A Model of the Immune System for Constraint-Handling in Genetic Algorithms

Carlos A. Coello Coello and Nareli Cruz Cortés
CINVESTAV-IPN
Evolutionary Computation Group
Departamento de Ingeniería Eléctrica
Sección de Computación
Av. Instituto Politécnico Nacional No. 2508
Col. San Pedro Zacatenco
México, D. F. 07300
ccoello@cs.cinvestav.mx, nareli@computacion.cs.cinvestav.mx

Abstract

In this paper we propose an algorithm based on a negative selection model of the immune system to handle constraints of all types (linear, nonlinear, equality and inequality) in a genetic algorithm used for global optimization. The approach is implemented both in serial and parallel forms, and it is validated using several test functions taken from the specialized literature. Our results indicate that the proposed approach is highly competitive with respect to penalty-based techniques and with respect to another constraint-handling techniques which is considerably more complex to implement.

Keywords: artificial immune system, genetic algorithms, global optimization, parallel genetic algorithms.

1 Introduction

The use of evolutionary algorithms (i.e., heuristics based on the "survival of the fittest" concept adopted in Darwin's theory of evolution) for global optimization has noticeably increased in recent years [15, 29, 35]. However, the incorporation of constraints of any type (i.e., linear, non-linear, equality, and inequality) into the fitness function of an evolutionary algorithm remains as an open research topic. Despite the popularity of penalty functions in the evolutionary optimization literature (mainly in engineering applications), such techniques are known to have several drawbacks from which the most important is perhaps how to choose the penalty factors required by the penalty function [32, 3].

In this paper, we propose a serial and a parallel version of a constraint-handling technique used with a genetic algorithm (GA). The proposed approach incorporates an emulation of the immune system and uses genotypic-based distances to move from the infeasible to the feasible region of a problem. This technique is simple and efficient, and through a set of experiments we show that it is capable of outperforming the most popular penalty-based approaches used in the evolutionary optimization literature. Additionally, it also produces very competitive results with respect to a more complex constraint-handling technique recently developed (the homomorphous maps of Koziel & Michalewicz [23]) at a fraction of its computational cost.

2 Problem Statement

The problem that is of interest to us is the general non-linear programming problem in which we want to:

$$\text{Find } \vec{x} \text{ which optimizes } f(\vec{x}) \tag{1}$$

subject to:

$$g_i(\vec{x}) \leq 0, \quad i = 1, \ldots, n \tag{2}$$

$$h_j(\vec{x}) = 0, \quad j = 1, \ldots, p \tag{3}$$

where $\vec{x}$ is the vector of solutions $\vec{x} = [x_1, x_2, \ldots, x_r]^T$, n is the number of inequality constraints and p is the number of equality constraints (in both cases, constraints could be linear or non-linear).

For an inequality constaint that satisfies $g_i(\vec{x}) = 0$, then we will say that is active at $\vec{x}$. All equality constraints h_j (regardless of the value of $\vec{x}$ used) are considered active at all points of $\mathcal{F}$.

3 Related Work

Despite the wide variety of constraint-handling techniques that have been developed for evolutionary algorithms (see [3, 28]), penalty functions remain as the most popular constraint-handling technique among practitioners [32]. The idea behind penalty-based methods is to transform a constrained optimization problem into an uncontrained one by adding (or subtracting) a certain value to/from the objective function based on the amount of constraint violation present in a certain solution.

In classical optimization, two kinds of penalty functions are considered: exterior and interior. In the case of exterior methods, we start with an infeasible solution and from there we move towards the feasible region. In the case of interior methods, the penalty term is chosen such that its value will be small at points away from the constraint boundaries and will tend to infinity as the constraint boundaries are approached. Then, if we start from a feasible point, the subsequent points generated will always lie within the feasible region since the constraint boundaries act as barriers during the optimization process [31].

The most common method used with evolutionary algorithms is the exterior penalty approach and therefore, we will concentrate our discussion only on such technique. The main reason why most researchers in the evolutionary computation community tend to choose exterior penalties is because they do not require an initial feasible solution. This sort of requirement (an initial feasible solution) is precisely the main drawback of interior penalties. This is an important drawback, since in many of the applications for which evolutionary algorithms are intended the problem of finding a feasible solution is itself NP-hard [38].

The general formulation of the exterior penalty function is:

$$\phi(\vec{x}) = f(\vec{x}) \pm \left[\sum_{i=1}^{n} r_i \times G_i + \sum_{j=1}^{p} c_j \times L_j \right] \tag{4}$$

where $\phi(\vec{x})$ is the new (expanded) objective function to be optimized, G_i and L_j are functions of the constraints $g_i(\vec{x})$ and $h_j(\vec{x})$, respectively, and r_i and c_j are positive constants normally called "penalty factors".

The most common form of G_i and L_j is:

$$G_i = \max[0, g_i(\vec{x})]^{\beta} \tag{5}$$

$$L_j = |h_j(\vec{x})|^{\gamma} \tag{6}$$

where β and γ are normally 1 or 2.

Ideally, the penalty should be kept as low as possible, just above the limit below which infeasible solutions are optimal (this is called, the *minimum penalty rule* [8, 33, 39]). This is due to the fact that if the penalty is too high or too low, then the problem might become very difficult for an evolutionary algorithm [8, 33, 34]. If the penalty is too high and the optimum lies at the boundary of the feasible region, the evolutionary algorithm will be pushed inside the feasible region very quickly, and will not be able to move back towards the boundary with the infeasible region. A large penalty discourages the exploration of the infeasible region since the very beginning of the search process. If, for example there are several disjoint feasible regions in the search space, the evolutionary algorithm would tend to move to one of them, and would not be able to move to a different feasible region unless the disjoint feasible regions are very close from each other.

On the other hand, if the penalty is too low, a lot of the search time will be spent exploring the infeasible region because the penalty will be negligible with respect to the objective function [38]. These issues are very important in evolutionary optimization, because many of the problems in which evolutionary algorithms are used have their optimum lying on the boundary of the feasible region [37, 39].

The minimum penalty rule is conceptually simple, but it is not necessarily easy to implement. The reason is that the exact location of the boundary between the feasible and infeasible regions is unknown in many of the problems for which evolutionary algorithms are intended (e.g., in many cases the constraints are not given in algebraic form, but are the outcome generated by a simulator [4]).

Penalty functions can deal both with equality and inequality constraints, and the normal approach is to transform an equality to an inequality of the form:

$$|h_j(\vec{x})| - \epsilon \leq 0 \tag{7}$$

where ϵ is the tolerance allowed (a very small value). Since we will be comparing the approach proposed in this paper with respect to several penalty-based approaches, we will proceed to review the techniques adopted in our study in the remainder of this section.

3.1 Static Penalties

Any approach in which the penalty factors do not depend on the current generation number in any way, and therefore, remain constant during the entire evolutionary process, is considered as a static penalty approach.

In this paper, we will implement the static penalty approach proposed by Homaifar, Lai and Qi [19]. In this technique the user defines several levels of violation, and a penalty coefficient is chosen for each in such a way that the penalty coefficient increases as we reach higher levels of violation. This approach starts with a random population of individuals (feasible or infeasible).

An individual is evaluated using [27]:

$$\text{fitness}(\vec{x}) = f(\vec{x}) + \sum_{i=1}^{m} \left(R_{k,i} \times \max\,[0, g_i(\vec{x})]^2 \right) \tag{8}$$

where $R_{k,i}$ are the penalty coefficients used, m is total the number of constraints (Homaifar et al. [19] transformed equality constraints into inequality constraints), $f(\vec{x})$ is the unpenalized objective function, and $k = 1, 2, \ldots, l$, where l is the number of levels of violation defined by the user. The idea of this approach is to balance individual constraints separately by defining a different set of factors for each of them through the application of a set of deterministic rules.

3.2 Dynamic Penalties

Any approach in which the current generation number is involved in the computation of the corresponding penalty factors (normally the penalty function is defined in such a way that it increases over time—i.e., generations) is considered a dynamic penalty approach.

In this paper, we will implement the dynamic penalty approach proposed by Joines and Houck [21]. In this proposal, individuals are evaluated (at generation t) using (we assume minimization):

$$\text{fitness}(\vec{x}) = f(\vec{x}) + (C \times t)^{\alpha} \times SVC(\beta, \vec{x}) \tag{9}$$

where C, α and β are constants defined by the user (the authors used $C = 0.5$, $\alpha = 1$ or 2, and $\beta = 1$ or 2), and $SVC(\beta, \vec{x})$ is defined as [21]:

$$SVC(\beta, \vec{x}) = \sum_{i=1}^{n} D_i^{\beta}(\vec{x}) + \sum_{j=1}^{p} D_j(\vec{x}) \tag{10}$$

and

$$D_i(\vec{x}) = \begin{cases} 0 & g_i(\vec{x}) \leq 0 \\ |g_i(\vec{x})| & \text{otherwise} \end{cases} \quad 1 \leq i \leq n \tag{11}$$

$$D_j(\vec{x}) = \begin{cases} 0 & -\epsilon \leq h_j(\vec{x}) \leq \epsilon \\ |h_j(\vec{x})| & \text{otherwise} \end{cases} \quad 1 \leq j \leq p \tag{12}$$

This dynamic function increases the penalty as we progress through generations.

3.3 Annealing Penalties

Michalewicz and Attia [25] considered a method based on the idea of simulated annealing [22]: the penalty coefficients are changed once in many generations (after the algorithm has been trapped in a local optima). Only active constraints are considered at each iteration, and the penalty is increased over time (i.e., the temperature decreases over time) so that infeasible individuals are heavily penalized in the last generations.

The method of Michalewicz and Attia [25] requires that constraints are divided into four groups: linear equalities, linear inequalities, nonlinear equalities and nonlinear inequalities. Also, a set of active constraints $\mathcal{A}$ has to be created, and all nonlinear equalities together with all violated nonlinear inequalities have to be included there. The population is evolved using [26]:

$$\text{fitness}(\vec{x}) = f(\vec{x}) + \frac{1}{2\tau} \sum_{i \in \mathcal{A}} \phi_i^2(\vec{x}) \tag{13}$$

where τ is the cooling schedule [22],

$$\phi_i(\vec{x}) = \begin{cases} \max[0, g_i(\vec{x})] & \text{if } 1 \leq i \leq n \\ |h_i(\vec{x})| & \text{if } n+1 \leq i \leq m \end{cases} \tag{14}$$

and m is the total number of constraints.

At each iteration, the temperature τ is decreased and the new population is created using the best solution found in the previous iteration as the starting point for the next iteration. The process stops when a pre-defined final 'freezing' temperature τ_f is reached.

3.4 Adaptive Penalties

Bean and Hadj-Alouane [1, 16] developed a method that uses a penalty function which takes a feedback from the search process. Each individual is evaluated by the formula:

$$\text{fitness}(\vec{x}) = f(\vec{x}) + \lambda(t)\left[\sum_{i=1}^{n} g_i^2(\vec{x}) + \sum_{j=1}^{p} |h_j(\vec{x})|\right] \tag{15}$$

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

$$\lambda(t+1) = \begin{cases} (1/\beta_1) \cdot \lambda(t), & \text{if case \#1} \\ \beta_2 \cdot \lambda(t), & \text{if case \#2} \\ \lambda(t), & \text{otherwise,} \end{cases} \tag{16}$$

where cases #1 and #2 denote situations where the best individual in the last k generations was always (case #1) or was never (case #2) feasible, $\beta_1, \beta_2 > 1$, $\beta_1 > \beta_2$, and $\beta_1 \neq \beta_2$ (to avoid cycling). In other words, the penalty component $\lambda(t+1)$ for the generation $t+1$ is decreased if all the best individuals in the last k generations were feasible or is increased if they were all infeasible. If there are some feasible and infeasible individuals tied as best in the population, then the penalty does not change.

3.5 Death penalty

The rejection of infeasible individuals (also called "death penalty") is probably the easiest way to handle constraints and it is also computationally efficient, because when a certain solution violates a constraint, it is assigned a fitness of zero. Therefore, no further calculations are necessary to estimate the degree of infeasibility of such a solution. The normal approach taken is to iterate recursively, generating a new point at each recursive call, until a feasible solution is found [18]. This, however, might be a rather lengthy process in problems in which is very difficult to approach the feasible region.

Some researchers have implemented a variation of the death penalty approach in which all infeasible individuals are given a zero fitness. Such is the approach taken in this paper for the sake of simplicity and in order to make more efficient the implementation. Note, however, that when using this sort of approach the initial population may "stagnate" when no feasible individuals are present and, in consequence, the search space would be randomly sampled in such case.

4 The Immune System

Nature has long served as inspiration for many of the scientific and technological advancements of humankind. Humans, after all, can be seen as complex biological information processing systems capable of performing complex tasks even in the presence of uncertainty and lack of information.

We can consider three main information processing systems with a strong biological inspiration [7]: artificial neural networks (inspired on neural systems), evolutionary algorithms (inspired on the mechanism of "survival of the fittest"), and the artificial immune system (inspired on our immune system). Our current discussion will concentrate on the last of these systems.

From the information processing perspective, the immune system can be seen as a parallel and distributed adaptive system [14, 6]. It is capable of learning, it uses memory and is able of associative retrieval of information in recognition and classification tasks. Particularly, it learns to recognize patterns, it remembers patterns that it has been shown in the past and its global behavior is an emergent property of many local interactions [7]. All these features of the immune system provide, in consequence, great robustness, fault tolerance, dynamism and adaptability [12]. These are the properties of the immune system that mainly attract researchers to try to emulate it in a computer.

There are several models in which most of the current work is based; the main ones are [9]: a negative selection algorithm [11], a positive selection algorithm [36], a clonal selection algorithm [9], continuous immune network models [20, 10, 40] and discrete immune network models [9].

Our algorithm is based on a model in which both antigens and antibodies are represented by binary strings and a matching rule is used to estimate "similarity" between an antigen and an antibody. This model is based on the work of Forrest et al. [11].

5 The Proposed Approach

The proposed algorithm emulates the invaders recognition process by combining antibodies' libraries in order to attain antigen specificity. Furthermore, the purpose is to learn to identify the proper antibodies. The search process of our approach is led by a genetic algorithm. Thus, what we propose is a scheme in which a genetic algorithm is embedded into another genetic algorithm. Note however that the computational complexity of the approach is not really $O(N^2)$, because the internal genetic algorithm does not evaluate the original fitness function of the problem as we will see later on.

In the remainder of this paper, we present a serial and a parallel version of a genetic algorithm in which constraints are handled through emulations of the immune system. We also compare our results with respect to other constraint-handling techniques using several test functions traditionally adopted in the specialized literature. We conclude with a discussion of the results obtained and with some possible paths of future research.

6 A Serial Version of the Proposed Algorithm

Our serial algorithm version to handle constraints using a immune system is described below:

1. Generate randomly an initial population for the genetic algorithm.

2. If the initial population contains a mixture of feasible and infeasible individuals, then we divide the population in two groups. The first group contains the infeasible individuals, which are denominated "antibodies", and the second contains the feasible individuals, which are called "antigens".

3. If none of the individuals in the initial population is feasible, then we use the magnitude of constraint violation of each individual as its fitness. Then, we use to the best individual in the population as the "antigen", where "best" refers to the individual with the lowest amount of constraint violation. The "antibodies" are a randomly selected portion of the population. For the assignment of antibodies, is important to consider only a portion of the population, to avoid premature convergence. Under this scheme (i.e., no feasible individuals in the initial population), we run our simulation of the immune system based on minimum constraint violation and we verify at certain intervals if we have reached the feasible region. Whenever that happens, then we use the feasible individuals found as our antigens, and we proceed to divide the population as in the previous step.

4. The fitness of all the antibodies is set to zero.

5. The fitness of the antibodies is computed according to their similarity with a set of pre-determined antigens in the following way:

   - An antigen is randomly selected from the antigens population.
   - From the population of N antibodies, we randomly select a sample of size σ without replacement.
   - Each antibody in the sample is compared against the antigen selected, and we compute the result of the comparison, to which we will call Z (matching magnitude). Z represents a distance (normally but not necessarily Euclidean) measured at the genotype level (i.e., at the level of the chromosomic encoding).
   - We identify the antibody in the population that has the highest matching magnitude (Z). Ties are solved by randomly choosing a winner.
   - The matching magnitude of the winner antibody is added to its fitness. The fitness of the other antibodies remains unaltered.
   - The antibodies are returned to the population, and the process is repeated typically during three times the number of antibodies.

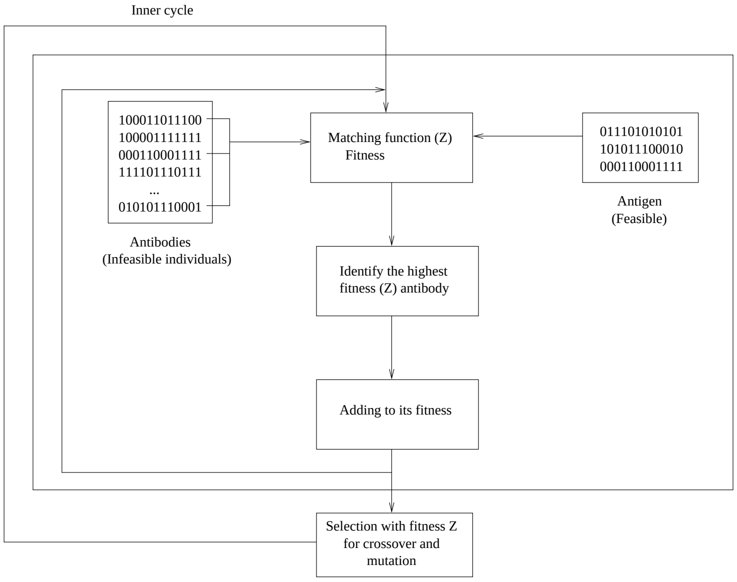

Figure 1: A schematic of the serial version of our algorithm based on the artificial immune system.

6. Based on the fitness computed in the previous step, the population of antibodies is reproduced in a traditional genetic algorithm (using crossover and mutation).

7. The process is repeated from the fourth step until convergence (e.g., when the mean and the maximum fitness in the population are practically the same) or until we reach a maximum number of iterations.

8. Individuals are returned to the external GA and we proceed in the conventional way.

This algorithm is illustrated in Figure 1. The amount of similarity Z of an antibody against an antigen is computed using:

$$Z = \sum_{i=1}^{L} t_i \tag{17}$$

where $t_i = 1$ if there is a matching at position $i = 1, \cdots, L$ (L is the length of the chromosome), or zero if there is no match. This is illustrated in Figure 2. A large Z value means a high matching between the two strings compared and, therefore, a high fitness value.

There are a few issues that need to be mentioned. First, the approach is really using a GA embedded inside another GA used to optimize a certain function. However, the GA that is run with the emulation of the immune system

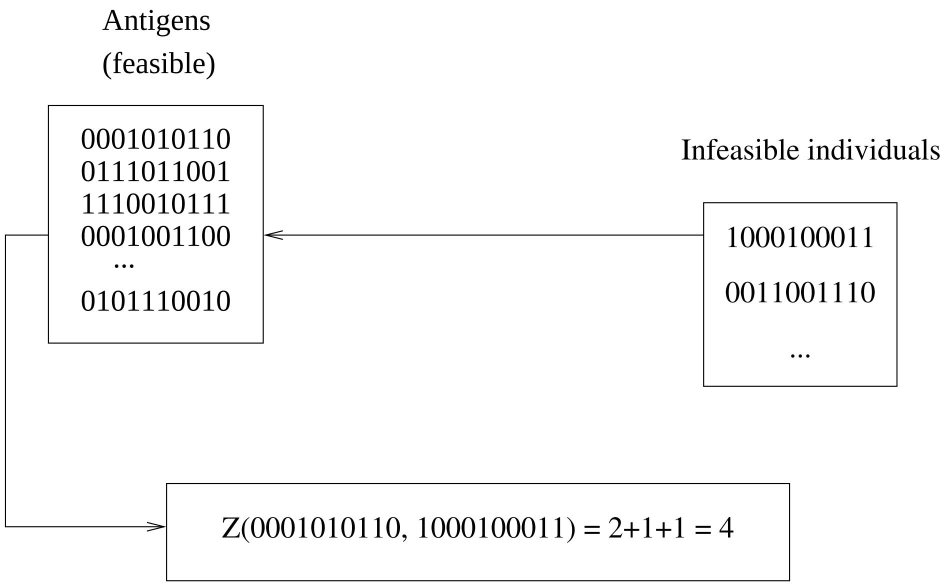

Figure 2: Matching measure Z.

does not use the fitness function directly and, therefore, does not require to evaluate the objective function of the problem. Also, the implicit premise of the technique is that, under certain conditions, the reduction of genotypic differences between two individuals will produce, as a consequence, a phenotypic similarity, which, in our case, will make that an infeasible individual approaches the feasible region.[1]

7 Examples

We used some test functions of the well-known benchmark of Michalewicz [28] to validate our approach. This benchmark has been extensively used to validate new constraint-handling techniques because it includes constraints of different types (linear, nonlinear, equality and inequality) and functions of different degrees of difficulty. We chose some of the test functions of the benchmark that combine the previously indicated features and that are known to be particularly difficult for evolutionary algorithms.

1. **Example 1**:

[1] This can be debatable, but there is some theory that sustains this assumption [13, 39]. Also, the results obtained by Hajela & Lee [17] and us, are empirical evidence of this assumption.

Minimize:

$$f(\vec{x}) = 5\sum_{i=1}^{4} x_i - 5\sum_{i=1}^{4} x_i^2 - \sum_{i=5}^{13} x_i \tag{18}$$

subject to:

$$\begin{aligned}
g_1(\vec{x}) &= 2x_1 + 2x_2 + x_{10} + x_{11} - 10 \leq 0 \\
g_2(\vec{x}) &= 2x_1 + 2x_3 + x_{10} + x_{12} - 10 \leq 0 \\
g_3(\vec{x}) &= 2x_2 + 2x_3 + x_{11} + x_{12} - 10 \leq 0 \\
g_4(\vec{x}) &= -8x_1 + x_{10} \leq 0 \\
g_5(\vec{x}) &= -8x_2 + x_{11} \leq 0 \\
g_6(\vec{x}) &= -8x_3 + x_{12} \leq 0 \\
g_7(\vec{x}) &= -2x_4 - x_5 + x_{10} \leq 0 \\
g_8(\vec{x}) &= -2x_6 - x_7 + x_{11} \leq 0 \\
g_9(\vec{x}) &= -2x_8 - x_9 + x_{12} \leq 0
\end{aligned}$$

where $0 \leq x_i \leq 1$ ($i = 1, \ldots, 9$), $0 \leq x_i \leq 100$ ($i = 10, 11, 12$) and $0 \leq x_{13} \leq 1$. The global minimum is at $\vec{x}^* = (1, 1, 1, 1, 1, 1, 1, 1, 1, 3, 3, 3, 1)$ where six constraints are active (g_1, g_2, g_3, g_7, g_8 and g_9) and $f(\vec{x}^*)$=-15.

2. **Example 2**:

   Minimize:

   $$f(\vec{x}) = 5.3578547x_3^2 + 0.8356891x_1x_5 + 37.293239x_1 - 40792.141$$

   subject to:

   $$\begin{aligned}
   g_1(\vec{x}) &= 85.334407 + 0.0056858x_2x_5 + 0.0006262x_1x_4 - 0.0022053x_3x_5 - 92 \leq 0 \\
   g_2(\vec{x}) &= -85.334407 - 0.0056858x_2x_5 - 0.0006262x_1x_4 + 0.0022053x_3x_5 \leq 0 \\
   g_3(\vec{x}) &= 80.51249 + 0.0071317x_2x_5 + 0.0029955x_1x_2 + 0.0021813x_3^2 - 110 \leq 0 \\
   g_4(\vec{x}) &= -80.51249 - 0.0071317x_2x_5 - 0.0029955x_1x_2 - 0.0021813x_3^2 + 90 \leq 0 \\
   g_5(\vec{x}) &= 9.300961 + 0.0047026x_3x_5 + 0.0012547x_1x_3 + 0.0019085x_3x_4 - 25 \leq 0 \\
   g_6(\vec{x}) &= -9.300961 - 0.0047026x_3x_5 - 0.0012547x_1x_3 - 0.0019085x_3x_4 + 20 \leq 0
   \end{aligned}$$

   where: $78 \leq x_1 \leq 102$, $33 \leq x_2 \leq 45$, $27 \leq x_i \leq 45$ ($i = 3, 4, 5$).

   The global optimum of this problem is located at $\vec{x}^* = (78, 33, 29.995256, 45, 36.775812)$ where $f(\vec{x}^*) = -30665.539$. Two constraints are active at the optimum: g_1 and g_6.

3. **Example 3**:

   Minimize:

   $$f(x) = (x_1 - 10)^3 + (x_2 - 20)^3 \tag{19}$$

subject to:

$$
\begin{aligned}
g_1(x) &= -(x_1-5)^2-(x_2-5)^2+100 \leq 0 \\
g_2(x) &= (x_1-6)^2+(x_2-5)^2-82.81 \leq 0
\end{aligned}
$$

where $13 \leq x_1 \leq 100$ y $0 \leq x_2 \leq 100$. The global optimum is located at $\vec{x}^* = (14.095, 0.84296)$, where $f(\vec{x}^*) = -6961.81388$. Both constraints are active at the optimum.

4. **Example 4**:

   Minimize:

$$
\begin{aligned}
f(\vec{x}) = (x_1-10)^2 + 5(x_2-12)^2 + x_3^4 + 3(x_4-11)^2 \\
+10x_5^6 + 7x_6^2 + x_7^4 - 4x_6x_7 - 10x_6 - 8x_7
\end{aligned}
$$

   subject to:

$$
\begin{aligned}
g_1(\vec{x}) = -127 - 2x_1^2 + 3x_2^4 + x_3 + 4x_4^2 + 5x_5 \leq 0 \\
g_2(\vec{x}) = -282 + 7x_1 + 3x_2 + 10x_3^2 + x_4 - x_5 \leq 0 \\
g_3(\vec{x}) = -196 + 23x_1 + x_2^2 + 6x_6^2 - 8x_7 \leq 0 \\
g_4(\vec{x}) = 4x_1^2 + x_2^2 - 3x_1x_2 + 2x_3^2 + 5x_6 - 11x_7 \leq 0
\end{aligned}
$$

   where $-10 \leq x_i \leq 10$ for $(i = 1, \ldots, 7)$. The optimum solution is $\vec{x}^*$ = (2.330499, 1.951372, -0.4775414, 4.365726, -0.6244870, 1.038131, 1.594227) where $f(\vec{x}^*)$= 680.6300573. Two constraints are active (g_1 and g_4).

8 Comparison of Results

To have a better idea of the degree of difficulty of each of the test functions selected, we computed the metric ρ, suggested by Koziel & Michalewicz [23] to determine how difficult is to reach the feasible region of a certain constrained optimization problem. We determined $\rho = \mid F \mid / \mid S \mid$ experimentally, by generating one million random points in the search space $\mathcal{S}$ and determining whether or not they lied or not within the feasible region $\mathcal{F}$.

Table 1 shows the values of ρ obtained for each of the test functions chosen to validate our approach. In Table 1, n refers to the number of variables, LI are the linear inequalities, and NI are the nonlinear inequalities.

In all the experiments reported with the serial version of the algorithm, our genetic algorithm used binary representation, roulette-wheel selection, two-point crossover (crossover rate of 0.8), and uniform mutation (mutation rate of 0.05).

Test Function	**n**	**Type of f**	ρ	LI	NI
1	13	quadratic	0.0111%	9	0
2	5	quadratic	52.1230%	0	6
3	2	cubic	0.0066%	0	2
4	7	polynomial	0.5121%	0	4

Table 1: Computation of ρ for the test functions selected. The value of ρ is expressed as a percentage (i.e., in a 0–100 scale).

We used a population size of 30 individuals and ran our genetic algorithm for 5000 generations. In all cases, we used a value of σ of one third of the number of antigens in the population. Two other choices of σ were also tried (e.g., we used σ equals to the number of antibodies or the number of antigens depending on the number of feasible individuals in the population). However, our experiments indicated that the performance of the algorithm was not significantly affected by the choice of either of these values of σ. Finally, the number of internal cycles of the algorithm was set to 15 (per iteration).

For the test functions chosen, we compared our results against the homomorphous mappings of Koziel & Michalewicz (KM) [23] which is one of the best constraint-handling technique known to date. The results reported by Koziel & Michalewicz [23] for each of the examples previously described were obtained after performing $70 \times 20000 = 1,400,000$ fitness function evaluations. Our algorithm, in contrast, performed only $30 \times 5000 = 150,000$ fitness function evaluations.

Since penalty functions continue to be a very popular constraint-handling technique in the current evolutionary optimization literature, we decided to compare also our results against five different penalty-based approaches coupled to a genetic algorithm. The techniques selected were the following:

- Static penalty [19] (see Section 3.1).
- Dynamic penalty [21] (see Section 3.2).
- Annealing penalty [25] (see Section 3.3).
- Adaptive penalty [1, 16] (see Section 3.4).
- Death penalty (see Section 3.5).

The five penalty-based approaches previously indicated above are representative of the techniques most commonly used in the standard literature on evolutionary optimization. All the penalty-based approaches used the following parameters: population size = 30, maximum number of generations = 5000, crossover rate = 0.8, and mutation rate = 0.05. Therefore, all the penalty-based approaches performed also $150,000$ fitness function evaluations and used the same parameters that the serial version of our technique.

Results	**This paper**	**death penalty**	**static penalty [19]**	**dynamic penalty [21]**
Best	-14.7841	-3.89599	-4.967788	-5.899826
Mean	-14.5266	-3.124349	-3.393627	-3.569680
Worst	-13.8417	-2.320083	-2.108043	-2.158655
Std. dev.	0.2335	0.439053	0.786642	0.833536

Table 2: Comparison of the serial version of our algorithm against several constraint-handling techniques for the first example (PART I).

Results	**This paper**	**annealing penalty [25]**	**adaptive penalty [1, 16]**	**KM [23]**
Best	-14.7841	-7.343341	-5.165591	-14.7864
Mean	-14.5266	-5.071362	-3.640049	-14.7082
Worst	-13.8417	-3.595369	-2.725182	-14.6154
Std. dev.	0.2335	0.772477	0.606248	N.A.

Table 3: Comparison of the serial version of our algorithm against several constraint-handling techniques for the first example (PART II). N.A. = Not Available.

The comparison of results for the first example are shown in Tables 2 and 3. It can be clearly seen in this case that none of the penalty-based approaches was able to converge in the vicinity of the global optimum. Note that our approach had a slightly poorer performance than the homomorphous maps of Koziel and Michalewicz. However, it behaved much better than any penalty-based technique and its results are not too far from the optimum.

The comparison of results for the second example are shown in Tables 4 and 5. In this case, our serial implementation generates results very similar to those produced by the homomorphous maps of Koziel and Michalewicz. Note also that the best result produced by our algorithm is slightly better than the best result found using the homomorphous maps. The penalty-based approaches do

Results	**This paper**	**death penalty**	**static penalty [19]**	**dynamic penalty [21]**
Best	-30665.51	-30577.244692	-30575.86282	-30543.47755
Mean	-30654.98	-30395.486786	-30403.878656	-30405.47011
Worst	-30517.44	-30273.8398	-30294.503128	-30198.05763
Std. dev.	32.67	70.3136	64.191724	94.0364

Table 4: Comparison of the serial version of our algorithm against several constraint-handling techniques for the second example (PART I).

Results	This paper	annealing penalty [25]	adaptive penalty [1, 16]	KM [23]
Best	-30665.51	-30568.309514	-30560.36108	-30664.5
Mean	-30654.98	-30436.946918	-30397.402836	-30655.3
Worst	-30517.44	-30230.415178	-30255.372464	-30645.9
Std. dev.	32.67	84.105812	73.803218	N.A.

Table 5: Comparison of the serial version of our algorithm against several constraint-handling techniques for the second example (PART II). N.A. = Not Available.

Results	This paper	death penalty	static penalty [19]	dynamic penalty [21]
Best	-6961.7608	-6725.090596	-6752.130219	-6579.364885
Mean	-6961.2733	-5902.52151	-6083.905938	-5930.376403
Worst	-6960.6070	-3323.330807	-4755.210341	-3930.299682
Std. dev.	0.3598	806.966677	447.8283	560.355338

Table 6: Comparison of the serial version of our algorithm against several constraint-handling techniques for the third example (PART I).

better in this case (this is mainly due to the higher value of ρ shown in Table 1). Nevertheless, none of the penalty-based approaches was able to reach the global optimum.

The comparison of results for the third example are shown in Tables 6 and 7. In this case, our serial implementation outperformed all the other approaches, including the homomorphous maps. Our technique was able to converge consistently to the optimum (or very close to it) in this case. Note again the difficulties of all the penalty-based techniques to reach the global optimum of this problem.

The comparison of results for the fourth example are shown in Tables 8 and 9. The serial version of our technique outperformed once more all the penalty-based approaches in this example. Also, its results are very similar to those

Results	This paper	annealing penalty [25]	adaptive penalty [1, 16]	KM [23]
Best	-6961.7608	-6839.393708	-6272.710459	-6952.21
Mean	-6961.2733	-5986.068897	-5558.644717	-6342.6
Worst	-6960.6070	-4647.529779	-3347.866131	-5473.9
Std. dev.	0.3598	550.853075	2337.080599	N.A.

Table 7: Comparison of the serial version of our algorithm against several constraint-handling techniques for the third example (PART II). N.A. = Not Available.

Results	This paper	death penalty	static penalty [19]	dynamic penalty [21]
Best	680.9599	815.023589	781.885832	848.901462
Mean	681.6192	1393.874298	1338.497435	1210.549585
Worst	683.7651	2970.694653	2147.804876	1909.872525
Std. dev.	0.7733	480.734897	313.489052	225.436343

Table 8: Comparison of the serial version of our algorithm against several constraint-handling techniques for the fourth example (PART I).

Results	This paper	annealing penalty [25]	adaptive penalty [1, 16]	KM [23]
Best	680.9599	897.711272	867.491057	680.91
Mean	681.6192	1381.541513	1269.355295	681.16
Worst	683.7651	2148.620678	2008.210163	683.13
Std. dev.	0.7733	308.032119	303.472942	N.A.

Table 9: Comparison of the serial version of our algorithm against several constraint-handling techniques for the fourth example (PART II). N.A. = Not Available.

generated by the homomorphous maps of Koziel and Michalewicz.

Summarizing, we can conclude that all the penalty-based techniques implemented had difficulties to reach the vicinity of the global optimum in all the test functions chosen. None of the penalty-based approaches was able to converge (not even once in the 30 runs performed with each of the techniques) to the global optimum. The serial version of our algorithm produced very competitive results while performing only about 1/10th of the fitness function evaluations required by the homomourphous maps.

9 A Parallel Version of the Proposed Algorithm

We also developed a parallel implementation of our approach. The algorithm is described below:

1. Determine the number of processors available and make it equal to the number of demes created.
2. Determine the size of each deme (d) dividing the population size of the original (i.e., sequential) GA by the total number of demes.
3. For each deme do the following:
   (a) Generate randomly an initial population for the GA.

(b) If the initial population contains a mixture of feasible and infeasible individuals, then the population is divided in two groups. The first contains the infeasible individuals, which are denominated "antibodies". The second contains the feasible individuals, which are called "antigens".

(c) If none of the individuals in the initial population is feasible, then we use the magnitude of constraint violation of each individual as its fitness. Then, we use as the "antigen" to the best individual in the population, where "best" means the one with the lowest constraint violation. The "antibodies" are a randomly selected portion of the population. For the assignment of antibodies, it is important to consider only a portion of the population, to avoid premature convergence. Under this scheme (i.e., no feasible individuals in the initial population), we run our simulation of the immune system based on minimum constraint violation and we verify at certain intervals if we have reached the feasible region. Whenever that happens, we then use the feasible individuals found as our antigens, and we proceed to divide the population as in the previous step.

(d) The fitness of all the antibodies is set to zero.

(e) The fitness of the antibodies is computed according to their similarity with a set of pre-determined antigens in the following way:

- An antigen is randomly selected from the antigens population.
- From the population of N antibodies, we randomly select a sample of size σ without replacement.
- Each antibody in the sample is compared against the antigen selected, and we compute the result of the comparison, to which we will call Z (matching magnitude). Z represents a distance (normally but not necessarily Euclidean) measured at the genotype level (i.e., at the level of the chromosomic encoding).
- We identify the antibody in the population that has the highest matching magnitud (Z). Ties are solved by randomly choosing a winner.
- The matching magnitude of the winner antibody is added to its fitness. The fitness of the other antibodies remains unaltered.
- The antibodies are returned to the population, and the process is repeated typically during three times the number of antibodies.

(f) Based on the fitness computed in the previous step, the population of antibodies is reproduced in a traditional genetic algorithm (using crossover and mutation).

(g) The process is repeated C times, where C refers to the result of dividing the number of predefined iterations for the sequential immune system by the total number of demes.

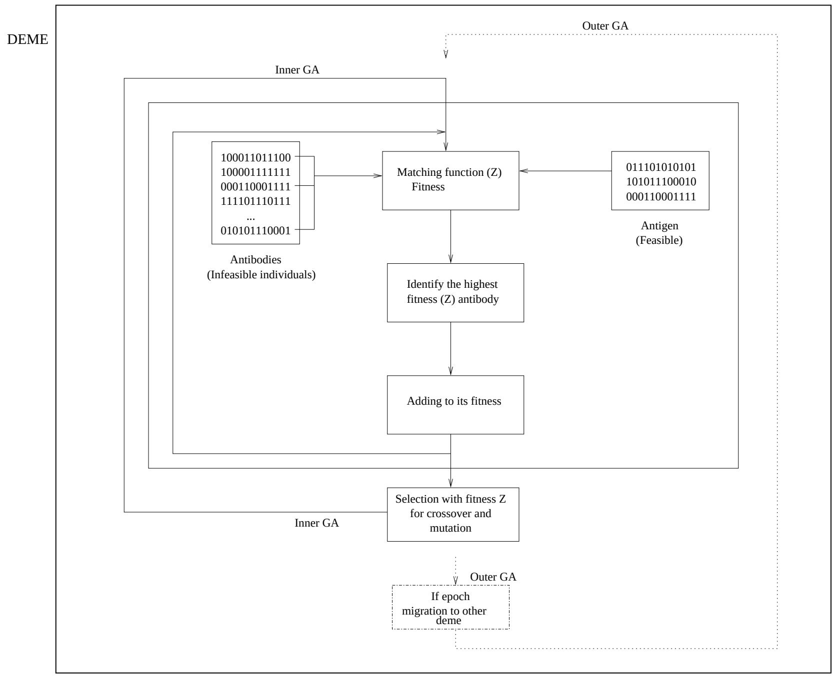

Figure 3: Our algorithm of the immune system.

(h) Individuals are returned to the external GA and we proceed in the conventional way.

4. If the epoch was completed, then:

   (a) Migrate the union of the best individual of the deme plus r randomly selected individuals to a different deme at each iteration.

   (b) Receive $r+1$ migrating individuals that are to replace $r+1$ randomly selected individuals within the deme.

5. Repeat from (b) until reaching convergence.

This algorithm is illustrated in Figure 3.

The parallelization of the algorithm took place applying a coarse-grained GA with a distributed memory using MPI (Message Passing Interface). The population is divided in as many demes as processors are available. Each deme evolves independently from the others exchanging individuals at regular intervals.

Since the demes have a lower number of individuals that the population size of the sequential GA, then the number of iterations required for the internal cycle (see Figure 1) of the immune system is proportionally reduced according to its size. This means that the number of iterations that each deme executes is obtained dividing the population size of the sequential version of the algorithm by the total number of demes.

9.1 Experimental Setup

The efficiency of a parallel algorithm tends to be measured in terms of its correctness and its speedup. The speedup (SP) of an algorithm is obtained by dividing the processing time of the best serial algorithm (T_s) by the processing time of the parallel version (T_p)[2] [24]:

$$SP = \frac{T_s}{T_p} \tag{20}$$

To obtain the best serial algorithm, we performed a set of experiments to determine the minimum parameters required by our algorithm to operate reasonably well [30]. From these experiments, we determined the following parameters for the sequential version of our algorithm[3]: population size = 30, maximum number of generations = 5000, mutation rate = 0.05, crossover rate = 0.8, number of internal GA cycles = 15. The value of σ was defined by dividing the number of infeasible individuals in the current population by two. If there were three or less infeasible individuals in the current population, then all of them were considered as the value of σ. Additionally, we used binary tournament selection and two-point crossover.

The parallel implementation used the following parameters: number of demes = number of processors available (between 2 and 4), deme size = total population size (30) divided by the number of processors available[4], epoch size = 500 generations, migration rate = 0.25. The migration policy adopted was the following: we select the best individual within a deme to be migrated plus an additional set of individuals randomly selected from the same deme. The candidates for replacement within each deme are also randomly selected. The topology adopted is a diamond in which migration is allowed to one different deme at a time (see Figure 4). Finally, the number of internal cycles in the parallel version is obtained by dividing the number used in the sequential algorithm (15) by the number of processors. Again, if this result is not an integer, the `ceiling` function is applied to it to make it an integer.

In order to experiment with different evolutionary characteristics, we decided to use different setups in each deme (these setups were randomly determined) for the following parameters: mutation rate, crossover rate, type of selection (roulette-wheel, stochastic remainder with and without replacement, binary tournament and stochastic universal sampling), type of crossover (one-point, two-point, uniform).

---

[2]This expression assumes identical processors and identical input sizes (i.e., number of fitness function evaluations in our case).

[3]The parameters indicated herein were the same adopted for the experiments performed with the serial version of the algorithm reported in Section 8.

[4]When this result is not an integer, we apply the `ceiling` operator to make it an integer value.

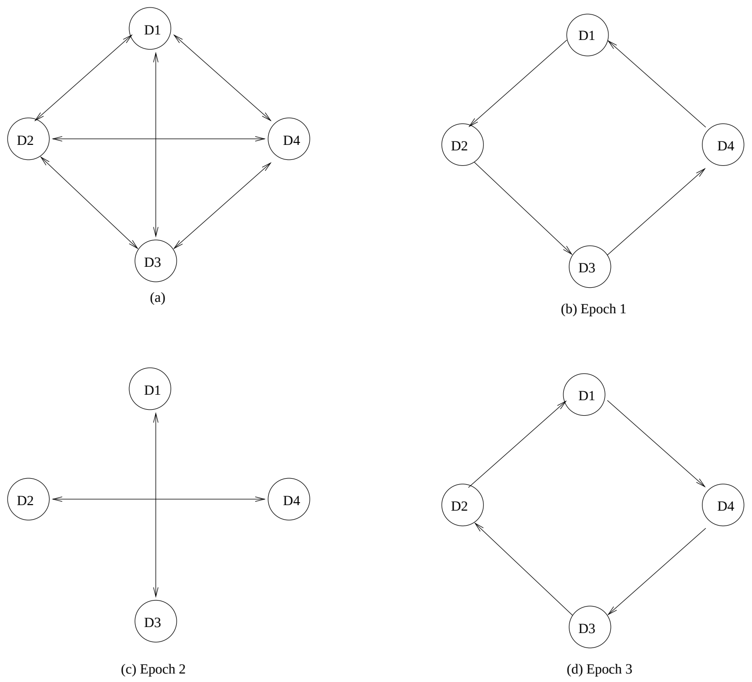

Figure 4: Figure that illustrates the topology and migration policies adopted by the parallel version of our algorithm assuming 4 processors. In (a) we can see the possible interconnections among deme. In (b), (c) and (d) we can see the migration possibilities for each deme. Note how these migration possibilities change at each epoch.

	Serial	2 P SP=5.11	3 P SP=10.71	4 P SP=20.17
Mean	-14.5266	-14.7773	-14.5922	-14.8201
Best	-14.7841	-14.9995	-14.9995	-14.9978
Worst	-13.8417	-12.9892	-11.8787	-12.9930
Std. Dev.	0.2335	0.5103	0.8797	0.4830

Table 10: Comparison of results between the serial version of our algorithm and three setups of the parallel version for the first example. P indicates the number of processors used and SP refers to the speedup achieved.

	Serial	2 P SP=1.92	3 P SP=4.5	4 P SP=8.87
Mean	-30654.98	-30662.32	-30648.86	-30657.03
Best	-30665.51	-30665.00	-30664.19	-30664.62
Worst	-30517.44	-30652.21	-30604.85	-30634.81
Std. Dev.	32.67	2.99	18.05	8.96

Table 11: Comparison of results between the serial version of our algorithm and three setups of the parallel version for the second example. P indicates the number of processors used and SP refers to the speedup achieved.

	Serial	2 P SP=1.37	3 P SP=2.56	4 P SP=3.9
Mean	-6961.2733	-6961.3423	-6960.0504	-6959.6613
Best	-6961.7608	-6961.7596	-6961.4539	-6961.4072
Worst	-6960.6070	-6958.6652	-6958.4271	-6956.7837
Std. Dev.	0.3598	0.6695	0.8767	1.5567

Table 12: Comparison of results between the serial version of our algorithm and three setups of the parallel version for the third example. P indicates the number of processors used and SP refers to the speedup achieved.

Table 10 shows the results produced by the sequential version of our algorithm and three versions of the parallelized algorithm for the first example.[5] Each of these three parallel versions only differ in the number of processors employed. In this case, the parallel versions improve the results produced by the serial version of algorithm (they practically achieve the optimum on average), but with a higher performance variability (i.e., a higher standard deviation). More remarkable is, however, the considerable speedup achieved as more processors are used.

Table 11 shows the results for the second example. The parallel versions of our algorithm maintain similar results than the serial version but with a lower standard deviation. The speedup achieved in this case is once more, remarkable.

Table 12 shows the results for the third example. In this example, the parallel versions obtained slightly inferior results than the serial version of the algorithm and presented a higher standard deviation. However, the differences are not significant. The speedup achieved in this case, althought inferior than in the two previous examples, is still remarkable.

Table 13 shows the results for the fourth example. In this case, the parallel versions obtained similar results than the serial version of our algorithm but with a higher standard deviation. The speedup achieved in this case is, again,

[5] Only the comparison of the serial and the parallel versions of the algorithm are reported here since the results produced by the other techniques have been already reported in Section 8.

	Serial	2 P SP=2.85	3 P SP=5.8	4 P SP=13.48
Mean	681.6192	681.6655	682.5332	682.5125
Best	680.9599	680.7499	680.9414	681.2212
Worst	683.7651	683.2584	687.1950	686.8906
Std. Dev.	0.7733	0.6910	1.8856	1.2949

Table 13: Comparison of results between the serial version of our algorithm and three setups of the parallel version for the fourth example. P indicates the number of processors used and SP refers to the speedup achieved.

remarkable.

A final comment must be made regarding the speedups achieved by our algorithm. Since we used different selection operators and different crossover and mutation rates, we expected that the parallel version of our algorithm would behave differently. Therefore, the computation of the speedup may be somewhat misleading. Although other evolutionary computation researchers have reported superlinear speedups when using evolutionary algorithms (see for example [2]) mainly produced by increased selection pressure of migration, the speedups obtained in our case seem excessive. This behavior may be explained by the extra diversity introduced in the demes as an effect of the diverse selection schemes and crossover and mutation rates introduced. In any case, this issue deserves further study using parallel processing methodologies and is part of our current rersearch.

10 Conclusions and Future Work

We have presented an approach that incorporates constraints into a genetic algorithm using an emulation of the immune system. Two versions of the proposed algorithm were presented in this paper: a serial version and parallel version that uses islands.

Both versions of the proposed approach seem to be competitive with respect to several penalty-based approaches and one technique that represents the state-of-the-art in evolutionary constrained optimization. While none of the penalty-based approaches was able to reach the global optimum (or even its vicinity in some cases), our approach (in all of its variants) produced reasonably good results and even improved in some cases the solutions found by the homomorphous maps of Koziel and Michalewicz [23] at a fraction of its computational cost. Whereas the homomorphous maps performed $1,400,000$ fitness function evaluations per run, our approach only performed $150,000$ fitness function evaluations per run (in both versions of the algorithm). Furthermore, our parallel versions produced remarkable speedups (in one test function, they performed up to 20 times faster than the serial version) while producing similar results. Although more careful studies are required to explain such speedups, the results reported herein seem to indicate that our algorithm highly benefits from a deme-based implementation.

Some of the future work currently under way involves the design and implementation of an algorithm based on the immune system to solve multiobjective optimization problems [5]. We are also exploring other models of the immune system and we are interested in analyzing the convergence properties of the algorithm proposed in this paper.

11 Acknowledgments

The first author acknowledges support from CONACyT through the NSF-CONACyT project number 32999-A. The second author acknowledges support from CONACyT through a scholarship to pursue graduate studies in Computer Science at the Sección de Computación of the Electrical Engineering Department at CINVESTAV-IPN. The work presented in this paper is representative of the research conducted by the Evolutionary Computation Group at CINVESTAV-IPN (EVOCINV).

References

[1] James C. Bean and Atidel Ben Hadj-Alouane. A Dual Genetic Algorithm for Bounded Integer Programs. Technical Report TR 92-53, Department of Industrial and Operations Engineering, The University of Michigan, 1992. To appear in R.A.I.R.O.-R.O. (invited submission to special issue on GAs and OR).

[2] Erick Cantú-Paz. Migration Policies, Selection Pressure, and Parallel Evolutionary Algorithms. *Journal of Heuristics*, 7(4):311–334, 2001.

[3] Carlos A. Coello Coello. Theoretical and Numerical Constraint Handling Techniques used with Evolutionary Algorithms: A Survey of the State of the Art. *Computer Methods in Applied Mechanics and Engineering*, 191(11-12):1245–1287, January 2002.

[4] Carlos A. Coello Coello, Michael Rudnick, and Alan D. Christiansen. Using genetic algorithms for optimal design of trusses. In *Proceedings of the Sixth International Conference on Tools with Artificial Intelligence*, pages 88–94, New Orleans, LA, U.S.A., November 1994. IEEE Computer Society Press.

[5] Carlos A. Coello Coello, David A. Van Veldhuizen, and Gary B. Lamont. *Evolutionary Algorithms for Solving Multi-Objective Problems.* Kluwer Academic Publishers, New York, March 2002. ISBN 0-3064-6762-3.

[6] Dipankar Dasgupta, editor. *Artificial Immune Systems and Their Applications.* Springer-Verlag, Berlin, 1999.

[7] Dipankar Dasgupta and Nii Attoh-Okine. Immunity-Based Systems: A Survey. In *IEEE International Conference on Systems, Man and Cybernetics*, Orlando, Florida, October 12-15 1997.

[8] Lawrence Davis. *Genetic Algorithms and Simulated Annealing*. Pitman, London, 1987.

[9] Leandro N. de Castro and F. J. Von Zuben. The Clonal Selection Algorithm with Engineering Applications. In *Proceedings of the Workshop on Artificial Immune Systems and Their Applications (GECCO'2000)*, pages 36–37, Las Vegas, Nevada, July 2000.

[10] J. D. Farmer, N. H. Packard, and A. S. Perelson. The Immune System, Adaptation, and Machine Learning. *Physica D*, 22:187–204, 1986.

[11] S. Forrest, A.S. Perelson, L. Allen, and R. Cherukuri. Self-nonself discrimination in a computer. In *IEEE Symposium on Research in Security and Privacy*, pages 202–212, Oakland, CA, May 16-18 1994. Springer-Verlag.

[12] Stephanie Forrest and Steven A. Hofmeyr. Immunology as Information Processing. In L.A. Segel and I. Cohen, editors, *Design Principles for the Immune System and Other Distributed Autonomous Systems*, Santa Fe Institute Studies in the Sciences of Complexity, pages 361–387. Oxford University Press, 2000.

[13] Stephanie Forrest and Alan S. Perelson. Genetic algorithms and the immune system. In Hans-Paul Schwefel and R. Männer, editors, *Parallel Problem Solving from Nature*, Lecture Notes in Computer Science, pages 320–325. Springer-Verlag, Berlin, Germany, 1991.

[14] Steven A. Frank. *The Design of Natural and Artificial Adaptative Systems*. Academic Press, New York, 1996.

[15] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Publishing Co., Reading, Massachusetts, 1989.

[16] Atidel Ben Hadj-Alouane and James C. Bean. A Genetic Algorithm for the Multiple-Choice Integer Program. *Operations Research*, 45:92–101, 1997.

[17] P. Hajela and J. Lee. Constrained Genetic Search via Schema Adaptation. An Immune Network Solution. *Structural Optimization*, 12:11–15, 1996.

[18] Frank Hoffmeister and Joachim Sprave. Problem-independent handling of constraints by use of metric penalty functions. In Lawrence J. Fogel, Peter J. Angeline, and Thomas Bäck, editors, *Proceedings of the Fifth Annual Conference on Evolutionary Programming (EP'96)*, pages 289–294, San Diego, California, February 1996. The MIT Press.

[19] A. Homaifar, S. H. Y. Lai, and X. Qi. Constrained Optimization via Genetic Algorithms. *Simulation*, 62(4):242–254, 1994.

[20] N. K. Jerne. Towards a network theory of the immune system. *Ann. Immunol.*, 125:373–389, 1974.

[21] J. Joines and C. Houck. On the use of non-stationary penalty functions to solve nonlinear constrained optimization problems with GAs. In David Fogel, editor, *Proceedings of the first IEEE Conference on Evolutionary Computation*, pages 579–584, Orlando, Florida, 1994. IEEE Press.

[22] S. Kirkpatrick, Jr. C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220:671–680, 1983.

[23] Slawomir Koziel and Zbigniew Michalewicz. Evolutionary Algorithms, Homomorphous Mappings, and Constrained Parameter Optimization. *Evolutionary Computation*, 7(1):19–44, 1999.

[24] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms.* The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994.

[25] Z. Michalewicz and N. Attia. Evolutionary Optimization of Constrained Problems. In *Proceedings of the 3rd Annual Conference on Evolutionary Programming*, pages 98–108. World Scientific, 1994.

[26] Zbigniew Michalewicz. Genetic Algorithms, Numerical Optimization, and Constraints. In Larry J. Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 151–158, San Mateo, California, July 1995. University of Pittsburgh, Morgan Kaufmann Publishers.

[27] Zbigniew Michalewicz, Dipankar Dasgupta, R. Le Riche, and Marc Schoenauer. Evolutionary algorithms for constrained engineering problems. *Computers & Industrial Engineering Journal*, 30(4):851–870, September 1996.

[28] Zbigniew Michalewicz and Marc Schoenauer. Evolutionary Algorithms for Constrained Parameter Optimization Problems. *Evolutionary Computation*, 4(1):1–32, 1996.

[29] Melanie Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, Cambridge, Massachusetts, 1996.

[30] Erick Cantú Paz. *Efficient and Accurate Parallel Genetic Algorithms.* Kluwer Academic Publishers, Massachusetts, USA, 2000.

[31] Singiresu S. Rao. *Engineering Optimization.* John Wiley and Sons, third edition, 1996.

[32] Jon T. Richardson, Mark R. Palmer, Gunar Liepins, and Mike Hilliard. Some guidelines for genetic algorithms with penalty functions. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 191–197, George Mason University, 1989. Morgan Kaufmann Publishers.

[33] Rodolphe G. Le Riche and Raphael T. Haftka. Optimization of Laminate Stacking Sequence for Buckling Load Maximization by Genetic Algorithm. *AIAA Journal*, 31(5):951–970, 1993.

[34] Rodolphe G. Le Riche, Catherine Knopf-Lenoir, and Raphael T. Haftka. A Segregated Genetic Algorithm for Constrained Structural Optimization. In Larry J. Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 558–565, San Mateo, California, July 1995. University of Pittsburgh, Morgan Kaufmann Publishers.

[35] Ruhul Sarker, Masoud Mohammadian, and Xin Yao. *Evolutionary Optimization*. Kluwer Academic Publishers, Boston, Massachusetts, February 2002. ISBN 0-7923-7654-4.

[36] P.E. Seide and F. Celada. A Model for Simulating Cognate Recognition and Response in the Immune System. *Journal of Theoretical Biology*, 158:329–357, 1992.

[37] W. Siedlecki and J. Sklanski. Constrained Genetic Optimization via Dynamic Reward-Penalty Balancing and Its Use in Pattern Recognition. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 141–150, San Mateo, California, jun 1989. George Mason University, Morgan Kaufmann Publishers.

[38] Alice E. Smith and David W. Coit. Constraint Handling Techniques—Penalty Functions. In Thomas Bäck, David B. Fogel, and Zbigniew Michalewicz, editors, *Handbook of Evolutionary Computation*, chapter C 5.2. Oxford University Press and Institute of Physics Publishing, 1997.

[39] Alice E. Smith and David M. Tate. Genetic Optimization Using a Penalty Function. In Stephanie Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 499–503, San Mateo, California, July 1993. University of Illinois at Urbana-Champaign, Morgan Kaufmann Publishers.

[40] F. J. Varela and A. Coutinho. Second Generation Immune Networks. *Immunology Today*, 12(5):159–166, 1991.