Evolutionary Algorithms

M. Tomassini

Swiss Scientific Computing Center, Manno
and
Logic Systems Laboratory
Swiss federal Institute of Technology
CH-1015, Lausanne, Switzerland
e-mail: tomassini@di.epfl.ch

Abstract. Evolutionary algorithms have been gaining increased attention the past few years because of their versatility and are being successfully applied in several different fields of study. We group under this heading a family of new computing techniques rooted in biological evolution that can be used for solving hard problems. In this chapter we present a survey of genetic algorithms and genetic programming, two important evolutionary techniques. We discuss their parallel implementations and some notable extensions, focusing on their potential applications in the field of evolvable hardware.

1 Introduction

The performance of modern computers is quite impressive; it seems fair to say that computers are far better than humans in many domains and that they comprise a powerful tool that is constantly changing our view of the world. On scientific and engineering number-crunching problems performance increases steadily and we are able to tackle so-called "grand challenge" problems with gigaflops and soon teraflops parallel machines. The best chess-playing programs to date are able to beat chess masters, thereby proving worthy opponents in a task requiring high-level symbolic processing. Many other tasks, which although less glamourous are highly important, are performed superbly by an average laptop. For example, writing high-quality documents with color graphics or connecting to the Internet in order to obtain valuable information, is quite commonplace these days. Despite this success there still exists a large "gray" area in which our current leading computing paradigms do not seem to work well. Children, and even animals are much better than computers in real-life tasks carried out in a dynamic environment. Indeed, biology has long served as inspiration in the construction of artifacts; this could also be applied to computers, once we recognize the fundamental underlying common digital structure of both molecular biology and machines. While this extrapolation holds in principle, there are many impediments and difficulties in practice; however, the paradigm is a valuable and far-reaching one, as the collection before you attempts to demonstrate.

Problem solving methods inspired by the biological world, such as evolutionary computing and neural networks, are by now an accepted and popular

addition to the tool-case of scientists and engineers in many different areas. Relegated for decades to the backyard by artificial intelligence groups, concerned with general problem solvers and universal symbol manipulators, they were considered too sloppy an approach for describing the supposed mathematical beauty hiding behind the apparent complexity of natural and man-made systems. Computational resources available at the time also posed a serious problem for such methods. The symbolic approach worked quite well for highly idealized problem subsets such as theorem proving or game-playing. However, when it come to tasks such as recognizing simple patterns, people soon realized that the approach met with enormous problems and did not scale well. The difficulty stems from the impossibility of the description to adapt itself to changing conditions, i.e., it is "brittle" and thereby does not adapt to a dynamic environment.

One can try to patch up such a system by adding more rules, more parameters, more computer power or more human expert information but sooner or later the entire structure collapses. Formal systems definitely have their place and are very useful in many instances. However, when dealing with dynamic, ill-defined environments, new approaches must be considered. Continuous change and adaptation is intrinsic to many phenomena and nature's "sloppy" ways may turn out to be best for such situations. In fact, fuzziness and approximation seems to be the rule and not the exception in many daily activities and in the surrounding world.

In this chapter we will concentrate on one class of methods, those inspired by natural evolution, demonstrating that they constitute a flexible and powerful metaphor for problem solving.

2 Evolutionary Algorithms

Evolutionary algorithms are search and optimization procedures that find their origin and inspiration in the biological world. The Darwinian theory of evolution, emphasizing the survival of the fittest in a dynamic environment, seems to be generally accepted, at least on the grounds of evidence accumulated so far on Earth.

Evolutionary algorithms is a general term encompassing a number of related methodologies, all of which are based on the natural evolution paradigm. *Genetic Algorithms*, *Evolution Strategies* and *Evolutionary Programming* are the historically prominent approaches with *Genetic Programming* rapidly emerging in recent years. For lack of space we shall concentrate on genetic algorithms (GA) and on genetic programming (GP), the two most widely used techniques in conjunction with evolvable hardware. After presenting the basics of genetic algorithms through a simple example, extensions and refinements will be discussed. Next, we introduce the genetic programming approach.

Evolutionary algorithms are intrinsically parallel since evolution takes place through the simultaneous interactions of individuals in spatially extended domains. Parallel GAs are generally easy to implement and offer increased performance at low programming cost. Since evolutionary methods are computation-

ally intensive, parallel GAs offer a promising approach; furthermore, they suggest novel ways in which artificial evolution can be put to use. We shall elborate upon these issues in section 8.

Evolutionary algorithms have been applied to many problems in diverse fields of study, including: hard function and combinatorial optimization, neural network design, planning and scheduling, industrial design, management and economics, machine learning, and pattern recognition. It is not our intention to discuss here particular applications of evolutionary algorithms, and the interested reader is referred to the extended literature available [1]

Artificial evolution holds promise for fundamentally changing the way in which computing machines are designed. *Evolutionary engineering*, as it is often called, might well be the only way to fabricate systems that exhibit better adaptablity and fault tolerance. This book is dedicated to a discussion of an interdisciplinary frontier, i.e., *evolvable hardware*. Whereas evolutionary computing is a mature research field with many existing applications, evolvable hardware is still taking its first steps. It is important to note the distinction between *intrinsic* and *extrinsic* hardware evolution (Thompson *et al.* [1], Kitano [2]), as well as the role and limitations of software simulations in hardware evolutionary design (Mondada and Floreano [3], Thompson *et al.* [1]). Thompson *et al.* also provide interesting arguments for using variants of evolutionary algorithms for machine evolution; these stress continuous adaptation in a noisy and changing fitness landscape, rather than straight optimization of some fixed objective.

3 Genetic Algorithms

Genetic algorithms were invented by John Holland, finding their inspiration in the evolutionary process occurring in nature. The main idea is that in order for a population of individuals to collectively adapt to some environment, it should behave like a natural system; survival, and therefore reproduction, is promoted by the elimination of useless or harmful traits and by rewarding useful behavior. Holland's insight was in abstracting the fundamental biological mechanisms that permit system adaptation into a mathematically well specified algorithm.

Genetic algorithms have been used essentially for searching and optimization problems and for machine learning. However, it is still an unresolved question whether the natural evolutionary process is really an optimization process. Evolution is essentially a one-shot experiment, although many alternatives were tried along the way and discarded through the selection process; we cannot start from zero and try again. A careful discussion of why straight optimization might not be the right point of view in the machine evolution domain is given in [1]. Optimization is meaningful only in a given context and with given constraints. The dynamics of the evolutionary process are extremely complex and as yet mostly unknown; any hypothesis concerning optimization in nature would therefore be

[1] The work is scattered in many conference proceedings and journals; a good starting place is the International Conference on Genetic Algorithms (ICGA) series (see references).

tenuous at best. Nevertheless, the artificial evolution approach to optimization is viable if demonstrably good solutions can be obtained, in comparison to other approaches. We do not require of our evolutionary algorithms to be completely faithful to nature, rather, we seek to find efficient solutions. An advantage of artificial evolution is our ability to simulate the evolutionary processes as many times as we wish, under varying conditions, and at electronic speeds. The natural world provides an endless source of inspiration, once we realize the freedom to shape our ideas in pragmatic ways that do not necessarily follow nature to the hilt.

A GA is an iterative procedure that consists of a constant-size population of individuals, each one represented by a finite string of symbols encoding a possible solution in some problem space. This space, also known as the *search space*, comprises all possible solutions to the problem at hand. The symbol alphabet used is often binary, due to its generality and some other advantageous mathematical properties. The "standard" GA works as follows: an initial population of individuals is generated at random or heuristically. Every evolutionary step, called *generation*, the individuals in the current population are decoded and evaluated according to some pre-defined quality criterion, referred to as the *fitness*. To form a new population, individuals are selected with a probability proportional to their relative fitness. This ensures that the expected number of times an individual is chosen is approximately proportional to its relative performance in the population; thus, high-fitness ("good") individuals stand a better chance of reproducing, while low-fitness ones are more likely to disappear. The selection procedure alone cannot introducce any new points in the search space; these are generated by genetic operators of which the most popular ones are *crossover* and *mutation*. Crossover is a recombination operator in which two individuals, called *parents*, exchange parts, forming two new individuals called *offspring*; in its simplest, substrings are exchanged after a randomly selected crossover point. This operator enables the evolutionary process to move toward promising regions of the search space. The second operator, mutation, is essentially background noise that is introduced to prevent premature convergence to local optima by randomly sampling new points in the search space. It is carried out be flipping bits at random, with some (small) probability.

GAs are stochastic iterative algorithms that are not guaranteed to converge. Termination may be triggered by reaching a maximum number of generations or by finding an acceptable solution. The following general schema summarizes a standard genetic algorithm:

```
produce an initial population of individuals
evaluate the fitness of all individuals
while termination condition not met do
   select fitter individuals for reproduction
```

```
        recombine individuals

        mutate some individuals

        evaluate the fitness of the new individuals

        generate a new population by inserting some new good
        individuals and by discarding some old bad individuals

    end while
```

In the next section we present a tutorial example of a simple problem solved using a standard GA. The reader is referred to [4] and [5], for a general introduction and detailed discussion of GAs.

In terms of the biological analogy, it should be noted that GAs focus on individual evolution through genotypic changes, i.e., by operating on the individual's coding sequences. Other evolutionary algorithms, such as evolution strategies and evolutionary programming, take the phenotypic view instead, whereby evolutionary operators act on the individuals themselves, which are simply the natural representations for the problem at hand; for example, real numbers in parameter optimization problems. A comparison between different forms of evolutionary algorithms can be found in Fogel's book [6].

4 A Simple Example

In this section we present an example involving function optimization, demonstrating the operation of the genetic algorithm. Although GAs are not limited to this domain, their workings are probably better understood in an optimization setting. The problem is purely of illustrative value and can in fact be solved by hand.

The non-constrained function minimization problem can be cast as follows. Given a function $f(x)$ and a domain $D \in R^n$, find x^* such that:

$$f(x^*) = \min\{f(x) \mid \forall x \in D\}$$

where $x = (x_1, x_2, \ldots, x_n)^T$.

Let us consider the following function (see Fig.1):

$$f(x) = - \mid x \sin(\sqrt{\mid x \mid}) \mid + C$$

The problem is to find x^* in the interval $[-512, 512]$ which minimizes f. Since $f(x)$ is symmetric, studying it in the positive portion of the x axis will suffice.

Let us examine in turn the components of the genetic algorithm for solving the given problem.

The initial population will be formed by 50 randomly chosen trial points in the interval $[0, 512]$. Therefore, one individual is a value of the real variable x.

A binary string will be used to represent the values of x. The length of the string will be a function of the required precision; the longer the string the better

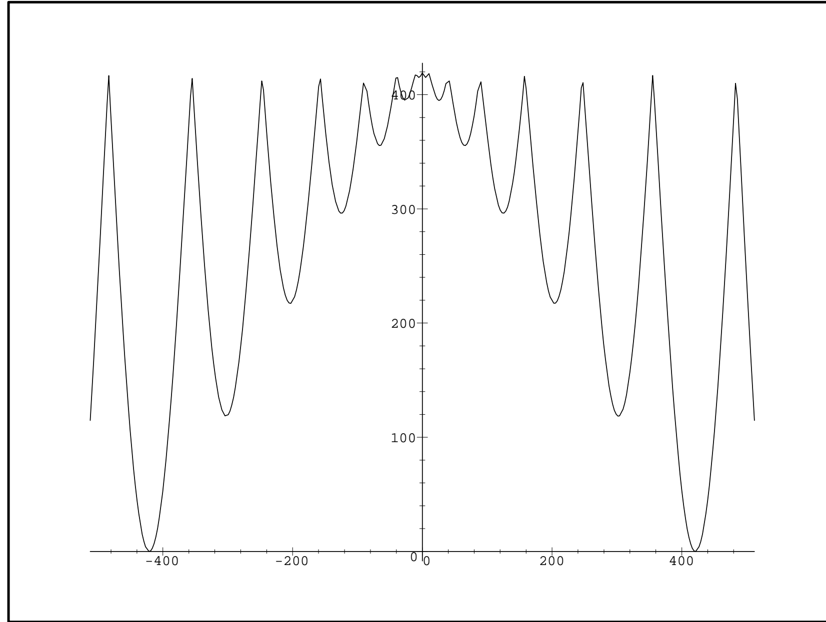

Fig. 1. Graph of $f(x), x \in [-512, 512]$.

the precision. For example, if each point x is represented by 10 bits then 1024 different values are available for covering the interval $[0, 512]$ with 1024 points, which gives a granularity of 0.5 for x i.e., the genetic algorithm will be able to sample points no less than 0.5 apart from each other.

The strings 0000000000 and 1111111111 will represent respectively the lower and upper bounds of the search interval. Any other 10-bit string will be mapped to an interior point. In order to map the binary string to a real number, the string is first converted to a decimal number and then to the corresponding real x. Note that our use of 10-bit strings is only for illustrative purposes; in real applications, finer granularities and therefore longer strings are often needed.

The fitness of each sample point x is simply the value of the function at that point. Since we want to minimize f, the lower the value of $f(x)$ the fitter is x.

How are strings selected for reproduction as a function of their fitness? Several possibilities exist, some of which shall be discussed ahead. For our current example we delineate one of the most common methods, known as *fitness-proportionate*: After evaluating the fitness f_i of each individual i in a given generation, the total fitness of the entire population, S, is computed:

$$S = \sum_{i=1}^{popsize} f_i,$$

A probability p_i is then assigned to each string i:

$$p_i = \frac{f_i}{S}$$

Finally, a cumulative probability is obtained for each individual string by adding up the fitness values of the preceding population members:

$$c_i = \sum_{k=1}^{i} p_k, \quad i = 1, 2, \ldots, popsize$$

A random number r, uniformly distributed in the range $[0, 1]$, is drawn *popsize* times and each time the i-th string is selected such that $c_{i-1} < r \leq c_i$ (if $r < c_1$, the first string is selected). This process can be visualized as the spinning of a biased roulette wheel divided into *popsize* slots, each with a size proportional to the respective individual's fitness. For example, suppose that there are only four strings with the following p_i values: $p_1 = 0.30$, $p_2 = 0.20$, $p_3 = 0.40$, $p_4 = 0.10$. Thus we have: $c_1 = 0.30$, $c_2 = 0.50$, $c_3 = 0.90$, $c_4 = 1.0$. If $r = 0.25$ (the random number generated), then individual 1 is selected since $r < c_1$; if $r = 0.96$ then individual 4 is selected since $c_3 < 0.96 < c_4$.

With such *roulette-wheel* selection fitter members are more likely to be reproduced; furthemore, strings can be selected more than once. Note that as probability measures are involved positive fitness values must be used (for this reason a positive constant C was added to our function so that $f(x) \geq 0$ in the given interval).

Once the new population has been produced, strings are paired at random and recombined through crossover. Several techniques are available, the most popular being *one-point* crossover, delineated ahead. Suppose the following two strings have been selected for recombination:

0010011010 *and* 1110010001

A crossover point is selected at random between 1 and the string length minus one, with uniform probability. Suppose that position 6 has been chosen (marked by the vertical bar):

001001 | 1010   111001 | 0001

Then, the two substrings from position 6 to the end are swapped, thus obtaining two new strings called the *offspring*:

001001 | 0001   111001 | 1010

The offspring replace their parents in the population of the next generation. Crossover is applied with a certain frequency, p_c, called *crossover rate*; any given individual takes part in the recombination process if a uniformly distributed random variable in the interval $[0, 1]$ has a value $\leq p_c$. A common empirical value for p_c is 0.6.

After crossover, mutation is applied to population members with a frequency p_m (a common empirical value is 0.01). In the standard mutation process a random number r is generated (uniformly distributed in $[0,1]$) for each bit of each string in the population; if $r \leq p_m$ then the bit is flipped. The aforementioned values of p_c and p_m are experimentally derived, proving successful in many applications. In more sophisticated GAs these probabilities need not stay constant during a run.

What is the role of these genetic operators? There is an abundant literature about different variants of crossover and mutation, along with their relative importance. In the classical GA view, crossover is the fundamental operator with mutation playing an auxiliary role. In this view, the importance of crossover stems from the fact that it combines possible beneficial traits of two individuals (parents), thereby increasing the likelihood of generating fitter individuals; in contrast, mutation is a single-individual operator. The usefulness of crossover is related to the combination of so-called *building blocks*, i.e., better-than-average substrings originating in different individuals (see next section). It "mixes" good substrings, i.e., ones that have proven successful in previous generations. Despite the advantages of crossover, mutation is still important; although the combined effect of selection and crossover is the generation of new solutions throughout the search space, they tend to cause rapid convergence and there is the danger of loosing potentially useful genetic material. It is important to note that we are in fact restricted to relatively small populations in practice, thereby entailing possibile sampling errors. In order to re-introduce diversity and to avoid premature convergence, mutation is essential; however, mutation frequencies have to be low, otherwise the search tends to degenerate into a random walk.

The relative importance of mutation and crossover is still controversial; indeed, some evolutionary techniques, such as evolution strategies and evolutionary programming, rely mainly on selection and sophisticated mutation techniques (see ref.[6]).

Equipped with these notions, let us now come back to our function minimization problem, as solved by the GA. To measure the quality of our solutions we record both the average population fitness and the fitness of the best individual, both at a given generation. As an example consider the following table, showing the results of a particular evolutionary run.

As generation 0 consists of randomly generated individuals, we find, as expected, that both the average and the best fitness values are of low quality. We observe that fairly rapid improvement ensues, with the minimum already found at generation 9. However, the average population fitness continues to improve until the population becomes homogenous and fitness values level off. This behavior is in fact characteristic of evolutionary algorithms in general. Note that in our simple example the probability of getting "stuck" in a local minimum is practically zero. In harder problems, a compromise must be reached between *exploitation* of "good" regions of the search space, (i.e., local improvement), and further *exploration* of this space, in order to find possibly better extrema points.

One final remark is in order; genetic algorithms are stochastic, thus their

Generation	Best	Average
0	1.0430	268.70
3	1.0430	78.61
9	0.00179	32.71
18	0.00179	14.32
26	0.00179	5.83
36	0.00179	2.72
50	0.00179	1.77
69	0.00179	0.15

performance varies between different runs (unless the same random number generator with the same seed is used). Thus, the average performance taken over several runs is a more useful indicator of their behavior, rather than a single run.

The problem presented above is an easy one for GAs, as well as for any other optimization method. GAs have been shown to be effective in solving hard mathematical optimization problems, involving multimodal functions of several variables [7].

5 Schemata and Building Blocks

In this section we take an in-depth look at the workings of the standard genetic algorithm, explaining why GAs constitute an effective search procedure. For simplicity we discuss binary string representation of individuals; let $\{0, 1, \#\}$ be the symbol alhapabet, where $\{\#\}$ is a special *wild card* symbol that matches both 0 and 1. A *schema* is a template consisting of a string composed of these three symbols. For example, the schema [01#1#] is a template that matches the following strings: [01010], [01011], [01110] and [01111]. The symbol # is never actually manipulated by the GA; it is only a notational device used to describe sets of strings.

Holland's idea was that as a specific string is evaluated and assigned a fitness value, this actually gives partial information about the fitness of all schemata to which the string belongs; he referred to this as *implicit parallelism* (not to be confused with the kind of parallelism discussed in section 8). Holland then analyzed the influence of selection, crossover and mutation on the expected number of schemata as evolution proceeds from one generation to the next. The details of the analysis are relatively simple but beyond our scope; a good discussion can be found in ref. [4]. In what follows we outline the main results and their significance.

Let $m(H, t)$ denote the number of individuals in the population belonging to a particular schema H at time t. Under fitness-proportionate selection this number at the next time step, $m(H, t+1)$, is related to $m(H, t)$ as follows:

$$m(H, t+1) = m(H, t)(f_H(t)/(\overline{f}(t))$$

where $f_H(t)$ is the average fitness value of the strings belonging to schema H, and $\overline{f}(t)$ is the average fitness value over all strings in the population. If one assumes that a particular schema remains above the average by a fixed amount $c\overline{f}(t)$ for a number t of generations then the solution of the above recursive equation is the following exponential growth equation:

$$m(H,t) = m(H,0)(1+c)^t$$

Where $m(H,0)$ stands for the number of individuals belonging to schema H at time 0, c is a positive constant, and $t \geq 0$. The significance of this result is that fitness-proportionate selection allocates an exponentially increasing number of trials to above-average schemata.

Now crossover and mutation enter into the picture. The effect of crossover, with its swapping of substrings, is to diminish the exponential increase by a quantity that depends on the crossover rate p_c, the *defining length* δ of the schema and on the string length l:

$$p_c \frac{\delta(H)}{l-1}$$

The defining length δ of a given schema is the distance between the first and the last fixed (i.e., non-#) string positions. For example, for the schema [01#1#] $\delta = 4 - 1 = 3$ and for [##1#1010] $\delta = 8 - 3 = 5$. Intuitively, we observe that schemata of short defining length are less likely to be "broken" under single-point crossover. Above-average schemata with short defining length, known as *building blocks*, will still be sampled at an exponentially increasing rate. These building blocks play an important role in the Holland's theory.

The effect of mutation is straightforward to describe. If the bit mutation probability is p_m, then $1 - p_m$ is the probability of a bit remaining unchanged. Since single bit mutations are independent, the probability of a string remaining unchanged is $(1-p_m)^l$, where l is the string length. For schemeta, only the fixed (i.e., non-#) positions matter; their number is called the *order* of a schema H, denoted $o(H)$, equaling l minus the number of # symbols. For example, the two schemata above have $o = 3$ and $o = 5$ respectively. The probability of shcema H "surviving" a mutation is $(1-p_m)^{o(H)}$; for $p_m \ll 1$ this is approximately given by $1 - o(H)p_m$.

The combined effect of selection, crossover and mutation, comprise Holland's *schema theorem*:

$$m(H,t+1) \geq m(H,t)\frac{f_H(t)}{\overline{f}(t)}[1 - p_c\frac{\delta(H)}{l-1} - o(H)p_m]$$

Essentially, the number of above-average, low-order schemata with short defining lengths grows exponentially in subsequent generations of a genetic algorithm.

Although the schema theorem is an important result, it was obtained under somewhat idealized conditions. The building-block hypothesis has been found to apply in many cases but it depends on the representation of individuals and the genetic operators used, both of which can be different than those used by

Holland. It is easy to find or to construct examples for which the above theorem does not hold. Such problems have been studied in the past few years in order to find out the inherent limitations of genetic algorithms, and the representations and operators, if any, that make them more tractable. Despite the above limitations, the theory sketched in this section represents a firm footing for the workings of standard genetic algorithms.

6 Extensions and Variations

In practice, the "plain" GA shown earlier is seldom used as such. Many modifications and extensions of the simple genetic algorithm have been proposed, especially for dealing with real-life problems. In this section we discuss several issues including individual representation, alternatives to fitness-proportionate selection, different forms of the genetic operators and extensions of the GA paradigm itself. For a more detailed account the reader is referred to [5].

6.1 Representation Issues

Usually, population individuals are coded as binary strings. This coding is general but it is not always the most natural nor the most adequate representation. One problem with binary-coded, unsigned integers is that numbers that are close to each other may have large Hamming distances in the binary form. For example, the string 0111 represents the decimal number 7 whereas 8 is represented by 1000, i.e., the strings differ in all bits, giving a Hamming distance of four. This means that it will be difficult for a search point that is close to an optimum to move toward it by mutation. The Gray code, in which consecutive numbers differ by one bit, is often used to alleviate this problem. However, even with Gray-coded strings there are difficulties when dealing with numerical parameter optimization problems. One problem is that mutation of high-order bits causes more drastic changes than that of low-order bits. Furthermore, the precision is a function of the number of bits in the bit string representing an individual point. Attaining sufficient precision may require many bits, which is problematic, especially where multidimensional problems are concerned. Dealing with very long bit strings is time-consuming and the search spaces are huge; thus, it is natural to consider floating point representations in this case. In fact, such representations are the standard ones in other evolutionary algorithms such as evolution strategies. Floating point representations have been used in GAs for solving numerical problems and they have been shown to outperform binary codings in many cases ([5], [6]). Note that switching to floating point representation requires careful rethinking of the genetic operators, which will differ from those used for bit strings [5].

Representation issues also appear when dealing with *combinatorial optimization* problems, i.e., discrete variable problems in which a particular solution is saught, among a finite set of possible solutions. There are no known efficient

algorithms for many hard combinatorial problems such as the Traveling Salesman Problem (TSP), the bin-packing problem, and the graph-coloring problem. Since these are paradigmatic versions of very important management problems in the fields of sequencing, routing and scheduling it is important to be able to quickly find good solutions to large instances. Various kinds of approximation and heuristic algorithms are used, among them genetic algorithms.

The following number partitioning problem will show how representation issues can be critical for genetic algorithms. Given n numbers x_i $(i = 1, \ldots, n)$, they are to be partitioned into K groups in such a way that the differences between the group sums are minimized. The number of partitions grows with K^n and the decision version of the problem is NP-complete, although it can be solved in pseudo-polynomial time by dynamic programming.

The simplest and more intuitive encoding represents each partition as a string of integer numbers

$$(p_1, p_2, \ldots, p_n) \quad p_i \in \{1, 2, \ldots, K\}, \quad i = 1, 2, \ldots, n$$

where p_i denotes the partition to which number i belongs to.

This encoding, called *group number encoding*, can make use of standard crossover and mutation operators; however, it has a major drawback: an offspring may contain less than K groups. For example, with $n = 6$ and $K = 3$, the following parents: (1 2 1 1 3 2) and (1 2 3 3 1 2), with the crossover point at position four , would give rise to the offsprings (1 2 1 1 1 2) and (1 2 3 3 3 2). The first of these lacks group 3; such an individual is obviously unacceptable since it cannot represent a near-optimal solution. Such individuals must be repaired or penalized in some way. Another possibility is to define representations and genetic operators such that only legal solutions can be produced. In fact, it can be shown that there exist much better encodings for the partition problem [5]. These new representations and operators bear little resemblance to the classical ones discussed above and theoretical results obtained for bit strings are not immediately applicable to other representations. However, the empirical evidence suggests that specific genetic representations and operators may lead to efficient evolutionary solutions to difficult real world problems.

A detailed discussion of genetic representation issues for combinatorial and numerical problems can be found in ref. [5].

6.2 Selection

In section 4 fitness-proportionate selection was introduced. This selection method is not without problems, however. One problem is that during the course of evolution, as fitter individuals obtain more copies, the differences in fitness become small, rendering selection ineffective. In this case the *selection pressure* must be modified such that better individuals reproduce more often than they would under the normal fitness evaluation.

Another problem is the possible existence of a *super individual* in the population, i.e., an individual with an unusually high fitness. Under fitness-proportionate

reproduction this individual will quickly profilerate and come to dominate the population, thus causing premature convergence to a possibly local optimum.

It is possible to partially avoid these effects by suitably *scaling* the evaluation function, which amounts to the use of a modified fitness measure. Several scaling methods have been suggested and are discussed for example in [4].

Another approach to alleviate the above effects is to use selection methods that do not allocate trials in direct proportion to fitness. Two such methods are *rank selection* and *tournament selection*.

In rank selection, the individuals in the population are ordered by fitness and copies assigned in such a way that the best individual receives a predetermined multiple of the number of copies than the worst one. Rank selection reduces the domineering effect of super individuals without the need for scaling, and at the same time it amplifies the differences between close fitness values, thus increasing the selection pressure in convergent populations. Rank selection methods have been used with some success, however, they ignore the information about relative fitness of different individuals and violate the schema theorem.

In tournament selection n individuals are selected at random with uniform probability and the best one among them (i.e., with the highest fitness) is selected (the winner can also be chosen probabilistically). The process is then repeated *popsize* times. The selection pressure is proportional to the *tournament size* n, of which a widely used value is two. Tournament selection has the advantage that it need not be global so that local tournaments can be held simultaneously in a spatially distributed population (see section 8).

6.3 Genetic Operators

Crossover and mutation as previously described are simple to understand and to deal with but they also present some drawbacks in practice. Consequently, many variants have been proposed. Let us start with crossover. It can be shown that one-point crossover may be less effective in combining certain schemata. A form of crossover that partially alleviates this problem is *multi-point crossover*. For example, in two-point crossover there are two cut points (marked below by vertical bars) and substrings are swapped between the two points:

Before crossover:

$$001 \mid 101 \mid 1010$$

$$111 \mid 001 \mid 0001$$

After crossover:

$$001 \mid 001 \mid 1010$$

$$111 \mid 101 \mid 0001$$

Multi-point crossover seems to fare better in combining certain good features present in strings.

Another widely used crossover form is *uniform crossover* [8]. Given two parent strings, for each bit in the first offspring a bit in the corresponding position is copied randomly with some probability from one of the parents. The second offspring gets the corresponding bit from the remaining parent. For example, given the two parents above and a probability of 1/2, suppose that the following series of random choices is made (where 1 stands for the first parent and 2 for the second):

1221211212

then we would obtain the following offspring:

0111011011   1010010000

Uniform crossover violates the customary form of the schema theorem and is less likely to preserve good building blocks; however, for some problems, it has given good results. For a good discussion of crossover-related issues and further references, see chapter 4 of ref. [5].

Mutation has received less attention than crossover in the GA literature. Adaptive mutation schemes have been suggested, partly borrowed from evolution strategies, in which either the rate or the form of mutation (or both), vary during a GA run. For instance mutation is sometimes defined in such a way that the search space is explored uniformly at first and more locally toward the end, such that candidate solutions are locally improved. Further information on sophisticated mutation techniques can be found in ref. [5]. The particular role of mutation as applied to machine design evolution is discussed in [1].

6.4 Steady-State Genetic Algorithm

In the standard GA the whole population changes after an evaluation-selection-recombination-mutation cycle. Each such cycle is called a generation and this reproduction technique is accordingly called *generational*. Generational replacement has some drawbacks since many good individuals may not get a chance to reproduce; on the other hand, duplicate individuals may enter the population and quickly propagate further copies, thus wasting computational resources. In addition, crossover and mutation may modify some good building blocks, thus depriving the population of potentially useful genetic material. With a limited population size, the sum of these effects can often lead to a situation in which the population quickly loses diversity. One remedy is to use *elitism*, i.e., one or more of the best individuals find their way into the next generation, together with their offspring. But even elitism does not completely avoid the premature loss of useful individuals. For these reasons a different replacement technique called *steady-state* has been proposed [8]. In steady-state GAs there is a parameter, n, determining the number of new individuals to be created. Usually, only a few members of the population are changed at a time by deleting the worst n

individuals (note that for $n = popsize$ we attain the generational model). Steady-state reproduction is particularly effective if care is taken to prevent duplicate individuals from being produced. Although checking this constraint represents a (small) overhead, a much more efficient use of the population is made in the search process.

6.5 Multimodality and Niching GAs

For objective functions having multiple extrema it may be of interest to find several optima rather than just the global one. With the standard GA this is difficult because, due to *genetic drift*, individuals tend to concentrate on the highest peak as evolution proceeds. Qualitatively, the phenomenon is caused by the convergence of a finite population when the selection pressure becomes too low. Several methods for optimizing multimodal functions with GAs have been proposed; most of them strive to form and maintain stable subpopulations around the different peaks in a manner analogous to the biological phenomenon found at the species level, called *niching*.

One of the earliest methods was proposed by Goldberg and Richardson [9]. The idea is that the GA perception of the fitness function is modified in such a way that when individuals tend to concentrate around a high peak, its fitness is reduced by a factor proportional to the number of individuals in the region. This has the effect of diminishing the "attractive power" of the peak and allowing parts of the population to concentrate on other regions. The effective fitness of an individual i, $s_f(i)$, called shared fitness is given by:

$$s_f(i) = \frac{f(i)}{m(i)} \tag{1}$$

where $f(i)$ is the original fitness and $m(i)$ is called the niche count. For an individual i, the quantity $m(i)$ is calculated by summing the sharing function values sh contributed by all N individuals in the population:

$$m(i) = \sum_{j=1}^{N} \mathrm{sh}(d_{ij}) \tag{2}$$

where d_{ij} is the distance between two individuals i and j defined as the Euclidean distance computed in the real parameter space (phenotypic sharing) and

$$\mathrm{sh}(d_{ij}) = \begin{cases} 1 - \left(\frac{d_{ij}}{\sigma_s}\right)^{\alpha} & \text{if } d_{ij} < \sigma_s \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

where α and σ_s are constants.

A difficulty of this method is to choose an adequate value of σ_s, which requires prior knowledge about the number of peaks in the problem space, information that is usually not readily available. Furthermore, it is computationally intensive because of the shared fitness and distance metric calculations. Other methods

based on clustering ideas and on subpopulation labels have been proposed and shown to be effective [10,11].

Another recently suggested method consists of executing a number of successive GA runs. After each run, the fitness function is updated at the point represented by the best individual found in the run in such a way that the new function has a "depression" at this place, i.e, the peak is decreased. The best individual is recorded as a solution if it exceeds a given threshold. In this way several optima are found successively [12].

Finally, parallel GA models, where different regions of the problem space are independently searched, can also be useful for multimodal optimization.

6.6 Coevolution

In the customary evolutionary paradigm a single population evolves under the selection pressure of a given fixed fitness function that plays the role of the environment. However, in nature the environment of a given population is actually comprised of the physical environment, which normally changes very slowly, and by the other biological populations which are simultaneously adapting. Interactions between (evolving) populations are omnipresent; consider, for example, prey-predator or host-parasite relationships. Under these conditions, it is best to think of evolution as being a *co-evolutionary* process where changes in a certain species (population) influence the other ones, i.e., the environment is altered. Thus, a kind of "arms race" develops in which evolutionary changes in one species trigger counter adaptative changes in other species, and vice versa.

These observations have only recently been exploited for creating more robust artificial evolutionary algorithms. One advantage of co-evolutionary methods is that one need not necessarily specify a global fitness function whereby population members are ranked, rather only relative fitness is needed. This can be useful since sometimes providing an adequate fitness function for a given problem can be difficult or even impossible, for example, in complex games or when the suite of test cases is very large.

The methods based on co-evolution can roughly be classified as being either *competitive* or *cooperative*. Hillis presented one of the first successful competitive co-evolutionary approaches to optimization problems [13]. The problem consisted in evolving a sorting network for 16 integers involving a minimum number of exchanges. Hillis used both a classical evolutionary approach and a co-evolutionary one. In the latter there are two populations, the first consisting of sorting networks, the second of sorting problems. These problems are permutations of integers that are to be used as test cases by the sorting networks of the first population; this second population can be viewed as an opportunistic or *parasitic* one. Both populations evolve concomitantly, i.e. co-evolve, on a two-dimensional grid, in parallel, with selection and mating carried out locally. The fitness for the sorting networks is defined as how well they sort the numbers of the immediate neighbor parasites; the parasites are scored according to their capacity for producing difficult problems for the sorting networks. In this way, the best networks learn to sort increasingly difficult sets of numbers,

thus avoiding the need for testing all 16! possible permutations of numbers. The co-evolutionary approach outperformed the standard one for this problem and Hillis was able to evolve a nearly optimal sorting network with 61 exchanges, the best known designed solutions having 60 exchanges.

Another highly parallel, local, co-evolutionary algorithm has recently been used by Sipper for evolving non-uniform cellular automata to perform computational tasks [14]. This model belongs to the cooperative class of co-evolutionary algorithms, since the individual units must work in unison to attain a global goal.

De Jong *et al.* [15] and Husbands [16] have proposed related co-evolutionary models that are also based on species cooperation rather than competition. The methods differ in their implementation but they share the notion of multiple species cooperating so as to attain a common objective. I will briefly describe De Jong's approach, referring the reader to the original papers for more details.

In this model, given a problem to be solved, multiple populations are evolved independently by a standard genetic algorithm. Each subpopulation evolves a species of individuals that represent (hopefully) useful components in the solution of the global problem. Species are then combined into full solutions and evaluated on the common global task. Credit is assigned to the species according to how well they collaborate to solve the common problem. In this way, selection pressure favors cooperation rather than competition between species, although within a single species evolution is still competition-based.

Good results, often better than standard GA-based ones, have been reported by De Jong *et al.* on simple function optimization problems [15] and on neural network evolutionary design [17].

Finally, we note that previously described niching methods and distributed subpopulation approaches to be discussed in section 8 can also be seen as instances of co-evolutionary algorithms; these are all based on the evolution and competition of semi-isolated species of the same type.

6.7 Hybrid Algorithms

Genetic algorithms are a robust, general-purpose search procedure. They belong to the so-called "weak" problem solving methods because they require little knowledge about the problem to be solved. In principle, a coding convention, crossover and mutation operators, and a suitable fitness function is all that is needed. Although GAs can quickly explore huge search spaces and find those regions that have above-average fitness, the search lacks focus.

In the realm of actual industrial optimization problems it is often the case that efficient search methods using problem-specific representations already exist. This raises the question as to whether GAs can be competitive for real-world applications when compared to more specialized algorithms and heuristics. In fact, if genetic algorithms are to be preferred in these settings, they have to be at least as good as the established methods.

The answer may lie in *hybrid genetic algorithms*. Hybrid genetic algorithms work by incorporating a fast and efficient problem-specific search procedure in

the framework of the evolutionary algorithm. They also tend to use encodings and genetic operators that are tailored to the problem at hand. The strength of hybrid algorithms lies in the combination of two different but complementary search principles: the evolutionary part performs a wide search of the problem space, while the local method, by exploiting its knowledge of the problem, searches promising regions in-depth. There are many possible combinations of these two principles and choosing the right blend is still more an art than a science. Nevertheless, efficient algorithms that outperform both the domain-specific heuristic and a non-specialized GA have been devised and successfully applied (see footnote in section 2).

Clearly, much more work is needed to put these empirical methods on firmer mathematical basis. Hybrid GAs are even less amenable to theoretical analysis than standard genetic algorithms but they are very interesting in practice and their use is increasing. Davis gives a convincing description of the motivations for using hybrid GAs [18].

7 Genetic Programming

Genetic programming (GP) is a new evolutionary approach that extends the genetic model of learning to the space of programs. It is a major variation of genetic algorithms in which the evolving individuals are themselves computer programs instead of fixed length strings from a rather limited alphabet of symbols. The present form of GP is principally due to Koza [19].

Individual programs in GP might be expressed in principle in any current programming language. However, the syntax of most languages is such that GP operators would create a large percentage of syntactically incorrect programs. For this reason, Koza chose a syntax in prefix form analogous to LISP and a restricted language with an appropriate number of variables, constants and operators defined to fit the problem to be solved. In this way syntax constraints are respected and the program search space is limited. The restricted language is formed by a user-defined *function set* F and *terminal set* T. The functions chosen are those that are a-priori believed to be useful for the problem at hand, and the terminals are usually either variables or constants. In addition, each function in the function set must be able to accept as arguments any other function return value and any data type in the terminal set T, a property that is called *syntactic closure*. Thus, the space of possible programs is constituted by the set of all possible compositions of functions that can be recursively formed from the elements of F and T.

As an example, suppose that we are dealing with simple arithmetic expressions in three variables. In this case suitable function and terminal sets might be defined as:

$$F = \{+, -, *, /\}$$

and

$$T = \{A, B, C\}$$

and the following are legal programs: (+ (* A B) (/ C D)), and (* (- (+ A C) B) A).

It is important to note that GP does not need to be implemented in the LISP language (though this was the original implementation). Any language that can represent programs internally as parse trees is adequate. Thus, most GP packages today are written in C or C++ rather than LISP.

Programs are thus represented as trees with ordered branches in which the internal nodes are functions and the leaves are the terminals of the problem. For the examples given above, we would have the trees in fig. 2.

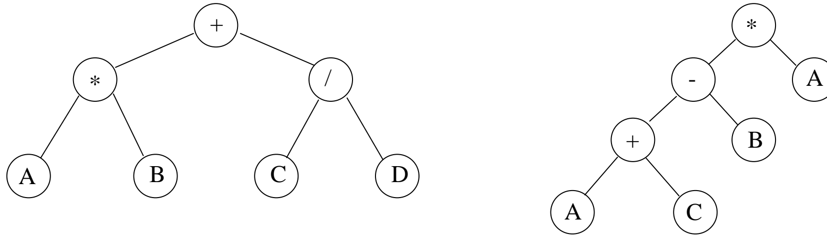

Fig. 2. Two GP trees corresponding to the LISP expressions in the text

Evolution by GP is similar to GAs, using different individual representation and genetic operators. Once suitable functions and terminals are determined for the problem at hand, an initial random population of trees (programs) is constructed. From there on the population evolves as with a GA where fitness is assigned after actual execution of the program (individual) on a suitable set of test cases, and with genetic operators adapted to the tree representation.

The crossover operation starts by selecting a random crossover point in each parent tree and then exchanging the sub-trees, giving rise to two offspring trees, as shown in figure 3. Mutation is implemented by randomly removing a subtree at a selected point and replacing it with a randomly generated subtree, although this operator is seldom used.

Genetic programming has been successfully applied to a wide variety of problems from many fields, described in [19]. For example, Gruau's chapter in this volume describes a form of genetic programming for his *Cellular Programming Language* as applied to neural network design [30].

One problematic step in GP is the choice of the appropriate language for a given problem. In general, the problem itself suggests a reasonable set of functions and terminals but this is not always the case. Although experimental evidence has shown that good results can be obtained with slightly different choices of F and T, it is clear that the choice of language directly bears on how hard the problem will be to solve with GP. For the time being, there is no guideline for estimating this dependence.

Another controversial issue has to do with the size of the GP trees. The depth of the trees can in principle increase without limits under the influence of

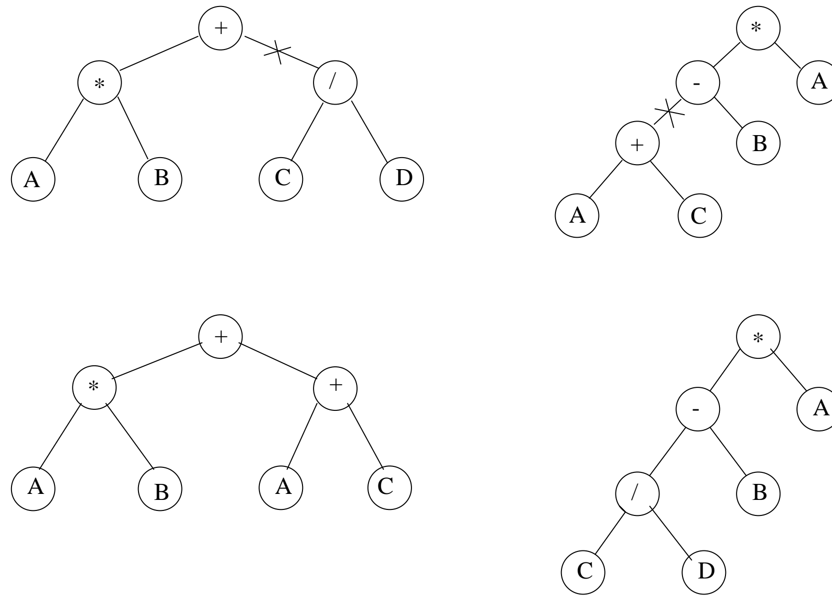

Fig. 3. Above: parent individuals. Below: offspring. Crossover points are marked by a cross in the parents.

crossover. Most GP systems have a parameter that prevents trees from becoming too deep, thus filling all the available memory and requiring longer evaluation times. There is still some debate among practitioners in the field as to whether one should let the trees breed and grow until the maximum depth or whether to edit and simplify them along the way in order to obtain shorter programs. The argument for larger trees is that the often redundant genetic material has a richer set of breeding possibilities and may lead to increased diversity in successive populations. On the other hand, the use of MDL (Minimum Description Length) principles can give rise to compact and efficient solutions in some cases [21]. The issue is difficult to settle due to our currently limited knowledge about the dynamics of the evolution of program populations.

Plain GP works well for problems that are not too complex and that give rise to relatively short programs. To extend GP to more complex problems some hierarchical principle has to be introduced. In any problem-solving activity hierarchical considerations are needed to produce economically viable solutions. This is true in classical top-down design where some form of divide-and-conquer strategy is routinely used to decompose the problem into manageable subproblems. The same considerations are also useful when working bottom-up, as in artificial evolutionary methods. It has been observed by several researchers that during evolution some subtrees appear repeatedly within the population as parts of successful individuals. These trees that seem to perform a useful function might be identified, encapsulated into modules, and reused as single units in the evo-

lutionary process. These modules might be considered as the analogous of the building blocks described in section 5 although the similarity is only qualitative. Methods for automatically identifying and extracting useful modules within GP have been discussed by Koza under the name of Automatically Defined Functions (ADF) [20] and by Angeline and Kinnear [20].

In conclusion, GP has been empirically shown to be quite a powerful program-induction methodology. However, it remains to be seen whether the approach can be extended to automatically evolve programs for more difficult tasks and for general programming.

8 Parallel Evolutionary Algorithms

Parallel computing in its various forms is becoming a key computer technology, driven by an increasing demand for better performance and productivity. In principle, these goals can be met by adding processors, memory and an interconnection network and putting them to work together on a given problem. By sharing the workload, it is hoped that an N-processor system will do the job nearly N times faster than a uniprocessor system, thereby allowing researchers to treat larger and more interesting problem instances. In reality, things are not so simple since several overhead factors contribute to significantly lower the theoretical performance improvement expectations. Furthermore, general parallel programming models turn out to be difficult to design due to the large architectural space that they must span and to the resistance represented by current programming paradigms and languages. In any event, parallel computing is becoming more and more ubiquitous and there exist many important problems that are sufficiently regular in their space and time dimensions to be suitable for it. Fortunately, evolutionary algorithms belong to this class of "easy" parallel problems.

There are two main reasons for parallelizing an evolutionary algorithm: one is to achieve time savings by distributing the computational effort and the second is to benefit from a parallel setting from the algorithmic point of view, in analogy with the natural parallel evolution of spatially distributed populations.

We will start by describing simple though very useful parallel evolutionary algorithms whereby performance improvements can be obtained without changing the general sequential algorithm schema. In many real-world problems, the calculation of the individual's fitness is by far the most time consuming step of the algorithm. In this case an obvious approach is to evaluate each individual fitness simultaneously on a different processor. If there are more individuals than processors, which is often the case, then the individuals to be evaluated are divided as evenly as possible among the available processors. It is assumed that fitness evaluation takes about the same time for any individual. The other parts of the algorithm are as before and remain centralized. The following is an informal description of the algorithm:

```
produce an initial population of individuals
do in parallel
   evaluate the fitness of all individuals
end parallel do
while termination condition not met do
   select fitter individuals for reproduction
   produce new individuals
   mutate some individuals
   do in parallel
      evaluate the fitness of all individuals
   end parallel do
   generate a new population by inserting some new good
   individuals and by discarding some old bad individuals
end while
```

Another method consists of simultaneously and independently running N copies of the algorithm on the N available processors. The best of the multiple independent runs is then the required result. Since EAs are stochastic, several runs are in general needed anyway to draw statistically significant conclusions. This method is in general to be preferred with respect to a very long single run since improvements are more difficult at later stages of the simulated evolution. The various runs must differ in the generation of the initial population and possibly in the setting of some parameters such as the crossover and mutation rate. Each processor computes for a given number of generations. In practice no communication is needed between processors except for stopping the computation when one processor has satisfactorily solved the problem before the allotted maximal number of generations.

We now turn to more genuinely parallel approaches for evolutionary algorithms. All these find their inspiration in the observation that natural populations tend to possess a *spatial* structure. As a result, so-called *demes* make their appearance. Demes are semi-independent groups of individuals or subpopulations having only a loose coupling to other neighboring demes. This coupling takes the form of the slow migration or diffusion of some individuals from one deme to another. A number of models based on spatial structure have been proposed. The two most important categories are the *island* and the *grid* models.

The *island* model [22,23] features geographically separated subpopulations of relatively large size. Subpopulations may exchange information from time to time by allowing some individuals to migrate from one subpopulation to another according to various patterns. The main reason for this approach is to periodically reinject diversity into otherwise converging subpopulations. When

the migration takes place between nearest neighbor subpopulations the model is called *stepping stone*. Within each subpopulation a standard sequential evolutionary algorithm is executed between migration phases. Figure 4 depicts this distributed model and the following is a high-level algorithmic description of the process:

```
initialize P subpopulations of size N each
generation number := 1
while termination condition not met do
   for each subpopulation do in parallel
       evaluate and select individuals by fitness
       if generation number mod frequency = 0 then
          send K<N best individuals to
          a neighboring subpopulation
          receive K individuals from a
          neighboring population
          replace K individuals in
          the subpopulation
       end if
       produce new individuals
       mutate individuals
   end parallel do
   generation number := generation number + 1
end while
```

Here *frequency* is the number of generations before an exchange takes place. Several individual replacement policies have been described in the literature. One of the most common is for the migrating K individuals to displace the K worst individuals in the subpopulation. It is to be noted that the subpopulation size, the frequency of exchange, the number of migrating individuals, and the migration topology are all new parameters of the algorithm that have to be set in some way. At present there is no rigorous way for choosing them. However, several works have empirically arrived at rather similar topologies and parameter values [22, 24].

In the *grid* or *fine-grained* model [25] individuals are placed on a large toroidal one or two-dimensional grid, one individual per grid location (fig. 5). The model is also called *cellular* [26].

Fitness evaluation is done simultaneously for all individuals and selection, reproduction and mating take place locally within a small neighborhood. In time,

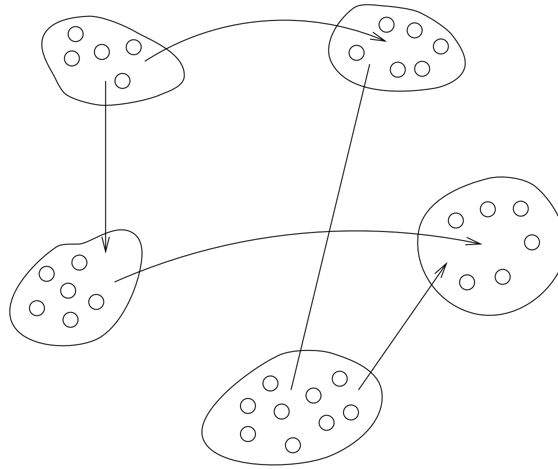

Fig. 4. Illustration of the *Island* model of semi-isolated populations

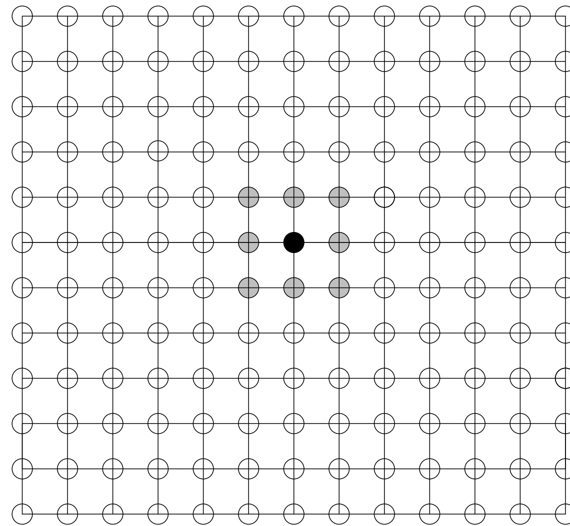

Fig. 5. A 2-D spatially extended population of individuals. A possible neighborhood of an individual (black) is marked in gray.

semi-isolated niches of genetically homogeneous individuals emerge across the grid as a result of slow individual diffusion. This phenomenon is called *isolation by distance* and is due to the fact that the probability of interaction of two individuals is a fast-decaying function of their distance. The following is a pseudo-code description of a grid evolutionary algorithm.

```
for each grid point do in parallel
    generate a random individual
end parallel do
while termination condition not met do
    for each grid point k do in parallel
        evaluate individual in k
        select a neighboring individual q
        produce offspring from k and q
        assign one of the offspring to k
        mutate k with probability p_m
    end parallel do
end while
```

In the preceding description the neighborhood is generally formed by the four or eight nearest neighbors of a given grid point (see fig. 5). The selection of an individual in the neighborhood for mating with the central individual can be done in various ways. Tournament selection is commonly used (see 6.2). This makes full use of the available parallelism and is probably more appropriate from a biological point of view. Likewise, the replacement of the original individual can be done in several ways. For example, it can be replaced by the best among itself and the offspring or one of the offspring can replace it at random. The model can be made more dynamical by adding provisions for longer range individual movement through random walks, instead of having individuals interacting exclusively with their nearest neighbors [26].

Although both island and grid models can be implemented on serial machines, thus comprising useful variants of the standard globally communicating GA, they are ideally suited for parallel computers. From an implementation point of view, coarse-grained island models, where the ratio of computation to communication is high, are more adapted to multiprocessor systems and even for clusters of workstations [27]. Massively parallel SIMD (Single Instruction Multiple Data) machines such as the Connection Machine CM-200 [26] are appropriate for cellular models, since the necessary local communication operations, though frequent, are very efficiently implemented in hardware. Furthermore, it should be noted that hybrid models are also possible; for example, one might consider an island model in which each island is structured as a grid of individuals interacting locally. In a recent work Kapsalis *et al.* [31] have described a software system called the Unified Parallel GA system that allows the user to select one or more parallel GA approaches by setting some parameters.

In general, it has been found that parallel evolutionary algorithms, apart from being significantly faster, help in alleviating the premature convergence problem

and are effective for multimodal optimization. This is due to the larger total population size and to the relative isolation of the spatial regions where solutions start to co-evolve. Both of these factors help to preserve diversity while at the same time promoting local search. As in the sequential case, the effectiveness of the search can be improved at the expense of generality by permitting hill-climbing, i.e. local improvement around promising search points [7].

Until now only the spatial dimension entered into the picture. If we take into account the temporal dimension as well, we observe that parallel evolutionary algorithms can be synchronous or asynchronous. Island models are in general synchronous, using SPMD (Single Program Multiple Data), coarse-grain parallelism in which communication phases synchronize processes. This is not necessary and experiments have been carried out with asynchronous EAs in which subpopulations evolve at their own pace and exchange individuals only when some internally measured level of convergence has been attained. This avoids constraining all co-evolving populations to swap individuals at the same time irrespective of subpopulation evolution.

Fine-grained parallel EAs are fully synchronous when they are implemented on SIMD machines and are an example of data-parallelism.

Genetic programming displays the same general advantages as the other evolutionary algorithms when implemented on parallel architectures. One important difference is that individuals may widely vary in size and complexity. This precludes SIMD implementations of GP both because of the amount of local memory needed to store individuals as well as for efficiency reasons (e.g., sequential execution of different branches of code belonging to individuals stored on different processors). Coarse-grain models such as the island model or the parallel fitness evaluation model are much more suitable to GP. There is still a load-balancing problem to be solved because of the aforementioned variability of individuals in GP. Refs.[28] and [29] describe two different ways of attaining an even load on the different processors by renouncing generational synchronization altogether or by using a simple load-balancing algorithm, respectively.

Triggered by the widespread availability of parallel computers and workstation clusters, parallel and distributed EAs have been used with success for some time. They are simple to implement and offer advantages over the standard sequential algorithm. However, parallelism introduces new degrees of freedom that have to be dealt with by the implementer, their theoretical analysis is more difficult, and very little is known to date about their properties.

Evolutionary Computation Resources

A large amount of useful information on evolutionary algorithms is available on the Internet. The most important GA site can be reached on the web at the following URL:

http://www.aic.nrl.navy.mil/galist

It contains a wealth of information on GA-related activities, conferences, courses and workshops, technical reports, source code and links to related sites. It can also be accessed by anonymous ftp at the following address:

ftp.aic.nrl.navy.mil in /pub/galist

It is possible to subscribe to a "low-noise" GA list (only digests are sent about once a week) by sending a message to: GA-list-request@aic.nrl.navy.mil. Another useful address, with several links to interesting information is the Encore site at:

ftp://ftp.cs.wayne.edu/pub/EC/Welcome.html

Acknowledgement

I thank my colleagues at the Logic Systems Laboratory and the participants to the *Towards Evolvable Hardware* workshop for the stimulating atmosphere and useful discussions. Special thanks to M. Sipper who carefully read the manuscript and helped improve its style.

References

1. A. Thompson, I, Harvey and P. Husbands, *Unconstrained Evolution and Hard Consequences*, this volume.
2. H. Kitano, *Morphogenesis for Evolvable Systems*, this volume.
3. F. Mondada and D. Floreano, *Evolution and Mobile Autonomous Robots*, this volume.
4. G. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison Wesley, Reading, MA, 1989.
5. Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlag, Second Edition, Berlin, 1994.
6. D.B. Fogel, *Evolutionary Computation*, IEEE Press, New York, 1995.
7. H. Mühlenbein, M. Schomish and J. Born, *The Parallel Genetic Algorithm as Function Optimizer*, Parallel Comput. **17**, 619, 1991.
8. G. Syswerda, *Uniform Crossover in Genetic Algorithms*, in Proc. of the Third Int. Conf. on Genetic Algorithms, J. D. Schaffer (Editor), Morgan Kauffman, 2-9, 1989.
9. D.E. Goldberg and J. Richardson, *Genetic Algorithms with Sharing for Multimodal Function Optimization*, in Proc. of the Second Int. Conf. on Genetic Algorithms, J.J. Grefenstette (Editor), Lawrence Erlbaum Associates, Hillsdale, 41-49, 1987.
10. X. Yin and N. Germay, *A Fast Genetic Algorithm with Sharing Scheme Using Cluster analysis Methods in Multimodal Function Optimization*, in Proc. Inter. Conf. Artificial Neural Nets and Genetic Algorithms, Innsbruck, Austria, 450-457, 1993.

11. W. Spears, *Simple Subpopulation Schemes*, Proc. of the Evolutionary Programming Conference, 296-307, 1994.
12. D. Beasley, D.R. Bull and R.R. Martin, *A Sequential Niche Technique for Multimodal Function Optimization*, Evolutionary Computation, **1**, 101-125, 1993.
13. D.W. Hillis, *Co-evolving Parasites Improve Simulated Evolution as an optimization Procedure*, In *Artificial Life II*, C. Langton et al. Editors, Addison-Wesley, 313-324, 1992.
14. M. Sipper, *Co-evolving Non-Uniform Cellular Automata to Perform Computations*, Physica D, To appear, 1995.
15. M. Potter and K. De Jong, *A Cooperative Coevolutionary Approach to Function Optimization*, in Procs. of the Third Conference on Parallel Problem Solving from Nature, Y. Davidor and H.-P. Schwefel Editors, Lecture Notes in Computer Science Vol. 866, Springer-Verlag, 249-257, 1994.
16. P. Husbands, *An Ecosystem Model for Integrated Production Planning*, Intl. Journal of Computer Integrated Manufacturing, **6**, 74-86, 1993.
17. M. Potter and K. De Jong, *Evolving Neural Networks with Collaborative Species*, Procs. of the 1995 Summer Computer Simulation Conference, The Society for Computer Simulation, Ottawa, Canada, 340-345, 1995.
18. L. Davis, *Handbook of Genetic Algorithms*, Van Nostrand, New York, 1991.
19. J. Koza, *Genetic Programming*, MIT Press, Cambridge, MA, 1992.
20. K. Kinnear (Editor), *Advances in Genetic Programming*, MIT Press, 1994.
21. H. Iba, H. de Garis and T. Sato, *Genetic Programming Using a Minimum Description Length Principle*, in [20], 1994.
22. J.P. Cohoon, S.U. Hegde, W.N. Martin and D. Richards: *Punctuated Equilibria: a Parallel genetic Algorithm*, in Proc. of the Second Int. Conf. on Genetic Algorithms, J.J Grefenstette (Editor), Lawrence Erlbaum Associates, 148, 1987.
23. R. Tanese: *Parallel genetic Algorithm for a Hypercube*, in Proc. of the Second Int. Conf. on Genetic Algorithms, J.J Grefenstette (Editor), Lawrence Erlbaum Associates, 177, 1987.
24. T. Starkweather, D. Whitley and K. Mathias: *Optimization Using Distributed Genetic Algorithms*, in Parallel Problem Solving from Nature, Lecture Notes in Computer Science Vol. 496, H.-P. Schwefel and R. Männer (Editors), Springer-Verlag, 176, 1991.
25. B. Manderick and P. Spiessens: *Fine-Grained Parallel Genetic Algorithms*, in Proc. of the Third Int. Conf. on Genetic Algorithms, J. D. Schaffer (Editor), Morgan Kauffman, 428, 1989.
26. M. Tomassini, *The Parallel Genetic Cellular Automata: Application to Global Function Optimization*, Proceedings of the International Conference on Artificial Neural Networks and Genetic Algorithms, Springer-Verlag, Wien, 385, 1993.
27. A. Loraschi, A. Tettamanzi, M. Tomassini and P. Verda, *Distributed Genetic Algorithms with an Application to Portfolio Selection Problems*, in Proceedings of the Int. Conf. on Artificial Neural Nets and Genetic Algorithms,

D.W. Pearson, N.C. Steele and R.F. Albrecht (Editors), Springer-Verlag, 384, 1995.
28. J.R. Koza and D. Andre, *Parallel Genetic Programming on a Network of Transputers*, Computer Science Department, Stanford University, Technical Report CS-TR-95-1542, 1995.
29. M. Oussaidene, B. Chopard and M. Tomassini, *Learning Trading Models Using a Parallel Genetic Programming System*, submitted, 1995.
30. F. Gruau, *Artificial Cellular Development in Optimization and Compilation*, this volume.
31. A. Kapsalis, G.D. Smith and V.J. Rayward-Smith, *A Unified Paradigm for Parallel Genetic Algorithms*, in Evolutionary Computing, AISB Workshop, T.C. Fogarty (Editor), Lecture Notes in Computer Science 865, Springer-Verlag, 1994.

This article was processed using the LaTeX macro package with LLNCS style