

Introducción a la Computación Evolutiva

Dr. Carlos A. Coello Coello

Departamento de Computación

CINVESTAV-IPN

Av. IPN No. 2508

Col. San Pedro Zacatenco

México, D.F. 07300

email: ccoello@cs.cinvestav.mx

[http: //delta.cs.cinvestav.mx/~ccoello](http://delta.cs.cinvestav.mx/~ccoello)

Operadores Avanzados

- Diploides y Dominancia
- Inversión
- Micro-operadores

Diploides y Dominancia

- En AGs, usamos normalmente cromosomas haploides.
- En la naturaleza, sin embargo, los genotipos suelen ser diploides y contienen uno o más pares de cromosomas (a los que se les llama **homólogos**), cada uno de los cuales contiene información (redundante) para las mismas funciones.

Diploides y Dominancia

Ejemplo de un cromosoma diploide:

AbCDefGhIj
aBCdeFgHij

Si suponemos que los genes representados por letras mayúsculas son los dominantes y los representados mediante letras minúsculas son los recesivos, entonces el fenotipo correspondiente al cromosoma anterior sería:

ABCDeFGHIj

Diploides y Dominancia

El operador utilizado en el ejemplo anterior se denomina **dominancia**.

La idea es que un alelo (o un gen) dominante toma precedencia sobre uno recesivo (por ejemplo, los ojos negros son un alelo dominante y los azules uno recesivo).

A un nivel más abstracto, podemos concebir a la dominancia como un mapeo reductor del genotipo hacia el fenotipo.

Diploides y Dominancia

¿Por qué usa esta redundancia la naturaleza?

- Realmente no se sabe.
- Las teorías biológicas más aceptadas, sugieren que los diploides son una especie de “registro histórico” que protegen ciertos alelos (y combinaciones de ellos) del daño que puede causar la selección en un ambiente hostil.

Diploides y Dominancia

En AGs, los diploides suelen usarse para mantener soluciones múltiples (al mismo problema), las cuales pueden conservarse a pesar de que se exprese sólo una de ellas.

La idea es la misma que en Biología: preservar soluciones que fueron efectivas en el pasado, pero que eliminó el mecanismo de selección del AG.

Diploides y Dominancia

Los diploides parecen ser particularmente útiles en problemas en los que el ambiente cambia con el paso de las generaciones (por ejemplo, optimización de funciones dinámicas).

Diploides y Dominancia

El siguiente ejemplo se debe a Hillis (1990, 1992):

Padre 1 (diploide):

A:	10110101		011110011110010010101001
B:	00000101		001001110011110010101001

Padre 2 (diploide):

C:	00000111000000111110		000010101011
D:	11111111000010101101		010111011100

Hijo (diploide):

10110101001001110011110010101001
00000111000000111110010111011100

Diploides y Dominancia

El genotipo de un individuo en este ejemplo consiste de 15 pares de cromosomas (por claridad, sólo un par por cada padre se muestra en esta figura). Se elige aleatoriamente un punto de cruza para cada par, y se forma un gameto tomando los alelos antes del punto de cruza en el primer cromosoma, y los alelos después del punto de cruza en el segundo. Los 15 gametos de un padre se unen con los 15 gametos del otro padre para formar un nuevo individuo diploide (nuevamente por claridad sólo un gameto se muestra en esta figura).

Inversión

Holland (1975) propuso formas de adaptar la codificación de su algoritmo genético original, pues advirtió que el uso de cruza de un punto no trabajaría correctamente en algunos casos.

Inversión

El operador de inversión es un operador de reordenamiento inspirado en una operación que existe en genética. A diferencia de los AGs simples, en genética la función de un gene es frecuentemente independiente de su posición en el cromosoma (aunque frecuentemente los genes en un área local trabajan juntos en una red regulatoria), de manera que invertir parte del cromosoma retendrá mucha (o toda) la “semántica” del cromosoma original.

Inversión

Para usar inversión en los AGs, tenemos que encontrar la manera de hacer que la interpretación de un alelo sea la misma sin importar la posición que guarde en una cadena. Holland propuso que a cada alelo se le diera un índice que indicara su posición “real” que se usaría al evaluar su aptitud.

Inversión

Por ejemplo, la cadena 00010101 se codificaría como:

$$\{ (1,0) \ (2,0) \ (3,0) \ (4,1) \ (5,0) \ (6,1) \ (7,0) \ (8,1) \ }$$

en donde el primer elemento de cada uno de estos pares proporciona la posición “real” del alelo dado.

Inversión

La inversión funciona tomando dos puntos (aleatoriamente) a lo largo de la cadena, e invirtiendo la posición de los bits entre ellos. Por ejemplo, si usamos la cadena anterior, podríamos escoger los puntos 3 y 6 para realizar la inversión; el resultado sería:

$$\{ (1,0) \ (2,0) \ (6,1) \ (5,0) \ (4,1) \ (3,0) \ (7,0) \ (8,1) \ }$$

Inversión

Esta nueva cadena tiene la misma aptitud que la anterior porque los índices siguen siendo los mismos. Sin embargo, se han cambiado los enlaces alélicos. La idea de este operador es producir ordenamientos en los cuales los esquemas benéficos puedan sobrevivir con mayor facilidad.

Inversión

Por ejemplo, supongamos que en el ordenamiento original el esquema $00^{**}01^{**}$ es muy importante. Tras usar este operador, el esquema nuevo será 0010^{****} .

Si este nuevo esquema tiene una aptitud más alta, presumiblemente la cruce de un punto lo preservará y esta permutación tenderá a diseminarse con el paso de las generaciones.

Inversión

Debe advertirse que el uso de este operador introduce nuevos problemas cuando se combina con la cruce de un punto.

Supongamos, por ejemplo, que se cruzan las cadenas:

{ (1,0) (2,0) (6,1) (5,0) (4,1) (3,0) (7,0) (8,1) }

y

{ (5,1) (2,0) (3,1) (4,1) (1,1) (8,1) (6,0) (7,0) }

Inversión

Si el punto de cruce es la tercera posición, los hijos producidos serán:

{ (1, 0) (2, 0) (6, 1) (4, 1) (1, 1) (8, 1) (6, 0) (7, 0) }

y

{ (5, 1) (2, 0) (3, 1) (5, 0) (4, 1) (3, 0) (7, 0) (8, 1) }

Inversión

Estas nuevas cadenas tienen algo mal. La primera tiene 2 copias de los bits 1 y 6 y ninguna copia de los bits 3 y 5. La segunda tiene 2 copias de los bits 3 y 5 y ninguna copia de los bits 1 y 6.

¿Cómo podemos asegurarnos de que este problema no se presente?

Inversión

Holland propuso 2 soluciones posibles:

(1) Permitir que se realice la cruce sólo entre cromosomas que tengan los índices en el mismo orden. Esto funciona pero limitaría severamente la cruce.

Inversión

(2) Emplear un enfoque “amo/esclavo”:

Escoger un padre como el amo, y reordenar temporalmente al otro padre para que tenga el mismo ordenamiento que su amo. Usando este tipo de ordenamiento se producirán cadenas que no tendrán redundancia ni posiciones faltantes.

Inversión

La inversión se usó en algunos trabajos iniciales con AGs, pero nunca mejoró dramáticamente el desempeño de un AG. Más recientemente se ha usado con un éxito limitado en problemas de “ordenamiento” tales como el del viajero.

Inversión

Sin embargo, no hay todavía un veredicto final en torno a los beneficios que este operador produce y se necesitan más experimentos sistemáticos y estudios teóricos para determinarlos.

Inversión

Adicionalmente, cualquier beneficio que produzca este operador debe sopesarse con el espacio extra (para almacenar los índices de cada bit) y el tiempo extra (para reordenar un padre antes de efectuar la cruce) que se requiere.

Micro-Operadores

En la Naturaleza, muchos organismos tienen genotipos con múltiples cromosomas. Por ejemplo, los seres humanos tenemos 23 pares de cromosomas diploides. Para adoptar una estructura similar en los algoritmos genéticos necesitamos extender la representación a fin de permitir que un genotipo sea una lista de k pares de cadenas (asumiendo que son diploides).

Micro-Operadores

Pero, ¿para qué tomarnos estas molestias? Holland (1975) sugirió que los genotipos con múltiples cromosomas podrían ser útiles para extender el poder de los algoritmos genéticos cuando se usan en combinación con 2 operadores: la segregación y la traslocación.

Micro-Operadores

- **Segregación** : Para entender cómo funciona este operador, imaginemos el proceso de formación de gametos cuando tenemos más de un par cromosómico en el genotipo. La cruce se efectúa igual que como vimos antes, pero cuando formamos un gameto, tenemos que seleccionar aleatoriamente uno de los cromosomas haploides.

Micro-Operadores

A este proceso de selección aleatoria se le conoce como segregación. Este proceso rompe cualquier enlace que pueda existir entre los genes dentro de un cromosoma, y es útil cuando existen genes relativamente independientes en cromosomas diferentes.

Micro-Operadores

- **Traslocación** : Puede verse como un operador de cruza intercromosómico. Para implementar este operador en un algoritmo genético necesitamos asociar los alelos con su “nombre genético” (su posición), de manera que podamos identificar su significado cuando se cambien de posición de un cromosoma a otro mediante la traslocación.

Micro-Operadores

El uso de este operador permite mantener la organización de los cromosomas de manera que la segregación pueda explotar tal organización.

Micro-Operadores

- La segregación y la traslocación no se han usado mucho en la práctica, excepto por algunas aplicaciones de aprendizaje de máquina (e.g., Schaffer, 1984 y Smith, 1980).

Micro-Operadores

- **Duplicación y Borrado** : Son un par de operadores de bajo nivel sugeridos para la búsqueda artificial efectuada por el AG. La duplicación intracromosómica produce duplicados de un gen en particular y lo coloca junto con su progenitor en el cromosoma. El borrado actúa a la inversa, removiendo un gen duplicado del cromosoma.

Micro-Operadores

Holland (1975) ha sugerido que estos operadores pueden ser métodos efectivos de controlar adaptativamente el porcentaje de mutación. Si el porcentaje de mutación permanece constante y la duplicación ocasiona k copias de un gen en particular, la probabilidad de mutación efectiva para este gen se multiplica por k . Por otra parte, cuando ocurre el borrado de un gen, el porcentaje efectivo de mutación se decrementa.

Micro-Operadores

Cabe mencionar que una vez que ha ocurrido una mutación en uno de los nuevos genes, debemos decidir cuál de las alternativas será la que se exprese, en un proceso similar al que enfrentamos con los diploides.

Micro-Operadores

De hecho, podemos considerar la presencia de múltiples copias de un gen como una *dominancia intracromosómica*, en contraposición con la *dominancia intercromosómica* que resulta más tradicional en los diploides.

Micro-Operadores

Holland ha sugerido el uso de un esquema de arbitraje para hacer la elección necesaria entre las diferentes alternativas presentes, aunque no se han publicado estudios sobre este mecanismo hasta la fecha.

La duplicación puede permitir cosas más interesantes en un AG, como por ejemplo cadenas de longitud variable (AGs desordenados o mGAs).

Constraint-Handling Techniques used with EAs

Traditional mathematical programming techniques used to solve constrained optimization problems have several limitations when dealing with the general nonlinear programming problem:

$$\text{Min } f(\vec{x}) \quad (1)$$

subject to:

$$g_i(\vec{x}) \leq 0, \quad i = 1, \dots, m \quad (2)$$

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

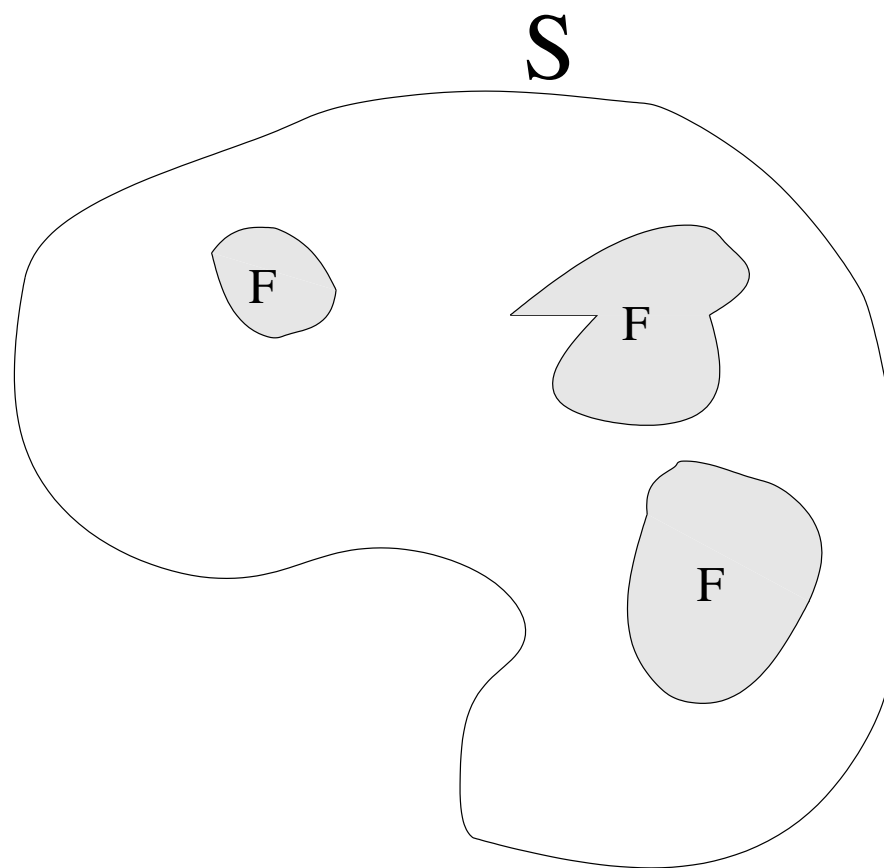
where \vec{x} is the vector of decision variables $\vec{x} = [x_1, x_2, \dots, x_n]^T$, m is the number of inequality constraints and p is the number of equality constraints (in both cases, constraints can be either linear or nonlinear).

Motivation

Evolutionary Algorithms (EAs) have been found successful in the solution of a wide variety of optimization problems. However, EAs are unconstrained search techniques. Thus, incorporating constraints into the fitness function of an EA is an open research area.

There is a considerable amount of research regarding mechanisms that allow EAs to deal with equality and inequality constraints; both type of constraints can be linear or nonlinear. Such work is precisely the scope of this tutorial.

Search Space



A Taxonomy of Constraint-Handling Approaches

- Penalty Functions
- Special representations and operators
- Separation of constraints and objectives
- Hybrid Methods

Penalty Functions



The most common approach in the EA community to handle constraints (particularly, inequality constraints) is to use penalties. Penalty functions were originally proposed by Richard Courant in the 1940s and were later expanded by Carroll and Fiacco & McCormick.

Penalty Functions

The idea of penalty functions is to transform a constrained optimization problem into an unconstrained one by adding (or subtracting) a certain value to/from the objective function based on the amount of constraint violation present in a certain solution.

Penalty Functions

In mathematical programming, two kinds of penalty functions are considered: exterior and interior. In the case of **exterior** methods, we start with an infeasible solution and from there we move towards the feasible region.

Penalty Functions

In the case of **interior methods**, the penalty term is chosen such that its value will be small at points away from the constraint boundaries and will tend to infinity as the constraint boundaries are approached. Then, if we start from a feasible point, the subsequent points generated will always lie within the feasible region since the constraint boundaries act as barriers during the optimization process.

Penalty Functions

EAs normally adopt external penalty functions of the form:

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

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

Penalty Functions

The most common form of G_i and L_j is:

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

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

where β and γ are normally 1 or 2.

Penalty Functions

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

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

where ϵ is the tolerance allowed (a very small value).

Types of Penalty Functions used with EAs

- Death Penalty
- Static Penalty
- Dynamic Penalty
- Adaptive Penalty
- Recent Approaches
 - Self-Adaptive Fitness Formulation
 - ASCHEA
 - Stochastic Ranking

Death Penalty

The rejection of infeasible individuals (also called “death penalty”) is probably the easiest way to handle constraints and it is also computationally efficient, because when a certain solution violates a constraint, it is rejected and generated again. Thus, no further calculations are necessary to estimate the degree of infeasibility of such a solution.

Criticism to Death Penalty

- Not advisable, except in the case of problems in which the feasible region is fairly large.
- No use of information from infeasible points.
- Search may “stagnate” in the presence of very small feasible regions.
- A variation that assigns a zero fitness to infeasible solutions may work better in practice.

Static Penalty

Under this category, we consider approaches in which the penalty factors do not depend on the current generation number in any way, and therefore, remain constant during the entire evolutionary process.

Static Penalty

An example of this sort of approach is the following:

- The approach proposed by Homaifar, Lai and Qi [1994] in which they define levels of violation of the constraints (and penalty factors associated to them):

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

where $R_{k,i}$ are the penalty coefficients used, m is total the number of constraints, $f(\vec{x})$ is the unpenalized objective function, and $k = 1, 2, \dots, l$, where l is the number of levels of violation defined by the user.

Criticism to Static Penalty

- It may not be a good idea to keep the same penalty factors along the entire evolutionary process.
- Penalty factors are, in general, problem-dependent.
- Approach is simple, although in some cases (e.g., in the approach by Homaifar, Lai and Qi [1994]), the user may need to set up a high number of penalty factors.

Dynamic Penalty

Within this category, we will consider any penalty function in which the current generation number is involved in the computation of the corresponding penalty factors (normally the penalty factors are defined in such a way that they increase over time—i.e., generations).

Dynamic Penalty

An example of this sort of approach is the following:

- The approach from Joines and Houck [1994] in which individuals are evaluated (at generation t) using:

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

where C , α and β are constants defined by the user (the authors used $C = 0,5$, $\alpha = 1$ or 2 , and $\beta = 1$ or 2).

Dynamic Penalty

$SVC(\beta, \vec{x})$ is defined as:

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

and

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

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

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

Criticism to Dynamic Penalty

- Some researchers have argued that dynamic penalties work better than static penalties.
- In fact, many EC researchers consider dynamic penalty as a good choice when dealing with an arbitrary constrained optimization problem.
- Note however, that it is difficult to derive good dynamic penalty functions in practice as it is difficult to produce good penalty factors for static functions.

Adaptive Penalty

Bean and Hadj-Alouane [1992,1997] developed a method that uses a penalty function which takes a feedback from the search process. Each individual is evaluated by the formula:

$$\text{fitness}(\vec{x}) = f(\vec{x}) + \lambda(t) \left[\sum_{i=1}^n g_i^2(\vec{x}) + \sum_{j=1}^p |h_j(\vec{x})| \right] \quad (13)$$

where $\lambda(t)$ is updated at every generation t .

Adaptive Penalty

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

$$\lambda(t + 1) = \begin{cases} (1/\beta_1) \cdot \lambda(t), & \text{if case \#1} \\ \beta_2 \cdot \lambda(t), & \text{if case \#2} \\ \lambda(t), & \text{otherwise,} \end{cases} \quad (14)$$

where cases #1 and #2 denote situations where the best individual in the last k generations was always (case #1) or was never (case #2) feasible, $\beta_1, \beta_2 > 1$, $\beta_1 > \beta_2$, and $\beta_1 \neq \beta_2$ (to avoid cycling).

Adaptive Penalty

In other words, the penalty component $\lambda(t + 1)$ for the generation $t + 1$ is decreased if all the best individuals in the last k generations were feasible or is increased if they were all infeasible. If there are some feasible and infeasible individuals tied as best in the population, then the penalty does not change.

Criticism to Adaptive Penalty

- Setting the parameters of this type of approach may be difficult (e.g., what generational gap (k) is appropriate?).
- This sort of approach regulates in a more “intelligent” way the penalty factors.
- An interesting aspect of this approach is that it tries to avoid having either an all-feasible or an all-infeasible population. More recent constraint-handling approaches pay a lot of attention to this issue.

Penalty Functions: Central Issues

The main problem with penalty functions is that the “ideal” penalty factor to be adopted in a penalty function cannot be known *a priori* for an arbitrary problem. If the penalty adopted is too high or too low, then there can be problems.

Penalty Functions: Central Issues

If the penalty is too high and the optimum lies at the boundary of the feasible region, the EA will be pushed inside the feasible region very quickly, and will not be able to move back towards the boundary with the infeasible region. On the other hand, if the penalty is too low, a lot of the search time will be spent exploring the infeasible region because the penalty will be negligible with respect to the objective function.

Recent Approaches

Three modern constraint-handling approaches that use penalty functions deserve special consideration, since they are highly competitive:

- Self-Adaptive Fitness Formulation
- ASCHEA
- Stochastic Ranking

Self-Adaptive Fitness Formulation

- Proposed by Farmani and Wright [2003].
- The approach uses an adaptive penalty that is applied in 3 steps:
 1. The sum of constraint violation is computed for each individual.
 2. The best and worst solutions are identified in the current population.
 3. A penalty is applied in two parts:
 - It is applied only if one or more feasible solutions have a better objective function value than the best solution found so far. The idea is to increase the fitness of the infeasible solutions.
 - Increase the fitness of the infeasible solutions as to favor those solutions which are nearly feasible and also have a good objective function value.

Self-Adaptive Fitness Formulation

- The penalty factor is defined in terms of both the best and the worst solutions.
- The authors use a genetic algorithm with binary representation (with Gray codes) and roulette-wheel selection.
- Good results, but not better than the state-of-the-art techniques (e.g., Stochastic Ranking).

Self-Adaptive Fitness Formulation

- The number of fitness function evaluations required by the approach is high (1,400,000).
- Its main selling point is that the approach does not require of any extra user-defined parameters. Also, the implementation seems relatively simple.
- Other self-adaptive penalty functions have been proposed more recently (see for example [Tessema & Yen, 2006]).

ASCHEA

The Adaptive Segregational Constraint Handling Evolutionary Algorithm (ASCHEA) was proposed by Hamida and Schoenauer [2000]. It uses an evolution strategy and it is based on three main components:

- An adaptive penalty function.
- A recombination guided by the constraints.
- A so-called “segregational” selection operator.

ASCHEA

The **adaptive penalty** adopted is the following:

$$fitness(\vec{x}) = \begin{cases} f(\vec{x}) & \text{if the solution is feasible} \\ f(\vec{x}) - penal(\vec{x}) & \text{otherwise} \end{cases} \quad (15)$$

where

$$penal(\vec{x}) = \alpha \sum_{j=1}^q g_j^+(\vec{x}) + \alpha \sum_{j=q+1}^m |h_j(\vec{x})| \quad (16)$$

where $g_j^+(\vec{x})$ is the positive part of $g_j(\vec{x})$ and α is the penalty factor adopted for all the constraints.

ASCHEA

The penalty factor is adapted based on the desired ratio of feasible solutions (with respect to the entire population) τ_{target} and the current ratio at generation t τ_t :

$$\begin{aligned} \text{if}(\tau_t > \tau_{target}) \quad & \alpha(t+1) = \alpha(t) / fact \\ \text{else} \quad & \alpha(t+1) = \alpha(t) * fact \end{aligned}$$

where $fact > 1$ and τ_{target} are user-defined parameters and

$$\alpha(0) = \left| \frac{\sum_{i=1}^n f_i(\vec{x})}{\sum_{i=1}^n V_i(\vec{x})} \right| * 1000 \quad (17)$$

where $V_i(\vec{x})$ is the sum of the constraint violation of individual i .

ASCHEA

The **Recombination guided by the constraints** combines an infeasible solution with a feasible one when there are few feasible solutions, based on τ_{target} . If $\tau_t > \tau_{target}$, then the recombination is performed in the traditional way (i.e., disregarding feasibility).

The **Segregational Selection** operator aims to define a ratio τ_{select} of feasible solutions such that they become part of the next generation. The remaining individuals are selected in the traditional way based on their penalized fitness. τ_{select} is another user-defined parameter.

ASCHEA

- In its most recent version [2002], it uses a penalty factor for each constraint, as to allow more accurate penalties.
- This version also uses niching to maintain diversity (this, however, adds more user-defined parameters).
- The approach requires a high number of fitness function evaluations (1,500,000).

Stochastic Ranking

This approach was proposed by Runarsson and Yao [2000], and it consists of a multimembered evolution strategy that uses a penalty function and a selection based on a ranking process. The idea of the approach is try to balance the influence of the objective function and the penalty function when assigning fitness to an individual. An interesting aspect of the approach is that it doesn't require the definition of a penalty factor. Instead, the approach requires a user-defined parameter called P_f , which determines the balance between the objective function and the penalty function.

Stochastic Ranking

```
Begin  
  For i=1 to N  
    For j=1 to P-1  
      u=random(0,1)  
      If ( $\phi(I_j) = \phi(I_{j+1}) = 0$ ) or ( $u < P_f$ )  
        If ( $f(I_j) > f(I_{j+1})$ )  
          swap( $I_j, I_{j+1}$ )  
        Else  
          If ( $\phi(I_j) > \phi(I_{j+1})$ )  
            swap( $I_j, I_{j+1}$ )  
        End For  
      If no swap is performed  
        break  
    End For  
  End For  
End
```

Stochastic Ranking

The population is sorted using an algorithm similar to bubble-sort (which sorts a list based on pairwise comparisons). Based on the value of P_f , the comparison of two adjacent individuals is performed based only on the objective function. The remainder of the comparisons take place based on the sum of constraint violation. Thus, P_f introduces the “stochastic” component to the ranking process, so that some solutions may get a good rank even if they are infeasible.

Stochastic Ranking

- The value of P_f certainly impacts the performance of the approach. The authors empirically found that $0,4 < P_f < 0,5$ produces the best results.
- The authors report the best results found so far for the benchmark adopted with only 350,000 fitness function evaluations.
- The approach is easy to implement.