Use of a Self-Adaptive Penalty Approach for Engineering Optimization Problems

Carlos A. Coello Coello [1]

Laboratorio Nacional de Informática Avanzada
Rébsamen 80, Xalapa, Veracruz 91090, México
e-mail: ccoello@xalapa.lania.mx

Abstract

This paper introduces the notion of using co-evolution to adapt the penalty factors of a fitness function incorporated in a genetic algorithm for numerical optimization. The proposed approach produces solutions even better than those previously reported in the literature for other (GA-based and mathematical programming) techniques that have been particularly fine-tuned using a normally lengthy trial and error process to solve a certain problem or set of problems. The present technique is also easy to implement and suitable for parallelization, which is a necessary further step to improve its current performance.

Key words: genetic algorithms, constraint handling, co-evolution, penalty functions, self-adaptation, evolutionary optimization, numerical optimization.

1 Introduction

The importance of genetic algorithms (GAs) as a powerful tool for engineering optimization has been widely shown in the last few years through a vast amount of applications ([1,2]). However, even when GAs have been successful in many practical applications, the quality of the solutions that they produce rely not only on the stochastic nature of the technique, but also on the way in which the objective function is converted to a "fitness function" that can "guide" the GA to the desired region of the search space. Being a heuristic method, the GA operates really like a "black box", completely independent

[1] Part of this work was performed while the author was affiliated to the Engineering Design Centre at the University of Plymouth, in the United Kingdom.

from the characteristics of the problem. By using the basic Darwinian mechanism of "survival of the fittest", the GA tries to evolve only those solutions (represented by a population of "chromosomes") that are fitter and, by applying crossover and mutation operators, it attempts to produce descendants that are better (in terms of a certain quantitative measure that we call "fitness") than their parents.

One of the key problems for using GAs in practical applications is how to design the fitness function, particularly when we do not know where is the global optimum located. A comparative estimate of how good is a solution turns out to be enough in most cases (e.g., the largest value has to be closer to the global maximum if we are trying to maximize the objective function), but if we are dealing with constrained problems, we have to find a way of estimating also how close is an infeasible solution from the feasible region. This is not an easy task, since most real-world problems have complex linear and non-linear constraints, and several approaches have been proposed in the past to handle them. From those, the penalty function seems to be yet the most popular technique for engineering problems, but the intrinsic difficulties to define good penalty values makes even harder the optimization process using a GA. In this paper, a technique based on the concept of co-evolution is used to create two populations that interact with each other in such a way that one population evolves the penalty factors to be used by the fitness function of the main population, which is responsible for optimizing the objective function. The approach has been tested with several single-objective optimization problems with linear and non-linear inequality constraints and its results are compared with those produced by other (GA-based and mathematical programming) approaches reported in the literature.

2 Previous Work

The most common approach in the GA community to handle constraints (particularly, inequality constraints) is to use penalties. The basic approach is to define the fitness value of an individual i by extending the domain of the objective function f using [3]

$$fitness_i = f_i(\mathbf{X}) \pm Q_i \tag{1}$$

where Q_i represents either a penalty for an infeasible individual i, or a cost for repairing such an individual (i.e., the cost for making it feasible). It is assumed that if i is feasible then $Q_i = 0$.

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*

[4]). Although simple, this rule is quite difficult to apply in most real-world problems, because we normally do not know the exact location of the boundary between the feasible and the infeasible regions.

It is known that the relationship between an infeasible individual and the feasible part of the search space plays a significant role in penalizing such individual. However, it is not completely clear how to exploit this relationship to guide the search in the most desirable direction.

There are at least three main choices to define a relationship between an infeasible individual and the feasible region of the search space [3]:

(1) an individual might be penalized just for being infeasible (i.e., we do not use any information about how close it is from the feasible region),
(2) the 'amount' of its infeasibility can be measured and used to determine its corresponding penalty, or
(3) the effort of 'repairing' the individual might be taken into account.

Several researchers have studied heuristics on the design of penalty functions. Probably the most well-known of these studies is the one conducted by Richardson et al. [5] from which the following guidelines were derived:

(1) Penalties which are functions of the distance from feasibility are better performers than those which are merely functions of the number of violated constraints.
(2) For a problem having few constraints, and few full solutions, penalties which are solely functions of the number of violated constraints are not likely to find solutions.
(3) Good penalty functions can be constructed from two quantities: the *maximum completion cost* and the *expected completion cost*. By *completion cost* it is meant the cost of making feasible an infeasible solution.
(4) Penalties should be close to the *expected completion cost*, but should not frequently fall below it. The more accurate the penalty, the better will be the solution found. When a penalty often underestimates the completion cost, then the search may not find a solution.

Based mainly on these guidelines, several researchers have attempted to derive good techniques to build penalty functions. Homaifar, Lai and Qi [6] proposed an approach in which 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. The inconvenience of this technique is the high number of parameters required [7]. For m constraints, this approach requires $m(2l + 1)$ parameters in total, where l is the number of levels defined. So, if we have for example 5 constraints and 3 levels, we would need 35 parameters, which is a very high number considering the small size of the problem.

Joines and Houck [8] proposed a technique in which dynamic penalties (i.e., penalties that change over time) are used. Individuals are evaluated (at generation t) using:

$$fitness_i(\mathbf{X}) = f_i(\mathbf{X}) + (C \times t)^\alpha \sum_{j=1}^{m} f_j^\beta(\mathbf{X}) \tag{2}$$

where C, α and β are constants. This dynamic function increases the penalty as we progress through generations. Some researchers [9] have argued that dynamic penalties work better than static penalties. However, it is difficult to derive good dynamic penalty functions in practice as it is to produce good penalty factors for static functions. For example, in this approach the quality of the solution found is very sensitive to changes in the values of the parameters. Even when a certain set of values for these parameters ($C = 0.5$, $\alpha = \beta = 2$) were found by the authors of this method to be a reasonable choice, Michalewicz [7] found that these values produce "premature convergence" most of the time. Also, it was found that the technique normally either converges to an infeasible solution or to a feasible one that is far away from the global optimum [7,3].

Powell and Skolnick [10] incorporated a heuristic rule (suggested by Richardson et al. [5]) for processing infeasible solutions: evaluations of feasible solutions are mapped into the interval $(-\infty, 1)$, and infeasible solutions into the interval $(1, \infty)$. This is equivalent (for ranking and tournament selection procedures [11,12]) to the following evaluation procedure:

$$fitness_f(\mathbf{X}) = f(\mathbf{X}) \tag{3}$$

$$fitness_u(\mathbf{X}) = f(\mathbf{X}) + r \sum_{j=1}^{m} f_j(\mathbf{X}) \tag{4}$$

In this expression, r is a constant, and

$$fitness(\mathbf{X}) = \begin{cases} fitness_f(\mathbf{X}), & \text{if X is feasible} \\ fitness_u(\mathbf{X}) + \rho(\mathbf{X}, t), & \text{otherwise} \end{cases} \tag{5}$$

$$\rho(\mathbf{X}, t) = max\{0, max\{fitness_f(\mathbf{X})\}\} - min\{fitness_u(\mathbf{X})\} \tag{6}$$

The key concept of this approach is the assumption of the superiority of feasible solutions over infeasible ones, and as long as such assumption holds, the technique is expected to behave well [10]. However, in cases where the ratio between the feasible region and the whole search space is too small, the tech-

nique will fail unless a feasible point is introduced in the initial population [13].

Michalewicz and Attia [14] considered a method based on the idea of simulated annealing [15]: the penalty coefficients are changed once in many generations (after the convergence of the algorithm to a local optima). At every iteration the algorithm considers active constraints only, and the pressure on infeasible solutions is increased due to the decreasing values of the temperature of the system.

The method of Michalewicz and Attia [14] 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 [7]:

$$fitness(\mathbf{X}) = f(\mathbf{X}) + \frac{1}{2\tau} \sum_{j \in \mathcal{A}} f_j^2(\mathbf{X}) \tag{7}$$

An interesting aspect of this approach is that the initial population is not really diverse, but consists of multiple copies of a single individual that satisfies all linear constraints. At each iteration, the temperature τ is decreased and the new population is created using the best solution found in the previous iteration. The process stops when a pre-defined final 'freezing' temperature τ_f is reached.

This approach has the inconvenience of being very sensitive to the values of its parameters, and the difficulties for choosing an appropriate cooling scheme is a typical drawback of simulated annealing [15]. Also, the approach used to handle linear constraints (treated separately by this technique) is very efficient, but it requires that the user provides an initial feasible point to the algorithm.

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

$$fitness(\mathbf{X}) = f(\mathbf{X}) + \lambda(t) \sum_{j=1}^{m} f_j^2(\mathbf{X}) \tag{8}$$

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

$$\lambda(t+1) = \begin{cases} (1/\beta_1) \cdot \lambda(t), & if\ case\#1 \\ \beta_2 \cdot \lambda(t), & if\ case\#2 \\ \lambda(t), & otherwise, \end{cases} \tag{9}$$

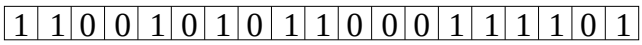
where *cases* #1 and #2 denote situations where the best individual in the last k generation was always (*case* #1) or was never (*case* #2) feasible, $\beta_1, \beta_2 > 1$, and $\beta_1 \neq \beta_2$ (to avoid cycling). In other words, the penalty component $\lambda(t+1)$ for the generation $t+1$ is decreased if all 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 doesn't change.

The obvious drawback of this dynamic penalty approach is how to choose the generational gap (i.e., the appropriate value of k) that provides reasonable information to guide the search, and more important, how do we define the values of β_1 and β_2 to penalyze fairly a given solution.

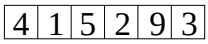
Le Riche et al. [4] designed a segregated genetic algorithm which uses two values of penalty parameters (for each constraint) instead of one; these two values aim at achieving a balance between heavy and moderate penalties by maintaining two subpopulations of individuals. The population is split into two cooperating groups, where individuals in each group are evaluated using either one of the two penalty parameters. The idea is to combine those 2 sub-populations into a single one, mixing then individuals which are feasible with those that are not. Linear ranking is used to decrease the selection pressure that could cause premature convergence.

The problem with this approach is again the way of choosing the penalties for each of the 2 sub-populations, and even when some guidelines have been provided by the authors of this method [13] to define such penalties, they also admit that it is difficult to produce generic values that can be used with this approach.

Finally, some researchers who work with evolution strategies [17] and evolutionary programming [18] have frequently used the "death penalty" approach that consists of rejecting infeasible individuals without even looking at their fitness values. This approach is, with no doubt, quite efficient (computationally speaking), but it is expected to work well only when the feasible search space is convex and it constitutes a reasonable part of the whole search space [3].



Representation of the number 415.293
using binary encoding



Representation of the number 415.293
using fixed point encoding

Fig. 1. Representing the same number using binary and fixed point encodings.

3 Genetic Operators

The genetic algorithm used for the experiments presented in this paper uses a fixed-point representation [19,20], according to which a chromosome is a string of the form $\langle d_1, d_2, \ldots, d_m \rangle$, where $d_1, d_2, \ldots, d_m$ are digits (numbers between zero and nine). Consider the examples shown in Figure 1, in which the same value is represented using binary and fixed point encoding.

Fixed point representation is faster and easier to implement, and provides a higher precision than its binary counterpart, particularly in large domains, where binary strings would be prohibitively long. One of the advantages of fixed point representation is that it has the property that two points close to each other in the representation space must also be close in the problem space, and vice versa [12]. This is not generally true in the binary representation, where the distance in a representation is normally defined by the number of different bit positions. In previous work, the author has shown the usefulness of fixed-point representation where this representation has compared favorably to its binary counterpart [21,19,20].

For crossover, it was decided to use uniform crossover, which is a relatively recent operator proposed by Syswerda [22], that can be seen as a generalization of the more traditional one-point and two-point crossover operators [23,12]. In this case, for each gene (i.e., string position) in the first offspring it is decided (with some probability p) which parent will contribute its value for that position. The second offspring would receive the gene from the other parent. An example of 0.5-uniform crossover can be seen in Figure 2. For the experiments whose results are presented next, a crossover probability (p) of

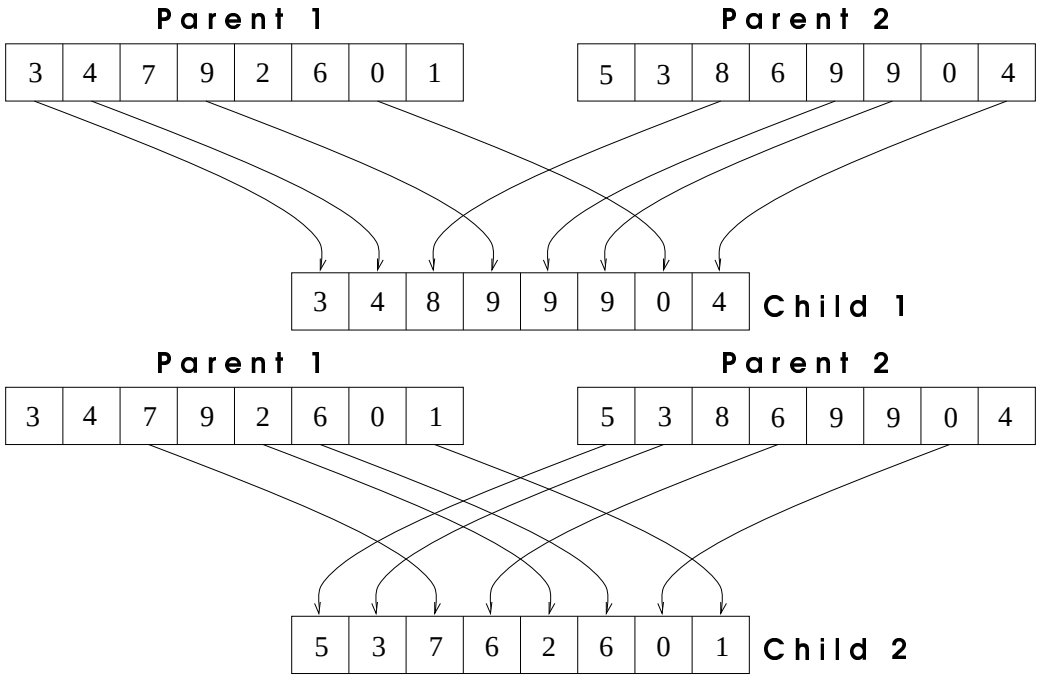


Fig. 2. Use of 0.5-uniform crossover (using 50% probability) between two chromosomes. Notice how half of the genes of each parent goes to each of the two children. First, the bits to be copied from each parent are selected randomly using the probability desired, and after the first child is generated, the same values are used to generate the second child, but inverting the source of procedence of the genes.

0.8 was chosen.

It has been argued that a non-uniform mutation operator is more useful when optimizing with a GA because it allows us to search in different ways as needed (i.e., exploring wider or narrower regions) over time [12]. Due to some previous favorable experience with non-uniform mutation in the context of numerical optimization [24] it was decided to use this approach instead of the traditional uniform mutation operator.

To illustrate this operator, we will assume that at generation t, we have a string $S_t = \langle s_1, s_2, \ldots, s_l \rangle$. After randomly selecting a position along the string, in generation $t+1$, the new chromosome after mutation will be $S_{t+1} = \langle s_1, s_2, \ldots, s'_k, \ldots, s_l \rangle$, where:

$$s'_k = \begin{cases} s_k + \Delta(t, 9 - s_k) & if \ \ flip(0.5) = 0 \\ s_k - \Delta(t, s_k) & if \ \ flip(0.5) = 1 \end{cases} \tag{10}$$

The function $flip(0.5)$ returns randomly and with equal probability one of two possible values: either zero or one. The function $\Delta(t, y)$ returns a value in

the range $[0, y]$ such that the probability of $\Delta(t, y)$ being close to 0 increases as t increases. The expression used here for the variation of the mutation step is the function originally suggested by Michalewicz [12]:

$$\Delta(t, y) = y \cdot \left(1 - r^{\left(1 - \frac{t}{T}\right)^b}\right) \tag{11}$$

where r is a randomly generated real number in the range [0..1], T is the maximum number of generations ($Gmax1$ or $Gmax2$), and b is a system parameter that determines the degree of dependency on the current generation number. The value adopted for the current implementation was $b = 5$, as suggested by Michalewicz [12]. The mutation rate chosen was 0.1, to allow a high exploratory behavior of the GA at earlier generations, and focus more the search into certain regions as the GA reaches its last generations.

4 Use of Self-Adaptive Penalties

Michalewicz et al. [13] have recognized the importance of using adaptive penalties in evolutionary optimization, and considered this approach as a very promising direction of research on evolutionary optimization. The technique proposed in this paper aims to implement this idea using the concept of co-evolution, under which two (or more) populations are evolved either concurrently or interactively, and such populations exchange information in the process. Paredis [25] has used co-evolution for constraint satisfaction (combinatorial optimization) problems, but not for numerical optimization. In his approach, a population of potential solutions co-evolves with a population of constraints: fitter solutions satisfy more constraints, whereas fitter constraints are violated by more solutions.

The approach introduced in this paper uses a conventional penalty function [23,19] rather than trying to handle constraints in an entirely different way (see for example [3]). The reason is that penalty functions are still the most popular approach to handle constraints in practical applications [7,20], whereas the newer approaches have normally being used only to deal with very specific (and generally unrealistic) problems.

The problem that we want to solve is:

$$Optimize \quad f(\mathbf{X}) \tag{12}$$

Subject to :

$$g_i(\mathbf{X}) \leq 0 \quad i = 1, \ldots, p \tag{13}$$

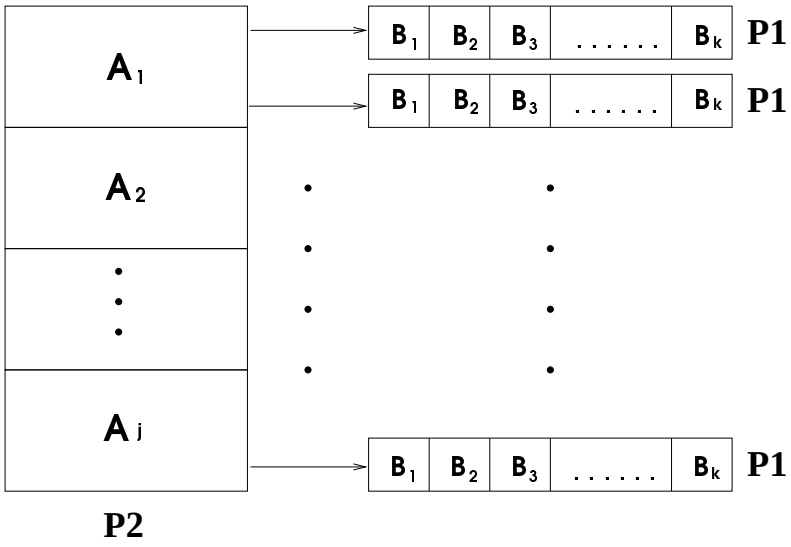


Fig. 3. Graphical representation of the GA-based approach to handle constraints proposed in this paper.

Only inequality constraints are considered in this paper, since penalty functions are not very suitable to handle equality constraints as hard constraints, and there are other approaches which are more suitable to handle them [12].

In previous applications, a penalty function that included information about both the number of constraints violated and the degree of violation of each, has been found very effective by a number of researchers [20,3] to guide the genetic algorithm to (at least near) optimal solutions. The expression used to compute the fitness value of an individual for the purposes of this paper is (assuming maximization):

$$fitness_i = f_i(\mathbf{X}) - (coef \times w_1 + viol \times w_2) \tag{14}$$

where $f_i(\mathbf{X})$ is the value of the objective function for the given set of variable values encoded in the chromosome i; w_1 and w_2 are 2 penalty factors (considered as integers in this paper); $coef$ is the sum of all the amounts by which the constraints are violated:

$$coef = \sum_{i=1}^{p} g_i(\mathbf{X}) \quad \forall g_i(\mathbf{X}) > 0 \tag{15}$$

$viol$ is an integer factor, initialized to zero and incremented by one for each constraint of the problem that is violated, regardless of the amount of violation (i.e., we only count the number of constraints violated but not the magnitude in which each constraint is violated).

According to this approach, the penalty is actually split into two values ($coef$ and $viol$), so that the GA has enough information not only about how many constraints were violated, but also about the amounts in which such constraints were violated. This follows Richardson's suggestion [5] about using penalties that are guided by the distance to feasibility.

We will assume that we have 2 different populations $P1$ and $P2$ with corresponding sizes $M1$ and $M2$. The second of these populations ($P2$) encodes the set of weight combinations (w_1 and w_2) that will be used to compute the fitness value of the individuals in $P1$ (i.e., $P2$ contains the penalty factors that will be used in the fitness function). The idea is to use one population to evolve solutions (as in a conventional genetic algorithm), and another to evolve the penalty factors w_1 and w_2. A graphical representation of this approach may be seen in Figure 3. Notice that for each individual A_j in $P2$ there is an instance of $P1$. However, the population $P1$ is reused for each new element A_j processed from $P2$.

Each individual A_j ($1 \leq j \leq M2$) in $P2$ is decoded and the weight combination produced (i.e., the penalty factors) is used to evolve $P1$ during a certain number ($Gmax1$) of generations. The fitness of each individual B_k ($1 \leq k \leq M1$) is computed using Equation (14), keeping the penalty factors constant for every individual in the instance of $P1$ corresponding to the individual A_j being processed.

After evolving each $P1$ corresponding to every A_j in $P2$ (there is only one instance of $P1$ for each individual in $P2$), we compute the best average fitness produced using:

$$average\_fitness_j = \sum_{i=1}^{M1} \left(\frac{fitness_i}{count\_feasible} \right) + count\_feasible \quad \forall \mathbf{X} \in \mathcal{F} (16)$$

In Equation (16), we add the fitnesses of all feasible solutions in $P1$, and obtain an average of them (the integer variable $count\_feasible$ is a counter that indicates how many feasible solutions were found in the population). The reason for considering only feasible individuals is that if we do not exclude infeasible solutions from this computation, the selection mechanism of the GA may bias the population towards regions of the search space where there are solutions with a very low weight combination (w_1 and w_2). Such solutions may have good fitness values, and still be infeasible. The reason for that is that low values of w_1 and w_2 may produce penalties that are not big enough to outweight the value of the objective function.

Notice also the use of $count\_feasible$ to avoid stagnation (i.e., loss of diversity in the population) at certain regions in which only very few individuals will have a good fitness or will be even feasible. By adding this quantity to the

average fitness of the feasible individuals in the population, we will be encouraging the GA to move towards regions in which lie not only feasible solutions with good fitness values, but there are also a lot of them. In practice, it may be necessary to apply a scaling factor to the average of the fitness before adding $count\_feasible$, to avoid that the GA gets trapped in local optima. However, such scaling factor is not very difficult to compute because we are assuming populations of constant size (such size must be defined before running the GA), and the range of the fitness values can be easily obtained at each generation, because we know the maximum and minimum fitness values in the population at each generation.

The process indicated above is repeated until all individuals in $P2$ have a fitness value (the best $average\_fitness$ of their corresponding $P1$). Then, $P2$ is evolved one generation using conventional genetic operators (i.e., crossover and mutation) and the new $P2$ produced is used to start the same process all over again. It is important to notice that the interaction between $P1$ and $P2$ introduces diversity in both populations, which keeps the GA from easily converging to a local optimum.

5 Examples

Several examples taken from the optimization literature will be used to show the way in which the proposed approach works. These examples have linear and nonlinear constraints, and have been previously solved using a variety of other techniques (both GA-based and traditional mathematical programming methods), which is useful to determine the quality of the solutions produced by the proposed approach.

It should be mentioned that the initial goal of this work was to reproduce the quality of the results found with simple genetic algorithms whose fitness functions and parameters were fine-tuned to solve a specific problem using an empirical approach (normally by simple trial and error). However, as will be seen later, the new technique proposed in this paper not only matched previous results, but it improved them. In each example, a single GA was used to encode all the design variables

5.1 Example 1 : Design of a Pressure Vessel

A cylindrical vessel is capped at both ends by hemispherical heads as shown in Figure 4. 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

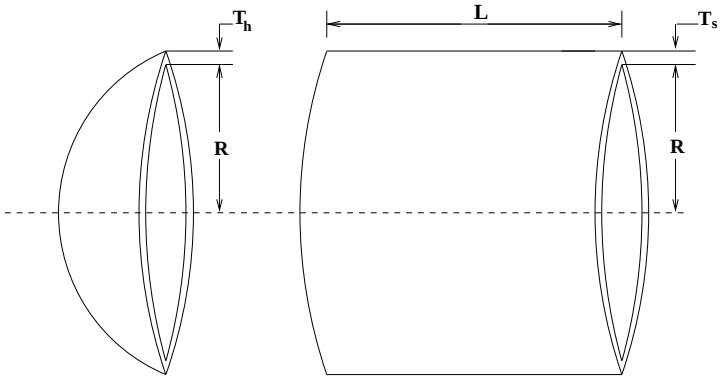


Fig. 4. Center and end section of the pressure vessel used for the first example.

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 [26], the problem can be stated as follows:

Minimize :

$$F(\mathbf{X}) = 0.6224x_1x_3x_4 + 1.7781x_2x_3^2 + 3.1661x_1^2x_4 + 19.84x_1^2x_3 \qquad (17)$$

Subject to :

$$g_1(\mathbf{X}) = -x_1 + 0.0193x_3 \leq 0 \qquad (18)$$

$$g_2(\mathbf{X}) = -x_2 + 0.00954x_3 \leq 0 \qquad (19)$$

$$g_3(\mathbf{X}) = -\pi x_3^2 x_4 - \frac{4}{3}\pi x_3^3 + 1,296,000 \leq 0 \qquad (20)$$

$$g_4(\mathbf{X}) = x_4 - 240 \leq 0 \qquad (21)$$

5.2 Example 2 : Welded Beam Design

A welded beam is designed for minimum cost subject to constraints on shear stress (τ), bending stress in the beam (σ), buckling load on the bar (P_c), end deflection of the beam (δ), and side constraints [27]. There are four design variables as shown in Figure 5 [27]: h (x_1), l (x_2), t (x_3) and b (x_4).

The problem can be stated as follows:

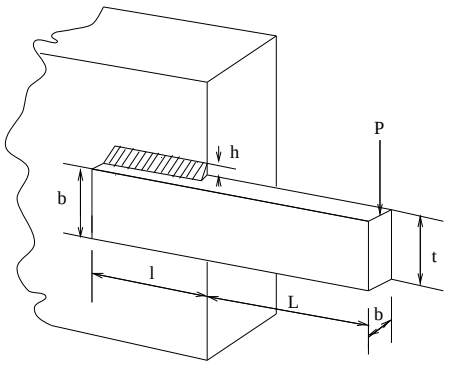


Fig. 5. The welded beam used for the second example.

Minimize:

$$F(\mathbf{X}) = 1.10471x_1^2x_2 + 0.04811x_3x_4(14.0 + x_2) \tag{22}$$

Subject to:

$$g_1(\mathbf{X}) = \tau(\mathbf{X}) - \tau_{max} \leq 0 \tag{23}$$

$$g_2(\mathbf{X}) = \sigma(\mathbf{X}) - \sigma_{max} \leq 0 \tag{24}$$

$$g_3(\mathbf{X}) = x_1 - x_4 \leq 0 \tag{25}$$

$$g_4(\mathbf{X}) = 0.10471x_1^2 + 0.04811x_3x_4(14.0 + x_2) - 5.0 \leq 0 \tag{26}$$

$$g_5(\mathbf{X}) = 0.125 - x_1 \leq 0 \tag{27}$$

$$g_6(\mathbf{X}) = \delta(\mathbf{X}) - \delta_{max} \leq 0 \tag{28}$$

$$g_7(\mathbf{X}) = P - P_c(\mathbf{X}) \leq 0 \tag{29}$$

where

$$\tau(\mathbf{X}) = \sqrt{(\tau')^2 + 2\tau'\tau''\frac{x_2}{2R} + (\tau'')^2} \tag{30}$$

$$\tau' = \frac{P}{\sqrt{2}x_1x_2}, \tau'' = \frac{MR}{J}, M = P\left(L + \frac{x_2}{2}\right) \tag{31}$$

$$R = \sqrt{\frac{x_2^2}{4} + \left(\frac{x_1 + x_3}{2}\right)^2} \tag{32}$$

$$J = 2\left\{\sqrt{2}x_1x_2\left[\frac{x_2^2}{12} + \left(\frac{x_1 + x_3}{2}\right)^2\right]\right\} \tag{33}$$

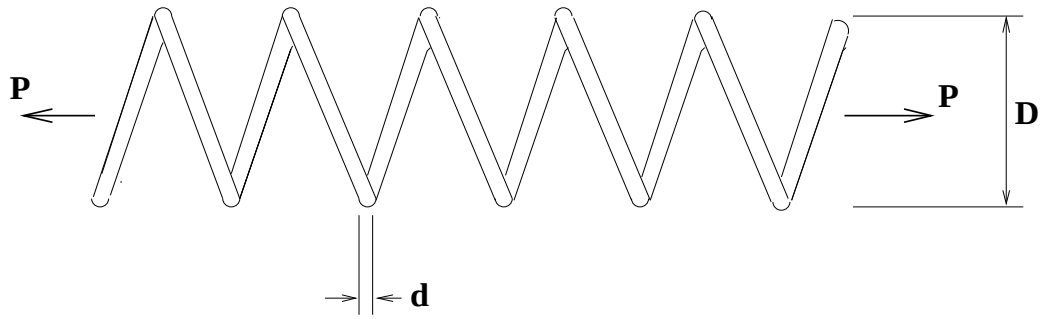


Fig. 6. Tension/compression string used for the third example.

$$\sigma(\mathbf{X}) = \frac{6PL}{x_4 x_3^2}, \delta(\mathbf{X}) = \frac{4PL^3}{Ex_3^3 x_4} \tag{34}$$

$$P_c(\mathbf{X}) = \frac{4.013E\sqrt{\frac{x_3^2 x_4^6}{36}}}{L^2}\left(1 - \frac{x_3}{2L}\sqrt{\frac{E}{4G}}\right) \tag{35}$$

$$P = 6000\ lb, \quad L = 14\ in, \quad E = 30 \times 10^6\ psi, \quad G = 12 \times 10^6\ psi \tag{36}$$

$$\tau_{max} = 13,600\ psi, \quad \sigma_{max} = 30,000\ psi, \quad \delta_{max} = 0.25\ in \tag{37}$$

5.3 Example 3 : Minimization of the Weight of a Tension/Compression String

This problem was described by Arora [28] and Belegundu [29], and it consists of minimizing the weight of a tension/compression spring (see Figure 6) subject to constraints on minimum deflection, shear stress, surge frequency, limits on outside diameter and on design variables. The design variables are the mean coil diameter D, the wire diameter d and the number of active coils N.

Formally, the problem can be expressed as:

$$Minimize \quad (N+2)Dd^2 \tag{38}$$

Subject to

$$g_1(\mathbf{X}) = 1 - \frac{D^3 N}{71785d^4} \leq 0 \tag{39}$$

$$g_2(\mathbf{X}) = \frac{4D^2 - dD}{12566(Dd^3 - d^4)} + \frac{1}{5108d^2} - 1 \leq 0 \tag{40}$$

$$g_3(\mathbf{X}) = 1 - \frac{140.45d}{D^2 N} \leq 0 \tag{41}$$

$$g_4(\mathbf{X}) = \frac{D+d}{1.5} - 1 \leq 0 \tag{42}$$

5.4 Example 4 : Himmelblau's Nonlinear Optimization Problem

This problem was originally proposed by Himmelblau [30], and it was chosen to try the approach proposed here because it has been used before as a benchmark for several other GA-based techniques that use penalties [31]. In this problem, there are five design variables $(x_1, x_2, x_3, x_4, x_5)$, 6 nonlinear inequality constraints and ten boundary conditions. The problem can be stated as follows:

$$Minimize \quad f(\mathbf{X}) = 5.3578547x_3^2 + 0.8356891x_1x_5 + 37.29329x_1 - 40792.141 \tag{43}$$

Subject to:

$$g_1(\mathbf{X}) = 85.334407 + 0.0056858x_2x_5 + 0.00026x_1x_4 - 0.0022053x_3x_5 \tag{44}$$

$$g_2(\mathbf{X}) = 80.51249 + 0.0071317x_2x_5 + 0.0029955x_1x_2 + 0.0021813x_3^2 \tag{45}$$

$$g_3(\mathbf{X}) = 9.300961 + 0.0047026x_3x_5 + 0.0012547x_1x_3 + 0.0019085x_3x_4 \tag{46}$$

$$0 \leq g_1(\mathbf{X}) \leq 92 \tag{47}$$

$$90 \leq g_2(\mathbf{X}) \leq 110 \tag{48}$$

$$20 \leq g_3(\mathbf{X}) \leq 25 \tag{49}$$

$$78 \leq x_1 \leq 102 \tag{50}$$

$$33 \leq x_2 \leq 45 \tag{51}$$

$$27 \leq x_3 \leq 45 \tag{52}$$

$$27 \leq x_4 \leq 45 \tag{53}$$

$$27 \leq x_5 \leq 45 \tag{54}$$

6 Comparison of Results

To make a fair comparison, all the following examples were solved using the same set of parameters shown in Table 1.

Table 1
Parameters of the GA used to solve all the examples.

Parameter	**Value**
Pop_size_1	60
Pop_size_2	30
Gmax_1	25
Gmax_2	20

6.1 Example 1

This problem was solved before by Deb [32] using GeneAS (Genetic Adaptive Search), by Kannan and Kramer using an augmented Lagrangian Multiplier approach [26], and by Sandgren [33] using a branch and bound technique. Their results were compared against those produced by the approach proposed in this paper, and are shown in Table 2. The solution shown for the technique proposed here is the best produced after 11 runs, and using the following ranges for the design variables and the weights: $1 \leq x_1 \leq 99$, $1 \leq x_2 \leq 99$, $10.0000 \leq x_3 \leq 200.0000$, $10.0000 \leq x_4 \leq 200.0000$, $1 \leq w_1 \leq 999$, and $1 \leq w_2 \leq 999$. The values for x_1 and x_2 were considered as integer multiples of 0.0625, the weights w_1 and w_2 were considered as integers, and the values of x_3 and x_4 were considered with a 4-decimals precision.

The mean from the 11 runs performed was $f(\mathbf{X}) = 6293.84323196$, with a standard deviation of 7.41328537. The worst solution found was $f(\mathbf{X}) = 6308.14965192$, which is better than any of the solutions previously reported in the literature. The solution at the median was $f(\mathbf{X}) = 6290.01873568$ (corresponding to $x_1 = 0.8125$, $x_2 = 0.4372$, $x_3 = 40.3302$ and $x_4 = 200.0000$), which is still about 2% better than the best solution previously reported.

6.2 Example 2

This problem was solved before by Deb [34] using a simple genetic algorithm with binary representation, and a traditional penalty function as suggested by Goldberg [23], and by Ragsdell and Phillips [35] using geometric programming. Ragsdell and Phillips also compared their results with those produced by the methods contained in a software package called "Opti-Sep" [36], which includes the following numerical optimization techniques: ADRANS (Gall's adaptive random search with a penalty function), APPROX (Griffith and Stewart's successive linear approximation), DAVID (Davidon-Fletcher-Powell with a penalty function), MEMGRD (Miele's memory gradient with a penalty function), SEEK1 & SEEK2 (Hooke and Jeeves with 2 different penalty func-

Table 2
Comparison of the results for the first example (optimization of a pressure vessel).

Design	**Best solution found**			
Variables	**This paper**	**GeneAS [32]**	**Kannan [26]**	**Sandgren [33]**
$x_1(T_s)$	0.8125	0.9375	1.125	1.125
$x_2(T_h)$	0.4375	0.5000	0.625	0.625
$x_3(R)$	40.3239	48.3290	58.291	47.700
$x_4(L)$	200.0000	112.6790	43.690	117.701
$g_1(\mathbf{X})$	-0.034324	-0.004750	0.000016	-0.204390
$g_2(\mathbf{X})$	-0.052847	-0.038941	-0.068904	-0.169942
$g_3(\mathbf{X})$	-27.105845	-3652.876838	-21.220104	54.226012
$g_4(\mathbf{X})$	-40.00000	-127.321000	-196.310000	-122.299000
$f(\mathbf{X})$	**6288.7445**	**6410.3811**	**7198.0428**	**8129.1036**

tions), SIMPLX (Simplex method with a penalty function) and RANDOM (Richardson's random method).

Their results against those produced by the approach proposed in this paper, and are shown in Table 3. In the case of Siddall's techniques [36], only the best solution produced by the techniques contained in "Opti-Sep" is displayed. The solution shown for the technique proposed here is the best produced after 11 runs, and using the following ranges for the design variables and the weights: $0.1000 \leq x_1 \leq 2.0000$, $0.1000 \leq x_2 \leq 10.0000$, $0.1000 \leq x_3 \leq 10.0000$, $0.1000 \leq x_4 \leq 2.0000$, $1 \leq w_1 \leq 999$, and $1 \leq w_2 \leq 999$. The values for x_1 to x_4 were considered with a 4-decimals precision, and the weights w_1 and w_2 were considered as integers.

The mean from the 11 runs performed was $f(\mathbf{X}) = 1.77197269$, with a standard deviation of 0.01122281. The worst solution found was $f(\mathbf{X}) = 1.7858346524$, which is better than any of the solutions produced by any of the other techniques depicted in Table 3. The solution at the median was $f(\mathbf{X}) = 1.77358615$ (corresponding to $x_1 = 0.1996$, $x_2 = 3.6428$, $x_3 = 9.0507$ and $x_4 = 0.2100$), which is about 27% better than the best solution previously reported.

6.3 Example 3

This problem was solved before by Belegundu [29] using eight numerical optimization techniques (CONMIN, OPTDYN, LINMR, GRP-UI, SUMT, M-3, M4, and M-5). Only the best feasible result reported by him is shown in

Table 3
Comparison of the results for the second example (optimal design of a welded beam).

Design	**Best solution found**			
Variables	**This paper**	**Deb [34]**	**Siddall [36]**	**Ragsdell [35]**
$x_1(h)$	0.2088	0.2489	0.2444	0.2455
$x_2(l)$	3.4205	6.1730	6.2189	6.1960
$x_3(t)$	8.9975	8.1789	8.2915	8.2730
$x_4(b)$	0.2100	-0.2533	0.2444	0.2455
$g_1(\mathbf{X})$	-0.337812	-5758.603777	-5743.502027	-5743.826517
$g_2(\mathbf{X})$	-353.902604	-255.576901	-4.015209	-4.715097
$g_3(\mathbf{X})$	-0.00120	-0.004400	0.000000	0.000000
$g_4(\mathbf{X})$	-3.411865	-2.982866	-3.022561	-3.020289
$g_5(\mathbf{X})$	-0.08380	-0.123900	-0.119400	-0.120500
$g_6(\mathbf{X})$	-0.235649	-0.234160	-0.234243	-0.234208
$g_7(\mathbf{X})$	-363.232384	-4465.270928	-3490.469418	-3604.275002
$f(\mathbf{X})$	**1.74830941**	**2.43311600**	**2.38154338**	**2.38593732**

Table 4. Additionally, Arora [28] also solved this problem using a numerical optimization technique called Constraint Correction at constant Cost (CCC). It is important to notice that Arora's solution is actually infeasible because it violates one of the constraints slightly. In the experiments reported here, the GA handled all constraints are hard, so that the solutions produced were considered valid only if all of them were fully satisfied. Nevertheless, the proposed approach was able to find a better (feasible) solution than Arora's technique, as can be seen in Table 4.

The solution shown for the technique proposed here is the best produced after 11 runs, and using the following ranges for the design variables and the weights: $0.050000 \leq x_1 \leq 2.000000$, $0.250000 \leq x_2 \leq 1.300000$, $2.000000 \leq x_3 \leq 15.000000$, $1 \leq w_1 \leq 999$, and $1 \leq w_2 \leq 999$. The values for x_1 to x_4 were considered with a 6-decimals precision, and the weights w_1 and w_2 were considered as integers.

The mean from the 11 runs performed was $f(\mathbf{X}) = 0.01276920$, with a standard deviation of 3.939×10^{-5}. The worst solution found was $f(\mathbf{X}) = 0.0128220825$, which is better than Belegundu's result. The solution at the median was $f(\mathbf{X}) = 0.0127557615$ (corresponding to $x_1 = 0.051461$, $x_2 = 0.351022$, and $x_3 = 11.721943$), which is better than the best feasible solution previously reported.

Table 4
Comparison of the results for the third example (minimization of the weight of a tension/compression spring).

Design Variables	**Best solution found**		
	This paper	**Arora [28]**	**Belegundu [29]**
$x_1(d)$	0.051480	0.053396	0.050000
$x_2(D)$	0.351661	0.399180	0.315900
$x_3(N)$	11.632201	9.185400	14.25000
$g_1(\mathbf{X})$	-0.002080	0.000019	-0.000014
$g_2(\mathbf{X})$	-0.000110	-0.000018	-0.003782
$g_3(\mathbf{X})$	-4.026318	-4.123832	-3.938302
$g_4(\mathbf{X})$	-4.026318	-0.698283	-0.756067
$f(\mathbf{X})$	**0.0127047834**	**0.0127302737**	**0.0128334375**

6.4 Example 4

This problem was originally proposed by Himmelblau [30] and solved using the Generalized Reduced Gradient method (GRG). Gen and Cheng [31] solved this problem using a genetic algorithm based on both local and global reference. The result shown in Table 5 is the best found with their approach.

Homaifar, Qi, and Lai [6] solved this problem using a genetic algorithm with a population size of 400, and their results were previously the best reported in the literature (see Table 5).

The solution shown for the technique proposed here is the best produced after 11 runs, and using the following ranges for the design variables and the weights: $78.0000 \leq x_1 \leq 102.0000$, $33.0000 \leq x_2 \leq 45.0000$, $27.0000 \leq x_3 \leq 45.0000$, $27.0000 \leq x_4 \leq 45.0000$, $27.0000 \leq x_5 \leq 45.0000$, $1 \leq w_1 \leq 999$, and $1 \leq w_2 \leq 999$. The values for x_1 to x_5 were considered with a 4-decimals precision, and the weights w_1 and w_2 were considered as integers.

The mean from the 11 runs performed was $f(\mathbf{X}) = -30984.24070309$, with a standard deviation of 73.63353661.The worst solution found was $f(\mathbf{X}) = -30792.4077377525$, which is better than the best solution previously reported in the literature. The solution at the median was $f(\mathbf{X}) = -31017.21369099$ (corresponding to $x_1 = 78.010$, $x_2 = 33.030$, $x_3 = 27.119$, $x_4 = 45.000$, and $x_5 = 44.872$).

Table 5
Comparison of the results for the fourth example (Himmelblau's function).

Design Variables	Best solution found			
	This paper	**Gen [31]**	**Homaifar [6]**	**GRG [30]**
x_1	78.0495	81.4900	78.0000	78.6200
x_2	33.0070	34.0900	33.0000	33.4400
x_3	27.0810	31.2400	29.9950	31.0700
x_4	45.0000	42.2000	45.0000	44.1800
x_5	44.9400	34.3700	36.7760	35.2200
$g_1(\mathbf{X})$	91.997635	90.522543	90.714681	90.520761
$g_2(\mathbf{X})$	100.407857	99.318806	98.840511	98.892933
$g_3(\mathbf{X})$	20.001911	20.060410	19.999935	20.131578
$f(\mathbf{X})$	**−31020.859**	**−30183.576**	**−30665.609**	**−30373.949**

7 Discussion

Despite the fact that the proposed approach requires more function evaluations than running a GA on a single population, it could be argued that in practice the proposed approach turns out to be more efficient because it does not require the traditional fine-tuning of a simple GA which is normally performed by trial and error and normally takes a considerable amount of time. In any case, the introduction of parallel techniques (for which the approach is very suitable) should eliminate this potential drawback in the future.

It is worth mentioning that during the development of this approach several other variations of the same idea were tried without much success. For example, it was attempted to encode the weights of the penalties in the string itself, to avoid the use of another population, but the selection pressure turned out to be too high and the GA would tend to prematurely converge unless sharing [37] was used, and even in that case, the optimization results were normally very poor.

An interesting remark derived from the experiments performed was that the direct use of the final penalty values obtained with the proposed approach did not drive a simple GA to the solution expected even if this was run for a fairly large number of generations or with large populations. The reason seems to be the constant reuse of $P1$ that introduces different penalty factors during the evolution process rather than using a fixed (static) set of values (as with a simple GA). These constant changes in the penalty factors allow not only to keep enough diversity in the population (i.e., there are enough chromosomes

encoding different solutions) as to encourage that better solutions emerge from the main population (i.e., $P2$, which is the population responsible for optimizing the objective function) but also produce a dynamic penalty function that is being adjusted based on its effectiveness.

Finally, several experiments were run to try to find suitable values for the four parameters needed: $Gmax1$, $Gmax2$, $M1$ and $M2$. Initially, it was found that in most cases a fairly small population size for $P2$ ($\leq$ 40 chromosomes) would suffice to find reasonable solutions (unless within a 5% vicinity of the best solution known), but the size of $P1$ was much more dependent on the nature of the problem, although in all cases it was sufficient to use sizes smaller than those normally used with a simple GA (between 30 and 60 chromosomes). Similarly, the effect that the maximum number of generations produced in the results seemed to be more significant for $P1$ than for $P2$. This is not very surprising, since $P1$ is really the population responsible for performing the optimization. It is interesting to mention that in the experiments performed, it was found that the increment of the maximum number of generations for $P1$ would normally improve the quality of the solution, but there was always a threshold after which an increment did not affect the results in a significant manner. On the other hand, the increment of the maximum number of generations for $P2$ was normally not very beneficial, and that was the reason why it was normally preferred to use smaller values for $Gmax2$ than for $Gmax1$.

8 Conclusions

This paper has introduced a new GA-based technique that uses co-evolution to adjust automatically the weight factors of a penalty function to find the optimum of a constrained optimization problem. Due to the intrinsic limitations of penalty functions to handle equality constraints, only inequality constraints were considered in this work, although alternative hybrid approaches [13] may be used in combination with the proposed technique in order to deal with equality constraints, too.

The new technique worked well in several test problems that had been previously solved using GA-based and mathematical programming techniques, producing in all cases results better than those previously reported in the literature. The technique was able to achieve such good results with relatively small populations, and using a relatively low number of generations. However, performance issues remain to be solved, and it is desirable to develop a parallel version of this algorithm in the future.

9 Future Work

The first extension of this work is to develop a parallel implementation of the algorithm, so that instead of re-using $P1$ (see Figure 3) for each A_j, all sub-populations required by $P2$ can co-evolve concurrently. In this new version of the algorithm, currently under development, the top chromosomes (in terms of fitness values) of each sub-population interacting with $P2$ will be kept to re-start the evolution process once $P2$ had been evaluated. Nevertheless, there are still some issues to be solved with this new version of the technique, mainly with respect to the sort of interaction that will be imposed among the different populations, which will condition the topology used for the implementation of the corresponding distributed system.

It would also be interesting to conduct more studies of the effect of the four parameters required to execute the new algorithm ($Gmax1$, $Gmax2$, $M1$ and $M2$), to draw more general conclusions about its behavior (the selection of these parameters has been always an important issue when using a simple genetic algorithm [23]).

10 Acknowledgments

The author would like to thank the two anonymous reviewers for their valuable comments that helped him improve this paper, and he also acknowledges support from CONACyT through project number I-29870 A.

References

[1] Ian Parmee, editor. *The Integration of Evolutionary and Adaptive Computing Technologies with Product/System Design and Realisation.* Springer-Verlag, Plymouth, United Kingdom, 1998.

[2] Thomas Bäck, editor. *Proceedings of the Seventh International Conference on Genetic Algorithms.* Morgan Kaufmann Publishers, San Mateo, California, July 1997.

[3] Dipankar Dasgupta and Zbigniew Michalewicz, editors. *Evolutionary Algorithms in Engineering Applications.* Springer-Verlag, Berlin, 1997.

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] David Powell and Michael M. Skolnick. Using genetic algorithms in engineering design optimization with non-linear constraints. In Stephanie Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 424–431, San Mateo, California, jul 1993. University of Illinois at Urbana-Champaign, Morgan Kaufmann Publishers.

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

[12] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs.* Springer-Verlag, second edition, 1992.

[13] 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.

[14] 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.

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

[16] J. C. Bean and A. B. 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.

[17] Hans-Paul Schwefel. *Numerical Optimization of Computer Models.* John Wiley & Sons, Great Britain, 1981.

[18] David B. Fogel and L. C. Stayton. On the Effectiveness of Crossover in Simulated Evolutionary Optimization. *BioSystems*, 32:171–182, 1994.

[19] Carlos A. Coello Coello, Filiberto Santos Hernández, and Francisco Alonso Farrera. Optimal design of reinforced concrete beams using genetic algorithms. *Expert Systems with Applications : An International Journal*, 12(1), January 1997.

[20] Carlos A. Coello Coello and Alan D. Christiansen. A simple genetic algorithm for the design of reinforced concrete beams. *Engineering with Computers*, 13(4):185–196, 1997.

[21] Carlos Artemio Coello Coello. *An Empirical Study of Evolutionary Techniques for Multiobjective Optimization in Engineering Design.* PhD thesis, Department of Computer Science, Tulane University, New Orleans, LA, apr 1996.

[22] Gilbert Syswerda. Uniform Crossover in Genetic Algorithms. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 2–9, San Mateo, California, jun 1989. George Mason University, Morgan Kaufmann Publishers.

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

[24] Carlos A. Coello Coello, Alan D. Christiansen, and Arturo Hernández Aguirre. Using a new GA-based multiobjective optimization technique for the design of robot arms. *Robotica*, 16:401–414, 1998.

[25] J. Paredis. Co-evolutionary Constraint Satisfaction. In *Proceedings of the 3rd Conference on Parallel Problem Solving from Nature*, pages 46–55, New York, 1994. Springer Verlag.

[26] B. K. Kannan and S. N. Kramer. 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, 1994.

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

[28] Jasbir S. Arora. *Introduction to Optimum Design.* McGraw-Hill, New York, 1989.

[29] Ashok Dhondu Belegundu. *A Study of Mathematical Programming Methods for Structural Optimization.* Dept. of civil and environmental engineering, University of Iowa, Iowa, Iowa, 1982.

[30] David M. Himmelblau. *Applied Nonlinear Programming.* McGraw-Hill, New York, 1972.

[31] Mitsuo Gen and Runwei Cheng. *Genetic Algorithms & Engineering Design.* John Wiley & Sons, Inc, New York, 1997.

[32] Kalyanmoy Deb. GeneAS: A Robust Optimal Design Technique for Mechanical Component Design. In Dipankar Dasgupta and Zbigniew Michalewicz, editors, *Evolutionary Algorithms in Engineering Applications*, pages 497–514. Springer-Verlag, Berlin, 1997.

[33] E. Sandgren. Nonlinear integer and discrete programming in mechanical design. In *Proceedings of the ASME Design Technology Conference*, pages 95–105, Kissimine, Florida, 1988.

[34] Kalyanmoy Deb. Optimal Design of a Welded Beam via Genetic Algorithms. *AIAA Journal*, 29(11):2013–2015, November 1991.

[35] K. M. Ragsdell and D. T. Phillips. Optimal Design of a Class of Welded Structures Using Geometric Programming. *ASME Journal of Engineering for Industries*, 98(3):1021–1025, 1976. Series B.

[36] James N. Siddall. *Analytical Design-Making in Engineering Design.* Prentice-Hall, 1972.

[37] Kalyanmoy Deb and David E. Goldberg. An investigation of niche and species formation in genetic function optimization. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 42–50, San Mateo, California, jun 1989. George Mason University, Morgan Kaufmann Publishers.