A Combined Genetic Adaptive Search (GeneAS) for Engineering Design

Kalyanmoy Deb and Mayank Goyal
Department of Mechanical Engineering
Indian Institute of Technology
Kanpur, UP 208 016, INDIA
email: deb@iitk.ernet.in

Abstract

In this paper, a flexible yet efficient algorithm for solving engineering design optimization problems is presented. The algorithm is developed based on both binary-coded and real-coded genetic algorithms (GAs). Since both GAs are used, the variables involving discrete, continuous, and zero-one variables are handled quite efficiently. The algorithm restricts its search only to the permissible values of the variables, thereby reducing the search effort in converging to the optimum solution. The efficiency and ease of application of the proposed method is demonstrated by solving three different mechanical component design problems borrowed from the optimization literature. The proposed technique is compared with binary-coded genetic algorithms, Augmented Lagrange multiplier method, Branch and Bound method and Hooke and Jeeves pattern search method. In all cases, the solutions obtained using the proposed technique are superior than those obtained with other methods. These results are encouraging and suggest the use of the proposed technique to other engineering design problems.

1 Introduction

The aim in an engineering design problem is to minimize or maximize a design objective and satisfy certain constraints. The constraints can be either inequality or equality type. In general, an engineering design optimization problem (also known as a NLP problem) can be expressed in the following form (Deb, 1995; Rao, 1984):

$$\left.\begin{array}{lll} & \text{Minimize} & f(\boldsymbol{x}) \\ \text{subject to} & & \\ & g_j(\boldsymbol{x}) \geq 0, & j = 1, 2, \ldots, J; \\ & h_k(\boldsymbol{x}) = 0, & k = 1, 2, \ldots, K; \\ & x_i^{(L)} \leq x_i \leq x_i^{(U)}, & i = 1, 2, \ldots, N. \end{array}\right\} \quad (1)$$

The variable vector $\boldsymbol{x}$ represents a set of variables x_i, $i = 1, 2, \ldots, N$. When faced with NLP problems involving different types of objective function, constraints, and variables, designers must rely on an optimization algorithm which is flexible enough to handle many such problems. However, the traditional optimization algorithms are handicapped in terms of handling different types of design variables efficiently. The engineering design optimization problems usually involve two distinct types of variables, which take any real value and which take a few discrete values only. We shall refer to the former type of variables as continuous variables and the latter as discrete variables. Discrete variables can take values in steps of a fixed or variable size. Some discrete variables can only take one of two options (zero-one variables), such as the presence or absence of a member in a truss structure design. Since most traditional optimization methods are designed to work with continuous variables only, in dealing with design problems having discrete variables, these methods do not work as well. Most methods handle discrete variables by adding artificial constraints (Kannan and Kramer, 1993). In those methods, discrete variables are treated as continuous variables during optimization and the search algorithm pushes the solutions to converge to feasible values of the variables (Deb, 1995; Kannan and Kramer, 1993; Reklaitis, Ravindran, and Ragsdell, 1983). This fix-up not only increases the complexity of the underlying problem, but the algorithm also spends a considerable amount of effort in evaluating non-feasible solutions.

In this paper, we suggest a combined genetic adaptive search technique to handle mixed variables and mixed functional form for objective functions and constraints. The normalized penalty function approach is used to handle constraints and objective function complexities, however a *natural* coding scheme is used to represent mixed variables. The efficacy of the proposed method is demonstrated by solving three different engineering design problems and by comparing the solutions with that obtained using a few traditional optimization methods.

2 Genetic Adaptive Search

The **Gene**tic **A**daptive **S**earch (GeneAS) is a modification of the standard genetic algorithm (GA) search technique. Although the basic working principles of both GeneAS and GA are the same, GeneAS is more flexible and efficient in solving engineering design problems. To demonstrate the need to have a flexible optimization method for solving engineering design problems, let us consider a simple cantilever beam design problem. A cantilever beam is a beam that is rigidly supported at one end and carries a certain load on the other end. In trying to design such a beam, designers face a number of possibilities. For simplicity, the beam can be assumed to have a circular cross-section or a square cross-section. Thus, the shape of the beam constitute a binary (zero-one) design variable. Once the shape is determined, the next choice is the size of the shape. If the chosen shape is circular, the diameter of the cross-section is a variable. However, if the chosen shape is a square, the side of the square cross-section is a design variable. Since circular or square cross-sections are not available in any arbitrary sizes, this variable is a discrete variable taking a few permissible values. The length of the beam could be another design variable. Since the beam can be cut to a large precision, we may consider this variable as a continuous variable taking any positive real value. These three design variables fix the size and shape of the beam to their permissible values. Another variable which is very difficult to include in any traditional optimization method is the material of the beam. This variable is again a discrete variable taking only a few chosen materials. One distinct difference between this variable and the second discrete variable representing the size of the cross-section is that no value other than the permissible values of this variable is strictly allowed during the optimization process. For example, for the material variable if the steel is represented as 1 and cast iron is represented as 2, no intermediate value (such as 1.4 or other) is allowed. Ideally, a designer is interested in an optimization algorithm which has the flexibilities to handle such a mixed type of variables. As mentioned earlier, traditional optimization methods cannot handle such flexibilities (or repudiates) in the design variables. In the following, we describe the GeneAS technique which is flexible enough to handle such engineering design problems.

GeneAS uses a combination of binary-coded and real-coded GA, depending on the nature of the design variables. In the following, we first describe the binary-coded GA and real-coded GA and the discuss the GeneAS approach.

2.1 Binary GA

In a binary-coded GA, the variables are represented in binary strings formed with `1` and `0`. A detailed discussion of the binary-coded GA is provided in the preface. To illustrate further, we present how the above cantilever beam design problem can be handled using a binary-coded GA. Recall that there are four design variables. A typical binary string representing four variables may be as follows:

```
(1)  (0010111)  (0110)  (101)
```

The first variable is a one bit substring and can take only one of two values (either `1` or `0`). The value `1` represents a circular cross-section for the cantilever beam and the value `0` represents a square cross-section of the cantilever beam. The second variable is represented as a 7 bit substring representing the diameter of the circular section if the first variable is a `1` or the side of the square if the first variable is a `0`. The substring of size 7 is chosen to have a particular accuracy in the solution. As outlined in the preface, with a 7-bit substring representing the diameter or the side in the range (3, 130) mm, the accuracy in the solution would be 1 mm. The above substring codes to a value 26 mm. The third variable is represented as a 4 bit substring. In the range of (50, 125) cm, the length variable has an accuracy of 5 cm and the above substring decodes to a value 80 cm. The fourth variable indicates the 6-th material in the list of the 8 possible materials. Thus, the above string represents a cantilever beam having a circular cross-section having 26 mm diameter and 80 cm long and made of the 6-th material from a list of 8 possible materials.

The standard tripartite binary-coded GA (with reproduction, crossover, and mutation) can be applied on the above coding. The above problem involves a number of constraints restricting the stress and deflection values at certain critical points in the cantilever beam. GAs usually handle such constraints by using a penalty function, as described in the preface. Sample codes for a single-point crossover operator and a bit-wise mutation operator are given in the Appendix.

Although the binary-coded GA as mentioned above can be used to find an optimal or a near-optimal solution, the binary-coding is not all that flexible to handle vagaries of design variables. For example, in the above coding the size variable (the second variable) is conveniently assumed to take all diameter or side values ranging from 3 mm to 130 mm in an interval of 1 mm, thereby making the total number of choices equal to 128, which is 2^k where $k = 7$, an integer. However, it is not necessary that the total number of permissible choices should be 2^k (where k is an integer) always. For example, the total number of choices for the diameter or side may be 98 (3 to 100 mm). In such cases, it becomes difficult to use a fixed length binary coding to represent all permissible values. To tackle such cases, researchers use a coding having total number alternatives equal to the next available 2^k. In the case of 98 alternatives, 7-bits (with 128 alternatives) can be used. Thereafter, in order to discourage the non-permissible values, they use an additional constraint (such as $x_2 \leq 100$) penalizing extra values. This increases the complexity of the problem. Another problem with the fixed-length binary coding is the arbitrary precision that can achieved. With a fixed-length coding, the attainable precision is also fixed. Moreover, the binary-coding has Hamming cliff problems which sometimes cause difficulty in the case of coding continuous variables (Deb and Agrawal, 1995).

In the following, we present a real-coded GA which eliminates all the above problems of coding a continuous variable in a fixed-length string. An extension of the real-coded GA can also be used handle discretely-coded variables having a total number of alternatives which is not 2^k.

2.2 Real-coded GA

In a real-coded GA, the variables are used directly. For example, if the above cantilever beam design problem is solved using a real-coded GA, the following is solution:

$$\underbrace{26.0}_{\text{diameter/side}} \quad \underbrace{80.0}_{\text{length}}$$

Since the shape of the beam and the material take only discrete values, they cannot be used as a variable in the real-coded GAs. The above coding assumes a particular cross-sectional shape and a particular material. In the above coding, we have also assumed that the size variable (the first variable) takes any real value. The second variable is the length variable taking a value of 80 cm. Although we do not claim that this real coding is flexible to handle different types of design variables, we show how the proposed real-coded GA can be used to tackle continuous variables efficiently and eliminate the problem of binary-coded GAs in handling continuous variables. Later, we present a combined approach where the best features of binary and real-coded GAs are used together to constitute a flexible optimization algorithm.

The evaluation procedure and the reproduction operator in the real-coded GA remain the same as that in binary-coded GA, however the crossover and mutation operators must be different. Although there exist a number of real-coded crossover operators (Wright, 1991;Eshelman and Schaffer, 1993), recently Deb and Agrawal (1995) have suggested a Simulated Binary Crossover (SBX) operator which respects the *interval* schema processing and matches the search power of a binary-coded crossover operator. Without going into the details, we provide here the procedure of performing the crossover and mutation operators.

2.2.1 Crossover handling continuous variables

The procedure of computing the children solutions $x_i^{(1,t+1)}$ and $x_i^{(2,t+1)}$ from parent solutions $x_i^{(1,t)}$ and $x_i^{(2,t)}$ is described as follows. A spread factor β is defined as the ratio of the absolute difference in children values to that of the parent values:

$$\beta = \left| \frac{x_i^{2,t+1} - x_i^{1,t+1}}{x_i^{2,t} - x_i^{1,t}} \right|. \tag{2}$$

First, a random number u between 0 and 1 is created. Thereafter, from a specified probability distribution function, the ordinate $\bar{\beta}$ is found so that the area under the probability curve from 0 to $\bar{\beta}$ is equal to the chosen random number u. The probability distribution used to create a child solution is derived from an analysis of search power and is given as follows (Deb and Agrawal, 1995):

$$\mathcal{C}(\beta) = \begin{cases} 0.5(n+1)\beta^n, & \text{if } \beta \le 1; \\ 0.5(n+1)\frac{1}{\beta^{n+2}}, & \text{otherwise.} \end{cases} \tag{3}$$

Figure 1 shows the above probability distribution with $n = 2$ for creating children solutions from two parent solutions ($x = 5$ and $x = 10$) in the real space. In the above expressions, the distribution index n is any nonnegative real number. A large value of n gives a higher probability for creating near parent values and a small value of n allows distant points to be selected as children solutions. Using equation 3, we calculate $\bar{\beta}$ by equating the area under the probability curve equal to u, as follows (refer to the code `get_beta(u)` given in the Appendix):

$$\bar{\beta} = \begin{cases} (2u)^{\frac{1}{n+1}}, & \text{if } u \le 0.5; \\ \left(\frac{1}{2(1-u)}\right)^{\frac{1}{n+1}}, & \text{otherwise.} \end{cases} \tag{4}$$

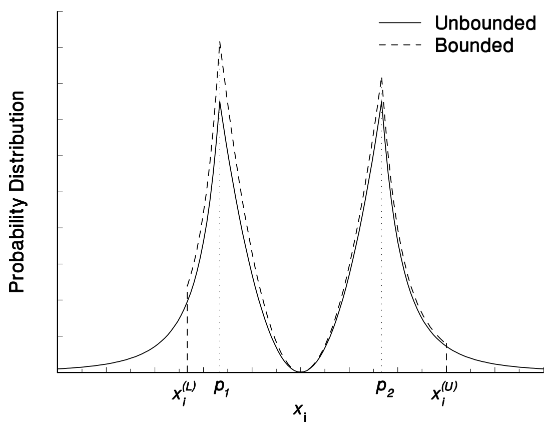

Figure 1: Probability distribution for creating children solutions of continuous variables

After obtaining $\bar{\beta}$ from the above probability distribution, the children solutions are calculated as follows:

$$x_i^{(1,t+1)} = 0.5\left[(1+\bar{\beta})x_i^{(1,t)} + (1-\bar{\beta})x_i^{(2,t)}\right], \tag{5}$$

$$x_i^{(2,t+1)} = 0.5\left[(1-\bar{\beta})x_i^{(1,t)} + (1+\bar{\beta})x_i^{(2,t)}\right]. \tag{6}$$

Note that two children solutions are symmetric about the parent solutions. This is deliberately used to avoid any bias towards any particular parent solution. Another interesting aspect of this crossover operator is that for a fixed probability distribution if the parent values are far from each other, the distant children values are possible to be created. But if the parent values are close by, distant children values are not likely. Initially when the solutions are random, this allows almost any values to be created as a child solution. But when the solutions tend to converge due to the action of genetic operators, distant solutions are not allowed, thereby focusing the search to a narrow region. A sample code for this crossover operator is given in the Appendix.

2.2.2 Crossover handling continuous variables having fixed bounds

Although the above SBX procedure can create a child solution almost anywhere in the real space, a bounded SBX can also be implemented by restricting solutions within specified limits $(x_i^{(L)}, x_i^{(U)})$. The probability distribution function given in Equation 3 is multiplied by a suitable factor depending on these limits and the location of the parent solutions, so as to make a zero probability of creating any solution outside these limits (dashed line in Figure 1). For the child solution closer to parent p_1, this factor can be computed as $1/(1-v_1)$, where v_1 is the cumulative probability of creating solutions from $x_i = -\infty$ to $x_i = x_i^{(L)}$. Similarly, a factor for the child solution closer to p_2 can also be calculated.

The above variable-by-variable crossover scheme allows any binary crossover operator to be simulated easily. For example, a single-point crossover can be simulated by using the SBX operator on k-th variable (chosen at random) and by swapping all $(k+1)$-st to N-th variables between the two parent solutions. Here, for simplicity, we use a scheme where, on an average, 50% of the variables get operated by SBX.

2.2.3 Crossover handling discrete variables

In order to handle discrete variables having arbitrary number of permissible values, a discrete version of the above probability distribution (equation 3) can be used. Figure 2 shows such a probability distribution, which creates only permissible values of the design variable.

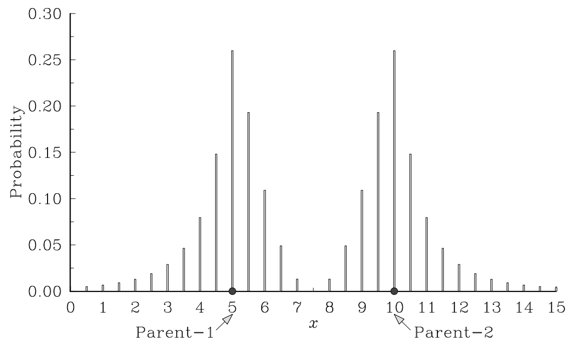

Figure 2: Probability distribution for creating children solutions of discrete variables

2.2.4 Mutation handling continuous variables

After the reproduction and crossover operators are applied, a mutation operator is used with a small mutation probability p_m. For a continuous variable, the current value of the variable is changed to a neighboring value using a polynomial probability distribution having its mean at the current value and its variance as a function of the distribution index n. To perform mutation, a perturbance factor δ is defined as follows:

$$\delta = \frac{c - p}{\Delta_{\max}}, \tag{7}$$

where $\Delta_{\max}$ is a fixed quantity, representing the maximum permissible perturbance in the parent value p and c is the mutated value. Similar to the crossover operator, the mutated value is calculated with a probability distribution that depends on the perturbance factor δ:

$$\mathcal{P}(\delta) = 0.5(n+1)(1-|\delta|)^n. \tag{8}$$

The above probability distribution is valid in the range $\delta \in (-1, 1)$. This distribution is shown in the Figure 3 for different values of distribution index, n. To create a mutated value, a random number u is created in the range $(0, 1)$. Thereafter, the following equation can be used to calculate the perturbance factor $\bar{\delta}$ corresponding to u using the above probability distribution:

$$\bar{\delta} = \begin{cases} (2u)^{\frac{1}{n+1}} - 1, & \text{if } u < 0.5; \\ 1 - [2(1-u)]^{\frac{1}{n+1}}, & \text{if } u \geq 0.5. \end{cases} \tag{9}$$

Thereafter, the mutated value is calculated as follows:

$$c = p + \bar{\delta}\Delta_{\max}. \tag{10}$$

A sample code (`real_mutation()`) for this mutation operator is given in the Appendix.

For discrete variables coded directly, a neighboring permissible value is chosen with a discrete version of the above probability distribution $\mathcal{P}(\delta)$.

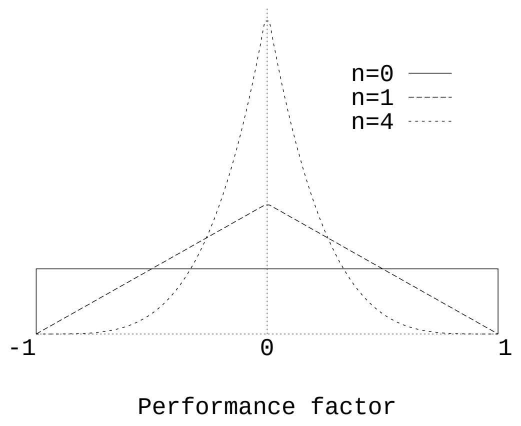

Figure 3: Probability distribution for creating a mutated value for continuous variables

2.3 Combined GA (GeneAS)

It is evident from the discussion of binary GAs and real-coded GAs that none of them can be used alone to efficiently handle different kinds of variables which are usually encountered in the engineering design problems. In the combined GA (GeneAS) proposed here, we combine appropriate features of both these GAs in a way so that any variable (whether discrete or continuous) can be handled efficiently.

The main difference between GeneAS and the two GAs discussed earlier is the way the variables are coded and the way crossover and mutation operators are applied. The coding procedure is as follows: If a variable is continuous, the variable is used directly, otherwise either a binary substring is used to represent the discrete variable or the variable is used directly. For example, in order to code the cantilever beam design problem mentioned earlier, a typical coding would be as follows:

```
(1)  26  90.25  (101)
```

The first and fourth variables are coded in binary substrings because these two variables are discrete in nature. The second variable is assumed to take discrete values but used directly (because the number of permissible choices is not equal to 2^k, where k is an integer) and the third variable is assumed to take any real value. It is noteworthy that this flexibility in the representation of mixed variables is difficult not only with binary or real-coded GAs alone, but also with traditional methods. Since the variables are represented in a mixed string, the crossover operator has to be applied variable by variable. Every variable is crossed with a specific probability (usually 0.5 is suggested so that about 50% of the variables are crossed). If a variable to be crossed is a discrete variable, the binary crossover operator is used, otherwise, the SBX operator is used. Similar modification for the combined mutation operator is also followed. Since the crossover and mutation operator always produce permissible solutions, the search effort needed in GeneAS to converge to an optimal or a near-optimal solution is expected to be much less as compared to other schemes where any value is allowed during the optimization process. Figure 4 shows a flowchart of the GeneAS algorithm. A sample code for performing mixed crossover and mutation operators is presented in the Appendix. Once the variables are represented by the mixed coding scheme described above, the *fitness* of the solution is computed from the objective function and constraints, as outlined in the preface.

In order to show the efficiency of GeneAS, we solve the three different mechanical component design problems.

```
/* Choose a coding for handling mixed variables */
for i = 1 to population_size
    old_solution[i] = random_solution;
    old_solution[i].fitness = fitness(old_solution[i]);
generation = 0;
repeat
    generation = generation + 1;
    for i = 1 to population_size step 2
        parent1 = reproduction(old_solution);
        parent2 = reproduction(old_solution);

        (child1, child2) = crossover(parent1, parent2);

        new_solution[i] = mutation(child1);
        new_solution[i+1] = mutation(child2);

        new_solution[i].fitness = fitness(new_solution[i]);
        new_solution[i+1].fitness = fitness(new_solution[i+1]);

    for i = 1 to population_size
        old_solution[i] = new_solution[i];
until (termination);
end.
```

Figure 4: Pseudo-code of a Genetic Algorithm

3 Case Studies

Three different problems are solved using GeneAS in this section. These problems were solved using different traditional optimization algorithms such as augmented Lagrange multiplier method (Kannan and Kramer, 1993), the Branch and Bound method (Sandgren, 1988) and Hooke-Jeeves pattern search method (Siddall, 1982). In these methods, the zero-one or discrete variables are handled by adding one additional equality constraint for each variable. This constraint is carefully crafted to penalize the infeasible values of the variables. The authors have concluded that the added equality constraints are nondifferentiable and cause multiple artificial 'bumps' making the search space unnecessarily complex and multimodal. In all runs of GeneAS, a crossover probability of 0.9 and a mutation probability of 0.0 are used, although a small non-zero mutation probability may be beneficial depending upon the problem. Since constraints are normalized, a fixed penalty parameter of $r = 100$ is used in all problems. A run is terminated after a specific number of function evaluations have been computed.

3.1 Gear train design

A compound gear train is to be designed to achieve a specific gear ratio between the driver and driven shafts (Figure 5). The objective of the gear train design is to find the number of teeth in

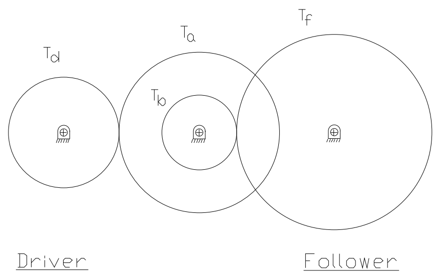

Figure 5: A compound gear train

each of the four gears so as to minimize the error between the obtained gear ratio and a required gear ratio of 1/6.931 (Kannan and Kramer, 1993). Since the number of teeth must be integers, all four variables are discrete. By denoting the variable vector $\boldsymbol{x} = (x_1, x_2, x_3, x_4) = (T_d, T_b, T_a, T_f)$., we write the NLP problem:

$$\begin{aligned} \text{Minimize} \quad & f(\boldsymbol{x}) = \left[\frac{1}{6.931} - \frac{x_1 x_2}{x_3 x_4}\right]^2 \\ \text{Subject to} \quad & 12 \le x_1, x_2, x_3, x_4 \le 60, \\ & \text{all } x_i\text{'s are integers.} \end{aligned}$$

Since all the four variables are discrete, each variable is coded in six-bit binary strings (having 2^6 or 64 values), so that variables take values between 12 and 75 (integer values) and four constraints ($60 - x_i \ge 0$, for $i = 1, 2, 3, 4$) are added to penalize infeasible solutions. Table 1 shows the best solutions found in each case with a population of 50 solutions. The 24-bit string best solution found is as follows:

$$\underbrace{(000101)}_{T_d} \quad \underbrace{(000010)}_{T_b} \quad \underbrace{(010101)}_{T_a} \quad \underbrace{(100110)}_{T_f}$$

It is clear from the table that GeneAS has found much better solutions than the previously known optimal solutions. When all four variables are coded directly as discrete variables and

Table 1: Gear train design

Design variables	Optimal solution (GeneAS)	(Kannan and Kramer)	(Sandgren)
x_1 (T_d)	17	13	18
x_2 (T_b)	14	15	22
x_3 (T_a)	33	33	45
x_4 (T_f)	50	41	60
$f(\boldsymbol{x})$	$1.362(10^{-09})$	$2.146(10^{-08})$	$5.712(10^{-06})$
Gear Ratio	0.144242	0.144124	0.146667
Error	0.026%	0.11%	1.65%

GeneAS is used, the following global solution emerged:

$$T_d = 19 \quad T_b = 16 \quad T_a = 49 \quad T_f = 43$$

The above solution has an objective function equal to $2.701(10^{-12})$ and a gear ratio of 0.144281, which is only 0.001% different from the ratio 1/6.931. This implementation eliminates the need of using any artificial constraint, since all four variables are allowed to take integer values in the range (12, 60). Both these GeneAS applications show the efficiency of the proposed method.

3.2 Belleville Spring Design

The objective is to design a Belleville spring having minimum weight and satisfying a number of constraints. A detailed description of the optimal design problem is presented in Siddall (1982). The problem has four design variables—external diameter of the spring (D_e), internal diameter of the spring (D_i), the thickness of the spring (t), and the height of the spring (h) as shown in Figure 6. In this problem, we choose t as a discrete variable and the rest three variables as

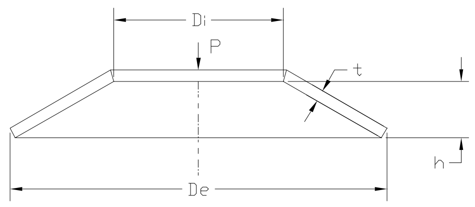

Figure 6: A belleville spring

continuous variables. Seven different inequality constraints based on stress, deflection, and some physical limitations are used. The complete NLP problem is shown below:

$$\begin{aligned}
\text{Minimize} \quad & f(\boldsymbol{x}) = 0.07075\pi(D_e^2 - D_i^2)t \\
\text{Subject to} \quad & g_1(\boldsymbol{x}) = S - \tfrac{4E\delta_{\max}}{(1-\mu^2)\alpha D_e^2}\left[\beta\left(h - \delta_{\max}/2\right) + \gamma t\right] \geq 0, \\
& g_2(\boldsymbol{x}) = \tfrac{4E\delta}{(1-\mu^2)\alpha D_e^2}\left[\left(h - \delta/2\right)\left(h - \delta\right)t + t^3\right]\bigg|_{\delta=\delta_{\max}} - P_{\max} \geq 0, \\
& g_3(\boldsymbol{x}) = \delta_l - \delta_{\max} \geq 0, \\
& g_4(\boldsymbol{x}) = H - h - t \geq 0, \\
& g_5(\boldsymbol{x}) = D_{\max} - D_e \geq 0, \\
& g_6(\boldsymbol{x}) = D_e - D_i \geq 0, \\
& g_7(\boldsymbol{x}) = 0.3 - \tfrac{h}{D_e - D_i} \geq 0.
\end{aligned}$$

The parameters $P_{\max} = 5,400$ lb and $\delta_{\max} = 0.2$ inch are the desired maximum load and deflection, respectively. The parameters $S = 200$ kPsi, $E = 30(10^6)$ psi and $\mu = 0.3$ are the allowable strength, the modulus of elasticity, and the Poisson's ratio for the material used. The parameter δ_l is the limiting value of the maximum deflection. The parameters $H = 2$ inch and $D_{\max} = 12.01$ inch are the maximum limit on the overall height and outside diameter of the spring. Assuming $K = D_e/D_i$, we present the other parameters in the following:

$$\begin{aligned}
\alpha &= \frac{6}{\pi \ln K}\left(\frac{K-1}{K}\right)^2, \\
\beta &= \frac{6}{\pi \ln K}\left(\frac{K-1}{\ln K} - 1\right), \\
\gamma &= \frac{6}{\pi \ln K}\left(\frac{K-1}{2}\right).
\end{aligned}$$

The first constraint limits the maximum compressive stress developed in the spring to the allowable compressive strength (S) of the spring material (Shigley and Mischke, 1986). The second constraint limits the maximum deflection (P) of the spring to be at least equal to the desired maximum deflection ($P_{\max}$). Although the ideal situation would be to have an equality constraint $P = P_{\max}$, the optimal solution will achieve the equality condition together with the first constraint and the objective of minimizing the weight of the spring. In order to achieve the desired maximum deflection to be smaller than the height of the spring, the third constraint is added. In this constraint, a nonlinear functional form of the limiting deflection δ_l is assumed (Siddall, 1982), so that δ_l is always smaller than or equal to h. The fourth constraint takes care of the total height of the spring to be lower than the specified maximum limit (H). The fifth constraint limits the outside diameter of the spring to be at most equal to the maximum limit ($D_{\max}$). The sixth constraint allows the outside diameter of the spring to be larger than the inside diameter of the spring. The final constraint limits the slope of the conical portion of the spring to be at most equal to 0.3 (about 16.7 degrees).

Constraints are normalized and handled using the penalty function method. The population is initialized in the following intervals:

$$0.01 \leq t \leq 0.6, \quad 0.05 \leq h \leq 0.5, \quad 5 \leq D_i \leq 15, \quad 5 \leq D_e \leq 15.$$

GeneAS is run with a population size of 100 for 100 generations. Realizing that the thickness of the spring may be available only in discrete sizes, we have considered thickness as a discrete variable (available in steps of 0.01 inches) and kept the other three variables as continuous. The

Table 2: Optimal solutions obtained using two different techniques

Design	Optimal solution	
variables	GeneAS	(Siddall)
t	0.210	0.204
h	0.204	0.200
D_i	9.268	10.030
D_e	11.499	12.010
g_1	1988.370	134.082
g_2	197.726	-12.537
g_3	0.004	0.000
g_4	1.586	1.596
g_5	0.511	0.000
g_6	2.230	1.980
g_7	0.208	0.199
Weight	2.162	1.980

crossover operator with the discrete probability distribution is used to search this variable. Using the same parameters, we obtain a solution with 2.162 lb. The solution is shown in Table 2 and the load-deflection characteristic of this spring in shown in Table 3. In this case, the maximum load required to achieve the maximum deflection of 0.2 inch need not be exactly 5,400 lb, because of the discreteness in the search space. The latter table shows that this maximum deflection is 5597.82 lb, which is larger than 5,400 lb.

We have also compared these results with the solution reported in Siddall (1982). With our parameter setting[1] and using the optimal values of the variables reported in Siddall (1982), we observe that the second constraint is not satisfied (g_2 is negative) for the solution presented in Siddall (1982). This is because the maximum load required to achieve the desired maximum deflection is smaller than the desired load of 5,400 lb (Table 3). Although this solution has a weight of 1.980 lb (better in weight than even GeneAS), the solution is not feasible according to our calculation. Thus, we can not compare the solutions of GeneAS with that reported in Siddall (1982).

3.3 Welded beam design

A rectangular beam needs to be welded as a cantilever beam to carry a certain load. The objective of the design of the welded beam is to minimize the overall cost of the fabrication which involves the cost of beam material and the cost of weld deposit. Two different welding configurations are assumed as shown in Figure 7. A binary variable x_1 is assigned to code this choice of configurations (`1` is used for four-sided welding and `0` is used for two-sided welding). One of the four materials

[1]Sometimes, the accuracy of the solution depends on the machine precision and the exact value of the constant such as π etc.

Table 3: Load-deflection characteristics of the optimum solutions obtained using different methods (Load in lb. and deflection in inch)

Deflection	0.02	0.04	0.06	0.08	0.10
Siddall	981.49	1823.17	2540.57	3149.22	3664.67
GeneAS	1003.77	1868.65	2609.73	3242.09	3780.81
Deflection	0.12	0.14	0.16	0.18	0.20
Siddall	4102.44	4478.07	4807.10	5105.06	5387.48
GeneAS	4240.98	4637.68	4986.00	5301.02	5597.82

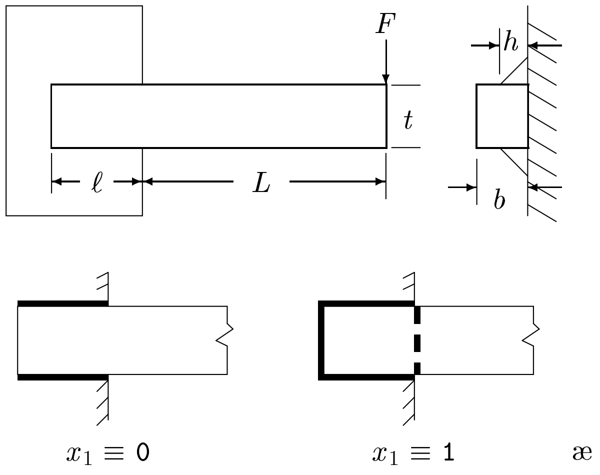

Figure 7: Welded beam problem

(steel, cast iron, aluminum, and brass) is to be used for the beam. A two-bit string is used to code the material as x_2 (the string 00 is assigned steel, 01 is assigned cast iron, 10 is assigned aluminum, and 11 is assigned brass). Thereafter, three discrete variables (with steps of 0.0625) are used to code the thickness of the weld ($x_3 = h$), the width of the beam ($x_4 = t$), and the thickness of the beam ($x_5 = b$). Finally, the length of welded joint ($x_6 = \ell$) is declared as a continuous variable. A typical GeneAS solution of the above problem looks like the following:

$$\underbrace{(0)}_{\text{weld type}} \quad \underbrace{(10)}_{\text{material}} \quad \underbrace{0.0625}_{h} \quad \underbrace{0.625}_{t} \quad \underbrace{0.0625}_{b} \quad \underbrace{1.563}_{\ell}$$

The NLP problem of the above welded beam problem is as follows:

$$\begin{aligned}
\text{Minimize} \quad & f(\boldsymbol{x}) = (1+c_1)x_3^2(x_6+x_1x_4) + c_2x_4x_5(L+x_6) \\
\text{Subject to} \quad & g_1(\boldsymbol{x}) = S - \sigma(\boldsymbol{x}) \geq 0, \\
& g_2(\boldsymbol{x}) = P_c(\boldsymbol{x}) - F \geq 0, \\
& g_3(\boldsymbol{x}) = \delta_{\max} - \delta(\boldsymbol{x}) \geq 0, \\
& g_4(\boldsymbol{x}) = 0.577S - \tau(\boldsymbol{x}) \geq 0, \\
& x_1 \text{ is a zero-one variable and } x_2 \text{ is a ternary variable,} \\
& x_3,\ x_4, \text{ and } x_5 \text{ are discrete and } x_6 \text{ is continuous.}
\end{aligned}$$

In the above formulation, the stress, deflection, and buckling terms are the same as that derived in Reklaitis, Ravindran, and Ragsdell (1983). The parameters $L = 14$ inch, $\delta_{\max} = 0.25$ inch, and $F = 6,000$ lb are used. The cost term and material properties used in the above formulation are as follows:

$$\begin{cases}
x_1 = 0, \quad A = 1.414x_3x_6, & J = 1.414x_3x_6\left[\frac{(x_3+x_4)^2}{4} + \frac{x_6^2}{12}\right], \quad \text{if } x_1 \text{ is } \texttt{0}; \\
x_1 = 1, \quad A = 1.414x_3(x_4+x_6), & J = 1.414x_3\frac{(x_3+x_4+x_6)^3}{12}, \quad \text{otherwise.}
\end{cases}$$

$$\begin{cases}
S = 30(10^3), & E = 30(10^6), & G = 12(10^6), & c_1 = 0.1047, & c_2 = 0.0481, & \text{if } x_2 \text{ is } \texttt{00} \text{ (Steel);} \\
S = 8(10^3), & E = 14(10^6), & G = 6(10^6), & c_1 = 0.0489, & c_2 = 0.0224, & \text{if } x_2 \text{ is } \texttt{01} \text{ (CI);} \\
S = 5(10^3), & E = 10(10^6), & G = 4(10^6), & c_1 = 0.5235, & c_2 = 0.2405, & \text{if } x_2 \text{ is } \texttt{10} \text{ (Al);} \\
S = 8(10^3), & E = 16(10^6), & G = 6(10^6), & c_1 = 0.5584, & c_2 = 0.2566, & \text{if } x_2 \text{ is } \texttt{11} \text{ (Brass).}
\end{cases}$$

An initial random population of 50 solutions are created in the range $0.0625 \leq x_3, x_5 \leq 2$ and $2 \leq x_4, x_6 \leq 20$. However, GeneAS is allowed to create any other solution (h, b, and t in steps of 0.0625 and any real value of ℓ) in subsequent iterations. All constraints and the objective function are normalized. Table 4 shows that the optimal solution to the above problem is a steel beam (0.25×8.25 inch rectangular cross-section) being welded on four sides (8.25×1.6849 inch rectangle) of weld size 0.1875 inch. It is interesting to note that the choice of steel as an optimal material is intuitive. The cost factors and the material properties are so chosen that steel becomes the intuitive choice. Although, the cast iron is cheaper than the steel (about half the price), the material strength of cast iron is inferior to steel (almost 4 times). Thus, steel has a better strength per unit cost of material than cast iron. The other two materials have much lower strength per cost of material. The above solution is likely to be very near to the true optimal solution, as one of the constraints (g_4) is very close to zero, meaning that the material is well utilized to have a shear strength of the weld equal to the allowable limit (In fact, the shear stress is only $9.4(10^{-4})\%$ smaller than the allowable limit). The bending stress is 1.3% smaller than the allowable limit and the critical buckling load is 6.7% higher than the applied load.

It is noteworthy that this problem is computationally expensive to solve using a traditional optimization method. The only way to solve this problem is to form 8 different optimization problems for each combination of weld configuration (2 choices) and material (4 choices). When all 8 problems are solved and an optimum solution is found for each case, they can be compared to find the best solution. If the number of such discrete choices increase, the computational complexity to solve the problem increases. However, the application of GeneAS in this problem demonstrates the ease and flexibility of solving such complex engineering design problems.

Table 4: Welded beam design

Design Variables	Optimum solution (GeneAS)
x_1 (1), zero-one	Four-sided
x_2 (00), discrete	Steel
x_3 (h), discrete	0.1875
x_4 (t), discrete	8.2500
x_5 (b), discrete	0.2500
x_6 (ℓ), continuous	1.6849
$g_1(\boldsymbol{x})$	380.1660
$g_2(\boldsymbol{x})$	402.0473
$g_3(\boldsymbol{x})$	0.2346
$g_4(\boldsymbol{x})$	0.1621
$f(\boldsymbol{x})$	1.9422

4 Conclusions

In this article, a new yet efficient optimization algorithm is developed and used to solve a number of mechanical component design problems. The algorithm allows a natural coding of design variables by considering discrete and/or continuous variables. In the coding of GeneAS, the discrete variables are either coded in a binary string or used directly and the continuous variables are coded directly. In the search of discrete variables coded in binary strings, the binary single-point crossover operator is used. However, in the case of continuous variables, the SBX operator developed in an earlier study (Deb and Agrawal, 1995) is used. When discrete variables are used directly, a discrete probability distribution is used in the crossover operator. The mutation operator is also modified suitably in order to create permissible solutions only. The results presented in this paper show that GeneAS is a flexible yet efficient optimization technique in handling mixed types of variables. In order to handle discrete or integer variables, no additional constraint or no other special consideration is needed. Thus, this method can be used in a wide variety of problem domain. In solving two different mechanical design problems, GeneAS has always outperformed three previously-known methods. GeneAS has also found a near-optimal solution in a welded-beam design problem having mixed variables. Moreover, the ease of accommodating different types of variables in a mixed-variable programming problem is also demonstrated. Most engineering design problems have mixed variables. The flexibility and efficiency of the GeneAS technique demonstrated here suggest its immediate application to other engineering design problems.

It is appropriate here to mention that due to the population approach, it becomes convenient to use GeneAS in solving multiobjective and multimodal design optimization problems (Deb and Goldberg, 1989; Goldberg and Richardson, 1987; Srinivas and Deb, 1989). In a multiobjective problem the objective is to find multiple Pareto-optimal solutions and in a multimodal problem the objective is to find multiple optimal solutions. Since GeneAS finds a number of solutions as an end-result, multiple optimal solutions can be simultaneously captured in the GeneAS's population. This may give designers more flexibility in solving multiobjective and multimodal design optimization problems.

Acknowledgment

This work is funded by the Department of Science and Technology, New Delhi under grant SR/SY/E-06/93. Authors also acknowledge the programming help provided by Ram Bhushan Agrawal.

References

Deb, K. and Agrawal, R. (1995). Simulated binary crossover for continuous search space. *Complex Systems, 9* 115–148.

Deb, K. (1995). *Optimization for engineering design: Algorithms and examples.* New Delhi: Prentice-Hall.

Deb, K. and Goldberg, D. E. (1989). An investigation of niche and species formation in genetic function optimization, In J. D. Schaffer (Ed.), *Proceedings of the Third International Conference on Genetic Algorithms* (pp. 42–50).

Eshelman, L. J. and Schaffer, J. D. (1993). Real-coded genetic algorithms and interval schemata. In D. Whitley (Ed.), *Foundations of Genetic Algorithms, II* (pp. 187–202).

Goldberg, D. E. (1989). *Genetic algorithms in search, optimization, and machine learning.* Reading: Addison-Wesley.

Goldberg, D. E. and Richardson, J. (1987). Genetic algorithms with sharing for multimodal function optimization. *Proceedings of the Second International Conference on Genetic Algorithms*, 41–49.

Kannan, B. K. and Kramer, S. N. (1995). An augmented Lagrange multiplier based method for mixed integer discrete continuous optimization and its applications to mechanical design. *Journal of Mechanical Design, 116*, 405–411.

Rao, S. S. (1984). *Optimization Theory and Applications.* New Delhi: Wiley Eastern.

Reklaitis, G. V., Ravindran, A., and Ragsdell, K. M. (1983). *Engineering Optimization—Methods and Applications.* New York: Wiley.

Sandgren, E. (1988). Nonlinear integer and discrete programming in mechanical design. *Proceedings of the ASME Design Technology Conference*, Kissimee, FL, 95–105.

Shigley, J. E. and Mischke, C. R. (1986). *Standard handbook of machine design.* New York: McGraw-Hill.

Siddall, J. N. (1982). *Optimal engineering design.* New York: Marcel Dekker.

Srinivas, N. and Deb, K. (1995). Multiobjective function optimization using nondominated sorting genetic algorithms, *Evolutionary Computation, 2*(3), 221–248.

Wright, A. (1991). Genetic Algorithms for Real Parameter Optimization, in *Foundations of Genetic Algorithms*, edited by G. J. E. Rawlins (Morgan Kaufmann, San Mateo).

A Combined Crossover Operator

The string in the GeneAS approach is a combination of binary strings representing discrete variables and real numbers representing continuous variables. Although discrete variables can be coded directly and discrete probability distribution can be used to create children values, we do not include the code for this operation here, for simplicity. The binary string is a collection of `1` and `0`, each of which is represented as an integer. Each variable k coded in a binary substring has a length `lchrom[`k`]` and there are `Max_bvar` number of variables represented in binary substrings. There are `Max_rvar` number of continuous variables. For continuous variables, the code is given only for creating children values in the whole real space. The code can be modified for creating children values in a given lower and upper bound.

```
/*===================================================================
Binary crossover and mutation operator where
two parent strings are used to create two children strings
===================================================================*/
binary_oper (lchr, parent1, parent2, child1, child2)
int lchr, *parent1, *parent2, *child1, *child2;
{
    int jcross, k;

    jcross = rnd(1, lchr-1);   /* Cross between 1 and l-1 */
    for(k = 0; k < jcross; k++) {
       child1[k] = bit_mutation(parent1[k]);
       child2[k] = bit_mutation(parent2[k]);
    }
    for (k=jcross; k < lchr; k++) {   /* swap bits */
       child1[k] = bit_mutation(parent2[k]);
       child2[k] = bit_mutation(parent1[k]);
    }
}

/*=====================================================================
Creates two children from parents parent1 and parent2, stores them in
addresses pointed by child1 and child2.
random() is the random number generator.
=====================================================================*/
real_oper(parent1,parent2,child1,child2)
float parent1,parent2,*child1,*child2;
{
    float difference, x_mean, beta, rand_var;

    rand_var = random();       /* a random number between 0 and 1 */
    beta = get_beta(rand_var); /* returns a beta using polynomial
                                  probability distribution */

    x_mean = 0.5 * (parent1 + parent2);
    difference = parent2 - parent1;

    /* perform crossover */
    c1 = x_mean - beta * 0.5 * difference;
    c2 = x_mean + beta * 0.5 * difference;

    /* perform mutation  */
    *child1 = real_mutation(c1);
    *child2 = real_mutation(c1);
}

/*==================================================================
Calculates beta value for given random number u (from 0 to 1)
using polynomial probability distribution (Equation~5)
==================================================================*/
float get_beta(u)
float u;
{
   float beta;

   if (1.0 - u < EPSILON ) u = 1.0 - EPSILON;
   if (u < 0.0) u = 0.0;
   if (u < 0.5) beta = pow(2.0*u,        (1.0/(n+1.0)));
   else         beta = pow((0.5/(1.0-u)),(1.0/(n+1.0)));
   return beta;
}

/*==================================================================
The combined crossover used in GeneAS. Each variable is crossed over
with a probability of 0.5.
flip is a function that returns a 1 with a probability p_xover
==================================================================*/
combined_crossover(first,second,childno1,childno2)
int first,second,childno1,childno2;
{
    int k, bvar, rvar;

    if flip(p_xover) {   /* whether to cross the parents */
       for (bvar = 1; bvar <= Max_bvar; bvar++)
          if flip(0.5)        /* whether to cross the binary variable */
             binary_oper(lchrom[bvar], oldpop[first].chrom[bvar],
                oldpop[second].chrom[bvar], &newpop[childno1].chrom[bvar],
                &newpop[childno2].chrom[bvar]);
          else                /* if not to cross the binary variable */
             for (k=0; k < lchrom[bvar]; k++) {
                newpop[childno1].chrom[bvar][k] = oldpop[first ].chrom[bvar][k];
                newpop[childno2].chrom[bvar][k] = oldpop[second].chrom[bvar][k];
             }
       for (rvar = 1; rvar <= Max_rvar; rvar++)
          if flip(0.5)        /* whether to cross the real variable */
             real_oper(oldpop[first].x[var], oldpop[second].x[var],
                &(newpop[childno1].x[var]), &(newpop[childno2].x[var]),
          else {              /* if not to cross the real variable */
             newpop[childno1].x[rvar] = oldpop[first ].x[rvar];
             newpop[childno2].x[rvar] = oldpop[second].x[rvar];
          }

    } else {             /* if not to cross the parents */

       for (bvar=1; bvar <= Max_bvar; bvar++)
          for (k=0; k < lchrom[bvar]; k++) {
            newpop[childno1].chrom[bvar][k] = oldpop[first ].chrom[bvar][k];
            newpop[childno2].chrom[bvar][k] = oldpop[second].chrom[bvar][k];
         }
       for (rvar=1; rvar <= Max_rvar; rvar++) {
          newpop[childno1].x[var] = oldpop[first ].x[var];
          newpop[childno2].x[var] = oldpop[second].x[var];
      }
}

/*=====================================================================
Bit mutation using a probability p_mutation
=====================================================================*/
bit_mutation(value)
int value;
{
    if flip(p_mutation)    /* if mutation is to be performed */
      if (value == 1) return (0);
      else return (1);
    else return(value);
}

/*=====================================================================
Mutation Using polynomial probability distribution. Picks up a random
site and generates a random number u between 0 to 1 and calculates
a mutated value. Max_mut is the maximum permissible perturbation
======================================================================*/
real_mutation(value)
float value;
{
    float delta, u;

    if flip(p_mutation) {
        u = random();         /* a random number in (0,1) */

        if (u < 0.5)
             delta = pow(2*u, (1.0/(n+1))) - 1.0;
        else delta = 1.0 - pow(2*(1.0-u), (1.0/(n+1)));
    }
    else delta = 0.0;         /* no mutation */
    return (value + delta * Max_mut);
}
```