Hybridizing a Genetic Algorithm with an Artificial Immune System for Global Optimization

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`

Tel. +52 55 5061 3800 x 6564

Fax +52 55 5061 3757

February 17, 2004

Abstract

In this paper we propose an algorithm based on a 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 other constraint-handling techniques which are considerably more complex to implement.

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

1 Introduction

Despite the success of genetic algorithms (GAs) as optimization techniques [21, 2], they are really an unconstrained search technique. Therefore, an additional mechanism is required to incorporate constraints of any type (linear, nonlinear, equality, inequality) into the fitness function.

---

[*] Corresponding author

In recent years, a number of constraint-handling techniques have been proposed, some of which are very sophisticated [37, 6]. From those techniques, penalty functions remain as the most popular choice for practitioners [40]. However, the performance of a penalty function depends on the type of function adopted and on the values defined for its parameters (namely, the penalty factor).

In this paper, we propose and validate an alternative approach to incorporate constraints into the fitness function of a 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. Both a serial and a parallel version of the algorithm are presented in this paper. The parallel version is compared to its sequential counterpart to evaluate the performance gains obtained and the differences in the results produced (because of the type of paradigm adopted to parallelize our algorithm, it was expected that the results of both versions would differ). Our preliminary results indicate that the algorithm remains as a highly competitive approach (in terms of quality of the results produced) with respect to contemporary constraint-handling techniques. Additionally, the parallelization of the algorithm seems to provide remarkable increases in performance with respect to the sequential version (in some cases, much higher than expected).

2 Problem Statement

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

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

subject to:

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

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

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 nonlinear).

3 The Immune System

The main target of our immune system is to protect the body against harmful organisms (called antigens). This function is performed by lymphocyte cells (B and T cells mainly). The immune system is able to detect a huge number of antigens (about 10^{16} different types) using a fairly limited repertory (the human genome contains only 10^5 genes). To carry out this recognition task, segments of genes are combined to accomplish the specificity of almost all the invader antigens known.

A self-recognition task keeps the immune system from attacking itself, because immune cells are capable of recognizing themselves. Also, upon repeated expositions to a certain antigen, the immune system develops more effective and faster responses over time.

From an information processing perspective, the immune system can be seen as a parallel and distributed adaptive system [19]. It is capable of learning, it has memory and is able of associative information retrieval 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 [11]. All these features of the immune system provide, in consequence, great robustness, fault tolerance, dynamism and adaptability [18]. These are the precisely the properties of the immune system that mainly attract researchers to try to emulate it in a computer.

Farmer et al. [16] were the first to suggest a way of representing the immune system in a computer. In their model, both antigens and detectors are represented as strings of symbols in a small alphabet, and the interactions among the strings represent molecular bonds. Recent examples of this model can be seen in the work of Detours & Perelson [15].

There are some models within the area of the artificial immune system on which most of the current work is based, some of them are: a *negative selection algorithm* [17], a *positive selection algorithm* [44], a *clonal selection algorithm* [13], *continuous immune network models* [28, 16, 49] and *discrete immune network models* [14, 48].

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. [17].

4 Handling Constraints with Evolutionary Algorithms

Despite the wide variety of constraint-handling techniques that have been developed for evolutionary algorithms (see [6, 37]), penalty functions remain as the most popular constraint-handling technique among practitioners [40]. 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 [39].

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 [46].

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* [12, 41, 47]). 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 [12, 41, 42]. 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 light, then the algorithm will not be able to reach the feasible region, and all of the search effort will be spent evaluating infeasible solutions [46]. 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 [45, 47].

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 [7]).

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.

4.1 Static Penalties

In this paper, we will implement the static penalty approach proposed by Homaifar, Lai and Qi [27]. 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 [36]:

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

where $R_{k,i}$ are the penalty coefficients used, m is total the number of constraints (Homaifar et al. [27] 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.

4.2 Dynamic Penalties

Dynamic penalties are those that depend on past search experience, one common way of which is to increment with generation or iteration.

In this paper, we will implement the dynamic penalty approach proposed by Joines and Houck [29]. 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 [29]:

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

and

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

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

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

4.3 Annealing Penalties

Michalewicz and Attia [34] considered a method based on the idea of simulated annealing [31]: the penalty coefficients are changed once in many generations (after the algorithm has been trapped in a local optimum). 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 [34] 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 [35]:

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

where τ is the cooling schedule [31],

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

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.

4.4 Adaptive Penalties

Bean and Hadj-Alouane [3, 23] 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.

Smith and Tate [47] proposed another adaptive penalty approach later refined by Coit and Smith [9] and Coit et al. [10] in which the magnitude of the penalty is dynamically modified according to the fitness of the best solution found so far. An individual is evaluated using the formula (only inequality constraints were considered in this work):

$$\text{fitness}(\vec{x}) = f(\vec{x}) + (B_{feasible} - B_{all}) \sum_{i=1}^{n} \left(\frac{g_i(\vec{x})}{NFT(t)} \right)^k \tag{17}$$

where $B_{feasible}$ is the best known objective function at generation t, B_{all} is the best (unpenalized) overall objective function at generation t, $g_i(\vec{x})$ is the amount by which the constraint i is violated, k is a constant that adjusts the "severity" of the penalty (a value of $k = 2$ has been previously suggested by Coit and Smith [9]), and NFT is the so-called *Near Feasibility Threshold*, which is defined as the threshold distance from the feasible region at which the user would consider that the search is "reasonably" close to the feasible region [37, 20].

4.5 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. 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.

5 The Proposed Approach

The main goal of the research reported in this paper was to explore alternative constraint-handling schemes for genetic algorithms used in global optimization. In the past, we have explored the use of penalty functions in engineering optimization (see for example [7]). However, this previous work indicated a high correlation between performance of the genetic algorithm and the fine-tuning of its penalty factors. Additionally, an important issue for us was not to increase in an important way, the number of fitness function evaluations (as when using, for example, the coevolutionary penalties proposed in [5], which do not require a manual fine-tuning of the penalty factors, but whose computational cost is extremelly high).

The previous requirements led us to the development of the approach proposed in this paper. In the proposed approach, we use the search engine of the genetic algorithm to conduct the search towards the global optimum. However, the genetic algorithm is hybridized with a scheme inspired on an artificial immune model (as in [25]), which acts as a local search mechanism that helps the genetic algorithm to reach the feasible region in a more efficient way. Since this local search mechanism is based only on similarities between chromosomic strings, no additional evaluations of the fitness function are required. Thus, we keep a low computational cost for the approach, which was one of its main design goals.

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 simple emulation of an artificial immune system is embedded into a genetic algorithm. Note however that the computational complexity of the approach is not really $O(N^2)$, because the internal scheme (i.e., the artificial immune system) does not evaluate the original fitness function of the problem as we will see later on (see Figure 1). This internal scheme (which is indeed another genetic algorithm) guides its search based on string similarities and not on objective function values.

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 (see Figure 1):

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

   BEGIN inner GA

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 the best individual in the population as the "antigen", where "best" refers to the individual with the lowest amount of constraint violation.
4. Select randomly a sample of antibodies of size σ.
5. The fitness of the sample of antibodies is computed according to their similarity with a set of antigens in the following way:

- An antigen is randomly selected from the antigens population.
- 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). Z is computed using:

  $$Z = \sum_{i=1}^{L} t_i \tag{18}$$

  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. A large Z value means a high matching between the two strings compared and, therefore, a high fitness value.

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 conven-

tional way.

END inner GA

9. Apply binary tournament selection (with the objective function of the problem) using special rules (as described below).
10. Apply crossover and mutation in a conventional way.
11. The process is repeated from step 2 until reaching stopping condition.

The binary tournament used in step 9 is defined in the following way (two individuals are compared each time):

- If one individual is infeasible and the other one is feasible, then the feasible individual wins.
- If both individuals are feasible, then the one with the highest fitness value is the winner.
- If both individuals are infeasible, then the winner is the one with the lowest constraint violation value.

There are a few issues that need to be mentioned. First, as we indicated before, 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 does not use the fitness function directly; it only computes Hamming distances, which are very inexpensive with respect to evaluating 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. Our algorithm is an extension of the proposal of Hajela & Lee [24, 25].

To clarify the way in which our approach works, we provide next both, the internal and the external GA's adopted:

Internal GA

Step 1: Initialize the fitness of all antibodies to zero.

Step 2: Compute the fitness of the antibody pool based on similarity to the antingens (or based on complementarity); this requires the following specific steps:

(a) An antigen is selected at random.

(b) A sample of antibodies of size μ is selected from the antibody pool without replacement.

(c) The match score of each antibody is computed by comparing against the selected antigen, and the antibody with the highest score has the match score added to its fitness value; the fitness of the other antibodies is unchanged.

(d) The antibodies are then returned to the antibody population, and the process is repeated a number of times (typically two or three times the antibody population size).

Step 3: Based on the fitness computed in Step 2, a GA simulation is conducted with prescribed probabilities of crossover and mutation to evolve the antibody population through one generation of evolution.

Step 4: The process is then repeated from Step 1 until convergence in the antibody population is attained.

External GA:

Step 1: A population of designs is randomly generated.

Step 2: The fitness function, a composite of the objective function and a penalty associated with constraint violation, is obtained for the entire population.

Step 3: Members within the top 3% of the population obtained at the end of Step 2 are designated as antigens, and the entire population (including the antigens) is defined as the starting population of antibodies.

Step 4: Using and antibody sample size μ smaller than the number of antigens, the degree of match Z is obtained for each member of the population according to the steps described before.

Step 5: The match score af each design is used as a fitness measure in a traditional selection or reproduction operation. During this reproduction operation, the size of the population is unchanged.

Step 6: The crossover and mutation operations are performed on the new population of antibodies formed in Step 5.

Step 7: The process is then repeated from Step 2 with an intent of evolving the population to maximize the Z function and cycled to convergence.

Note that the proposed approach is designed to operate only on binary strings. Although we know that the binary alphabet can be used to encode any type of decision variables [26], it may be useful in some cases to use alternative encodings [43]. Should that be the case, the proposed approach is not directly applicable, and its generalization to alphabets of higher cardinality (e.g., real-numbers encoding) remains as an open research area.

At this point, it is important to clarify the main weaknesses that we identified in Hajela's algorithm [25] and that led us to develop the algorithm proposed herein:

- Hajela's approach requires a penalty function in order to sort the population and assign the antigens. This makes necessary to evaluate twice the objective function of the problem (per individual) at each generation of the external genetic algorithm. This may become considerably expensive (computationally speaking) when dealing with real-world applications. In contrast, our approach only evaluates once the objective function (for each individual) per generation, since we do not use a penalty function. We relate the values of the antigens to the constraint violation of each solution. This keeps us from evaluating the fitness function more than once and makes unnecessary to sort the population.

- In Hajela's approach, the computation of the fitness Z in the internal genetic algorithm is performed through a cycle in which the population of antibodies must be traversed several times. In our case, we compute Z in the internal genetic algorithm by performing a single traversal of the antibodies.

- The approach of Hajela & Lee [24] is only validated with a few engineering optimization problems, and no information about its computational cost is provided. In our case, we have used some benchmarks reported in the evolutionary computation literature [37] and we have compared our results against a highly competitive constraint-handling technique which is representative of the state-of-the-art in the area (the homomorphous maps [32]).

Since it may seem that the changes introduced in our algorithm do not make an important difference in terms of performance, we decided to include an example in which we directly compare the results produced by Hajela's algorithm [25] and our own (see Section 8.1).

7 Examples

We used some test functions of the well-known benchmark of Michalewicz [37] 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**:

   Minimize:

$$f(\vec{x}) = 5\sum_{i=1}^{4} x_i - 5\sum_{i=1}^{4} x_i^2 - \sum_{i=5}^{13} x_i \tag{19}$$

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:

$$g_1(\vec{x}) = 85.334407 + 0.0056858x_2x_5 + 0.0006262x_1x_4 - 0.0022053x_3x_5 - 92 \leq 0$$

$$g_2(\vec{x}) = -85.334407 - 0.0056858x_2x_5 - 0.0006262x_1x_4 + 0.0022053x_3x_5 \leq 0$$

$$g_3(\vec{x}) = 80.51249 + 0.0071317x_2x_5 + 0.0029955x_1x_2 + 0.0021813x_3^2 - 110 \leq 0$$

$$g_4(\vec{x}) = -80.51249 - 0.0071317x_2x_5 - 0.0029955x_1x_2 - 0.0021813x_3^2 + 90 \leq 0$$

$$g_5(\vec{x}) = 9.300961 + 0.0047026x_3x_5 + 0.0012547x_1x_3 + 0.0019085x_3x_4 - 25 \leq 0$$

$$g_6(\vec{x}) = -9.300961 - 0.0047026x_3x_5 - 0.0012547x_1x_3 - 0.0019085x_3x_4 + 20 \leq 0$$

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:

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

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:

$$g_1(\vec{x}) = -127 - 2x_1^2 + 3x_2^4 + x_3 + 4x_4^2 + 5x_5 \leq 0$$

$$g_2(\vec{x}) = -282 + 7x_1 + 3x_2 + 10x_3^2 + x_4 - x_5 \leq 0$$

$$g_3(\vec{x}) = -196 + 23x_1 + x_2^2 + 6x_6^2 - 8x_7 \leq 0$$

$$g_4(\vec{x}) = 4x_1^2 + x_2^2 - 3x_1x_2 + 2x_3^2 + 5x_6 - 11x_7 \leq 0$$

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).

5. **Example 5**

Maximize:

$$f(\vec{x}) = (\sqrt{n})^n \prod_{i=1}^{n} x_i$$

subject to:

$$h_1(\vec{x}) = \sum_{i=1}^{n} x_i^2 - 1 = 0$$

where $n = 10$ and $0 \leq x_i \leq 1 (i = 1, ..., n)$. The global maximum is at $x_i^* = 1/\sqrt{n} (i = 1, ..., n)$ where $f(x^*) = 1$.

Additionally, we chose two engineering optimization problems that have been used in the specialized literature.

6. **Example 6 : Design of a Pressure Vessel**

   This problem was proposed by Kannan and Kramer [30]. A cylindrical vessel is capped at both ends by hemispherical heads as shown in Figure 2. The objective is to minimize the total cost, including the cost of the material, forming and welding. There are four design variables: T_s (thickness of the shell), T_h (thickness of the head), R (inner radius) and L (length of the cylindrical section of the vessel, not including the head). T_s and T_h are integer multiples of 0.0625 inch, which are the available thicknesses of rolled steel plates, and R and L are continuous. Using the same notation given by Kannan and Kramer [30], the problem can be stated as follows:

Minimize :

$$F(\vec{x}) = 0.6224x_1x_3x_4 + 1.7781x_2x_3^2 + 3.1661x_1^2x_4 + 19.84x_1^2x_3 \quad (21)$$

Subject to :

$$g_1(\vec{x}) = -x_1 + 0.0193x_3 \leq 0 \quad (22)$$

$$g_2(\vec{x}) = -x_2 + 0.00954x_3 \leq 0 \quad (23)$$

$$g_3(\vec{x}) = -\pi x_3^2 x_4 - \frac{4}{3}\pi x_3^3 + 1,296,000 \leq 0 \quad (24)$$

$$g_4(\vec{x}) = x_4 - 240 \leq 0 \quad (25)$$

7. **Example 7 : Design of a Speed Reducer**

This problem was proposed by Golinski [22]. We want to design the speed reducer shown in Figure 3.

Minimize:

$$f(x) = 0.7854x_1^2x_2^2(3.3333x_3^2 + 14.9334x_3 + 43.0934) - 1.508x_1(x_6^2 + x_7^2)$$
$$+7.4777(x_6^3 + x_7^3) + 0.7854(x_4x_6^2 + x_5x_7^2) \quad (26)$$

subject to:

$$\frac{27}{x_1 x_2^2 x_3} - 1 \leq 0$$

$$\frac{397.5}{x_1 x_2^2 x_3^2} - 1 \leq 0$$

$$\frac{1.93 x_4^3}{x_2 x_3 x_6^4} - 1 \leq 0$$

$$\frac{1.93 x_5^3}{x_2 x_3 x_7^4} - 1 \leq 0$$

$$\frac{[(745 x_4/(x_2 x_3))^2 + 16.9x10^6]^{1/2}}{110.0 x_6^3} - 1 \leq 0$$

$$\frac{[(745 x_5/(x_2 x_3))^2 + 157.5x10^6]^{1/2}}{85.0 x_7^3} - 1 \leq 0$$

$$\frac{x_2 x_3}{40} - 1 \leq 0$$

$$\frac{5 x_2}{x_1} - 1 \leq 0$$

$$\frac{x_1}{12 x_2} - 1 \leq 0$$

$$\frac{1.5 x_6 + 1.9}{x_4} - 1 \leq 0$$

$$\frac{1.1 x_7 + 1.9}{x_5} - 1 \leq 0$$

where $2.6 \leq x_1 \leq 3.6$, $0.7 \leq x_2 \leq 0.8$, $17 \leq x_3 \leq 28$, $7.3 \leq x_4 \leq 8.3$, $7.8 \leq x_5 \leq 8.3$, $2.9 \leq x_6 \leq 3.9$ and $5.0 \leq x_7 \leq 5.5$

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 [32] 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, NI are the nonlinear inequalities and NE are the nonlinear equalities.

In all the experiments reported with the serial version of the algorithm, our genetic algorithm used binary representation, binary tournament selection, two-point crossover (crossover rate of 0.8), and uniform mutation (mutation rate of 0.05). We used a population size of 30 individuals and ran our genetic algorithm for 5000 generations. The number of internal cycles of our algorithm was set to 15. Note that the number of internal cycles refers to the number of iterations to be performed by our artificial immune system that uses Hamming distances as its fitness criterion.

In order to be able to assess the impact of the parameters of our approach on its performance, we performed an analysis of variance (ANOVA). The experimental design is described next.

The parameters (or independent variables) considered for this study are the following:

1. Number of generations of the internal genetic algorithm.
2. Population size.
3. Mutation rate.
4. Size of the antibodies sample (called σ).
5. Type of crossover.

The role of the dependent variable is taken from the result obtained with the algorithm. For our study, the values or levels that each variable could take were, in each case, the following:

- For the number of generations of the internal genetic algorithm, we defined two levels: $\mathbf{15}$ or $\mathbf{30}$.
- For the population size, we defined two levels: $\mathbf{30}$ or $\mathbf{90}$.
- For the mutation rate, we defined the three following levels:
  1. $2/$(length of the chromosomic string)
  2. Dynamic variation, starting in 0.4 and finishing in $1/$(length of the chromosomic string) for both genetic algorithms (internal and external).

3. Dynamic variation, starting in 0.4 and finishing in 1/(length of the chromosomic string) for the external genetic algorithm. The mutation rate for the internal genetic algorithm remains fixed in 1/(length of the chromosomic string).

- For the size of the sample σ, we defined two levels: (a) take ALL the infeasible individuals, and (b) take only HALF of the infeasible individuals.

- For the type of crossover, we defined three levels: (a) one-point crossover, (b) two-point crossover, and (c) uniform crossover.

We performed 30 independent experiments (i.e., runs) for each combination of levels of each variable, using different random seeds in each case. The study was conducted on the serial version of our algorithm.

From the results obtained by ANOVA, we could conclude that the mutation rate has the most significant impact on the performance of the algorithm. The population size and the type of crossover occupy both a second place in importance regarding their impact on performance. Additionally, we found out that the number of generations of the internal genetic algorithm and the size of the antibodies sample (σ) did not have a significant effect on the performance of the algorithm.

In all the examples reported in this paper, 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.

For the test functions chosen, we compared our results against the homomorphous mappings of Koziel & Michalewicz (KM) [32] which is one of the best constraint-handling technique known to date. The results reported by Koziel & Michalewicz [32] 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.[1]

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 [27] (see Section 4.1).
- Dynamic penalty [29] (see Section 4.2).

---

[1] Note that we are only counting objective function evaluations, since we consider the cost of computing Hamming distances (i.e., the fitness function of the immune system embedded in our genetic algorithm) as negligible with respect to the cost of computing an evaluation of the objective function of a problem.

- Annealing penalty [34] (see Section 4.3).
- Adaptive penalty [3, 23] (see Section 4.4).
- Death penalty (see Section 4.5).

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. We performed 30 runs per method per function.

8.1 Example 1

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 to the vicinity of the global optimum. Note that our approach had a slightly poorer performance than the homomorphous maps of Koziel and Michalewicz. However, our approach behaved much better than any penalty-based technique and its results are on the optimum. In Table 4, we compare the results produced by our algorithm with respect to the results obtained using Hajela et al.'s algorithm [25]. It can be clearly seen that the performance of our approach is significantly better than that of Hajela's algorithm, which always converged far away from the global optimum.

8.2 Example 2

The comparison of results for the second example are shown in Tables 5 and 6. 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 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.

8.3 Example 3

The comparison of results for the third example are shown in Tables 7 and 8. 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.

8.4 Example 4

The comparison of results for the fourth example are shown in Tables 9 and 10. 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 generated by the homomorphous maps of Koziel and Michalewicz.

8.5 Example 5

Tables 11 and 12 show the results obtained by our approach with respect to the different penalty-based techniques implemented and for the homomorphous maps technique. For this example, it is difficult for most approaches to reach the feasible region. Since the equality constraint is handled as an inequality, a tolerance is required. In our case, we used $\pm 1 \times 10^{-3}$ for all the approaches compared.

Note that both death and dynamic penalties fail to find any feasible solution. Static, annealing and adaptive techniques managed to find feasible solutions, but such solutions are far away from the optimum value. In contrast, both our algorithm and the homomorphous maps were able to consistently reach the optimum solution (or a very good approximation of it) in all cases.

8.6 Example 6

Tables 13 and 14 show the results obtained by our approach with respect to the different penalty-based techniques implemented (no results on this example are available for the homomorphous maps). It can be clearly seen that our approach produces better numerical results (with respect to all the statistical measures adopted) than any of the penalty-based techniques (i.e., the average behavior of our approach is better than that of the other algorithms compared). The decision variables corresponding to the best solution found is: $\vec{x} = \{0.812500, 0.437500, 42.086994, 176.779128\}$, with $f(\vec{x}) = 6061.1229$. In this example, we also compared results with respect to the Socio-Behavioral Approach (SB) proposed in [1]. SB is a swarm-like based approach proposed to solve engineering optimization problems. The authors simulate societies that conform a civilization. In each society there is a leader which is followed by the other members of its society. Besides these societies, there is a leaders' society that groups the leaders of each society. They are called "general leaders". Constraints are handled by ranking the solutions based on nondominance checkings inside their corresponding society. The authors also adopt a special operator that allows individuals to be assigned values different from those of the leader or the solution selected from the society. This can be seen as an individual that is not following its leader. The main advantage of SB is that it requires a low number of evaluations of the objective function to obtain a reasonably good result which is normally not the optimum. Its main drawback is that the implementation is quite complicated. Besides this, the computational cost increases because of the ranking process and the clustering algorithm that the approach requires to initialize the societies.

In Table 14, we can see that our approach was able to obtain better results than SB, but it is worth noting that SB used less fitness function evaluations than our approach (SB used 12,630 fitness function evaluations, whereas our approach used 150,000 as in the previous examples).

8.7 Example 7

Tables 15 and 16 show the results obtained by our approach with respect to the different penalty-based techniques implemented (no results on this example are available for the homomorphous maps). It can be clearly seen that our approach produces better results (with respect to all the statistical measures adopted) than any of the penalty-based techniques. The decision variables corresponding to the best solution found is: $\vec{x} = \{3.500000, 0.700000, 17.000000, 7.300008,\ 7.715322, 3.350215, 5.286655\}$, with $f(\vec{x}) = 2994.3419$. In this example, we also compared results with respect to the Socio-Behavioral Approach (SB) proposed in [1]. In Table 16, we can see that our approach was able to obtain better results than SB, but it is worth noting that SB used less fitness function evaluations than our approach

(SB used 19,154 fitness function evaluations, whereas our approach used 150,000 as in the previous examples).

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. Regarding the first set of test functions, 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. Regarding the two engineering optimization problems, our approach was able to outperform all the other (penalty) techniques with respect to which it was compared.

9 A Parallel Version of the Proposed Algorithm

We also developed a parallel implementation of our approach. The approach adopted is a multi-population GA (or coarse-grained GA). 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 orig-

inal (i.e., sequential) GA by the total number of demes.

3. For each deme, follow steps 1–11 from the algorithm provided in Section 6.

4. If the epoch was completed, then:

    (a) Copy 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 (3) until reaching convergence.

The parallelization of the algorithm took place applying a multiple-population or multiple-deme genetic algorithm [38]. The approach consists on several subpopulations (called demes) that exchange individuals occasionally. This exchange of individuals is called migration. The parameters that control the migration must be defined by the user, as to determine the number and size of the demes, frequency of migration (epoch), number and destination of migrants, and the way by which the migrants are selected.

On our algorithm we used 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. The total number of fitness function evaluations performed (considering only objective function evaluations) is exactly the same in this case as for the serial version (150,000).

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.

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] [33]:

$$SP = \frac{T_s}{T_p} \tag{27}$$

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 [38]. From these experiments, we determined the following parameters for the sequential version of our algorithm[3]: population size = 30, maximum number of

---

[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

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 (this migration rate indicates the percentage of the population that must be migrated). 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 of cycles used in the sequential algorithm (15) by the number of processors.

In order to experiment with different evolutionary characteristics, we decided to use different setups in each deme (these setups were randomly determined) for

---

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.

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).

Table 17 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). Note that in this example, and all the following, a considerable speedup is achieved as more processors are used.

Table 18 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. Note however, that there is a slight decrease in performance when using three processors.

Table 19 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. In this case, the performance consistently decreases as the number of

---

[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.

processor increases.

Table 20 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. Thus, the parallel version showed a slightly lower robustness than the serial version. As in the previous example, the performance consistently decreases as the number of processor increases.

Note that the fifth example was not solved using the parallel versions of our algorithm. Table 21 shows the results for the sixth example. In this case, the parallel versions of our algorithm were able to improve both the average results and the standard deviation of the results. The best performance was obtained using three processors.

Finally, Table 22 shows the results for the seventh example. In this case, the parallel versions obtained similar results than the serial version of our algorithm but with a higher standard deviation. The differences, however, are not significant.

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. However, the remarkable speedups achieved are related to the reduction of the maximum number of generations of the internal genetic algorithm of each deme (see Section 9).

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 [4]) 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 a 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 is representative of 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 [32] 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 important 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 [8]. We are also interested in generalizing our approach to alphabets of higher cardinality. Finally, 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

We thank the anonymous reviewers for their valuable comments which greatly helped us to improve the contents of this paper.

The first author acknowledges support from CONACyT through project number 34201-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.

References

[1] Akhtar, S., Tai, K. and Ray, T. (2002) A Socio-Behavioural Simulation Model for Engineering Design Optimization. *Engineering Optimization*, **34**(4):341–354.

[2] Bäck, T., Fogel, D.B. and Michalewicz, Z. (Eds.) (1997) *Handbook of Evolutionary Computation.* Institute of Physics Publishing and Oxford University Press.

[3] Bean, J.C. (1994) Genetics and random keys for sequencing and optimization. *ORSA Journal on Computing*, **6**(2):154–160.

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

[5] Coello, C.A.C. (2000) Use of a Self-Adaptive Penalty Approach for Engineering Optimization Problems. *Computers in Industry*, **41**(2):113–127, January.

[6] Coello, C.A.C. (2002) 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.

[7] Coello, C.A.C., Rudnick, M. and Christiansen, A.D. (1994) Using Genetic Algorithms for Optimal Design of Trusses. In *Proceedings of the Sixth International Conference on Tools with Artificial Intelligence*, pages 88–94, IEEE Computer Society Press, New Orleans, Louisiana, USA, November.

[8] Coello, C.A.C., Van Veldhuizen, D.A., and Lamont, G.B. (2002) *Evolutionary Algorithms for Solving Multi-Objective Problems.* Kluwer Academic Publishers, New York, March.

[9] Coit, D.W. and Smith, A.E. (1996) Penalty guided genetic search for reliability design optimization. *Computers and Industrial Engineering*, **30**(4):895–904, September.

[10] Coit, D.W., Smith, A.E. and Tate, D.M. (1996) Adaptive Penalty Methods for Genetic Optimization of Constrained Combinatorial Problems. *INFORMS Journal on Computing*, **8**(2):173–182, Spring.

[11] Dasgupta, D. and Attoh-Okine, N. (1997) Immunity-Based Systems: A Survey. In *IEEE International Conference on Systems, Man and Cybernetics*,

Orlando, Florida, October.

[12] Davis, L. (1987) *Genetic Algorithms and Simulated Annealing.* Pitman, London.

[13] de Castro, L.N. and Von Zuben, F.J. (2000) 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.

[14] de Castro, L.N. and Von Zuben, F.J. (2000) An Evolutionary Immune Network for Data Clustering. In *Proceedings of the IEEE SBRN'00*, pages 84–89.

[15] Detours, V. and Perelson, A.S. (1999) Explaining High Alloreactivity as a Quantitative Consequence of Affinity-Driven Thymocyte Selection. *Proceedings of the National Academy of Science*, **22**:5153–5158.

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

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

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

[19] Frank, S.A. (1996) *The Design of Natural and Artificial Adaptative Systems.* Academic Press, New York.

[20] Gen, M. and Cheng, R. (1996) A Survey of Penalty Techniques in Genetic Algorithms. *Proceedings of the 1996 International Conference on Evolutionary Computation*, T. Fukuda and T. Furuhashi (Eds.). IEEE, pages 804–809, Nagoya, Japan.

[21] Goldberg, D.E. (1989) *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Publishing Co., Reading, Massachusetts.

[22] Golinski, J. (1973) An adaptive optimization system applied to machine synthesis. *Mechanism and Machine Synthesis*, **8**:419–436.

[23] Hadj-Alouane, A.B. and Bean, J.C. (1997) A Genetic Algorithm for the Multiple-Choice Integer Program. *Operations Research*, **45**:92–101.

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

[25] Hajela, P. and Yoo, J.S. (1999) Immune Network Modelling in Design Optimization. *New Ideas in Optimization*, D. Corne, M. Dorigo, and F. Glover (Eds.). McGraw-Hill, pages 167–183.

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

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

[28] Jerne, N.K. (1974) Towards a network theory of the immune system. *Annals of Immunology*, **125**:373–389.

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

[30] Kannan, B.K. and Kramer, S.N. (1994) An Augmented Lagrange Multiplier Based Method for Mixed Integer Discrete Continuous Optimization and Its Applications to Mechanical Design. *Journal of Mechanical Design. Transactions of the ASME*, **116**:318–320.

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

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

[33] Kumar, V., Grama, A., Gupta, A. and Karypis, G. (1994) *Introduction to Parallel Computing: Design and Analysis of Algorithms.* The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA.

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

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

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

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

[38] Cantú-Paz, E. (2000) *Efficient and Accurate Parallel Genetic Algorithms.* Kluwer Academic Publishers, Massachusetts, USA.

[39] Rao, S.S. (1996) *Engineering Optimization.* John Wiley and Sons, Third edition.

[40] Richardson, J.T., Palmer, M.R., Liepins, G. and Hilliard, M. (1989) Some guidelines for genetic algorithms with penalty functions. *Proceedings of the Third International Conference on Genetic Algorithms*, J. David Schaffer (Ed.). Morgan Kaufmann Publishers, San Mateo, California, pages 191–197.

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

[42] Le Riche, R.G., Knopf-Lenoir, C. and Haftka, R.T. (1995) A Segregated Genetic Algorithm for Constrained Structural Optimization. *Proceedings of the Sixth International Conference on Genetic Algorithms*, L.J. Eshelman (Ed.). Morgan Kaufmann Publishers, San Mateo, California, pages 558–565.

[43] Rothlauf, F. (2002) *Representations for Genetic and Evolutionary Algorithms*. Physica-Verlag, New York.

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

[45] Siedlecki, W. and Sklanski, J. (1989) Constrained Genetic Optimization via Dynamic Reward-Penalty Balancing and Its Use in Pattern Recognition. *Proceedings of the Third International Conference on Genetic Algorithms*, J.D. Schaffer (Ed.). Morgan Kaufmann Publishers, San Mateo, California, pages 141–150, June.

[46] Smith, A.E. and Coit, D.W. (1997) Constraint Handling Techniques—Penalty Functions. *Handbook of Evolutionary Computation*, T. Bäck, D.B. Fogel and Z. Michalewicz (Eds.). Oxford University Press and Institute of Physics Publishing, pages C5.2:1–C5.2:6.

[47] Smith, A.E. and Tate, D.M. (1993) Genetic Optimization Using a Penalty Function. *Proceedings of the Fifth International Conference on Genetic Algorithms*, S. Forrest (Ed.). Morgan Kaufmann Publishers, San Mateo, California, pages 499–503, July.

[48] Timmis, J. (2000) *Artificial Immune Systems: A Novel Data Analysis Technique Inspired by the Immune Network Theory*. PhD thesis, Departament of Computer Science, University of Wales, U.K.

[49] Varela, F.J. and Stewart, J. (1990) Dynamics of a class of immune networks I. Global stability of idiotype interactions. *Journal of Theoretical Biology*, **144**(1):93–101.

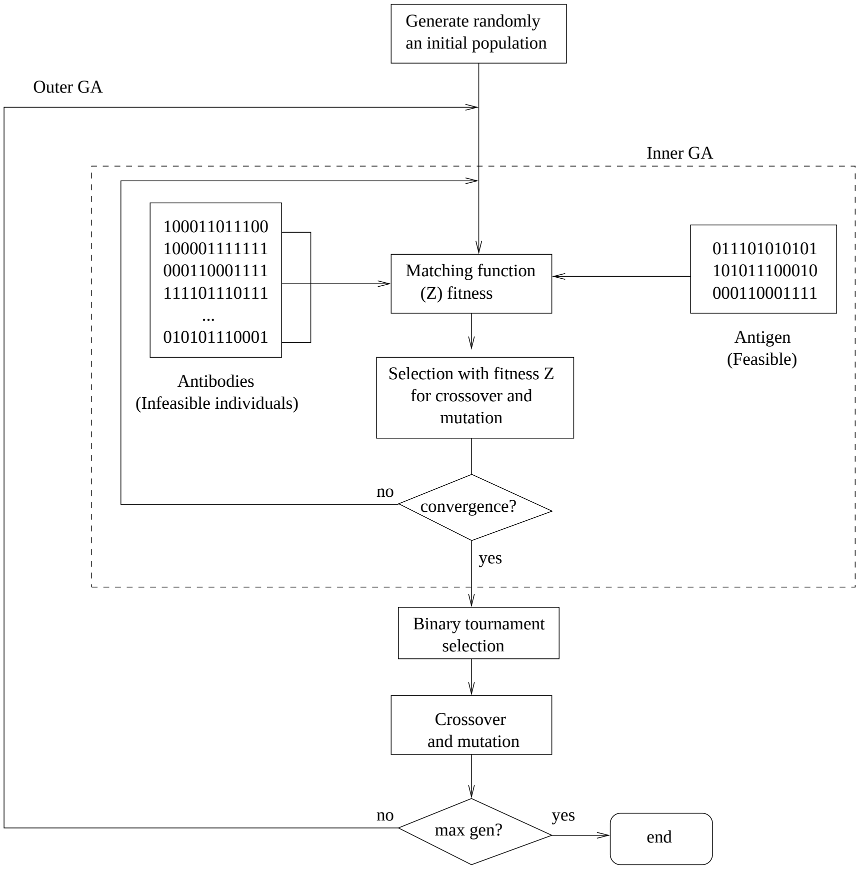


Figure 1: A schematic of the serial version of our algorithm based on the artificial immune system.

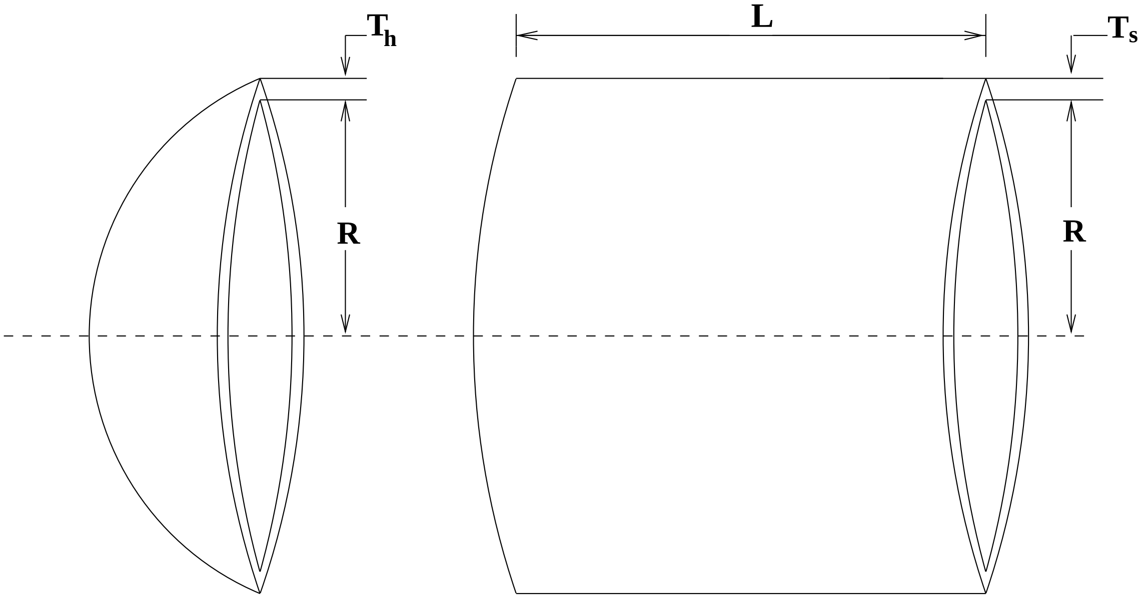


Figure 2: Center and end section of the pressure vessel used for the sixth example.

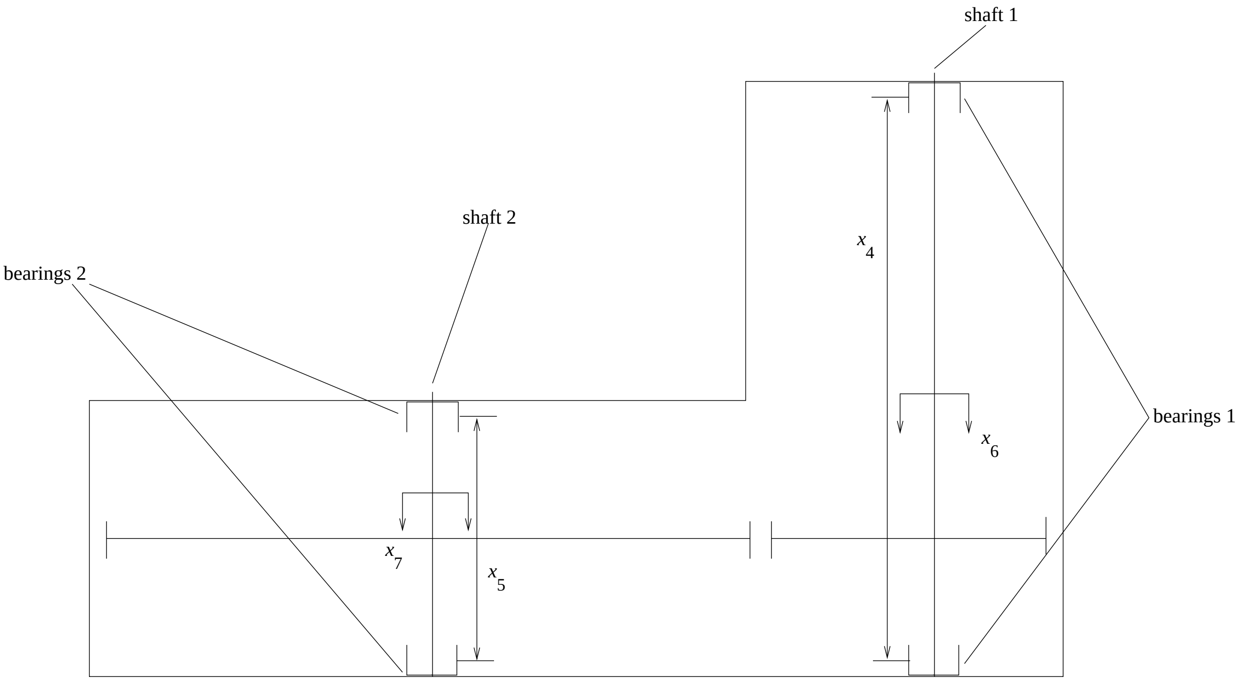
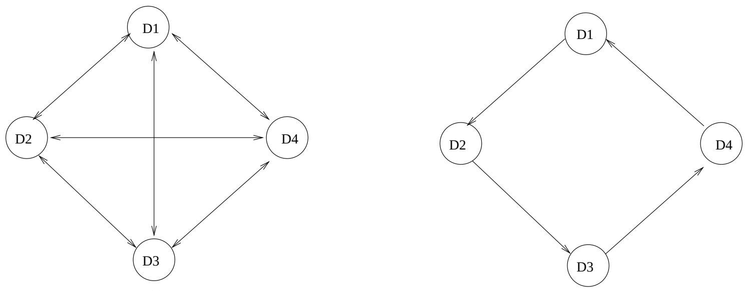
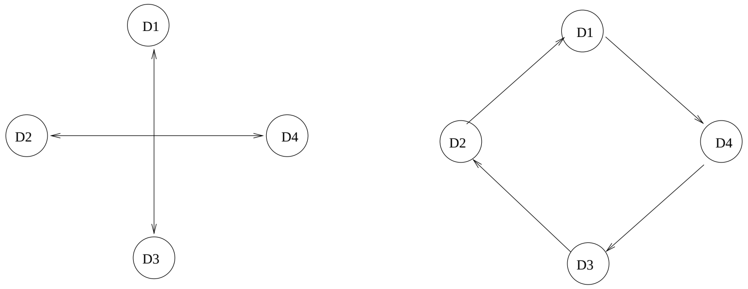


Figure 3: Speed reducer used in the seventh example.



(a)

(b) Epoch 1



(c) Epoch 2

(d) Epoch 3

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 demes. In (b), (c) and (d) we can see the migration possibilities for each deme. Note how these migration possibilities change at each epoch (an epoch in our experiments, comprised 500 generations). The arrows indicate the only available directions of migration for individuals in each deme.

Test Function	**n**	**Type of f**	ρ	LI	NI	NE
1	13	quadratic	0.0111%	9	0	0
2	5	quadratic	52.1230%	0	6	0
3	2	cubic	0.0066%	0	2	0
4	7	polynomial	0.5121%	0	4	0
5	10	polynomial	0.0000%	0	0	1
6	4	nonlinear	0.41%	3	3	0
7	11	nonlinear	0.00097%	0	11	0

Table 1: Computation of ρ for the test functions selected. The value of ρ is expressed as a percentage (i.e., in a 0.0–100.0 scale). LI = Linear Inequalities, NI = Nonlinear Inequalities, NE = Nonlinear Equalities

Results	**This paper**	**death penalty**	**static penalty [27]**	**dynamic penalty [29]**
Best	-15.0	-3.89599	-4.967788	-5.899826
Mean	-15.0	-3.124349	-3.393627	-3.569680
Worst	-15.0	-2.320083	-2.108043	-2.158655
Std. dev.	0.0	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 [34]**	**adaptive penalty [3, 23]**	**KM [32]**
Best	-15.0	-7.343341	-5.165591	-14.7864
Mean	-15.0	-5.071362	-3.640049	-14.7082
Worst	-15.0	-3.595369	-2.725182	-14.6154
Std. dev.	0.0	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.

Results	**This paper**	**Hajela’s proposal**
Best	-15.0	-5.2735
Mean	-15.0	-3.7435
Worst	-15.0	-2.4255
Std. dev.	0.0	0.9696

Table 4: Comparison of the serial version of our algorithm against Hajela’s algorithm [25] in the first example.

Results	**This paper**	**death penalty**	**static penalty [27]**	**dynamic penalty [29]**
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 5: Comparison of the serial version of our algorithm against several constraint-handling techniques for the second example (PART I).

Results	**This paper**	**annealing penalty [34]**	**adaptive penalty [3, 23]**	**KM [32]**
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 6: 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 [27]**	**dynamic penalty [29]**
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 7: Comparison of the serial version of our algorithm against several constraint-handling techniques for the third example (PART I).

Results	**This paper**	**annealing penalty [34]**	**adaptive penalty [3, 23]**	**KM [32]**
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 8: 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 [27]**	**dynamic penalty [29]**
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 9: Comparison of the serial version of our algorithm against several constraint-handling techniques for the fourth example (PART I).

Results	**This paper**	**annealing penalty [34]**	**adaptive penalty [3, 23]**	**KM [32]**
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 10: Comparison of the serial version of our algorithm against several constraint-handling techniques for the fourth example (PART II). N.A. = Not Available.

Results	**This paper**	**death penalty**	**static penalty [27]**	**dynamic penalty [29]**
Best	1.0046	N.F.	0.4508	N.F.
Mean	1.0031	N.F.	0.2359	N.F.
Worst	0.9987	N.F.	0.0105	N.F.
Std. dev.	0.0016	-	0.1678	-

Table 11: Comparison of the serial version of our algorithm against several constraint-handling techniques for the fifth example (PART I). N.F. means that the approach converged to an infeasible solution.

Results	**This paper**	**annealing penalty [34]**	**adaptive penalty [3, 23]**	**KM [32]**
Best	1.0046	0.6611	0.7050	0.9997
Mean	1.0031	0.3176	0.3828	0.9989
Worst	0.9987	0.0798	0.0018	0.9978
Std. dev.	0.0016	0.1538	0.2497	N.A.

Table 12: Comparison of the serial version of our algorithm against several constraint-handling techniques for the fifth example (PART II). N.A. = Not Available.

Results	**This paper**	**death penalty**	**static penalty [27]**	**dynamic penalty [29]**
Best	6061.1229	6480.86	6295.11	6273.28
Mean	6734.0848	8266.67	8098.03	8092.87
Worst	7368.0602	10895.09	9528.07	10382.10
Std. dev.	457.9959	1091.10	831.69	1017.99

Table 13: Comparison of the serial version of our algorithm against several constraint-handling techniques for the sixth example (PART I).

Results	**This paper**	**annealing penalty [34]**	**adaptive penalty [3, 23]**	**SB [1]**
Best	6061.1229	6572.62	6303.50	6171.00
Mean	6734.0848	8164.56	8065.66	6335.05
Worst	7368.0602	9580.51	10569.65	6453.65
Std. dev.	457.9959	789.65	821.30	N.A.

Table 14: Comparison of the serial version of our algorithm against several constraint-handling techniques for the sixth example (PART II). N.A. = Not Available.

Results	**This paper**	**death penalty**	**static penalty [27]**	**dynamic penalty [29]**
Best	2994.3419	3031.9063	3030.6446	3025.7400
Mean	2994.3472	3135.4040	3159.1310	3194.5500
Worst	2994.3768	3359.7894	3502.4819	4544.7992
Std. dev.	0.00768	78.95751	101.4993	266.2394

Table 15: Comparison of the serial version of our algorithm against several constraint-handling techniques for the seventh example (PART I).

Results	**This paper**	**annealing penalty [34]**	**adaptive penalty [3, 23]**	**SB [1]**
Best	2994.3419	3021.4178	3035.2213	3008.0800
Mean	2994.3472	3135.1270	3181.2120	3012.1200
Worst	2994.3768	3848.0370	4401.2518	3028.2800
Std. dev.	0.00768	142.6810	239.2461	N.A.

Table 16: Comparison of the serial version of our algorithm against several constraint-handling techniques for the seventh example (PART II). N.A. = Not Available.

	Serial	2 P SP=5.11	3 P SP=10.71	4 P SP=20.17
Mean	-15.0	-14.7773	-14.5922	-14.8201
Best	-15.0	-14.9995	-14.9995	-14.9978
Worst	-15.0	-12.9892	-11.8787	-12.9930
Std. Dev.	0.0	0.5103	0.8797	0.4830

Table 17: 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 18: 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 19: 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.

	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 20: 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.

	Serial	2 P SP=1.75	3 P SP=3.82	4 P SP=6.27
Mean	6734.0848	6465.3608	6233.5345	6247.7907
Best	6061.1229	6091.5154	6062.652563	6060.6444
Worst	7368.0602	7334.6122	6785.2204	6830.5782
Std. Dev.	457.9959	345.32922	186.0586	203.4138

Table 21: Comparison of results between the serial version of our algorithm and three setups of the parallel version for the sixth example. P indicates the number of processors used and SP refers to the speedup achieved.

	Serial	2 P SP=1.18	3 P SP=3.07	4 P SP=6.75
Mean	2994.3472	2994.3596	2994.4153	2994.3878
Best	2994.3419	2994.3442	2994.3477	2994.3498
Worst	2994.3768	2994.4006	2994.5375	2994.4796
Std. Dev.	0.00768	0.01425	0.05692	0.03113

Table 22: Comparison of results between the serial version of our algorithm and three setups of the parallel version for the seventh example. P indicates the number of processors used and SP refers to the speedup achieved.