence here refers to the iterative approach of populations to the Pareto set of the underlying optimization problem.

An algorithm should ensure convergence to Pareto set and provide a “good” distribution of solutions in order to find or to approximate the set of efficient or Pareto-optimal solutions. Fitness assignment methods based on the notion of dominance seem to produce better solution distributions than plain aggregating methods. Density based selection methods maintain diversity and the use of elitism speeds up the search in the direction of the Pareto set and ensure convergence properties. (Laumanns, Zitzler & Thiele 2001; Rudolph & Agapie 2000; Laumanns et al. 2001). However, virtually all implementations focusing on the role the variation operators in evolutionary multi-objective optimization use standard non-adaptive operators from the single objective case (Laumanns et al. 2001).

Approximating the Pareto set of a multi-objective optimization problem by evolutionary algorithms faces two main problems:

1. The velocity and reliability of convergence to the Pareto set, same as in single objective optimization.
2. Distribution of solutions, caused by the existence of multiple Preto-optimal solutions in the multi-objective case.

Laumanns et al. (2001) present the so-called Predator-Prey model. In this model predator individuals move across the spatial structure so that they delete the worst prey individual of their neighborhood according to their associated objective function. The authors present two adaptation rules capable of increasing the step sizes on need for the model: 1) the standard mutative self-adaptation and 2) self-adaptation through recombination frequency. These rules have worked well. The model converges to the Pareto set due to the superior success probability of single criteria selection in the vicinity of the Pareto set, where it gets increasingly difficult to make cooperative steps from one mutation alone.

The second method, self-adaptation through recombination frequency, combines the implicit evaluation of good “inner models” through fitness evaluation and selection with a fixed but flexible schedule depending on the recombination frequency. In this way all individuals use bigger step sizes for the early offspring and smaller for later ones. The mutation of the step sizes is done deterministically according to the rule:

$$\sigma^{(t+1)} = \gamma^{d-d_0}\sigma^{(t)}, \quad \gamma \in \left]0,1\right[\, . \tag{14}$$

Here d denotes the number of descendants an individual has produced so far. The delay parameter, d_0 determines the number of descendants that must be created before smaller step sizes are passed on to the offspring. The adaptation rate can be controlled by γ. In this schedule for step sizes just one order of magnitude greater than the optimal ones the success rate rapidly goes to zero. Therefore it is a matter of time until appropriate step sizes will be found.

5. Adaptive Mutation Operators

One major problem with the classical implementation of binary mutation, the multiple point mutation or the crossover operator is that it is difficult to control their effect or to restrict changes caused by them within certain limits.

Therefore, several techniques are developed to implement the genetic operators intelligently so that the resulting modifications on the binary string will cause changes in the real values within the desired limits. This idea is implemented so that the real value of the variable is randomly changed within the desired limits and the modified value then is converted to the binary representation and stored as the value of the variable. In this way we can cause more intelligent mutations in the bit strings and make sure that changes in real values are within the desired bounds.

Apparently, changes of different magnitudes are required at different stages of the evolutionary process. Thus, two types of decimal mutation operators have been implemented:

1. For modifying variables with integer values. The bounds for the absolute values of such changes are at least 1 and at most the integer part of the real value representation of the variable. This means that the upper bound of the range may vary even for each variable of the same individual. The randomly selected mutation value may be either positive or negative. Thus, if the integer part of the variable is $|\mathrm{int}(\mathit{variable})|$, the integer mutation range is $\pm\left[1, |\mathrm{int}(\mathit{variable})|\right]$.
2. For modifying variables with values from the range $(0,1)$. The lower bound for the absolute value of such changes is determined by the required precision of the real value presentation of the variable, like 10^{-6}. The upper bound for the absolute value of such changes is determined by decimal part of the variable. Here also the mutation value can be either positive or negative. Thus, if the number of digits after the decimal point for a variable is $\mathrm{precision}(\mathit{variable})$, and the decimal part of the variable is $\mathrm{decimal}(\mathit{variable})$, the range for the real mutation values is $\pm\left[10^{-\mathrm{precision}(\mathit{variable})}, |\mathrm{decimal}(\mathit{variable})|\right]$.

5.1 The Integer Mutation Operator

The *integer mutation* (IM) operator mutates the individuals of the population in relatively great magnitudes. During this operation an integer mutation value Γ is selected randomly from the following range:

$$\Gamma \in \pm\left[1, |\mathrm{int}(\mathit{variable})|\right] \qquad (15)$$

and added to the variable under mutation. Here, $|\mathrm{int}(\mathit{variable})|$ stands for the absolute value of the integer part of the variable. Clearly, this integer part does not necessarily cover the whole range of the variable. To avoid wasting resources special care is taken to make sure that the generated random number is not 0. Thus, the upper bound for the integer mutation value is different for each variable and is defined by the absolute value of the integer part of the variable. This will make the process more flexible and intelligent.

If the upper bound of the mutation value is set to a fixed value, the operator becomes inefficient or the probability for its failure will rise. For instance, if the optimal value of a variable is 0.05 and its present value 80.64, we will need 80 successful integer mutations of magnitude 1 in order to get close to the optimal value of the variable. However, if the magnitude of the integer mutation value can be dynamically determined by the magnitude of the variable, the operator will have a much greater chance to improve the value of the variable dramatically in a short time.

During this operation for each variable in the individual, first a Boolean value is randomly generated that determines whether the mutation operation for the variable at hand should take place or not. If the Boolean value is true, then a randomly generated integer number within the specified bounds is added to the decimal value of the variable. This process is repeated for each variable of the individual separately and the binary representation of the resulting offspring is updated. The offspring is then evaluated and put through the survivor selection procedure. Once this process is ready, the new offspring goes through the variable replacement operator for possible improvement. Next, the offspring, which has possibly been modified by the variable replacement operator, goes through the survivor selection procedure for possible substitution. There is no fixed rate for this operator. All population members go through this operator at least once. This operation is combined with the variable replacement operator described previously. After going through the variable replacement operator, the offspring goes again through the survivor selection procedure for evaluation and possible selection.

5.2 The Decimal Mutation Operator

The *decimal mutation* (DM) operator is used in order to implement changes of smaller magnitudes on individuals. During this operation non-zero decimal numbers in the specific range are randomly generated and added to the randomly selected variables in the individual. The upper bound for the decimal mutation values is determined by the absolute value of the decimal part of variables. If the decimal part of a *variable* is denoted by $\text{decimal}(variable)$, the maximum distance of the mutation values from 0 is $|\text{decimal}(variable)|$. The lower bound of the mutation range is determined by the number of digits after the decimal point. Thus, if $\text{precision}(variable)$ shows the number of required decimal places of a *variable*, the decimal mutation value is determined in the following way:

$$\varphi \in \pm\left[10^{-\text{precision}(variable)}, |\text{decimal}(variable)|\right] \qquad (16)$$

The difference between this operator and the integer mutation operator is the way the mutation value is determined. Otherwise, these operators are similar. Figure 17 provides the pseudo code for the operator.

6. Experimentation and Conclusions

The integer mutation operator, which can be compared to the multiple point mutation operator, selects randomly an integer mutation number between 0 and the magnitude of the integer part of the variable and randomly decides whether to add or subtract the mutation number from the variable.

The decimal mutation operator selects randomly a decimal mutation value greater than 0 and less than 1. Then it randomly decides whether to add or subtract the mutation decimal value from the variable.

These operators were tested on 44 test problems in 2200 runs. Experimentation showed that the most efficient operators are the integer mutation and the decimal mutation operators, which were able to improve the population fitness values the most.

7. References

Angeline, P.J., D.B. Fogel & L.J. Fogel (1996). A comparison of self-adaptation Methods for finite state machines in dynamic environments. In: *Proceedings of the 5th Annual Conference on Evolutionary Programming*. Eds L.J. Fogel, P.J. Angeline & T. Bäck, MIT Press.

Bäck, T. & M. Schütz (1996*).* Intelligent mutation rate control in canonical genetic algorithms. In: Foundations of intelligent systems, 1079. In: *Lecture Notes in Artificial Intelligence*, 158-167. Eds Z. Ras & M. Michalewicz. Springer-Verlag.

Bäck, T. (1992 a). The interaction of mutation rate, selection, and self-adaptation within a genetic algorithm. In: *Proceedings of the 2nd Conference on Parallel Problem Solving from Nature*, 85-94. Eds R. Männer & B. Manderick, B. North-Holland.

Bäck, T. (1992 b). Self-adaptation in genetic algorithms. In: *Toward a Practice of Autonomous Systems: Proceedings of the 1st European Conference on Artificial Life*, 263-271. Eds F. J. Varela & P. Bourgine. MIT Press.

Bäck, Thomas, Ulrich Hammel & Hans-Paul Schwefel (1997). Evolutionary computation: comments on the history and current state. In: *Handbook of Evolutionary Computation*. Eds Thomas Bäck, D. B. Fogel & Z. Michalewicz. New York: Oxford University Press.

Bak, P. (1996). *How Nature Works*. Copernicus, Springer-Verlag, 1st edition.

Eiben, Ágoston E., Robert Hinterding & Zbigniew Michalewicz (1999). Parameter control in evolutionary algorithms. *IEEE Transactions on Evolutionary Computation* 3(2), 124-141. This paper has won the 2001 IEEE Transactions on Evolutionary Computation Outstanding Paper.

Fogarty, T. (1989). Varying the probability of mutation in the genetic algorithm. In: *Proceedings of the 3rd International Conference on Genetic Algorithms*, 104-109. Ed. J.D. Schaffer. Morgan Kaufmann.

Fogel, D.B. (1995). *Evolutionary Computation*. IEEE Press.

Fogel, L. J., A.J. Owens & M.J. Walsh (1966). *Artificial Intelligence through Simulated Evolution*. Chichester, UK: John Wiley.

Hesser, J. & R. Männer (1991). Towards an optimal mutation probability for genetic algorithms. In: Proceedings of the 1st conference on parallel problem solving from nature. Eds H.-P. Schwefel & R. Männer. In: *Lecture Notes in Computer Science* 496, 23-32. Springer-Verlag.

Hinterding, R. (1995). Gaussian mutation and self-adaptation in numeric genetic algorithms. In: *Proceedings of the 2nd IEEE Conference on Evolutionary Computation*, 384-389. IEEE Press.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*. Ann Arbor: MI: University of Michigan Press.

Janikow, C. Z. & Z. Michalewicz (1991). An experimental comparison of binary and floating-point representations in genetic algorithms. In: *Proceedings of the Fourth International Conference on Genetic Algorithms*. Eds R. K. Belew & L. B. Booker. Morgan Kaufmann.

Julstrom, B.A. (1995). What have you done lately for me? Adapting operator probabilities in a steady-state genetic algorithm. In: *Proceedings of the 6th International Conference on Genetic Algorithms*, 81-87. Ed. L. Eshelman. Morgan Kaufmann.

Krink, T., R. Thomsen & P. Rickers (2000). Applying self-organized criticality to evolutionary algorithms. In: *Parallel Problem Solving from Nature VI (PPSN-2000)*, 1, 375-384. Eds Schoenauer et al.

Krink, Thiemo, Bogdan Filipič & Gary B. Fogel (2004). *Noisy Optimization Problems- A Particular Challenge for Differential Evolution?* Available at: http://www.natural-selection.com/Library/2004/CEC04_Noisy.pdf. Checked in December 2005.

Kursawe, Frank (1991). A variant of evolution strategies for vector optimization. In: *Parallel Problem Solving from Nature*, 193-197. Eds H.-P. Schwefel & R. Männer. Berlin: Springer.

Laumanns, Marco, Eckart Zitzler & Lothar Thiele (2001). On the effects of archiving, elitism and density based selectio in evolutionary multi-objective optimization. In: Evolutionary multi-criterion optimization (emo 2001). Ed. E. Zitzler. In: *Lecture Notes in Computer Science* 1993, 181-196. Berlin: Springer-Verlag.

Laumanns, Marco, Günter Rudolph & Hans-Paul Schwefel (1998). A spatial predator-prey approach to multi-objective optimization: a preliminary study. In: *Parallel Problem Solving From Nature (PPSN-V)*, 241-249. Eds Agoston E. Eiben et al. Berlin: Springer.

Laumanns, Marco, Günter Rudolph & Hans-Paul Schwefel (2001). Mutation control and convergence in evolutionary multi-objective optimization. In: *7th International Conference on Soft Computing MENDEL 2001*, Brno, Czech Republic, June 6-8.

Laumanns, Marco, Günter Rudolph & Hans-Paul Schwefel (2001). Mutation control and convergence in evolutionary multi-objective optimization. In: *7th International Conference on Soft Computing MENDEL 2001*, Brno, Czech Republic, June 6-8.

Lis, J. & M. Lis (1996). Self-adapting parallel genetic algorithm with the dynamic mutation probability, crossover rate and population size. In: *Proceedings of the 1st Polish National Conference on Evolutionary Computation*, 324-329. Ed. J. Arabas. Oficina Wydawnica Politechniki Warszawskiej.

Lis, J. (1996). Parallel Genetic algorithm with dynamic parameter control. In: *Proceedings of the 3rd IEEE Conference on Evolutionary Computation*, 324-329. IEEE Press.

Michalewicz, Zbigniew & David B. Fogel (2004). *How to Solve It: Modern Heuristics.* Second, Revised and Extended Edition. Germany: Springer-Verlag Berlin Heidelberg. ISBN 3-540-22494-7.

Michalewicz, Zbigniew & David B. Fogel (2004). *How to Solve It: Modern Heuristics.* Second, Revised and Extended Edition. Germany: Springer-Verlag Berlin Heidelberg. ISBN 3-540-22494-7.

Michalewicz, Zbigniew (1996). *Genetic Algorithms + Data Structures = Evolution Programs*. Third, Revised and Extended Edition. USA: Springer. ISBN 3-540-60676-9.

Mitchell, Melanie (1998). *An Introducton to Genetic Algorithms*. United States of America: A Bradford Book. First MIT Press Paperback Edition.

Mühlenbein, H. (1992). How genetic algorithms really work: 1. mutation and hill-climbing. In: *Parallel Problem Solving from Nature 2*. Eds R. Männer & B. Manderick. North-Holland.

Rudolph, G. & A. Agapie (2000). Convergence properties of some multi-objective evolutionary algorithms. In: *Congress on Evolutionary Computation (CEC 2000)* 2, 1010-1016. Piscataway, NJ: IEEE Press.

Saravanan, N., D.B. Fogel & K.M. Nelson (1995). A comparison of methods for self-adaptation in evolutionary algorithms. *BioSystems* 36, 157-166.

Schwefel, H.-P. (1995). *Evolution and Optimum Seeking*. New York: Wiley.

Smith, J. & T. Fogarty (1996 a). Adaptively parameterized evolutionary systems: self adaptive recombination and mutation in genetic algorithm. In: Proceedings of the 4th conference on parallel problem solving from nature, 441-450. Eds H.-M. Voigt, W. Ebeling, I. Rechenberg & H.-P. Schwefel. In: *Lecture Notes in Computer Science* 1141. Berlin: Springer.

Srinivas, M. & L.M. Patnaik (1994). Adaptive probabilities of crossover and mutation in genetic algorithms. *IEEE Transactions on Systems, Man and Cybernetics* 24(4), 17-26.

Ursem, Rasmus K. (2003). *Models for Evolutionary Algorithms and Their Applications in System Identification and Control Optimization (PhD Dissertation).* A Dissertation Presented to the Faculty of Science of the University of Aarhus in Partial Fulfillment of the Requirements for the PhD Degree. Department of Computer Science, University of Aarhus, Denmark.