Evolutionary Algorithms

Zbigniew Michalewicz* Robert Hinterding†
Maciej Michalewicz‡

Abstract

Evolutionary algorithms (EAs), which are based on a powerful principle of evolution: survival of the fittest, and which model some natural phenomena: genetic inheritance and Darwinian strife for survival, constitute an interesting category of modern heuristic search. This introductory article presents the main paradigms of evolutionary algorithms (genetic algorithms, evolution strategies, evolutionary programming, genetic programming) and discusses other (hybrid) methods of evolutionary computation. We also discuss the ways an evolutionary algorithm can be tuned to the problem while it is solving the problem, as this can dramatically increase efficiency.

Evolutionary algorithms have been widely used in science and engineering for solving complex problems. An important goal of research on evolutionary algorithms is to understand the class of problems for which EAs are most suited, and, in particular, the class of problems on which they outperform other search algorithms.

1 Introduction

During the last two decades there has been a growing interest in algorithms which are based on the principle of evolution (survival of the fittest). A common term, accepted recently, refers to such techniques as *evolutionary computation* (EC) methods. The best known algorithms in this class include genetic algorithms, evolutionary programming, evolution strategies, and genetic programming. There are also many hybrid systems which incorporate various features of the above paradigms, and consequently are hard to classify; anyway, we refer to them just as EC methods.

*Department of Computer Science, University of North Carolina, Charlotte, NC 28223, USA, and Institute of Computer Science, Polish Academy of Sciences, ul. Ordona 21, 01-237 Warsaw, Poland

†Department of Computer and Mathematical Sciences, Victoria University of Technology, PO Box 14428 MMC, Melbourne 3000, Australia

‡Institute of Computer Science, Polish Academy of Sciences, ul. Ordona 21, 01-237 Warsaw, Poland

The field of evolutionary computation has reached a stage of some maturity. There are several, well established international conferences that attract hundreds of participants (International Conferences on Genetic Algorithms—ICGA [48, 50, 104, 12, 41, 27], Parallel Problem Solving from Nature—PPSN [112, 69, 14, 121], Annual Conferences on Evolutionary Programming—EP [35, 36, 113, 70, 37]); new annual conferences are getting started, e.g., IEEE International Conferences on Evolutionary Computation [91, 92, 93]. Also, there are many workshops, special sessions, and local conferences every year, all around the world. A relatively new journal, *Evolutionary Computation* (MIT Press) [21], is devoted entirely to evolutionary computation techniques; a new journal *IEEE Transactions on Evolutionary Computation* was just approved. many other journals organized special issues on evolutionary computation (e.g., [32, 74]). Many excellent tutorial papers [10, 11, 98, 122, 33] and technical reports provide more-or-less complete bibliographies of the field [1, 46, 103, 84]. There is also *The Hitch-Hiker's Guide to Evolutionary Computation* prepared initially by Jörg Heitkötter and currently by David Beasley [52], available on comp.ai.genetic interest group (Internet), and a new text, *Handbook of Evolutionary Computation*, is in its final stages of preparation [7].

In this introductory article we provide a general overview of the field. The next section provides a short introductory description of evolutionary algorithms. Section 3 discusses the paradigms of genetic algorithms, evolution strategies, evolutionary programming, and genetic programming, as well as some other evolutionary techniques. Section 4 provides with a discussion on one of the most interesting developments in the field: adaption of the algorithm to the problem, and section 5 concludes this article.

2 Evolutionary computation

In general, any abstract task to be accomplished can be thought of as solving a problem, which, in turn, can be perceived as a search through a space of potential solutions. Since usually we are after "the best" solution, we can view this task as an optimization process. For small spaces, classical exhaustive methods usually suffice; for larger spaces special artificial intelligence techniques must be employed. The methods of evolutionary computation are among such techniques; they are stochastic algorithms whose search methods model some natural phenomena: genetic inheritance and Darwinian strife for survival. As stated in [18]:

> "... the metaphor underlying genetic algorithms[1] is that of natural evolution. In evolution, the problem each species faces is one of searching for beneficial adaptations to a complicated and chang-

[1] The best known evolutionary computation techniques are genetic algorithms; very often the terms *evolutionary computation* methods and *GA-based* methods are used interchangeably.

ing environment. The 'knowledge' that each species has gained is embodied in the makeup of the chromosomes of its members."

As already mentioned in the Introduction, the best known techniques in the class of evolutionary computation methods are genetic algorithms, evolution strategies, evolutionary programming, and genetic programming. There are also many hybrid systems which incorporate various features of the above paradigms; however, the structure of any evolutionary computation algorithm is very much the same; a sample structure is shown in Figure 1.

```
procedure evolutionary algorithm
begin
    t ← 0
    initialize P(t)
    evaluate P(t)
    while (not termination-condition) do
    begin
        t ← t + 1
        select P(t) from P(t − 1)
        alter P(t)
        evaluate P(t)
    end
end
```

Figure 1: The structure of an evolutionary algorithm

The evolutionary algorithm maintains a population of individuals, $P(t) = \{x_1^t, \ldots, x_n^t\}$ for iteration t. Each individual represents a potential solution to the problem at hand, and is implemented as some data structure S. Each solution x_i^t is evaluated to give some measure of its "fitness". Then, a new population (iteration $t+1$) is formed by selecting the more fit individuals (select step). Some members of the new population undergo transformations (alter step) by means of "genetic" operators to form new solutions. There are unary transformations m_i (mutation type), which create new individuals by a small change in a single individual ($m_i : S \rightarrow S$), and higher order transformations c_j (crossover type), which create new individuals by combining parts from several (two or more) individuals ($c_j : S \times \ldots \times S \rightarrow S$).[2] After some number of generations the algorithm converges—it is hoped that the best individual represents a near-optimum (reasonable) solution.

Despite powerful similarities between various evolutionary computation techniques there are also many differences between them (often hidden on a lower level of abstraction). They use different data structures S for their chromoso-

[2] In most cases crossover involves just two parents, however, it need not be the case. In a recent study [25] the authors investigated the merits of 'orgies', where more than two parents are involved in the reproduction process. Also, scatter search techniques [42] proposed the use of multiple parents.

mal representations, consequently, the 'genetic' operators are different as well. They may or may not incorporate some other information (to control the search process) in their genes. There are also other differences; for example, the two lines of the Figure 1:

select $P(t)$ from $P(t-1)$
alter $P(t)$

can appear in the reverse order: in evolution strategies first the population is altered and later a new population is formed by a selection process (see section 3.2). Moreover, even within a particular technique there are many flavors and twists. For example, there are many methods for selecting individuals for survival and reproduction. These methods include (1) proportional selection, where the probability of selection is proportional to the individual's fitness, (2) ranking methods, where all individuals in a population are sorted from the best to the worst and probabilities of their selection are fixed for the whole evolution process,[3] and (3) tournament selection, where some number of individuals (usually two) compete for selection to the next generation: this competition (tournament) step is repeated population-size number of times. Within each of these categories there are further important details. Proportional selection may require the use of scaling windows or truncation methods, there are different ways for allocating probabilities in ranking methods (linear, nonlinear distributions), the size of a tournament plays a significant role in tournament selection methods. It is also important to decide on a generational policy. For example, it is possible to replace the whole population by a population of offspring, or it is possible to select the best individuals from two populations (population of parents and population of offspring)—this selection can be done in a deterministic or nondeterministic way. It is also possible to produce few (in particular, a single) offspring, which replace some (the worst?) individuals (systems based on such generational policy are called 'steady state'). Also, one can use an 'elitist' model which keeps the best individual from one generation to the next[4]; such model is very helpful for solving many kinds of optimization problems.

However, the data structure used for a particular problem together with a set of 'genetic' operators constitute the most essential components of any evolutionary algorithm. These are the key elements which allow us to distinguish between various paradigms of evolutionary methods. We discuss this issue in detail in the following section.

[3] For example, the probability of selection of the best individual is always 0.15 regardless its precise evaluation; the probability of selection of the second best individual is always 0.14, etc. The only requirements are that better individuals have larger probabilities and the total of these probabilities equals to one.

[4] It means, that if the best individual from a current generation is lost due to selection or genetic operators, the system force it into next generation anyway.

3 Main Paradigms of Evolutionary Computation

As indicated earlier, there are a few main paradigms of evolutionary computation techniques. In the following subsections we discuss them in turn; the discussion puts some emphasis on the data structures and genetic operators used by these techniques.

3.1 Genetic Algorithms

The beginnings of genetic algorithms can be traced back to the early 1950s when several biologists used computers for simulations of biological systems [43]. However, the work done in late 1960s and early 1970s at the University of Michigan under the direction of John Holland led to genetic algorithms as they are known today. A GA performs a multi-directional search by maintaining a population of potential solutions and encourages information formation and exchange between these directions.

Genetic algorithms (GAs) were devised to model *adaptation processes*, mainly operated on binary strings and used a recombination operator with mutation as a background operator [56]. Mutation flips a bit in a chromosome and crossover exchanges genetic material between two parents: if the parents are represented by five-bits strings, say $(0,0,0,0,0)$ and $(1,1,1,1,1)$, crossing the vectors after the second component would produce the offspring $(0,0,1,1,1)$ and $(1,1,0,0,0)$.[5] Fitness of an individual is assigned proportionally to the value of the objective function for the individual; individuals are selected for next generation on the basis of their fitness.

The combined effect of selection, crossover, and mutation gives so-called the reproductive schema growth equation [56]:

$$\xi(S,t+1) \geq \xi(S,t) \cdot eval(S,t)/\overline{F(t)} \left[1 - p_c \cdot \frac{\delta(S)}{m-1} - o(S) \cdot p_m\right],$$

where S is a schema defined over the alphabet of 3 symbols ('0', '1', and '$\star$' of length m; each schema represents all strings which match it on all positions other than '$\star$'); $\xi(S,t)$ denoted the number of strings in a population at the time t, matched by schema S; $\delta(S)$ is the defining length of the schema S — the distance between the first and the last fixed string positions; $o(S)$ denotes the order of the schema S — the number of 0 and 1 positions present in the schema; Another property of a schema is its *fitness* at time t, $eval(S,t)$ is defined as the average fitness of all strings in the population matched by the schema S; and $F(t)$ is the total fitness of the whole population at time t. Parameters p_c and p_m denote probabilities of crossover and mutation, respectively.

The above equation tells us about the expected number of strings matching a schema S in the next generation as a function of the actual number of strings matching the schema, the relative fitness of the schema, and its defining length

[5] This is an example of so-called 1-point crossover.

and order. Again, it is clear that above-average schemata with short defining length and low-order would still be sampled at exponentially increased rates.

The growth equation shows that selection increases the sampling rates of the above-average schemata, and that this change is exponential. The sampling itself does not introduce any new schemata (not represented in the initial $t = 0$ sampling). This is exactly why the crossover operator is introduced — to enable structured, yet random information exchange. Additionally, the mutation operator introduces greater variability into the population. The combined (disruptive) effect of these operators on a schema is not significant if the schema is short and low-order. The final result of the growth equation can be stated as:

> **Schema Theorem**: Short, low-order, above-average schemata receive exponentially increasing trials in subsequent generations of a genetic algorithm.

An immediate result of this theorem is that GAs explore the search space by short, low-order schemata which, subsequently, are used for information exchange during crossover:

> **Building Block Hypothesis**: A genetic algorithm seeks near-optimal performance through the juxtaposition of short, low-order, high-performance schemata, called the building blocks.

As stated in [43]:

> "Just as a child creates magnificent fortresses through the arrangement of simple blocks of wood, so does a genetic algorithm seek near optimal performance through the juxtaposition of short, low-order, high performance schemata."

A population of pop_size individuals of length m processes at least 2^m and at most 2^{pop_size} schemata. Some of them are processed in a useful manner: these are sampled at the (desirable) exponentially increasing rate, and are not disrupted by crossover and mutation (which may happen for long defining length and high-order schemata).

Holland [56] showed, that at least pop_size^3 of them are processed usefully — he has called this property an *implicit parallelism*, as it is obtained without any extra memory/processing requirements. It is interesting to note that in a population of pop_size strings there are many more than pop_size schemata represented. This constitutes possibly the only known example of a combinatorial explosion working to our advantage instead of our disadvantage.

To apply a GA to a particular problem, it is necessary to design a mapping between a space of potential solutions for the problem and a space of binary strings of some length. Sometimes it is not trivial task and quite often the process involved some additional heuristics (decoders, problem-specific operators, etc). For additional material on applications of genetic algorithms, see, for example, [72].

3.2 Evolution Strategies

Evolution strategies (ESs) were developed as a method to solve parameter optimization problems [109]; consequently, a chromosome represents an individual as a pair of float-valued vectors,[6] i.e., $\vec{v} = (\vec{x}, \vec{\sigma})$.

The earliest evolution strategies were based on a population consisting of one individual only. There was also only one genetic operator used in the evolution process: a mutation. However, the interesting idea (not present in GAs) was to represent an individual as a pair of float–valued vectors, i.e., $\vec{v} = (\vec{x}, \vec{\sigma})$. Here, the first vector $\vec{x}$ represents a point in the search space; the second vector $\vec{\sigma}$ is a vector of standard deviations: mutations are realized by replacing $\vec{x}$ by

$$\vec{x}^{t+1} = \vec{x}^t + N(0, \vec{\sigma}),$$

where $N(0, \vec{\sigma})$ is a vector of independent random Gaussian numbers with a mean of zero and standard deviations $\vec{\sigma}$. (This is in accordance with the biological observation that smaller changes occur more often than larger ones.) The offspring (the mutated individual) is accepted as a new member of the population (it replaces its parent) iff it has better fitness and all constraints (if any) are satisfied. For example, if f is the objective function without constraints to be maximized, an offspring $(\vec{x}^{t+1}, \vec{\sigma})$ replaces its parent $(\vec{x}^t, \vec{\sigma})$ iff $f(\vec{x}^{t+1}) > f(\vec{x}^t)$. Otherwise, the offspring is eliminated and the population remain unchanged.

The vector of standard deviations $\vec{\sigma}$ remains unchanged during the evolution process. If all components of this vector are identical, i.e., $\vec{\sigma} = (\sigma, \ldots, \sigma)$, and the optimization problem is *regular*[7], it is possible to prove the convergence theorem [8]:

> **Convergence Theorem**: For $\sigma > 0$ and a regular optimization problem with $f_{opt} > -\infty$ (minimalization) or $f_{opt} < \infty$ (maximization),
>
> $$p\{\lim_{t \to \infty} f(\vec{x}^t) = f_{opt}\} = 1$$
>
> holds.

The evolution strategies evolved further [109] to mature as

$(\mu + \lambda)$–ESs and (μ, λ)–ESs;

the main idea behind these strategies was to allow control parameters (like mutation variance) to self-adapt rather than changing their values by some deterministic algorithm.

---

[6] However, they started with integer variables as an experimental optimum-seeking method.

[7] An optimization problem is regular if the objective function f is continuous, the domain of the function is a closed set, for all $\epsilon > 0$ the set of all internal points of the domain for which the function differs from the optimal value less than ϵ is non-empty, and for all $\vec{x}_0$ the set of all points for which the function has values less than or equal to $f(\vec{x}_0)$ (for minimalization problems; for maximization problems the relationship is opposite) is a closed set.

In the $(\mu+\lambda)$–ES, μ individuals produce λ offspring. The new (temporary) population of $(\mu+\lambda)$ individuals is reduced by a selection process again to μ individuals. On the other hand, in the (μ,λ)–ES, the μ individuals produce λ offspring ($\lambda > \mu$) and the selection process selects a new population of μ individuals from the set of λ offspring only. By doing this, the life of each individual is limited to one generation. This allows the (μ,λ)–ES to perform better on problems with an optimum moving over time, or on problems where the objective function is noisy.

The operators used in the $(\mu+\lambda)$–ESs and (μ,λ)–ESs incorporate two-level learning: their control parameter $\vec{\sigma}$ is no longer constant, nor it is changed by some deterministic algorithm (like the 1/5 success rule), but it is incorporated in the structure of the individuals and undergoes the evolution process. To produce an offspring, the system acts in several stages:

- select two individuals,

  $$(\vec{x}^1,\vec{\sigma}^1) = ((x_1^1,\ldots,x_n^1),(\sigma_1^1,\ldots,\sigma_n^1)) \text{ and}$$
  $$(\vec{x}^2,\vec{\sigma}^2) = ((x_1^2,\ldots,x_n^2),(\sigma_1^2,\ldots,\sigma_n^2)),$$

  and apply a recombination (crossover) operator. There are two types of crossovers:

  - discrete, where the new offspring is

    $$(\vec{x},\vec{\sigma}) = ((x_1^{q_1},\ldots,x_n^{q_n}),(\sigma_1^{q_1},\ldots,\sigma_n^{q_n})),$$

    where $q_i = 1$ or $q_i = 2$ (so each component comes from the first or second preselected parent),
  - intermediate, where the new offspring is

    $$(\vec{x},\vec{\sigma}) = (((x_1^1+x_1^2)/2,\ldots,(x_n^1+x_n^2)/2),((\sigma_1^1+\sigma_1^2)/2,\ldots,(\sigma_n^1+\sigma_n^2)/2)).$$

  Each of these operators can be applied also in a global mode, where the new pair of parents is selected for *each* component of the offspring vector.

- apply mutation to the offspring $(\vec{x},\vec{\sigma})$ obtained; the resulting new offspring is $(\vec{x}',\vec{\sigma}')$, where

  $$\vec{\sigma}' = \vec{\sigma}\cdot e^{N(0,\Delta\vec{\sigma})}, \text{ and}$$
  $$\vec{x}' = \vec{x} + N(0,\vec{\sigma}'),$$

  where $\Delta\vec{\sigma}$ is a parameter of the method.

The best source of complete information (including recent results) on evolution strategies is recent Schwefel's text [111].

3.3 Evolutionary Programming

The original evolutionary programming (EP) techniques were developed by Lawrence Fogel [38]. They aimed at evolution of artificial intelligence in the sense of developing ability to predict changes in an environment. The environment was described as a sequence of symbols (from a finite alphabet) and the evolving algorithm supposed to produce, as an output, a new symbol. The output symbol should maximize the payoff function, which measures the accuracy of the prediction.

For example, we may consider a series of events, marked by symbols $a_1, a_2, \ldots$; an algorithm should predict the next (unknown) symbol, say a_{n+1} on the basis of the previous (known) symbols, $a_1, a_2, \ldots, a_n$. The idea of evolutionary programming was to evolve such an algorithm.

Finite state machines (FSM) were selected as a chromosomal representation of individuals; after all, finite state machines provide a meaningful representation of behavior based on interpretation of symbols. Figure 2 provides an example of a transition diagram of a simple finite state machine for a parity check. Such transition diagrams are directed graphs that contain a node for each state and edges that indicate the transition from one state to another, input and output values (notation a/b next to an edge leading from state S_1 to the state S_2 indicates that the input value of a, while the machine is in state S_1, results in output b and the next state S_2.

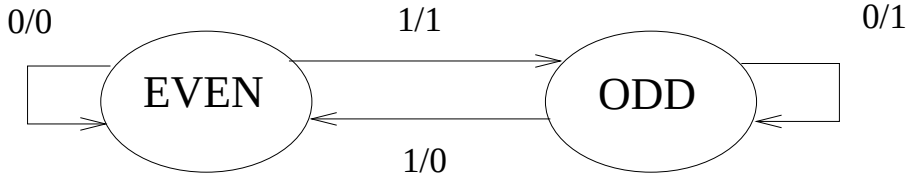

Figure 2: A FSM for a parity check

There are two states 'EVEN' and 'ODD' (machine starts in state 'EVEN'); the machine recognizes a parity of a binary string.

So, evolutionary programming technique maintains a population of finite state machines; each such individual represents a potential solution to the problem (i.e., represents a particular behavior). As already mentioned, each FSM is evaluated to give some measure of its "fitness". This is done in the following way: each FSM is exposed to the environment in the sense that it examines all previously seen symbols. For each subsequence, say, $a_1, a_2, \ldots, a_i$ it produces an output a'_{i+1}, which is compared with the next observed symbol, a_{i+1}. For example, if n symbols were seen so far, a FSM makes n predictions (one for each of the substrings a_1, a_1, a_2, and so on, until $a_1, a_2, \ldots, a_n$); the fitness function takes into account the overall performance (e.g., some weighted average of accuracy of all n predictions).

Like in evolution strategies, evolutionary programming technique first creates offspring and later selects individuals for the next generation. Each parent produces a single offspring; hence the size of the intermediate population doubles (like in (pop_size, pop_size)-ES). Offspring (a new FSMs) are created by random mutations of parent population (see Figure 3). There are five possible mutation operators: change of an output symbol, change of a state transition, addition of a state, deletion of a state, and change of the initial state (there are some additional constraints on the minimum and maximum number of states). These mutations are chosen with respect to some probability distribution (which can change during the evolutionary process); also it is possible to apply more than one mutation to a single parent (a decision on the number of mutations for a particular individual is made with respect to some other probability distribution).

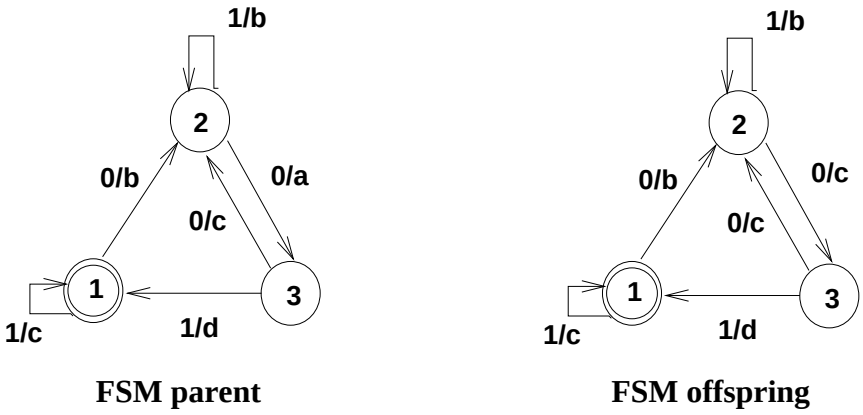

Figure 3: A FSM and its offspring. Machines start in state 1

The best pop_size individuals are retained for the next generation; i.e., to qualify for the next generation an individual should rank in the top 50% of the intermediate population. In original version [38] this process was iterated several times before the next output symbol was made available. Once a new symbol is available, it is added to the list of known symbols, and the whole process is repeated.

Of course, the above procedure can be extended in many way; as stated in [34]:

> "The payoff function can be arbitrarily complex and can posses temporal components; there is no requirement for the classical squared error criterion or any other smooth function. Further, it is not required that the predictions be made with a one-step look ahead. Forecasting can be accomplished at an arbitrary length of time into the future. Multivariate environments can be handled, and the en-

vironmental process need not be stationary because the simulated evolution will adapt to changes in the transition statistics."

Recently evolutionary programming techniques were generalized to handle numerical optimization problems; for details see [29] or [34]. For other examples of evolutionary programming techniques, see also [38] (classification of a sequence of integers into primes and nonprimes), [30] (for application of EP technique to the iterated prisoner's dilemma), as well as [35, 36, 113, 70] for many other applications.

3.4 Genetic Programming

Another interesting approach was developed relatively recently by Koza [64, 65]. Koza suggests that the desired program should evolve itself during the evolution process. In other words, instead of solving a problem, and instead of building an evolution program to solve the problem, we should rather search the space of possible computer programs for the best one (the most fit). Koza developed a new methodology, named Genetic Programming (GP), which provides a way to run such a search.

There are five major steps in using genetic programming for a particular problem. These are:

- selection of terminals,
- selection of a function,
- identification of the evaluation function,
- selection of parameters of the system, and
- selection of the termination condition.

It is important to note that the structure which undergoes evolution is a hierarchically structured computer program.[8] The search space is a hyperspace of valid programs, which can be viewed as a space of rooted trees. Each tree is composed of functions and terminals appropriate to the particular problem domain; the set of all functions and terminals is selected *a priori* in such a way that some of the composed trees yield a solution.

For example, two structures e_1 and e_2 (Figure 4) represent expressions $2x + 2.11$ and $x \cdot \sin(3.28)$, respectively. A possible offspring e_3 (after crossover of e_1 and e_2) represents $x \cdot \sin(2x)$.

The initial population is composed of such trees; construction of a (random) tree is straightforward. The evaluation function assigns a fitness value which evaluates the performance of a tree (program). The evaluation is based on a preselected set of test cases; in general, the evaluation function returns the sum

[8] Actually, Koza has chosen LISP's S-expressions for all his experiments. Currently, however, there are implementations of GP in C and other programming languages.

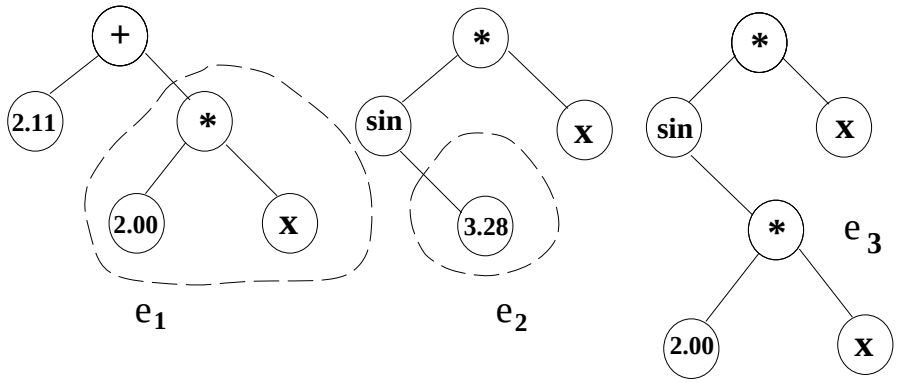

Figure 4: Expression e_3: an offspring of e_1 and e_2. Broken line includes areas being exchanged during the crossover operation

of distances between the correct and obtained results on all test cases. The selection is proportional; each tree has a probability of being selected to the next generation proportional to its fitness. The primary operator is a crossover that produces two offspring from two selected parents. The crossover creates offspring by exchanging subtrees between two parents. There are other operators as well: mutation, permutation, editing, and a define-building-block operation [64]. For example, a typical mutation selects a node in a tree and generates a new (random) subtree which originates in the selected node.

In addition to five major steps for building a genetic program for a particular problem, Koza [66] recently considered the advantages of adding an additional feature: a set of procedures. These procedures are called Automatically Defined Functions (ADF). It seems that this is an extremely useful concept for genetic programming techniques with its major contribution in the area of code reusability. ADFs discover and exploit the regularities, symmetries, similarities, patterns, and modularities of the problem at hand, and the final genetic program may call these procedures at different stages of its execution.

The fact that genetic programming operates on computer programs has a few interesting aspects. For example, the operators can be viewed also as programs, which can undergo a separate evolution during the run of the system. Additionally, a set of functions can consist of several programs which perform complex tasks; such functions can evolve further during the evolutionary run (e.g., ADF). Clearly, it is one of the most exiting areas of the current development in the evolutionary computation field with already a significant amount of experimental data (apart from [65] and [66], see also [63] and [3]).

3.5 Other techniques

Many researchers modified further evolutionary algorithms by 'adding' the problem specific knowledge to the algorithm. Several papers have discussed initialization techniques, different representations, decoding techniques (mapping from genetic representations to 'phenotypic' representations), and the use of heuristics for genetic operators. Davis [17] wrote (in the context of classical, binary GAs):

> "It has seemed true to me for some time that we cannot handle most real-world problems with binary representations and an operator set consisting only of binary crossover and binary mutation. One reason for this is that nearly every real-world domain has associated domain knowledge that is of use when one is considering a transformation of a solution in the domain [...] I believe that genetic algorithms are the appropriate algorithms to use in a great many real-world applications. I also believe that one should incorporate real-world knowledge in one's algorithm by adding it to one's decoder or by expanding one's operator set."

Such hybrid/nonstandard systems enjoy a significant popularity in evolutionary computation community. Very often these systems, extended by the problem-specific knowledge, outperform other classical evolutionary methods as well as other standard techniques [71, 72]. For example, a system Genetic-2N [71] constructed for the nonlinear transportation problem used a matrix representation for its chromosomes, a problem-specific mutation (main operator, used with probability 0.4) and arithmetical crossover (background operator, used with probability 0.05). It is hard to classify this system: it is not really a genetic algorithm, since it can run with mutation operator only without any significant decrease of quality of results. Moreover, all matrix entries are floating point numbers. It is not an evolution strategy, since it did not encode any control parameters in its chromosomal structures. Clearly, it has nothing to do with genetic programming and very little (matrix representation) with evolutionary programming approaches. It is just an evolutionary computation technique aimed at particular problem.

There are a few heuristics to guide a user in selection of appropriate data structures and operators for a particular problem. For numerical optimization problems it is generally best to use an evolution strategy or genetic algorithm with floating point representation as the reproduction operators are more suited to the representation and numerical problems, whereas other versions of genetic algorithms would be the best to handle combinatorial optimization problems. Genetic programs are great in discovery of rules given as a computer program, and evolutionary programming techniques can be used successfully to model a behavior of the system (e.g., prisoner dilemma problem). It seems also that neither of the evolutionary techniques is perfect (or even robust) across the problem spectrum; only the whole family of algorithms based on evolutionary

computation concepts (i.e., evolutionary algorithms) have this property of robustness. But the main key to successful applications is in heuristics methods, which are mixed skilfully with evolutionary techniques.

In the next section we discuss one of the most promising direction of evolutionary computation: adaption of the algorithm to the problem.

4 Adapting Algorithm to the Problem

As evolutionary algorithms (EAs) implement the idea of evolution, and as evolution itself must have evolved to reach its current state of sophistication, it is natural to expect adaption to be used in not only for finding solutions to a problem, but also for tuning the algorithm to the particular problem.

In EAs, not only do we need to choose the algorithm, representation and operators for the problem, but we also need to choose parameter values and operator probabilities for the evolutionary algorithm so that it will find the solution and, what is also important, find it efficiently. This is a time consuming task and a lot of effort has gone into automating this process. Researchers have used various ways of finding good values for the strategy parameters as these can affect the performance of the algorithm in a significantly. Many researchers experimented with problems from a particular domain, tuning the strategy parameters on the basis of such experimentation (tuning "by hand"). Later, they reported their results of applying a particular EA to a particular problem, stating:

> For these experiments, we have used the following parameters: population size = 80, probability of crossover = 0.7, etc.

without much justification of the choice made. Other researchers tried to modify the values of strategy parameters during the run of the algorithm; it is possible to do this by using some (possibly heuristic) rule, by taking feedback from the current state of the search, or by employing some self-adaptive mechanism. Note that these changes may effect a single component of a chromosome, the whole chromosome (individual), or even the whole population. Clearly, by changing these values while the algorithm is searching for the solution of the problem, further efficiencies can be gained.

Self-adaption, based on the evolution of evolution, was pioneered in Evolution Strategies to adapt mutation parameters to suit the problem during the run. The method was very successful in improving efficiency of the algorithm. This technique has been extended to other areas of evolutionary computation, but fixed representations, operators, and control parameters are still the norm.

Other research areas based on the inclusion of adapting mechanisms are:

- representation of individuals (as proposed by Shaefer [114]; the Dynamic Parameter Encoding technique, Schraudolph & Belew [108] and messy genetic algorithms, Goldberg et al. [45] also fall into this category).

- operators. It is clear that different operators play different roles at different stages of the evolutionary process. The operators should adapt (e.g., adaptive crossover Schaffer & Morishima [105], Spears [117]). This is true especially for time-varying fitness landscapes.

- control parameters. There have been various experiments aimed at adaptive probabilities of operators [17, 62, 118]. However, much more remains to be done.

The action of determining the variables and parameters of an EA to suit the problem has been termed *adapting the algorithm* to the problem, and in EAs this can be done while the algorithm is finding the problem solution.

In this section we provide with a comprehensive classification of adaption and give examples of their use. The classification is based on the mechanism of adaption and the level (in the EA) it occurs. We give classifications of adaption in Table 1; this classification is based on the mechanism of adaption (*adaption type*) and on which level inside the EA adaption occurs (*adaption level*). These classifications are orthogonal and encompass all forms of adaption within EAs. Angeline's classification [2] is from a different perspective and forms a subset of our classifications.

Type / *Level*	Static	Dynamic		
		Deterministic	Adaptive	Self-adaptive
Environment	S	E-D	E-A	E-SA
Population	S	P-D	P-A	P-SA
Individual	S	I-D	I-A	I-SA
Component	S	C-D	C-A	C-SA

Table 1: Classification of adaption in EAs

The *Type* of parameters' change consists of two main categories: static (no change) and dynamic, with the latter one divided further into deterministic (D), adaptive (A), and self-adaptive (SA) mechanisms. In the following section we discuss these types of adaption.

The *Level* of parameters' change consists of four categories: environment (E), population (P), individual (I), and component (C). These categories indicate the scope of the changed parameter; we discuss these types of adaption in section 4.2.

Whether examples are discussed in section 4.1 or in section 4.2 is completely arbitrary. An example of adaptive individual level adaption (I-A) could have been discussed in section 4.1 as an example of adaptive dynamic adaption or in section 4.2 as an example of individual level of adaption.

4.1 Types of Adaption

The classification of the *type of adaption* is made on the basis of the mechanism of adaption used in the process; in particular, attention is paid to the issue of whether feedback from the EA is used.

4.1.1 Static

Static adaption is where the strategy parameters have a constant value throughout the run of the EA. Consequently, an external agent or mechanism (e.g., a person or a program) is needed to tune the desired strategy parameters and choose the most appropriate values. This method is commonly used for most of the strategy parameters.

De Jong [20] put a lot of effort in finding parameter values which were good for a number of numeric test problems using a traditional GA. He determined experimentally recommended values for the probability of using single-point crossover and bit mutation. Grefenstette [49] used a GA as a meta-algorithm to optimize values for some parameter values.

4.1.2 Dynamic

Dynamic adaption happens if there is some mechanism which modifies a strategy parameter without external control. The class of EAs that use dynamic adaption can be sub-divided further into three classes where the *mechanism of adaption* is the criterion.

Deterministic

Deterministic dynamic adaption takes place if the value of a strategy parameter is altered by some deterministic rule; this rule modifies the strategy parameter deterministically without using any feedback from the EA. Usually, the rule will be used when a set number of generations have elapsed since the last time the rule was activated.

This method of adaption can be used to alter the probability of mutation so that the probability of mutation changes with the number of generations. For example:

$$mut\% = 0.5 + 0.3 \cdot \frac{g}{G},$$

where g is the generation number from $1 \ldots G$. Here the mutation probability $mut\%$ will increase from 0.5 to 0.8 as the number of generations increases to G.

This method of adaption was used also in defining a mutation operator for floating-point representations [72]: non-uniform mutation. For a parent $\vec{x}$, if the element x_k was selected for this mutation, the result is $\vec{x}' = (x_1, \ldots, x'_k, \ldots, x_n)$,

where

$$x'_k = \begin{cases} x_k + \triangle(t, right(k) - x_k) \\ \qquad \text{if a random binary digit is 0} \\ x_k - \triangle(t, x_k - left(k)) \\ \qquad \text{if a random binary digit is 1.} \end{cases}$$

The function $\triangle(t, y)$ returns a value in the range $[0, y]$ such that the probability of $\triangle(t, y)$ being close to 0 increases as t increases (t is the generation number). This property causes this operator to search the space uniformly initially (when t is small), and very locally at later stages.

Deterministic dynamic adaption was also used for changing the objective function of the problem; the point was to increase the penalties for violated constraints with evolution time [59, 75]. Joines & Houck used the following formula:

$$F(\vec{x}) = f(\vec{x}) + (C \times t)^{\alpha} \sum_{j=1}^{m} f_j^{\beta}(\vec{x}),$$

whereas Michalewicz and Attia experimented with

$$F(\vec{x}, \tau) = f(\vec{x}) + \frac{1}{2\tau} \sum_{j=1}^{m} f_j^2(\vec{x}).$$

In both cases, functions f_j measure the violation of the j-th constraint.

Eiben & Ruttkay [26] described an implementation of evolutionary algorithm for constraint satisfaction problems, where the penalty coefficients were increased after specified number of generations.

Adaptive

Adaptive dynamic adaption takes place if there is some form of feedback from the EA that is used to determine the direction and/or magnitude of the change to the strategy parameter. The assignment of the value of the strategy parameter may involve credit assignment, and the action of the EA may determine whether or not the new value persists or propagates throughout the population.

Early examples of this type of adaption include Rechenberg's '1/5 success rule', which was used to vary the step size of mutation [97]. This rule states that the ratio of successful mutations to all mutations should be 1/5, hence if the ratio is greater than 1/5 then decrease the step size, and if the ration is less than 1/5 then decrease the step size. Another example is Davis's 'adaptive operator fitness', which used feedback from the performance of reproduction operators to adjust their probability of being used [16].

Adaption was also used to change the objective function by increasing or decreasing penalty coefficients for violated constraints. For example, Bean & Hadj-Alouane [9] designed a penalty function where its one component takes a feedback from the search process. Each individual is evaluated by the formula:

$$F(\vec{x}) = f(\vec{x}) + \lambda(t) \sum_{j=1}^{m} f_j^2(\vec{x}),$$

where $\lambda(t)$ is updated every generation t in the following way:

$$\lambda(t+1) = \begin{cases} (1/\beta_1) \cdot \lambda(t), \\ \qquad \text{if } \vec{b}(i) \in \mathcal{F} \text{ for all} \\ \qquad t-k+1 \leq i \leq t \\ \beta_2 \cdot \lambda(t), \\ \qquad \text{if } \vec{b}(i) \in \mathcal{S} - \mathcal{F} \text{ for all} \\ \qquad t-k+1 \leq i \leq t \\ \lambda(t), \text{ otherwise,} \end{cases}$$

where $\vec{b}(i)$ denotes the best individual, in terms of function $eval$, in generation i, $\beta_1, \beta_2 > 1$ and $\beta_1 \neq \beta_2$ (to avoid cycling). In other words, the method (1) decreases the penalty component $\lambda(t+1)$ for the generation $t+1$, if all best individuals in the last k generations were feasible, and (2) 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, $\lambda(t+1)$ remains without change.

Other examples include adaption of probabilities of eight operators for adaptive planner/navigator [125], where the feedback from the evolutionary process includes, through the operator performance index, effectiveness of operators in improving the fitness of a path, their operation time, and their side effect to future generations.

Self-adaptive

The idea of the evolution of evolution can be used to implement the self-adaption of parameters. Here the parameters to be adapted are encoded onto the chromosome(s) of the individual and undergo mutation and recombination. These encoded parameters do not affect the fitness of individuals directly, but "better" values will lead to "better" individuals and these individuals will be more likely to survive and product offspring and hence propagate these "better" parameter values.

Schwefel [110, 111] pioneered this method to self-adapt the mutation step size and the mutation rotation angles in Evolution Strategies. Self-adaption was extended to EP by Fogel et al. [31] and to GAs by Bäck [6] and Hinterding [53].

The parameters to self adapt can be parameter values or probabilities of using alternative processes, and as these are numeric quantities this type of self-adaption has been used mainly for the optimization of numeric functions. This has been the case when single chromosome representations are used (which is the overwhelming case), as otherwise numerical and non-numerical representations would need to be combined on the same chromosome. Examples of self-adaption for non-numerical problems are Fogel et al. [40] where they self-adapted the relative probabilities of five mutation operators for the components of a finite state machine. The other example is Hinterding [55], where a multi-chromosome

GA is used to implement the self-adaption in the Cutting Stock Problem with contiguity. Here self-adaption is used to adapt the probability of using one of the two available mutation operators, and the strength of the group mutation operator.

4.2 Levels of Adaption

We can also define at what level within the EA and the solution representation adaption takes place. We define four levels: environment, population, individual and component. These levels of adaption can be used with each of the types of adaption, and a mixture of levels and types of adaption can be used within an EA.

4.2.1 Environment Level Adaption

Environment level adaption is where the response of the environment to the individual is changed. This covers cases such as when the penalties in the fitness function change, where weights within the fitness function change and the fitness of an individual changes in response to niching considerations (some of these were discussed in the previous section, in the context of types of adaption).

Darwen & Yao [19], explore both deterministic and adaptive environmental adaption in their paper comparing fitness sharing methods.

4.2.2 Population Level Adaption

In EAs some (or all in simple EAs) of the parameters are global, modifying these parameters when they apply to all members of the population is population level adaption.

Dynamic adaption of these parameters is in most cases deterministic or adaptive. No cases of population level self-adaption have been seen yet. The example mutation rate adaption in the section on deterministic adaption is deterministic population level adaption, and Rechenberg's '1/5 success rule' is an example of adaptive population level adaption.

Population level adaption also covers cases where a number of populations are used in a parallel EA or otherwise, Lis [68] uses feedback from a number of parallel populations to dynamically adapt the mutation rate. She uses feedback from a number of parallel populations running with different mutation probabilities to adjust the mutation probabilities of all the populations up or down. Schlierkamp-Voosen & Mühlenbein [106] uses competition between sub-populations to determine which populations will lose or gain individuals. Hinterding et al. [54] uses feedback from three sub-populations with different population sizes to adaptively change some or all of the sub-population sizes.

4.2.3 Individual Level Adaption

Individual-level adaption adjusts strategy parameters held within individuals and whose value affects only that individual. Examples are: the adaption of the mutation step size in ESs, EP, and GAs; the adaption of crossover points in GAs [105].

In [4] there is a description of a method for adapting population size by defining age of individuals; the size of the population after single iteration is

$$PopSize(t+1) = PopSize(t) + N(t) - D(t),$$

where $D(t)$ is the number of chromosomes which die off during generation t and $N(t)$ is the number of offspring produced during the generation t (for details, see [72]). The number of produced offspring $N(t)$ is proportional to the size of the population at given generation t, whereas the number of individuals "to die" $D(t)$ depends on age of individual chromosomes. There are several heuristics one can use for the age allocation for individuals [4]; all of them require a feedback from the current state of the search.

4.2.4 Component Level Adaption

Component-level adaption adjusts strategy parameters local to some component or gene of an individual in the population. The best known example of component level adaption is the self-adaption of component level mutation step sizes and rotation angles in ESs.

Additionally, in [40] the mechanism of adapting probabilities of mutation for each component of a finite states machine is discussed.

4.3 Combining forms of adaption

The classic example of combining forms of adaption is in ESs, where the algorithm can be configured for individual level adaption (one mutation step size per individual), component level adaption (one mutation step size per component) or with two types of component level adaption where both the mutation step size and rotation angle is self-adapted for individual components [110].

Hinterding et al. [54] combine global level adaption of the population size with individual level self-adaption of the mutation step size for optimizing numeric functions.

Combining forms of adaption has not been used much as the interactions are complex, hence deterministic or adaptive rules will be difficult to work out. But self-adaption where we use evolution to determine the beneficial interactions (as in finding solutions to problems) would seem to be the best approach.

5 Discussion

The effectiveness of evolutionary computations depend on the representation used for the problem solutions, the reproduction operators used and the con-

figuration of the evolutionary algorithm used.

Adaption gives us the opportunity to customize the evolutionary algorithm to the problem and to modify the configuration and the strategy parameters used while the problem solution is sought. This enables us not only to incorporate domain information and multiple reproduction operators into the EA more easily, but can allow the algorithm itself to select those values and operators which give better results. Also these values can be modified during the run of the EA to suit the situation during that part of the run.

Although evolutionary algorithms have been successfully applied to many practical problems, there have been a number of failures as well, and there is little understanding of what features of these domains make them appropriate or inappropriate for these algorithms. Three important claims have been made about why evolutionary algorithms perform well: (1) independent sampling is provided by populations of candidate solutions, (2) selection is a mechanism that preserves good solutions, and (3) partial solutions can be efficiently modified and combined through various 'genetic' operators.

References

[1] Alander, J.T., *An Indexed Bibliography of Genetic Algorithms: Years 1957–1993*, Department of Information Technology and Production Economics, University of Vaasa, Finland, Report Series No.94-1, 1994.

[2] Angeline, P.J., *Adaptive and Self-Adaptive Evolutionary Computation*, in Palaniswami, M., Attikiouzel, Y., Marks, R.J.II, Fogel, D., & Fukuda, T. (Eds), *Computational Intelligence, A Dynamic System Perspective*, IEEE Press, pp.152–161, 1995.

[3] Angeline, P.J. and Kinnear, K.E. (Editors), *Advances in Genetic Programming II*, MIT Press, Cambridge, MA, 1996.

[4] Arabas, J., Michalewicz, Z., and Mulawka, J., *GAVaPS — a Genetic Algorithm with Varying Population Size*, in [91].

[5] Bäck, T., and Hoffmeister, F., *Extended Selection Mechanisms in Genetic Algorithms*, in [12], pp.92–99.

[6] Bäck, T., *Self-adaption in Genetic Algorithms*, Proceedings of the First European Conference on Artificial Life, pp.263–271, 1992.

[7] Bäck, T., Fogel, D., and Michalewicz, Z. (Editors), *Handbook of Evolutionary Computation*, Oxford University Press, New York, 1996.

[8] Bäck, T., Hoffmeister, F., and Schwefel, H.-P., *A Survey of Evolution Strategies*, in [12], pp.2–9.

[9] Bean, J.C. and Hadj-Alouane, A.B., *A Dual Genetic Algorithm for Bounded Integer Programs*, Department of Industrial and Operations Engineering, The University of Michigan, TR 92-53, 1992.

[10] Beasley, D., Bull, D.R., and Martin, R.R., *An Overview of Genetic Algorithms: Part 1, Foundations*, University Computing, Vol.15, No.2, pp.58–69, 1993.

[11] Beasley, D., Bull, D.R., and Martin, R.R., *An Overview of Genetic Algorithms: Part 2, Research Topics*, University Computing, Vol.15, No.4, pp.170–181, 1993.

[12] Belew, R. and Booker, L. (Editors), Proceedings of the Fourth International Conference on Genetic Algorithms, Morgan Kaufmann Publishers, Los Altos, CA, 1991.

[13] Brooke, A., Kendrick, D., and Meeraus, A., *GAMS: A User's Guide*, The Scientific Press, 1988.

[14] Davidor, Y., Schwefel, H.-P., and Männer, R. (Editors), Proceedings of the Third International Conference on Parallel Problem Solving from Nature (PPSN), Springer-Verlag, New York, 1994.

[15] Davis, L., (Editor), *Genetic Algorithms and Simulated Annealing*, Morgan Kaufmann Publishers, Los Altos, CA, 1987.

[16] Davis, L., *Handbook of Genetic Algorithms*, New York, Van Nostrand Reinhold, 1991.

[17] Davis, L., *Adapting Operator Probabilities in Genetic Algorithms*, in [104], pp.61–69.

[18] Davis, L. and Steenstrup, M., *Genetic Algorithms and Simulated Annealing: An Overview*, in [15], pp.1–11.

[19] Darwen, P and Yao, X., *Every Niching Method has its Niche: Fitness sharing and Implicit Sharing Compared*, in [121], pp.398–407.

[20] De Jong, K.A., "An Analysis of the Behavior of a Class of Genetic Adaptive Systems", (Doctoral dissertation, University of Michigan), *Dissertation Abstract International*, 36(10), 5140B. (University Microfilms No 76-9381).

[21] De Jong, K.A., (Editor), *Evolutionary Computation*, MIT Press, 1993.

[22] De Jong, K., *Genetic Algorithms: A 10 Year Perspective*, in [48], pp.169–177.

[23] De Jong, K., *Genetic Algorithms: A 25 Year Perspective*, in [126], pp.125–134.

[24] Dhar, V. and Ranganathan, N., *Integer Programming vs. Expert Systems: An Experimental Comparison*, Communications of ACM, Vol.33, No.3, pp.323–336, 1990.

[25] Eiben, A.E., Raue, P.-E., and Ruttkay, Zs., *Genetic Algorithms with Multi-parent Recombination*, in [14], pp.78–87.

[26] Eiben, A.E. and Ruttkay, Zs., *Self-adaptivity for Constraint Satisfaction: Learning Penalty Functions*, in [93], pp.258–261.

[27] Eshelman, L.J., (Editor), Proceedings of the Sixth International Conference on Genetic Algorithms, Morgan Kaufmann, San Mateo, CA, 1995.

[28] Eshelman, L.J. and Schaffer, J.D., *Preventing Premature Convergence in Genetic Algorithms by Preventing Incest*, in [12], pp.115–122.

[29] Fogel, D.B., *Evolving Artificial Intelligence*, Ph.D. Thesis, University of California, San Diego, 1992.

[30] Fogel, D.B., *Evolving Behaviours in the Iterated Prisoner's Dilemma*, Evolutionary Computation, Vol.1, No.1, pp.77–97, 1993.

[31] Fogel, D.B., Fogel, L.J. and Atmar, J.W. *Meta-Evolutionary Programming*, Informatica, Vol.18, No.4, pp.387–398, 1994.

[32] Fogel, D.B. (Editor), IEEE Transactions on Neural Networks, special issue on Evolutionary Computation, Vol.5, No.1, 1994.

[33] Fogel, D.B., *An Introduction to Simulated Evolutionary Optimization*, IEEE Transactions on Neural Networks, special issue on Evolutionary Computation, Vol.5, No.1, 1994.

[34] Fogel, D.B., *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*, IEEE Press, Piscataway, NJ, 1995.

[35] Fogel, D.B. and Atmar, W., *Proceedings of the First Annual Conference on Evolutionary Programming*, La Jolla, CA, 1992, Evolutionary Programming Society.

[36] Fogel, D.B. and Atmar, W., *Proceedings of the Second Annual Conference on Evolutionary Programming*, La Jolla, CA, 1993, Evolutionary Programming Society.

[37] Fogel, L.J., Angeline, P.J., Bäck, T. (Editors), Proceedings of the Fifth Annual Conference on Evolutionary Programming, The MIT Press, 1996.

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

[39] Fogel, L.J., *Evolutionary Programming in Perspective: The Top-Down View*, in [126], pp.135–146.

[40] Fogel, L.J., Angeline, P.J. and Fogel, D.B. *An Evolutionary Programming Approach to Self-Adaption on Finite State Machines*, in [70], pp.355–365.

[41] Forrest, S. (Editor), Proceedings of the Fifth International Conference on Genetic Algorithms, Morgan Kaufmann Publishers, Los Altos, CA, 1993.

[42] Glover, F., *Heuristics for Integer Programming Using Surrogate Constraints*, Decision Sciences, Vol.8, No.1, pp.156–166, 1977.

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

[44] Goldberg, D.E.., *Simple Genetic Algorithms and the Minimal, Deceptive Problem*, in [15], pp.74–88.

[45] Goldberg, D.E., Deb, K., and Korb, B., *Do not Worry, Be Messy*, in [12], pp.24–30.

[46] Goldberg, D.E., Milman, K., and Tidd, C., *Genetic Algorithms: A Bibliography*, IlliGAL Technical Report 92008, 1992.

[47] Gorges-Schleuter, M., *ASPARAGOS An Asynchronous Parallel Genetic Optimization Strategy*, in [104], pp.422–427.

[48] Grefenstette, J.J., (Editor), Proceedings of the First International Conference on Genetic Algorithms, Lawrence Erlbaum Associates, Hillsdale, NJ, 1985.

[49] Grefenstette, J.J., *Optimization of Control Parameters for Genetic Algorithms*, IEEE Transactions on Systems, Man, and Cybernetics, Vol. 16, No.1, pp.122–128, 1986.

[50] Grefenstette, J.J., (Editor), Proceedings of the Second International Conference on Genetic Algorithms, Lawrence Erlbaum Associates, Hillsdale, NJ, 1987.

[51] Hadj-Alouane, A.B. and Bean, J.C., *A Genetic Algorithm for the Multiple-Choice Integer Program*, Department of Industrial and Operations Engineering, The University of Michigan, TR 92-50, 1992.

[52] Heitkötter, J., (Editor), *The Hitch-Hiker's Guide to Evolutionary Computation*, FAQ in comp.ai.genetic, issue 1.10, 20 December 1993.

[53] Hinterding, R., *Gaussian Mutation and Self-adaption in Numeric Genetic Algorithms*, in [91], pp.384–389.

[54] Hinterding, R., Michalewicz, Z. and Peachey, T.C., *Self-Adaptive Genetic Algorithm for Numeric Functions*, in [121], pp.420–429.

[55] Hinterding, R., *Self-adaption using Multi-chromosomes*, Submitted to: 1997 IEEE International Conference on Evolutionary Computation, 1996.

[56] Holland, J.H., *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor, 1975.

[57] Holland, J.H., *Royal Road Functions*, Genetic Algorithm Digest, Vol.7, No.22, 12 August 1993.

[58] Homaifar, A., Lai, S. H.-Y., Qi, X., *Constrained Optimization via Genetic Algorithms*, Simulation, Vol.62, No.4, 1994, pp.242–254.

[59] Joines, J.A. and Houck, C.R., *On the Use of Non-Stationary Penalty Functions to Solve Nonlinear Constrained Optimization Problems With GAs*, in [91], pp.579–584.

[60] Jones, T., *A Description of Holland's Royal Road Function*, Evolutionary Computation, Vol.2, No.4, 1994, pp.409–415.

[61] Jones, T. and Forrest, S., *Fitness Distance Correlation as a Measure of Problem Difficulty for Genetic Algorithms*, in [27], pp.184–192.

[62] Julstrom, B.A., *What Have You Done for Me Lately? Adapting Operator Probabilities in a Steady-State Genetic Algorithm*, in [27], pp.81–87.

[63] Kinnear, K.E. (Editor), *Advances in Genetic Programming*, MIT Press, Cambridge, MA, 1994.

[64] Koza, J.R., *Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems*, Report No. STAN–CS–90–1314, Stanford University, 1990.

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

[66] Koza, J.R., *Genetic Programming – 2*, MIT Press, Cambridge, MA, 1994.

[67] Le Riche, R., Knopf-Lenoir, C., and Haftka, R.T., *A Segregated Genetic Algorithm for Constrained Structural Optimization*, in [27], pp.558–565.

[68] Lis, J., *Parallel Genetic Algorithm with Dynamic Control Parameter*, in [93], pp.324–329.

[69] Männer, R. and Manderick, B. (Editors), Proceedings of the Second International Conference on Parallel Problem Solving from Nature (PPSN), North-Holland, Elsevier Science Publishers, Amsterdam, 1992.

[70] McDonnell, J.R., Reynolds, R.G., and Fogel, D.B. (Editors), Proceedings of the Fourth Annual Conference on Evolutionary Programming, The MIT Press, 1995.

[71] Michalewicz, Z., *A Hierarchy of Evolution Programs: An Experimental Study*, Evolutionary Computation, Vol.1, No.1, 1993, pp.51–76.

[72] Michalewicz, Z., *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlag, 3rd edition, 1996.

[73] Michalewicz, Z., *Heuristic Methods for Evolutionary Computation Techniques*, Journal of Heuristics, Vol.1, No.2, 1995, pp.177-206.

[74] Michalewicz, Z. (Editor), Statistics & Computing, special issue on evolutionary computation, Vol.4, No.2, 1994.

[75] Michalewicz, Z., and Attia, N., *Evolutionary Optimization of Constrained Problems*, in [113], pp.98–108.

[76] Michalewicz, Z., Dasgupta, D., Le Riche, R.G., and Schoenauer, M., *Evolutionary Algorithms for Constrained Engineering Problems*, Computers & Industrial Engineering Journal, Vol.30, No.4, September 1996, pp.851–870.

[77] Michalewicz, Z. and Nazhiyath, G., *Genocop III: A Co-evolutionary Algorithm for Numerical Optimization Problems with Nonlinear Constraints*, in [92], pp.647–651.

[78] Michalewicz, Z. and Schoenauer, M., *Evolutionary Algorithms for Constrained Parameter Optimization Problems*, Evolutionary Computation, Vol.4, No.1, 1996.

[79] Michalewicz, Z., Vignaux, G.A., and Hobbs, M., *A Non-Standard Genetic Algorithm for the Nonlinear Transportation Problem*, ORSA Journal on Computing, Vol.3, No.4, 1991, pp.307–316.

[80] Michalewicz, Z. and Xiao, J., *Evaluation of Paths in Evolutionary Planner/Navigator*, Proceedings of the 1995 International Workshop on Biologically Inspired Evolutionary Systems, Tokyo, Japan, May 30–31, 1995, pp.45–52.

[81] Mühlenbein, H., *Parallel Genetic Algorithms, Population Genetics and Combinatorial Optimization*, in [104], pp.416-421.

[82] Mühlenbein, H. and Schlierkamp-Vosen, D., *Predictive Models for the Breeder Genetic Algorithm*, Evolutionary Computation, Vol.1, No.1, pp.25–49, 1993.

[83] Nadhamuni, P.V.R., *Application of Co-evolutionary Genetic Algorithm to a Game*, Master Thesis, Department of Computer Science, University of North Carolina, Charlotte, 1995.

[84] Nissen, V., *Evolutionary Algorithms in Management Science: An Overview and List of References*, European Study Group for Evolutionary Economics, 1993.

[85] Orvosh, D. and Davis, L., *Shall We Repair? Genetic Algorithms, Combinatorial Optimization, and Feasibility Constraints*, in [41], p.650.

[86] Palmer, C.C. and Kershenbaum, A., *Representing Trees in Genetic Algorithms*, in [91], pp.379–384.

[87] Paredis, J., *Genetic State-Space Search for Constrained Optimization Problems*, Proceedings of the Thirteen International Joint Conference on Artificial Intelligence, Morgan Kaufmann, San Mateo, CA, 1993.

[88] Paredis, J., *Co-evolutionary Constraint Satisfaction*, in [14], pp.46–55.

[89] Powell, D. and Skolnick, M.M., *Using Genetic Algorithms in Engineering Design Optimization with Non-linear Constraints*, in [41], pp.424–430.

[90] Potter, M. and De Jong, K., *A Cooperative Coevolutionary Approach to Function Optimization*, George Mason University, 1994.

[91] Proceedings of the First IEEE International Conference on Evolutionary Computation, Orlando, 26 June – 2 July, 1994.

[92] Proceedings of the Second IEEE International Conference on Evolutionary Computation, Perth, 29 November – 1 December, 1995.

[93] Proceedings of the Third IEEE International Conference on Evolutionary Computation, Nagoya, 18–22 May, 1996.

[94] Radcliffe, N.J., *Forma Analysis and Random Respectful Recombination*, in [12], pp.222–229.

[95] Radcliffe, N.J., *Genetic Set Recombination*, in [124], pp.203–219.

[96] Radcliffe, N.J., and George, F.A.W., *A Study in Set Recombination*, in [41], pp.23–30.

[97] Rechenberg, R., *Evolutionsstrategie: Optimierung technischer Syseme nach Prinzipien der biologischen Evolution*, Frommann-Holzboog, Stuttgart, 1973.

[98] Reeves, C.R., *Modern Heuristic Techniques for Combinatorial Problems*, Blackwell Scientific Publications, London, 1993.

[99] Reynolds, R.G., *An Introduction to Cultural Algorithms*, in [113], pp.131–139.

[100] Reynolds, R.G., Michalewicz, Z., and Cavaretta, M., *Using Cultural Algorithms for Constraint Handling in Genocop*, in [70], pp.289–305.

[101] Richardson, J.T., Palmer, M.R., Liepins, G., and Hilliard, M., *Some Guidelines for Genetic Algorithms with Penalty Functions*, in [104], pp.191–197.

[102] Ronald, E., *When Selection Meets Seduction*, in [27], pp.167–173.

[103] Saravanan, N. and Fogel, D.B., *A Bibliography of Evolutionary Computation & Applications*, Department of Mechanical Engineering, Florida Atlantic University, Technical Report No. FAU-ME-93-100, 1993.

[104] Schaffer, J., (Editor), Proceedings of the Third International Conference on Genetic Algorithms, Morgan Kaufmann Publishers, Los Altos, CA, 1989.

[105] Schaffer, J.D. and Morishima, A., *An Adaptive Crossover Distribution Mechanism for Genetic Algorithms*, in [50], pp.36–40.

[106] Schlierkamp-Voosen, D. and Mühlenbein, H., *Adaption of Population Sizes by Competing Subpopulations*, in [93], pp.330–335.

[107] Schoenauer, M., and Xanthakis, S., *Constrained GA Optimization*, in [41], pp.573–580.

[108] Schraudolph, N. and Belew, R., *Dynamic Parameter Encoding for Genetic Algorithms*, CSE Technical Report #CS90–175, University of San Diego, La Jolla, 1990.

[109] Schwefel, H.-P., *On the Evolution of Evolutionary Computation*, in [126], pp.116–124.

[110] Schwefel, H.-P., *Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie*, Interdisciplinary systems research, Vol.26, Birhäuser, Basel, 1977.

[111] Schwefel, H.-P., *Evolution and Optimum Seeking*, John Wiley, Chichester, UK, 1995.

[112] Schwefel, H.-P. and Männer, R. (Editors), Proceedings of the First International Conference on Parallel Problem Solving from Nature (PPSN), Springer-Verlag, Lecture Notes in Computer Science, Vol.496, 1991.

[113] Sebald, A.V. and Fogel, L.J., *Proceedings of the Third Annual Conference on Evolutionary Programming*, San Diego, CA, 1994, World Scientific.

[114] Shaefer, C.G., *The ARGOT Strategy: Adaptive Representation Genetic Optimizer Technique*, in [50], pp.50–55.

[115] Siedlecki, W. and Sklanski, J., *Constrained Genetic Optimization via Dynamic Reward–Penalty Balancing and Its Use in Pattern Recognition*, in [104], pp.141–150.

[116] Smith, A. and Tate, D., *Genetic Optimization Using A Penalty Function*, in [41], pp.499–503.

[117] Spears, W.M., *Adapting Crossover in Evolutionary Algorithms*, in [70], pp.367–384.

[118] Srinivas, M. and Patnaik, L.M., *Adaptive Probabilities of Crossover and Mutation in Genetic Algorithms*, IEEE Transactions on Systems, Man, and Cybernetics, Vol.24, No.4, 1994, pp.17–26.

[119] Surry, P.D., N.J. Radcliffe, and I.D. Boyd, *A Multi-objective Approach to Constrained Optimization of Gas Supply Networks*. Presented at the AISB-95 Workshop on Evolutionary Computing, Sheffield, UK, April 3–4, 1995, pp.166–180.

[120] Vignaux, G.A., and Michalewicz, Z., *A Genetic Algorithm for the Linear Transportation Problem*, IEEE Transactions on Systems, Man, and Cybernetics, Vol.21, No.2, 1991, pp.445–452.

[121] Voigt, H.-M., Ebeling, W., Rechenberg, I., Schwefel, H.-P. (Editors), Proceedings of the Fourth International Conference on Parallel Problem Solving from Nature (PPSN), Springer-Verlag, New York, 1996.

[122] Whitley, D., *Genetic Algorithms: A Tutorial*, in [74], pp.65–85.

[123] Whitley, D., *GENITOR II: A Distributed Genetic Algorithm*, Journal of Experimental and Theoretical Artificial Intelligence, Vol.2, pp.189–214.

[124] Whitley, D. (Editor), *Foundations of Genetic Algorithms–2*, Second Workshop on the Foundations of Genetic Algorithms and Classifier Systems, Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[125] Xiao, J., Michalewicz, Z. and Zhang, L *Evolutionary Planner/Navigator: Operator Performance and Self-Tuning*, in [93], pp.366–371.

[126] Zurada, J., Marks, R., and Robinson, C. (Editors), *Computational Intelligence: Imitating Life*, IEEE Press, 1994.