Complex Systems **6** (1992) 193–212

Global Optimization of Functions with the Interval Genetic Algorithm

Marco Muselli[*]
Istituto per i Circuiti Elettronici,
Consiglio Nazionale delle Ricerche, 16145 Genova, Italy

Sandro Ridella
Dipartimento di Ingegneria Biofisica ed Elettronica,
Università di Genova, 16145 Genova, Italy

Abstract. A new evolutionary method for the global optimization of functions with continuous variables is proposed. This algorithm can be viewed as an efficient parallelization of the simulated annealing technique, although a suitable interval coding shows a close analogy between real-coded genetic algorithms and the proposed method, called *interval genetic algorithm.*

Some well-defined genetic operators allow a considerable improvement in reliability and efficiency with respect to conventional simulated annealing even on a sequential computer. Results of simulations on Rosenbrock valleys and cost functions with flat areas or fine-grained local minima are reported.

Furthermore, tests on classical problems in the field of neural networks are presented. They show a possible practical application of the interval genetic algorithm.

1. Introduction

The solutions of many important problems, belonging to different scientific fields, derive from the minimization of a suitable cost function with continuous variables. Sometimes the behavior of this function is regular and unimodal, and nonlinear programming techniques [1] can reach good results in a short time.

In other cases, owing to function complexity, these methods are almost useless and global optimization algorithms are required to obtain a satisfactory value for the cost function. Apart from particular situations in which specific methods can be employed [2, 3], iterative random search procedures are frequently a compelling choice for solving the problem.

[*]Electronic mail address: `muselli@aritta.ice.ge.cnr.it`.

In analogy with the behavior of natural organisms random search algorithms have often been called *evolutionary methods*; the generation of a new trial point corresponds to mutation while a step toward the minimum can be viewed as selection. If we call *basepoint* the point used for the current mutation, the term *hard selection* is employed when we choose the optimum up to that moment as the basepoint. On the contrary, when the basepoint is selected in a probabilistic way inside a population of points or between two consecutive points reached by the algorithm, we refer to this process as *soft selection* [4].

Devroye [5] has shown that under weak conditions hard selection methods always converge to the global minimum of the cost function. Unfortunately, in many cases these random searches are not applicable since the convergence time is practically infinite.

A well-known soft selection technique is the *simulated annealing* algorithm, widely used in combinatorial problems [6]. It comes from statistical mechanics, and its convergence properties have a theoretical foundation [7]. The application of simulated annealing to optimization problems with continuous variables has been the object of many publications. In some cases the simple implementation used in combinatorial optimization has been changed radically in order to maintain the theoretical validity of the resulting algorithms [8, 9]; in other cases, simple heuristic methods with better convergence time have been developed [10].

Other evolutionary techniques with soft selection search for the global minimum by proceeding with a population of points [4]. In combinatorial optimization a great interest has been excited by the class of *genetic algorithms* [11, 12], which localize the optimum by repeatedly applying some well-suited random operators. These *genetic operators* include not only selection and mutation, but also other more complex string manipulators.

The algorithm we propose in this paper is essentially an evolutionary method in which the selection follows the Metropolis criterion [13], widely used in classical simulated annealing. Furthermore, the introduction of some particular genetic operators considerably speeds up the convergence to the global minimum. In section 2 some fundamental concepts of simulated annealing are reviewed and possible accelerations are emphasized. Section 3 analyzes the similarities between the proposed method and classical genetic algorithms, while section 4 deals with the details of implementation. Simulation results and discussions are reported in section 5.

2. Simulated annealing

Let us define the optimization problem we are dealing with. Let the function $f(\boldsymbol{x})$ be defined on a domain $D \subset \mathcal{R}^n$. We want to find one of its global minima $\boldsymbol{x}_{\text{opt}} \in D$ for which:

$$f(\boldsymbol{x}_{\text{opt}}) \leq f(\boldsymbol{x}) \qquad \forall \boldsymbol{x} \in D \tag{2.1}$$

Let $(\boldsymbol{x})_i$ denote the ith component of point $\boldsymbol{x}$; we use the equivalent notation x_i when it is not ambiguous.

The function $f(\boldsymbol{x})$ must be measurable and bounded on D, but it may be neither continuous nor differentiable; in such a situation methods that use information on the gradient are not applicable. For sake of simplicity, we consider domains D of the following form:

$$D = \{\boldsymbol{x} \in \mathcal{R}^n : \boldsymbol{a} \leq \boldsymbol{x} \leq \boldsymbol{b} \;,\; \boldsymbol{a}, \boldsymbol{b} \in \mathcal{R}^n\} \tag{2.2}$$

where $\boldsymbol{a} \leq \boldsymbol{x} \leq \boldsymbol{b}$ corresponds to $a_i \leq x_i \leq b_i$ for $i = 1, 2, \ldots, n$, even though the presented optimization algorithms also works on functions with more complex domains.

A direct application of classical simulated annealing to the problem above could be realized as follows. The domain D is subdivided into r regions that are indexed in progressive order, and every region must be small enough that the function $f(\boldsymbol{x})$ is almost constant in its interior. Thus, we can associate with the jth region, for $j = 1, \ldots, r$, the value $f(\boldsymbol{x}_j)$ at a particular point $\boldsymbol{x}_j$ belonging to it.

Our general optimization problem is then solved by determining a region j_{opt} for which

$$f(\boldsymbol{x}_{j_{\text{opt}}}) \leq f(\boldsymbol{x}_k) \qquad k = 1, \ldots, r \tag{2.3}$$

It must be pointed out that every practical problem can be put in this form owing to the limited precision offered by a computer.

We are now dealing with a combinatorial optimization problem to which classical simulated annealing can be applied. For every region j, a set S_j, called the *neighborhood set*, is defined. S_j contains the indexes of the regions that are "close" to the jth in some sense (according to a problem-dependent criterion). A possible choice in our case is the following:

$$S_j = \{k : \|\boldsymbol{x}_k - \boldsymbol{x}_j\| \leq \delta\} \tag{2.4}$$

where the parameter $\delta > 0$ can depend on j.

The algorithm starts in a particular region h_0, then it continuously tries to find a better solution by searching the neighborhood set of the current region. In practice, if h_i is the region considered at the ith iteration, a step of the simulated annealing is formed by the following two actions:

Mutation A region $k \in S_{h_i}$ is chosen with uniform probability.

Selection The region h_{i+1} for the next iteration is obtained by applying the Metropolis criterion [6]: if $\Delta f = f(\boldsymbol{x}_k) - f(\boldsymbol{x}_{h_i})$, a random number $\xi \in [0, 1]$ is taken and

$$h_{i+1} = \begin{cases} k & \text{if } \xi \leq e^{-\Delta f/T} \\ h_i & \text{if } \xi > e^{-\Delta f/T} \end{cases} \tag{2.5}$$

where T is a control parameter, called *temperature*.

The convergence to the region j_{opt} (and consequently to the global minimum of the cost function) is guaranteed [14] if the temperature T is decreased at fixed intervals (*annealing schedule*). The sequence of values T_t, $t = 1, 2 \ldots$, must satisfy the relations

$$\lim_{t \to \infty} T_t = 0 \qquad \text{and} \qquad T_t \geq \frac{c}{\log t} \tag{2.6}$$

for a large constant $c \in \mathcal{R}$. Moreover, enough time must be spent at each temperature T_t to reach thermal equilibrium. In practice, an annealing schedule satisfying (2.6) is too slow; the search for adequate annealing schedules is still a subject of study.

The accuracy in finding the global minimum depends on the number r of regions into which the domain D has been subdivided. Such a number rapidly increases with the dimension n of D, leading to excessive search time. The number of iterations needed for the convergence can be reduced considerably by removing the constraint of fixed neighborhood sets. In particular, if the amplitude in (2.4) changes dynamically, the optimization process can be adaptive; thus, the search can be rough at first and subsequently more refined [10]. Unfortunately, this approach does not meet the requirements for the theoretical convergence to the global minimum; nevertheless, modifications of this kind and changes to the annealing schedule are important for a practical application of the method.

A further increase in convergence speed can be obtained by an efficient implementation on parallel computers. From a detailed analysis of this problem [7, pages 95–114] arises the sequential nature of simulated annealing; in fact, a good efficiency can be achieved only with a high interaction among the processors. In the next section we propose a simple parallelization of simulated annealing that leads to a fast and reliable optimization method, even if it does not satisfy the convergence theorems mentioned above.

3. The interval genetic algorithm

The most common way to parallelize an evolutionary method is by performing the search using a population of points; at every iteration the individuals are updated by applying a hard or soft selection mechanism. Among the methods of this kind, the class of genetic algorithms [11, 12] turns out to be very interesting. They simulate natural selection and recombination in order to obtain a high degree of robustness in the search.

A genetic algorithm is generally defined by four components:

1. A chromosomal representation of the solution space (domain D).
2. A method of creating the initial population.
3. A method of assigning a cost function value to the chromosomal representation.
4. Some genetic operators that cause the evolution of the population.

Let us analyze in detail the choices that lead to the definition of the interval genetic algorithm.

3.1 Chromosomal representation

Some theoretical considerations regarding the convergence mechanisms of genetic algorithms advise us to perform the search for the global minimum not in the domain D, but in an associated space. A suitable coding combines the elements of such space, called *chromosomes*, with the corresponding points of D. Such a coding is then called *chromosomal representation*.

The schema theory proposed by Holland [11] and in particular the principle of minimal alphabet give valid motivations for the use of binary strings as chromosomes. In this way the number of schemata available for genetic processing is maximized. On the other hand, other considerations, such as the advantage of a more natural problem coding or the necessity of a better accuracy in the location of minimum, lead to real chromosomes. An interesting analysis of this controversy can be found in [15], where Goldberg lays the foundations for a theoretical understanding of the efficiency of real genetic algorithms.

In the interval genetic algorithm the following coding choice has been made: every point $\boldsymbol{x} \in D$ is associated with an interval of the type

$$[\boldsymbol{x}]_\delta = \{\boldsymbol{y} \in D : |y_i - x_i| \leq \delta_i,\ i = 1, \ldots, n\} \tag{3.1}$$

where x_i, y_i, and δ_i are the ith components of $\boldsymbol{x}$, $\boldsymbol{y}$, and $\boldsymbol{\delta}$, respectively. The parameter $\boldsymbol{\delta}$ is called *amplitude*, and changes in an adaptive way during the search.

The choice of the coding (3.1) is based on the following considerations:

1. Any optimization method finds at every iteration a point $\boldsymbol{x}^*$, which is an approximation of the global minimum $\boldsymbol{x}_{\text{opt}}$. If $\boldsymbol{\delta}$ refers to the error made in this approximation, then we have

   $$\boldsymbol{x}_{\text{opt}} \in [\boldsymbol{x}]_\delta \tag{3.2}$$

   As the algorithm continues, it finds new optimal points with lower cost function values, and the error $\boldsymbol{\delta}$ changes with time. Thus, at any iteration we can associate with every trial point an uncertainty interval $[\boldsymbol{x}]_\delta$ containing the global minimum $\boldsymbol{x}_{\text{opt}}$.

2. The concept of schema in the classical theory of genetic algorithms [12] has a natural correspondence in the interval coding (3.1). Suppose we use binary strings with five bits for coding numbers in the range $[0, 32]$. We have implicitly chosen a minimum approximation of the location of every point. Consequently, we can know the global minimum x_{opt} only within an error $\delta = 0.5$.

   Now, consider the schema 0*11* (where * is the *don't care* symbol). The schema corresponds to the strings 00100, 00111, 01110, and 01111, or, with the normal binary coding, to the points 6.5, 7.5, 14.5, and 15.5. But, as previously noted, the minimum approximation is $\delta = 0.5$, so this schema identifies the following union of intervals:

$$0{*}11{*} \rightarrow [6,7] \cup [7,8] \cup [14,15] \cup [15,16] = [6,8] \cup [14,16] \qquad (3.3)$$

Thus, the global minimum of the cost function is located by the schema $0{*}11{*}$ at most within an error $8\delta = 4$.

Since the use of disjointed intervals for a single schema is not a theoretical constraint, we can alternatively deal with contiguous-interval schemata having the same uncertainty (e.g., $[6, 10]$). Therefore, from definition (3.1) we find that a point is associated with all the schemata (intervals) centered in it.

From this particular kind of coding derives the name *interval genetic algorithm*, used for the proposed optimization method.

This chromosomal representation is analogous to the concept of virtual characters introduced by Goldberg [15]. Both definitions refer to subsets of the domain D that contain the attraction basins of the cost function minima. The use of monodimensional slice for the definition of virtual characters is motivated by the exponential reduction in the selection probability for high-dimensional intervals. Such a problem can be overcome by adaptively changing the amplitude $\boldsymbol{\delta}$ in the optimization process. Codings of the type in (3.1) are called *compound virtual characters* in [15], and the possible advantages deriving from their use are mentioned in a footnote.

Finally, the interval coding overcomes (at least partially) the blocking problem widely examined in [15]. In fact, the minimization is simultaneously performed on all of the function variables; this prevents a search in a given direction from hiding the true position of the global minimum. The overcoming of blocking is shown by the simulation results (section 5).

3.2 Initial population and cost function value

Let $[\boldsymbol{x}_j]_{\boldsymbol{\delta}_j}$ denote the jth interval, with $j = 1, \ldots, m$, of a generic population in the interval genetic algorithm (m is the population size). At the first iteration the centers $\boldsymbol{x}_j$ are randomly chosen in D, whereas the amplitudes $\boldsymbol{\delta}_j$ must be wide enough to include the whole domain D. The initial uncertainty in the knowledge of the global minimum is actually the highest; thus a possible choice for $\boldsymbol{\delta}_j$ at the first iteration is

$$\boldsymbol{\delta}_j = \boldsymbol{b} - \boldsymbol{a} \qquad j = 1, \ldots, m \qquad (3.4)$$

where $\boldsymbol{a}$ and $\boldsymbol{b}$ are the bounds of D in (2.2).

The choice of a cost function value for the generic interval $[\boldsymbol{x}]_{\boldsymbol{\delta}}$ is also of great importance. An estimate of the minimum for $f(\boldsymbol{x})$ in $[\boldsymbol{x}]_{\boldsymbol{\delta}}$ could allow efficient selection of the most promising intervals in the search for the global optimum. Such an estimate could be obtained through some steps of a local optimization algorithm. Nevertheless, as a first approximation we have omitted this possibility and have used the value $f(\boldsymbol{x})$ in the center of the interval $[\boldsymbol{x}]_\delta$. In this way we can verify the effectiveness of the proposed method without possible (and useful) accelerations coming from fast local optimization techniques.

At this point we want to emphasize that the interval genetic algorithm does not require a suitable function scaling, as most genetic algorithms do. Although the presented results refer to positive functions, negative values are equally allowed.

3.3 Genetic operators

The evolution of the population is caused by the repeated application of five genetic operators: reproduction, crossover, merging, mutation, and selection. The *reproduction* operator chooses inside the current population two intervals $[\boldsymbol{x}]_\delta$ and $[\boldsymbol{y}]_\varepsilon$, which will be used for the generation of a new offspring. In the interval genetic algorithm this choice is made by suitably applying the Boltzmann distribution

$$p\left([\boldsymbol{x}]_\delta\right) \propto \exp\left(-\frac{f(\boldsymbol{x})}{T}\right) \tag{3.5}$$

where $f(\boldsymbol{x})$ is the cost function value corresponding to the interval $[\boldsymbol{x}]_\delta$.

Besides its importance from a physical point of view, the Boltzmann distribution yields two interesting properties:

- It does not need a cost function scaling; negative values of $f(\boldsymbol{x})$ can be used directly in (3.5).
- The temperature value regulates the choice made by the reproduction operator, as in the Metropolis criterion. In fact, if $T \ll f(\boldsymbol{x})$, all the population intervals will have the same probability of being chosen; in contrast, if $T \gg f(\boldsymbol{x})$, only the intervals with lower cost function values will be selected.

Thus, by using an annealing process for the temperature T, the search for the global minimum is initially uniform on the whole domain D and subsequently dwells upon the most promising population intervals.

The *crossover* operator generates a new interval $[\boldsymbol{z}]_\gamma$ starting from the points $[\boldsymbol{x}]_\delta$ and $[\boldsymbol{y}]_\varepsilon$ chosen by reproduction. It is based on the following relations:

$$z_i = \begin{cases} x_i & \text{with prob. } 0.5 \\ y_i & \text{with prob. } 0.5 \end{cases} \qquad \gamma_i = \begin{cases} \delta_i & \text{with prob. } 0.5 \\ \varepsilon_i & \text{with prob. } 0.5 \end{cases} \tag{3.6}$$

These two relations are tightly bound to each other: the probabilistic choice is the same for the two assignments. So, if $z_i = x_i$ then $\gamma_i = \delta_i$; likewise, if $z_i = y_i$ then $\gamma_i = \varepsilon_i$. Such an operator can be viewed as a direct extension of a discrete multiple-point crossover.

From (3.6) we find that crossover is useless for $n = 1$ since the offspring is equivalent to a parent interval. For $n \geq 2$ this operator considerably increases the convergence speed; in fact, the component swapping helps overcome local minima and avoid the blocking problem [15].

The *merging* operator is applied as an alternative to crossover, and generates a new offspring $[\boldsymbol{z}]_\gamma$ starting from two parents $[\boldsymbol{x}]_\delta$ and $[\boldsymbol{y}]_\varepsilon$ chosen by reproduction. Its purpose is to join the information contained in the intervals $[\boldsymbol{x}]_\delta$ and $[\boldsymbol{y}]_\varepsilon$. The merging operator is performed in the following way:

$$[\boldsymbol{z}]_\gamma = [\boldsymbol{x}]_\delta \cap [\boldsymbol{y}]_\varepsilon \tag{3.7}$$

taking into account that if the intersection is empty, the operator is not applied.

With the simple definition (3.7), merging points out the most promising regions of the domain D that the algorithm has encountered in its execution. Therefore, such an operator makes a synthesis rather than a real search, and its application probability must be kept small.

The *mutation* operator searches an interval $[\boldsymbol{x}]_\delta$ for a better point $\boldsymbol{y}$. The interval $[\boldsymbol{x}]_\delta$ has been obtained directly by reproduction or derives from the application of crossover or merging. The amplitude $\boldsymbol{\delta}$ is not modified by mutation since the uncertainty in the location of the global minimum is not affected by this operator.

The choice of the point $\boldsymbol{y}$ can be done in two ways:

- *At random:* $\boldsymbol{y}$ is chosen inside the interval $[\boldsymbol{x}]_\delta$ according to a given probability distribution.
- *By minimization:* $\boldsymbol{y}$ is obtained through some steps of a local optimization method.

In analogy with mutation in the classical simulated annealing, the interval genetic algorithm uses a uniform probability distribution for the choice of the point $\boldsymbol{y}$. Since the search process is essentially based on the mutation operator, this is applied at every iteration.

Through the repeated application of reproduction, crossover, merging, and mutation, m new intervals $[\boldsymbol{x}_{m+j}]_{\delta_{m+j}}$, with $j = 1, \dots, m$, are generated starting from the current population $[\boldsymbol{x}_1]_{\delta_1}, \dots, [\boldsymbol{x}_m]_{\delta_m}$. The *selection* operator chooses among these $2m$ intervals the m individuals that will form the next population. It applies the Metropolis criterion in the same way as classical simulated annealing does. If $[\boldsymbol{y}_j]_{\varepsilon_j}$ is the jth interval of the next population, we have

$$[\boldsymbol{y}_j]_{\varepsilon_j} = \begin{cases} [\boldsymbol{x}_{m+j}]_{\delta_{m+j}} & \text{if } \xi_j \leq p_j \\ [\boldsymbol{x}_j]_{\delta_j} & \text{if } \xi_j > p_j \end{cases} \tag{3.8}$$

where ξ_j is a random number in the range $[0, 1]$, and p_j is given by

$$p_j = \exp\left(-\frac{f(\boldsymbol{x}_{m+j}) - f(\boldsymbol{x}_j)}{T}\right) \qquad j = 1, \dots, m \tag{3.9}$$

The Metropolis criterion (3.8) has theoretical foundations that ensure the convergence of the simulated annealing algorithm to the global minimum under suitable conditions [7]. In any case it is an interesting method for controlling the behavior of the optimization process. In general, the Metropolis criterion prefers configurations with lower cost function values, but allows local minima to be overcome by accepting uphill moves. Therefore, its application has great importance even from the point of view of implementation.

4. Implementation

The interval genetic algorithm proceeds by successively applying reproduction, crossover, merging, mutation, and selection until it satisfies the stopping criterion. While reproduction, mutation, and selection are performed at every iteration, crossover and merging have a corresponding application probability denoted by p_C and p_M, respectively. From the peculiarities of these operators we directly obtain the relation $p_M \ll 1$, whereas the inequality $p_C \leq 0.5$ prevents the destruction of promising intervals that the mutation operator has still not examined.

A simple trick in the reproduction operator increases the convergence speed of the method. Let $[\boldsymbol{x}_{j^*}]_{\delta_{j^*}}$ be the interval in the current population with the minimum cost function value. The following quantities are evaluated:

$$\eta_j = \exp\left(-\frac{f(\boldsymbol{x}_j) - f(\boldsymbol{x}_{j^*})}{T}\right) - \xi_j \qquad j = 1, \ldots, m \tag{4.1}$$

where ξ_j is a random number in the range $[0, 1]$. The two intervals with minimum values of η_j are then returned by the reproduction operator.

4.1 Temperature

As previously noted, the parameter temperature controls the application of reproduction and selection. A variety of considerations and the analogy with simulated annealing advise us to lower this parameter starting from a suitable initial value. However, determining the proper annealing schedule for a given problem is frequently a matter of trial and error, even if it is a decision of great importance.

For example, an initial temperature that is too high leads to a pure random search with a possible waste of iterations; if the initial temperature is too low, the algorithm can get stuck in a local minimum. A simple method for an adaptive choice of the annealing schedule is comparing, at regular intervals, the current temperature with the differences between the values of $f(\boldsymbol{x})$ in the population.

In the interval genetic algorithm the value of T is updated every N_T iterations according to the following relation:

$$T(k + N_T) = T(k)/\alpha_T \tag{4.2}$$

where $T(k)$ is the temperature at the kth iteration and $\alpha_T > 1$ is a fixed constant. Every time (4.2) is applied, the term $T(k + N_T)$ is compared with $T_{\min} \cdot G$, where $T_{\min} \ll 1$ is a small parameter and G is the geometric mean of the differences between the cost function values in the population intervals and the minimum value $f(\boldsymbol{x}^*)$ up to the moment,

$$G = \left(\prod_{j=1}^{m} (f(\boldsymbol{x}_j) - f(\boldsymbol{x}^*)) \right)^{1/m} \tag{4.3}$$

If $T(k + N_T) < T_{\min} \cdot G$, the algorithm sets

$$T(k + N_T) = \max(G, |f(\boldsymbol{x}^*)|) \tag{4.4}$$

and continues its search. By means of this control a small value for N_T can be used without having too low a temperature T during the optimization process. The relation (4.4) is also used for initializing the value of T.

4.2 Amplitudes

The amplitudes $\boldsymbol{\delta}_j$ in the population intervals are updated in a similar way: every N_δ iterations the components of $\boldsymbol{\delta}_j$ are multiplied (divided) by a fixed constant $\alpha_\delta > 1$ if the optimum value is improved (is not improved). When

$$(\boldsymbol{\delta}_j)_i < \delta_{\min} \cdot x_i^* \qquad i = 1, \ldots, n \text{ and } j = 1, \ldots, m \tag{4.5}$$

where $\boldsymbol{x}^*$ is the current optimum point, all the amplitudes are reset to the initial value (3.4). $\delta_{\min}$ is a positive constant that contains the desired maximum approximation in the location of the global minimum.

N_r consecutive amplitude resets are allowed without variations of the current optimum; afterwards, the search is stopped.

Unfortunately, this updating mechanism leads to a common value for each amplitude component and for each population interval. This problem can be avoided by using a small correction. Every time a population interval $[\boldsymbol{x}_j]_{\boldsymbol{\delta}_j}$ has a lower cost function value than the current optimum $\boldsymbol{x}^*$, its amplitude $\boldsymbol{\delta}_j$ is changed according to the following assignment:

$$(\boldsymbol{\delta}_j)_i = (\boldsymbol{\delta}_j)_i \cdot \left(1 + \frac{|x_i^* - (\boldsymbol{x}_j)_i|}{m \cdot \max_i |x_i^* - (\boldsymbol{x}_j)_i|} \right) \tag{4.6}$$

In this way the amplitudes are increased mostly in the direction of current optimum.

4.3 Parallelization and simulated annealing

The implementation of the interval genetic algorithm on a parallel computer with a shared memory is almost immediate. In fact, the only global operations are the updating of the amplitudes (every N_δ iterations) and temperature (every N_T iterations). Thus, it is possible to assign the generation of a new interval to a different processor, which consecutively performs reproduction, crossover, merging, and mutation. This can be done in an asynchronous way, by loading and storing information in the common memory.

Only the selection operator must be performed by the processors at the same time, after the generation of m new intervals. The temperature and amplitude updating can be assigned to one of the processors considered as a master.

With this simple implementation we can reach a speed-up that is close to m, the population size (if the number of processors is greater than m). Such a result does not take into account possible parallelizations in the cost function evaluation.

If we leave out the important contributions of reproduction, crossover, and merging, the interval genetic algorithm becomes a simple execution of m simulated annealings [10]. We have already pointed out that the neighborhood sets change during the optimization process, leading to inhomogeneous Markov chains even if the temperature is kept constant. The classical theory is then no more applicable, and the convergence to the global minimum is not theoretically ensured. On the other hand, the results presented in [10] shows that such simulated annealing algorithm has a good reliability.

The addition of reproduction, crossover, and merging simply introduces other variations in the neighborhood set structure. Thus, the interval genetic algorithm can be viewed as a possible parallelization of simulated annealing. Goldberg [16] has shown that the application of a more complex selection operator leads to a Boltzmann distribution across the population in a discrete genetic algorithm. A similar operator can probably be defined for the interval coding.

5. Tests and results

To analyze the features of the interval genetic algorithm (IGA), we have made some tests on different cost functions. We have already noted the intrinsic parallelism of the method, but it is important to know its performance on a sequential computer in comparison with other algorithms.

We have chosen for this purpose two well-known optimization techniques: simulated annealing (SA) [10], which is slow but reliable; and the Powell method (PM) [17], a fast local optimization algorithm. We have not used discrete genetic algorithms since the desired precision requires too long strings; moreover, different coding and function scaling can lead to high variations in the results.

Three groups of tests were made on conventional functions, and two other groups concerned the application of optimization in the field of neural networks. In the latter case we considered the back-propagation (BP) algorithm [18], widely used in practical problems, instead of the Powell method.

For every test function we took four values for the dimension n of the domain D, and for each dimension we made 50 runs of each algorithm. This led to a reasonable set of statistics for the convergence speed, measured by the mean time (in terms of evaluations of the cost function) needed for satisfaction of the stopping criterion.

If any of the fifty runs exceeded ten million evaluations, that search was aborted and the remaining runs skipped. Ten million evaluations was therefore the limit for any single search (indicated in the tables of results with the notation $> 10M$). Moreover, when a method converged at a local minimum, it was stopped and restarted at a new point; all the initial points were chosen randomly with a uniform probability inside the cost function domain D.

Some preliminary runs (which are not taken into account in the tables of results) were made to obtain good values for the parameters of each algorithm. Such values were then kept constant throughout the tests, except for the initial temperature T_0 of SA. In fact, for some test functions a constant T_0 did not allow the convergence of SA for all the chosen dimensions n.

The parameters of IGA were assigned the following values:

$m = 20$	$N_T = 200$	$N_\delta = 100$
$p_C = 0.2$	$\alpha_T = 1.5$	$\alpha_\delta = 2$
$p_M = 0.005$	$T_{\min} = 0.001$	$N_r = 50$

$\delta_{\min}$ was set to different values for tests on conventional functions and those in the field of neural networks, since the desired accuracy in the location of the global minimum is different for the two cases. Thus, we chose:

- $\delta_{\min} = 10^{-6}$ for tests on conventional cost functions.
- $\delta_{\min} = 0.1$ for tests on neural networks.

5.1 Rosenbrock function

The first group of tests refers to the Rosenbrock function, which represents a classical test for optimization algorithms. It is defined in the following way:

$$f(\boldsymbol{x}) = \sum_{i=1}^{n-1} 100 \cdot (x_{i+1} - x_i^2)^2 - (1 - x_i)^2 \qquad (5.1)$$

for $n \geq 2$. This function has a single local-global minimum at the point $\boldsymbol{x}_{\text{opt}}$ having components

$$(\boldsymbol{x}_{\text{opt}})_i = 1 \qquad i = 1, \ldots, n \qquad (5.2)$$

A three-dimensional sketch of this cost function is presented in figure 1.

We chose four values of n, $n = 2$, 4, 6, and 8, while the bounds for the domain D were

$$a_i = -1000 \quad \text{and} \quad b_i = 1000 \qquad i = 1, \ldots, n \qquad (5.3)$$

The convergence criterion we adopted is

$$\max_{1 \leq i \leq n} |(\boldsymbol{x}^*)_i - (\boldsymbol{x}_{\text{opt}})_i| < 10^{-3} \qquad (5.4)$$

which depends only on the optimal point $\boldsymbol{x}^*$ found by the algorithm.

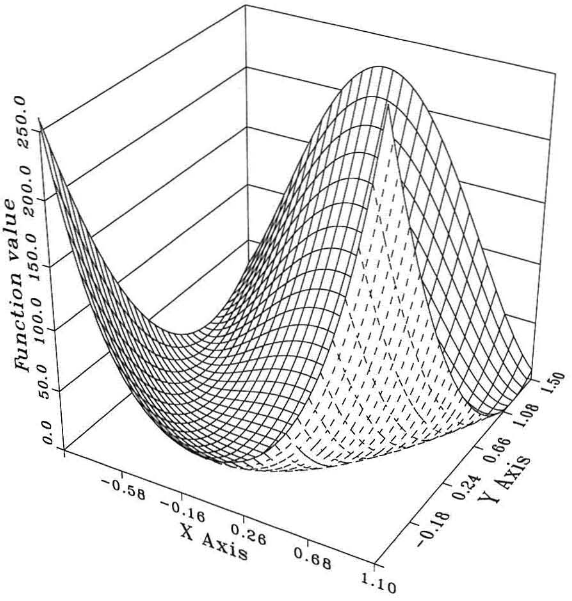

Figure 1: Two-dimensional representation of the Rosenbrock function.

Rosenbrock function				
n	2	4	6	8
Algorithm	*Evaluations performed*			
IGA	28,595	319,428	536,409	723,138
SA	627,047	1,331,321	3,317,919	$> 10M$
PM	2,905	6,277	12,545	26,965

Table 1: Comparative simulation results on the Rosenbrock function.

The simulation results are reported in table 1. In this case the test function is unimodal and differentiable in the whole domain D. Thus, local optimization algorithms like PM are considerably faster than iterative random search procedures (like IGA and SA), as shown in table 1. IGA, particularly suited for the optimization of multimodal cost functions, is slackened by its major complexity, but it is considerably faster than SA.

5.2 Plateau function

From [19] we obtained two test functions with continuous variables having relevant complexity. The first of them is called the *plateau function* and is

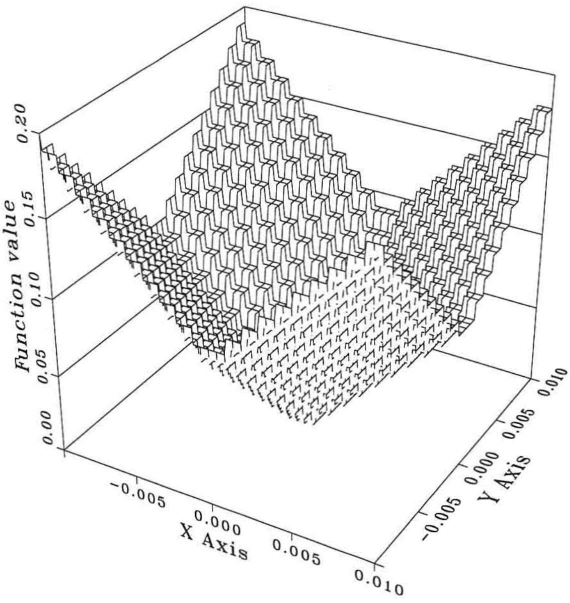

Figure 2: Two-dimensional representation of the plateau function.

defined by

$$f(\boldsymbol{x}) = \sum_{j=1}^{4} 2500 \cdot \max_{i} \lfloor 1000 \cdot |x_i| \rfloor \qquad \text{for } (j-1)h < i \leq jh \tag{5.5}$$

where $\lfloor y \rfloor$ denotes the truncation of y, and $h = n/4$. For the sake of simplicity, we took values for n that are multiples of 4.

As one can see, the plateau function is formed by a large number of flat regions whose values gradually decrease toward the global minimum. All the points $\boldsymbol{x}$ for which

$$\max_{1 \leq i \leq n} |x_i| < 10^{-3} \tag{5.6}$$

are actually global minima having $f(\boldsymbol{x}) = 0$; each optimization algorithm converges when it finds one of these points.

When two variables were kept constant, we obtained for the plateau function with $n = 4$ the plot presented in figure 2. The bounds for the domain D are given by (5.3); the results for $n = 4$, 8, 12, and 16 are shown in table 2.

In this group of tests, PM was stopped in the flat areas of the cost function; thus, its performance degraded from $n = 4$ to $n = 16$, where the limit of ten million was reached. In contrast, the number of evaluations for IGA increased almost linearly with n, improving the values obtained for SA. Despite its slowness, SA always converged, emphasizing its reliability.

Plateau function				
n	4	8	12	16
Algorithm	*Evaluations performed*			
IGA	11,238	18,168	29,976	52,764
SA	745,478	1,585,093	2,436,362	3,477,751
PM	319	10,938	357,735	$> 10M$

Table 2: Comparative simulation results on the plateau function.

Porcupine function				
n	2	4	6	8
Algorithm	*Evaluations performed*			
IGA	34,124	18,214	21,715	40,708
SA	$> 10M$	$> 10M$	$> 10M$	$> 10M$
PM	$> 10M$	$> 10M$	$> 10M$	$> 10M$

Table 3: Comparative simulation results on the porcupine function.

5.3 Porcupine function

A second interesting test function, derived from [19], is called the *porcupine function*, and is defined by

$$f(\boldsymbol{x}) = 10000 \cdot (c + 1.5z) \tag{5.7}$$

where

$$c = 10^{-3} \sum_{i=1}^{n} |x_i| \qquad \text{and} \qquad z = 10^6(n-c) - 2 \cdot \left\lfloor \frac{10^6(n-c)}{2} \right\rfloor \tag{5.8}$$

The truncation in (5.8) causes the particular behavior of this function which has a huge number of local minima in its domain (i.e., every time $\lfloor 10^6(n-c) \rfloor$ is an even number). The value of $f(\boldsymbol{x})$ at these points slowly decreases toward the global minimum $\boldsymbol{x}_{\text{opt}}$ in the axes' origin having $f(\boldsymbol{x}_{\text{opt}}) = 0$. The behavior of the porcupine function is presented in figure 3 for $n = 2$.

Also in this group of tests the bounds for the domain D are given by (5.3), and the convergence criterion is (5.4). The results for $n = 2$, 4, 6, and 8 are reported in table 3.

The high density of local minima stopped both SA and PM. In contrast, IGA always converged within a small number of cost function evaluations, which shows the reliability and efficiency of IGA. IGA searches for the global minimum using a population of points, allowing exchange of information on the cost function behavior among the individuals.

5.4 Parity function

Supervised learning of neural networks is an important application for the optimization techniques. The choice of a weight matrix that minimizes the

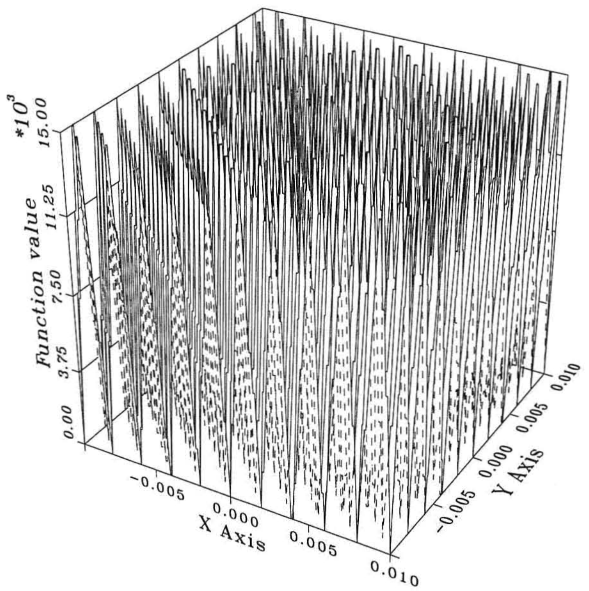

Figure 3: Two-dimensional representation of the porcupine function.

error for a given training set has indeed great practical interest. Unfortunately, the cost functions deriving from this problems have in general a high number of local minima; thus, many trials starting from different initial points are often required to obtain correct values for the network weights.

We have chosen two classical problems that are widely used for testing the learning rules. The first concerns the training of a two-layer feed-forward network that has to generate the parity bit for a binary string of length q. It is known that q hidden neurons are needed for doing such operations [20], so the optimization algorithm must find the $(q+1)^2$ values for the weights of the networks (including the biases).

We considered cases with $q = 2$, 3, 4, and 5, where the corresponding cost functions have, respectively, $n = 9$, 16, 25, and 36 variables and are defined in the following way:

$$f(\boldsymbol{x}) = \sum_{j=1}^{2^q} (t_j - o_j(\boldsymbol{x}))^2 \tag{5.9}$$

$t_j = \pm 1$ is the correct output for the jth input pattern, while $o_j(\boldsymbol{x})$ is the output obtained by using the weights contained in $\boldsymbol{x}$.

The neurons have a hyperbolic tangent transfer function:

$$g(y) = \tanh y = \frac{e^y - e^{-y}}{e^y + e^{-y}} \tag{5.10}$$

Parity function				
n	9 ($q = 2$)	16 ($q = 3$)	25 ($q = 4$)	36 ($q = 5$)
Algorithm	*Evaluations performed*			
IGA	457	1,912	312,438	148,113
SA	10,760	519,936	$> 10M$	$> 10M$
BP	889	20,913	$> 10M$	$> 10M$

Table 4: Comparative simulation results on the parity function.

where y is the neuron input and $g(y)$ the corresponding output.

In this group of tests the domain D had the following bounds:

$$a_i = -10 \quad \text{and} \quad b_i = 10 \qquad i = 1, \ldots, n \tag{5.11}$$

The convergence is reached when

$$\max_{1 \le i \le 2^q} |t_j - o_j(\boldsymbol{x}^*)| < 0.1 \tag{5.12}$$

This stopping criterion does not explicitly depend on the cost function value; nevertheless, the condition (5.12) requires that the point $\boldsymbol{x}$ is very close to the global minimum. With this choice we can analyze in depth the characteristics of IGA.

The simulation results are shown in table 4. In the runs with BP we used the acceleration procedure suggested by Vogl et al. [21].

In this case IGA was the fastest method; only for $q = 2$ did BP have a similar performance. For $q = 3$ and $q = 4$ both SA and BP reached the maximum limit of ten million evaluations.

5.5 Symmetry function

The last group of tests concerns the problem of training a two-layer neural network that must find the presence of symmetry in a binary string of length q. In this case only two hidden neurons are needed for doing such operation [20]; the number of weights in the network is therefore $n = 2q + 5$.

The tests performed considered the values $q = 3$, 4, 5, and 6, which corresponds to cost functions (5.9) with $n = 11$, 13, 15, and 17 variables, respectively. The bounds for the domain D and the stopping criterion were (5.11) and (5.12), respectively, as for parity. The simulation results are reported in table 5.

The reliability and the efficiency of IGA are again emphasized with respect to BP and SA.

6. Conclusions

A new global optimization method, called the interval genetic algorithm, has been described. It can be viewed as an efficient parallelization of the

Symmetry function				
n	11 ($q = 3$)	13 ($q = 4$)	15 ($q = 5$)	17 ($q = 6$)
Algorithm	*Evaluations performed*			
IGA	3,481	44,302	54,944	140,458
SA	113,793	881,288	994,640	$> 10M$
BP	9,351	408,960	2,347,939	$> 10M$

Table 5: Comparative simulation results on the symmetry function.

simulated annealing technique, or as a particular type of real-coded genetic algorithm.

Although the tests presented in this paper do not claim to give exhaustive information on the interval genetic algorithm, they provide a first examination of its properties, such as convergence speed and reliability. On the basis of the results we have shown, some considerations follow:

1. The interval genetic algorithm seems faster and more reliable than the well-known simulated annealing, since the search for the global minimum is done using a population of points.
2. Because of its complexity the proposed method is well suited for the optimization of multimodal functions; in simpler cases a local minimization procedure, like the Powell method, is more efficient.
3. The interval genetic algorithm can be made parallel with a speed-up close to the population size.

Finally, in the field of neural networks the presented method is considerably faster than the back-propagation algorithm, widely used up to now.

Further tests on different cost functions (also derived from the training of neural networks) are proceeding in order to analyze in greater detail the real possibilities of the interval genetic algorithm.

Acknowledgments

We thank the referee for his careful reading of the manuscript and for his useful suggestions.

References

[1] D. G. Luenberger, *Introduction to Linear and Nonlinear Programming* (Reading, MA, Addison Wesley, 1984).

[2] E. Hansen, "Global Optimization Using Interval Analysis: The Multi-Dimensional Case," *Numerische Mathematik*, **34** (1980) 247–270.

[3] C. C. Meewella and D. Q. Mayne, "Efficient Domain Partitioning Algorithms for Global Optimization of Rational and Lipschitz Continuous Functions," *Journal of Optimization Theory and Applications*, **61** (1989) 247–270.

[4] R. Galar, "Evolutionary Search with Soft Selection," *Biological Cybernetics*, **60** (1989) 357–364.

[5] L. P. Devroye, "On the Convergence of Statistical Search," *IEEE Transactions on Systems, Man, and Cybernetics*, **6** (1976) 46–56.

[6] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, **220**(4598) (1983) 671–680.

[7] E. Aarts and J. Korst, *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing* (Chicester, Wiley, 1989).

[8] S. Geman and C.-R. Hwang, "Diffusions for Global Optimization," *SIAM Journal of Control and Optimization*, **24** (1986) 1031–1043.

[9] F. Aluffi-Pentini, V. Parisi, and F. Zirilli, "A Global Optimization Algorithm Using Stochastic Differential Equations," *ACM Transactions on Mathematical Software*, **14** (1988) 345–365.

[10] A. Corana, M. Marchesi, C. Martini, and S. Ridella, "Minimizing Multimodal Functions of Continuous Variables with the Simulated Annealing Algorithm," *ACM Transactions on Mathematical Software*, **13** (1987) 262–280.

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

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

[13] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller, "Equations of State Calculations by Fast Computing Machines," *Journal of Chemical Physics*, **21** (1953) 1087–1091.

[14] D. Geman and S. Geman, "Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **6** (1984) 721–741.

[15] D. E. Goldberg, "Real-coded Genetic Algorithms, Virtual Alphabets, and Blocking," *Complex Systems*, **5** (1991) 139–167.

[16] D. E. Goldberg, " A Note on Boltzmann Tournament Selection for Genetic Algorithms and Population-Oriented Simulated Annealing," *Complex Systems*, **4** (1990) 445–460.

[17] F. S. Acton, *Numerical Methods That Work* (New York, Harper and Row, 1970), 464–467.

[18] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Representations by Back-Propagating Errors," *Nature*, **323** (1986) 533–536.

[19] D. H. Ackley, "An Empirical Study of Bit Vector Function Optimization," pages 194–200 in *Genetic Algorithms and Simulated Annealing*, edited by L. Davis (London, Pitman, 1987).

[20] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Internal Representations by Error Propagation," chapter 8 in *Parallel Distribute Processing: Volume 1*, edited by D. E. Rumelhart and J. L. McClelland (Cambridge, MIT Press, 1986).

[21] T. P. Vogl, J. K. Mangis, A. K. Rigler, W. T. Zink, and D. L. Alkon, "Accelerating the Convergence of the Back-Propagation Method," *Biological Cybernetics*, **59** (1988) 257–263.