Agoston E. Eiben
Faculteit der Exacte Wetenschappen, Vrije Universiteit
De Boelelaan 1081a, 1081 HV Amsterdam
gusz@cs.vu.nl

Overzichtsartikel

The most powerful compu

De handelsreiziger die de kortste route wil bepalen om N steden te bezoeken, is een bekend voorbeeld van een probleem dat voor grotere waarden van N op geen enkele een computer is uit te rekenen. Een alternatieve, in dit artikel beschreven manier om een dergelijk probleem aan te pakken is het aanleggen van een grote verzameling van 'oplossingen', waar de beste de andere oplossingen zal verdringen. De wetenschap die de beginselen van Darwin van natuurlijke selectie toepast bij het ontwerpen van automatische probleemoplossers, waartoe bovenstaande methode ook behoort, heet evolutionary computing. Guszti Eiben, sinds augustus 1999 hoogleraar computational intelligence, geeft een overzicht van het vakgebied.

Developing automated problem solvers (that is, algorithms) is one of the central themes of mathematics and computer science. Similarly to engineering, where looking at Nature's solutions has always been a source of inspiration, copying 'natural problem solvers' is a stream within these disciplines. When looking for the most powerful problem solver of the universe, two candidates are rather straightforward:

– the human brain;
– the evolutionary process that created the human brain.

Trying to design problem solvers based on these answers leads to the fields of neurocomputing, and evolutionary computing, respectively. The fundamental metaphor of evolutionary computing relates natural evolution to problem solving in a trial-and-error (also known as generate-and-test) fashion.

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

Surprisingly enough, this idea of applying Darwinian principles to automated problem solving dates back to the forties, long before the breakthrough of computers [14]. As early as in 1948 Turing proposed 'genetical or evolutionary search' and already in 1962 Bremermann actually executed computer experiments on optimization through evolution and recombination. During the sixties three different implementations of the basic idea have been developed at three different places. In the USA Fogel introduced evolutionary programming [13, 15], while Holland called his method a genetic algorithm [16–17, 21]. In Germany Rechenberg and Schwefel invented evolution strategies [22–23]. For about 15 years these areas developed separately; it is since the early nineties that they are envisioned as different representatives ('dialects') of one technology that was termed evolutionary computing [1–3, 10, 20]. It was also in the early nineties that a fourth stream following the general ideas has emerged: Koza's genetic programming [4, 18]. The contemporary terminology denotes the whole field by evolutionary computing, or evolutionary algorithms, and considers evolutionary programming, evolution strategies, genetic algorithms, and genetic programming as subareas.

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

Table 1 The basic evolutionary computing metaphor linking natural evolution to problem solving

ting in the universe?

What is an evolutionary algorithm?

As the history of the field suggests there are many different variants of evolutionary algorithms. The common underlying idea behind all these techniques is the same: given a population of individuals the environmental pressure causes natural selection (survival of the fittest) and hereby the fitness of the population is growing. It is easy to see such a process as optimization. Given an objective function to be maximized we can randomly create a set of candidate solutions, that is to say, elements of the domain of the objective function, and apply the objective function as an abstract fitness measure – the higher the better. Based on this fitness, some of the better candidates are chosen to seed the next generation by applying recombination and mutation to them. Recombination is a binary operator applied to two selected candidates (the so-called parents) and results one or two new candidates (the children). Mutation is a unary operator, it is applied to one candidate and results in one new candidate. Executing recombination and mutation leads to a set of new candidates (the offspring) that compete – based on their fitness – with the old ones for a place in the next generation. This process can be iterated until a solution is found or a previously set computational limit is reached. In this process selection acts as a force pushing quality, while variation operators (recombination and mutation) create the necessary diversity. Their combined application leads to improving fitness values in consecutive populations, that is, the evolution is optimizing. (Actually, evolution is not 'optimizing' as there are no general guarentees for finding an optimum, it is rather 'approximizing', by approaching optimal values closer and closer over its course.)

Let us note that many components of such an evolutionary process are stochastic. So is selection, where fitter individuals have a higher chance to be selected than less fit ones, but typically even the weak individuals have a chance to become a parent or to survive. For recombination of two individuals the choice on which pieces will be recombined is random. Similarly for mutation, the pieces that will be mutated within a candidate solution and the new pieces replacing the old ones are chosen randomly. The general scheme of an evolutionary algorithm is given in figure 1.

For the sake of completeness, let us note that the replacement step is often called survivor selection. It is easy to see that the above scheme falls in the category of generate-and-test algorithms. The fitness function represents a heuristic estimation of solution quality and the search process is driven by the variation and the selection operators. Evolutionary algorithms posses a number of features that can help to position them within in the family of generate-and-test methods. For example, they are population based, that is to say, they process a whole collection of candidate solutions simultaneously, they mostly use recombination to mix information of two candidate solutions into a new one and they are stochastic.

The aforementioned dialects of evolutionary computing follow the above general outlines and differ only in technical details. For instance, the representation of a candidate solution is often used to characterize different streams. Traditionally, the candidates are represented by (that is to say, the data structure encoding a solu-

```
INITIALIZE population with random candidate solutions
COMPUTE FITNESS of each candidate
  while not STOP-CRITERION do
    SELECT parents
    RECOMBINE pairs of parents
    MUTATE the resulting offspring
    COMPUTE FITNESS of new candidates
    REPLACE some parents by some offspring
  od
```

Figure 1 General scheme of an evolutionary algorithm

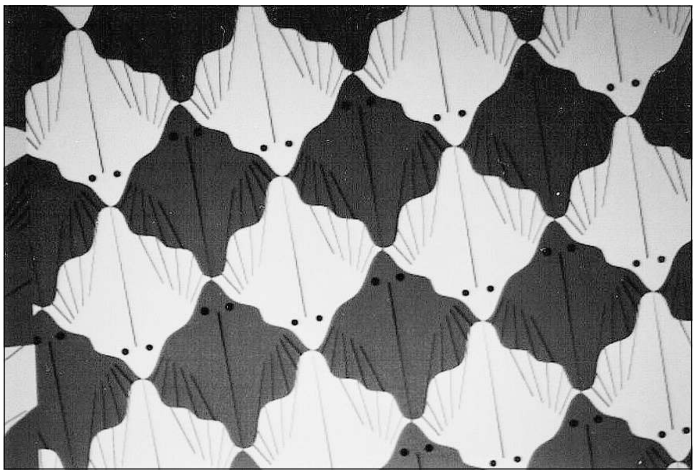
Figure 2 A result of the Escher machine

tion has the form of) bit-strings in genetic algorithms, real-valued vectors in evolution strategies, finite state machines in evolutionary programming and trees in genetic programming. These differences have a mainly historical origin. Technically, a given representation might be preferable over others if it matches the given problem better, that is, it makes the encoding of candidate solutions easier or more natural. For instance, for solving a satisfiability problem the straightforward choice is to use bit-strings of length n, where n is the number of logical variables, hence the appropriate evolutionary algorithm would be a genetic algorithm. For evolving a computer program that can play checkers trees are well-suited (namely, the parse trees of the syntactic expressions forming the programs), thus a genetic programming approach is likely. It is important to note that the recombination and mutation operators working on candidates must match the given representation. That is, for instance in genetic programming the recombination operator works on trees, while in genetic algorithms it operates on bit-strings. As opposed to variation operators, selection takes only the fitness information into account, hence it works independently from the actual representation. Differences in the commonly applied selection mechanisms in each stream are therefore rather a tradition than a technical necessity.

It is worth to note that the borders between the four main evolutionary computing streams are diminishing in the last decade. This 'unionism' has a technical and a psychological aspect. On the one hand, many evolutionary algorithms have been proposed that are hard to classify along the traditional lines. On the other hand, more and more evolutionary computing researchers and practitioners follow a pragmatic attitude choosing whichever type of representation, variation operators, or selection procedures are appropriate, without bothering much about whether the resulting combination fits in one of the traditional categories.

Representation	bit-strings
Recombination	1-point crossover
Mutation	bit-flip
Selection	fitness-proportional
Replacement	generational

Table 2 Sketch of the simple genetic algorithm

Genetic algorithms

In this section we go into more details and illustrate the working of evolutionary algorithms by discussing a well-known type: genetic algorithms.

The simple genetic algorithm

During the last two decades several GA variants have been introduced. Here we will discuss the oldest version, named simple GA, that can be easily specified by the particular instantiations of the EA components as shown in table 2.

In the sequel we will consider these components one by one. The first step in handling a problem by an evolutionary algorithm is to define the representation. In evolutionary terms, one needs to specify which genotypes, also called chromosomes, represent (encode) the given phenotypes, the candidate solutions. To illustrate this matter let us take an extremely simple problem: maximizing $f(x) = x^2$ on $\mathbf{N}$ between 0 en 31. The simple genetic algorithm representation would use bit-strings of length 5 with the obvious encoding, for example 11000 denoting 24. This defines the genotype space $\{0,1\}^5$, where the genetic search will take place. The fitness value of a given genotype (bit-string) is the square of the uniquely defined phenotype (integer) it represents.

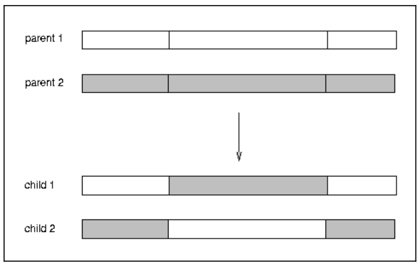

Figure 3 Example of 2-point crossover

Fitness proportional selection assigns selection probabilities to chromosomes proportionally with their fitness:

$$Prob(c_i) = f(x(c_i)) / \sum_{j=1}^{m} f((x(c_j)),$$

where m is the population size, $c_j \in \{1,\ldots,m\}$ denote the chromosomes, and $x(c_i)$ stands for the integer encoded by c_i. Then m independent drawings with replacement, based on these selection probabilities, are performed to select m chromosomes that will undergo crossover and mutation. The selected chromosomes form the mating pool. Note that this mechanism is biased, in the sense that chromosomes with a higher fitness get a higher selection probability and expectedly deliver more copies into the mating pool. After the mating pool has been established recombination, implemented by 1-point crossover, and mutation are executed.

For recombination, the mating pool is divided into randomly selected pairs and 1-point crossover is applied to each pair. An algorithm parameter called crossover rate p_c prescribes the probability of actually executing the operator on a given pair — with a chance of $1 - p_c$ it will not be performed and the two children are simply identical copies of the parents. When executing 1-point crossover, first a crossover point is selected randomly, telling after which bit position the strings will be crossed. Then the two strings are broken at this point and the tails are exchanged. A straightforward generalization of this operator is the n-point crossover that uses n crossover points and combines the resulting segments from the parents in an alternating fashion. Figure 3 illustrates this operator for $n = 2$.

The mutation operator traverses a given genotype to be mutated from left to right and at each position it flips the bit with probability p_m, the so-called mutation rate. Note that by the stochastic character of crossover and mutation four types of offspring can be generated: offspring that 'escaped' both operators and are identical to old chromosomes, offspring that result from application of one of the operators only, and offspring that is obtained by crossover and mutation. The last steps to execute in a simple genetic algorithm cycle are to determine the fitness of the newborn offspring and, based on the fitness values, to decide which of them can enter the population. Since the population size is kept constant, allowing a new chromosome is always at the cost of removing an old one. In this, the simple genetic algorithm is indeed very simple: the whole old population is deleted and each newly generated chromosome is allowed into the new one. This mechanism is called generational replacement. The Darwinian survival-of-the-fittest is thus somewhat implicit here, the simple genetic algorithm rather features mating-of-the-fittest.

Maximizing the function $f(x) = x^2$

Here we show the details of one selection-reproduction cycle on a simple (thus traceable) problem after Goldberg [16]. Table 3 shows a random initial population of four genotypes, the corresponding phenotypes, their fitness values and figures corresponding to fitness proportional selection. Table 4 exhibits the results of crossover on the given mating pool, together with the corresponding fitness values. In the table $\text{Prob}_i = f_i / \sum f_j$, the expected count after selection is $f_i/\bar{f}$ (displayed values are rounded up), and the actual count stands for the number of copies in the mating pool, that is to say, it shows the simulated outcomes of the drawings. Although manually engineered, this example shows a typical progress: the average fitness grows from 293 to 439 and the best fitness in the population from 576 to 729.

Other genetic algorithms

As mentioned earlier, there are many variants of the simple genetic algorithm, usually intended to fix one of its shortcomings. Various representations have been introduced to circumvent the limitations of using bit-strings. A natural extension is using real-valued vectors – an obvious choice for multi-dimensional ($\mathbf{R^n} \to \mathbf{R}$) optimization problems. Another commonly used option is using integer vectors, allowing a finite number of possible values at each position in the chromosomes. A specific case of this is the so called order-based representation, where chromosomes are permutations over a finite alphabet. Obviously, crossover and mutation operators must be adopted to new representations. For instance, in case of order-based representation, the operators must create permutations from permutations.

Another line of extending the simple genetic algorithm is to change the parent selection mechanism. A popular one is k-tournament selection, where k candidates are drawn randomly from the population with a uniform distribution, their fitness values are compared, and the best one winning the tournament gets selected. A clear advantage of this mechanism is that the selective pressure is scalable by the parameter k.

string nr.	initial population	x value	fitness $f(x) = x^2$	Prob_i	expected count	actual count
1	0 1 1 0 1	13	169	0.14	0.58	1
2	1 1 0 0 0	24	576	0.49	1.97	2
3	0 1 0 0 0	8	64	0.06	0.22	0
4	1 0 0 1 1	19	361	0.31	1.23	1
Sum			1170	1.00	4.00	4
Average			293	0.25	1.00	1
Max			576	0.49	1.97	2

Table 3 The x^2 example 1: initialization, evaluation, and selection.

string nr.	mating pool	crossover point	offspring after xover	x value	fitness $f(x) = x^2$
1	0 1 1 0 | 1	4	0 1 1 0 0	12	144
2	1 1 0 0 | 0	4	1 1 0 0 1	25	625
2	1 1 | 0 0 0	2	1 1 0 1 1	27	729
4	1 0 | 0 1 1	2	1 0 0 0 0	16	256
Sum					1754
Average					439
Max					729

Table 4 The x^2 example 2: crossover and offspring evaluation. Mutation is omitted for the sake of simplicity.

There are also variations of the replacement (survivor selection) mechanism. In the so-called steady-state genetic algorithms only a part of the whole population is replaced in one cycle. For instance, 2 new chromosomes are created after parent selection, crossover and mutation, and they are immediately reinserted in the population removing two old solutions. To choose the old ones to be replaced there are several options again.

Evolution strategies and self-adaptation

In this section we sketch evolution strategies. Hereby we present a second member of the evolutionary algorithm family and illustrate a very useful feature in evolutionary computing: self-adaptation. In general, self-adaptivity means that some parameters of the evolutionary algorithm are varied during a run in a specific manner: the parameters are included in the chromosomes and co-evolve with the solutions. This feature is inherent for evolution strategies, that is to say, from the earliest versions ESs are self-adaptive. During the last couple of years also other EAs are adopting self-adaptivity. The one glance summary of evolution strategie is given in table 5, detailed explanation is given in the text below.

Evolution strategies are typically used for continuous parameter optimization problems, that is to say, functions of the type $f : \mathbf{R^n} \to \mathbf{R}$, using real-valued vectors as candidate solutions (no

Representation	real-valued vectors
Recombination	discrete or intermediary
Mutation	Gaussian perturbation
Selection	uniform random
Replacement	(μ, λ) or $(\mu + \lambda)$
Speciality	self-adaptation of mutation step sizes

Table 5 Sketch of evolution strategie

encoding step needed). Parent selection is done by drawing individuals with a uniform distribution from the population of μ. Thus, unlike in GAs, there is no bias for quality here. Selective pressure comes from creating $\lambda > \mu$ offspring (very often μ/λ is about $1/7$). After creating λ offspring and calculating their fitness the best μ of them is chosen *deterministically* either from the offspring only, called (μ, λ) selection, or from the union of parents and offspring, called $(\mu + \lambda)$ selection. Recombination in evolution strategies is rather straightforward, two parent vectors $\bar{u}$ and $\bar{v}$ create one child $\bar{w}$, where:

$$w_i = \begin{cases} (u_i + v_i)/2 & \text{in case of intermediary recombination,} \\ u_i \text{ or } v_i \text{ chosen randomly} & \text{in case of discrete recombination.} \end{cases}$$

The so-called global variant of recombination is performed for an arbitrary number of parents. The mutation operator is based on a Gaussian distribution requiring two parameters: the mean, which is always set at zero, and the standard deviation σ, which is interpreted as the mutation step size. Mutations then are realized by replacing components of the vector $\vec{x}$ by

$$x_i' = x_i + N(0, \sigma),$$

where $N(0, \sigma)$ denotes a random number drawn from a Gaussian distribution with zero mean and standard deviation σ. By using a Gaussian distribution here, small mutations are more likely then large ones. The particular feature of mutation in evolution strategies is that the step-sizes are also included in the chromosomes, in the most general case one for each position $i \in \{1, \ldots, n\}$. A typical candidate is thus $\langle x_1, \ldots, x_n, \sigma_1, \ldots, \sigma_n \rangle$ and mutations are realized by replacing $\langle x_1, \ldots, x_n, \sigma_1, \ldots, \sigma_n \rangle$ by $\langle x_1', \ldots, x_n', \sigma_1', \ldots, \sigma_n' \rangle$, where

$$\sigma_i' = \sigma_i \cdot e^{N(0, \Delta\sigma)}, \qquad x_i' = x_i + N(0, \sigma_i'),$$

and $\Delta\sigma$ is a parameter of the method.

By this mechanism the mutation step sizes are not set by the user, they (the $\bar{\sigma}$ part) are co-evolving with the solutions (the $\bar{x}$ part). To this feature it is essential to modify the σ's first and mutate the x's with the new σ values. The rationale behind it is that an individual $\langle \bar{x}, \bar{\sigma} \rangle$ is evaluated twice. Primarily, it is evaluated directly for its viability during survivor selection based on $f(\bar{x})$. Secondarily, it is evaluated for its ability to create good offspring. This happens indirectly: a σ value evaluates favourably if the offspring generated by using it turns viable (in the first sense). Thus, an individual $\langle \bar{x}, \bar{\sigma} \rangle$ represents a good $\bar{x}$ that survived (μ, λ) or $(\mu + \lambda)$ selection and a good $\bar{\sigma}$ that proved successful in generating this good $\bar{x}$.

In general, modifying algorithm parameters during a run is motivated by the fact that the search process has different phases and a fixed parameter value might not be appropriate for each phase [9]. For instance, in the beginning of the search an exploration takes place, where the population is wide spread, locating promising areas in the search space. In this phase large leaps are appropriate. Later on the search becomes more focused, exploiting information gained by exploration. During this phase the population is concentrated around peaks on the fitness landscape and small variations are desirable. Self-adaptivity is a specific on-line parameter calibration technique, where the parameters (mutation step sizes in evolution strategies) are changed by the algorithm itself with only minimal influence from the user (for example, the value of $\Delta\sigma$). In fact, the algorithm is performing two tasks simultaneously, it is solving a given problem and it is calibrating itself for solving that problem. A particularly nice feature of evolutionary algorithms is their robustness – the evolution of the evolution works not only in carbon, but also in computer simulations.

What are evolutionary algorithms good for?

Evolutionary algorithms form a metaheuristic, applicable to a wide range of problems. They are not developed with a specific problem domain in mind and are also not particularly tailored for some application area. Therefore, it is hard to give a list of problems or problem types where they provably outperform alternative techniques. However, there is a widely shared view that in general evolutionary algorithms are very good on complex, ill-understood problems without an adequate analytical model or problems that do not have an optimal, analytical solution algorithm. In particular, an evolutionary algorithm is likely to succeed in comparison with other approaches if one or more of the following hold:

- the given problem has many parameters leading to a large search space,
- the problem has parameters of different types (for example, reals and integers),
- there are complex non-linear interactions between the parameters leading to a complex non-linear objective function,
- the objective function has many local optima,
- the objective function is changing over time,
- there is noise in the data hindering exact calculations.

Common wisdom within evolutionary computing also states that evolutionary algorithms are seldomly superior on problems that have been subjects of intensive study and have sophisticated solution methods, possibly with theoretical guarantees for an optimal solution. The same common wisdom says, however, that it costs only little effort (in comparison with those sophisticated methods: much less effort) to develop an evolutionary algorithm delivering an acceptable solution in acceptable running time. This is an often mentioned trade-off and it is widely believed that evolutionary algorithm solutions are very often very good, making them a serious alternative to other approaches. Citing an unknown evolutionary algorithm researcher: “An evolutionary algorithm is the second best algorithm for any problem”.

Systematic and thorough comparisons between evolutionary algorithms and other methods on (academic) benchmarks are scarce, but there are reported good results ranging from combinatorial problems, for example, graph coloring [12], to data mining, see [7], for instance. On the other hand, there are a lot of reports on successful applications, typically spread over journals and conference proceedings related to specific areas. A query on web page [24] in the subject area Computer Science using “genetic” as search term, restricting the search to the journal section and the year 2001 resulted in 366 hits (October 2001 figure). A very interesting account of the power of evolutionary computing is given in [19]. The authors set eight criteria to establish whether a computer program (an evolutionary algorithm, in this case) delivers human-competitive results. These criteria include:

1. The result was patented as an invention in the past, is an improvement over a patented invention, or would qualify today as a patentable new invention.
2. The result is equal to or better than a result that was accepted as a new scientific result at the time when it was published in a peer-reviewed scientific journal.

Using these criteria the authors show numerous instances where evolutionary computing has produced a result that is competitive with, and often better than, human performance, for instance in design of electrical circuits, computational molecular biology, quantum computing, or the automated creation of a checker player playing as well as 'class A' humans.

It is often claimed that an evolutionary algorithm is not an optimizer in the strict sense [8]. Rather, evolutionary computing – similarly to natural evolution – is a very good designer of complex structures that are well adapted to a given environment or task. The examples from [19] fit into this view and evolutionary design is forming a rapidly growing, successful application area [5–6]. One particular form of design is represented by evolutionary art, where the user interactively breeds pieces of art (pictures or music, for instance). In this creative symbiosis the evolutionary system is generating pieces of art and the user executes subjective selection. The selected pieces are then randomly recombined and mutated thus forming the next generation. A popular example, the Escher Evolver, has been on exhibition in the Gemeentemuseum in the Hague between May and October 2000 [11], see figure 2 for an image evolved during this exhibition.

Finally, let us mention that evolutionary systems are also frequently used for simulation purposes. In such applications the evolutionary process is not supposed to solve a problem, but serves as an underlying mechanism behind a certain phenomenon. Simulations are then ran in a what-if mode, where the emphasis lies on the emerging behavior and its dependencies on system parameters. Artificial life and evolutionary economy (at least certain branches of them) are two well-known example areas, sharing the view that survival-of-the-fittest is a major driving force in the animal world as well as in the arena of economic players.

Concluding remarks

We explained the basics of evolutionary computing providing the metaphor linking natural evolution to problem solving and illustrating the matter with two specific evolutionary algorithms. Evolutionary computing can be seen in a broader perspective of natural computing, an emerging discipline uniting various developments concerned with algorithms mimicking natural processes. Well-known other members of this family are neural networks, simulated annealing, and DNA computing. The power of this movement is rooted in the power of certain natural processes that can be envisioned as problem solving. With some exaggeration it can be said that Nature is the best general problem solver we know. It is thus nothing but natural (sic!) to try to adapt Nature's tricks for solving problems by computers. Therefore, it's safe to state that this movement is viable, and to expect that it will bring along further interesting developments.

References

1 T. Bäck, *Evolutionary Algorithms in Theory and Practice*, Oxford University Press, New York, 1996.

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

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

4 W. Banzhaf, P. Nordin, R.E. Keller, and F.D. Francone, *Genetic Porgramming: An Introduction*, Morgan Kaufmann, 1998.

5 P.J. Bentley, editor, *Evolutionary Design by Computers*, Morgan Kaufmann, 1999.

6 P.J. Bentley and D.W. Corne, editors, *Creative Evolutionary Systems*, Academic Press, 2002.

7 M. Brameier and W. Banzhaf, 'A comparison of linear genetic programming and neural networks in medical data mining', *IEEE Transactions on Evolutionary Computation*, 5(1), 17–26, 2001.

8 K. A. DeJong, 'Are genetic algorithms function optimizers?', In R. Männer and B. Manderick, editors, *Proceedings of the 2nd Conference on Parallel Problem Solving from Nature*, 3–13, North-Holland, 1992.

9 A.E. Eiben, R. Hinterding, and Z. Michalewicz, 'Parameter control in evolutionary algorithmsm', *IEEE Transactions on Evolutionary Computation*, 3(2), 124–141, 1999.

10 A.E. Eiben and Z. Michalewicz, editors, *Evolutionary Computation*, IOS Press, 1998.

11 A.E. Eiben, R. Nabuurs, and I. Booij, 'The Escher evolver: Evolution to the people', In P.J. Bentley and D.W. Corne, editors, *Creative Evolutionary Systems*, 425–439, Academic Press, 2001.

12 A.E. Eiben, J.K. van der Hauw, and J.I. van Hemert, 'Graph coloring with adaptive evolutionary algorithms', *Journal of Heuristics*, 4(1), 25–46, 1998.

13 D.B. Fogel, *Evolutionary Computation*, IEEE Press, 1995.

14 D.B. Fogel, editor, *Evolutionary Computation: the Fossile Record*, IEEE Press, 1998.

15 L.J. Fogel, A.J. Owens, and M.J. Walsh, *Artificial Intelligence through Simulated Evolution*, John Wiley, 1966.

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

17 J.H. Holland, *Adaption in natural and artificial systems*, MIT Press, 1992, First edition: 1975, The University of Michigan.

18 J.R. Koza, *Genetic Programming*, MIT Press, 1992.

19 J.R. Koza, M.A. Keane, J. Yu, F.H. Bennett, W. Mydlowec, 'Automatic creation of human-competitive programs and controllers by means of genetic programming', *Genetic Programming and Evolvable Machines*, 1(1/2):121–164, 2000.

20 Z. Michalewicz, *Genetic Algorithms + Data structures = Evolution programs*. Springer, Berlin, 3rd edition, 1996.

21 M. Mitchell, *An Introduction to Genetic Algorithms*. MIT Press, 1996.

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

23 H.-P. Schwefel, *Evolution and Optimum Seeking*, Wiley, New York, 1995.

24 http://www.sciencedirect.com/science