Appl. Math. and Comp. Sci., 1996, vol.6, No.3, 393–413

EVOLUTIONARY COMPUTATION: MAIN PARADIGMS AND CURRENT DIRECTIONS

ZBIGNIEW MICHALEWICZ*, MACIEJ MICHALEWICZ**

Evolutionary computation techniques, which are based on a powerful principle of evolution: survival of the fittest, constitute an interesting category of heuristic search. Evolutionary computation techniques are stochastic algorithms whose search methods model some natural phenomena: genetic inheritance and Darwinian strife for survival. This paper discusses the main paradigms of evolutionary computation techniques (genetic algorithms, evolution strategies, evolutionary programming, genetic programming) as well as other methods, which are hard to classify. Also, we discuss the major current trends in this field.

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.

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 (Belew and Booker, 1991; Forrest, 1993; Grefenstette, 1985; 1987; Schaffer, 1989), Parallel Problem Solving from Nature—PPSN (Männer and Manderick, 1992; Schwefel and Männer, 1991), Annual Conferences on Evolutionary Programming—EP (Fogel and Atmar, 1992)); new annual conferences are getting started, e.g., IEEE International Conferences on Evolutionary Computation (EC, 1994; 1995). Also, there are many workshops, special sessions, and local conferences every year, all around the world. A relatively new journal, *Evolutionary Computation* (DeJong, 1993), is devoted entirely to evolutionary computation techniques; many other journals organized special issues on evolutionary computation (e.g., Fogel, 1994; Michalewicz, 1994)). Many excellent tutorial papers (Beasley *et al.*, 1993a; 1993b; Fogel, 1994; Reeves, 1993; Whitley, 1994) and technical reports provide more-or-less complete

* Department of Computer Science, University of North Carolina, Charlotte, NC 28223, USA, e-mail: zbyszek@uncc.edu

** Institute of Computer Science, Polish Academy of Sciences, ul. Ordona 21, 01-237 Warsaw, Poland, e-mail: michalew@ipipan.waw.pl

bibliographies of the field (Alander, 1994; Goldberg, Milman and Tidd, 1992; Nissen, 1993; Saravanan and Fogel, 1993). There is also *The Hitch-Hiker's Guide to Evolutionary Computation* prepared initially by Jörg Heitkötter and currently by David Beasley (Heitkötter, 1993), available on comp.ai.genetic interest group (Internet), and a new text, *Handbook of Evolutionary Computation*, is currently being prepared (Bäck *et al.*, 1996).

In this paper we survey evolutionary computation algorithms and discuss the major current trends in this 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 presents some current research direction and Section 5 concludes the paper.

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 (Davis and Steenstrup, 1987):

> "... 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 changing 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 Fig. 1.

The evolutionary algorithm maintains a population of individuals, $P(t) = \{x_1^t, \dots, x_n^t\}$ 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),

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

Fitness of an individual is assigned proportionally to the value of the objective function for the individual; individuals are selected for the next generation on the basis of their fitness.

The combined effect of selection, crossover, and mutation gives the so-called reproductive schema growth equation (Holland, 1975):

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

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 '$\star$'); $\xi(S,t)$ denotes the number of strings in a population at 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 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 a 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:

Theorem 1. (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 (Goldberg, 1989):

> "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 a long defining length and high-order schemata).

Holland (1975) showed that at least *pop_size* 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 a 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 e.g. (Michalewicz, 1996).

3.2. Evolution Strategies

Evolution strategies (ESs) were developed as a method to solve parameter optimization problems (Schwefel, 1994); 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 zero mean 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 $(\vec{x}^{t+1}, \vec{\sigma})$

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

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

Fig. 1. The structure of an evolutionary algorithm.

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 chromosomal 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 Fig. 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 se-

[2] In most cases crossover involves just two parents, however, it need not be the case. In a recent study (Eiben *et al.*, 1994) the authors investigated the merits of 'orgies', where more than two parents are involved in the reproduction process. Also, scatter search techniques (Glover, 1977) proposed the use of multiple parents.

[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 one.

lection, where some number of individuals (usually two) compete for selection to the next generation: this competition (tournament) step is repeated the 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. 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 (Goldberg, 1989). However, the work done in the 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 (Holland, 1975). Mutation flips a bit in a chromosome and crossover exchanges genetic material between two parents: if the parents are represented by five-bit 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]

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

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

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 remains unchanged.

The vector of standard deviations $\vec{\sigma}$ remains unchanged during the evolution process. If all components of this vector are identical, i.e., $\vec{\sigma} = (\sigma, \dots, \sigma)$, and the optimization problem is *regular*[7], it is possible to prove the convergence theorem (Bäk, 1991):

Theorem 2. (Convergence Theorem): *For $\sigma > 0$ and a regular optimization problem with $f_{opt} > -\infty$ (minimization) or $f_{opt} < \infty$ (maximization), we have*

$$p\left\{\lim_{t\to\infty} f(\vec{x}^t) = f_{opt}\right\} = 1$$

The evolution strategies evolved further (Schwefel, 1994) to mature as

$$(\mu + \lambda)\text{-ESs} \qquad \text{and} \qquad (\mu, \lambda)\text{-ESs}$$

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.

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,

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

[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 minimization problems; for maximization problems the relationship is opposite) is a closed set.

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

- discrete, where the new offspring is

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

  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

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

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 $\vec{\sigma}' = \vec{\sigma}\cdot e^{N(0,\Delta\vec{\sigma})}$, $\vec{x}' = \vec{x} + N(0,\vec{\sigma}')$, and 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 (Schwefel, 1995).

3.3. Evolutionary Programming

The original evolutionary programming (EP) techniques were developed by Lawrence Fogel (1996). They aimed at evolution of artificial intelligence in the sense of developing an 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 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.

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

So, the evolutionary programming technique maintains a population of finite state machines; each such individual represents a potential solution to the problem

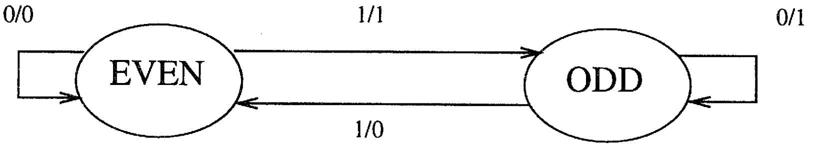

Fig. 2. An FSM for a parity check.

(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 have been seen so far, an 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 (Section 8.1), the 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 the (*pop_size*,*pop_size*)-ES). Offspring (new FSMs) are created by random mutations of the parent population (see Fig. 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).

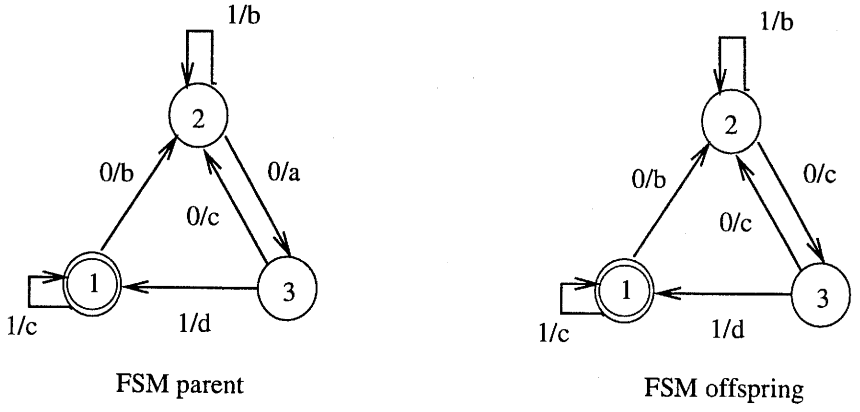

Fig. 3. An FSM and its offspring. Machines start in state 1.

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 the original version (Fogel *et al.*, 1996) 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 ways, as stated in (Fogel, 1995):

> "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 environmental 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 (Fogel, 1992; 1995). For other examples of evolutionary programming techniques, see also (Fogel *et al.*, 1996) (classification of a sequence of integers into primes and nonprimes), (Fogel, 1993) (for application of EP technique to the interated prisoner's dilemma), as well as (Fogel, 1992; 1993; McDonnel *et al.*, 1995; Sebald and Fogel, 1994) for many other applications.

3.4. Genetic Programming

Another interesting approach was developed relatively recently by Koza (1990; 1992). 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 fittest). 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

[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.

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 (Fig. 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 $x \cdot \sin(2x)$.

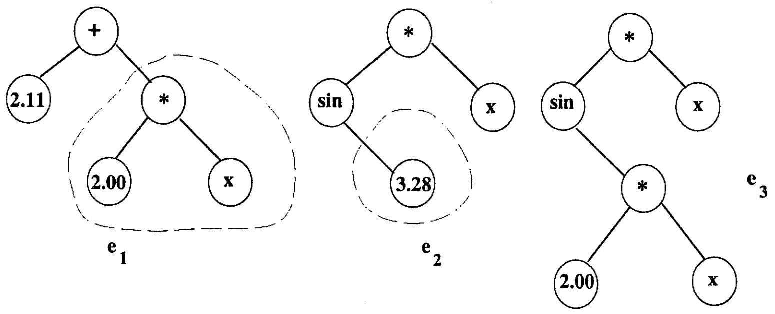

Fig. 4. Expression e_3: an offspring of e_1 and e_2. Dashed lines include areas being exchanged during the crossover operation.

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 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 (Koza, 1990). 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 (1994) 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 exciting areas of the current development in the evolutionary computation field with already a significant amount of experimental data (see, for example, apart from (Koza, 1992; 1994), also (Angeline, 1996; Kinnear, 1994)).

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 (1989) 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 the evolutionary computation community. Very often these systems, extended by the problem-specific knowledge, outperform other classical evolutionary methods as well as other standard techniques (Michalewicz, 1993; 1996). For example, a system Genetic-2N (Michalewicz, 1993) constructed for the nonlinear transportation problem used a matrix representation for its chromosomes, a problem-specific mutation (the main operator, used with probability 0.4) and arithmetical crossover (the 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 a particular problem.

There are a few heuristics to guide a user in selection of appropriate data structures and operators for a particular problem. It is a common knowledge that for a numerical optimization problem one should use an evolutionary strategy or genetic algorithm with floating-point representation, whereas some versions of the genetic algorithm would be the best to handle combinatorial optimization problems. Genetic programs are great in the 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., the 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 skillfully with evolutionary techniques.

4. Evolutionary Computation: Current Directions

The field of evolutionary computation has been growing rapidly over the last few years. Yet, there are still many gaps to be filled, many experiments to be done, many questions to be answered. In this section of the paper, we examine a few important directions in which we can expect a lot of activities and significant results; we discuss them in turn.

4.1. Theoretical Foundations

As indicated in the previous section, some evolution programs enjoy some theoretical foundations. For evolution strategies applied to regular problems a convergence property can be shown. Genetic algorithms, on the other hand, have the Schema Theorem which explains why they work. However, many techniques are modified while applied to particular real world problems. For example, to adapt a GA to the task of function optimization, it is usually necessary to extend them by additional features (e.g., dynamic scaled fitness, rank-proportional selection, inclusion of elitist strategy, adaptation of various parameters of the search, various representations, problem-specific operators, etc). Evolution strategies, applied to a constrained numerical optimization problem, usually incorporate some heuristic method for constraint-handling. Most of these modifications pushed simple algorithms away from their theoretical bases, however, they usually enhanced the performance of the systems in a significant way. In the context of genetic algorithms, these modifications (De Jong, 1993):

> "... had pushed the application of simple GAs well beyond our initial theories and understanding, creating a need to revisit and extend them."

This is one of the most challenging tasks for researchers in the field of evolutionary computation.

It is also important to continue research on factors affecting the ability of evolutionary systems to solve various (usually optimization) problems. What makes a problem hard or easy for an evolutionary method? This is a fundamental issue of evolutionary computation; some results related to deceptive problems, rugged fitness landscapes, epistasis, royal road functions, are steps towards approximating an answer to this challenging question.

4.2. Function Optimization

For many years, most evolutionary techniques have been evaluated and compared with each other in the domain of function optimization. It seems also that this domain of function optimization would remain the primary test-bed for many new

4.6. Co-Evolutionary Systems

There is a growing interest in co-evolutionary systems, where more than one evolution process takes place: usually there are different populations there (e.g., additional populations of parasites or predators) which interact with each other. In such systems the evaluation function for one population may depend on the state of the evolution processes in the other population(s). This is an important topic for modeling artificial life, some business applications, etc.

Co-evolutionary systems might be important for approaching large-scale problems (Potter and De Jong, 1994), where a (large) problem is decomposed into smaller subproblems; there is a separate evolutionary process for each of the subproblems, however, these evolutionary processes are connected with each other. Usually, evaluation of individuals in one population depends also on developments in other populations.

The recently developed Genocop III system (Michalewicz and Nazhiyath, 1995) for numerical optimization problems with nonlinear constraints incorporates some co-evolutionary ideas: two populations (of not necessarily feasible search points and fully feasible reference points) co-exist with each other. In this system, evaluation of search points depends on the current population of reference points. In an approach proposed by Le Riche *et al.* (1995), two populations of individuals co-operate with each other and approach a feasible, optimum solution from two directions (from the feasible and infeasible parts of the search space). Also, Paredis (Paredis, 1993) experimented with co-evolutionary systems in the context of constraint satisfaction problems.

Recently, a co-evolutionary system was used (Nadhamuni, 1995) to model strategies of two competing companies (bus and rail companies) which compete for passengers on the same routes. Clearly, profits of one company depend on the current strategy (capacities and prices) of the other company; the study investigated the interrelationship between various strategies over time.

4.7. Diploid/Polyploid versus Haploid Structures

Diploids (or polyploids) can be viewed as a way to incorporate memory into the individual's structure. Instead of a single chromosome (haploid structure) representing a precise information about an individual, a diploid structure is made up of a pair of chromosomes: the choice between two values is made by some dominance function. The diploid (polyploid) structures are of particular significance in non-stationary environments (i.e., for time-varying objective functions) and for modeling complex systems (possibly using co-evolution models). However, there is no theory to support the incorporation of a dominance function into the system; there is also quite little experimental data in this area.

4.8. Parallel Models

The parallelism promises to put within our reach solutions to problems untractable before; clearly, it is one of the most important areas of computer science. Evolutionary algorithms are very suitable for parallel implementations; as Goldberg (1989) observed:

> "In a world where serial algorithms are usually made parallel through countless tricks and contortions, it is no small irony that genetic algorithms (highly parallel algorithms) are made serial through equally unnatural tricks and turns."

However, there is not any standard methodology for incorporating parallel ideas into GAs: existing parallel implementations can be classified into one of the following categories:

- *massively parallel GAs.* Such algorithms use a large number of processors (usually 2^{10} or more). Often a single processor is assigned to an individual in the population. In this model there are many possibilities for the selection method and mating (combining strings for crossover). Some experimental work in this area is reported by Mühlenbein (1989).
- *parallel island models.* Such algorithms assume that several subpopulations evolve in parallel. The models include a concept of migration (movement of an individual string from one subpopulation to another) and crossovers between individuals from different subpopulations. There are many reported experiments in this parallel model; the reader is referred to Whitley's work (1990) for full discussion.
- *parallel hybrid GAs.* It is similar to the first model (massively parallel GAs) in that there is one-to-one correspondence between processors and individuals. However, only a small number of processors is used. Additionally, such algorithms incorporate other (heuristic) algorithms (e.g., hill-climbing) to improve the performance of the system. Usually the reported experimental results are satisfactory (Gorges-Schleuter, 1989; Mühlenbein, 1989), however, an analysis of such systems is far from trivial.

Parallel models can also provide a natural embedding for other paradigms of evolutionary computation, like non-random mating, some aspects of self-adaptation, or co-evolutionary systems.

5. Conclusions

It is worthwhile to note that there are many other approaches to learning, optimization, and problem solving, which are based on other natural metaphors from nature—the best known examples include neural networks and simulated annealing. There is a growing interest in all these areas; the most fruitful and challenging direction seems to be a 'recombination" of some ideas at present scattered in different fields.

Moreover, it seems that the whole field of artificial intelligence should lean towards evolutionary techniques; as Lawrence Fogel stated (Fogel, 1994) in his plenary talk during the World Congress on Computational Intelligence (Orlando, 27 June–2 July 1994):

> "If the aim is to generate artificial intelligence, that is, to solve new problems in new ways, then it is inappropriate to use any fixed set of rules. The rules required for solving each problem should simply evolve ..."

References

Alander J.T. (1994): *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.

Angeline P.J. and Kinnear, K.E., Eds. (1996): *Advances in Genetic Programming II.* — Cambridge: The MIT Press.

Arabas J., Michalewicz, Z. and Mulawka, J. (1994): *GAVaPS — a genetic algorithm with varying population size*, In: (EC, 1994), pp.73–78.

Bäck T., Fogel D. and Michalewicz Z., Eds. (1996): *Handbook of Evolutionary Computation.* — New York: Oxford University Press.

Bäck T., Hoffmeister F. and Schwefel H.-P. (1991): *A survey of evolution strategies*, In: (Belew and Booker, 1991), pp.2–9.

Beasley D., Bull D.R. and Martin R.R. (1993a): *An overview of genetic algorithms: Part 1, Foundations.* — University Computing, v.15, No.2, pp.58–69.

Beasley D., Bull D.R. and Martin, R.R. (1993b): *An overview of genetic algorithms: Part 2, Research topics.* — University Computing, v.15, No.4, pp.170–181.

Belew R. and Booker L., Eds. (1991): Proc. 4th Conf. *Genetic Algorithms.* — Los Altos: Morgan Kaufmann Publishers.

Davidor Y., Schwefel H.-P. and Männer R., Eds. (1994): Proc. 3rd Int. Conf. *Parallel Problem Solving from Nature (PPSN).* — New York: Springer-Verlag.

Davis L., Ed. (1987): *Genetic Algorithms and Simulated Annealing.* — Los Altos: Morgan Kaufmann Publishers.

Davis L. (1989): *Adapting operator probabilities in genetic algorithms*, In: (Shaffer, 1989), pp.61–69.

Davis L. and Steenstrup M. (1987): *Genetic algorithms and simulated annealing: An overview*, In: (Davis, 1987), pp.1–11.

De Jong K.A., Ed. (1993): *Evolutionary Computation.* — Cambridge: The MIT Press.

De Jong K. (1994): *Genetic algorithms: A 25 year perspective*, In: (Żurada *et al.*, 1994), pp.125–134.

EC (1994): Proc. 1th IEEE Int. Conf. *Evolutionary Computation.* — Orlando, 26 June–2 July.

EC (1995): Proc. 2nd IEEE Int. Conf. *Evolutionary Computation.* — Perth, 29 November–1 December.

Eiben A.E, Raue P.-E. and Ruttkay Zs. (1994): *Genetic algorithms with multi-parent recombination*, In: (Davidor *et al.*, 1994), pp.78–87.

Eshelman L.J., Ed. (1995): Proc. 6th Int. Conf. *Genetic Algorithms.* — San Mateo: Morgan Kaufmann.

Eshelman L.J. and Schaffer J.D. (1991): *Preventing Premature Convergence in Genetic Algorithms by Preventing Incest*, In: (Belew and Booker, 1991), pp.115–122.

Fogel D.B. (1992): *Evolving artificial intelligence.* — Ph.D. Thesis, University of California, San Diego.

Fogel D.B. (1993): *Evolving behaviours in the iterated prisoner's dilemma.* — Evolutionary Computation, v.1, No.1, pp.77–97.

Fogel D.B., Ed. (1994): IEEE Trans. on Neural Networks, special issue on *Evolutionary Computation*, v.5, No.1.

Fogel D.B. (1994): *An Introduction to Simulated Evolutionary Optimization.* — IEEE Trans. Neural Networks, special issue on *Evolutionary Computation*, v.5, No.1.

Fogel D.B. (1995): *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence.* — Piscataway: IEEE Press.

Fogel D.B. and Atmar W. (1992): Proc. 1st Ann. Conf. *Evolutionary Programming.* — La Jolla: Evolutionary Programming Society.

Fogel D.B. and Atmar W. (1993): Proc. 2nd Ann. Conf. *Evolutionary Programming.* — La Jolla: Evolutionary Programming Society.

Fogel L.J., Owens A.J. and Walsh M.J. (1996): *Artificial Intelligence Through Simulated Evolution.* — Chichester: John Wiley.

Fogel L.J. (1994): *Evolutionary programming in perspective: The top-down view*, In: (Żurada *et al.*, 1994), pp.135–146.

Forrest S., Ed. (1993): Proc. 5th Int. Conf. *Genetic Algorithms.* — Los Altos: Morgan Kaufmann Publishers.

Glover F. (1977): *Heuristics for integer programming using surrogate constraints.* — Decision Sciences, v.8, No.1, pp.156–166.

Goldberg D.E. (1989): *Genetic Algorithms in Search, Optimization and Machine Learning.* — Reading: Addison-Wesley.

Goldberg D.E. (1987): *Simple genetic algorithms and the minimal, deceptive problem*, In: (Davis, 1987), pp.74–88.

Goldberg D.E., Deb K. and Korb B. (1991): *Do not worry, be messy*, In: (Belew and Booker, 1991), pp.24–30.

Goldberg D.E., Milman K. and Tidd C. (1992): *Genetic algorithms: A bibliography.* — IlliGAL Technical Report 92008.

Gorges-Schleuter M. (1989): *ASPARAGOS An asynchronous parallel genetic optimization strategy*, In: (Shaffer, 1989), pp.422–427.

Grefenstette J.J., Ed. (1985): Proc. 1st Int. Conf. *Genetic Algorithms.* — Hillsdale: Lawrence Erlbaum Associates.

Grefenstette J.J., Ed. (1987): Proc. 2nd Int. Conf. *Genetic Algorithms.* — Hillsdale: Lawrence Erlbaum Associates.

Heitkötter J., Ed. (1993): *The hitch-hiker's guide to evolutionary computation.* — FAQ in comp.ai.genetic, issue 1.10, 20 December.

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

Holland J.H. (1993): *Royal road functions.* — Genetic Algorithm Digest, v.7, No.22.

Jones T. (1994): *A description of Holland's royal road function.* — Evolutionary Computation, v.2, No.4, pp.409–415.

Jones T. and Forrest S. (1995): *Fitness distance correlation as a measure of problem difficulty for genetic algorithms,* In: (Eshelman, 1995), pp.184–192.

Julstrom B.A. (1995): *What have you done for me lately? Adapting operator probabilities in a steady-state genetic algorithm,* In: (Eshelman, 1995), pp.81–87.

Kinnear K.E., Ed. (1994): *Advances in Genetic Programming.* — Cambridge: The MIT Press.

Koza J.R. (1990): *Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems.* — Report No. STAN–CS–90–1314, Stanford University.

Koza J.R. (1992): *Genetic Programming.* — Cambridge: MIT Press.

Koza J.R. (1994): *Genetic Programming - 2.* — Cambridge: MIT Press.

Le Riche R., Knopf-Lenoir C. and Haftka, R.T. (1995): *A segregated genetic algorithm for constrained structural optimization,* In: (Eshelman, 1995), pp.558–565.

Männer R. and Manderick B., Eds. (1992): Proc. 2nd Int. Conf. *Parallel Problem Solving from Nature (PPSN).* — Amsterdam: North-Holland, Elsevier Science Publishers.

McDonnell J.R., Reynolds R.G. and Fogel D.B., Eds. (1995): Proc. 4th Ann. Conf. *Evolutionary Programming.* — Cambridge: The MIT Press.

Michalewicz Z. (1993): *A hierarchy of evolution programs: An experimental study.* — Evolutionary Computation, v.1, No.1, pp.51–76.

Michalewicz Z. (1996): *Genetic Algorithms + Data Structures = Evolution Programs. 3rd edition.* — New York: Springer-Verlag,

Michalewicz Z., Ed. (1994): Statistics & Computing, special issue on *Evolutionary Computation.* — v.4, No.2.

Michalewicz Z. and Nazhiyath G. (1995): *Genocop III: A co-evolutionary algorithm for numerical optimization problems with nonlinear constraints.* — Proc. 2nd IEEE Int. Conf. *Evolutionary Computation,* v.2, Perth, 29 November–1 December, pp.647–651.

Mühlenbein H. (1989): *Parallel genetic algorithms, population genetics and combinatorial optimization,* In: (Schaffer, 1989), pp.416–421.

Mühlenbein H. and Schlierkamp-Vosen D. (1993): *Predictive models for the breeder genetic algorithm.* — Evolutionary Computation, v.1, No.1, pp.25–49.

Nadhamuni P.V.R. (1995): *Application of co-evolutionary genetic algorithm to a game.* — Charlotte: Master Thesis, Department of Computer Science, University of North Carolina.

Nissen V. (1993): *Evolutionary Algorithms in Management Science: An Overview and List of References.* — European Study Group for Evolutionary Economics.

Paredis J. (1993): *Genetic state-space search for constrained optimization problems.* — Proc. 13th Int. Joint Conf. *Artificial Intelligence,* San Mateo: Morgan Kaufmann Publishers.

Potter M. and De Jong K. (1994): *A Cooperative Coevolutionary Approach to Function Optimization.* — George Mason University.

Radcliffe N.J. (1991): *Formal analysis and random respectful recombination*, In: (Belew and Booker, 1991), pp.222–229.

Radcliffe N.J. (1993): *Genetic set recombination*, In: (Whitly, 1993), pp.203–219.

Radcliffe N.J. and George F.A.W. (1993): *A study in set recombination*, In: (Forrest, 1993), pp.23–30.

Reeves C.R. (1993): *Modern Heuristic Techniques for Combinatorial Problems.* — London: Blackwell Scientific Publications.

Ronald E. (1995): *When selection meets seduction*, In: (Eshelman, 1995), pp.167–173.

Saravanan N. and Fogel D.B. (1993): *A bibliography of evolutionary computation and applications.* — Dept. of Mechanical Engineering, Florida Atlantic University, Technical Report No.FAU-ME-93-100.

Schaffer J., Ed. (1989): Proc. 3rd Int. Conf. *Genetic Algorithms.* — Los Altos: Morgan Kaufmann Publishers.

Schaffer J.D. and Morishima A. (1987): *An adaptive crossover distribution mechanism for genetic algorithms*, In: (Grefenstette, 1987), pp.36–40.

Schraudolph N. and Belew R. (1990): *Dynamic parameter encoding for genetic algorithms.* — La Jolla: CSE Technical Report #CS90–175, University of San Diego .

Schwefel H.-P. (1994): *On the evolution of evolutionary computation*, In: (Żurada *et al.*, 1994), pp.116–124.

Schwefel H.-P. (1995): *Evolution and Optimum Seeking.* — Chichester: John Wiley.

Schwefel H.-P. and Männer R., Eds. (1991): Proc. 1st Int. Conf. *Parallel Problem Solving from Nature (PPSN).* — Lecture Notes in Computer Science, v.496., New York: Springer-Verlag.

Sebald A.V. and Fogel L.J. (1994): Proc. 3rd Ann. Conf. *Evolutionary Programming.* — San Diego: World Scientific.

Shaefer C.G. (1987): *The ARGOT strategy: Adaptive representation genetic optimizer technique*, In: (Grefenstette, 1987), pp.50–55.

Spears W.M. (1995): *Adapting crossover in evolutionary algorithms*, In: (Männer and Manderick, 1992), pp.367–384.

Srinivas M. and Patnaik L.M. (1994): *Adaptive probabilities of crossover and mutation in genetic algorithms.* — IEEE Trans. Systems, Man, and Cybernetics, v.24, No.4, pp.17–26.

Whitley D. (1990): *GENITOR II: A distributed genetic algorithm.* — J. Experimental and Theoretical Artificial Intelligence, v.2, pp.189–214.

Whitley D. (1994): *Genetic algorithms: A tutorial*, In: (Michalewicz, 1994), pp.65–85.

Whitley D., Ed. (1993): *Foundations of genetic algorithms-2.* — 2nd Workshop *Foundations of Genetic Algorithms and Classifier Systems*, San Mateo: Morgan Kaufmann Publishers.

Zurada J., Marks R. and Robinson C., Eds. (1994): *Computational Intelligence: Imitating Life.* — Piscataway: IEEE Press.