Introducción a la Computación Evolutiva (Notas de Curso)

Dr. Carlos A. Coello Coello
CINVESTAV-IPN
Departamento de Computación
Av. Instituto Politécnico Nacional No. 2508
Col. San Pedro Zacatenco
México, D.F. 07300
ccoello@cs.cinvestav.mx

Mayo, 2020

Contenido

Lista de Tablas

Lista de Figuras

Introducción

Estas notas de curso son producto de la experiencia de haber impartido clases de computación evolutiva a nivel posgrado durante los últimos cinco años.

El material contenido en estas páginas ha sido producto de muchas horas de trabajo y esfuerzo y pretenden dar una visión general (si bien no completa) de lo que es la computación evolutiva y de sus alcances. El objetivo de este material, es servir como apoyo para la enseñanza de un curso introductorio a la computación evolutiva a nivel maestría o doctorado, aunque se requiere obviamente de material complementario (principalmente referencias bibliográficas). Por lo demás, estas notas pretenden ser auto-contenidas, de manera que se haga innecesario consultar otras fuentes bibliográficas (introductorias) adicionales.

La organización de estas notas es la siguiente: en el Capítulo 1 se proporcionan algunos conceptos básicos de análisis de algoritmos y de optimización con técnicas clásicas. Partiendo de las limitaciones de las técnicas clásicas, se plantea la motivación para el uso de heurísticas en problemas de búsqueda y optimización de alta complejidad.

En el Capítulo 2 se hace un rápido recorrido histórico de la computación evolutiva, yendo desde el Lamarckismo hasta las corrientes más modernas, como la dinámica evolutiva y la programación genética.

El Capítulo 3 da un panorama general de los tres grandes paradigmas en computación evolutiva (la programación evolutiva, las estrategias evolutivas y el algoritmo genético), describiendo el algoritmo básico de cada uno de ellos así como algunas de sus aplicaciones. En ese capítulo también se mencionan algunas de las críticas de que ha sido objeto la computación evolutiva (sobre todo, de parte de los investigadores de IA simbólica).

En el Capítulo 4 se proporciona una breve terminología biológica, junto con explicaciones del uso que se da a dichos términos en computación evolutiva.

El problema de la representación, que resulta vital en los algoritmos genéticos, es tratado en el Capítulo 5.

Las principales técnicas de selección utilizadas con los algoritmos genéticos se estudian en el Capítulo 6.

El Capítulo 7 se ocupa de las diferentes ténicas de cruza utilizadas en los algoritmos genéticos. Además de estudiar las cruzas para representación binaria, se analizan también las variantes existentes para representación real y de permutaciones.

El operador de mutación es estudiado en detalle en el Capítulo 8. Además de analizar su uso con representación real y de permutaciones, se le compara contra la cruza.

El Capítulo 9 se ocupa de revisar algunos de los estudios más importantes en torno a la forma de ajustar los parámetros de un algoritmo genético (sobre todo, porcentajes de cruza y mutación y tamaño de población). También se discute la auto-adaptación de parámetros y sus problemas principales, así como otros mecanismos de adaptación en línea tales como la lógica difusa y las representaciones adaptativas.

Los algoritmos genéticos funcionan como una técnica de búsqueda u optimización sin restricciones. De tal forma, se hace necesario acoplarle algún mecanismo para incorporar restricciones de todo tipo (lineales, no lineales, igualdad y desigualdad). En el Capítulo 10 se estudian algunas de las propuestas principales en torno al manejo de restricciones mediante funciones de penalización.

En el Capítulo 11 se revisan varios sistemas de software relacionados con computación evolutiva (tanto comerciales como de dominio público).

El Capítulo 12 comprende el estudio de algunos de los conceptos teóricos más importantes de la computación evolutiva, incluyendo el teorema de los esquemas, el paralelismo implícito, la decepción, el *No Free Lunch Theorem* y el análisis de convergencia usando cadenas de Markov.

En el Capítulo 13 se habla sobre algunos operadores avanzados, tales como los diploides, la segregación, traslocación, duplicación y borrado.

El Capítulo 14 habla sobre algunas aplicaciones exitosas de la computación evolutiva en el mundo real.

Con el creciente uso de las computadoras en problemas cada vez más complejos, se ha popularizado la adopción de esquemas de paralelismo y de cómputo distribuido en la computación evolutiva. Este tema es precisamente el que se discute en el Capítulo 15.

Finalmente, el Capítulo 16 discute brevemente algunos de los paradigmas emergentes dentro de la computación evolutiva y se concluye hablando brevemente sobre el futuro de la computación evolutiva.

Reconocimientos

La elaboración de estas notas no habría sido posible sin la ayuda de la Mat. Ma. Margarita Reyes Sierra, quien capturó mucho de este texto en formato LaTeX2ε y convirtió mis bosquejos a mano en nítidas imágenes en Xfig.

Agradezco al Dr. Arturo Hernández Aguirre sus incontables (y a veces interminables) discusiones en torno a diferentes temas, entre los que se incluye la computación evolutiva. Sus comentarios constructivos siempre me han resultado de enorme valía.

Manifiesto también mi agradecimiento por la generosa ayuda del Laboratorio Nacional de Informática Avanzada (LANIA), del CINVESTAV-IPN y de REDII-CONACyT, a quienes se debe en gran parte, haber podido preparar este material didáctico. Doy las gracias muy especialmente a la Dra. Cristina Loyo Varela, Directora General del LANIA, quien siempre me brindó su apoyo incondicional en todos mis proyectos.

Extiendo también un cordial reconocimiento a todos los estudiantes que he tenido a lo largo de mi vida. Ha sido a través de ellos que me he nutrido de ricas experiencias docentes que han forjado mi actual estilo de enseñanza. Gracias a todos por los buenos y, ¿por qué no?, también por los malos ratos que me hicieron pasar.

Capítulo 1

Conceptos Básicos

Antes de aventurarse a tomar un curso de computación evolutiva (o sobre heurísticas de cualquier otro tipo), es muy importante tener frescos en la memoria algunos conceptos básicos a fin de poder entender de manera clara la motivación para desarrollar y usar heurísticas.

De tal forma, iniciaremos con un repaso de algunos conceptos fundamentales de análisis de algoritmos y teoría de la computación.

1.1 Análisis de algoritmos

El análisis de algoritmos comprende 2 etapas: el análisis *a priori* y el análisis *a posteriori*. En el primero de ellos, obtenemos una función que acota el tiempo de cálculo del algoritmo. En el análisis *a posteriori* lo que hacemos es recolectar estadísticas acerca del consumo de tiempo y espacio del algoritmo mientras éste se ejecuta [130].

Es importante tener en mente que durante el análisis *a priori* de un algoritmo, se ignoran los detalles que sean dependientes de la arquitectura de una computadora o de un lenguaje de programación y se analiza el orden de magnitud de la frecuencia de ejecución de las instrucciones básicas del algoritmo.

Veamos un ejemplo simple de análisis *a priori*. Consideremos los 3 segmentos de código siguientes:

(1) $a = a + b$

(2) **for** i = 1 to n **do**
$a = a + b$

end for i

(3) **for** i = 1 to n **do**
for j = 1 to n **do**
$a = a + b$
end for j
end for i

La frecuencia de ejecución de la sentencia $a = a + b$ es:

- 1 para el segmento (1).
- n para el segmento (2).
- n^2 para el segmento (3).

Una de las notaciones más usadas para expresar la complejidad de un algoritmo es la denominada "O" (big-O, en inglés). Formalmente, la definimos de la siguiente manera:

Definición: $f(n) = O(g(n))$ si y sólo si existen dos constantes positivas c y n_0 tales que $|f(n)| \leq c|g(n)|$ para toda $n \geq n_0$.

Supongamos que nos interesa determinar el tiempo de procesamiento (o cómputo) de un algoritmo. Dicho tiempo lo denotaremos como $f(n)$. La variable n puede referirse al número de entradas o salidas, su suma o la magnitud de una de ellas. Puesto que $f(n)$ depende de la computadora en que se ejecute el algoritmo, un análisis *a priori* no será suficiente para determinarlo. Sin embargo, podemos usar un análisis *a priori* para determinar una $g(n)$ tal que $f(n) = O(g(n))$. Cuando decimos que un algoritmo tiene un tiempo de cómputo $O(g(n))$, lo que queremos decir es que al ejecutar el algoritmo en una computadora usando los mismos tipos de datos, pero valores incrementales de n, el tiempo resultante siempre será menor que algún tiempo constante $|g(n)|$.

Dados dos algoritmos que realicen la misma tarea sobre n entradas, debe resultar obvio que preferiremos al que requiera menos tiempo de ejecución. Algunos tiempos comunes de los algoritmos son los siguientes:

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$

Algunos algoritmos conocidos y sus complejidades correspondientes en esta notación son los siguientes:

1. Buscar un elemento en una lista no ordenada: $O(n)$
2. Buscar un elemento en una lista ordenada: $O(\log n)$
3. Quicksort: $O(n \log n)$
4. Calcular el determinante de una matriz: $O(n^3)$
5. Multiplicación matricial: $O(n^{2.81})$
6. Ordenamiento por el método de la burbuja (*Bubble Sort*): $O(n^2)$

Los problemas cuya complejidad está acotada por un polinomio (los primeros seis órdenes de magnitud de la jerarquía mostrada anteriormente) son los denominados problemas **P**. Más detalladamente, podemos decir que un problema pertenece a la clase si puede ser resuelto en tiempo polinomial en una computadora determinista.

El término **determinista** significa que sin importar lo que haga el algoritmo, sólo hay una cosa que puede hacer a continuación (es decir, el paso siguiente se determina por los pasos anteriores). Los ejemplos de algoritmos conocidos dados anteriormente, pertenecen todos a la clase **P**.

Un problema pertenece a la clase **NP** si puede ser resuelto en tiempo polinomial pero usando una computadora no determinista.

Cuando una computadora **no determinista** es confrontada con varias opciones, tiene el poder de "adivinar" la correcta (en caso de que ésta exista). Una computadora no determinista no hace nunca elecciones incorrectas que la hagan regresar a un estado previo.

Consideremos el siguiente ejemplo de un algoritmo no determinista en el cual queremos buscar un elemento x en un conjunto de elementos A[1:n], $n \geq 1$. Se quiere determinar un índice j tal que A[j]=x, o $j = 0$ si $x \notin A$.

```
j = elige [1:n]
  if A[j]=x then print(j)
  else print('0')
```

Este algoritmo imprime 0 sólo si $x \notin A$. Su complejidad es $O(1)$.

Obviamente, las computadoras no deterministas no existen en el mundo real. El no determinismo es una herramienta imaginaria que hace que los problemas difíciles parezcan triviales. Su mayor valía radica en el hecho de que existe forma

de convertir un algoritmo no determinista a uno determinista, aunque a un costo computacional que suele ser muy elevado.

Los siguientes puntos son de gran importancia al comparar la clase **P** contra la clase **NP**:

- La clase **P** contiene problemas que pueden resolverse rápidamente.
- La clase **NP** contiene problemas cuya solución puede verificarse rápidamente.
- En 1971 se planteó la pregunta: ¿Es **P=NP**? Desde entonces, sigue siendo una pregunta abierta para los teóricos.
- Se cree que **P** $\neq$ **NP**.

Existe otra clase de problemas, denominados **NP Completos** que resultan de gran interés en computación. Un problema pertenece a esta clase si todos los algoritmos requeridos para resolverlo requieren tiempo exponencial en el peor caso. En otras, palabras, estos problemas son sumamente difíciles de resolver. Un ejemplo típico de un problema **NP Completo** es el del viajero. Este problema consiste en encontrar una permutación que represente el recorrido de una serie de ciudades de tal forma que todas sean visitadas (una sola vez) minimizando la distancia total viajada. El mejor algoritmo que se conoce para resolver este problema es $O(n^2 2^n)$.

El tamaño del espacio de búsqueda del problema del viajero crece conforme a la expresión: $\frac{(n-1)!}{2}$. Algunos ejemplos de lo que esto significa, son los siguientes:

- Para $n = 10$, hay unas 181,000 soluciones posibles.
- Para $n = 20$, hay unas 10,000,000,000,000,000 soluciones posibles.
- Para $n = 50$, hay unas 100,000,000,000,000,000,000,000,000,000,000, 000,000,000,000,000,000,000,000,000,000,000,000 soluciones posibles.

Para tener idea de estas magnitudes, basta decir que sólo hay 1,000,000,000,000,000,000,000,000 litros de agua en el planeta.

1.2 Técnicas Clásicas de Optimización

Existen muchas técnicas clásicas para resolver problemas con ciertas características específicas (por ejemplo, funciones lineales con una o más variables).

Es importante saber al menos de la existencia de estas tcnicas, pues cuando el problema por resolverse se adecúa a ellas, no tiene ningún sentido usar heurísticas.

Por ejemplo, para optimización lineal, el método Simplex sigue siendo la opción más viable.

Para optimización no lineal, hay métodos directos (p. ej. la búsqueda aleatoria) y métodos no directos (p. ej., el método del gradiente conjugado) [180].

Uno de los problemas de las técnicas clásicas de optimización es que suelen requerir información que no siempre está disponible. Por ejemplo, métodos como el del gradiente conjugado requieren de la primera derivada de la función objetivo. Otros, como el de Newton, requieren además de la segunda derivada. Por tanto, si la función objetivo no es diferenciable (y en algunos problemas del mundo real, ni siquiera está disponible en forma explícita), estos métodos no pueden aplicarse.

A fin de ilustrar la forma en la que operan las técnicas clásicas de optimización, resolveremos paso por paso un ejemplo sencillo con el método del descenso empinado (*steepest descent*), el cual fue propuesto originalmente por Cauchy en 1847. La idea del método es comenzar de un cierto punto cualquier $\vec{X}_1$ y luego moverse a lo largo de las direcciones de descenso más empinado hasta encontrar el óptimo. El algoritmo es el siguiente:

Algoritmo del Descenso Empinado

1. Comenzar con un punto arbitrario $\vec{X}_1$. Hacer $i = 1$.

2. Calcular el gradiente ∇f_i.

3. Encontrar la dirección de búsqueda $\vec{S}_i$, definida como:

$$\vec{S}_i = -\nabla f_i = -\nabla f(\vec{x}_i) \tag{1.1}$$

4. Determinar la longitud óptima de incremento λ_i^* en la dirección $\vec{S}_i$, y hacer:

$$\vec{X}_{i+1} = \vec{X}_i + \lambda_i^* \vec{S}_i = \vec{X}_i - \lambda_i^* \nabla f_i \tag{1.2}$$

5. Checar la optimalidad de $\vec{X}_{i+1}$. Si el punto es óptimo, detenerse. De lo contrario, ir al paso 6.

6. Hacer $i = i + 1$. Ir al paso 2.

Ejemplo:

Min $f(x_1, x_2) = x_1 - x_2 + 2x_1^2 + 2x_1x_2 + x_2^2$

usando el punto inicial:

$$\vec{X}_1 = \left\{ \begin{array}{c} 0 \\ 0 \end{array} \right\}$$

Para iniciar la primera iteración, necesitamos calcular el gradiente de la función:

$$\nabla_f = \left\{ \begin{array}{c} \partial f/\partial x_1 \\ \partial f/\partial x_2 \end{array} \right\} = \left\{ \begin{array}{c} 1 + 4x_1 + 2x_2 \\ -1 + 2x_1 + 2x_2 \end{array} \right\}$$

$$\nabla f_1 = \nabla f(\vec{X}_1) = \left\{ \begin{array}{c} 1 \\ -1 \end{array} \right\}$$

Calcular la dirección de búsqueda:

$$\vec{S}_1 = -\nabla f_1 = \left\{ \begin{array}{c} -1 \\ 1 \end{array} \right\}$$

Determinar λ_1^*:

$$\vec{X}_2 = \vec{X}_1 + \lambda_1^* \vec{S}_1$$

$f(\vec{X}_1 + \lambda_1^* \vec{S}_1) = f(-\lambda_1, \lambda_1)$, lo cual se obtiene de:

$$\left\{ \begin{array}{c} 0 \\ 0 \end{array} \right\} + \lambda_1 \left\{ \begin{array}{c} -1 \\ 1 \end{array} \right\}$$

Y sabemos que:

$$f(-\lambda_1, \lambda_1) = -\lambda_1 - \lambda_1 + 2(-\lambda_1)^2 + 2(-\lambda_1)(\lambda_1) + \lambda_1^2 =$$

$$\lambda_1^2 - 2\lambda_1$$

Para hallar λ_1^*, debemos derivar esta expresión con respecto a λ_1 e igualar el resultado con cero:

$df/\lambda_1 = 2\lambda_1 - 2 = 0$

de donde obtenemos: $\lambda_1^* = 1$.

Ahora $i = 2$, y:

$$\vec{X}_2 = \left\{ \begin{array}{c} 0 \\ 0 \end{array} \right\} + 1 \left\{ \begin{array}{c} -1 \\ 1 \end{array} \right\} = \left\{ \begin{array}{c} -1 \\ 1 \end{array} \right\}$$

Chequemos ahora la optimalidad de esta solución:

$$\nabla f_2 = \nabla f(\vec{X}_2) = \left\{ \begin{array}{c} 1 - 4 + 2 \\ -1 - 2 + 2 \end{array} \right\} = \left\{ \begin{array}{c} -1 \\ -1 \end{array} \right\}$$

Como $\nabla f_2 \neq \left\{ \begin{array}{c} 0 \\ 0 \end{array} \right\}$ entonces tenemos que continuar con una iteración más:

$$\vec{S}_2 = \left\{ \begin{array}{c} 1 \\ 1 \end{array} \right\}$$

Calculemos ahora λ_2^* a partir de:

$$\left\{ \begin{array}{c} -1 \\ 1 \end{array} \right\} + \lambda_2 \left\{ \begin{array}{c} 1 \\ 1 \end{array} \right\}$$

de donde obtenemos:

$f(-1 + \lambda_2, 1 + \lambda_2) = -1 + \lambda_2 - (1 + \lambda_2) + 2(-1 + \lambda_2)^2 + 2(-1 + \lambda_2)(1 + \lambda_2) + (1 + \lambda_2)^2 = 5\lambda_2^2 - 2\lambda_2 - 1$

Para hallar λ_2^*:

$df/d\lambda_2 = 10\lambda_2 - 2 = 0$

de donde: $\lambda_2 = 1/5 = 0.2$

Por lo que:

$$\vec{X}_3 = \vec{X}_2 + \lambda_2^* \vec{S}_2$$

$$\vec{X}_3 = \begin{Bmatrix} -1 \\ 1 \end{Bmatrix} + \frac{1}{5} \begin{Bmatrix} 1 \\ 1 \end{Bmatrix} = \begin{Bmatrix} -1 + \frac{1}{5} \\ 1 + \frac{1}{5} \end{Bmatrix} = \begin{Bmatrix} -0.8 \\ 1.2 \end{Bmatrix}$$

Procedemos nuevamente a checar optimalidad de esta solución:

$$\nabla f_3 = \begin{Bmatrix} 1 + 4(-0.8) + 2(1.2) \\ -1 + 2(-0.8) + 2(1.2) \end{Bmatrix} = \begin{Bmatrix} 0.2 \\ -0.2 \end{Bmatrix}$$

Como $\nabla f_3 \neq \begin{Bmatrix} 0 \\ 0 \end{Bmatrix}$ tenemos que seguir iterando.

Método de Fletcher-Reeves (Gradiente Conjugado)

Propuesto originalmente por Hestenes & Stiefel en 1952, como un método para resolver sistemas de ecuaciones lineales derivadas de las condiciones estacionarias de una cuadrática. Puede verse como una variante del método del "descenso empinado", en la cual se usa el gradiente de una función para determinar la dirección más prometedora de búsqueda. El algoritmo es el siguiente:

1. Iniciar con un punto arbitrario $\vec{X}_1$.

2. Calcular la dirección de búsqueda:

$$\vec{S}_1 = -\nabla f(\vec{X}_1) = \nabla f_1$$

3. Obtener $\vec{X}_2$ usando:

$$\vec{X}_2 = \vec{X}_1 + \lambda_1^* \vec{S}_1$$

   donde λ_1^* es el paso óptimo de movimiento en la dirección $\vec{S}_1$. Hacer $i = 2$ y proceder al paso siguiente.

4. Obtener $\nabla f_i = \nabla f(\vec{X}_i)$, y hacer:

$$\vec{S}_i = -\nabla f_i + \frac{|\nabla f_i|^2}{|\nabla f_{i-1}|^2} \vec{S}_{i-1}$$

5. Calcular λ_i^* y obtener el nuevo punto:

$$X_{i+1} = X_i + \lambda_i^* \vec{S}_i$$

6. Evaluar optimalidad de X_{i+1}. Si X_{i+1} es el óptimo, detener el proceso. De lo contrario $i = i + 1$ y regresar al paso 4.

Ejemplo:

Min $f(x_1, x_2) = x_1 - x_2 + 2x_1^2 + 2x_1x_2 + x_2^2$

usando el punto inicial:

$$\vec{X}_1 = \left\{ \begin{array}{c} 0 \\ 0 \end{array} \right\}$$

Iteración 1

$$\nabla f = \left\{ \begin{array}{c} \partial f/\partial x_1 \\ \partial f/\partial x_2 \end{array} \right\} = \left\{ \begin{array}{c} 1 + 4x_1 2x_2 \\ -1 + 2x_1 + 2x_2 \end{array} \right\}$$

$$\nabla f_1 = \nabla f(\vec{X}_1) = \left\{ \begin{array}{c} 1 \\ -1 \end{array} \right\}$$

La dirección de búsqueda es: $\vec{S}_1 = -\nabla f_1 = \left\{ \begin{array}{c} -1 \\ 1 \end{array} \right\}$.

Para obtener λ_1^*, debemos minimizar $f(\vec{X}_1 + \lambda_1 \vec{S}_1)$ con respecto a λ_1. Por tanto:

$$\vec{X}_1 + \lambda_1 \vec{S}_1 = \left\{ \begin{array}{c} 0 \\ 0 \end{array} \right\} + \lambda_1 \left\{ \begin{array}{c} -1 \\ 1 \end{array} \right\}$$

$$f(\vec{X}_1 + \lambda_1 \vec{S}_1) = f(-\lambda_1, \lambda_1) =$$

$$-\lambda_1 - \lambda_1 + 2(-\lambda_1)^2 + 2(-\lambda_1)(\lambda_1) + \lambda_1^2 =$$

$$= -2\lambda_1 + 2\lambda_1^2 - 2\lambda_1^2 + \lambda_1^2 = \lambda_1^2 - 2\lambda_1$$

$$df/d\lambda_1 = 2\lambda_1 - 2 = 0$$

$$\lambda_1^* = 1$$

$$\vec{X}_2 = \vec{X}_1 + \lambda_1^* \vec{S}_1 = \left\{ \begin{array}{c} 0 \\ 0 \end{array} \right\} + 1 \left\{ \begin{array}{c} -1 \\ 1 \end{array} \right\} = \left\{ \begin{array}{c} -1 \\ 1 \end{array} \right\}$$

$$\vec{X}_2 = \left\{ \begin{array}{c} -1 \\ 1 \end{array} \right\}$$

Iteración 2

$$\nabla f_2 = \nabla f(\vec{X}_2) = \left\{ \begin{array}{c} 1 + 4(-2) + 2(1) \\ -1 + 2(-1) + 2(1) \end{array} \right\}$$

$$\nabla f_2 = \left\{ \begin{array}{c} -1 \\ -1 \end{array} \right\}$$

$$\vec{S}_2 = -\nabla f_2 + \frac{|\nabla f_2|^2}{|\nabla f_1|^2} \vec{S}_1$$

$$= \left\{ \begin{array}{c} 1 \\ 1 \end{array} \right\} + \left(\frac{2}{2}\right) \left\{ \begin{array}{c} -1 \\ 1 \end{array} \right\}$$

$$\vec{S}_2 = \left\{ \begin{array}{c} 0 \\ 2 \end{array} \right\}$$

$$\vec{X}_2 + \lambda_2 \vec{S}_2 = \left\{ \begin{array}{c} -1 \\ 1 \end{array} \right\} + \lambda_2 \left\{ \begin{array}{c} 0 \\ 2 \end{array} \right\}$$

$$f(\vec{X}_2 + \lambda_2 \vec{S}_2) = f(-1, 1 + 2\lambda_2) =$$

$$-1 - (1 + 2\lambda_2) + 2(-1)^2 + 2(-1)(1 + 2\lambda_2) + (1 + 2\lambda_2)^2$$

$$-1 - 1 - 2\lambda_2 + 2 - 2(1 + 2\lambda_2) + 1 + 4\lambda_2 + 4\lambda_2^2 =$$

$$= -2\lambda_2 - 2 - 4\lambda_2 + 1 + 4\lambda_2 + 4\lambda_2^2 = 4\lambda_2^2 - 2\lambda_2 - 1$$

$$df/d\lambda_2 = 8\lambda_2 - 2 = 0$$

$$\lambda_2^* = \frac{1}{4}$$

$$\vec{X}_3 = \vec{X}_2 + \lambda_2^* \vec{S}_2 = \left\{ \begin{array}{c} -1 \\ 1 \end{array} \right\} + \frac{1}{4} \left\{ \begin{array}{c} 0 \\ 2 \end{array} \right\}$$

$$\vec{X}_2 = \left\{ \begin{array}{c} -1 \\ 1.5 \end{array} \right\}$$

Iteración 3

$$\nabla f_3 = \nabla f(\vec{X}_3) = \left\{ \begin{array}{c} 1 + 4(-1) + 2(1.5) \\ -1 + 2(-1) + 2(1.5) \end{array} \right\}$$

$$\nabla f_3 = \left\{ \begin{array}{c} 0 \\ 0 \end{array} \right\}$$

$$\vec{S}_3 = -\nabla f_3 + \frac{|\nabla f_3|^2}{|\nabla f_2|^2} \vec{S}_2$$

$$= \left\{ \begin{array}{c} 0 \\ 0 \end{array} \right\} + \left(\frac{0}{2}\right) \left\{ \begin{array}{c} 0 \\ 2 \end{array} \right\}$$

$$\vec{S}_3 = \left\{ \begin{array}{c} 0 \\ 0 \end{array} \right\}$$

Al no poderse reducir más f, podemos decir que $\vec{X}_3$ es la solución óptima.

$$x_1^* = -1, x_2^* = 1.5$$

Existen también técnicas que construyen parcialmente una solución a un problema. Por ejemplo, la programación dinámica y el método de ramificación y búsqueda (*branch & bound*).

Cuando enfrentamos un cierto problema de optimización, si la función a optimizarse se encuentra definida en forma algebraica, es importante intentar resolverla primero con técnicas clásicas, antes de utilizar cualquier heurística.

1.3 Técnicas heurísticas

Hasta ahora, debió haber quedado claro que existen problemas que no pueden resolverse usando un algoritmo que requiera tiempo polinomial. De hecho, en muchas aplicaciones prácticas, no podemos siquiera decir si existe una solución eficiente.

Asimismo, hay muchos problemas para los cuales el mejor algoritmo que se conoce requiere tiempo exponencial.

Cuando enfrentamos espacios de búsqueda tan grandes como en el caso del problema del viajero, y que además los algoritmos más eficientes que existen para resolver el problema requieren tiempo exponencial, resulta obvio que las técnicas clásicas de búsqueda y optimización son insuficientes. Es entonces cuando recurrimos a las "heurísticas".

La palabra "heurística" se deriva del griego *heuriskein*, que significa "encontrar" o "descubrir".

El significado del término ha variado históricamente. Algunos han usado el término como un antónimo de "algorítmico". Por ejemplo, Newell et al. [173] dicen:

> A un proceso que puede resolver un cierto problema, pero que no ofrece ninguna garantía de lograrlo, se le denomina una 'heurística' para ese problema.

Las heurísticas fueron un área predominante en los orígenes de la Inteligencia Artificial.

Actualmente, el término suele usarse como un adjetivo, refiriéndose a cualquier técnica que mejore el desempeño en promedio de la solución de un problema, aunque no mejore necesariamente el desempeño en el peor caso [197].

Una definición más precisa y adecuada para los fines de este curso es la proporcionada por Reeves [186]:

> Una heurística es una técnica que busca soluciones buenas (es decir, casi óptimas) a un costo computacional razonable, aunque sin garantizar factibilidad u optimalidad de las mismas. En algunos casos, ni siquiera puede determinar qué tan cerca del óptimo se encuentra una solución factible en particular.

Algunos ejemplos de técnicas heurísticas son los siguientes:

- Búsqueda Tabú
- Recocido Simulado
- Escalando la Colina

La **búsqueda tabú** [102] es realmente una meta-heurística, porque es un procedimiento que debe acoplarse a otra técnica, ya que no funciona por sí sola. La búsqueda tabú usa una "memoria" para guiar la búsqueda, de tal forma que algunas soluciones examinadas recientemente son "memorizadas" y se vuelven tabú (prohibidas) al tomar decisiones acerca del siguiente punto de búsqueda. La búsqueda tabú es determinista, aunque se le pueden agregar elementos probabilísticos.

El **recocido simulado** [142] está basado en el enfriamiento de los cristales. El algoritmo requiere de una temperatura inicial, una final y una función de variación de la temperatura. Dicha función de variación es crucial para el buen desempeño del algoritmo y su definición es, por tanto, sumamente importante. Este es un algoritmo probabilístico de búsqueda local.

La técnica **escalando la colina** se aplica a un punto a la vez (es decir, es una técnica local). A partir de un punto, se generan varios estados posibles y

Figura 1.1: Dos ejemplos de conjuntos convexos.

se selecciona el mejor de ellos. El algoritmo no tiene retroceso ni lleva ningún tipo de registro histórico (aunque éstos y otros aditamentos son susceptibles de ser incorporados). El algoritmo puede quedar atrapado fácilmente en óptimos locales. Asimismo, el algoritmo es determinista.

1.4 Conceptos Importantes

Comenzaremos por definir **convexidad**. El conjunto $\mathcal{F}$ es convexo si para toda $\vec{a}^1$, $\bar{a}^2 \in F$ y para toda $\theta \in [0, 1]$:

$$\vec{f}(\theta\vec{a}^1 + (1-\theta)\vec{a}^2) \leq \theta\vec{f}(\vec{a}^1) + (1-\theta)\vec{f}(\vec{a}^2) \tag{1.3}$$

En otras palabras, $\mathcal{F}$ es convexo si para dos puntos cualquiera $\vec{a}^1$ y $\vec{a}^2$ en el conjunto, el segmento rectilíneo que une estos puntos está también dentro del conjunto. De tal forma, los conjuntos mostrados en la figura 1.1 son convexos y los mostrados en la figura 1.2 no lo son.

El objetivo principal de cualquier técnica de optimización es encontrar el óptimo (o los óptimos) globales de cualquier problema. En matemáticas, existe un área que se ocupa de desarrollar los formalismos que nos permitan garantizar la convergencia de un método hacia el óptimo global de un problema. Apropiadamente, se denomina **optimización global** [131].

Desgraciadamente, sólo en algunos casos limitados, puede garantizarse convergencia hacia el óptimo global. Por ejemplo, para problemas con espacios de búsqueda convexos, las condiciones de Kuhn-Tucker son necesarias y suficientes

Figura 1.2: Dos ejemplos de conjuntos no convexos.

para garantizar optimalidad global de un punto.

En problemas de optimización no lineal, las condiciones de Kuhn-Tucker no son suficientes para garantizar optimalidad global. De hecho, todas las técnicas usadas para optimización no lineal pueden localizar cuando mucho óptimos locales, pero no puede garantizarse convergencia al óptimo global a menos que se usen técnicas exhaustivas o que se consideren tiempos infinitos de convergencia.

Existen muchos tipos de problemas de optimización, pero los que nos interesan más para los fines de este curso, son de los de **optimización numérica**, que pueden definirse de la siguiente manera:

Minimizar $f(\vec{x})$

sujeta a:

$g_i(\vec{x}) \leq 0, \quad i = 1, \ldots, p$
$h_j(\vec{x}) = 0, \quad j = 1, \ldots, n$

donde: $\vec{x}$ son las **variables de decisión** del problema, $g_i(\vec{x})$ son las **restricciones de desigualdad**, y $h_j(\vec{x})$ son las **restricciones de igualdad**. Asimismo,

Figura 1.3: Representación gráfica de la zona factible (denotada con $\mathcal{F}$) de un problema. Advierta que en este caso la zona factible es disjunta.

$f(\vec{x})$ es la función objetivo del problema (la que queremos optimizar).

A las restricciones de igualdad y desigualdad expresadas algebraicamente, se les denomina "restricciones explícitas". En algunos problemas, existen también "restricciones implícitas", relacionadas sobre todo con las características del problema. Por ejemplo, si decimos:

$$10 \leq x_1 \leq 20$$

estamos definiendo que el rango de una variable de decisión debe estar contenido dentro de un cierto intervalo. De tal forma, estamos "restringiendo" el tipo de soluciones que se considerarán como válidas.

Todas las soluciones a un problema que satisfagan las restricciones existentes (de cualquier tipo), se consideran ubicadas dentro de la **zona factible**. De tal forma, podemos decir que el espacio de búsqueda de un problema se divide en la regin (o zona) factible y la no factible. La figura 1.3 ilustra la diferencia entre la zona factible y no factible de un problema

Para concluir este primer capítulo, mencionaremos que existe una clase especial de problemas que también serán de interés para este curso, en los cuales

las variables de decisión son discretas y las soluciones suelen presentarse en la forma de permutaciones. A estos problemas se les denomina de **optimización combinatoria** (p. ej. el problema del viajero).

1.5 Problemas propuestos

1. Para los siguientes pares de funciones, determine el valor entero más pequeño de $n \geq 0$ para el cual la primera función se haga mayor o igual a la segunda función (la de la derecha). En cada caso, muestre la forma en que obtuvo el valor de n:

   a) n^2, $10n$

   b) 2^n, $2n^3$

   c) $n^2/\log_2 n$, $n(\log_2 n)^2$

   d) $n^3/2$, $n^{2.81}$

2. Busque en un libro de métodos numéricos información sobre alguna de las siguientes técnicas para resolver ecuaciones trascendentes de una sola variable (elija sólo uno): Bisección (o Bolzano), Newton-Raphson, Regla Falsa o Secante. Se recomienda consultar el siguiente libro:

   Richard L. Burden y J. Douglas Faires, *Análisis Numérico*, 6a. Edición, International Thompson Editores, 1998.

   a) Usando como base el seudo-código del método, impleméntelo en C/C++ (la función a resolverse puede proporcionarse directamente dentro del código del programa). Incluya el código fuente en su tarea.

   b) Dos escaleras se cruzan en un pasillo, tal y como se muestra en la figura 1.4. Cada escalera está colocada de la base de una pared a algún punto de la pared opuesta. Las escaleras se cruzan a una altura H arriba del pavimento. Dado que las longitudes de las escaleras son $x_1 = 20$ m y $x_2 = 30$ m, y que $H = 8$ m, encontrar A, que es el ancho del pasillo, usando el programa escrito en el inciso anterior (una solución real es suficiente). Imprima las

Figura 1.4: Escaleras entrecruzadas del problema 2.b.

iteraciones efectuadas por su programa al resolver este problema.

Bonificación. El problema del inciso b) tiene más de una solución real. Encuentre al menos una solución real más (distinta de la reportada en el inciso anterior). Adicionalmente, encuentre una solución real que sea válida pero que no tenga sentido reportar dada la interpretación física del problema. Discuta brevemente acerca de esto.

3. Muestre las 2 primeras iteraciones del método de ascenso empinado al minimizar $f = 2x_1^2 + x_2^2$ usando el punto inicial $(1, 2)$.

4. Lea el capítulo 1 del siguiente libro (las fotocopias se proporcionarán en clase) para resolver las preguntas de esta sección:

   Zbigniew Michalewicz & David B. Fogel, *How to Solve It: Modern Heuristics*, Springer, Berlin, 2000.

1. Mencione al menos 2 razones por las cuales un problema del mundo real puede no ser resuelto fácilmente. NO USE ninguna de las razones enumeradas en la página 11 del capítulo proporcionado.

2. Supongamos que la optimización de una función f requiere de 10 variables de decisión x_i $(i = 1, \ldots, 10)$, cada una de las cuales tiene el rango: $-50 \leq x_i \leq 50$.

a) Si x_i puede tomar sólo valores enteros, ¿cuál es el tamaño del espacio de búsqueda de este problema?

b) Si x_i puede tomar valores reales y usaremos una precisión de ocho lugares decimales, ¿cuál es el tamaño del espacio de búsqueda del problema?

3. Defina las restricciones rígidas (*hard constraints*) y las restricciones flexibles (*soft constraints*) con sus propias palabras. Compárelas entre sí.

Capítulo 2

Un vistazo histórico a la computación evolutiva

2.1 El origen de las ideas

Contrario a lo que muchos creen, las ideas evolucionistas que bien hiciera en popularizar Charles Darwin en 1858 no se originaron con él, sino que estuvieron presentes en las mentes de una serie de científicos y pensadores en general que no se sentían satisfechos con la (entonces popular) idea de que había un Dios originador de todas las especies del planeta (las cuales habían sido creadas de forma separada) y de que las especies estaban jerarquizadas por Dios de tal manera que el hombre ocupaba el rango superior, al lado del Creador.

Georges Louis Leclerc (Conde de Buffon) fue tal vez el primero en especular (100 años antes que Darwin) en su *Historie Naturelle* (una impresionante enciclopedia de 44 tomos que describía todo lo que se sabía en aquel entonces sobre la naturaleza), que las especies se originaron entre sí. Leclerc no sólo notó las similitudes entre el hombre y los simios, sino que incluso habla sobre un posible ancestro común entre estas dos especies. Leclerc creía en los cambios orgánicos, pero no describió un mecanismo coherente que fuera responsable de efectuarlos, sino que especuló que era el ambiente el que influía directamente sobre los organismos.

Figura 2.1: Georges Louis Leclerc (Conde de Buffon).

Figura 2.2: Jean Baptiste Pierre Antoine de Monet (Caballero de Lamarck).

2.2 Lamarckismo

A partir de 1801, el zoólogo francés Jean Baptiste Pierre Antoine de Monet (Caballero de Lamarck) comienza a publicar detalles de su propia teoría evolutiva. Conocedor del trabajo de Leclerc (quien era además su mentor), Lamarck enfatizó la importancia de la naturaleza en los cambios de las especies.

A diferencia de Leclerc, Lamarck sí explicó un mecanismo responsable de los cambios en las especies, al cual ahora se le conoce como "Lamarckismo". A pesar de que el término Lamarckismo se usa hoy en día en sentido peyorativo para referirse a la teoría de que las características adquiridas por un individuo son hereditarias[1], la verdad es que sus ideas fueron más complejas. Lamarck creía que los organismos no son alterados de forma pasiva por su ambiente, tal y como afirmaba Étienne Geoffroy Saint-Hilaire en su *Philosophie Anatomique*, sino que más bien un cambio en el ambiente produce cambios en las necesidades de los organismos, lo que hace que, en consecuencia, éstos cambien su comportamiento. Estos cambios de comportamiento conducen al mayor uso (o al desuso) de ciertos órganos o estructuras corporales de un individuo, los cuales harán que dichos órganos o estructuras crezcan (ante un mayor uso) o se reduzcan (ante el menor uso) con el paso de las generaciones. Además, Lamarck creía que estos cambios eran hereditarios, lo que implicaba que los organismos se van adaptando gradualmente a su ambiente.

Es interesante hacer notar que aunque el mecanismo evolutivo propuesto por Lamarck difiere notablemente del propuesto varios años después por Darwin, los resultados a los que ambas teorías conducen son los mismos: las especies sufren cambios adaptativos debido a la influencia del ambiente a lo largo de periodos de tiempo considerables. De hecho, llama la atención que en su libro *Philosophie Zoologique* [150], Lamarck utiliza muchas de las mismas evidencias que después adoptara Darwin para enunciar su famosa teoría.

Sin embargo, las ideas de Lamarck no fueron muy populares en su época, y sólo sirvieron para desacreditarlo con sus contemporáneos, incluyendo a su mentor Leclerc. Curiosamente, varias de las ideas de Lamarck fueron re-descubiertas (de forma independiente) por Erasmus Darwin (el abuelo de Charles Darwin) en su libro *Zoonomia*, que data de 1794. Tal vez a ello se deba que Charles Darwin haya sido uno de los pocos naturalistas en defender las ideas de Lamarck, a pesar de que éstas se oponían a su propia teoría evolutiva.

[1]El ejemplo clásico usado por Lamarck era el del crecimiento del cuello y las patas frontales de las jirafas, ante la necesidad de alcanzar las hojas más altas de los árboles.

Figura 2.3: August Weismann.

Hasta el re-descubrimiento de las leyes de Gregor Mendel (enunciadas originalmente en 1865) a principios del siglo XX, nadie entendía de forma precisa los mecanismos de la herencia, y la teoría de Lamarck (con diversas variantes) gozó de gran popularidad entre varios científicos destacados del siglo XIX como el alemán Ernst Haeckel y el norteamericano Edward Drinker Cope, considerándose como una alterantiva viable al mecanismo de selección natural que Charles Darwin propusiera en su libro *El Origen de las Especies* [59].

2.3 La teoría del germoplasma

El científico alemán August Weismann formuló en el siglo XIX una teoría denominada del *germoplasma*, según la cual el cuerpo se divide en células germinales (o germoplasma) que pueden transmitir información hereditaria y en células somáticas (o somatoplasma), que no pueden hacerlo [224]. Sus ideas entraban en contraposición con el Lamarckismo, por lo que decidió efectuar una serie de experimentos en los cuales cortó la cola a un grupo de ratas durante 22 generaciones (1,592 ratones en total), haciendo ver que esto no afectaba la longitud de la cola de las nuevas generaciones de ratas. Con ello, Weismann demostró que la teoría Lamarckista de la herencia de características a través de generaciones estaba equivocada, y condujo (incidentalmente) al re-descrubrimiento del trabajo de Mendel sobre las leyes de la herencia.

Para Weismann, la selección natural era el único mecanismo que podía cam-

Figura 2.4: Alfred Russell Wallace.

biar al germoplasma (hoy conocido como *genotipo*[2]), y creía que tanto el germoplasma como el ambiente podían influenciar al somatoplasma (hoy conocido como *fenotipo*[3]).

Con el advenimiento de la genética en los 1930s, las ideas de Lamarck fueron nuevamente desacreditadas y se consideran hoy en día como obsoletas, aunque algunos investigadores las han incorporado con éxito en diversas aplicaciones de la computación evolutiva, como veremos en capítulos posteriores.

2.4 Russell y Darwin

El naturalista inglés Alfred Russell Wallace era un auto-didacta que se interesó en el origen de las especies a mediados de los 1850s, publicando varios artículos al respecto que pasaron totalmente desapercibidos. En 1858, de manera súbita intuyó la teoría de la selección natural sin saber que Darwin se le había adelantado, e iróricamente le escribió a éste para pedirle que le ayudara a publicar sus ideas. El resultado de esta curiosa cooperación fue la presentación de un trabajo conjunto a la *Linnean Society* de Londres, el 1 de julio de 1858, el cual fue publicado posteriormente (ese mismo año) en el *Journal of the Linnean Society*.

Tanto Charles Darwin como Alfred Russell Wallace estuvieron fuertemente influenciados por el trabajo del economista Thomas Robert Malthus[4] y del geólogo

[2]En genética moderna, el término *genotipo* se usa para denotar la composición genética de un organismo.

[3]En genética moderna, se denomina *fenotipo* a los rasgos específicos de un individuo.

[4]Malthus afirmaba que mientras las especies se reproducen de forma geométrica, los recursos

Figura 2.5: Charles Robert Darwin.

Charles Lyell[5].

Tras percatarse del trabajo de Wallace, Darwin decidió interrumpir la elaboración de un libro sobre la selección natural que inició en 1856, y mejor se enfocó a escribir otro sobre la evolución. Su libro, titulado *El origen de las especies*, se publicó en 1859 [59] con gran éxito (el tiraje inicial de 1,250 ejemplares se agotó en sólo 24 horas).

Darwin se percató de que una especie que no sufriera cambios se volvería incompatible con su ambiente, ya que éste tiende a cambiar con el tiempo. Asimismo, las similitudes entre hijos y padres observada en la naturaleza, le sugirieron a Darwin que ciertas características de las especies eran hereditarias, y que de generación a generación ocurrían cambios cuya principal motivación era hacer a los nuevos individuos más aptos para sobrevivir.

2.5 La teoría de la combinación

Una teoría popular sobre la herencia en la época de Darwin era la teoría de la "combinación", según la cual las características hereditarias de los padres se mezclaban o combinaban de alguna forma en sus hijos. La mayor debilidad de esta teoría es que no explicaba la ocurrencia de cambios repentinos en una especie, los cuales no habían sido heredados de ningún ancestro.

naturales sólo crecen de forma aritmética, por lo que ocurren periódicamente crisis para estabilizar las poblaciones de las especies con respecto a los recursos que requieren.

[5]Lyell se convenció que la forma de las montañas y demás formaciones geológicas se debía a causas naturales y no divinas.

Figura 2.6: Johann Gregor Mendel.

La explicación que esta teoría daba a este fenómeno era que ciertas características, al combinarse, se diluían con el paso de las generaciones, lo cual contradecía la teoría de la selección natural de Darwin, pues según la teoría de la combinación, los cambios adaptativos no podrían ser preservados. El ingeniero británico Fleming Jenkins se contaba entre los más fervientes defensores de la teoría de la combinación, y para defenderse de él, Darwin hubo de elaborar su propia teoría de la herencia, denominada de la "pangénesis".

2.6 Las leyes de la herencia de Mendel

El monje austriaco Johann Gregor Mendel realizó una serie de experimentos con guisantes durante buena parte de su vida, estudiando las características básicas de esta planta. Mediante un cuidadoso análisis de las diferentes características manifestadas por las plantas de guisantes, Mendel descubrió tres leyes básicas que gobernaban el paso de una característica de un miembro de una especie a otro. La primera ley (llamada de Segregación) establecía que los miembros de cada par de alelos de un gene se separan cuando se producen los gametos durante la meiosis. La segunda ley (llamada de la Independencia) establecía que los pares de alelos se independizan (o separan entre sí durante la formación de gametos[6]. La tercera ley (llamada de la Uniformidad) establecía que cada característica heredada se determina mediante dos factores provenientes de ambos padres, lo cual decide si un

[6]Hoy sabemos que esta ley sólo es válida para los genes localizados en cromosomas diferentes.

cierto gene[7] es dominante o recesivo. De tal manera, la teoría de la combinación de la que hablamos anteriormente, quedaba desechada de acuerdo a los resultados producidos en los experimentos de Mendel.

Aunque Mendel descubrió las leyes de la herencia, realmente no llegó a entender el mecanismo detrás de ella, y los genes, así como el mecanismo de transmisión de los mismos de generación en generación no fue descubierto sino hasta varios años después. Asimismo, su trabajo permaneció largamente ignorado por la comunidad científica, debido a que los dio a conocer originalmente en dos conferencias dictadas ante la Sociedad de Ciencias de Brünn, el 8 de febrero y el 8 de marzo de 1865. El manuscrito original fue publicado en las memorias de dicha Sociedad en 1866, en alemán [157], y no se tradujeron al inglés sino hasta 1901 [158].

2.7 La teoría de la pangénesis

Esta teoría esbozada por Darwin (explicada en el libro *On the Variation of Animals and Plants under Domestication* [58]) esgrime que cada órgano del cuerpo produce pequeñas partículas hereditarias llamadas "gémulas" o "pangenes". Según Darwin, estas gémulas eran transportadas a través de la sangre y se recolectaban en los gametos[8] durante su formación. Esto significaba que, según esta teoría, las características de los padres se transmitían directamente a la sangre de sus hijos.

Esta teoría estaba claramente influenciada por el Lamarckismo, y curiosamente fue desacreditada por un primo de Darwin llamado Francis Galton, quien efectuó transfusiones sanguíneas de un conejo negro a uno blanco, mostrando que los descendientes de este último permanecían blancos, sin importar a cuántas generaciones se les inyectara sangre del conejo negro.

Cabe destacar que Darwin desconocía el trabajo contemporáneo de Gregor Mendel, que le habría dado el elemento faltante en su teoría de la evolución: una explicación clara del mecanismo de la herencia.

Figura 2.7: Francis Galton.

Figura 2.8: Hugo De Vries.

2.8 La teoría de la mutación

El botánico danés Hugo De Vries creyó haber descubierto una nueva especie de planta al encontrar (alrededor del año 1900) una flor roja entre una gran cantidad de flores amarillas. Según De Vries, esto se debía a una mutación abrupta e infrecuente de las flores amarillas. Accidentalmente, DeVries re-descubrió nuevamente las leyes de la herencia que enunciara varios años atrás Gregor Mendel, y puso de moda la teoría de las "mutaciones espontáneas" [223], expandiendo con ella la teoría Darwiniana de la evolución.

Según De Vries, los cambios en las especies no eran graduales y adaptativos como afirmaba Darwin, sino más bien abruptos y aleatorios (es decir, al azar). Varios años más tarde se descubrió que las flores rojas que motivaron esta teoría no eran más que una variedad más de las mismas flores amarillas de que estaban rodeadas, y no una nueva especie como De Vries creía. De hecho, se ha logrado demostrar que las mutaciones son siempre dañinas y que no producen nuevas especies, aunque algunos aspectos de la teoría de De Vries han servido para complementar la teoría evolutiva de Darwin.

2.9 La teoría cromosómica de la herencia

En 1903, Walter Sutton (entonces un estudiante de posgrado en la Universidad de Columbia), leyó el trabajo de Mendel y el de DeVries, y determinó correctamente (y sin la ayuda de experimentos genéticos) que los cromosomas en el núcleo de las células eran el lugar donde se almacenaban las características hereditarias. También afirmó que el comportamiento de los cromosomas durante la división de las células sexuales era la base para las leyes de la herencia de Mendel. Un poco después indicó que los cromosomas contenían genes, y que los genes de un mismo cromosoma estaban ligados y, por tanto, se heredaban juntos. A esto se le llamó la "teoría cromosómica de la herencia".

Thomas Hunt Morgan confirmaría experimentalmente las hipótesis de Sutton algunos años más tarde, con lo que Sutton pasó a ser uno de los pioneros más importantes de la genética moderna.

[7] El término "gene" fue acuñado en una época posterior, pero los factores hereditarios o unidades de la herencia a los que se refirió Mendel son precisamente los genes.

[8] Se denominan gametos a las células que llevan información genética de sus padres con el propósito de llevar a cabo una reproducción sexual. En los animales, a los gametos masculinos se les llama espermas y a los femeninos se les llama óvulos.

Figura 2.9: Thomas Hunt Morgan.

2.10 Neo-Darwinismo

La teoría evolutiva propuesta originalmente por Charles Darwin en combinación con el seleccionismo de August Weismann y la genética de Gregor Mendel, se conoce hoy en día como el paradigma Neo-Darwiniano.

El Neo-Darwinismo establece que la historia de la vasta mayoría de la vida en nuestro planeta puede ser explicada a través de un puñado de procesos estadísticos que actúan sobre y dentro de las poblaciones y especies [124]: la reproducción, la mutación, la competencia y la selección.

La reproducción es una propiedad obvia de todas las formas de vida de nuestro planeta, pues de no contar con un mecanismo de este tipo, la vida misma no tendría forma de producirse.

En cualquier sistema que se reproduce a sí mismo continuamente y que está en constante equilibrio, la mutación está garantizada [80]. El contar con una cantidad finita de espacio para albergar la vida en la Tierra garantiza la existencia de la competencia. La selección se vuelve la consecuencia natural del exceso de organismos que han llenado el espacio de recursos disponibles. La evolución es, por lo tanto, el resultado de estos procesos estocásticos (es decir, probabilísticos) fundamentales que interactúan entre sí en las poblaciones, generación tras generación.

Figura 2.10: Alan Mathison Turing.

2.11 Inspiración biológica

La evolución natural fue vista como un proceso de aprendizaje desde los 1930s. W. D. Cannon, por ejemplo, plantea en su libro *The Wisdom of the Body* [38] que el proceso evolutivo es algo similar al aprendizaje por ensayo y error que suele manifestarse en los humanos.

El célebre matemático inglés Alan Mathison Turing reconoció también una conexión "obvia" entre la evolución y el aprendizaje de máquina en su artículo (considerado hoy clásico en Inteligencia Artificial) titulado "*Computing Machinery and Intelligence*" [219].

2.12 Alexander Fraser

A fines de los 1950s y principios de los 1960s, el biólogo Alexander S. Fraser [89, 90, 91] publicó una serie de trabajos sobre la evolución de sistemas biológicos en una computadora digital, dando la inspiración para lo que se convertiría más tarde en el algoritmo genético [127].

El trabajo de Fraser incluye, entre otras cosas, el uso de una representación binaria, de un operador de cruza probabilístico, de una población de padres que generaban una nueva población de hijos tras recombinarse y el empleo de un mecanismo de selección. Además, Fraser estudió el efecto de la epístasis[9], la

[9]En biología se dice que un gen es "espistático" cuando su presencia suprime el efecto de un gen que se encuentra en otra posición.

Figura 2.11: George E. P. Box.

segregación[10], los porcentajes de cruza y varios otros mecanismos biológicos que hoy son de sumo interés para la comunidad de computación evolutiva. Su trabajo de más de 10 años en este tema se resume en un libro titulado *Computer Models in Genetics* [93]. De tal forma, puede decirse que el trabajo de Fraser anticipó la propuesta del algoritmo genético simple de Holland y la de la estrategia evolutiva de dos miembros de Schwefel [81]. Fraser además llegó a utilizar el término "aprendizaje" para referirse al proceso evolutivo efectuado en sus simulaciones, y anticipó el operador de inversión, la definición de una función de aptitud y el análisis estadístico de la convergencia del proceso de selección [92]. Más interesante aún, resulta el hecho de que el trabajo de Fraser no fue el único efectuado en su época, sino que más bien los modelos computacionales de la genética poblacional se volvieron bastante populares durante fines de los 1950s y principios de los 1960s, al grado de que dio pie a reseñas sobre el tema, como las de J. L. Crosby [57] y B. R. Levin [153]

2.13 EVOP

Aproximadamente en la misma época en que iniciara su trabajo Fraser, el experto en estadística inglés George E. P. Box propuso un enfoque evolutivo para la optimización de la producción industrial [26]. Su técnica, denominada EVOP (*Evolutionary Operation*) consistía en efectuar pequeños cambios a un conjunto de parámetros de producción, monitoreando ciertos datos estadísticos de los procesos para guiar la búsqueda. Box [26] llegó a establecer claramente la analogía

[10]Cuando se forman los gametos y tenemos más de un par de cromosomas en el genotipo, entonces, para fines de la recombinación sexual, es necesario elegir sólo uno de los cromosomas existentes. A este proceso se le denomina *segregación*.

entre estos cambios y las mutaciones que ocurren en la naturaleza, e hizo ver también que el proceso de ajuste de parámetros que efectuaba con técnicas estadísticas era similar al proceso de selección natural.

EVOP funcionaba mediante un proceso iterativo, pero requería de intervención humana en cada etapa, si bien Box [26] reconoció que la técnica podía automatizarse. Como Fogel indica en su libro sobre historia de la computación evolutiva [81], aunque en forma limitada, EVOP sigue en uso hoy en día en la industria química[11].

2.14 La evolución de programas de Friedberg

R. M. Friedberg [94] es considerado como uno de los primeros investigadores en intentar evolucionar programas de computadora. Aunque Friedberg no usa explícitamente la palabra "evolución" en su trabajo, resulta claro que ese es el enfoque que adoptó en su artículo original y en una versión posterior, publicada en 1959 [95]. De hecho, en un artículo posterior, sus co-autores modelan la evolución como un proceso de optimización [70].

El trabajo de Friedberg consistió en generar un conjunto de instrucciones en lenguaje máquina que pudiesen efectuar ciertos cálculos sencillos (por ejemplo, sumar dos números) [81]. Fogel [81] considera que Friedberg fue el primero en enunciar de manera informal los conceptos de *paralelismo implícito*[12] y *esquemas*[13], que popularizara Holland en los 1970s [127].

Friedberg [94] utilizó un algoritmo de asignación de crédito para dividir la influencia de diferentes instrucciones individuales en un programa. Este procedimiento fue comparado con una búsqueda puramente aleatoria, y en algunos casos fue superado por ésta. Tras ciertas modificaciones al procedimiento, Friedberg fue capaz de superar a una búsqueda totalmente aleatoria, pero no pudo resolver satisfactoriamente el problema de "estancamiento" (*stagnation*, en inglés) de la población que se le presentó y por ello fue cruelmente criticado por investigadores de la talla de Marvin Minsky, quien en un artículo de 1961 [165] indicó que el tra-

[11] Ver por ejemplo la página `http://www.multisimplex.com/evop.htm`

[12] El paralelismo implícito que demostrara Holland para los algoritmos genéticos se refiere al hecho de que mientras el algoritmo calcula las aptitudes de los individuos de una población, estima de forma implícita las aptitudes promedio de un número mucho más alto de cadenas cromosómicas a través del cálculo de las aptitudes promedio observadas en los "bloques constructores" que se detectan en la población.

[13] Un esquema es un patrón de valores de los genes en un cromosoma.

bajo de Friedberg era "una falla total". Minsky atribuyó el fracaso del método de Friedberg a lo que él denominó el "fenómeno de mesa" [166], según el cual el estancamiento de la población se debía al hecho de que sólo una instrucción del programa era modificada a la vez, y eso no permitía explorar una porción significativa del espacio de búsqueda. Aunque estas observaciones no son del todo precisas [81], el problema del estancamiento siguió siendo el principal inconveniente del procedimiento de Friedberg, aunque Fogel [81] considera que su trabajo precedió el uso de los sistemas clasificadores que popularizara varios años después John Holland [127].

Dunham et al. [70, 72, 71] continuaron el trabajo de Friedberg, y tuvieron éxito con algunos problemas de aprendizaje de mayor grado de dificultad que los intentados por éste.

2.15 Friedman y la robótica evolutiva

George J. Friedman [97] fue tal vez el primero en proponer una aplicación de técnicas evolutivas a la robótica: su tesis de maestría propuso evolucionar una serie de circuitos de control similares a lo que hoy conocemos como redes neuronales, usando lo que él denominaba "retroalimentación selectiva", en un proceso análogo a la selección natural. Muchos consideran a este trabajo, como el origen mismo de la denominada "robótica evolutiva", que es una disciplina en la que se intentan aplicar técnicas evolutivas a diferentes aspectos de la robótica (planeación de movimientos, control, navegación, etc.). Desgraciadamente, las ideas de Friedman nunca se llevaron a la práctica, pero aparentemente fueron re-descubiertas por algunos investigadores varios años después [81].

Los circuitos de control que utilizara Friedman en su trabajo modelaban a las neuronas humanas, y eran capaces de ser excitadas o inhibidas. Además, era posible agrupar estos circuitos simples (o neuronas) para formar circuitos más complejos. Lo interesante es que Friedman propuso un mecanismo para construir, probar y evaluar estos circuitos de forma automática, utilizando mutaciones aleatorias y un proceso de selección. Este es probablemente el primer trabajo en torno a lo que hoy se denomina "hardware evolutivo".

Friedman [96] también especuló que la simulación del proceso de reproducción sexual (o cruza) y el de mutación nos conduciría al diseño de "máquinas pensantes", remarcando específicamente que podrían diseñarse programas para jugar ajedrez con este método.

2.16 Vida artificial

Nils Aall Barricelli [16] desarrolló las que tal vez fueron las primeras simulaciones de un sistema evolutivo en una computadora, entre 1953 y 1956. Para ello utilizó la computadora IAS[14] (desarrollada por el legendario matemático John von Neumann) del Instituto de Estudios Avanzados de Princeton, ubicado en Nueva Jersey, en los Estados Unidos de Norteamérica. Sus experimentos siguieron los lineamientos de una disciplina popularizada a mediados de los 1980s bajo el nombre de "Vida Artificial" [151].

La investigación original de Barricelli se publicó en italiano, pero debido a algunos errores de traducción y a algunas adiciones realizadas posteriormente, se republicó en 1957 [17]. El principal énfasis de su investigación consistía en determinar las condiciones que los genes deben cumplir para poder dar pie al desarrollo de formas de vida más avanzadas. Sus conclusiones fueron que los genes deben satisfacer lo siguiente [17]: (1) una cierta capacidad para reproducirse, (2) una cierta capacidad para cambiar a formas alternas (a través de mutaciones) y, (3) una necesidad de simbiosis[15] (por ejemplo, a través de vida parásita) con otros genes u organismos.

Barricelli desarrolló uno de los primeros juegos co-evolutivos[16] (llamado *Tac Tix*), en el cual se hacen competir entre sí a dos estrategias para jugar. Asimismo, en un trabajo posterior [18, 19], Barricelli explicó la función de la recombinación sexual en forma muy similar a la noción de bloques constructores que enunciara Holland en los 1970s [127].

En otros trabajos realizados con J. Reed y R. Toombs [185], Barricelli demostró que la recombinación aceleraba la evolución de los caracteres fenotípicos que no eran poligénicos (es decir, que no se basaban en la interacción de múltiples genes).

Otra de las contribuciones de Barricelli [18] fue el haber reconocido la naturaleza Markoviana de sus simulaciones, en un preludio al modelo matemático por excelencia utilizado en tiempos modernos para analizar la convergencia de los algoritmos genéticos.

---

[14] Las siglas *IAS* significan *Institute for Advanced Studies*, que es el lugar donde la computadora se desarrolló.

[15] La "simbiosis" es la asociación de dos tipos diferentes de organismos en la cual cada uno se beneficia del otro y pueden incluso ser esenciales entre sí para su existencia.

[16] Se denomina "co-evolución" al proceso evolutivo en el cual la aptitud de un individuo se determina mediante la evaluación parcial de otros.

Figura 2.12: Lawrence J. Fogel.

2.17 La optimización evolutiva de Bremermann

Hans Joachim Bremermann [27] fue tal vez el primero en ver a la evolución como un proceso de optimización, además de realizar una de las primeras simulaciones de la evolución usando cadenas binarias que se procesaban por medio de reproducción (sexual o asexual), selección y mutación, en lo que sería otro claro predecesor del algoritmo genético [127].

Bremermann [28, 30] utilizó una técnica evolutiva para problemas de optimización con restricciones lineales. La idea principal de su propuesta era usar un individuo factible el cual se modificaba a través de un operador de mutación hacia un conjunto de direcciones posibles de movimiento. Al extender esta técnica a problemas más complejos, utilizó además operadores de recombinación especializados [31].

Bremermann fue uno de los primeros en utilizar el concepto de "población" en la simulación de procesos evolutivos, además de intuir la importancia de la coevolución [27] (es decir, el uso de dos poblaciones que evolucionan en paralelo y cuyas aptitudes están relacionadas entre sí) y visualizar el potencial de las técnicas evolutivas para entrenar redes neuronales [29].

2.18 La programación evolutiva

Lawrence J. Fogel et al. [87] concibieron el uso de la evolución simulada en la solución de problemas (sobre todo de predicción). Su técnica, denominada "Programación Evolutiva" [84] consistía básicamente en hacer evolucionar autómatas

de estados finitos, los cuales eran expuestos a una serie de símbolos de entrada (el ambiente), y se esperaba que, eventualmente, serían capaces de predecir las secuencias futuras de símbolos que recibirían. Fogel utilizó una función de "pago" que indicaba qué tan bueno era un cierto autómata para predecir un símbolo, y usó un operador modelado en la mutación para efectuar cambios en las transiciones y en los estados de los autómatas que tenderían a hacerlos más aptos para predecir secuencias de símbolos.

Esta técnica no consideraba el uso de un operador de recombinación sexual porque, como veremos en un capítulo posterior, pretendía modelar el proceso evolutivo al nivel de las especies y no al nivel de los individuos.

La programación evolutiva se aplicó originalmente a problemas de predicción, control automático, identificación de sistemas y teoría de juegos, entre otros [83, 86, 35].

Donald W. Dearholt y algunos otros investigadores, experimentaron con programación evolutiva en la Universidad de Nuevo México en los 1970s, de forma totalmente independiente a Fogel [85, 81, 193, 54].

Probablemente la programación evolutiva fue la primera técnica basada en la evolución en aplicarse a problemas de predicción, además de ser la primera en usar codificaciones de longitud variable (el número de estados de los autómatas podía variar tras efectuarse una mutación), además de constituir uno de los primeros intentos por simular la co-evolución.

2.19 Las estrategias evolutivas

Como estudiantes de posgrado en la Universidad Técnica de Berlín, en Alemania, Peter Bienert, Ingo Rechenberg y Hans-Paul Schwefel estudiaban la mecánica de los fluídos en 1963 con un particular énfasis en la experimentación en un túnel de viento. Los problemas que les interesaban eran de carácter hidrodinámico, y consistían en la optimización de la forma de un tubo curvo, la minimización del arrastre de una placa de unión y la optimización estructural de una boquilla intermitente de dos fases. Debido a la imposibilidad de describir y resolver estos problemas de optimización analíticamente o usando métodos tradicionales como el del gradiente [180], Ingo Rechenberg decidió desarrollar un método de ajustes discretos aleatorios inspirado en el mecanismo de mutación que ocurre en la naturaleza. Los resultados iniciales de esta técnica, a la que denominaron "estrategia evolutiva", fueron presentados al Instituto de Hidrodinámica de su universidad el 12 de junio de 1964 [81].

Figura 2.13: Hans-Paul Schwefel.

En los dos primeros casos (el tubo y la placa), Rechenberg procedió a efectuar cambios aleatorios en ciertas posiciones de las juntas y en el tercer problema procedió a intercambiar, agregar o quitar segmentos de boquilla. Sabiendo que en la naturaleza las mutaciones pequeñas ocurren con mayor frecuencia que las grandes, Rechenberg decidió efectuar estos cambios en base a una distribución binomial con una varianza prefijada. El mecanismo básico de estos primeros experimentos era crear una mutación, ajustar las juntas o los segmentos de boquilla de acuerdo a ella, llevar a cabo el análisis correspondiente y determinar qué tan buena era la solución. Si ésta era mejor que su predecesora, entonces pasaba a ser utilizada como base para el siguiente experimento. De tal forma, no se requería información alguna acerca de la cantidad de mejoras o deterioros que se efectuaban.

Esta técnica tan simple dio lugar a resultados inesperadamente buenos para los tres problemas en cuestión, y Peter Bienert [23] construyó un robot que podía efectuar de forma automáticamente el proceso de optimización usando este método. Simultáneamente, Hans-Paul Schwefel se dio a la tarea de implementar esta técnica en una computadora Z23 [202].

Aunque los primeros fundamentos teóricos de las estrategias evolutivas de dos miembros (su versión más simple) se esbozaron en la tesis doctoral de Ingo Rechenberg la cual se publicó como libro en 1973 [184], no fue sino hasta que el libro que Schwefel escribiera a fines de los 1970s [203] se tradujo al inglés [204] que la técnica atrajo la atención de los investigadores fuera del mundo germanoparlante.

Figura 2.14: John H. Holland.

2.20 Los algoritmos genéticos

John H. Holland se interesó en los 1960s en estudiar los procesos lógicos involucrados en la adaptación. Inspirado por los estudios realizados en aquella época con autómatas celulares [36] y redes neuronales [207], Holland se percató de que el uso de reglas simples podría generar comportamientos flexibles, y visualizó la posibilidad de estudiar la evolución de comportamientos en un sistema complejo.

Holland advirtió que un estudio de la adaptación debía reconocer que [126, 125]: (a) la adaptación ocurre en un ambiente, (b) la adaptación es un proceso poblacional, (c) los comportamientos individuales pueden representarse mediante programas, (d) pueden generarse nuevos comportamientos mediante variaciones aleatorias de los programas, y (e) las salidas de dos programas normalmente están relacionadas si sus estructuras están relacionadas.

De tal forma, Holland vio el proceso de adaptación en términos de un formalismo en el que los programas de una población interactúan y mejoran en base a un cierto ambiente que determina lo apropiado de su comportamiento. El combinar variaciones aleatorias con un proceso de selección (en función de qué tan apropiado fuese el comportamiento de un programa dado), debía entonces conducir a un sistema adaptativo general.

Este sistema fue desarrollado hacia mediados de los 1960s, y se dio a conocer en el libro que Holland publicase en 1975, donde lo denominó "plan reproductivo genético" [127], aunque después se popularizó bajo el nombre (más corto y conveniente) de "algoritmo genético".

Aunque concebido originalmente en el contexto del aprendizaje de máquina, el algoritmo genético se ha utilizado mucho en optimización, siendo una técnica sumamente popular en la actualidad.

Figura 2.15: Michael Conrad.

Figura 2.16: Howard H. Pattee.

2.21 Ecosistemas artificiales

Michael Conrad y Howard H. Pattee [51] se cuentan entre los primeros en simular un ecosistema artificial jerárquico en el que un conjunto de organismos unicelulares estaban sujetos a una estricta ley de conservación de la materia que les inducía a competir para sobrevivir. Los organismos simulados fueron capaces de efectuar cooperación mutua y de llevar a cabo estrategias biológicas tales como la recombinación genética y la modificación de la expresión de su genoma[17].

En esta implementación, los organismos realmente consistían de subrutinas genéticas, por lo que el fenotipo de cada individuo se determinaba mediante la forma en que estas rutinas era usadas por los organismos. Algunos de los puntos que enfatizó el trabajo de Conrad y Pattee fueron los siguientes: (a) el compor-

[17]Se denomina genoma a la colección completa de genes (y por tanto cromosomas) que posee un organismo.

tamiento que caracteriza a los procesos de sucesión ecológica deben emerger potencialmente, (b) los procesos de la búsqueda evolutiva deben corresponder con su contraparte biológica, y (c) la simulación debe ser lo más simple posible a fin de permitir el estudio de características fundamentales de los ecosistemas así como las condiciones mínimas necesarias para que ocurra la evolución natural. Sus esfuerzos en esta área se extendieron hasta los 1980s [48, 49, 53, 189, 52].

Michael Conrad [47] propuso también en los 1970s un "modelo de circuitos de aprendizaje evolutivo" en el cual especuló sobre la posibilidad de que el cerebro use el mismo tipo de mecanismos que usa la evolución para aprender. Su técnica fue uno de los primeros intentos por utilizar algoritmos evolutivos para entrenar redes neuronales. Conrad también sugirió [50] el uso de la evolución para lidiar con problemas como el reconocimiento de patrones en que los enfoques algorítmicos de alto nivel (como los sistemas expertos) no han proporcionado resultados satisfactorios.

2.22 Programación genética

Aunque los primeros intentos por evolucionar programas se remontan a los 1950s y 1960s [95, 83], no fue sino hasta los 1980s que se obtuvieron resultados satisfactorios.

Hicklin [121] y Fujiki [98] usaron expresiones-S en LISP para representar programas cuyo objetivo era resolver problemas de teoría de juegos. Hicklin [121] discutió la combinación de segmentos de programas mediante el copiado de subárboles de un individuo a otro, aunque sus árboles se limitaban a expresar sentencias condicionales. Adicionalmente, planteó el uso de mutación para introducir nuevos árboles en la población.

Nichael Lynn Cramer [56] y posteriormente John R. Koza [144] propusieron (de forma independiente) el uso de una representación de árbol en la que se implementó un operador de cruza para intercambiar sub-árboles entre los diferentes programas de una población generada al azar (con ciertas restricciones impuestas por la sintaxis del lenguaje de programación utilizado).

La diferencia fundamental del trabajo de Cramer [56] y el de Koza [144] es que el primero usó una función de aptitud interactiva (es decir, el usuario debía asignar a mano el valor de aptitud a cada árbol de la población), mientras el segundo logró automatizarla.

La propuesta de Koza [144] se denomina *programación genética* y fue implementada originalmente usando expresiones-S en LISP, aunque hoy en día existen

Figura 2.17: John R. Koza.

implementaciones en muchos otros lenguajes de programación. Su técnica es casi independiente del dominio y ha sido utilizada en un sinnúmero de aplicaciones de entre las que destacan la compresión de imágenes, el diseño de circuitos, el reconocimiento de patrones y la planeación de movimientos de robots, entre otras [145].

Más recientemente, Koza [146] extendió su técnica mediante la incorporación de "funciones definidas automáticamente", las cuales pueden ser reusadas a manera de subrutinas e incrementan notablemente el poder de la programación genética para generar programas automáticamente.

2.23 Dinámica evolutiva

Thomas S. Ray [183] desarrolló a principios de los 1990s un simulador muy original en el que se evolucionaban programas en lenguaje ensamblador, los cuales competían por ciclos de CPU de una computadora, a la vez que intentaban reproducirse (o sea, copiarse a sí mismos) en la memoria de dicha computadora. En este simulador, denominado *Tierra*, se partía de un programa único con la capacidad de auto-replicarse, al que se denominaba "ancestro". En base a este programa, se generaban "criaturas" nuevas (segmentos de código), las cuales a su vez se podían dividir para dar nacimiento a más criaturas. Al nacer, una criatura era colocada en una cola de espera aguardando a tener el turno correspondiente para que sus instrucciones fueran ejecutadas. Si dichas instrucciones producían un error al ejecutarse, la criatura se movería más arriba en la cola de espera, de acuerdo al nivel de errores acumulados en su periodo de vida. Las criaturas que se encontraran

Figura 2.18: Thomas S. Ray.

en la parte superior de la cola de espera eran destruidas (es decir, morían). Para introducir diversidad en las criaturas, Ray propuso dos formas de mutación: (1) a ciertos intervalos, algunos bits en memoria eran modificados aleatoriamente, y (2) durante el proceso de auto-replicación de una criatura, algunos de sus bits se cambiaban aleatoriamente. Otra fuente adicional de diversidad provenía del hecho de que los programas no eran determinísticos, sino que cada instrucción tenía asociada una probabilidad de ser ejecutada.

Uno de los fenómenos observados por Ray durante las simulaciones de *Tierra* fue el surgimiento de criaturas "parásitas", las cuales no podían reproducirse al carecer de un programa huésped que las contuviera, y que la dinámica evolutiva misma del sistema tendía a eliminar con el paso del tiempo.

Tierra es uno de los pocos intentos por simular un ecosistema con el propósito expreso de observar los comportamientos que emergen de la dinámica evolutiva del mismo.

2.24 Problemas propuestos

1. Muy relacionado con el Lamarckismo se encuentra un mecanismo que diera a conocer originalmente James Mark Baldwin en 1902 [15] y al cual denominó "selección orgánica".

   Hoy en día, a este mecanismo se le conoce como "efecto Baldwin" y es bastante popular en computación evolutiva. Explique en qué consiste el "efecto Baldwin" desde la perspectiva biológica y de qué forma se relaciona

Figura 2.19: James Mark Baldwin.

con el Lamarckismo.

2. El Neo-Darwinismo se compone básicamente de los siguientes fenómenos y propuestas [212]:

   - Herencia
   - Mutación
   - Mezcla aleatoria de cromosomas paternos y maternos
   - Recombinación
   - Selección y evolución natural
   - Aislamiento
   - Desvío genético
   - La barrera de Weismann es inviolable

   Investigue los argumentos del Neo-Darwinismo respecto de cada uno de estos puntos y elabore un ensayo donde los discuta y critique.

3. Investigue en qué consiste el *equilibro acentuado* [73, 124] (*punctuated equilibrium* en inglés) y escriba un ensayo en el que indique si considera que se opone a los preceptos del Neo-Darwinismo o no. Fundamente bien sus argumentos.

4. Defina lo que se entiende en biología por **desvío genético** (*genetic drift*, en inglés), indicando cómo puede contrarrestarse. Asegúrese de explicar el significado de cualquier término que utilice en su definición.

   Se le recomienda consultar:

   Paton, Raymond C. "Principles of Genetics", en Thomas Bäck, David B. Fogel & Zbigniew Michalewicz (editores), *Handbook of Evolutionary Computation*, pp. A2.2:1–A2.2:9, Institute of Physics Publishing and Oxford University Press, 1997.

Capítulo 3

Principales Paradigmas

3.1 Introducción

El término **computación evolutiva** o **algoritmos evolutivos**, realmente engloba una serie de técnicas inspiradas biológicamente (en los principios de la teoría Neo-Darwiniana de la evolución natural). En términos generales, para simular el proceso evolutivo en una computadora se requiere:

- Codificar las estructuras que se replicarán.
- Operaciones que afecten a los "individuos".
- Una función de aptitud.
- Un mecanismo de selección.

Aunque hoy en día es cada vez más difícil distinguir las diferencias entre los distintos tipos de algoritmos evolutivos existentes, por razones sobre todo históricas, suele hablarse de tres paradigmas principales:

- Programación Evolutiva
- Estrategias Evolutivas
- Algoritmos Genéticos

Cada uno de estos paradigmas se originó de manera independiente y con motivaciones muy distintas. Aunque este curso se concentrará principalmente en el tercero (los algoritmos genéticos), revisaremos rápidamente, y de manera muy general, los otros dos paradigmas en este capítulo.

Figura 3.1: Portada de la edición reciente (1999) del libro "Artificial Intelligence through Simulated Evolution", con el cual se originara la programación evolutiva.

3.1.1 Programación evolutiva

Lawrence J. Fogel propuso en los 1960s una técnica denominada "programación evolutiva", en la cual la inteligencia se ve como un comportamiento adaptativo [84, 85].

La programación evolutiva enfatiza los nexos de comportamiento entre padres e hijos, en vez de buscar emular operadores genéticos específicos (como en el caso de los algoritmos genéticos).

3.1.1.1 Algoritmo

El algoritmo básico de la programación evolutiva es el siguiente:

- Generar aleatoriamente una población inicial.
- Se aplica mutación.
- Se calcula la aptitud de cada hijo y se usa un proceso de selección mediante torneo (normalmente estocástico) para determinar cuáles serán las soluciones que se retendrán.

La programación evolutiva es una abstracción de la evolución al nivel de las especies, por lo que no se requiere el uso de un operador de recombinación (diferentes especies no se pueden cruzar entre sí). Asimismo, usa selección probabilística.

Figura 3.2: Autómata finito de 3 estados. Los símbolos a la izquierda de "/" son de entrada, y los de la derecha son de salida. El estado inicial es C.

3.1.1.2 Ejemplo

Veamos el ejemplo del funcionamiento de la programación evolutiva que se indica en la figura 3.2. La tabla de transiciones de este autómata es la siguiente:

Estado Actual	**C**	**B**	**C**	**A**	**A**	**B**
Símbolo de Entrada	0	1	1	1	0	1
Estado Siguiente	B	C	A	A	B	C
Símbolo de Salida	b	a	c	b	b	a

En este autómata pueden ahora aplicarse cinco diferentes tipos de mutaciones: cambiar un símbolo de salida, cambiar una transición, agregar un estado, borrar un estado y cambiar el estado inicial. El objetivo es hacer que el autómata reconozca un cierto conjunto de entradas (o sea, una cierta expresión regular) sin equivocarse ni una sola vez.

3.1.2 Aplicaciones

Algunas aplicaciones de la programación evolutiva son [80]:

- Predicción
- Generalización
- Juegos

- Control automático
- Problema del viajero
- Planeación de rutas
- Diseño y entrenamiento de redes neuronales
- Reconocimiento de patrones

3.1.3 Estrategias Evolutivas

Las estrategias evolutivas fueron desarrolladas en 1964 en Alemania para resolver problemas hidrodinámicos de alto grado de complejidad por un grupo de estudiantes de ingeniería encabezado por Ingo Rechenberg [9].

3.1.3.1 Algoritmo

La versión original (1+1)-EE usaba un solo padre y con él se generaba un solo hijo. Este hijo se mantenía si era mejor que el padre, o de lo contrario se eliminaba (a este tipo de selección se le llama *extintiva*, porque los peores individuos obtienen una probabilidad de ser seleccionado de cero).

En la (1+1)-EE, un individuo nuevo es generado usando:

$$\bar{x}^{t+1} = \bar{x}^t + N(0, \bar{\sigma})$$

donde t se refiere a la *generación* (o iteración) en la que nos encontramos, y $N(0, \bar{\sigma})$ es un vector de números Gaussianos independientes con una media de cero y desviaciones estándar $\bar{\sigma}$.

3.1.3.2 Ejemplo

Considere el siguiente ejemplo de una (1+1)-EE (estrategia evolutiva de dos miembros):

Supongamos que queremos optimizar:

$$f(x_1, x_2) = 100(x_1^2 - x_2)^2 + (1 - x_1)^2$$

donde: $-2.048 \leq x_1, x_2 \leq 2.048$

Ahora, supongamos que nuestra población consiste del siguiente individuo (generado de forma aleatoria):

$$(\bar{x}^t, \bar{\sigma}) = (-1.0, 1.0), (1.0, 1.0)$$

Supongamos también que las mutaciones producidas son las siguientes:

$$x_1^{t+1} = x_1^t + N(0, 1.0) = -1.0 + 0.61 = 0.39$$
$$x_2^{t+1} = x_2^t + N(0, 1.0) = 1 + 0.57 = 1.57$$

Ahora, comparamos al padre con el hijo:

Padre	$f(x_t) = f(-1.0, 1.0) = 4.0$
Hijo	$f(x_{t+1}) = f(-0.39, 1.57) = 201.416$

Dado que: $201.416 > 4.0$
el hijo reemplazará al padre en la siguiente generación.

Rechenberg [184] introdujo el concepto de población, al proponer una estrategia evolutiva llamada ((μ+1)-EE, en la cual hay μ padres y se genera un solo hijo, el cual puede reemplazar al peor padre de la población (selección extintiva).

Schwefel [203, 204] introdujo el uso de múltiples hijos en las denominadas (μ+λ)-EEs y (μ,λ)-EEs. La notación se refiere al mecanismo de selección utilizado:

- En el primer caso, los μ mejores individuos obtenidos de la unión de padres e hijos sobreviven.
- En el segundo caso, sólo los μ mejores hijos de la siguiente generación sobreviven.

3.1.3.3 Convergencia

Rechenberg [184] formuló una regla para ajustar la desviación estándar de forma determinística durante el proceso evolutivo de tal manera que el procedimiento convergiera hacia el óptimo.

Esta regla se conoce como la "regla del éxito 1/5", y en palabras dice:

Figura 3.3: Thomas Bäck.

> "La razón entre mutaciones exitosas y el total de mutaciones debe ser 1/5. Si es mayor, entonces debe incrementarse la desviación estándar. Si es menor, entonces debe decrementarse".

Formalmente:

$$\sigma(t) = \begin{cases} \sigma(t-n)/c & \text{si } p_s > 1/5 \\ \sigma(t-n) * c & \text{si } p_s < 1/5 \\ \sigma(t-n) & \text{si } p_s = 1/5 \end{cases}$$

donde n es el número de dimensiones, t es la generación, p_s es la frecuencia relativa de mutaciones exitosas medida sobre intervalos de (por ejemplo) $10n$ individuos, y $c = 0.817$ (este valor fue derivado teóricamente por Schwefel [204]). $\sigma(t)$ se ajusta cada n mutaciones.

Thomas Bäck [9] derivó una regla de éxito 1/7 para las (μ,λ)-EEs.

3.1.3.4 Auto-Adaptación

En las estrategias evolutivas se evoluciona no sólo a las variables del problema, sino también a los parámetros mismos de la técnica (es decir, las desviaciones estándar). A esto se le llama "auto-adaptación").

Los padres se mutan usando las siguientes fórmulas:

$$\sigma'(i) = \sigma(i) \times exp(\tau' N(0,1) + \tau N_i(0,1))$$

$$x'(i) = x(i) + N(0, \sigma'(i))$$

donde τ y τ' constantes de proporcionalidad que están en función de n.
Los operadores de recombinación de las estrategias evolutivas pueden ser:

- **Sexuales**: el operador actúa sobre 2 individuos elegidos aleatoriamente de la población de padres.
- **Panmíticos**: se elige un solo padre al azar, y se mantiene fijo mientras se elige al azar un segundo padre (de entre toda la población) para cada componente de sus vectores.

Las estrategias evolutivas simulan el proceso evolutivo al nivel de los individuos, por lo que la recombinación es posible. Asimismo, usan normalmente selección determinística.

3.1.3.5 Estrategias Evolutivas vs Programación Evolutiva

La Programación Evolutiva usa normalmente selección estocástica, mientras que las estrategias evolutivas usan selección determinística.

Ambas técnicas operan a nivel fenotípico (es decir, no requieren codificación de las variables del problema).

La programación evolutiva es una abstracción de la evolución al nivel de las especies, por lo que no se requiere el uso de un operador de recombinación (diferentes especies no se pueden cruzar entre sí). En contraste, las estrategias evolutivas son una abstracción de la evolución al nivel de un individuo, por lo que la recombinación es posible.

3.1.3.6 Aplicaciones

Algunas aplicaciones de las estrategias evolutivas son [204]:

- Problemas de ruteo y redes
- Bioquímica
- Optica
- Diseño en ingeniería
- Magnetismo

Figura 3.4: Portada de una edición reciente (publicada por el MIT Press) del libro en el que Holland diera a conocer originalmente los algoritmos genéticos (en 1975).

3.1.4 Algoritmos Genéticos

Los algoritmos genéticos (denominados originalmente "planes reproductivos genéticos") fueron desarrollados por John H. Holland a principios de los 1960s [125, 126], motivado por resolver problemas de aprendizaje de máquina.

3.1.4.1 Algoritmo

El algoritmo genético enfatiza la importancia de la cruza sexual (operador principal) sobre el de la mutación (operador secundario), y usa selección probabilística.

El algoritmo básico es el siguiente:

- Generar (aleatoriamente) una población inicial.
- Calcular aptitud de cada individuo.
- Seleccionar (probabilísticamente) en base a aptitud.
- Aplicar operadores genéticos (cruza y mutación) para generar la siguiente población.
- Ciclar hasta que cierta condición se satisfaga.

La representación tradicional es la binaria, tal y como se ejemplifica en la figura 3.5.

Figura 3.5: Ejemplo de la codificación (mediante cadenas binarias) usada tradicionalmente con los algoritmos genéticos.

A la cadena binaria se le llama "cromosoma". A cada posición de la cadena se le denomina "gene" y al valor dentro de esta posición se le llama "alelo".

Para poder aplicar el algoritmo genético se requiere de los 5 componentes básicos siguientes:

- Una representación de las soluciones potenciales del problema.
- Una forma de crear una población inicial de posibles soluciones (normalmente un proceso aleatorio).
- Una función de evaluación que juegue el papel del ambiente, clasificando las soluciones en términos de su "aptitud".
- Operadores genéticos que alteren la composición de los hijos que se producirán para las siguientes generaciones.
- Valores para los diferentes parámetros que utiliza el algoritmo genético (tamaño de la población, probabilidad de cruza, probabilidad de mutación, número máximo de generaciones, etc.)

3.1.4.2 Algoritmos genéticos vs otras técnicas evolutivas

El AG usa selección probabilística al igual que la Programación Evolutiva, y en contraposición a la selección determinística de las Estrategias Evolutivas.

El AG usa representación binaria para codificar las soluciones a un problema, por lo cual se evoluciona el genotipo y no el fenotipo como en la Programación Evolutiva o las Estrategias Evolutivas.

El operador principal en el AG es la cruza, y la mutación es un operador secundario. En la Programación Evolutiva, no hay cruza y en las Estrategias Evolutivas es un operador secundario.

Ha sido demostrado [195] que el AG requiere de elitismo (o sea, retener intacto al mejor individuo de cada generación) para poder converger al óptimo.

Figura 3.6: Portada del libro de David E. Goldberg sobre algoritmos genéticos. A este importante libro se debe, en gran medida, el éxito (cada vez mayor) de que han gozado los algoritmos genéticos desde principios de los 1990s.

Los AGs no son, normalmente, auto-adaptativos, aunque el uso de dicho mecanismo es posible, y ha sido explorado extensivamente en la literatura especializada (ver por ejemplo: [6, 63, 201]).

Puede verse una comparación más detallada de los tres paradigmas anteriores en la tabla 3.1.

3.1.4.3 Aplicaciones

Algunas aplicaciones de los AGs son las siguientes [105]:

- Optimización (estructural, de topologías, numérica, combinatoria, etc.)
- Aprendizaje de máquina (sistemas clasificadores)
- Bases de datos (optimización de consultas)
- Reconocimiento de patrones (por ejemplo, imágenes)
- Generación de gramáticas (regulares, libres de contexto, etc.)
- Planeación de movimientos de robots
- Predicción

Tabla 3.1: Tabla comparativa de los tres paradigmas principales que conforman la computación evolutiva [9].

	Estrategias Evolutivas	**Programación Evolutiva**	**Algoritmo Genético**
Representación	Real	Real	Binaria
Función de Aptitud	Valor de la función objetivo	Valor de la función objetivo ajustada	Valor de la función objetivo ajustada
Auto-Adaptación	Desviaciones Estándar y ángulos de rotación	Ninguna Varianzas (PE-estándar), Coeficientes de correlación (meta-PE)	Ninguna
Mutación	Gausiana, operador principal	Gausiana, operador único	Inversión de bits, operador secundario
Recombinación	Discreta e intermedia, sexual y panmítica, importante para la auto-adaptación	Ninguna	Cruza de 2-puntos, cruza uniforme, únicamente sexual, operador principal
Selección	Determinística, extintiva o basada en la preservación	Probabilística, extintiva	Probabilística, basada en la preservación
Restricciones	Restricciones arbitrarias de desigualdad	Ninguna	Límites simples mediante el mecanismo de codificación
Teoría	Velocidad de Convergencia para casos especiales, (1+1)-ES, (1+λ)-ES, Convergencia global para $(\mu + \lambda)$-ES	Velocidad de Convergencia para casos especiales, (1+1)-PE, Convergencia global para meta-PE	Teoría de los Esquemas, Convergencia global para la versión elitista

3.2 Diferencias de las técnicas evolutivas con respecto a las tradicionales

Existen varias diferencias que vale la pena destacar entre los algoritmos evolutivos y las técnicas tradicionales de búsqueda y optimización [33, 105]:

- Las técnicas evolutivas usan una población de soluciones potenciales en vez de un solo individuo, lo cual las hace menos sensibles a quedar atrapadas en mínimos/máximos locales.
- Las técnicas evolutivas no necesitan conocimiento específico sobre el problema que intentan resolver.
- Las técnicas evolutivas usan operadores probabilísticos, mientras las técnicas tradicionales utilizan operadores determinísticos.
- Aunque las técnicas evolutivas son estocásticas, el hecho de que usen operadores probabilísticos no significa que operen de manera análoga a una simple búsqueda aleatoria.

3.3 Ventajas de las Técnicas Evolutivas

Es importante destacar las diversas ventajas que presenta el uso de técnicas evolutivas para resolver problemas de búsqueda y optimización [105, 80]:

- Simplicidad Conceptual.
- Amplia aplicabilidad.
- Superiores a las técnicas tradicionales en muchos problemas del mundo real.
- Tienen el potencial para incorporar conocimiento sobre el dominio y para hibridizarse con otras técnicas de búsqueda/optimización.
- Pueden explotar fácilmente las arquitecturas en paralelo.
- Son robustas a los cambios dinámicos.
- Generalmente pueden auto-adaptar sus parámetros.
- Capaces de resolver problemas para los cuales no se conoce solución alguna.

Figura 3.7: Portada del libro de John Koza sobre programación genética. Este libro marcó el inicio de una nueva área dentro de la computación evolutiva, dedicada principalmente a la solución de problemas de regresión simbólica.

3.4 Críticas a las Técnicas Evolutivas

Los algoritmos evolutivos fueron muy criticados en sus orígenes (en los 1960s), y todavía siguen siendo blanco de ataques por parte de investigadores de IA simbólica [80, 85]. Se creía, por ejemplo, que una simple búsqueda aleatoria podía superarlas. De hecho, algunos investigadores lograron mostrar esto en algunas de las primeras aplicaciones de la computación evolutiva en los 1960s, aunque eso se debió más bien en limitantes en cuanto al poder de cómputo y ciertas fallas en los modelos matemáticos adoptados en aquella época.

La programación automática fue considerada también como una "moda pasajera" en IA y el enfoque evolutivo fue visto como "un intento más" por lograr algo que lucía imposible. Sin embargo, los resultados obtenidos en los últimos años con programación genética[145, 146, 147] hacen ver que el poder de la computación evolutiva en el campo de la programación automática es mayor del que se creía en los 1960s.

Actualmente, todavía muchas personas creen que un AG funciona igual que una técnica "escalando la colina" que comienza de varios puntos. Se ha demostrado que esto no es cierto, aunque el tema sigue siendo objeto de acalorados debates.

Las técnicas sub-simbólicas (redes neuronales y computación evolutiva) gozan de gran popularidad entre la comunidad científica en general, excepto por algunos especialistas de IA clásica que las consideran "mal fundamentadas" e "inestables".

3.5 Problemas Propuestos

1. Investigue en qué consisten los algoritmos culturales y su relación con los algoritmos genéticos. Consulte, por ejemplo:

   - Robert G. Reynolds, "An Introduction to Cultural Algorithms", In A. V. Sebald, , and L. J. Fogel, editors, *Proceedings of the Third Annual Conference on Evolutionary Programming*, pages 131–139. World Scientific, River Edge, New Jersey, 1994.
   - Robert G. Reynolds, Zbigniew Michalewicz, and M. Cavaretta, "Using cultural algorithms for constraint handling in GENOCOP", In J. R. McDonnell, R. G. Reynolds, and D. B. Fogel, editors, *Proceedings of the Fourth Annual Conference on Evolutionary Programming*, pages 298–305. MIT Press, Cambridge, Massachusetts, 1995.

2. Investigue en qué consiste la búsqueda dispersa (*scatter search*) y su relación con la computación evolutiva. Se le sugiere leer:

   - Fred Glover, "Genetic Algorithms and Scatter Search: Unsuspected Potentials", *Statistics and Computing*, Vol. 4, pp. 131–140, 1994.
   - Fred Glover, "Scatter Search and Star-Paths: Beyond the Genetic Metaphor", *OR Spektrum*, Vol. 17, pp. 125–137, 1995.

3. Implemente una estrategia evolutiva de 2 miembros (la denominada $(1+1)-EE$) cuyo algoritmo que se presenta a continuación:

   t = contador de generaciones, n = número de variables
   Gmax = número máximo de generaciones
   t $\leftarrow$ 0
   Inicializar variables $\bar{x}$,
   Evaluar $f(\bar{x})$
   while (t $\leq$ Gmax) **do**
   inicializar semilla de aleatorios
   mutar el vector x_i usando:
   $x_i^{'} = x_i + \sigma[t] \times N_i(0,1) \ \ \forall i \in n$
   Evaluar $f(\bar{x}')$
   Comparar $\bar{x}$ con $\bar{x}'$ y seleccionar el mejor
   Imprimir en un archivo los resultados

t = t+1
if (t **mod** n == 0) **then**

$$\sigma[t] = \begin{cases} \sigma[t-n]/c & \text{if } p_s > 1/5 \\ \sigma[t-c] \cdot c & \text{if } p_s < 1/5 \\ \sigma[t-n] & \text{if } p_s = 1/5 \end{cases}$$

else $\sigma[t] = \sigma[t-1]$

Algunos puntos importantes que deben observarse en este programa son los siguientes:

- El valor de la constante c oscila entre 0.817 y 1.0, aunque muchos suelen establecerlo en 0.817 ó 0.85.
- p_s es la frecuencia de éxito de las mutaciones. Para calcularlo, se registrará como exitosa aquella mutación en la que el hijo reemplace a su padre. La actualización de p_s normalmente se efectúa cada $10 \cdot n$ iteraciones.
- Usaremos $\sigma[0] = 3.0$.
- $Gmax$ será proporcionada por el usuario.
- La semilla de aleatorios se inicializará con $time(0)$, que es una función en C que devuelve un entero en base al reloj interno de la computadora.
- $N(0,1)$ es una función que genera números aleatorios Gaussianos (distribución normal) con media cero y desviación estándar uno.
- Cuidar que los valores de las variables no se salgan del rango especificado.

Escribir un programa en C que implemente el algoritmo arriba indicado y utilizarlo para **minimizar**:

$$\mathbf{f(x_1, x_2) = 100(x_1^2 - x_2)^2 + (1 - x_1)^2} \tag{3.1}$$

donde $-2.048 \leq x_1, x_2 \leq 2.048$.

Asegúrese de incluir lo siguiente en su reporte:

(a) El código fuente del programa, con comentarios suficientemente claros como para entender el funcionamiento del mismo.

(b) Una gráfica de la función a optimizarse dentro de los rangos permisibles para las variables. La gráfica deberá estar en 3 dimensiones.

(c) Una corrida de ejemplo (presumiblemente, una corrida representativa de su funcionamiento).

Capítulo 4

Terminología Biológica y de Computación Evolutiva

4.1 Introducción

El **Acido Desoxirribonucleico** (ADN) es el material genético fundamental de todos los organismos vivos. El ADN es una macro-molécula doblemente trenzada que tiene una estructura helicoidal como se ilustra en la Figura 4.1. Ambos filamentos trenzados son moléculas de ácido nucleico lineales y sin ramificaciones, formadas de moléculas alternadas de desoxirribosa (azúcar) y fosfato. El ADN de un organismo puede contener desde una docena de genes (como un virus), hasta decenas de miles (como los humanos).

Las 4 bases de nucleótido (ver figura 4.2): Adenina (A), Timina (T), Citosina (C) y Guanina (G) son el alfabeto de información genética. Las secuencias de estas bases en la molécula de ADN determinan el plan constructor de cualquier organismo.

Figura 4.1: Estructural helicoidal del ADN.

Figura 4.2: Las cuatro bases de nucleótido

Figura 4.3: Cromosomas

Un **gene** es una sección de ADN que codifica una cierta función bioquímica definida, usualmente la producción de una proteína. Es fundamentalmente una unidad de herencia.

El ADN de un organismo puede contener desde una docena de genes (como un virus), hasta decenas de miles (como los humanos).

Se denomina **cromosoma** a una de las cadenas de ADN que se encuentra en el núcleo de las células (ver figura 4.3). Los cromosomas son responsables de la transmisión de información genética.

Cada gene es capaz de ocupar sólo una región en particular de un cromosoma (su "lugar" o "locus"). En cada determinado lugar pueden existir, en la población, formas alternativas del gene. A estas formas alternativas se les llama **alelos**.

Se llama **Genoma** a la colección total de genes (y por tanto, cromosomas) que posee un organismo (ver figura 4.4).

Se denomina **Gametos** a las células que llevan información genética de los padres con el propósito de efectuar reproducción sexual. En los animales, se denomina **esperma** a los gametos masculinos y **óvulos** a los gametos femeninos.

Se denomina **haploide** a la célula que contiene un solo cromosoma o conjunto de cromosomas, cada uno de los cuales consiste de una sola secuencia de genes.

Figura 4.4: Genoma

Un ejemplo es un **gameto**.

Se denomina **diploide** a una célula que contiene 2 copias de cada cromosoma. Las copias son homólogas, es decir, contienen los mismos genes en la misma secuencia.

En muchas especies que se reproducen sexualmente, los genes en uno de los conjuntos de cromosomas de una célula diploide se heredan del gameto del padre, mientras que los genes del otro conjunto son del gameto de la madre.

Se denomina **individuo** a un solo miembro de una población. Se denomina **población** a un grupo de individuos que pueden interactuar juntos, por ejemplo, para reproducirse.

Se denomina **fenotipo** a los rasgos (observables) específicos de un individuo. Se denomina **genotipo** a la composición genética de un organismo (la información contenida en el genoma). Es decir, es lo que potencialmente puede llegar a ser un individuo.

El **genotipo** da origen, tras el desarrollo fetal y posterior, al **fenotipo** del organismo (ver figura 4.5).

En la naturaleza, la mayoría de las especies capaces de reproducirse sexualmente son diploides (ver figura 4.6).

Durante la reproducción sexual ocurre la recombinación (o cruza):

- **Caso Haploide**: Se intercambian los genes entre los cromosomas (haploides) de los dos padres.
- **Caso Diploide**: En cada padre, se intercambian los genes entre cada par de cromosomas para formar un gameto, y posteriormente los gametos de los 2

Figura 4.5: Un feto humano.

Figura 4.6: Célula diploide.

Figura 4.7: Una mutación (error de copiado).

padres se aparean para formar un solo conjunto de cromosomas diploides.

Durante la **mutación** (ver figura 4.7), se cambian nucleótidos individuales de padre a hijo. La mayoría de estos cambios se producen por errores de copiado.

La **aptitud** de un individuo se define como la probabilidad de que éste viva para reproducirse (**viabilidad**), o como una función del número de descendientes que éste tiene (**fertilidad**).

Se denomina **ambiente** a todo aquello que rodea a un organismo. Puede ser "físico" (abiótico) o biótico. En ambos casos, el organismo ocupa un nicho que ejerce una influencia sobre su aptitud dentro del ambiente total.

Un ambiente biótico puede presentar funciones de aptitud dependientes de la frecuencia dentro de una población. En otras palabras, la aptitud del comportamiento de un organismo puede depender de cuántos más estén comportándose igual.

A través de varias generaciones, los ambientes bióticos pueden fomentar la **co-evolución**, en la cual la aptitud se determina mediante la selección parcial de otras especies.

La **selección** es el proceso mediante el cual algunos individuos en una población son seleccionados para reproducirse, típicamente con base en su aptitud.

La **selección dura** se da cuando sólo los mejores individuos se mantienen para generar progenia futura.

La **selección blanda** se da cuando se usan mecanismos probabilísticos para mantener como padres a individuos que tengan aptitudes relativamente bajas.

Se llama **pleitropía** al efecto en el cual un solo gene puede afectar simultáneamente a varios rasgos fenotípicos. Un ejemplo es un problema con la célula responsable

de formar la hemoglobina. Al fallar, se afecta la circulación sanguínea, las funciones del hígado y las acciones capilares.

Cuando una sola característica fenotípica de un individuo puede ser determinada mediante la interacción simultánea de varios genes, se denomina al efecto: **poligenia**. El color del cabello y de la piel son generalmente rasgos poligénicos.

Aunque no existe una definición universalmente aceptada de **especie**, diremos que es una colección de criaturas vivientes que tienen características similares, y que se pueden reproducir entre sí. Los miembros de una especie ocupan el mismo **nicho ecológico**.

Se denomina **especiación** al proceso mediante el cual aparece una especie. La causa más común de especiación es el aislamiento geográfico.

Si una subpoblación de una cierta especie se separa geográficamente de la población principal durante un tiempo suficientemente largo, sus genes divergirán. Estas divergencias se deben a diferencias en la presión de selección en diferentes lugares, o al fenómeno conocido como **desvío genético**.

Se llama **desvío genético** a los cambios en las frecuencias de genes/alelos en una población con el paso de muchas generaciones, como resultado del azar en vez de la selección. El **desvío genético** ocurre más rápidamente en poblaciones pequeñas y su mayor peligro es que puede conducir a que algunos alelos se extingan, reduciendo en consecuencia la variabilidad de la población.

En los ecosistemas naturales, hay muchas formas diferentes en las que los animales pueden sobrevivir (en los árboles, de la cacería, en la tierra, etc.) y cada estrategia de supervivencia es llamada un "nicho ecológico".

Dos especies que ocupan nichos diferentes (p.ej. una que se alimenta de plantas y otra que se alimenta de insectos) pueden coexistir entre ellas sin competir, de una manera estable.

Sin embargo, si dos especies que ocupan el mismo nicho se llevan a la misma zona, habrá competencia, y a la larga, la especie más débil se extinguirá (localmente).

Por lo tanto, la diversidad de las especies depende de que ocupen una diversidad de nichos (o de que estén separadas geográficamente).

Se denomina **reproducción** a la creación de un nuevo individuo a partir de:

- Dos progenitores (sexual)
- Un progenitor (asexual)

Se denomina **migración** a la transferencia de (los genes de) un individuo de una subpoblación a otra.

Se dice que un **gene** es **epistático** cuando su presencia suprime el efecto de un gene que se encuentra en otra posición. Los genes epistáticos son llamados algunas veces genes de inhibición por el efecto que producen sobre otros genes.

4.2 Tipos de Aprendizaje

Algunos científicos (como Atmar [5]) consideran que existen 3 tipos distintos de inteligencia en los seres vivos:

- Filogenética
- Ontogenética
- Sociogenética

En la **inteligencia filogenética**, el aprendizaje se efectúa en este caso a nivel de las especies. La unidad de mutabilidad es un solo par base nucleótido, y el acervo de la inteligencia es el genoma de la especie.

En la **inteligencia ontogenética**, el aprendizaje se efectúa a nivel del individuo. La unidad de mutabilidad (o cambio) es la propensión de una neurona para dispararse y la sensibilidad del sitio receptor de dicha neurona. El acervo de este tipo de inteligencia es la memoria neuronal y hormonal (a la supuesta ruta de conexiones neuronales aprendidas se le llama "engrama").

En la **inteligencia sociogenética**, el aprendizaje se efectúa a nivel del grupo. La unidad de mutabilidad es la "idea", o la experiencia compartida y el acervo de este tipo de inteligencia es la cultura.

4.3 Conceptos de Computación Evolutiva

Denominamos **cromosoma** a una estructura de datos que contiene una cadena de parámetros de diseño o genes. Esta estructura de datos puede almacenarse, por ejemplo, como una cadena de bits o un arreglo de enteros (ver figura 4.8).

Se llama **gene** a una subsección de un cromosoma que (usualmente) codifica el valor de un solo parámetro (ver figura 4.9).

Se denomina **genotipo** a la codificación (por ejemplo, binaria) de los parámetros que representan una solución del problema a resolverse (ver figura 4.8).

Figura 4.8: Un ejemplo de una cadena cromosómica. Esta cadena es el genotipo que codifica las variables de decisión de un problema.

Figura 4.9: Un ejemplo de un gene.

Figura 4.10: Un ejemplo de un fenotipo.

Figura 4.11: Un ejemplo de un individuo.

Figura 4.12: Un ejemplo de un alelo.

Se denomina **fenotipo** a la decodificación del cromosoma. Es decir, a los valores obtenidos al pasar de la representación (binaria) a la usada por la función objetivo (ver figura 4.10).

Se denomina **individuo** a un solo miembro de la población de soluciones potenciales a un problema. Cada individuo contiene un cromosoma (o de manera más general, un genoma) que representa una solución posible al problema a resolverse (ver figura 4.11).

Se denomina **aptitud** al valor que se asigna a cada individuo y que indica qué tan bueno es éste con respecto a los demás para la solución de un problema. Por ejemplo, si $f(x) = x^2$, entonces $f(1010_2) = 100$ (donde $f(x)$ es la aptitud de un individuo).

Se llama **paisaje de aptitud** (*fitness landscape*) a la hipersuperficie que se obtiene al aplicar la función de aptitud a cada punto del espacio de búsqueda.

Se denomina **alelo** a cada valor posible que puede adquirir una cierta posición genética. Si se usa representación binaria, un alelo puede valer 0 ó 1 (ver figura 4.12).

Llamamos **generación** a una iteración de la medida de aptitud y a la creación de una nueva población por medio de operadores de reproducción.

Una población puede subdividirse en grupos a los que se denomina **subpoblaciones**. Normalmente, sólo pueden cruzarse entre sí los individuos que pertenez-

Figura 4.13: Un ejemplo de migración.

can a la misma subpoblación.

En los esquemas con subpoblaciones, suele permitirse la **migración** de una subpoblación a otra (sobre todo en el contexto de algoritmos evolutivos paralelos).

Al hecho de permitir la cruza sólo entre individuos de la misma subpoblación se le llama **especiación** en una emulación del fenómeno natural del mismo nombre.

Se llama **migración** a la transferencia de (los genes de) un individuo de una subpoblación a otra.

Hay un tipo de población usada en computación evolutiva en la que cualquier individuo puede reproducirse con otro con una probabilidad que depende sólo de su aptitud. Se le llama **población panmítica**.

Lo opuesto de la población panmítica es permitir la reproducción sólo entre individuos de la misma subpoblación. La mayor parte de los algoritmos evolutivos (AEs) convencionales usan poblaciones panmíticas.

Debido a ruidos estocásticos, los AEs tienden a converger a una sola solución. Para evitar eso, y mantener la diversidad, existen técnicas que permiten crear distintos **nichos** para los individuos.

Se llama **epístasis** a la interacción entre los diferentes genes de un cromosoma. Se refiere a la medida en que la contribución de aptitud de un gene depende de los valores de los otros genes.

Cuando un problema tiene poca **epístasis** (o ninguna), su solución es trivial (un algoritmo escalando la colina es suficiente para resolverlo). Cuando un problema tiene una **epístasis** elevada, el problema será **deceptivo**, por lo que será muy difícil de resolver por un AE.

Se llama **bloque constructor** a un grupo pequeño y compacto de genes que han co-evolucionado de tal forma que su introducción en cualquier cromosoma tiene una alta probabilidad de incrementar la aptitud de dicho cromosoma.

Figura 4.14: Ejemplo del operador de inversión.

Se llama **decepción** a la condición donde la combinación de buenos bloques constructores llevan a una reducción de aptitud, en vez de un incremento. Este fenómeno fue sugerido originalmente por Goldberg [105] para explicar el mal desempeño del AG en algunos problemas.

Se llama **operador de reproducción** a todo aquel mecanismo que influencia la forma en que se pasa la información genética de padres a hijos. Los operadores de reproducción caen en tres amplias categorías:

- Cruza
- Mutación
- Reordenamiento

La **cruza** es un operador que forma un nuevo cromosoma combinando partes de cada uno de sus cromosomas padres.

Se denomina **mutación** a un operador que forma un nuevo cromosoma a través de alteraciones (usualmente pequeñas) de los valores de los genes de un solo cromosoma padre.

Un **operador de reordenamiento** es aquél que cambia el orden de los genes de un cromosoma, con la esperanza de juntar los genes que se encuentren relacionados, facilitando así la producción de bloques constructores.

La **inversión** es un ejemplo de un operador de reordenamiento en el que se invierte el orden de todos los genes comprendidos entre 2 puntos seleccionados al azar en el cromosoma (ver figura 4.14).

En un algoritmo genético (AG), cuando una población no tiene **variedad requisito**, la cruza no será útil como operador de búsqueda, porque tendrá propensión a simplemente regenerar a los padres.

Es importante aclarar que en los AGs los operadores de reproducción actúan sobre los **genotipos** y no sobre los **fenotipos** de los individuos.

Se denomina **elitismo** al mecanismo utilizado en algunos AEs para asegurar que los cromosomas de los miembros más aptos de una población se pasen a la siguiente generación sin ser alterados por ningún operador genético.

Usar **elitismo** asegura que la aptitud máxima de la población nunca se reducirá de una generación a la siguiente. Sin embargo, no necesariamente mejora la posibilidad de localizar el óptimo global de una función. No obstante, es importante hacer notar que se ha demostrado que el uso de elitismo es vital para poder demostrar convergencia de un algoritmo genético [195].

Cuando se atraviesa un espacio de búsqueda, se denomina **explotación** al proceso de usar la información obtenida de los puntos visitados previamente para determinar qué lugares resulta más conveniente visitar a continuación.

Se denomina **exploración** al proceso de visitar completamente nuevas regiones del espacio de búsqueda, para ver si puede encontrarse algo prometedor. La **exploración** involucra grandes saltos hacia lo desconocido. La **explotación** normalmente involucra movimientos finos. La **explotación** es buena para encontrar óptimos locales. La **exploración** es buena para evitar quedar atrapado en óptimos locales.

Se denomina **esquema** a un patrón de valores de genes de un cromosoma que puede incluir estados 'no importa' (*don't care*).

Usando un alfabeto binario, los esquemas se forman del alfabeto 0,1,#. Por ejemplo, el cromosoma 0110 es una instancia del esquema #1#0 (donde # significa 'no importa').

4.4 Problemas propuestos

1. Investigue acerca de los **intrones**, o segmentos de ADN no codificados y escriba un ensayo al respecto. Su trabajo debe incluir una discusión de los intrones desde la perspectiva biológica, así como su uso en computación evolutiva.

   Se recomienda consultar:

   Coello Coello, Carlos A.; Rocío Reyes Díaz; Héctor G. Lugo Guevara y Julio César Sandria Reynoso, "El Misterio de los Intrones", Lania-RD-2000-03, Laboratorio Nacional de Informática Avanzada, 2000.

2. Investigue alguno de los mecanismos de restricciones a la cruza que han sido propuestos en computación evolutiva y que tenga una fuerte inspiración biológica. Discuta, por ejemplo, la "prevención de incesto" que propusieron Eshelman y Shafer [75].

Capítulo 5

La Importancia de la Representación

5.1 Introducción

Las capacidades para procesamiento de datos de las técnicas de computación evolutiva dentro de una amplia gama de dominios han sido reconocidas en los últimos años y han recibido mucha atención por parte de científicos que trabajan en diversas disciplinas. Dentro de estas técnicas evolutivas, quizás la más popular sea el algoritmo genético (AG) [105]. Siendo una técnica heurística estocástica, el algoritmo genético no necesita información específica para guiar la búsqueda. Su estructura presenta analogías con la teoría biológica de la evolución, y se basa en el principio de la supervivencia del más apto [127]. Por lo tanto, el AG puede verse como una "caja negra" que puede conectarse a cualquier aplicación en particular. En general, se necesitan los cinco componentes básicos siguientes para implementar un AG que resuelva un problema cualquiera [159]:

1. Una representación de soluciones potenciales al problema.

2. Una forma de crear una población inicial de soluciones potenciales (esto se efectúa normalmente de manera aleatoria, pero también pueden usarse métodos determinísticos).

3. Una función de evaluación que juega el papel del ambiente, calificando a las soluciones producidas en términos de su "aptitud".

Figura 5.1: Un ejemplo de una cadena binaria.

4. Operadores genéticos que alteran la composición de los descendientes (normalmente se usan la cruza y la mutación).

5. Valores para los diversos parámetros utilizados por el algoritmo genético (tamaño de la población, probabilidad de cruza y mutación, número máximo de generaciones, etc.)

En este capítulo hablaremos exclusivamente del primero de estos componentes: la representación usada por el algoritmo genético. La representación tradicional usada para codificar un conjunto de soluciones es el esquema binario en el cual un cromosoma[1] es una cadena de la forma $\langle b_1, b_2, \ldots, b_m \rangle$ (ver figura 5.1), donde $b_1, b_2, \ldots, b_m$ se denominan *alelos* (ya sea ceros o unos).

Hay varias razones por las cuales suele usarse la codificación binaria en los AGs, aunque la mayoría de ellas se remontan al trabajo pionero de Holland en el área. En su libro, Holland [127] dio una justificación teórica para usar codificaciones binarias. Holland comparó dos representaciones diferentes que tuvieran aproximadamente la misma capacidad de acarreo de información, pero de entre ellas, una tenía pocos alelos y cadenas largas (por ejemplo, cadenas binarias de 80 bits de longitud) y la otra tenía un número elevado de alelos y cadenas cortas (por ejemplo, cadenas decimales de longitud 24). Nótese que 2^{80} (codificación binaria) $\approx 10^{24}$ (codificación decimal). Holland [127] argumentó que la primera codificación da pie a un grado más elevado de 'paralelismo implícito' porque permite más "esquemas" que la segunda (11^{24} contra 3^{80}).

El número de esquemas de una cadena se calcula usando $(c+1)^l$, donde c es la cardinalidad del alfabeto y l es la longitud de la cadena[2]. Un "esquema" es una plantilla que describe un subconjunto de cadenas que comparten ciertas similitudes en algunas posiciones a lo largo de su longitud [105, 127].

[1] Un **cromosoma** es una estructura de datos que contiene una 'cadena' de parámetros de diseño o genes.

[2] La razón por la que se suma uno a la cardinalidad es porque en los esquemas se usa un símbolo adicional (normalmente el asterisco o el símbolo de número) para indicar que "no nos importa" el valor de esa posición.

El hecho de contar con más esquemas favorece la diversidad e incrementa la probabilidad de que se formen buenos "bloques constructores" (es decir, la porción de un cromosoma que le produce una aptitud elevada a la cadena en la cual está presente) en cada generación, lo que en consecuencia mejora el desempeño del AG con el paso del tiempo de acuerdo al teorema de los esquemas [105, 127]. El 'paralelismo implícito' de los AGs, demostrado por Holland [127], se refiere al hecho de que mientras el AG calcula las aptitudes de los individuos en una población, estima de forma implícita las aptitudes promedio de un número mucho más alto de cadenas cromosómicas a través del cálculo de las aptitudes promedio observadas en los "bloques constructores" que se detectan en la población.

Por lo tanto, de acuerdo a Holland [127], es preferible tener muchos genes[3] con pocos alelos posibles que contar con pocos genes con muchos alelos posibles. Esto es sugerido no sólo por razones teóricas (de acuerdo al teorema de los esquemas formulado por Holland), sino que también tiene una justificación biológica, ya que en genética es más usual tener cromosomas con muchas posiciones y pocos alelos por posición que pocas posiciones y muchos alelos por posición [127].

Sin embargo, Holland [127] también demostró que el paralelismo implícito de los AGs no impide usar alfabetos de mayor cardinalidad [159], aunque debe estarse siempre consciente de que el alfabeto binario es el que ofrece el mayor número de esquemas posibles por bit de información si se compara con cualquier otra codificación posible [105, 159]. No obstante, ha habido un largo debate en torno a cuestiones relacionadas con estos alfabetos no binarios, principalmente por parte de los especialistas en aplicaciones de los AGs. Como veremos en este capítulo, el uso de la representación binaria tiene varias desventajas cuando el AG se usa para resolver ciertos problemas del mundo real. Por ejemplo, si tratamos de optimizar una función con alta dimensionalidad (digamos, con 50 variables), y queremos trabajar con una buena precisión (por ejemplo, cinco decimales), entonces el mapeo de números reales a binarios generará cadenas extremadamente largas (del orden de 1000 bits en este caso), y el AG tendrá muchos problemas para producir resultados aceptables en la mayor parte de los casos, a menos que usemos procedimientos y operadores especialmente diseñados para el problema en cuestión. Ronald [192] resume las principales razones por las que una codificación binaria puede no resultar adecuada en un problema dado:

- Epístasis : el valor de un bit puede suprimir las contribuciones de aptitud de

[3]Se denomina **gene** o **gen** a cualquier posición a lo largo de una cadena que representa a un individuo.

otros bits en el genotipo[4].

- Representación natural : algunos problemas (como el del viajero) se prestan de manera natural para la utilización de representaciones de mayor cardinalidad que la binaria (por ejemplo, el problema del viajero se presta de manera natural para el uso de permutaciones de enteros decimales).

- Soluciones ilegales : los operadores genéticos utilizados pueden producir con frecuencia (e incluso todo el tiempo) soluciones ilegales si se usa una representación binaria.

En el resto de este capítulo discutiremos algunos esquemas de representación alternativos que han sido propuestos recientemente para lidiar con éstas y otras limitaciones de la representación binaria.

5.2 Códigos de Gray

Un problema que fue notado desde los inicios de la investigación en AGs fue que el uso de la representación binaria no mapea adecuadamente el espacio de búsqueda con el espacio de representación [128]. Por ejemplo, si codificamos en binario los enteros 5 y 6, los cuales están adyacentes en el espacio de búsqueda, sus equivalentes en binario serán el 101 y el 110, los cuales difieren en 2 bits (el primero y el segundo de derecha a izquierda) en el espacio de representación. A este fenómeno se le conoce como el *risco de Hamming* (*Hamming cliff*) [40], y ha conducido a los investigadores a proponer una representación alternativa en la que la propiedad de adyacencia existente en el espacio de búsqueda pueda preservarse en el espacio de representación. La codificación de Gray es parte de una familia de representaciones que caen dentro de esta categoría [227].

Podemos convertir cualquier número binario a un código de Gray haciendo XOR a sus bits consecutivos de derecha a izquierda. Por ejemplo, dado el número 0101 en binario, haríamos[5]: $1 \oplus 0 = 1, 0 \oplus 1 = 1, 1 \oplus 0 = 1$, produciéndose (el último bit de la izquierda permanece igual) 0111, el cual es el código de Gray equivalente. Algunos investigadores han demostrado empíricamente que el uso de códigos de Gray mejora el desempeño del AG al aplicarse a las funciones de

[4]El **genotipo** es la cadena cromosómica utilizada para almacenar la información contenida en un individuo, mientras que el **fenotipo** se refiere a los valores que toman las variables tras "decodificar" el contenido cromosómico de un individuo.

[5]$\oplus$ indica XOR.

Figura 5.2: Un ejemplo de notación del IEEE.

prueba clásicas de De Jong [137] (ver por ejemplo [40, 155]. De hecho, Mathias y Whitley [156] encontraron que la codificación de Gray no sólo elimina los riscos de Hamming, sino que también altera el número de óptimos locales en el espacio de búsqueda así como el tamaño de las buenas regiones de búsqueda (aquellas que nos conducirán a la vecindad del óptimo global). En su trabajo, Mathias y Whitley mostraron empíricamente que un mutador aleatorio del tipo "escalando la colina" es capaz de encontrar el óptimo global de la mayor parte de las funciones de prueba utilizadas cuando se emplea la codificación de Gray, a pesar de que algunas de ellas fueron diseñadas explícitamente para presentar dificultades a los algoritmos de búsqueda tradicionales (sean evolutivos o no).

5.3 Codificando Números Reales

Aunque los códigos de Gray pueden ser muy útiles para representar enteros, el problema de mapear correctamente el espacio de búsqueda en el espacio de representación se vuelve más serio cuando tratamos de codificar números reales. En el enfoque tradicional [231], se usa un número binario para representar un número real, definiendo límites inferiores y superiores para cada variable, así como la precisión deseada. Por ejemplo, si queremos codificar una variable que va de 0.35 a 1.40 usando una precisión de 2 decimales, necesitaríamos $\log_2(140 - 35) \approx 7$ bits para representar cualquier número real dentro de ese rango. Sin embargo, en este caso, tenemos el mismo problema del que hablamos anteriormente, porque el número 0.38 se representaría como 0000011, mientras que 0.39 se representaría como 0000101.

Aunque se usen códigos de Gray, existe otro problema más importante cuando tratamos de desarrollar aplicaciones del mundo real: la alta dimensionalidad. Si tenemos demasiadas variables, y queremos una muy buena precisión para cada una de ellas, entonces las cadenas binarias que se produzcan se volverán extremadamente largas, y el AG tenderá a tener un desempeño pobre. Si en vez de usar este tipo de mapeo adoptamos algún formato binario estándar para represen-

Figura 5.3: Un ejemplo de un algoritmo genético con representación real.

tar números reales, como por ejemplo el estándar del IEEE para precisión simple, en el cual un número real se representa usando 32 bits, de los cuales 8 se usan para el exponente usando una notación en exceso-127, y la mantisa se representa con 23 bits (ver figura 5.2 [205]), podríamos manejar un rango relativamente grande de números reales usando una cantidad fija de bits (por ejemplo, de 2^{-126} a 2^{127} si usamos el estándar de precisión simple antes descrito). Sin embargo, el proceso de decodificación sería más costoso (computacionalmente hablando) y el mapeo entre el espacio de representación y el de búsqueda sería mucho más complejo que cuando se usa una representación binaria simple, porque cualquier pequeño cambio en el exponente produciría grandes saltos en el espacio de búsqueda, mientras que perturbaciones en la mantisa podrían no cambiar de manera significativa el valor numérico codificado.

Mientras los teóricos afirman que los alfabetos pequeños son más efectivos que los alfabetos grandes, los prácticos han mostrado a través de una cantidad significativa de aplicaciones del mundo real (particularmente problemas de optimización numérica) que el uso directo de números reales en un cromosoma funciona mejor en la práctica que la representación binaria tradicional [64, 76]. El uso de números reales en una cadena cromosómica (ver figura 5.3) ha sido común en otras técnicas de computación evolutiva tales como las estrategias evolutivas [204] y la programación evolutiva [82], donde la mutación es el operador principal. Sin embargo, los teóricos de los AGs han criticado fuertemente el uso de valores reales en los genes de un cromosoma, principalmente porque esta representación de cardinalidad más alta tiende a hacer que el comportamiento del AG sea más errático y difícil de predecir. Debido a esto, se han diseñado varios operadores especiales en los años recientes, para emular el efecto de la cruza y la mutación en los alfabetos binarios [231, 76, 67].

Los prácticos argumentan que una de las principales capacidades de los AGs que usan representación real es la de explotar la "gradualidad" de las funciones de variables continuas[6]. Esto significa que los AGs con codificación real pueden

[6]"Gradualidad" se refiere a los casos en los cuales un cambio pequeño en las variables se

Figura 5.4: Una representación entera de números reales. La cadena completa es decodificada como un solo número real multiplicando y dividiendo cada dígito de acuerdo a su posición.

lidiar adecuadamente con los "riscos" producidos cuando las variables utilizadas son números reales, porque un cambio pequeño en la representación es mapeado como un cambio pequeño en el espacio de búsqueda [231, 76]. En un intento por reducir la brecha entre la teoría y la práctica, algunos investigadores han desarrollado un marco teórico que justifique el uso de alfabetos de más alta cardinalidad [231, 76, 108, 215], pero han habido pocos consensos en torno a los problemas principales, por lo que el uso de AGs con codificación real sigue siendo una elección que se deja al usuario.

Se han usado también otras representaciones de los números reales. Por ejemplo, el uso de enteros para representar cada dígito ha sido aplicado exitosamente a varios problemas de optimización [44, 43]. La figura 5.4 muestra un ejemplo en el cual se representa el número 1.45679 usando enteros. En este caso, se supone una posición fija para el punto decimal en cada variable, aunque esta posición no tiene que ser necesariamente la misma para el resto de las variables codificadas en la misma cadena. La precisión está limitada por la longitud de la cadena, y puede incrementarse o decrementarse según se desee. Los operadores de cruza tradicionales (un punto, dos puntos y uniforme) pueden usarse directamente en esta representación, y la mutación puede consistir en generar un dígito aleatorio para una cierta posición o bien en producir una pequeña perturbación (por ejemplo ± 1) para evitar saltos extremadamente grandes en el espacio de búsqueda. Esta representación pretende ser un compromiso entre un AG con codificación real y una representación binaria de números reales, manteniendo lo mejor de ambos esquemas al incrementar la cardinalidad del alfabeto utilizado, pero manteniendo el uso de los operadores genéticos tradicionales casi sin cambios.

Alternativamente, podríamos también usar enteros largos para representar números reales (ver figura 5.5), pero los operadores tendrían que redefinirse de la misma manera que al usar números reales. El uso de este esquema de representación como una alternativa a los AGs con codificación real parece, sin embargo, un tanto

traduce en un cambio pequeño en la función.

Figura 5.5: Otra representación entera de números reales. En este caso, cada gene contiene un número real representado como un entero largo.

improbable, ya que se tendrían que hacer sacrificios notables en la representación, y los únicos ahorros importantes que se lograrían serían en términos de memoria (el almacenamiento de enteros toma menos memoria que el de números reales). No obstante, este esquema ha sido usado en algunas aplicaciones del mundo real [64].

5.4 Representaciones de Longitud Variable

En algunos problemas el uso de alfabetos de alta cardinalidad puede no ser suficiente, pues además puede requerirse el empleo de cromosomas de longitud variable para lidiar con cambios que ocurran en el ambiente con respecto al tiempo (por ejemplo, el decremento/incremento de la precisión de una variable o la adición/remoción de variables). Algunas veces, puede ser posible introducir símbolos en el alfabeto que sean considerados como posiciones "vacías" a lo largo de la cadena, con lo que se permite la definición de cadenas de longitud variable aunque los cromosomas tengan una longitud fija. Ese es, por ejemplo, el enfoque utilizado en [42] para diseñar circuitos eléctricos combinatorios. En ese caso, el uso de un símbolo llamado WIRE, el cual representa la ausencia de compuerta, permitió cambiar la longitud de la expresión Booleana generada a partir de una matriz bi-dimensional. Sin embargo, en otros dominios, este tipo de simplificación puede no ser posible y deben idearse representaciones alternativas. Por ejemplo, en problemas que tienen decepción parcial o total [117] (es decir, en aquellos problemas en los que los bloques constructores de bajo orden no guían al AG hacia el óptimo y no se combinan para formar bloques constructores de orden mayor), un AG no tendrá un buen desempeño sin importar cuál sea el valor de sus parámetros (tamaño de población, porcentajes de cruza y mutación, etc.). Para lidiar con este tipo de problemas en particular, Goldberg et al. [112, 109, 111] propusieron el uso de un tipo especial de AG de longitud variable el cual usa poblaciones de tamaño variable. A este AG especial se le denominó 'desordenado' (*messy* GA o mGA)

Figura 5.6: Dos ejemplos de cadenas válidas en un algoritmo genético desordenado.

en contraposición con el AG estándar (u ordenado), que tiene longitud y tamaño de población fijos [167]. La idea básica de los AGs desordenados es empezar con cromosomas cortos, identificar un conjunto de buenos bloques constructores y después incrementar la longitud del cromosoma para propagar estos buenos bloques constructores a lo largo del resto de la cadena.

La representación usada por los AGs desordenados es muy peculiar, puesto que cada bit está realmente asociado con una posición en particular a lo largo de la cadena, y algunas posiciones podrían ser asignadas a más de un bit (a esto se le llama *sobre-especificación*) mientras que otras podrían no ser asignadas a ninguno (a esto se le llama *sub-especificación*). Consideremos, por ejemplo, a las dos cadenas mostradas en la figura 5.6, las cuales constituyen cromosomas válidos para un AG desordenado (estamos suponiendo cromosomas de 4 bits). La notación adoptada en este ejemplo usa paréntesis para identificar a cada gene, el cual se define como un par consistente de su posición a lo largo de la cadena (el primer valor) y el valor del bit en esa posición (estamos suponiendo un alfabeto binario). En el primer caso, la primera y la cuarta posición no están especificadas, y la segunda y la tercera están especificadas dos veces. Para lidiar con la sobre-especificación pueden definirse algunas reglas determinísticas muy sencillas. Por ejemplo, podemos usar sólo la primera definición de izquierda a derecha para una cierta posición. Sin embargo, para la sub-especificación tenemos que hacer algo más complicado, porque una cadena sub-especificada realmente representa a un "esquema candidato" en vez de un cromosoma completo. Por ejemplo, la primera cadena de las antes descritas representa al esquema *10* (el * significa "no me importa"). Para calcular la aptitud de una cadena sub-especificada, podemos usar un explorador local del tipo "escalando la colina" que nos permita localizar el óptimo local y la información obtenida la podemos utilizar para reemplazar a los

Figura 5.7: Un ejemplo del operador de "corte" en un AG desordenado. La línea gruesa indica el punto de corte.

"no me importa" del esquema. A esta técnica se le denomina "plantillas competitivas" [109]. Los AGs desordenados operan en 2 fases [109]: la "fase primordial" y la "fase yuxtaposicional". En la primera, se generan esquemas cortos que sirven como los bloques constructores en la fase yuxtaposicional en la cual éstos se combinan. El problema es cómo decidir qué tan largos deben ser estos esquemas "cortos". Si son demasiado cortos, pueden no contener suficiente material genético como para resolver el problema deseado; si son demasiado largos, la técnica puede volverse impráctica debido a la "maldición de la dimensionalidad" (tendríamos que generar y evaluar demasiados cromosomas).

Durante la fase primordial generamos entonces estos esquemas cortos y evaluamos sus aptitudes. Después de eso, aplicamos sólo selección a la población (sin cruza o mutación) para propagar los buenos bloques constructores, y se borra la mitad de la población a intervalos regulares [167]. Después de un cierto número (predefinido) de generaciones, terminamos la fase primordial y entramos a la fase yuxtaposicional. A partir de este punto, el tamaño de la población permanecerá fijo, y usaremos selección y dos operadores especiales llamados "corte" y "unión" [112]. El operador de corte simplemente remueve una porción del cromosoma, mientras que el de unión junta dos segmentos cromosómicos. Considere los ejemplos mostrados en las figuras 5.7 y 5.8.

Debido a la naturaleza del AG desordenado, las cadenas producidas por los operadores de corte y unión siempre serán válidas. Si los bloques constructores producidos en la fase primordial acarrean suficiente información, entonces el AG desordenado será capaz de arribar al óptimo global aunque el problema tenga decepción [111]. Aunque es sin duda muy prometedor, los inconvenientes prácticos del AG desordenado [167] han impedido su uso extendido, y actualmente se reportan relativamente pocas aplicaciones (ver por ejemplo [41, 141, 119].

Figura 5.8: Un ejemplo del operador "unión" en un AG desordenado. La línea gruesa muestra la parte de la cadena que fue agregada.

5.5 Representación de árbol

Una de las metas originales de la Inteligencia Artificial (IA) fue la generación automática de programas de computadora que pudiesen efectuar una cierta tarea. Durante varios años, sin embargo, esta meta pareció demasiado ambiciosa puesto que normalmente el espacio de búsqueda se incrementa exponencialmente conforme extendemos el dominio de un cierto programa y, consecuentemente, cualquier técnica tenderá a producir programas no válidos o altamente ineficientes. Algunas técnicas de computación evolutiva intentaron lidiar con el problema de la programación automática desde su mera concepción, pero sus notables fallas aún en dominios muy simples, hicieron que otros investigadores de la comunidad de IA no se tomaran en serio estos esfuerzos [80]. Sin embargo, Holland desarrolló el concepto moderno del AG dentro del entorno del aprendizaje de máquina [127], y todavía existe una cantidad considerable de investigación dentro de esa área, aunque la programación automática fue hecha a un lado por los investigadores de IA durante varios años. Una de las razones por las que esto sucedió fue el hecho de que el AG tiene algunas limitaciones (obvias) cuando se pretende usar para programación automática, particularmente en términos de la representación. Codificar el conjunto de instrucciones de un lenguaje de programación y encontrar una forma de combinarlas que tenga sentido no es una tarea simple, pero si usamos una estructura de árbol junto con ciertas reglas para evitar la generación de expresiones no válidas, podemos construir un evaluador de expresiones primitivo que pueda producir programas simples. Este fue precisamente el enfoque tomado por John Koza [145] para desarrollar la denominada "programación genética", en la cual se usó originalmente el lenguaje de programación LISP para aprovechar su evaluador de expresiones integrado al intérprete.

La representación de árbol adoptada por Koza requiere obviamente de alfa-

Figura 5.9: Un ejemplo de un cromosoma usado en programación genética.

betos diferentes y operadores especializados para evolucionar programas generados aleatoriamente hasta que éstos se vuelvan 100% válidos para resolver cierta tarea predefinida, aunque los principios básicos de esta técnica pueden generalizarse a cualquier otro dominio. Los árboles se componen de funciones y terminales. Las funciones usadas normalmente son las siguientes [145]:

1. Operaciones aritméticas (por ejemplo: +, -,$\times$, $\div$)
2. Funciones matemáticas (por ejemplo: seno, coseno, logaritmos, etc.)
3. Operaciones Booleanas (por ejemplo: AND, OR, NOT)
4. Condicionales (IF-THEN-ELSE)
5. Ciclos (DO-UNTIL)
6. Funciones recursivas
7. Cualquier otra función definida en el dominio utilizado

Las terminales son típicamente variables o constantes, y pueden verse como funciones que no toman argumentos. Un ejemplo de un cromosoma que usa las funciones F=$\{$AND, OR, NOT$\}$ y las terminales T=$\{$A0, A1$\}$ se muestra en la figura 5.9.

La cruza puede efectuarse numerando los nodos de los árboles correspondientes a los 2 padres elegidos (ver figura 5.10) y seleccionando (al azar) un punto en cada uno de ellos de manera que los sub-árboles por debajo de dicho punto se intercambien (ver figura 5.11, donde suponemos que el punto de cruza para

Figura 5.10: Los nodos del árbol se numeran antes de aplicar el operador de cruza.

Figura 5.11: Los dos hijos generados después de efectuar la cruza.

Figura 5.12: Un ejemplo de mutación en la programación genética.

el primer padre es 2 y para el segundo es 6). Típicamente, los tamaños de los 2 árboles padres será diferente, como se muestra en el ejemplo anterior. También debe observarse que si el punto de cruza es la raíz de uno de los dos árboles padres, entonces todo ese cromosoma se volverá un sub-árbol del otro padre, lo cual permite la incorporación de subrutinas en un programa. También es posible que las raíces de ambos padres sean seleccionadas como puntos de cruza. En ese caso, no se efectúa la cruza, y los hijos pasan a ser idénticos a sus padres. Normalmente, la implementación de la programación genética impone un límite en cuanto a la máxima profundidad que puede alcanzar un árbol, a fin de evitar la generación (al azar y producto del uso de la cruza y la mutación) de árboles de gran tamaño y complejidad.

La mutación se efectúa mediante la selección (aleatoria) de un cierto punto de un árbol. El sub-árbol que se encuentre por debajo de dicho punto es reemplazado por otro árbol generado al azar. La figura 5.12 muestra un ejemplo del uso de este operador (el punto de mutación en este ejemplo es el 3).

La permutación es un operador asexual que emula el efecto del operador de inversión que se usa en los algoritmos genéticos [105]. Este operador reordena las hojas de un sub-árbol ubicado a partir de un punto elegido al azar, y su finalidad es fortalecer la unión de combinaciones de alelos con buen desempeño dentro de un cromosoma [127]. La figura 5.13 muestra un ejemplo del uso del operador de permutación (el punto seleccionado en este caso es el 4). En la figura 5.13, el '*' indica multiplicación y el '%' indica "división protegida", refiriéndose a un operador de división que evita que nuestro programa genere un error de sistema cuando el segundo argumento sea cero.

En la programación genética es posible también proteger o "encapsular" un

Figura 5.13: Un ejemplo de permutación en la programación genética.

Figura 5.14: Un ejemplo de encapsulamiento en programación genética.

Figura 5.15: Un ejemplo de estructura de dos niveles de un AG estructurado.

cierto sub-árbol que sepamos constituye un buen bloque constructor, a fin de evitar que sea destruido por los operadores genéticos. El sub-árbol seleccionado es reemplazado por un nombre simbólico que apunta a la ubicación real del sub-árbol, y dicho sub-árbol es compilado por separado y enlazado al resto del árbol de forma similar a las clases externas de los lenguajes orientados a objetos. La figura 5.14 muestra un ejemplo de encapsulamiento en el cual el sub-árbol de la derecha es reemplazado por el nombre (**E0**).

Normalmente, también es necesario editar las expresiones generadas a fin de simplificarlas, aunque las reglas para llevar a cabo este proceso dependen generalmente del problema. Por ejemplo, si estamos generando expresiones Booleanas, podemos aplicar reglas como las siguientes:

(AND X X) X
(OR X X) X
(NOT (NOT X)) X

Finalmente, la programación genética también proporciona mecanismos para destruir un cierto porcentaje de la población de manera que podamos renovar el material cromosómico después de un cierto número de generaciones. Este mecanismo, llamado **ejecución**, es muy útil en dominios de alta complejidad en los cuales nuestra población puede no contener ni un solo individuo factible aún después de un gran número de generaciones.

5.6 Algoritmo Genético Estructurado

Dasgupta [61] propuso una representación que es un compromiso entre el cromosoma lineal tradicional de longitud fija y la codificación de árbol usada en la programación genética. Su técnica, denominada *Algoritmo Genético Estructurado* (stGA por sus siglas en inglés), usa una representación jerárquica con un

Figura 5.16: Una representación cromosómica de la estructura de 2 niveles del AG estructurado.

Figura 5.17: Ejemplo de una estructura de datos usada para implementar un AG estructurado.

mecanismo similar a los diploides [105], en la cual ciertos genes actúan como operadores de cambio (o dominancia) que encienden o apagan ciertos genes, a los cuales se les llama activos o pasivos, respectivamente [61].

El stGA usa una cadena cromosómica lineal, pero realmente codifica una estructura genética de varios niveles (un grafo dirigido o un árbol) tal y como se indica en la figura 5.15. Los genes en cualquier nivel pueden ser activos o pasivos, pero los genes de alto nivel activan o desactivan conjuntos de genes de más bajo nivel, lo que significa que cualquier cambio pequeño a un alto nivel se magnifica en los niveles inferiores [60]. La idea es que los genes de alto nivel deben explorar las áreas potenciales del espacio y los de bajo nivel deben explotar ese sub-espacio.

La estructura jerárquica usada por el stGA es, sin embargo, codificada como un cromosoma lineal de longitud fija, tal y como se muestra en la figura 5.16. No obstante, la estructura de datos que se requiere para implementar un stGA es ligeramente más complicada que el arreglo unidimensional que requiere un AG tradicional. La figura 5.17 muestra un ejemplo de dicha estructura de datos. En esta figura, cada gene en los niveles superiores actúa como un puntero cambiable que tiene dos estados posibles: cuando el gene está activo (encendido), apunta a

su gene de más bajo nivel y cuando es pasivo (apagado), apunta al gene del mismo nivel en que se encuentre [60].

5.7 Otras propuestas

Las representaciones anteriores no son las únicas alternativas que existen en la literatura especializada. Por ejemplo, Antonisse [4] y Gero et al. [100] han propuesto el uso de gramáticas en el contexto de lenguajes de programación y diseño en ingeniería, respectivamente. De hecho, Antonisse asegura qe su representación es más poderosa que la de Koza [145], porque define gramáticas sensibles al contexto, las cuales son más generales (en la jerarquía de lenguajes propuesta por Noam Chomsky) que las expresiones-S usadas por Koza en LISP [4]. Algunas otras representaciones más dependientes del problema, tales como la matricial [220, 21], la multi-dimensional [34, 2] y la de permutaciones [159, 115] han sido propuestas también por algunos investigadores.

5.8 Tendencias futuras

Las limitaciones de la representación binaria en algunas aplicaciones ha hecho de ésta una ruta interesante de investigación, sobre todo en el contexto de problemas de manufactura y definición de rutas, en los cuales la respuesta se expresa en forma de permutaciones. Idealmente, debiera existir un tipo de representación suficientemente flexible como para permitir su utilización en una amplia gama de problemas de forma sencilla y natural.

Filho [77] desarrolló un sistema llamado GAME (*Genetic Algorithms Manipulation Environment*), que constituye un paso importante en esta dirección. GAME usa una codificación de árbol en la que las hojas pueden ser enteros, caracteres, números reales o cadenas binarias.

Gibson [101] también propuso una codificación híbrida similar a la de Filho. Su propuesta se basa en el uso de componentes de diferentes tipos en el contexto de modelado de procesos industriales.

Existe amplia evidencia en la literatura especializada [159, 155, 192, 100, 115] de que el usar una representación adecuada puede simplificar tremendamente el proceso de búsqueda de un algoritmo genético, pero a pesar de eso, muchos investigadores suelen pasar por alto este importante hecho. Por lo tanto, es vital no únicamente reconocer que se requieren codificaciones más poderosas, sino

también saber más acerca de las representaciones (distintas a la binaria tradicional) que actualmente existen, pues su uso adecuado puede ahorrar mucho trabajo innecesario y permitirnos concentrarnos en la aplicación misma del algoritmo genético a nuestro problema en vez de canalizar todas nuestras energías en el diseño de operadores genéticos especiales.

5.9 Recomendaciones para el Diseño de una Buena Representación

Palmer [175] analizó las propiedades de las representaciones de árbol y proporcionó una serie de recomendaciones que pueden generalizarse a cualquier otro tipo de representación. Los puntos clave de la propuesta de Palmer son los siguientes:

- Una codificación debe ser capaz de representar todos los fenotipos posibles.
- Una codificación debe ser carente de sesgos en el sentido de que todos los individuos deben estar igualmente representados en el conjunto de todos los genotipos posibles.
- Una codificación no debiera codificar soluciones infactibles (esto no es normalmente posible en la mayoría de los dominios).
- La decodificación debiera ser fácil.
- Una codificación debe poseer localidad (o sea, cambios pequeños en el genotipo debieran producir cambios pequeños en el fenotipo).

Roland [191, 192], también estudió diversas representaciones y proporcionó sus propios lineamientos de diseño:

- Las codificaciones deben ajustarse a un conjunto de operadores genéticos de tal forma que los buenos bloques constructores se preserven de padres a hijos [88].
- Las codificaciones deben minimizar la epístasis [21].
- Deben preferirse las soluciones factibles en una codificación.

- El problema debiera representarse a un nivel correcto de abstracción.
- Las codificaciones deben explotar un mapeo apropiado del genotipo al fenotipo en caso de que no sea posible realizar un mapeo simple.
- Las formas isomórficas, donde el fenotipo de un individuo es codificado con más de un genotipo, debieran ser evitadas.

Nótese, sin embargo, que existe evidencia reciente que contradice el último punto de los lineamientos de Ronald, pues las representaciones redundantes parecen mejorar el desempeño de un algoritmo evolutivo [194].

5.10 Problemas propuestos

1. Suponga que desea minimizar una función de 3 variables $f(x, y, z)$. Se sabe que la variable x varía entre -10.0 y 12.5, con una precisión de 1 decimal; la variable y varía entre 0 y 100, y asume sólo valores enteros. Finalmente, la variable z varía entre -0.5 y 1.0 con una precisión de 2 decimales. Si usamos una representación binaria (tradicional), responda lo siguiente:

   a) ¿Cuál es la longitud mínima de la cadena cromosómica que puede codificar a estas 3 variables?

   b) ¿Cuál es el tamaño intrínseco del espacio de búsqueda para un AG que se usara con estas cadenas cromosómicas.

   c) ¿Cuál es la cantidad de esquemas que esta representación codifica?

   d) Muestre las cadenas cromosómicas que representen los siguientes puntos: (-8.1,50,0.0), (11.1, 98, -0.2).

2. Resuelva el problema anterior usando representación entera de punto fijo.
3. Investigue la representación de permutaciones. Discuta por qué se considera más adecuada para problemas como el del viajero. ¿Qué problemas presenta una representación binaria en este dominio? ¿Qué operadores deben definirse en una representación de permutaciones?

Capítulo 6

Técnicas de Selección

Una parte fundamental del funcionamiento de un algoritmo genético es, sin lugar a dudas, el proceso de selección de candidatos a reproducirse. En el algoritmo genético este proceso de selección suele realizarse de forma probabilística (es decir, aún los individuos menos aptos tienen una cierta oportunidad de sobrevivir), a diferencia de las estrategias evolutivas, en las que la selección es **extintiva** (los menos aptos tienen cero probabilidades de sobrevivir).

Las técnicas de selección usadas en algoritmos genéticos pueden clasificarse en tres grandes grupos:

- Selección proporcional
- Selección mediante torneo
- Selección de estado uniforme

6.1 Selección Proporcional

Este nombre describe a un grupo de esquemas de selección originalmente propuestos por Holland [127] en los cuales se eligen individuos de acuerdo a su contribución de aptitud con respecto al total de la población.

Se suelen considerar 4 grandes grupos dentro de las técnicas de selección proporcional [110]:

1. La Ruleta
2. Sobrante Estocástico

3. Universal Estocástica

4. Muestreo Determinístico

Adicionalmente, las técnicas de selección proporcional pueden tener los siguientes aditamentos:

1. Escalamiento Sigma

2. Jerarquías

3. Selección de Boltzmann

6.1.1 La Ruleta

Esta técnica fue propuesta por DeJong [137], y ha sido el método más comúnmente usado desde los orígenes de los algoritmos genéticos. El algoritmo es simple, pero ineficiente (su complejidad es $O(n^2)$. Asimismo, presenta el problema de que el individuo menos apto puede ser seleccionado más de una vez. Sin embargo, buena parte de su popularidad se debe no sólo a su simplicidad, sino al hecho de que su implementación se incluye en el libro clásico sobre AGs de David Goldberg [105].

El algoritmo de la Ruleta (de acuerdo a DeJong [137]) es el siguiente:

- Calcular la suma de valores esperados T
- Repetir N veces (N es el tamaño de la población):
  - Generar un número aleatorio r entre 0.0 y T
  - Ciclar a través de los individuos de la población sumando los valores esperados hasta que la suma sea mayor o igual a r.
  - El individuo que haga que esta suma exceda el límite es el seleccionado.

Veamos ahora un ejemplo:

	aptitud	Ve
(1)	25	0.35
(2)	81	1.13

(3)	36	0.51
(4)	144	2.01
	$\sum = 286$	$\sum = 4.00$

$\bar{f} = \frac{286}{4} = 71.5$ $\quad Ve_i = \frac{f_i}{\bar{f}}$

$T =$ Suma de Ve

$r \in [0.0, T]$

Generar $r \in [0.0, 4.0]$

r = 1.3

(ind1) suma = $0.35 < r$
(ind2) suma = $1.48 > r$

Seleccionar a ind2

En este ejemplo, Ve se refiere al valor esperado (o número esperado de copias que se esperan obtener) de un individuo.

6.1.1.1 Análisis de la Ruleta

Problemas: diferencias entre Ve y el valor real (o sea, el verdadero número de copias obtenidas). El peor individuo puede seleccionarse varias veces.

Complejidad: $O(n^2)$. El algoritmo se torna ineficiente conforme crece n (tamaño de la población).

Mejoras posibles: Uso de búsqueda binaria en vez de búsqueda secuencial para localizar la posición correcta de la rueda. Esto requiere memoria extra y un recorrido $O(n)$ para calcular los totales acumulados por cada segmento de la rueda. Pero, en contraste, la complejidad total se reduce a $O(n \log n)$.

6.1.2 Sobrante Estocástico

Propuesta por Booker [24] y Brindle [32] como una alternativa para aproximarse más a los valores esperados ($Valesp$) de los individuos:

$$Valesp_i = \frac{f_i}{\bar{f}}$$

La idea principal es asignar determinísticamente las partes enteras de los valores esperados para cada individuo y luego usar otro esquema (proporcional) para la parte fraccionaria.

El sobrante estocástico reduce los problemas de la ruleta, pero puede causar convergencia prematura al introducir una mayor presión de selección.

El algoritmo es el siguiente:

1. Asignar de manera determinística el conteo de valores esperados a cada individuo (valores enteros).
2. Los valores restantes (sobrantes del redondeo) se usan probabilísticamente para rellenar la población.

Hay 2 variantes principales:

- **Sin reemplazo**: Cada sobrante se usa para sesgar el tiro de una moneda que determina si una cadena se selecciona de nuevo o no.
- **Con reemplazo**: Los sobrantes se usan para dimensionar los segmentos de una ruleta y se usa esta técnica de manera tradicional.

Veamos un ejemplo:

	Cadena	aptitud	e_i	enteros	dif
(1)	110100	220	1.23	1	0.23
(2)	011010	140	0.78	0	0.78
(3)	111001	315	1.76	1	0.76
(4)	001101	42	0.23	0	0.23
		$\sum = 717$	$\sum = 4.00$	$\sum = 2$	

$$\bar{f} = 179.25$$

Padres: 1 y 3 (partes enteras)

Sin reemplazo:

flip(0.23) $\longrightarrow$ ind 1

flip(0.78) $\longrightarrow$ ind 2

$\vdots$

flip(0.23) $\longrightarrow$ ind 4

Proceder hasta tener el número de padres requeridos.

Con reemplazo:

Armar una ruleta

	e_i	% del total	
(1)	0.23	0.12	(12%)
(2)	0.78	0.39	(39%)
(3)	0.76	0.38	(38%)
(4)	0.23	0.11	(11%)
	$\sum = 2.0$	$\sum = 1.00$	

6.1.2.1 Análisis del Sobrante Estocástico

Complejidad:

Versión con reemplazo:	$O(n^2)$
Versión sin reemplazo:	$O(n)$

La más popular es la versión sin reemplazo, la cual parece ser superior a la ruleta [24].

Tal y como mencionamos anteriormente, el sobrante estocástico reduce los problemas de la ruleta, pero puede producir convergencia prematura al introducir una mayor presión de selección (producto de la asignación determinística de los valores esperados de cada individuo).

6.1.3 Universal Estocástica

Propuesta por Baker [14] con el objetivo de minimizar la mala distribución de los individuos en la población en función de sus valores esperados.

El algoritmo es el siguiente:

ptr=Rand(); /* regresa un número aleatorio entre 0 y 1 */

for(sum=0,i=1;i<= n;i++)
for(sum+=$Valesp(i,t)$;sum>ptr;ptr++)
Seleccionar(i);

Ejemplo:

	Cadena	aptitud	e_i
(1)	110100	220	1.23
(2)	011010	140	0.78
(3)	111001	315	1.76
(4)	001101	42	0.23
			$\sum = 4.00$

ptr=0.4 inicialización
sum=0.0

i=1 sum=1.23 1.23>ptr
Seleccionar (1)

ptr=1.4
sum=1.23 1.23<ptr (termina ciclo)

i=2 sum=1.23 ptr=1.4
sum=2.01 2.01>ptr
Seleccionar (2)

ptr=2.4
sum=2.01 2.01<ptr (termina ciclo)

i=3 sum=3.77 3.77>ptr
Seleccionar (3)

ptr=3.4
sum=3.77 3.77>ptr
Seleccionar (3)

Padres: (1), (2), (3), (3)

6.1.3.1 Análisis de la selección universal estocástica

Complejidad: $O(n)$.

Problemas:

- Puede ocasionar convergencia prematura.
- Hace que los individuos más aptos se multipliquen muy rápidamente.
- No resuelve el problema más serio de la selección proporcional (o sea,la imprecisión entre los valores esperados y los números de copias de cada individuo que son realmente seleccionados).

6.1.4 Muestreo Determinístico

Es una variante de la selección proporcional con la que experimentó DeJong [137]. Es similar al sobrante estocástico, pero requiere un algoritmo de ordenación.

El algoritmo del muestreo determinístico es el siguiente:

- Calcular $P_{select} = \frac{f_i}{\sum f}$
- Calcular $Valesp_i = P_{select} * n$ (n=tamaño de la población)
- Asignar determinísticamente la parte entera de $Valesp_i$.
- Ordenar la población de acuerdo a las partes decimales (de mayor a menor).
- Obtener los padres faltantes de la parte superior de la lista.

Ejemplo:

	aptitud	P_{select}	e_i	enteros	ordenar fracciones
(1)	220	0.3068	1.23	1	0.78 (2)
(2)	140	0.1953	0.78	0	0.76 (3)
(3)	315	0.4393	1.76	1	0.23 (1)
(4)	42	0.0586	0.23	0	0.23 (4)

$\sum = 717 \qquad \sum = 1.000 \quad \sum = 4.00 \sum = 2$

Seleccionar: 1 y 3 (enteros)
Seleccionar: 2 y 3 (fracciones)

6.1.4.1 Análisis del muestreo determinístico

Complejidad: El algoritmo es $O(n)$ para la asignación determinística y es $O(n \log n)$ para la ordenación.

Problemas: Padece de los mismos problemas que el sobrante estocástico.

6.1.5 Escalamiento Sigma

Es una técnica ideada para mapear la aptitud original de un individuo con su valor esperado de manera que el AG sea menos susceptible a la convergencia prematura. La idea principal de esta técnica es mantener más o menos constante la **presión de selección** a lo largo del proceso evolutivo.

Usando esta técnica, el valor esperado de un individuo está en función de su aptitud, la media de la población y la desviación estándar de la población:

$$Valesp(i,t) = \begin{cases} 1 + \frac{f(i) - \bar{f}(t)}{2\sigma(t)} & \text{si } \sigma(t) \neq 0 \\ 1.0 & \text{si } \sigma(t) = 0 \end{cases} \tag{6.1}$$

$$\sigma(t) = \sqrt{\frac{n \sum f(t)^2 - (\sum f(t))^2}{n^2}} \tag{6.2}$$

Veamos un ejemplo de su funcionamiento:

n= tamaño de la población

Si $Valesp(i,t) < 0$ puede hacerse $Valesp(i,t) = 0.1$

Ejemplo:

	aptitud	$f_i(t)^2$	$Valesp$
(1)	220	48,400	1.20
(2)	140	19,600	0.81
(3)	315	99,225	1.67
(4)	42	1,764	0.32

$$\sum = 717 \qquad \sum = 168,989 \quad \sum = 4.00$$

$$\bar{f} = 179.25 \qquad \text{n=4}$$
$$(\sum f_i)^2 = 514,089$$

$$\sigma = \sqrt{\frac{4(168,989) - 514,089}{16}} = 100.5817$$

$$\sigma \neq 0 \qquad\qquad 1+\frac{f_i - \bar{f}}{2\sigma}$$

Observación: Bajó un poco el valor esperado del mejor individuo y subió el del peor.

6.1.5.1 Análisis del escalamiento sigma

El escalamiento sigma produce el siguiente comportamiento en la selección:

- Al inicio de una corrida, el valor alto de la desviación estándar impedirá que los mejores individuos obtengan los segmentos más grandes de la ruleta.
- Hacia el final, la desviación estándar será más baja y los individuos más aptos podrán muliplicarse más fácilmente.

6.1.6 Selección por Jerarquías

Propuesta por Baker [13] para evitar la convergencia prematura en las técnicas de selección proporcional. El objetivo de esta técnica es disminuir la presión de selección. En este caso, discutiremos el uso de jerarquías lineales, pero es posible también usar jerarquías no lineales, aunque la presión de selección sufre cambios más abruptos al usarse esta última.

Los individuos se clasifican con base en su aptitud, y se les selecciona con base en su rango (o jerarquía) y no con base en su aptitud. El uso de jerarquías hace que no se requiera escalar la aptitud, puesto que las diferencias entre las aptitudes absolutas se diluyen. Asimismo, las jerarquías previenen la convergencia prematura (de hecho, lo que hacen, es alentar la velocidad convergencia del algoritmo genético).

El algoritmo de las jerarquías lineales es el siguiente:

- Ordenar (o jerarquizar) la población con base en su aptitud, de 1 a N (donde 1 representa al menos apto).

- Elegir $Max(1 \leq Max \leq 2)$
- Calcular $Min = 2 - Max$
- El valor esperado de cada individuo será:
  $Valesp(i,t) = Min + (Max - Min)\frac{jerarquía(i,t)-1}{N-1}$
  Baker recomendó $Max = 1.1$
- Usar selección proporcional aplicando los valores esperados obtenidos de la expresión anterior.

Ejemplo: (jerarquía menor a la aptitud más baja)

	aptitud	jerarquías	$Valesp$	
(1)	12	2	0.95	
(2)	245	5	1.10	Aplicar ruleta
(3)	9	1	0.90	u otra técnica
(4)	194	4	1.05	proporcional
(5)	48	3	1.00	
			$\sum = 5.00$	

$Max = 1.1 \quad Min = 2 - 1.1 = 0.9 \quad N = 5$

$Valesp = 0.9 + (0.2)\frac{jerarquía_i - 1}{N-1}$

6.1.6.1 Análisis de las jerarquías lineales

Complejidad: $O(nlogn)$ + tiempo de selección.

Algunos puntos interesantes respecto a la aplicabilidad de esta técnica:

- Es útil cuando la función tiene ruido (p.ej., cuando hay una variable aleatoria).
- Diluye la presión de la selección, por lo que causa convergencia más lenta.
- Existen otros métodos de asignación de jerarquías además del lineal (p. ej., exponencial).
- Puede alentar sobremanera la convergencia del algoritmo genético, por lo que su uso suele limitarse a situaciones en las que el AG convergería prematuramente en caso de no aplicarse.

6.1.7 Selección de Boltzmann

Esta técnica fue propuesta por Goldberg [107] y está basada en el recocido simulado [142]: la idea es usar una función de variación de "temperatura" que controle la presión de selección. Se usa un valor alto de temperatura al principio, lo cual hace que la presión de selección sea baja. Con el paso de las generaciones, la temperatura disminuye, lo que aumenta la presión de selección. De esta manera se incita a un comportamiento exploratorio en las primeras generaciones y se acota a uno más explotatorio hacia el final del proceso evolutivo.

El algoritmo es el siguiente:

Típicamente, se usa la siguiente expresión para calcular el valor esperado de un individuo.

$$\text{Valesp(i,t)} = \frac{e^{\frac{f_i}{T}}}{\langle e^{\frac{f_i}{T}} \rangle_t} \tag{6.3}$$

donde T es la temperatura y $\langle\rangle_t$ denota el promedio de la población en la generación t.

Veamos un ejemplo de su uso:

Ejemplo:

aptitud	$e^{\frac{f_i}{T(t)}}$	$Valesp$	
220	9.025	0.952	(bajó)
140	4.055	0.428	(bajó)
315	23.336	2.460	(subió)
42	1.522	0.160	(bajó)
$\sum = 717$	$\sum = 37.938$	$\sum = 4.000$	

$$\langle\rangle_t = \frac{37.938}{4} = 9.4845$$

$T(0) = 100000$
$T(t) = T(t-1) * 0.1$ Problemas: rango de exp()

Supongamos: $t = 3$

$$T(3) = 100000 * 0.1 * 0.1 * 0.1 = 100$$

6.1.7.1 Análisis de la selección de Boltzmann

- Se ha utilizado más para optimización multimodal y multiobjetivo (formación de nichos).
- Existen pruebas de convergencia de la técnica hacia el óptimo global.
- Tiene el inconveniente de requerir la definición de la función de variación de temperatura.

6.2 Selección Mediante Torneo

Los métodos de selección proporcional antes descritos requieren de dos pasos a través de toda la población en cada generación:

1. Calcular la aptitud media (y, si se usa escalamiento sigma, la desviación estándar).
2. Calcular el valor esperado de cada individuo.

El uso de jerarquías requiere que se ordene toda la población (una operación cuyo costo puede volverse significativo en poblaciones grandes).

La selección mediante torneo es similar a la de jerarquías en términos de la presión de selección, pero es computacionalmente más adecuada para implementarse en paralelo.

Esta técnica fue propuesta por Wetzel [225] y estudiada en la tesis doctoral de Brindle [32].

La idea básica del método es seleccionar con base en comparaciones directas de los individuos.

Hay 2 versiones de la selección mediante torneo:

- Determinística
- Probabilística

El algoritmo de la versión determinística es el siguiente:

- Barajar los individuos de la población.
- Escoger un número p de individuos (típicamente 2).

- Compararlos con base en su aptitud.
- El ganador del "torneo" es el individuo más apto.
- Debe barajarse la población un total de p veces para seleccionar N padres (donde N es el tamaño de la población).

Veamos un ejemplo de su funcionamiento:

Orden	Aptitud	Barajar	Ganadores
(1)	254	(2)	
(2)	47	(6)	(6)
(3)	457	(1)	
(4)	194	(3)	(3)
(5)	85	(5)	
(6)	310	(4)	(4)

Barajar	Ganadores
(4)	
(1)	(1)
(6)	
(5)	(6)
(2)	
(3)	(3)

Padres:

(6) y (1), (3) y (6), (4) y (3)

Otra forma:

$r = rnd(0, 1)$ /*aleatorio real*/

		Ganador
Selecciona	$t1 = r = (3)$	
	$t2 = r = (6)$	(3)
Selecciona	$t1 = r = (1)$	
	$t2 = r = (4)$	(1)

Y así sucesivamente.

El algoritmo de la versión probabilística es idéntico al anterior, excepto por el paso en que se escoge al ganador. En vez de seleccionar siempre al individuo con aptitud más alta, se aplica $flip(p)$[1] y si el resultado es cierto, se selecciona al más apto. De lo contrario, se selecciona al menos apto.

El valor de p permanece fijo a lo largo de todo el proceso evolutivo y se escoge en el siguiente rango:

$$0.5 < p \leq 1$$

Observe que si $p = 1$, la técnica se reduce a la versión determinística.

Ejemplo: ($p = 0.7$)

	Aptitud	Barajar	
(1)	254	(2)	
(2)	47	(6)	$flip(p) = TRUE$ — elige (6)
(3)	457	(1)	
(4)	194	(3)	$flip(p) = FALSE$ — elige (1)
(5)	85	(5)	
(6)	310	(4)	$flip(p) = TRUE$ — elige (4)

Esta variante reduce un poco la presión de selección, permitiendo que en algunas cuantas ocasiones el individuo menos apto gane el torneo.

6.2.1 Análisis de la selección mediante torneo

- La versión determinística garantiza que el mejor individuo será seleccionado p veces (tam_torneo).
- Complejidad:
    1. Cada competencia requiere la selección aleatoria de un número constante de individuos de la población. Esta comparación puede realizarse en $O(1)$.

[1] $flip(p)$ devuelve cierto con una probabilidad p.

2. Se requieren "n" competencias de este tipo para completar una generación.
3. Por lo tanto, el algoritmo es $O(n)$.

- La técnica eficiente y fácil de implementar.
- No requiere escalamiento de la función de aptitud (usa comparaciones directas).
- Puede introducir una presión de selección muy alta (en la versión determinística) porque a los individuos menos aptos no se les da oportunidad de sobrevivir.
- ¿Tiene sentido aplicar jerarquías lineales a la selección mediante torneo? No, porque la selección mediante torneo realiza comparaciones directas entre individuos. El uso de jerarquías lineales no cambiaría en nada la presión de selección.

Puede regularse la presión de selección variando el tamaño del torneo:

- Si se usa tam_torneo=1, se produce una caminata aleatoria con una presión de selección muy baja.
- Si se usa tam_torneo=∞, la selección se vuelve totalmente determinística (los mejores individuos globales son seleccionados). A esto se le llama "elitismo global".
- Si se usa tam_torneo$\geq$10, la selección se considera "dura".
- Si se unsa tam_torneo entre 2 y 5, la selección se considera "blanda".

6.3 Selección de Estado Uniforme

Esta técnica fue propuesta por Whitley [228] y se usa en AGs no generacionales, en los cuales sólo unos cuantos individuos son reemplazados en cada generación (los menos aptos).

Esta técnica suele usarse cuando se evolucionan sistemas basados en reglas (p.ej., sistemas clasificadores) en los que el aprendizaje es incremental.

En general, la técnica resulta útil cuando los miembros de la población resuelven colectivamente (y no de manera individual) un problema.

Asimismo, los AGs generacionales se usan cuando es importante "recordar" lo que se ha aprendido antes.

El algoritmo de la selección de estado uniforme es el siguiente:

- Llamaremos G a la población original de un AG.
- Seleccionar R individuos ($1 \leq R < M$) de entre los más aptos. Por ejemplo, $R = 2$.
- Efectuar cruza y mutación a los R individuos seleccionados. Llamaremos H a los hijos.
- Elegir al mejor individuo en H. (o a los μ mejores).
- Reemplazar los μ peores individuos de G por los μ mejores individuos de H.

6.3.1 Análisis de la Selección de Estado Uniforme

- Mecanismo especializado de selección.
- Su complejidad (en la variante incluida en GENITOR, la cual usa jerarquías lineales) es $O(nlogn)$.
- Los AGs no generacionales no son muy comunes en aplicaciones de optimización, aunque sí pueden utilizarse.

6.4 Brecha Generacional

Muy ligado a la selección de estado uniforme se encuentra el concepto de "brecha generacional" (*generation gap*).

Para explicar este concepto, es importante reconocer en primer término que las poblaciones pueden ser "no traslapables" (*nonoverlapping*) o "traslapables" (*overlapping*).

Una población "no traslapable" es aquella en la que los padres nunca compiten contra sus hijos. Por el contrario, en una población "traslapable", los padres compiten contra sus hijos.

Se denomina “brecha generacional” a la cantidad de traslape existente entre padres e hijos. Una brecha generacional grande implica poco (o ningún) traslape poblacional y viceversa.

Históricamente, la programación evolutiva y las estrategias evolutivas han usado poblaciones “traslapables”, mientras que los AGs han usado poblaciones “no traslapables”.

DeJong [137] parece haber sido el primero en estudiar los AGs con poblaciones traslapables.

En su tesis doctoral, DeJong sugirió que las ventajas de las poblaciones traslapables se diluían debido a los efectos negativos del “desvío genético”.

Más tarde, Grefenstette [116] confirmaría que una brecha generacional alta parecía mejorar el desempeño del AG.

Los primeros experimentos con los “sistemas clasificadores”, confirmarían, sin embargo, un comportamiento exactamente opuesto [127].

En los sistemas clasificadores, el desempeño del AG parecía degradarse conforme se aumentaba la brecha generacional.

Algunos investigadores atribuyen los resultados de DeJong y Grefenstette al uso de poblaciones pequeñas.

Los AGs tradicionales siguen usando, sin embargo, poblaciones no traslapables.

Los algoritmos evolutivos de **estado uniforme** son aquellos en los que la población es traslapable. Normalmente, sólo uno o dos hijos se producen en cada iteración de un AE de estado uniforme.

6.5 Otras Técnicas de Selección

Hay varias técnicas más de selección cuyo uso es menos común [69]. Discutiremos brevemente las siguientes:

- Disruptiva
- Jerarquías no lineales
- Competitiva

6.5.1 Selección Disruptiva

Esta técnica fue sugerida por Kuo y Hwuang [149] para normalizar aptitudes con respecto a un cierto valor moderado (en vez de usar valores extremos).

Suele usarse:

$$f'_i(x) = |f_i(x) - \bar{f}(t)|$$

donde $\bar{f}(t)$ se refiere a la aptitud media de la población.

$$Valesp_i = \frac{f'_i(x)}{\bar{f}'(t)}$$

$\bar{f}'(x)$ son los valores normalizados.

Ejemplo:

Aptitud	Normalización (f')	Valesp	
220	40.75	0.462	
140	39.25	0.445	
315	135.75	1.538	extremos
42	137.25	1.555	más aptos
$\sum = 717$	$\sum = 353$	$\sum = 4.000$	

$\bar{f}(t) = \frac{717}{4} = 179.25$, $\bar{f}'(t) = \frac{353}{4} = 88.25$

$$Valesp = \frac{f'}{\bar{f}'(t)}$$

6.5.1.1 Análisis de la selección disruptiva

La principal motivación de la selección disruptiva es distribuir más los esfuerzos de la búsqueda hacia las soluciones extremadamente buenas y extremadamente malas. Los individuos cercanos a la media son desechados.

La utilidad de este método es altamente dependiente en la aplicación. Suele usársele más frecuentemente con funciones de aptitud dinámicas.

6.5.2 Jerarquías No Lineales

Esta técnica fue propuesta por Michalewicz [161]. Se usa:

$$Prob_i = q(1-q)^{jerarquía_i - 1}$$

donde:

$Prob_i$ es la probabilidad de que el individuo i sea seleccionado.

$q \in [0..1]$ es el factor de presión de selección.
$jerarquía_i$ es la jerarquía del individuo i en estudio.

Al igual que con las jerarquías lineales, se asigna la jerarquía más baja al peor individuo y la más alta al mejor.

Una vez que se conocen las probabilidades de que un individuo sea seleccionado, obtenemos $Valesp$ multiplicando por n.

Veamos un ejemplo:

aptitud	jerarquía	$Prob_{selecc}$	$Valesp$	
12	2	0.210	1.05	(mayor)
245	5	0.072	0.36	(menor)
9	1	0.300	1.50	(mayor)
194	4	0.103	0.52	(menor)
48	3	0.147	0.74	(menor)
		$\sum = 0.832$	$\sum = 4.17$	

Supongamos: $q = 0.3$

$Prob_i = 0.3(1 - 0.3)^{jerarquía_i - 1}$

$Valesp = Prob_i * n$

$n = 5$

Teniendo los valores esperados, puede usarse cualquier técnica de selección proporcional, al igual que en el caso de las jerarquías lineales.

6.5.2.1 Análisis de las jerarquías no lineales

Thomas Bäck [8] advirtió sobre el problema que tenemos en el ejemplo presentado anteriormente: las probabilidades obtenidas con este método no suman uno.

Bäck [8] también advirtió que esta técnica puede hacerse prácticamente idéntica al torneo dependiendo del valor de q que se use. Valores grandes de q implican una mayor presión de selección inversa (o sea, implica darle mayor oportunidad al individuo menos apto de reproducirse).

Michalewicz [161] advierte que la suma de probabilidades en su técnica puede hacerse igual a uno si usamos:

$$Prob_i = c * q(1-q)^{jerarquía_i - 1}$$

donde:

$c = \frac{1}{1-(1-q)^M}$

M= tamaño de la población

Ejemplo:

aptitud	jerarquía	P_{selecc}	$Valesp$	
12	2	0.2524	1.2620	(mayor)
245	5	0.0865	0.4325	(menor)
9	1	0.3606	1.8030	(mayor)
194	4	0.1238	0.6190	(menor)
48	3	0.1767	0.8835	(menor)
		$\sum = 1.000$	$\sum = 5.000$	

Usando de nuevo: $q = 0.3$

$M = 5$

$c = \frac{1}{1-(1-0.3)^5} = 1.202024209$

Cabe mencionar que se han usado también otras variantes de las jerarquías no lineales. Por ejemplo, Whitley & Kauth [226] propusieron el uso de una distribución geométrica de acuerdo a la cual la probabilidad de que un individuo i sea seleccionado está dada por:

$$p_i = a(1-a)^{n-r_i} \tag{6.4}$$

donde $a \in [0..1]$ es un parámetro definido por el usuario, n es el tamaño de la población y r_i es la jerarquía del individuo i (definida de tal forma que el menos apto tiene una jerarquía de 1 y el más apto tiene una jerarquía de n). Si el individuo i es el mejor en la población, entonces $r_i = n$, con lo que p_i se haría a. Por lo tanto, a es realmente la probabilidad de que seleccionar al mejor individuo en la población [69].

6.5.3 Selección Competitiva

En este caso, la aptitud de un individuo se determina mediante sus interacciones con otros miembros de la población, o con otros miembros de una población separada que evoluciona concurrentemente.

Esta técnica ha sido usada por Hillis [122], Angeline & Pollack [3] y Sebald & Schlenzig [206].

En la versión con 2 poblaciones puede verse como un esquema co-evolutivo: las aptitudes de dos individuos dependen mutuamente entre sí (uno del otro). Por ejemplo, una población puede contener soluciones a un problema de optimización, mientras la otra contiene las restricciones del mismo.

6.6 Clasificaciones de Técnicas de Selección

Bäck y Hoffmeister [11] distinguen entre:

1. **Métodos Estáticos**: Requieren que las probabilidades de selección permanezcan constantes entre generaciones.

   Ejemplo: jerarquías lineales.

2. **Métodos Dinámicos**: No se requiere que las probabilidades de selección permanezcan constantes.

   Ejemplo: selección proporcional.

Otros investigadores distinguen entre:

1. **Selección Preservativa**: Requiere una probabilidad de selección distinta de cero para todos los individuo.

   Ejemplo: selección proporcional.

2. **Selección Extintiva**: Puede asignar una probabilidad de selección de cero a algún individuo.

   Ejemplo: torneo determinístico.

A su vez, las técnicas **extintivas** se dividen en:

1. **Selección Izquierda**: Se impide a los mejores individuos reproducirse a fin de evitar convergencia prematura.

2. **Selección Derecha**: No se tiene control explícito sobre la capacidad reproductiva de los individuos más aptos.

Adicionalmente, algunas técnicas de selección son **puras** en el sentido de que a los padres se les permite reproducirse solamente en una generación (es decir, el tiempo de vida de cada individuo está limitado a sólo una generación, independientemente de su aptitud).

6.7 Problemas Propuestos

1. Implemente un algoritmo genético simple (cruza de un punto, codificación binaria, mutación uniforme) en C/C++ y pruébelo optimizando $f(x) = x^2$.

Capítulo 7

Técnicas de Cruza

7.1 Introducción

En los sistemas biológicos, la cruza es un proceso complejo que ocurre entre parejas de cromosomas. Estos cromosomas se alinean, luego se fraccionan en ciertas partes y posteriormente intercambian fragmentos entre sí.

En computación evolutiva se simula la cruza intercambiando segmentos de cadenas lineales de longitud fija (los cromosomas).

Aunque las técnicas de cruza básicas suelen aplicarse a la representación binaria, éstas son generalizables a alfabetos de cardinalidad mayor, si bien en algunos casos requieren de ciertas modificaciones.

Comenzaremos por revisar las tres técnicas básicas de cruza:

- Cruza de un punto
- Cruza de dos puntos
- Cruza uniforme

7.1.1 Cruza de un punto

Esta técnica fue propuesta por Holland [127], y fue muy popular durante muchos años. Hoy en día, sin embargo, no suele usarse mucho en la práctica debido a sus inconvenientes. Puede demostrarse, por ejemplo, que hay varios esquemas que no pueden formarse bajo esta técnica de cruza.

Considere, por ejemplo, los esquemas siguientes:

Figura 7.1: Cruza de un punto.

H1 = 11*****1
H2 = ****11**

Si aplicamos cruza de un punto a estos 2 esquemas, no hay manera de formar una instancia del esquema:

H = 11**11*1

7.1.1.1 Orden de un esquema

Definamos ahora O como el orden de un esquema:

O(H) = número de posiciones fijas en el esquema H.

Ejemplo: O(*11*0*0*) = 4
O(****1***) = 1

7.1.1.2 Longitud de definición

Definamos a δ como la longitud de definición:

$\delta(H)$ = distancia entre la primera y la última posición fija de un esquema H.

Ejemplo: $\delta(*11*0*0*) = 7 - 2 = 5$
$\delta(**1******) = 0$

7.1.1.3 Análisis de la cruza de un punto

La cruza de un punto destruye esquemas en los que la longitud de definición es alta. Esto produce el denominado **sesgo posicional**: los esquemas que pueden crearse o destruirse por la cruza dependen fuertemente de la localización de los bits en el cromosoma [74].

El problema fundamental de la cruza de un punto es que presupone que los bloques constructores son esquemas cortos y de bajo orden, y cuando esto no sucede (p.ej., con cadenas largas), suele no proporcionar resultados apropiados.

Obviamente, las aplicaciones del mundo real suelen requerir cadenas largas y de ahí que esta cruza no se use comúnmente en dichos problemas.

La cruza de un punto trata también preferencialmente algunas posiciones del cromosoma, como por ejemplo los extremos de una cadena.

La cruza de un punto suele preservar también los *hitchhikers*, que son bits que no son parte del esquema deseado, pero que debido a su similitud con ellos gozan de los beneficios de la cruza.

7.1.2 Cruza de dos puntos

DeJong [137] fue el primero en implementar una cruza de n puntos, como una generalización de la cruza de un punto.

El valor $n = 2$ es el que minimiza los efectos disruptivos (o destructivos) de la cruza y de ahí que sea usado con gran frecuencia.

No existe consenso en torno al uso de valores para n que sean mayores o iguales a 3.

Los estudios empíricos al respecto [137, 74] proporcionan resultados que no resultan concluyentes respecto a las ventajas o desventajas de usar dichos valores.

En general, sin embargo, es aceptado que la cruza de dos puntos es mejor que la cruza de un punto.

Asimismo, el incrementar el valor de n se asocia con un mayor efecto disruptivo de la cruza.

Figura 7.2: Cruza de dos puntos.

Figura 7.3: Cruza uniforme.

7.1.3 Cruza uniforme

Esta técnica fue propuesta originalmente por Ackley [1], aunque se le suele atribuir a Syswerda [217].

En este caso, se trata de una cruza de n puntos, pero en la cual el número de puntos de cruza no se fija previamente.

La cruza uniforme tiene un mayor efecto disruptivo que cualquiera de las 2 cruzas anteriores. A fin de evitar un efecto excesivamente disruptivo, suele usarse con $Pc = 0.5$.

Algunos investigadores, sin embargo, sugieren usar valores más pequeños de Pc [209].

Cuando se usa $Pc = 0.5$, hay una alta probabilidad de que todo tipo de cadena binaria de longitud L sea generada como máscara de copiado de bits.

7.1.4 Cruza Acentuada

Esta técnica fue propuesta por Schaffer y Morishima [201], en un intento por implementar un mecanismo de auto-adaptación para la generación de los patrones favorables (o sea, los buenos bloques constructores) de la cruza.

En vez de calcular directamente la máscara (o patrón) de cruza, la idea es usar una cadena binaria de "marcas" para indicar la localización de los puntos de cruza.

La idea fue sugerida por Holland [127], aunque en un sentido distinto.

La información extra que genera la cruza acentuada se agrega al cromosoma de manera que el número y localizaciones de los puntos de cruza pueda ser objeto de manipulación por el AG.

Por tanto, las cadenas tendrán una longitud del doble de su tamaño original.

La convención que suele adoptarse es la de marcar con '1' las posiciones donde hay cruza y con '0' las posiciones donde no la hay.

Asimismo, se suelen usar signos de admiración para facilitar la escritura de las cadenas.

La figura 7.4 muestra un ejemplo de la cruza acentuada.

El algoritmo de la cruza acentuada es el siguiente:

- Copiar los bits de cada padre hacia sus hijos, de uno en uno.
- En el momento en que se encuentra un signo de admiración en cualquiera de los padres, se efectúa la cruza (es decir, se invierte la procedencia de los bits en los hijos).

Figura 7.4: Cruza Acentuada.

- Cuando esto ocurre, los signos de admiración se copian también a los hijos, justo antes de que la cruza se efectúe.

Veamos el siguiente ejemplo:
Antes de la cruza:

P1 = a a a a a a a! b b b b b b b
P2 = c c c c! d d d d d d! e e e e

Después de la cruza:

H1 = a a a a d d d b b b e e e e
H2 = c c c c! a a a! d d d! b b b b

7.1.4.1 Observaciones sobre la cruza acentuada

Nótese que sólo se usa la primera parte de la cadena para calcular la aptitud, pero se espera que la selección, cruza y mutación tengan un efecto positivo sobre los puntos de cruza.

En esta representación, la mutación actúa sobre los dos segmentos cromosómicos.

Las probabilidades de que aparezcan unos en el segundo segmento se determinan de manera distinta a las del primero.

La cruza acentuada reportó buenos resultados en un pequeño conjunto de funciones de prueba [201]. Sin embargo, no hay evidencia contundente acerca de su efectividad.

La cruza acentuada tiene una buena inspiración biológica, porque estas marcas de cruza efectivamente existen en la naturaleza y se co-evolucionan junto con los cromosomas.

7.2 Sesgos de la Cruza

El "sesgo" de la cruza se refiere a las tendencias de este operador hacia favorecer o no un cierto tipo de búsqueda.

Realmente, la búsqueda aleatoria es la única que no presenta ningún tipo de sesgo, por lo que es de esperarse que cualquier operador heurístico como la cruza presente algún tipo de sesgo.

De hecho, desde hace algún tiempo, se ha determinado que se requiere de algún tipo de sesgo para que una técnica de búsqueda sea efectiva [170].

En algoritmos genéticos, se suelen considerar 2 tipos de sesgo para la cruza:

- Distribucional
- Posicional

7.2.1 Sesgo distribucional

El **sesgo distribucional** se refiere al número de símbolos transmitidos durante una recombinación. Asimismo, se refiere a la medida en la que algunas cantidades tienen más tendencia a ocurrir que otras.

El **sesgo distribucional** es importante porque está correlacionado con el número potencial de esquemas de cada padre que pueden ser recombinados por el operador de cruza.

- La cruza de un punto y la de dos puntos no tienen sesgo distribucional.
- La cruza de n puntos ($n > 2$) tiene un sesgo distribucional moderado.
- La cruza uniforme tiene un sesgo distribucional muy fuerte.

7.2.2 Sesgo posicional

El **sesgo posicional** caracteriza en qué medida la probabilidad de que un conjunto de símbolos se transmitan intactos durante la recombinación depende de las posiciones relativas de los mismos en el cromosoma.

El **sesgo posicional** es importante porque indica qué esquemas es más probable que se hereden de padres a hijos. Asimismo, también indica la medida en la que estos esquemas aparecerán en nuevos contextos.

- La cruza de un punto tiene un fuerte sesgo posicional.
- Todo parece indicar, que la cruza de n puntos tiene también un sesgo posicional fuerte, aunque éste varía en función de n.
- La cruza uniforme no tiene sesgo posicional.

7.3 Variantes de la Cruza

En la práctica, diversos aspectos de la cruza suelen modificarse para mejorar su desempeño. Una variante, por ejemplo, consiste en retener sólo a uno de los dos hijos producidos por una cruza sexual. Holland [127] describió una técnica de este tipo.

Estudios empíricos han mostrado, sin embargo, que retener a los 2 hijos producidos por una cruza sexual reduce sustancialmente la pérdida de diversidad en la población [24].

Otra variante muy común es la de restringir los puntos de cruza a aquellas posiciones en las que los padres difieran. A esta técnica se le conoce como **sustitución reducida** [25].

El objetivo de la sustitución reducida es mejorar la capacidad de la cruza para producir hijos que sean distintos a sus padres.

Otra variante interesante es la llamada **cruza con barajeo** [74]. En este caso, se aplica un operador de permutación a una parte de las cadenas de los padres antes de efectuar la cruza.Después de la cruza, se aplica la permutación inversa a fin de restaurar el orden original de los bits.

La **cruza con barajeo** tiene como objeto contrarrestar la tendencia de la cruza de n puntos ($n \geq 1$) a causar con más frecuencia disrupción en los conjuntos de bits que están dispersos que en los que están juntos.

7.4 Comportamiento Deseable de la Cruza

Todos estos operadores descritos anteriormente, siguen el principio Mendeliano de la herencia: cada gene que tiene un hijo, es una copia de un gene heredado de alguno de sus padres. Cabe mencionar, sin embargo, que esto no tiene que ser así en computación evolutiva.

Algunos investigadores han destacado que el énfasis de la cruza debe ser el poder generar todas las posibles combinaciones de bits (de longitud L) que hayan en el espacio de búsqueda del problema [178]. Dada una cierta representación binaria, ni la cruza de un punto, ni la de n puntos son capaces de lograr esto (generar cualquier combinación de bits posible). La cruza uniforme, sin embargo, sí puede hacerlo.

Algunos investigadores han propuesto otras variantes de la cruza motivados por este problema. Radcliffe [178] propuso una técnica denominada **recombinación respetuosa aleatoria**.

Según esta técnica, se genera un hijo copiando los bits en los que sus padres son idénticos, y eligiendo luego, valores al azar para llenar las posiciones siguientes. Si se usan cadenas binarias y $Pc = 0.5$, la cruza uniforme es equivalente a esta recombinación.

7.5 Cruza para representaciones alternativas

Al cambiar la representación, el operador de cruza puede requerir modificaciones a fin de mantener la integridad del nuevo esquema de codificación adoptado. A continuación discutiremos brevemente algunas definiciones de la cruza para las siguientes representaciones alternativas:

- Programación Genética
- Permutaciones
- Representación real

7.5.1 Cruza para Programación Genética

Al usarse representación de árbol, la cruza sigue funcionando de manera muy similar a la cruza convencional, sólo que en este caso, se intercambian sub-árboles entre los 2 padres.

Figura 7.5: Un ejemplo con dos padres seleccionados para cruzarse, en programación genética.

El primer hijo se produce borrándole al primer padre el fragmento indicado por el punto de cruza e insertando el fragmento (sub-árbol) correspondiente del segundo padre. El segundo hijo se produce de manera análoga.

Por ejemplo, considere los 2 padres siguientes mostrados en la figura 7.5. Las expresiones S de estos 2 padres son:

`(OR (`**NOT D1**`) (AND D0 D1))` y

`(OR (OR D1 (NOT D0))` **(AND (NOT D0) (NOT D1))**`)`

Si el punto de cruza del primer padre es 2, y el del segundo es 6, entonces los hijos resultantes serán los indicados en la figura 7.6.

7.5.1.1 Observaciones sobre la cruza para programación genética

- Típicamente, los 2 padres a cruzarse serán de tamaños distintos.
- Los padres son seleccionados mediante alguna de las técnicas que vimos antes (p.ej., la ruleta).
- Suele limitarse la profundidad máxima de un árbol.

Figura 7.6: Los 2 hijos resultantes de la cruza entre los padres de la figura 7.5

- La raíz es un punto de cruza válido.

7.5.2 Cruza para Permutaciones

La representación de permutaciones se usa frecuentemente en problemas de optimización combinatoria, como el del viajero y consiste básicamente en usar cadenas de enteros para representar una permutación:

1 2 3 4 5 6 7 8 9

Al efectuar cualquiera de las cruzas antes descritas entre 2 cadenas que usan representación de permutaciones, los hijos invariablemente serán no válidos.

Este problema requiere la definición de procedimientos de "reparación" de las cadenas inválidas que se producen a consecuencia de la cruza. Estudiaremos brevemente las siguientes técnicas:

- Order Crossover
- Partially Mapped Crossover
- Position-Based Crossover
- Order-Based Crossover
- Cycle Crossover
- Otras

7.5.2.1 Order Crossover (OX)

Esta técnica fue propuesta por Davis [62]. El algoritmo es el siguiente (los padres son P1 y P2):

1. Seleccionar (aleatoriamente) una sub-cadena P1.

2. Producir un hijo copiando la sub-cadena en las posiciones correspondientes a P1. Las posiciones restantes se dejan en blanco.

3. Borrar los valores que ya se encuentren en la sub-cadena de P2. La secuencia resultante contiene los valores faltantes.

4. Colocar los valores en posiciones no conocidas del hijo de izquierda a derecha.

5. Para obtener el segundo hijo, se repiten los pasos del 1 al 4, pero tomando ahora la sub-cadena de P2.

Ejemplo de **Order Crossover**:

P1 = 9 8 4 5 6 7 1 2 3 10
P2 = 8 7 1 2 3 10 9 5 4 6

Sub-cadena elegida: 5 6 7 1 (de P1)
Primer hijo:

H1 = X X X 5 6 7 1 X X X

Borrar de P2 la sub-cadena tomada de P1:

P2' = 8 X X 2 3 10 9 X 4 X

Determinar los valores faltantes de H1 sustituyendo (de izquierda a derecha) los valores que aparecen en P2':

H1 = 8 2 3 5 6 7 1 10 9 4

Para obtener H2, el procedimiento es similar, aunque ahora la sub-cadena se tomará de P2 y la sustitución se hará a partir de P1'.

7.5.2.2 Partially Mapped Crossover (PMX)

Esta técnica fue propuesta por Goldberg y Lingle [113] y tiene ciertas similitudes con OX. El algoritmo es el siguiente:

1. Elegir aleatoriamente dos puntos de cruza.
2. Intercambiar estos 2 segmentos en los hijos que se generan (como la cruza de 2 puntos convencional).
3. El resto de las cadenas que conforman los hijos se obtienen haciendo mapeos entre los 2 padres:

   a) Si un valor no está contenido en el segmento intercambiado, permanece igual.

   b) Si está contenido en el segmento intercambiado, entonces se sustituye por el valor que tenga dicho segmento en el otro padre.

Ejemplo de **Partially Mapped Crossover**:

P1 = 9 8 4 | 5 6 7 | 1 3 2 10
P2 = 8 7 1 | 2 3 10 | 9 5 4 6

Los hijos son:
H1 = X X X | 2 3 10 | X X X X
H2 = X X X | 5 6 7 | X X X X

Para completar H1 y H2, copiamos primero los valores que no están en el segmento intercambiado:
H1 = 9 8 4 | 2 3 10 | 1 X X X
H2 = 8 X 1 | 5 6 7 | 9 X 4 X

Ahora mapeamos los valores restantes:
H1 = 9 8 4 2 3 10 1 6 5 7
H2 = 8 10 1 5 6 7 9 2 4 3

7.5.2.3 Position-based Crossover

Esta técnica fue propuesta por Syswerda [218] como una adaptación de la cruza uniforme para permutaciones. El algoritmo es el siguiente:

1. Seleccionar (al azar) un conjunto de posiciones de P1 (no necesariamente consecutivas).

2. Producir un hijo borrando de P1 todos los valores, excepto aquéllos que hayan sido seleccionados en el paso anterior.

3. Borrar los valores seleccionados de P2. La secuencia resultante de valores se usará para completar el hijo.

4. Colocar en el hijo los valores faltantes de izquierda a derecha, de acuerdo a la secuencia de P2.

5. Repetir los pasos del 1 al 4, pero tomando ahora la secuencia de P2.

Ejemplo de **Position-based Crossover**:

P1 = 9 8 4 5 6 7 1 2 3 10
P2 = 8 7 1 2 3 10 9 5 4 6

Valores elegidos de P1: 8 6 2 10

Producir un hijo:
H1 = X 8 X X 6 X X 2 X 10

Borrar de P2 la secuencia usada para H1:
P2'= X 7 1 X 3 X 9 5 4 X

Sustituir de izquierda a derecha los valores que aparecen en P2':
H1 = 7 8 1 3 6 9 5 2 4 10

Para obtener H2, el procedimiento es similar, pero la secuencia se toma ahora de P2 y la sustitución se hace a partir de P1'.

7.5.2.4 Order-based Crossover

Esta técnica fue propuesta por Syswerda [218] como una ligera variante de **Position-based Crossover** en la que se cambia el orden de los pasos del algoritmo.

En este caso, primero seleccionamos una serie de valores de P1. Luego, removemos de P2 esos valores. A continuación generamos un hijo a partir de P2'.

Finalmente, completamos el hijo con los valores de la secuencia obtenida de P1 (insertada de izquierda a derecha en el orden impuesto por P1).

Ejemplo de **Order-based Crossover**:

P1 = 9 8 4 5 6 7 1 2 3 10
P2 = 8 7 1 2 3 10 9 5 4 6

Valores elegidos de P1: 8 6 2 10

Removamos de P2 estos valores:
P2' = X 7 1 X 3 X 9 5 4 X

Producir un hijo:
H1 = X 7 1 X 3 X 9 5 4 X

Insertamos ahora la secuencia elegida de P1:
H1 = 8 7 1 6 3 2 9 5 4 10

7.5.2.5 Cycle Crossover (CX)

Esta técnica fue propuesta por Oliver, Smith y Holland [174]. Es similar a la **Position-based Crossover**, porque toma algunos valores de un padre y selecciona los restantes del otro. La principal diferencia es que los valores tomados del primer padre no se seleccionan al azar, sino de acuerdo a ciertas reglas específicas.

El algoritmo de **Cycle Crossover** es el siguiente:

1. Encontrar un ciclo que se define mediante las posiciones correspondientes de los valores entre los padres.
2. Copiar los valores de P1 que sean parte del ciclo.
3. Borrar de P2 los valores que estén en el ciclo.
4. Rellenar el hijo con los valores restantes de P2 (sustituyendo de izquierda a derecha).
5. Repetir los pasos del 1 al 4, usando ahora P2.

El elemento clave de esta técnica es saber cómo encontrar un **ciclo**.
Es más fácil explicar este concepto con un ejemplo:

P1 = h k c e f d b l a i g j
P2 = a b c d e f g h i j k l

Posiciones:

1 2 3 4 5 6 7 8 9 10 11 12

Ejemplo de un **ciclo**:

Tomamos al azar una posición de cualquier padre. En este caso, tomaremos la posición 1:

- Los elementos (h,a) pertenecen al ciclo 1.
- Si observamos a P1 y P2, veremos que "h" y "a" aparecen también en las posiciones 8 y 9. Por tanto, el ciclo incluye ahora las posiciones (1,8,9) y se agregan los elementos "i" y "l". O sea, que los elementos del ciclo son: (h,a,i,l).
- Vemos ahora que "i" y "l" aparecen en las posiciones 10 y 12. Por tanto, el ciclo ahora incluye las posiciones (1,8,9,10,12), y se agrega el elemento "j".
- Vemos ahora que "j" ya no se encuentra en ninguna otra posición, por lo que el ciclo se termina.

Ejemplo de **Cycle Crossover**:

P1= 8 11 3 5 6 4 2 12 1 9 7 10
P2= 1 2 3 4 5 6 7 8 9 10 11 12

Posiciones:

1 2 3 4 5 6 7 8 9 10 11 12

Tomemos la posición 1 para iniciar el ciclo: Los elementos del ciclo serán entonces (1,8). Pero 1 y 8 también aparecen en las posiciones 9 y 8. Por lo tanto, el ciclo ahora incluye los elementos (1,8,12). Pero 12 aparece también en la posición 12. Por lo tanto, el ciclo ahora incluye los elementos (1,8,12,10). Pero 10 aparece también en la posición 10. Por lo tanto, el ciclo ahora incluye los elementos (1,8,12,10,9). Ya no hay nuevos elementos qué agregar, por lo que se concluye el ciclo.

Para generar al primer hijo, tomamos a P1, removiéndole los elementos que no sean parte del ciclo:

H1 = 8 X X X X X X 12 1 9 X 10

Remover de P2 los valores del ciclo:

P2'= X 2 3 4 5 6 7 X X X 11 X

Rellenar H1 usando los valores restantes de P2'.

H1 = 8 2 3 4 5 6 7 12 1 9 11 10

7.5.3 Otras propuestas

Hay muchas otras propuestas. Por ejemplo:

- Fox y McMahon [88] propusieron un operador de intersección que extrae características comunes de ambos padres.
- Blanton y Wainwright [140] propusieron un operador llamado **merge crossover**, que permite incorporar preferencias en una cierta ruta a la hora de efectuar la cruza.

7.6 Cruza para Representación Real

Si elegimos el uso directo de vectores de números reales para nuestra codificación, es entonces deseable definir operadores de cruza más acordes a esta representación. El énfasis principal es la capacidad de poder "romper" un cierto valor real, de manera análoga a como la cruza ordinaria "rompe" segmentos de cromosoma al usarse representación binaria. Las técnicas que estudiaremos son las siguientes:

- Un punto
- Dos puntos
- Uniforme
- Intermedia

- Aritmética simple
- Aritmética total
- Simulated Binary Crossover
- Cruza basada en dirección

7.6.1 Cruza Simple

Se trata realmente de la cruza de un punto aplicada a vectores de números reales:
Ejemplo: Dados

P1 = $\langle 3.2, 1.9, |-0.5\rangle$
P2 = $\langle 2.9, 0.8, |1.4\rangle$

Los hijos serían:
H1 = $\langle 3.2, 1.9, 1.4\rangle$
H2 = $\langle 2.9, 0.8, -0.5\rangle$

7.6.2 Cruza de dos puntos

Esta cruza es igual que para el caso binario.
Ejemplo: Dados

P1= $\langle 1.6, -2.1, |3.5, 0.4, |5.6\rangle$
P2= $\langle 0.2, 4.5, |-2.3, 8.6, |-1.4\rangle$

Los hijos serían:
H1= $\langle 1.6, -2.1, -2.3, 8.6, 5.6\rangle$
H2= $\langle 0.2, 4.5, 3.5, 0.4, -1.4\rangle$

7.6.3 Cruza uniforme

Es igual que para el caso binario.
Ejemplo: Dados

P1= $\langle 1.6, -2.1, 3.5, 0.4, 5.6, 7.8\rangle$
P2= $\langle 0.2, 4.5, -2.3, 8.6, -1.4, 0.5\rangle$

Si Pc=0.5, entonces los hijos podrían ser:
H1= $\langle 1.6, 4.5, -2.3, 0.4, 5.6, 0.5\rangle$
H2= $\langle 0.2, -2.1, 3.5, 8.6, -1.4, 7.8\rangle$

7.6.4 Cruza intermedia

En este caso, si suponemos que tenemos dos padres:
P1 = $\langle v_1, \ldots, v_m\rangle$ y P2 = $\langle w_1, \ldots, w_m\rangle$,
los cuales se cruzan en la posición k, entonces los hijos producidos son:
H1= $\langle v_1, \ldots, v_k, w_{k+1} * a + v_{k+1} * (1-a), \ldots, w_m * a + v_m * (1-a)\rangle$
H2= $\langle w_1, \ldots, w_k, v_{k+1} * a + w_{k+1} * (1-a), \ldots, v_m * a + w_m * (1-a)\rangle$
donde: $a \in [0, 1]$
Ejemplo de **cruza intermedia**, suponiendo k = 3, a = 0.3:
P1= $\langle 1.6, -2.1, 3.5, 0.4, 5.6, 7.8\rangle$
P2= $\langle 0.2, 4.5, -2.3, 8.6, -1.4, 0.5\rangle$
los hijos serían:
H1= $\langle 1.6, -2.1, 3.5, 8.6*0.3+0.4*(1-0.3), -1.4*0.3+5.6*(1-0.3), 0.5*0.3+7.8*(1-0.3)\rangle$
H2= $\langle 0.2, 4.5, -2.3, 0.4*0.3+8.6*(1-0.3), 5.6*0.3-1.4*(1-0.3), 7.8*0.3+0.5*(1-0.3)\rangle$

7.6.5 Cruza aritmética simple

Es una variante de la cruza intermedia, en la cual se toman en cuenta los límites inferiores y superiores de las variables del problema.

La única diferencia con respecto a la cruza intermedia es el cálculo de "a":

$$a \in \begin{cases} [max(\alpha,\beta), min(\gamma,\delta)] & \text{si } v_k > w_k \\ [0,0] & \text{si } v_k = w_k \\ [max(\gamma,\delta), min(\alpha,\beta)] & \text{si } v_k < w_k \end{cases} \tag{7.1}$$

Para fines del cálculo de "a":

$$\begin{aligned} \alpha &= \tfrac{l_k^w - w_k}{v_k - w_k} \quad \beta = \tfrac{u_k^v - v_k}{w_k - v_k} \\ \gamma &= \tfrac{l_k^v - v_k}{w_k - v_k} \quad \delta = \tfrac{u_k^w - w_k}{v_k - w_k} \end{aligned} \tag{7.2}$$

donde cada valor v_k está en el rango $[l_k^v, u_k^v]$ y cada valor w_k está en el rango $[l_k^w, u_k^w]$.

Ejemplo:

P1 = $\langle 2.3, 4.5, -1.2, 0.8\rangle$
P2 = $\langle 1.4, -0.2, 6.7, 4.8\rangle$

Supondremos, que para toda v_k y para toda w_k, el rango es: $[-2, 7]$.

Si ahora ubicamos el punto de cruza entre la segunda y la tercera variable, tenemos:

H1= $\langle 2.3, 4.5, 6.7 * a_1 - 1.2 * (1 - a_1), 4.8 * a_2 + 0.8 * (1 - a_2)\rangle$
H2= $\langle 1, 4, -0.2, -1.2 * a_1 + 6.7 * (1 - a_1), 0.8 * a_2 + 4.8 * (1 - a_2)\rangle$

Haciendo k =3, tenemos:
$v_k = -1.2$, $w_k = 6.7$, por lo que: $w_k > v_k$

$$\begin{aligned} \alpha &= \tfrac{-2-6.7}{-1.2-6.7} = 1.1 \qquad \beta = \tfrac{7+1.2}{6.7+1.2} = 1.04 \\ \gamma &= \tfrac{-2+1.2}{6.7+1.2} = -0.101 \quad \delta = \tfrac{7-6.7}{-1.2-6.7} = -0.038 \end{aligned} \tag{7.3}$$

Por lo que: $a_1 \in [-0.038, 1.04]$

Como los rangos para la cuarta variable son los mismos que para la tercera, los cálculos son idénticos. Es decir: $a_2 \in [-0.038, 1.04]$

Supongamos que $a_1 = 0.84$ y $a_2 = 0.9$

H1 = $\langle 2.3, 4.5, 5.12, 4.4\rangle$
H2 = $\langle 1.4, -0.2, 0.38, 1.2\rangle$

7.6.6 Cruza aritmética total

Es igual que la anterior, sólo que en este caso no se tiene que elegir punto de cruza, pues se aplica la misma fórmula a todos los genes del cromosoma (o sea, a todas las variables de decisión).

Si no se pre-define un rango para cada variable, entonces se usa simplemente $a \in [0, 1]$, como en el caso de la cruza intermedia.

Ejemplo de **cruza aritmética total**:

P1 = $\langle 2.3, 4.5, -1.2, 0.8 \rangle$
P2 = $\langle 1.4, -0.2, 6.7, 4.8 \rangle$

Usando a = 0.6, tenemos:
H1 = $\langle 1.76, 1.68, 3.54, 3.2 \rangle$
H2 = $\langle 1.94, 2.62, 1.96, 2.4 \rangle$

7.6.7 Simulated Binary Crossover (SBX)

Fue propuesta por Deb [67] e intenta emular el efecto de la cruza de un punto usada con representación binaria.

El algoritmo es el siguiente:

1. Generar un número aleatorio "u" entre 0 y 1.

2. Calcular $\bar{\beta}$.

   donde:

$$\bar{\beta} = \begin{cases} (2u)^{\frac{1}{n_c+1}} & \text{si } u \leq 0.5 \\ \left(\frac{1}{2(1-u)}\right)^{\frac{1}{n_c+1}} & \text{de lo contrario} \end{cases} \tag{7.4}$$

   Los autores de la técnica sugieren: $n_c = 1$ ó 2.

3. Producir los hijos con:

   H1 = $0.5[(P1 + P2) - \bar{\beta}|P2 - P1|]$
   H2 = $0.5[(P1 + P2) + \bar{\beta}|P2 - P1|]$

   Ejemplo de SBX:
   P1 = $\langle 2.3, 4.5, -1.2, 0.8 \rangle$
   P2 = $\langle 1.4, -0.2, 6.7, 4.8 \rangle$

   Si usamos u=0.42, $n_c = 2$, tenemos:
   H1 = $\langle 1.42, -0.067, -0.97, 0.9129 \rangle$
   H2 = $\langle 2.27, 4.37, 6.48, 4.69 \rangle$

7.7 Problemas Propuestos

1. El objetivo de este problema es que implemente un algoritmo genético con una representación de permutaciones, el cual se utilizará para resolver un problema clásico de optimización combinatoria: el problema de la asignación cuadrática.

   Implemente una de las siguientes técnicas de cruza para permutaciones:

   - *Partially Mapped Crossover* (PMX).
   - *Position-based Crossover*.
   - *Order-based Crossover*.

   El algoritmo genético implementado deberá tener las siguientes características:

   - **Remoción de duplicados**: En la generación cero y en las generaciones posteriores (tras haber generado a través de la selección, cruza, mutación e inversión a la siguiente población), deberá checarse que no existan duplicados (o sea, no deben existir en la misma población dos individuos que codifiquen la misma permutación). Los duplicados deberán reemplazarse por un individuo generado al azar o producto de una alteración del individuo a removerse (por ejemplo, una mutación).
   - **Mutación**: Puede implementar cualquier operador de mutación de permutaciones de los discutidos en el siguiente capítulo.
   - **Selección**: Deberá implementarse la selección mediante torneo binario determinístico, usando barajeo de la población.
   - **Representación**: Debe implementarse una representación de permutaciones (o sea, cadenas de enteros de 0 a $n-1$, donde n representa el tamaño de la matriz de entrada, de tal forma que la secuencia numérica no tiene ningún elemento repetido).
   - **Inversión**: Debe implementarse un operador de inversión que usará los siguientes parámetros: porcentaje de aplicabilidad (como en la cruza y la mutación), punto inicial y punto final. La política de reemplazo es opcional, pero debe aclararla en su reporte. El operador se aplicará siempre después de la cruza y la mutación, pero debe decidirse si se reemplazará siempre al individuo original, si será en cierto número (aleatorio) de veces, o si se hará cuando el individuo tras inversión tenga mejor aptitud que el original.

Lo que debe incluirse en el reporte correspondiente es lo siguiente:

(a) El código fuente en C/C++ de un programa que implemente un algoritmo genético con codificación de permutaciones, utilizando inversión, cruza y mutación (como se indicó anteriormente). El programa deberá pedir al usuario (de forma interactiva y NO a través de la línea de comandos) los parámetros principales del algoritmo genético: tamaño de la población, porcentaje de cruza, porcentaje de mutación, porcentaje de inversión, número máximo de generaciones y nombre del archivo donde se almacenarán los datos de cada corrida. En cada generación deberá retenerse a la mejor solución (elitismo), y deberán imprimirse al menos las siguientes estadísticas por generación: media de aptitud de la población, aptitud máxima y mínima, número total de cruzas efectuadas, número total de mutaciones efectuadas, número total de inversiones efectuadas y permutación correspondiente al mejor individuo (el que tenga la aptitud más alta). Adicionalmente, deberá imprimirse el mejor individuo global (es decir, contando todas las generaciones), su aptitud, la permutación que codifica y su costo.

(b) Una corrida de ejemplo por cada uno de los problemas incluidos en el punto siguiente, y los resultados promediados de AL MENOS 20 corridas independientes para cada problema en las que se usen los mismos parámetros (porcentaje de cruza, mutación e inversión, tamaño de la población y máximo número de generaciones) pero diferente semilla para generar números aleatorios. En el reporte deberá aparecer una corrida representativa por problema, así como la mejor solución obtenida, la media de aptitud, la peor solución obtenida y la desviación estándar de las 20 corridas efectuadas. Deberán mostrarse las permutaciones (especificando claramente si deben leerse de izquierda a derecha o viceversa) y sus costos correspondientes en cada caso. Asimismo, se incluirá la gráfica de convergencia correspondiente a la solución que esté en la mediana de los resultados obtenidos (de las 20 corridas) para cada problema. En esta gráfica se mostrará en el eje de las x el número de generaciones (de cero al máximo permisible) y en el eje de las y se mostrará la aptitud promedio de cada generación.

(c) **Funciones de prueba**: Se le proporcionarán 4 archivos de prueba para evaluar el desempeño de su programa. Estos archivos estarán disponibles en la página web del curso, y se llaman: **ajuste.dat**, **tai10.dat**,

tai12.dat, **tai15.dat**. El primero de ellos es para ajustar su programa, y los otros 3 son los ejemplos que debe utilizar para su reporte. El formato de los archivos es el siguiente:

`tamaño de la matriz` (n)

`matriz de distancias` (dist[n][n])

`matriz de flujos` (flujo[n][n])

El objetivo es minimizar el costo total, definido por:

$$\text{costo} = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} dist[i][j] \times flujo[x[i]][x[j]] \tag{7.5}$$

donde: $x[i]$ ($i = 0, \ldots, n-1$) se refiere a la permutación codificada en el algoritmo genético. Se le sugiere escribir una función (independiente) que calcule la aptitud de una permutación a partir de las matrices de distancias y flujos. Si lee los datos del archivo **ajuste.dat**, y proporciona la permutación: **2 3 4 0 1**, el costo debe ser **50** (valor óptimo).

Los costos óptimos para los demás archivos son los siguientes:

`tai10.dat`	Costo óptimo = 135028
`tai12.dat`	Costo óptimo = 224416
`tai15.dat`	Costo óptimo = 388214

Capítulo 8

Mutación

La mutación se considera como un operador secundario en los algoritmos genéticos canónicos. Es decir, su uso es menos frecuente que el de la cruza.

En la práctica, se suelen recomendar porcentajes de mutación de entre 0.001 y 0.01 para la representación binaria.

Algunos investigadores, sin embargo, han sugerido que el usar porcentajes altos de mutación al inicio de la búsqueda, y luego decrementarlos exponencialmente, favorece el desempeño de un AG [79].

Otros autores sugieren que $p_m = \frac{1}{L}$ (donde L es la longitud de la cadena cromosómica) es un límite inferior para el porcentaje óptimo de mutación [7].

El papel que juega la mutación en el proceso evolutivo, así como su comparación con la cruza, sigue siendo tema frecuente de investigación y debate en la comunidad de computación evolutiva. En este capítulo estudiaremos diferentes técnicas de mutación que se han propuesta en la literatura especializada.

8.1 Mutación para Permutaciones

Comenzaremos nuestra revisión de técnicas enfocándonos a la representación de permutaciones que se suele utilizar en problemas de optimización combinatoria (como el del viajero).

8.1.1 Mutación por Inserción

Se selecciona un valor en forma aleatoria y se le inserta en una posición arbitraria.

Ejemplo:

Figura 8.1: Ejemplo de una mutación por desplazamiento.

Dada:

P = 9 4 2 1 5 7 6 10 3 8

Si suponemos que elegimos la posición 7 y decidimos mover ese valor a la posición 2, tendríamos:

P'= 9 6 4 2 1 5 7 10 3 8

8.1.2 Mutación por Desplazamiento

Es una generalización de la mutación por inserción en la que en vez de mover un solo valor, se cambian de lugar varios a la vez. Considere el ejemplo de la figura 8.1, en el cual tres valores son movidos a posiciones distintas en la cadena.

8.1.3 Mutación por Intercambio Recíproco

En este caso, se seleccionan dos puntos al azar y se intercambian estos valores de posición.

Por ejemplo, dada:

P= 9 4 2 1 5 7 6 10 3 8

Tendríamos:

P'=9 10 2 1 5 7 6 4 3 8

8.1.4 Mutación Heurística

Fue propuesta por Gen y Cheng [99]. El procedimiento es el siguiente:

1. Seleccionar λ genes al azar.
2. Generar vecinos de acuerdo a todas las permutaciones posibles de los genes seleccionados.
3. Evaluar todos los vecinos y seleccionar el mejor.

Consideremos el siguiente ejemplo:

P= 9 4 2 1 5 7 6 10 3 8

Generar todas las permutaciones de:

4 5 10

1) 4 10 5 2) 5 4 10 3) 5 10 4
4) 10 5 4 5) 10 4 5

Individuos generados:

P1= 9 4 2 1 10 7 6 5 3 8
P2= 9 5 2 1 4 7 6 10 3 8
P3= 9 5 2 1 10 7 6 4 3 8
P4= 9 10 2 1 5 7 6 4 3 8
P5= 9 10 2 1 4 7 6 5 3 8

De entre ellas, se selecciona a la mejor. En este caso, supondremos que es P4:

P'= 9 10 2 1 5 7 6 4 3 8

8.2 Mutación para Programación Genética

La **mutación** es un operador asexual que, en el caso de la programación genética, se aplica sobre una sola expresión S (en el caso de usar LISP). Se selecciona aleatoriamente un punto de mutación en el árbol y se remueve todo lo que esté en dicho punto y abajo de él, insertándose en su lugar un sub-árbol generado aleatoriamente (ver figura 8.2). La profundidad máxima de este sub-árbol está acotada por la profundidad total permitida para cualquier expresión S en la población inicial.

Figura 8.2: Ejemplo: Suponemos que el punto de mutación es el nodo 3.

8.3 Mutación para Representación Real

Si usamos directamente vectores de números reales para representar las variables de nuestro problema, es importante tomar en cuenta la alta cardinalidad que tendrá el alfabeto en este caso, y se requiere diseñar operadores acordes a ella.

8.3.1 Mutación No Uniforme

Propuesta por Michalewicz [161].

Dado:

$$P = \langle V_1, ..., V_m \rangle$$

el individuo mutado será:

$$P' = \langle V_1, ..., V'_k, ..., V_m \rangle$$

donde:

$$V'_k = \begin{cases} V_k + \Delta(t, UB - V_k) & \text{si R=Cierto} \\ V_k - \Delta(t, V_k - LB) & \text{si R=Falso} \end{cases}$$

y la variable V_k está en el rango $[LB, UB]$

R = flip(0.5)

$\Delta(t, y)$ regresa un valor en el rango $[0, y]$ tal que la probabilidad de que $\Delta(t, y)$

esté cerca de cero se incrementa conforme t (generación actual) crece. Esto hace que este operador explore de manera más global el espacio de búsqueda al inicio (cuando t es pequeña) y de manera más local en etapas posteriores.

Michalewicz sugiere usar:

$$\Delta(t, y) = y * (1 - r^{(1-\frac{t}{T})^b})$$

donde:
r es un número aleatorio real entre 0 y 1, T es el número máximo de generaciones y b es un parámetro que define el grado de no uniformidad de la mutación (Michalewicz [161] sugiere usar $b = 5$).

Ejemplo:

$$P = \langle 2.3, 4.5, -1.2, 0.8 \rangle$$

$V_k = 4.5$, $l_k = -2.0$, $u_k = 6.5$,
$T = 50$, $t = 5$, R = Falso,
$r = 0.24$, $b = 5$.

$$V'_k = V_k - \Delta(t, v_k - l_k)$$
$$= 4.5 - \Delta(5, 4.5 + 2) = 4.5 - \Delta(5, 6.5)$$

$$\Delta(5, 6.5) = 6.5(1 - 0.24^{(1-\frac{5}{50})^5}) = 6.489435$$
$$V'_k = 4.5 - 6.489435 = -1.989435$$

8.3.2 Mutación de Límite

Dado:

$$P = \langle V_1, ..., V_m \rangle$$

el individuo mutado será:

$$P' = \langle V_1, ..., V'_k, ..., V_m \rangle$$

donde:

$$V'_k = \begin{cases} LB & \text{Si flip(0.5)=TRUE} \\ UB & \text{de lo contrario} \end{cases}$$

y $[LB, UB]$ definen los rangos mínimos y máximos permisibles de valores para la variable V'_k.

Ejemplo:

$$P = \langle 1.5, 2.6, -0.5, 3.8 \rangle$$

$$V'_k = -0.5, \qquad LB = -3.0, \qquad UB = 1.3$$

Supongamos que: flip(0.5) = TRUE

$$V'_k = -3.0$$

8.3.3 Mutación Uniforme

Dado:

$$P = \langle V_1, ..., V'_k, ..., V_m \rangle$$

el individuo mutado será:

$$P' = \langle V_1, ..., V'_k, ..., V_m \rangle$$

donde:

$$V'_k = rnd(LB, UB)$$

se usa una distribución uniforme y [LB, UB] definen los rangos mínimos y máximos de la variable V'_k.

Ejemplo:

$$P = \langle 5.3, -1.3, 7.8, 9.1 \rangle$$

$$V_k = 5.3, LB = 0.0, UB = 10.5$$

$$V'_k = rnd(0.0, 10.5) = 4.3$$

8.3.4 Parameter-Based Mutation

Utilizada en conjunción con SBX. Fue propuesta por Deb [67, 65]. Su objetivo es crear una solución c en la vecindad de una solución padre y. Se presupone que el padre y está acotado ($y \in [y_l, y_u]$). El procedimiento es el siguiente:

1. Crear un número aleatorio u entre 0 y 1

2. Calcular:

$$\delta_q = \begin{cases} [2u + (1-2u)(1-\delta)^{\eta_m+1}]^{\frac{1}{\eta_m+1}} - 1 & \text{si } u \leq 0.5 \\ 1 - [2(1-u) + 2(u-0.5)(1-\delta)^{\eta_m+1}]^{\frac{1}{\eta_m+1}} & \text{de lo contrario} \end{cases} \tag{8.1}$$

   donde $\delta = \min[(y - y_l), (y_u - y)]/(y_u - y_l)$.

   El parámetro η_m es el índice de distribución para la mutación y toma cualquier valor no negativo. Deb sugiere usar:

$$\eta_m = 100 + t \;\; (t = \text{generación actual})$$

3. El valor de la posición mutada se determina usando:

$$y' = y + \delta_q \Delta_{max}$$

   donde Δ_{max} es la máxima perturbación permitida:

$$\Delta_{max} = y_u - y_l$$

   considerando que:

$$y \in [y_l, y_u]$$

Ejemplo:

$P =$<2.3, 4.5, -1.2, 0.8>

$y =$ -1.2, $u =$ 0.72, $t =$20

y_l = -2.0, y_u = 6.0

$\eta_m = 100 + t = 120$

$\delta = \min\ [(-1.2 + 2.0), (6.0 + 1.2)]\,/(6.0 + 2.0)$

$\delta = 0.8/8.0 = 0.1$

$\delta_q = 1 - [2(1 - 0.72) + 2(0.72 - 0.5)(1 - 0.1)^{121}]^{\frac{1}{121}}$

$\delta_q = 4.7804 \times 10^3$

$\Delta_{max} = y_u - y_l = 6.0 + 2.0 = 8.0$

y' = -1.2 + 0.03824 = -1.16175.

8.4 Cruza vs. Mutación

La cruza uniforme es más "explorativa" que la cruza de un punto.

Por ejemplo, dados:

P1 = 1 * * * * 1
P2 = 0 * * * * 0

La cruza uniforme producirá individuos del esquema ******, mientras que la cruza de un punto producirá individuos de los esquemas 1****0 y 0****1.

8.4.1 ¿Cuál es el poder exploratorio de la mutación?

- Si el porcentaje de mutación es cero, no hay alteración alguna.
- Si es uno, la mutación crea siempre complementos del inidividuo original.
- Si es 0.5, hay una alta probabilidad de alterar fuertemente el esquema de un individuo.

En otras palabras, podemos controlar el poder de alteración de la mutación y su capacidad de exploración puede hacerse equivalente a la de la cruza.

El tipo de exploración efectuada por la mutación es, sin embargo, diferente a la de la cruza.

Por ejemplo, dados:

P1 = 10****

P2 = 11****

La cruza producirá sólo individuos del esquema 1*****.

El primer "1" en el esquema está garantizado (sin importar qué tipo de cruza se use), porque es común en los esquemas de ambos padres. La mutación, sin embargo, no respetará necesariamente este valor.

La cruza "preserva" los alelos que son comunes en los 2 padres. Esta preservación limita el tipo de exploración que la cruza puede realizar. Esta limitación se agudiza conforme la población pierde diversidad, puesto que el número de alelos comunes se incrementará.

Cuando buscamos localizar el óptimo global de un problema, la mutación puede ser más útil que la cruza. Si lo que nos interesa es ganancia acumulada (el objetivo original del AG), la cruza es entonces preferible.

La cruza parece trabajar bien con funciones que están altamente correlacionadas o tienen epístasis moderada.

8.5 Problemas Propuestos

1. Implemente un algoritmo genético con representación real y utilice alguna de las técnicas de cruza y mutación que hemos visto hasta ahora. Utilícelo para **minimizar** la siguiente función:

$$f(x_1, x_2) = [1.5 - x_1(1 - x_2)]^2 + [2.25 - x_1(1 - x_2^2)]^2 + [2.625 - x_1(1 - x_2^3)]^2 \tag{8.2}$$

   donde $0 \leq x_1 \leq 20$, $0 \leq x_2 \leq 30$ con una precisión de 5 dígitos después del punto decimal. Esta es la función de Beale.

2. Implemente la programación genética usando los operadores que hemos visto hasta ahora. Utilice su sistema para generar un circuito que produzca todas las salidas indicadas en la tabla 8.1 (las entradas son **X**, **Y**, **Z** y la salida es **F**. Utilice sólo compuertas AND, OR, NOT y XOR en su programa. Existen varias referencias bibliográficas que pueden serle de utilidad. Vea por ejemplo:

Z	**W**	**X**	**Y**	**F**
0	0	0	0	1
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	0

Tabla 8.1: Tabla de verdad para el segundo problema propuesto.

- Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Fancone, *Genetic Programming. An Introduction*, Morgan Kaufmann Publishers, San Francisco, California, 1998.
- John R. Koza, *Genetic Programming. On the Programming of Computers by Means of Natural Selection*, The MIT Press, Cambridge, Massachusetts, 1992.

Capítulo 9

Ajuste de Parámetros

9.1 Introducción

Los parámetros principales de un AG son:

- Tamaño de población
- Porcentaje de cruza
- Porcentaje de mutación

Estos parámetros normalmente interactúan entre sí de forma no lineal, por lo que no pueden optimizarse de manera independiente.

La forma óptima (o al menos razonablemente adecuada) de definir estos parámetros de un AG ha sido motivo de investigación desde los orígenes mismos de la técnica, y no existe hasta la fecha una solución satisfactoria a este problema.

A lo largo de este capítulo hablaremos de los diferentes intentos que han habido para determinar un conjunto de parámetros con los cuales un algoritmo genético pueda tener un desempeño razonablemente bueno.

9.2 Los experimentos de De Jong

De Jong [137] efectuó una serie de experimentos para comparar AGs con técnicas de gradiente. En su estudio, De Jong propuso cinco funciones de prueba que exhibían una serie de características que las hacían difíciles para las técnicas de gradiente.

Sin embargo, antes de proceder a realizar sus experimentos, De Jong decidió analizar la influencia de los parámetros de un AG en su desempeño.

Para medir el impacto de los parámetros de un AG en su desempeño, De Jong propuso dos métricas:

1. **Desempeño en Línea (*Online*)**: Es la aptitud promedio de todos los individuos que han sido evaluados en las últimas t generaciones.

2. **Desempeño Fuera de Línea (*Offline*)**: Es el promedio de las mejores aptitudes evaluadas en las últimas t generaciones.

Para que un algoritmo de búsqueda tenga un buen desempeño "en línea", debe decidir rápidamente dónde están las regiones más prometedoras de búsqueda y concentrar ahí sus esfuerzos.

El desempeño "fuera de línea" no penaliza al algoritmo de búsqueda por explorar regiones pobres del espacio de búsqueda, siempre y cuando ello contribuya a alcanzar las mejores soluciones posibles (en términos de aptitud).

Los parámetros hallados por De Jong que tuvieron mejor desempeño en y fuera de línea son:

Tamaño de población:	50 a 100 individuos
Porcentaje de cruza:	0.60
Porcentaje de mutación:	0.001

Algunas conclusiones interesantes de De Jong [137] fueron:

- Incrementar el tamaño de la población reduce los efectos estocásticos del muestreo aleatorio en una población finita, por lo que mejora el desempeño del algoritmo a largo plazo, aunque esto es a costa de una respuesta inicial más lenta.

- Incrementar el porcentaje de mutación mejora el desempeño "fuera de línea"a costa de sacrificar el desempeño "en línea".

- Reducir el porcentaje de cruza mejora la media de desempeño, lo que sugiere que producir una generación de inividuos completamente nuevos no es bueno.

- Observando el desempeño de diferentes operadores de cruza, De Jong concluyó que, aunque el incrementar el número de puntos de cruza afecta su disrupción de esquemas desde una perspectiva teórica, esto no parece tener un impacto significativo en la práctica.

Grefenstette [116] usó un AG para optimizar los parámetros de otro (un meta-AG).

El meta-AG fue usado para evolucionar unos 50 conjuntos de parámetros de un AG que se usó para resolver las funciones de De Jong.

Cada individuo codificaba seis parámetros:

1. Tamaño de la población
2. Porcentaje de Cruza
3. Porcentaje de Mutación
4. Intervalo generacional (porcentaje de la población que se reemplaza en cada generación)
5. Ventana de escalamiento (sin escalamiento, escalamiento basado en la $f(x)$ mínima de la primera generación, escalamiento basado en la $f(x)$ mínima de las últimas W generaciones)
6. Estrategia de selección (elitista o puramente seleccionista).

La aptitud de un individuo era una función de su desempeño en y fuera de línea.

El meta-AG usaba los parámetros de De Jong, y con él, Grefenstette [116] obtuvo los siguientes valores óptimos de los parámetros para el desempeño "en línea":

Tamaño de la población:	30 individuos
Porcentaje de cruza (un punto):	0.95
Porcentaje de mutación:	0.01
Selección:	Elitista
Intervalo generacional:	1.0 (100%)
Ventana de escalamiento:	1 (basado en la $f(x)$ mínima de la primera generación)

Estos parámetros mejoran ligera pero significativamente el desempeño "en línea" del AG con respecto a los de De Jong, pero Grefenstette no pudo mejorar el desempeño "fuera de línea".

Algunas observaciones realizadas por Grefenstette [116]:

- Los porcentajes de mutación por encima de 0.05 tienden a ser perjudiciales con respecto al desempeño "en línea", y el AG aproxima el comportamiento de la búsqueda aleatoria para porcentajes de mutación ≤ 0.1 sin importar qué otros parámetros se usen.

- La ausencia de mutación está también asociada con un desempeño pobre del AG, lo que sugiere que su papel es más importante de lo que normalmente se cree, pues permite refrescar valores perdidos del espacio de búsqueda.

- El tamaño óptimo de la población para el desempeño "fuera de línea" está entre 60 y 110 individuos. Un alto intervalo generacional y el uso de una estrategia elitista también mejoran el desempeño del AG.

- Para poblaciones pequeñas (20 a 40 individuos), el buen desempeño "en línea" está asociado con un porcentaje alto de cruza combinado con un porcentaje bajo de mutación o viceversa (un porcentaje bajo de cruza combinado con un porcentaje alto de mutación).

- Para poblaciones de tamaño medio (30 a 90 individuos), el porcentaje óptimo de cruza parece decrementarse conforme se aumenta el tamaño de la población.

9.3 Tamaño óptimo de población

Goldberg [103] realizó un estudio teórico del tamaño ideal de la población de un AG en función del número esperado de nuevos esquemas por miembro de la población. Usando una población inicial generada aleatoriamente con igual probabilidad para el cero y el uno, Goldberg derivó la siguiente expresión:

Tam_Población = $1.65^{(2^{0.21L})}$

donde: L = longitud de la cadena (binaria).

Esta expresión sugiere tamaños de población demasiado grandes para cadenas de longitud moderada.

Considere los siguientes ejemplos:

L = 30, Tam_Población = 130
L = 40, Tam_Población = 557
L = 50, Tam_Población = 2389
L = 60, Tam_Población = 10244

Han habido innumerables ataques al trabajo de Goldberg antes mencionado, ya que éste se basó en una interpretación errónea del teorema de los esquemas. Para entender la falacia del argumento de Goldberg, debemos comenzar por definir un concepto muy importante de computación evolutiva, el **paralelismo implícito**.

El **paralelismo implícito** se define así:

> Mientras un AG calcula explícitamente las aptitudes de los N miembros de una población, al mismo tiempo estima implícitamente las aptitudes promedio de una cantidad mucho mayor de esquemas, calculando implícitamente las aptitudes promedio observadas de los esquemas que tienen instancias en la población.

Según el teorema de los esquemas (que veremos más adelante), un AG procesa $O(N^3)$ esquemas. A partir de esta idea, Goldberg concluye entonces que a mayor valor de N (tamaño de la población), mejor desempeño tendrá el AG, y de ahí deriva su expresión para calcular el tamaño óptimo de una población.

El problema con este argumento es que sólo hay 3^L esquemas en una representación binaria, por lo que no se pueden procesar $O(N^3)$ esquemas si N^3 es mucho mayor que 3^L.

Robertson [190] determinó que en los AGs paralelos, el desempeño se incrementaba monotónicamente con el tamaño de la población (es decir, no corresponde a la expresión derivada por Goldberg).

Otros investigadores, han derivado expresiones según las cuales un incremento lineal del tamaño de la población corresponde con un buen desempeño del AG.

La regla empírica más común es usar una población de al menos 2 veces L.

Algunas observaciones de Goldberg [103, 106] son las siguientes:

- Cuando puede demostrarse convergencia de un AG, ésta parece no ser peor que una función cuadrática o cúbica del número de bloques constructores del problema, independientemente del tipo de esquema de solución utilizado.

- La teoría sugiere que el tamaño óptimo de la población es $N = 3$, sin importar la longitud de la cadena cromosómica. Esta observación dio pie al micro-AG (Krishnakumar [148]).

El funcionamiento de un micro-AG es el siguiente:

1. Generar al azar una población muy pequeña.
2. Aplicar los operadores genéticos hasta lograr convergencia nominal (por ejemplo, todas las cadenas son iguales).
3. Generar una nueva población transfiriendo los mejores individuos de la población anterior a la nueva, y generando al azar los individuos restantes.
4. Continuar hasta llegar al óptimo.

9.4 Los experimentos de Schaffer

Schaffer et al. [200] efectuaron una serie de experimentos que consumieron aproximadamente 1.5 años de tiempo de CPU (en una Sun 3 y una VAX), en los cuales intentaron encontrar los parámetros óptimos de un AG con codificación de Gray y usando muestreo estocástico universal.

Los parámetros sugeridos por estos experimentos (para el desempeño "en línea") fueron:

Tamaño de población:	20-30 individuos
Porcentaje de cruza (2 puntos):	0.75-0.95
Porcentaje de mutación:	0.005-0.01

Algunas de las observaciones de Schaffer et al. [200] fueron:

- El uso de tamaños grandes de población (> 200) con porcentajes altos de mutación (> 0.05) no mejora el desempeño de un AG.
- El uso de poblaciones pequeñas (< 20) con porcentajes bajos de mutación (< 0.002) no mejora el desempeño de un AG.
- La mutación parece tener mayor importancia de lo que se cree en el desempeño de un AG.

- El AG resultó relativamente insensible al porcentaje de cruza. Un NE (*naive evolution*), o sea, un AG sin cruza, funciona como un *hill climber*, el cual puede resultar más poderoso de lo que se cree.
- Los operadores genéticos pueden muestrear eficientemente el espacio de búsqueda sin necesidad de usar tamaños de población excesivamente grandes.
- La cruza de 2 puntos es mejor que la de un punto, pero sólo marginalmente.
- Conforme se incrementa el tamaño de la población, el efecto de la cruza parece diluirse.

9.5 Auto-adaptación

En general, es poco probable poder determinar "a priori" un conjunto óptimo de parámetros para un AG cualquiera aplicado a un problema en particular. Algunos investigadores creen que la mejor opción es la **auto-adaptación**.

Ejemplo de Adaptación en Línea

Srinivas y Patnaik [210] propusieron un esquema para adaptar las probabilidades de cruza y mutación de un algoritmo genético. La propuesta se basa en la detección de que el algoritmo genético ha convergido. Para ello, verifican qué diferencia existe entre la aptitud máxima de la población y la aptitud promedio. Da tal forma, se hacen variar los porcentajes de cruza y mutación en función de esta diferencia de valores (aptitud máxima y aptitud promedio de la población). Las expresiones propuestas son:

$$p_c = k_1/(f_{max} - \bar{f})$$

$$p_m = k_2/(f_{max} - \bar{f})$$

Sin embargo, con estas expresiones los porcentajes de curza y mutación se incrementan conforme el algoritmo genético converge y produce un comportamiento altamente disruptivo en la vecindad del óptimo, de manera que el algoritmo puede no converger jamás.

Para evitar este problema, estas expresiones deben modificarse de manera que se preserven las "buenas" soluciones.

La propuesta es ahora la siguiente:

$$p_c = k_1(f_{max} - f')/(f_{max} - \bar{f}), k_1 \leq 1.0$$

$$p_m = k_2(f_{max} - f)/(f_{max} - \bar{f}), k_2 \leq 1.0$$

donde k_1 y k_2 deben ser menores que 1.0 para hacer que los valores de p_c y p_m estén en el rango de 0.0 a 1.0. En estas fórmulas, f_{max} es la aptitud máxima de la población, f' es la aptitud más grande de los padres a recombinarse y f es la aptitud del individuo a mutarse. Así mismo, $\bar{f}$ es la aptitud promedio de la población.

Estas expresiones hacen que el porcentaje de cruza (p_c) y de mutación (p_m) disminuya cuando los individuos tienen una aptitud alta y que aumenten en caso contrario. Nótese sin embargo que p_c y p_m se harán cero al alcanzarse la aptitud máxima. También adviértase que $p_c = k_1$ si $f' = \bar{f}$ y $p_m = k_2$ si $f = \bar{f}$. Para evitar valores mayores a 1.0 para p_c y p_m, se imponen las restricciones siguientes:

$$p_c = k_3, f' \leq \bar{f}$$

$$p_m = k_4, f \leq \bar{f}$$

donde $k_3, k_4 \leq 1.0$.

Debido a que p_c y p_m se harán cero cuando el individuo sea el mejor en la población, su propagación puede llegar a ser exponencial, produciendo convergencia prematura. Para evitar eso, los autores proponen usar un porcentaje de mutación por omisión (0.005) en estos casos.

Las expresiones finales son:

$$p_c = k_1(f_{max} - f')/(f_{max} - \bar{f}), f' \geq \bar{f}$$

$$p_c = k_3, f' < \bar{f}$$

$$p_m = k_2(f_{max} - f)/(f_{max} - \bar{f}), f \geq \bar{f}$$

$$p_m = k_4, f < \bar{f}$$

donde: k_1,k_2, k_3 y $k_4 \leq 1.0$. Los autores sugieren:

$$k_2 = k_4 = 0.5$$

$$k_1 = k_3 = 1.0$$

Con estos valores, se usan soluciones con una aptitud inferior al promedio para buscar la región donde reside el óptimo global. Un valor de $k_4 = 0.5$ hará

que estas soluciones sean totalmente disruptivas. Lo mismo hará $k_2 = 0.5$ con las soluciones cuya aptitud iguale el promedio de la población.

Asignar $k_1 = k_3 = 1.0$ hará que todas las soluciones cuya aptitud sea menor o igual a $\bar{f}$ se sometan compulsivamente a cruza.

La probabilidad de cruza decrece conforme la aptitud del mejor de los padres a recombinarse tiende a f_{max} y se vuelve cero para los individuos con una aptitud igual a f_{max}.

Auto-adaptación de la probabilidad de mutación

En este caso, el porcentaje de mutación se agrega como un parámetro más al genotipo, de tal forma que se vuelva una variable más tal que su valor oscile entre 0.0 y 1.0.

Bäck y Schütz (1996) proponen usar:

$$p'_m = \frac{1}{1 + \frac{1-p_m}{p_m} e^{-\gamma N(0,1)}}$$

donde:

$p_m =$ porcentaje actual de mutación, $p'_m =$ nuevo porcentaje de mutación.

$\gamma =$ tasa de aprendizaje (se sugiere $\gamma = 0.2$).

$N(0,1)$ indica una variable aleatoria con una distribución normal tal que su esperanza es cero y su desviación estándar es uno.

Aplicando este operador, pasamos del genotipo:

$$c = (x, p_m)$$

al nuevo genotipo:

$$(x', p'_m)$$

La mutación de la variable x está dada por:

$$x' = \begin{cases} x_j & \text{si } q \geq p'_m \\ 1 - x_j & \text{si } q < p'_m \end{cases}$$

donde: q es un valor aleatorio (distribución uniforme) muestreado de nuevo para cada posición j.

Este esquema puede generalizarse incluyendo un vector p de porcentajes de mutación asociados a cada variable:

$$p = (p_1, \cdots, p_L)$$

El genotipo c tiene la forma:

$$c = (x, p)$$

Los porcentajes de mutación se actualizan usando:

$$p'_j = \frac{1}{1 + \frac{1-p_j}{p_j} e^{-\gamma N(0,1)}}, j = 1, \cdots, L.$$

9.5.1 La propuesta de Davis

Davis [63, 64] realizó un estudio muy interesante sobre un mecanismo de auto-adaptación aplicado a algoritmos genéticos. En su estudio, Davis asignó a cada operador genético una "aptitud", la cual era función de cuántos individuos con aptitud elevada habían contribuido a crear dicho operador en un cierto número de generaciones. Un operador era recompensado por crear buenos individuos directamente, o por "dejar la mesa puesta" para ello (es decir, por crear ancestros para los buenos individuos).

La técnica de Davis se usó en un AG de estado uniforme. Cada operador (cruza y mutación) empezaba con la misma aptitud y cada uno de estos operadores se seleccionaba con una probabilidad basada en su aptitud para crear un nuevo individuo, el cual reemplazaba al individuo menos apto de la población. Cada individuo llevaba un registro de quién lo creó. Si un individuo tenía una aptitud mayor que la mejor aptitud actual, entonces el individuo recibía una recompensa para el operador que lo creó y ésta se propagaba a su padre, su abuelo, y tantos ancestros como se deseara. La aptitud de cada operador sobre un cierto intervalo de tiempo estaba en función de su aptitud previa y de la suma de recompensas recibidas por todos los individuos que ese operador hubiese ayudado a crear en ese tiempo.

Para implementar la auto-adaptación, suelen codificarse los porcentajes de cruza y mutación (y a veces incluso el tamaño de la población) como variables adicionales del problema. Los valores de los parámetros del AG se evolucionan de acuerdo a su efecto en el desempeño del algoritmo.

9.5.2 Críticas a la auto-adaptación

La auto-adaptación no ha sido tan exitosa en los AGs, como lo es en otras técnicas evolutivas (p.ej., las estrategias evolutivas) ¿Por qué?

El problema fundamental es que nadie ha podido contestar satisfactoriamente la siguiente pregunta [167]:

¿qué tan bien corresponde la velocidad de adaptación de la población con la adaptación de sus parámetros?

Dado que la información necesaria para auto-adaptar los parámetros proviene de la población misma, esta información podría no poder viajar suficientemente rápido como para reflejar con fidelidad el estado actual de la población. De ahí que el uso de auto-adaptación en un AG siga siendo objeto de controversia.

9.6 Mecanismos de Adaptación

9.6.1 Mutaciones Variables

Varios investigadores han abordado el problema del ajuste del porcentaje de mutación de un algoritmo genético. La idea de usar porcentajes de mutación dependientes del tiempo fue sugerida por Holland [127], aunque no proporcionó una expresión en particular que describiera la variabilidad requerida. Fogarty [79] usó varias expresiones para variar p_m en las que se incluye el tiempo, logrando mejoras notables de desempeño. En ambos casos, la propuesta fue decrementar de manera determinística los porcentajes de mutación, de manera que tiendan a cero.

Otra propuesta es la de Hesser y Männer [120], en la cual se usa:

$$p_m(t) = \sqrt{\frac{\alpha}{\beta}} \frac{e^{-\gamma^{t/2}}}{\lambda\sqrt{l}}$$

donde: λ = tamaño de la población, l = longitud cromosómica, t = generación actual, α, β, γ son constantes definidas por el usuario (dependientes del problema).

Nótese que en todas estas propuestas se usa el mismo porcentaje de mutación para todos los individuos de la población en la generación t.

Bäck y Schütz (1996) propusieron un porcentaje de mutación que se decrementa usando:

$$p_m(t) = \frac{L}{2 + \frac{L-2}{T}t}$$

donde: $0 \leq t \leq T$, $L =$ longitud cromosómica, $t =$ generación actual y T es el número máximo de generaciones.

La variabilidad es:

$p_m(0) = 0.5$

$p_m(T) = \frac{1}{L}$

9.6.2 Mutación dependiente de la aptitud

Bäck (1992) sugirió el uso de un porcentaje de mutación que fuera función de la aptitud de cada individuo:

$$p_m(x) = \frac{1}{2(f(x) + 1) - L}$$

9.6.3 AGs adaptativos

Los objetivos principales de los AGs adaptativos son los siguientes:

- Mantener diversidad en la población.
- Mejorar la convergencia del AG, evitando a la vez la convergencia prematura.
- Evitar la disrupción de esquemas ocasionada por la cruza.

De acuerdo a como lo plantean Herera y Lozano (1996), un AGA incluye:

- Ajuste adaptativo de parámetros (probabilidad de cruza y mutación, longitud del genotipo y tamaño de la población).
- Función de aptitud adaptativa.
- Operador de selección adaptativo.
- Operadores de búsqueda (variación) adaptativos.
- Representación adaptativa.

El mecanismo de adaptación puede estar completamente separado del mecanismo de búsqueda del AG. Este tipo de esquema "no acoplado" no es muy atractivo, porque implica un control centralizado, superimpuesto al mecanismo de búsqueda del AG.

Otra posibilidad es que el mecanismo de búsqueda del AG sea usado parcialmente por el mecanismo adaptativo. En este caso, se dice que el AG y el mecanismo adaptativo están "ligeramente acoplados" (*loosely coupled*).

Si la adaptación es conducida por las fuerzas internas de la búsqueda evolutiva, podemos hablar de un "acoplamiento fuerte". En este caso, se origina un acoplamiento de los 2 espacios de búsqueda sobre los que opera el AG (el espacio de las soluciones y el de las variables de decisión).

9.6.4 Técnicas Adaptativas Basadas en Lógica Difusa

Los controladores difusos suelen usarse con frecuencia como técnica adaptativa con los AGs (Herrera y Lozano, 1996).

La integración de AGs y controladores difusos suelen orientarse hacia los temas siguientes:

1. Elegir los parámetros del AG antes de efectuar las corridas.

2. Ajustar los parámetros en línea, adaptándose a nuevas situaciones.

3. Asistir al usuario en detectar las soluciones emergentes útiles, en monitorear el proceso evolutivo con la finalidad de evitar convergencia prematura, y en diseñar AGs para una cierta tarea en particular.

9.6.5 Representaciones Adaptativas

Se han propuesto varios esquemas en los que lo que se adapta es la representación usada con un AG. A continuación veremos 2 propuestas muy interesante de este tipo.

Sistema ARGOT

Propuesto por Schaefer (1987), el método ARGOT es un algoritmo de búsqueda diseñado de tal manera que puede "aprender" la estrategia de búsqueda más adecuada.

ARGOT consiste de un AG convencional al que se agregan varios operadores que modifican el mapeo intermedio que traduce los cromosomas en parámetros (o variables) de decisión. Estos operadores se controlan por medio de 3 tipos de medidas internas de la población:

(a) Medidas de convergencia (p.ej., la uniformidad de los cromosomas en un cierto lugar en particular).

(b) Medidas de posicionamiento (posición promedio relativa de las soluciones actuales con respecto a sus rangos extremos).

(c) Medidas de varianza (p.ej., el "ancho" de la distribución de las soluciones con respecto a los rangos permisibles).

Estas medidas se aplican sobre cada gene del cromosoma y se usan para activar un cierto operador cuando resulta adecuado.

Los operadores incluyen uno que altera la resolución de un gene (número de bits empleados para representar una variable) y otros que mueven (*shift*), expanden y contraen el mapeo intermedio entre cromosomas y variables de decisión. Estos cambios (expansión y contracción) hacen que los rangos de cada variable se modifiquen con la finalidad de focalizar la búsqueda y permiten también aplicar restricciones de desigualdad a las variables de decisión.

Además de los operadores primarios antes mencionados, se usaron otros secundarios tales como un operador de mutación de Metropolis que acepta un cambio en un bit sólo si mejora la solución actual con la mutación. Si el cambio no mejora, se decide si se acepta o no el cambio usando una distribución de Boltzmann. También se propuso un operador de homotopía (búsqueda local) para evitar convergencia a un óptimo local.

Codificación Delta

La idea de esta propuesta (Matthias & Whitley 1994) es cambiar denámicamente la representación de un problema. Nótese, sin embargo, que no intenta "aprender" cuál es la mejor representación del espacio de búsqueda, sino más bien se cambia la representación de manera periódica para evitar los sesgos asociados con una representación determinada del problema.

El algoritmo de la codificación delta empieza con la ejecución inicial de un algoritmo genético usando cadenas binarias. Una vez que la diversidad de la población ha sido explotada adecuadamente, se almacena la mejor solución bajo

el nombre de "solución temporal". Se reinicializa entonces el AG con una nueva población aleatoria. En esta ocasión, sin embargo, las variables se decodifican de tal forma que representen una distancia o *valor delta* ($\pm\delta$) con respecto a la solución temporal.

El valor de δ se combina con la solución temporal de manera que los parámetros resultantes se evalúen usando la misma función de aptitud.

De esta manera, la codificación delta produce un nuevo mapeo del espacio de búsqueda a cada iteración. Esto permite explorar otras representaciones del problema que podrían "facilitar" la búsqueda.

Ejemplo de codificación binaria usando códigos delta.

Parámetros numéricos	0	1	2	3	4	5	6	7
Codificación binaria	000	001	010	011	100	101	110	111
Cambios numéricos	0	1	2	3	-3	-2	-1	-0
Codificación delta	000	001	010	011	111	110	101	100

9.7 Problemas Propuestos

1. Implemente un esquema de auto-adaptación para un algoritmo genético. Incluya los porcentajes de cruza y mutación como variables adicionales, codificadas en la misma cadena cromosómica. Permita que el rango de la cruza y la mutación sea un número real entre 0 y 1. Defina una función de aptitud en la que se premie a un individuo según la aptitud de sus descendientes.

2. Implemente el esquema de auto-adaptación propuesto por Srinivas y Patnaik [211].

3. Realice un estudio en el que evalúe el efecto de los parámetros en el desempeño de un algoritmo genético usado para **minimizar** la siguiente función:

$$\frac{1}{1/K + \sum_{j=1}^{25} f_j^{-1}(x_1, x_2)}, \text{ donde } f_j(x_1, x_2) = c_j + \sum_{i=1}^{2}(x_i - a_{ij})^6, \quad (9.1)$$

$$\text{donde: } -65.536 \leq x_i \leq 65.536,\ K = 500,\ c_j = j, \text{ y:} \quad (9.2)$$

$$[a_{ij}] = \begin{bmatrix} -32 & -16 & 0 & 16 & 32 & -32 & -16 & \cdots & 0 & 16 & 32 \\ -32 & -32 & -32 & -32 & -32 & -16 & -16 & \cdots & 32 & 32 & 32 \end{bmatrix} \quad (9.3)$$

Considere una precisión de 3 dígitos después del punto decimal. Esta es la función F5 de DeJong.

4. Considere la siguiente función de aptitud: $f(x)$=número de unos en x, donde x es un cromosoma de longitud 4. Suponga que el AG ha corrido durante tres generaciones, con las siguientes poblaciones:

generación 0:	1001, 1100, 0110, 0011
generación 1:	1101, 1101, 0111, 0111
generación 2:	1001, 1101, 1111, 1111

Calcule el desempeño en línea y fuera de línea después de cada generación.

Capítulo 10

Manejo de Restricciones

Hasta ahora hemos estudiado sólo problemas sin restricciones. Sin embargo, en la práctica normalmente tenemos problemas con restricciones de diferentes tipos (igualdad, desigualdad, lineales y no lineales), tales como:

$$Minimizar f(x)$$

Sujeto a:

$$g_i \leq 0 \qquad i = 1, 2, \ldots, n$$

Pero el algoritmo genético opera como una técnica de optimización sin restricciones, por lo que tenemos que diseñar algún mecanismo que permita incorporar la información pertinente sobre la violación de restricciones en la función de aptitud. Este capítulo habla precisamente de diferentes esquemas para hacer esto.

10.1 Funciones de Penalización

Son la técnica más común de incorporación de restricciones en la función de aptitud. La idea básica es extender el dominio de la función de aptitud usando:

$$fitness_i(X) = f_i(X) \pm Q_i$$

donde:

$$Q_i = c \times \textstyle\sum_{i=1}^{n} g_i(X)^2$$

para todas las restricciones violadas. En esta expresión, c es un factor de penalización definido por el usuario.

Hay al menos 3 formas de penalizar a un individuo de acuerdo a su violación de las restricciones [187]:

- Puede penalizársele simplemente por no ser factible, sin usar ninguna información sobre qué tan cerca se encuentra de la región factible.
- Puede usarse la 'cantidad' de infactibilidad de un individuo para determinar su penalización correspondiente.
- Puede usarse el esfuerzo de 'reparar' al individuo (o sea, el costo de hacerlo factible) como parte de la penalización.

Richardson et al. [187] definieron algunas de las reglas básicas para diseñar una función de penalización:

1. Las penalizaciones que son funciones de la distancia a la zona factible son mejores que aquellas que son sólo funciones del número de restricciones violadas.
2. Para un problema con pocas variables y pocas restricciones, una penalización que sea sólo función del número de restricciones violadas no producirá ninguna solución factible.
3. Pueden construirse buenos factores de penalización a partir de 2 factores: el costo de cumplimiento máximo y el costo de cumplimiento esperado. El primero de ellos se refiere al costo de hacer factible a una solución infactible.
4. Las penalizaciones deben estar cerca del costo de cumplimiento esperado, pero no deben caer frecuentemente por debajo de él. Entre más preciso sea el factor de penalización, mejores resultarán las soluciones producidas. Cuando una penalización frecuentemente subestime el costo de cumplimiento, el proceso de búsqueda fallará.

Existen, sin embargo, varios problemas para definir una función de penalización:

1. No es obvio combinar los 2 factores de los que hablan Richardson et al. [187] en una función de penalización.

2. Definir los valores óptimos del factor de penalización es una tarea virtualmente imposible a menos que conozcamos el problema a fondo, en cuyo caso puede diseñarse una función de penalización a la medida, pero hacerlo resultará de cualquier forma innecesaria ya que el óptimo podrá determinarse por métodos analíticos exactos.

3. El costo del cumplimiento esperado normalmente tiene que estimarse mediante métodos alternativos (por ejemplo, estimando el grado de violación de las restricciones) que pueden ser difíciles de implementar.

A continuación revisaremos rápidamente diversas variantes de la función de penalización que se han propuesto en la literatura, comentando brevemente sobre sus desventajas.

10.1.1 Pena de Muerte

En las estrategias evolutivas ha sido popular una técnica a la que se le conoce como "pena de muerte", y que consiste en asignar una aptitud de cero a un individuo que no sea factible, y tomar el valor de la función de aptitud para los que sí lo sean [204, 12].

10.1.1.1 Análisis

Esta técnica es muy eficiente porque no tenemos qué re-calcular las restricciones o la función objetivo ni tenemos que reparar a los individuos no factibles. Sin embargo, tiene varios inconvenientes. Por ejemplo, si la población inicial no tiene ningún individuo factible, la búsqueda no progresará (habrá estancamiento) porque todos los individuos tendrán la misma aptitud.

Algunos autores como Michalewicz [160] han explorado esta técnica, concluyendo que sólo puede usarse en espacios de búsqueda convexos y en aquellos casos en los que la zona factible constituya una parte razonablemente grande del espacio total de búsqueda. Asimismo, esta técnica sólo puede lidiar con restricciones de desigualdad.

10.1.2 Penalizaciones estáticas

En esta técnica, introducida por Homaifar, Lai y Qi [129], la idea es definir varios niveles de violación y elegir un coeficiente de violación para cada uno de ellos,

de manera que el coeficiente de penalización se incremente conforme alcanzamos niveles más altos de violación de las restricciones.

Un individuo se evalúa utilizando:

$$fitness_i = f_i(X) + \sum_{j=1}^{n} R_{k,j} g_j(X)$$

donde: $R_{i,j}$ son los coeficientes de penalización utilizados y $k = 1, 2, \ldots, l$, siendo l los niveles de violación definidos por el usuario.

10.1.2.1 Análisis

El principal problema de esta técnica es que requiere la definición de $n(2l + 1)$ parámetros. De tal forma que si tenemos un problema moderadamente pequeño, con 6 restricciones y 2 niveles, tendríamos que definir 30 parámetros, lo cual a todas luces es excesivo.

10.1.3 Penalizaciones Dinámicas

Joines y Houck [136] propusieron una técnica en la que los factores de penalización cambian con respecto al tiempo.

Los individuos de la generación t se evalúan de acuerdo a:

$$fitness_i(X) = f_i(X) + (C \times t)^{\alpha} \sum_{j=1}^{n} |g_i(X)|^{\beta}$$

donde C, α y β son constantes definidas por el usuario. Los valores sugeridos por los autores para las constantes son: $C = 0.5$, $\alpha = 1$ y $\beta = 1$ ó 2.

10.1.3.1 Análisis

La técnica es muy susceptible a los valores de los parámetros y suele converger prematuramente cuando éstos no son seleccionados adecuadamente. Sin embargo, parece funcionar muy bien cuando la función objetivo es cuadrática.

10.1.4 Uso de recocido simulado

Michalewicz y Attia [162] consideraron una técnica para manejo de restricciones que usa el concepto de recocido simulado: los coeficientes de penalización se cambian cada cierto número de generaciones (cuando el algoritmo ha quedado atrapado en un óptimo local).

Los individuos se evalúan usando:

$$fitness_i(X) = f_i(X) + \frac{1}{2\tau}\sum_{j=1}^{n} g_j(X)^2$$

donde: τ representa el horario de enfriamiento y es una función que debe ser definida por el usuario.

Michalewicz y Attia sugieren usar: $\tau_0 = 1$ y $\tau_f = 0.000001$, con incrementos $\tau_{i+1} = 0.1 \times \tau_i$.

10.1.4.1 Análisis

El principal problema de esta técnica es la definición del horario de enfriamiento.

10.1.5 Penalizaciones Adaptativas

Bean y Hadj-Alouane [20] desarrollaron una técnica para adaptar penalizaciones con base en un proceso de retroalimentación del ambiente durante la corrida de un algoritmo genético.

Cada individuo es evaluado usando:

$$fitness_i(X) = f_i(X) + \lambda(t)\sum_{j=1}^{n} g_j(X)^2$$

donde: $\lambda(t)$ se actualiza cada t generaciones usando las siguientes reglas:

$$\lambda(t) = \begin{cases} (\frac{1}{\beta_1})\lambda(t) & \text{si caso \# 1} \\ \beta_2\lambda(t) & \text{si caso \# 2} \\ \lambda(t) & \text{de lo contrario} \end{cases} \tag{10.1}$$

donde $\beta_1, \beta_2 > 1$ y con valores diferentes (para evitar ciclos).

El caso # 1 ocurre cuando el mejor individuo en las últimas k generaciones fue siempre factible. El caso # 2 ocurre cuando dicho individuo no fue nunca factible.

Esta técnica lo que hace entonces es disminuir la penalización cuando el mejor individuo resulta consistentemente factible y aumentar la penalización cuando resulta infactible. Si estos cambios son intermitentes (es decir, el mejor individuo es a veces factible y a veces no), entonces la penalización no se cambia.

Obviamente, es necesario definir el valor inicial λ_0.

10.1.5.1 Análisis

El principal problema de esta técnica es cómo elegir el factor inicial de penalización y el intervalo generacional (o sea, k) de manera que el monitoreo de factibilidad produzca resultados razonables.

Una k muy grande o muy pequeña puede hacer que la función de penalización nunca cambie, en cuyo caso la técnica se reduce a una penalización estática tradicional.

Finalmente, tampoco está claro cómo definir buenos valores de β_1 y β_2.

10.1.6 Algoritmo genético segregado

Esta técnica fue propuesta por Le Riche et al. [188] y consiste en usar 2 funciones de penalización en vez de una, empleando para ello dos poblaciones. Estas 2 funciones intentan balancear las penalizaciones moderadas con las fuertes.

Inicialmente, se divide la población en 2 grupos, de manera que los individuos de cada grupo se evalúan usando un factor de penalización distinto. Después se elige a los mejores individuos de cada grupo para ser padres de la siguiente generación, lo que hace que se combinen individuos factibles con infactibles (si se usa un factor grande y otro pequeño), manteniendo la diversidad y evitando quedar atrapado en máximos locales.

En la implementación original de esta técnica se utilizaron jerarquías lineales para decrementar la presión de selección y se le aplicó a un problema de diseño de elementos compuestos con gran éxito.

10.1.6.1 Análisis

El principal inconveniente de esta técnica es la forma de elegir los factores de penalización de las 2 poblaciones, lo cual es difícil de hacer cuando no hay información disponible sobre el problema que tratamos de resolver.

10.1.7 Penalización con base en la factibilidad

Esta técnica fue propuesta por Deb [66] y consiste en evaluar un individo de acuerdo a:

$$fitness_i(X) = \begin{cases} f_i(X) & \text{Si la solución es factible} \\ f_{peor} + \sum_{j=1}^{n} g_j(X) & \text{de lo contrario} \end{cases} \tag{10.2}$$

donde f_{peor} es el valor de la función objetivo de la peor solución factible de la población. Si no hay ninguna solución factible en la población, entonces f_{peor} se hace igual a cero.

Deb [66] usa torneo binario aplicando las siguientes reglas:

1. Una solución factible siempre es preferida sobre una no factible.

2. Entre 2 soluciones factibles, la que tenga mejor valor de su función objetivo es seleccionada.

3. Entre 2 soluciones no factibles, la que viole el menor número de restricciones es elegida.

10.1.7.1 Análisis

Uno de los problemas de esta técnica es que tiene dificultades para mantener diversidad en la población, y para ello se requiere del uso de nichos [68] y porcentajes de mutación inusualmente altos.

10.2 Técnicas que usan conceptos de Optimización Multiobjetivo

Técnicas para manejo de restricciones

- Funciones de penalización.
- Representaciones y operadores especiales.
- Algoritmos de reparación.
- Separación de restricciones y objetivos.
- Métodos híbridos.

Ejemplos

- Representaciones y operadores especiales: GENOCOP [164].
- Algoritmos de reparación: GENOCOP III [163].
- Separación de restricciones y objetivos: Técnicas de optimización multiobjetivo
- Métodos híbridos: Sistema inmune [233, 55].

Técnicas basadas en conceptos Multiobjetivo para Manejo de Restricciones.

Idea base: Un problema con un solo objetivo y restricciones, puede transformarse en un problema multiobjetivo.

Redefinición del problema: se tiene

$$\vec{v} = (f(\vec{x}), f_1(\vec{x}), \cdots, f_m(\vec{x}))$$

donde se quiere encontrar $\vec{x}$ tal que
$f_i(\vec{x}) \leq 0, i = 1, \cdots, m$ y $f(\vec{x}) \leq f(\vec{y}), \forall \vec{y}$ factible.

Conceptos Multiobjetivo Utilizados

- **Esquema Poblacional.** Consiste en dividir a la población en subpoblaciones, donde cada una de ellas enfocará sus esfuerzos en optimizar uno de los objetivos del problema (VEGA [199]).
- **No dominancia.** Un punto $\vec{x}^* \in \Omega$ es una **solución no dominada** si no hay $\vec{x} \in \Omega$ tal que: $f_i(\vec{x}) \leq f_i(\vec{x}^*)$ para $i = 1, \cdots, k$ y para al menos un valor i, $f_i(\vec{x}) < f_i(\vec{x}^*)$.
- **Jerarquización de Pareto.** Propuesta por Goldberg [105]. Otorgar a cada solución un valor de jerarquía (número de soluciones dentro de la población que dominan a esa solución determinada) con el objeto de encontrar el conjunto de soluciones no dominadas por el resto de la población. Su costo es $O(kM^2)$.

10.2.1 COMOGA

Propuesto por Surry et al. [216]. Combina las jerarquías de Pareto con la propiedad de VEGA de favorecer soluciones extremas en un AG no generacional.

- Calcula violación de restricciones.
- Jerarquias de Pareto (no dominancia).
- Calcula aptitud.
- Selecciona una parte de la población con base en su aptitud y otra parte con base en jerarquías.

Figura 10.1: Ejemplo de un frente de Pareto.

- Aplica operadores genéticos.
- Ajusta el factor de selección.

10.2.2 Direct Genetic Search

Propuesta por Parmee y Purchase [176]. Utiliza VEGA para dirigir la búsqueda de un algoritmo evolutivo hacia la zona factible. Posteriormente, utiliza una técnica de búsqueda local, en un hipercubo formado a partir de la solución arrojada por VEGA, para encontrar el óptimo global.

10.2.3 Reducción de objetivos, no dominancia y búsqueda lineal.

Propuesta por Camponogara y Taludkar [37]. Reduce el problema a sólo 2 objetivos:

- La función objetivo original.

- Minimizar la cantidad de violación de un individuo.

$$\Phi(\bar{x}) = \sum_{i=0}^{N} max(0, g_i(\bar{x}))$$

A partir de los frentes de Pareto obtenidos en cada generación se recombinan y se obtienen dos soluciones (una domina a la otra) para determinar una dirección de búsqueda. De ahí inicia una búsqueda lineal.

10.2.4 Selección por torneo con reglas especiales.

Propuesto por Jiménez y Verdegay [135]. Utiliza selección por torneo, población no traslapable y elitismo. Reglas del torneo:

- Un individuo factible es preferible a otro factible con menor aptitud.
- Un individuo factible es preferible a otro no factible, sin importar su aptitud.
- Entre dos individuos no factibles, se prefiere aquel que viole en menor cantidad las restricciones del problema.

10.2.5 VEGA para manejo de restricciones

Propuesto por Coello [46]. Divide a la población en $m + 1$ subpoblaciones (m restricciones). Cada subpoblación (de las m) se evalúa con una restricción y la restante se evalúa con la función objetivo. Se seleccionan los individuos (integrados todos en una sola población) y se aplican operadores genéticos.

Criterio de selección para las subpoblaciones que se evalúan con una restricción del problema:

if $(g_j(x) < 0)$	**then** $fitness = g_j(x)$
else if$(v <> 0)$	**then** $fitness = -v$
else $fitness = f$	

10.2.6 Jerarquías en distintos espacios y restricciones de apareamiento

Propuesto por Ray et al. [182]. Utiliza el concepto de no dominancia en el espacio de las funciones objetivo y en el de las restricciones. Jerarquiza la población en el espacio combinado de manera que aquellos con jerarquía alta se insertan en la nueva población. Los restantes individuos se obtienen de la cruza (utiliza cruza uniforme y random mix.) de un individuo A con uno de dos candidatos B y C:

- **Problema sin restricciones:** Se escogen A, B y C de acuerdo a su rango en el espacio de los objetivos.
- **Problema moderadamente restringido:** Se escoge a A de acuerdo a su rango en el espacio de los objetivos, y a B y C del espacio de las restricciones.
- **Problema altamente restringido:** Se escogen A, B y C de acuerdo a su rango en el espacio de las restricciones .

Para escoger entre B y C se tienen los siguientes criterios:

- Ambos factibles, se escoge aquel con mayor jerarquía en el espacio de los objetivos.
- Ambos no factibles, se escoge aquel con mayor jerarquía en el espacio de las restricciones.
- Uno factible y otro no factible, se escoge aquel que sea factible.

10.2.7 MOGA y no dominancia

Propuesto por Coello [45]. Utiliza jerarquías de Pareto de manera similar a MOGA y auto adaptación en cruza y mutación.

Reglas para criterio de selección:

- Todo individuo factible es preferible a otro no factible.
- Si ambos individuos son no factibles, se prefiere aquel que viole en menor número las restricciones del problema.
- Si ambos individuos son no factibles y violan el mismo número de restricciones, se prefiere aquel que viole en menor cantidad las restricciones del problema.

Las jerarquías se establecen al comparar a cada individuo con toda la población: Se establece un contador que se incrementa cada vez que se encuentra un individuo "mejor" en:

- Aptitud.
- Factibilidad.
- Número de restricciones violadas.
- Cantidad de violación de restricciones.

La aptitud está en función de la jerarquía del individuo. La selección se lleva a cabo mediante muestreo universal estocástico.

10.2.8 NPGA para manejo de restricciones

NPGA surge como una de las interpretaciones de la idea de Goldberg [105] de utilizar un AG para optimizar un vector de funciones. La selección de individuos es por torneo, modificado para optimización multiobjetivo. Utiliza nichos para mantener diversidad.

Algoritmo

1. Seleccionar candidatos.
2. Seleccionar subconjunto de soluciones.
3. Verifica no dominancia de ambos candidatos con respecto al subconjunto de soluciones.
4. Seleccionar aquel candidato no dominado.
5. En caso de empate (ambos dominados o ambos no dominados) seleccionar aquel con menor valor de su contador de nicho.

Proporción de Selección

Se introduce un parámetro llamado *selection ratio* (S_r) el cual indica el mínimo número de individuos que serán seleccionados a través de 4 criterios de selección.

Figura 10.2: Diagrama del algoritmo de NPGA para manejo de restricciones.

Los restantes se seleccionarán de una forma puramente aleatoria.

Los criterios de comparación usados son los siguientes: Entre dos individuos, si:

- **Ambos son factibles:** El individuo con la aptitud más alta gana.
- **Uno es factible y el otro es no factible:** El individuo factible gana.
- **Ambos son no factibles:** Revisión de no dominancia
- **Ambos son dominados o no dominados:** El individuo con la menor cantidad de violación gana.

El *selection ratio* (S_r) controla la diversidad de la población. Normalmente se utiliza un valor superior a 0.8 (Sr(0.8).

10.2.9 Constrained Robust Optimization

Propuesto por Ray et al. [181]. Se basa en la idea de generar soluciones que no sean sensibles a variaciones en los parámetros y que sean cercanas o iguales al óptimo.

Es un procedimiento costoso (requiere de un número mayor de evaluaciones de la función de aptitud). Genera matrices con base en los valores de restricciones y objetivos como en su propuesta anterior. Agrega el valor de violación de restricciones de k vecinos. Jerarquiza la población como lo hace NSGA I y II.

La selección se basa en las jerarquías en los distintos espacios, balanceando entre restricciones y objetivos.

Enfoque en línea. Permite manejo de restricciones. No se ha probado ampliamente en Optimización Global.

10.2.10 EMO para optimización con restricciones, satisfacción de restricciones y programación por metas

Propuesto por Jiménez et al. [134]. Utiliza un AG no generacional, un proceso de preselección y redefinición de las funciones objetivo (y/o restricciones). El proceso de preselección ayuda a la formación de nichos implícito y es un mecanismo elitista.

La selección de padres es puramente aleatoria.

1. Se cruzan los padres n veces para producir n hijos.

2. Los n hijos se mutan para obtener 2*n hijos.

3. El mejor de los primeros n descendientes reemplaza al primer padre

4. El mejor de los hijos mutados reemplaza al segundo padre.

Las comparaciones se basan en no dominancia. La transformación de las funciones de aptitud y restricciones asigna aptitudes altas a individuos factibles y aptitudes bajas a aquellos no factibles, buscando acerca a las soluciones a la zona factible del espacio de búsqueda.

Tiene un mecanismo más balanceado para aproximarse a la zona factible pero requiere de mucho tiempo computacional para obtener resultados aceptables. No se ha probado ampliamente para optimización global.

10.2.11 Asignación de aptitud en 2 fases

Propuesta por Wu & Azarm [232]. Asigna aptitud a un individuo en dos fases. Propuesta para EMO.

I Fase primaria:

- Paso 1: Se calcula el número de soluciones que dominan a un individuo x $\forall x$ de la población llamado valor dominante (jerarquización).

- Paso 2: Se calcula de nuevo la jerarquía del individuo pero ahora con sólo 2 objetivos: el valor dominante y la extensión de la violación de restricciones (número de restricciones violadas + cantidad de violación normalizada)
- Se asigna un valor de aptitud con base en la segunda jerarquía.

II Fase secundaria:

- Utiliza el concepto de "afinidad" para seleccionar al segundo padre (el primero se selecciona de acuerdo a la aptitud de la fase primaria).
- Individuos que son no- dominados con respecto al primer padre, que violan menos restricciones comunes y que tienen menor valor de extensión de violación se prefieren.
- Entre los individuos dominados por el primer padre, se prefieren aquellos con menor valor de extensión de violación.

No ha sido probado para Optimización Global no lineal, sólo se ha probado en un problema a nivel multiobjetivo. La doble jerarquización hace costoso el método.

10.2.12 Otras propuestas

- **CMEA (Contrained Method-Based Evolutionary Algorithm)** de Ranjithan et al. [179]. Utiliza conceptos de programación matemática con restricciones para problemas de optimización multiobjetivo.
- **Manejo de restricciones con incertidumbre o ruido para Optimización Multiobjetivo** de Hughes [132], jerarquiza de manera integrada (tomando en cuenta restricciones y prioridades) a la población con funciones simples basadas en probabilidades.

Estas técnicas sólo se han probado en optimización multiobjetivo con restricciones y no para problemas de optimizacion global.

10.3 Problemas Propuestos

1. Investigue en qué consiste la técnica denominada "decodificadores", para manejo de restricciones. Analice el comportamiento de la técnica e identi-

fique alguna propuesta al respecto para problemas de optimización en espacios continuos. Se recomienda leer:

- Slawomir Koziel and Zbigniew Michalewicz, "A Decoder-based Evolutionary Algorithm for Constrained Parameter Optimization Problems", In T. Bäck, A. E. Eiben, M. Schoenauer, and H.-P. Schwefel, editors, *Proceedings of the 5th Parallel Problem Solving from Nature (PPSN V)*, pages 231–240, Amsterdam, September 1998. Springer-Verlag.
- Slawomir Koziel and Zbigniew Michalewicz, "Evolutionary Algorithms, Homomorphous Mappings, and Constrained Parameter Optimization", *Evolutionary Computation*, Vol. 7, No. 1, pp. 19–44, 1999.

2. Investigue el funcionamiento de una técnica conocida como "memoria conductista" (*behavioral memory*), analizando sus ventajas y desventajas, así como las posibles dificultades para implementarla (si es que hubiese alguna). Lea:

   - Marc Schoenauer and Spyros Xanthakis, "Constrained GA Optimization", In Stephanie Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 573–580. Morgan Kauffman Publishers, San Mateo, California, July 1993.

3. Investigue la forma en la que se han utilizado los algoritmos culturales para manejar restricciones en problemas de optimización en espacios continuos. Analice las ventajas y desventajas de esta técnica, así como las posibles dificultades para implementarla. Se le recomienda leer:

   - Xidong Jin and Robert G. Reynolds, "Using Knowledge-Based Evolutionary Computation to Solve Nonlinear Constraint Optimization Problems: a Cultural Algorithm Approach", In *1999 Congress on Evolutionary Computation*, pages 1672–1678, Washington, D.C., July 1999. IEEE Service Center.
   - Chan-Jin Chung and Robert G. Reynolds, "A Testbed for Solving Optimization Problems using Cultural Algorithms", In Lawrence J. Fogel, Peter J. Angeline, and Thomas Bäck, editors, *Evolutionary Programming V: Proceedings of the Fifth Annual Conference on Evolutionary Programming*, Cambridge, Massachusetts, 1996. MIT Press.

Capítulo 11

Software

En general, podemos clasificar el software para computación evolutiva en 3 grandes grupos:

- **Orientado a las aplicaciones:** Son "cajas negras" diseñadas para ocultar detalles de los AGs y ayudar al usuario a desarrollar aplicaciones para dominios específicos tales como las finanzas, la programación de horarios, etc.

- **Orientado a los algoritmos:** Se basan en modelos de AGs específicos y pueden subdividirse en 2 grupos:

    1. Sistemas Específicos: Contienen un solo algoritmo.
    2. Bibliotecas: Contienen una gama de algoritmos y operadores genéticos que pueden estar disponibles sólo de manera pre-compilada.

- **Cajas de herramientas (tool kits):** Sistemas de programación que proporcionan muchas utilerías, algoritmos y operadores genéticos que pueden usarse para cualquier tipo de aplicación y que normalmente se proporcionan en forma de código fuente (al menos de manera parcial).

A su vez, las cajas de herramientas se pueden subdividir en 2 grupos:

1. Sistemas Educativos: Su objetivo es ayudar a los usuarios novatos a practicar los conceptos de computación evolutiva recién aprendidos. Normalmente estos sistemas tienen un número relativamente pequeo de opciones para configurar un cierto algoritmo.

2. Sistemas de propósito general: Proporcionan un rico conjunto de herramientas para programar cualquier tipo de AG y aplicarlo a lo que se desee. En algunos casos, incluso permiten que el usuario experto modifique partes del código de acuerdo a sus propias necesidades.

11.1 Software de Dominio Público

- **Nombre**: BUGS (*Better to Use Genetic Systems*)

  Descripción: Programa interactivo para demostrar el uso de un a lgoritmo genético. El usuario desempea el papel de la función de aptitud y trata de evolucionar una cierta forma de vida artificial (curvas).

  El uso de este programa suele facilitar la comprensión de lo que son los AGs y cómo funcionan para aquellos novatos en el área. Además de demostrar los operadores genéticos fundamentales (selección, cruza y mutación), BUGS permite visualizar el "desvío genético" (*genetic drift*) y la convergencia prematura.

  Lenguaje: C bajo X Windows.

  Autor: Joshua Smith (jrs@media.mit.edu)

  Disponibilidad: http://www.aic.nrl.navy.mil/pub/galist/src/BUGS.tar.Z

- **Nombre**: Genesis

  Descripción: Implementación de un AG que tiene gran valor histórico por haber sido el primer programa de su tipo liberado en el dominio público.

  Lenguaje: C para Unix.

  Autor: John J. Grefenstette (gref@aic.nrl.navy.mil)

  Disponibilidad: http://www.aic.nrl.navy.mil/pub/galist/src/genesis.tar.Z

- **Nombre**: GENEsYs

  Descripción: Implementación de un AG basada en GENESIS que incluye extensiones y nuevas funciones para propósitos experimentales. Por ejemplo, cuenta con selección mediante jerarquías lineales, selección de Boltzmann, selección (+), cruza uniforme, recombinación discreta e intermedia, auto-adaptación de porcentajes de mutación, etc.

También cuenta con una serie de funciones objetivo, incluyendo las funciones de De Jong, funciones continuas de alto grado de complejidad, una instancia del problema del viajero, funciones binarias y una función fractal. Finalmente, tiene también utilerías para monitorear resultados tales como vaciados de mapas de bit de la población, varianza de las variables objeto y de los porcentajes de mutación, etc.

Lenguaje: C para Unix.

Autor: Thomas Bäck (baeck@ls11.informatik.uni-dortmund.de)

Disponibilidad: http://www.aic.nrl.navy.mil/pub/galist/src/GENEsYs-1.0.tar.Z

- **Nombre**: DGenesis

  Descripción: Implementación de un AG distribuido desarrollada a partir de GENESIS 5.0. Corre en una red de estaciones de trabajo operando con Unix. Cada subpoblación es manejada por un proceso Unix y la comunicación entre ellas se efectúa usando sockets de Berkeley Unix.

  Lenguaje: C para Unix.

  Autor: Erick Cantú Paz (cantupaz@dirichlet.llnl.gov)

  Disponibilidad: http://www.aic.nrl.navy.mil/pub/galist/src/dgenesis-1.0.tar.Z

- **Nombre**: GECO (*Genetic Evolution through Combination of Objects*)

  Descripción: Ambiente de trabajo orientado a objetos para implementar prototipos de algoritmos genéticos. Usa el CLOS (Common LISP Object System) y cuenta con abundante documentación y algunos ejemplos de uso.

  Lenguaje: Common LISP para Macintosh o Unix.

  Autor: George P. Williams, Jr. (george.p.williams@boeing.com)

  Disponibilidad: http://www.aic.nrl.navy.mil/pub/galist/src/GECO-v2.0.tar.Z

- **Nombre**: GALOPPS (*Genetic Algorithm Optimized for Portability and Parallelism*)

  Descripción: Un sistema de AGs paralelos en el que usuario puede escoger:

  - El tipo de problema (con valores numéricos o permutaciones)
  - El tipo de cruza (de entre 7 posibles) y mutación (de entre 4 posibles)
  - El tipo de selección (de entre 6 posibles)

- Probabilidades de los operadores, escalamiento de la función de aptitud, frecuencia y patrones de migración
- Criterios de detención
- Elitismo (opcional)
- Uso de diferente representación para cada subpoblación, con transformación de los migrantes
- Inversión al nivel de subpoblaciones
- Control sobre el reemplazo poblacional, incluyendo "crowding" y reemplazo aleatorio
- Selección de parejas, usando prevención de incestos
- Selección de migrantes

El usuario puede definir una función objetivo (usando una plantilla) y cualquier función auxiliar que necesite. El sistema puede correr una o varias subpoblaciones, en una o varias PCs, estaciones de trabajo o Macs. El sistema corre de manera interactiva (con una interfaz gráfica o de texto) o desde archivos.

Puede interrumpirse y recomenzarse fácilmente. Existe una versión en PVM que incluso mueve los procesos automáticamente cuando una estación de trabajo está ocupada. Viene con 14 ejemplos que incluyen las funciones de De Jong, la carretera real, el viajero, etc.

Lenguaje: C para Unix.

Autor: Erik D. Goodman (goodman@egr.msu.edu)

Disponibilidad: http://GARAGE.cps.msu.edu/

- **Nombre**: ESCaPaDE

  Descripción: Sofisticado sistema que permite correr experimentos con algoritmos evolutivos tales como la estrategia evolutiva. El sistema cuenta con 2 tablas internas: una de funciones objetivo y una de monitores de datos, lo que permite una fácil implementación de funciones para monitorear todo tipo de información dentro del algoritmo evolutivo.

  Lenguaje: C para Unix (con rutinas en FORTRAN)

  Autor: Frank Hoffmeister (hoffmeister@ls11.informatik.uni-dortmund.de)

  Disponibilidad: Por e-mail a Hoffmeister

- **Nombre**: GANNET (*Genetic Algorithm/Neural NETwork*)

  Descripción: Paquete de software que permite evolucionar redes neuronales binarias. Ofrece toda una variedad de oPCiones de configuración relacionadas con los valores de los operadores genéticos.

  La evolución de las redes neuronales se basa en 3 funciones de aptitud: precisión entre las entradas y salidas, "estabilidad" de la salida y tamao de la red. Soporta redes con entradas y salidas binarias, con neuronas de 2 ó 4 entradas y pesos de entre -3 a $+4$, permitiendo hasta 250 neuronas en una red.

  Lenguaje: C para Unix (con rutinas en FORTRAN)

  Autor: Jason Spofford

  Disponibilidad: http://fame.gmu.edu/~dduane/thesis

- **Nombre**: GENOCOP (*Genetic Algorithm for Numerical Optimization for COnstrained Problems*)

  Descripción: Paquete de optimización numérica para funciones con cualquier cantidad de restricciones lineales (igualdades y desigualdades).

  Lenguaje: C para Unix.

  Autor: Zbigniew Michalewicz (zbyszek@uncc.edu)

  Disponibilidad: http://www.aic.nrl.navy.mil/pub/galist/src/genocop.tar.Z

- **Nombre**: GPC++

  Descripción: Biblioteca de clases en C++ para desarrollar aplicaciones de programación genética. Esta biblioteca define una jerarquía de clases y uno de sus componentes integrales es la capacidad de producir funciones definidas automáticamente.

  Lenguaje: C++ para Unix/MSDOS.

  Autor: Thomas Weinbrenner (thomasw@thor.emk.e-technik.th-darmstadt.de)

  Disponibilidad: http://www.emk.e-technik.thdarmstadt.de/~thomasw/gp.html

- **Nombre**: GPEIST (*Genetic Programming Environment in SmallTalk*)

  Descripción: Ambiente de programación genética en Smalltalk que puede correrse en HP/Sun/PC. Permite distribución de subpoblaciones en varias estaciones de trabajo (con intercambios entre ellas a ciertos intervalos)

Lenguaje: Smalltalk

Autor: Tony White (arpw@bnr.ca)

Disponibilidad: ENCORE (*The EvolutioNary COmputation REpository network*)

URL: http://www.cs.bham.ac.uk/Mirrors/ftp.de.uu.net/EC/clife/

- **Nombre**: PGAPack

  Descripción: Biblioteca de propósito general para desarrollar AGs paralelos. Incluye:

  - Capacidad de invocación desde FORTRAN o C.
  - Soporte de redes de estaciones de trabajo, arquitecturas en paralelo y uniprocesadores.
  - Tipos de datos binarios, enteros, reales y caracteres (nativos).
  - Soporte de nuevos tipos de datos.
  - Interfaz fácil de usar.
  - Niveles múltiples de acceso para usuarios expertos.
  - Facilidades extensivas de depuración.
  - Gran cantidad de ejemplos.
  - Detallada guía del usuario.
  - Soporte de diferentes tipos de selección, cruza y mutación.

  Lenguaje: C para Unix.

  Autor: David Levine (levine@mcs.anl.gov)

  Disponibilidad: http://www.mcs.anl.gov/pgapack.html

- **Nombre**: REGAL (*RElational Genetic Algorithm Learner*)

  Descripción: Sistema distribuido basado en AGs diseñado para aprender descriPCiones de conceptos en lógica de primer orden a partir de ejemplos. Se basa en un operador llamado "Sufragio Universal" que permite la probable convergencia asintótica de la población a un estado de equilibrio en el que coexisten varias especies.

  Lenguaje: C para Unix, usando PVM y Tcl/Tk

Autor: Attilio Giordana (attilio@di.unito.it)

Disponibilidad: ftp://ftp.di.unito.it/pub/MLprog/REGAL3.2

- **Nombre**: SCS-C (*Simple Classifier System in C*)

  Descripción: Versión en C del Sistema Clasificador Simple proporcionado en el libro de Goldberg.

  Lenguaje: C para Unix

  Autor: Jrg Heitkoetter (joke@de.uu.net)

  Disponibilidad: ENCORE

11.2 Software Comercial

- **Nombre**: ActiveGA

  Descripción: Un OLE que usa un algoritmo genético para solucionar un problema dado. Algunas de sus funciones incluidas son:

  - Selección de torneo o ruleta.
  - Parámetros del algoritmo genético definidos por el usuario.
  - Invisible durante tiempo de ejecución.
  - Ejemplos en Excel, Visual BASIC y Visual C++.

  Precio: $99 dólares

- **Nombre**: Evolver

  Descripción: Paquete de algoritmos genéticos para Windows. Los principiantes pueden usar el módulo para Excel en sus problemas.

  Los usuarios avanzados pueden usar el API incluido para desarrollar sus propias aplicaciones, las cuales pueden ser monitoreadas en tiempo real usando el *EvolverWatcher*.

  Precio: $349 dólares

- **Nombre**: PC-Beagle

  Descripción: Programa que examina una base de datos con ejemplos y usa técnicas de aprendizaje de máquina para crear un conjunto de reglas de decisión que clasifiquen los ejemplos.

El sistema contiene 6 componentes principales, de los cuales uno usa algoritmos genéticos.

Precio: £ 69.

- **Nombre**: MicroGA

  Descripción: Herramienta que permite la integración de algoritmos genéticos en cualquier pieza de software. Se trata de un ambiente de programación en C++ que viene en código fuente con documentación y 3 ejemplos completos.

  También incluye un generador de código que permite crear aplicaciones completas de manera interactiva y sin escribir una sola línea de código. Soporte para Macintosh y MS Windows.

  Precio: $249 dólares

- **Nombre**: GEATbx (*Genetic and Evolutionary Algorithm Toolbox*)

  Descripción: Conjunto de funciones en MATLAB que permiten implementar diferentes tipos de algoritmos genéticos para optimización con uno o varios objetivos.

  - Soporta diferentes tipos de selección (universal estocástica, torneo, jerarquías lineales y no lineales, etc.).
  - Incorpora diferentes técnicas de cruza (un punto, dos puntos, uniforme, intermedia, discreta, etc.).
  - Incluye mutación para representación entera, real y binaria.
  - Permite diferentes modelos poblacionales (globales, regionales y locales).
  - Permite el monitoreo de almacenamiento de resultados (análisis en línea y fuera de línea).
  - Cuenta con diversas funciones de prueba.
  - Cuenta con documentación y un tutorial (en HTML).
  - Permite la incorporación de conocimiento específico del dominio.

  Precio: 150-250 euros.

11.3 Problemas Propuestos

1. Instale dos cajas de herramientas de dominio público y compárelas. Evalúe su alcance, facilidad de extensión, documentación, facilidad de uso y utilidad práctica.

2. Desarrolle una caja de herramientas que incluya todos los operadores que hemos visto a lo largo de este curso. Asegúrese de que sus funciones trabajen con representacjón binaria y real. Además, la representación binaria puede ser con o sin códigos de Gray. Desarrolle documentación adecuada, así como ejemplos de uso de las funciones. El código fuente deberá estar también documentado y será liberado al dominio público.

Capítulo 12

Fundamentos Teóricos

Aunque los algoritmos genéticos son simples de describir y programar, su comportamiento puede ser muy complicado, y todavía existen muchas preguntas abiertas acerca de cómo funcionan y sobre el tipo de problemas en los que son más adecuados.

La teoría tradicional de los AGs (formulada originalmente por Holland en 1975 [127]) asume que, en un nivel muy general de descripción, los AGs trabajan descubriendo, enfatizando y recombinando buenos "bloques constructores" de soluciones en una manera altamente paralelizada. La idea básica aquí es que las buenas soluciones tienden a estar formadas de buenos bloques constructores (combinaciones de bits que confieren una aptitud alta a las cadenas en las que están presentes).

Holland [127] introdujo la noción de **esquemas** para formalizar la noción de **bloques constructores**. Tal vez la forma más fácil de visualizar un espacio de búsqueda es considerando todas las cadenas y esquemas de longitud 3. Gráficamente, podemos ver el espacio de búsqueda como un cubo donde los puntos son esquemas de orden 2 y los planos son esquemas de orden 1 (ver figura 12.1). Todo el espacio de búsqueda está cubierto por un esquema de orden cero (***).

Este resultado puede generalizarse a espacios de n dimensiones, donde hablaríamos de hiperplanos. De tal manera, podemos pensar que un AG corta a través de diferentes hiperplanos en busca de los mejores bloques constructores.

¿Cuánta información obtenemos de examinar los esquemas?

Esto está relacionado con el número de esquemas únicos contenidos en la

Figura 12.1: Representación gráfica de los esquemas de longitud tres.

población.

Cualquier cadena de bits de longitud L es una instancia de 2^L esquemas diferentes.

Ejemplo: C=10 L=2

H1=1* H2=*0 H3=10 H4=**

Consecuentemente, cualquier población dada de tamaño N contiene entre 2^L y $N2^L$ esquemas diferentes, dependiendo de la diversidad de la población.

¿Cuántos esquemas hay si todas las cadenas son idénticas? 2^L.

En todos los demás casos, el número es menor o igual a $N \times 2^L$.

¿Hay algún caso en el que una población de N cadenas de L bits contenga exactamente $N \times 2^L$ esquemas diferentes?

Esto sólo ocurre cuando cuando $N \leq 1$.

Una población sin individuos o con un solo individuo contiene exactamente $N2^L$ esquemas diferentes. En todos los demás casos, las cadenas de la población comparten esquemas.

12.1 Paralelismo Implícito

Como acabamos de ver, cuando todas las cadenas de una población son iguales, hay instancias de exactamente 2^L esquemas diferentes.

Esto significa que, en una cierta generación, mientras el AG está evaluando explícitamente las aptitudes de las N cadenas en la población, está también estimando implícitamente las aptitudes promedio de un número mucho mayor de esquemas [167].

La aptitud promedio de un esquema se define como la aptitud promedio de todas las instancias posibles de ese esquema.

Supongamos que generamos aleatoriamente una población de tamaño N.

En promedio la mitad de las cadenas serán instancias de **1***...*** y la mitad serán instancias de **0***...***.

Si evaluamos las (aproximadamente) $n/2$ cadenas que son instancias de **1***...*** obtendremos un estimado de la aptitud promedio de ese esquema.

Este valor es un estimado porque las instancias evaluadas en poblaciones de tamaños típicos son sólo una pequeña muestra de todas las instancias posibles.

Así como los esquemas no se representan o evalúan explícitamente por el AG, los estimados de las aptitudes promedio de los esquemas no se calculan o almacenan explícitamente por el AG. Sin embargo, el comportamiento del AG, en términos del incremento y decremento en números de instancias de esquemas dados en la población, puede describirse como si realmente estuviera calculando y almacenando estos promedios.

Veamos el siguiente ejemplo:

Defina la aptitud f de una cadena binaria x con longitud $l = 4$ como el entero representado por el número binario x (por ejemplo, $f(0011) = 3$).

¿Cuál es la aptitud promedio del esquema 1*** bajo f?

Para el esquema 1***, hay 2^3 cadenas posibles:

Bin	Aptitud	
1000	8	
1001	9	
1010	10	
1011	11	
1100	12	apt. prom= $\frac{92}{8} = 11.5$
1101	13	
1110	14	
1111	15	
	$\sum = 92$	

¿Cuál es la aptitud promedio del esquema 0*** bajo f?

Bin	Aptitud	
0000	0	
0001	1	
0010	2	
0011	3	
0100	4	apt. prom= $\frac{28}{8} = 3.5$
0101	5	
0110	6	
0111	7	
	$\sum = 28$	

12.2 Derivación del Teorema de los Esquemas

Podemos calcular la dinámica aproximada del incremento y decremento en las instancias de los esquemas de la manera siguiente.

Hagamos que H sea un esquema con al menos una instancia presente en la población en la generación t.

Hagamos que $m(H,t)$ sea el número de instancias de H en la generación t, y que $\hat{u}(H,t)$ sea la aptitud promedio observada de H en la generación t (osea, la aptitud promedio de las instancias de H en la población en la generación t).

Lo que queremos calcular es $E(m(H,t+1))$, o sea el número esperado de instancias de H en la generación $t+1$.

12.2.1 Efecto de la Selección

Supongamos que usamos selección proporcional. Bajo este esquema, el número esperado de descendientes de una cadena x es igual a $f(x)/\bar{f}(t)$, donde $f(x)$ es la aptitud de x y $\bar{f}(t)$ es la aptitud promedio de la población en la generación t.

Ignorando por ahora los efectos de la cruza y la mutación, y suponiendo que x está en la población en la generación t, tenemos:

$$E(m,(H,t+1)) = \sum_{x\in H} f(x)/\bar{f}(t)$$

donde: $x \in H$ denota "x es una instancia de H".

Sin embargo, por definición:

$$\hat{u}(H,t) = \left(\sum_{x\in H} f(x)\right)/m(H,t)$$

Por lo que:

$$E(m,(H,t+1)) = (\hat{u}(H,t)/\bar{f}(t))m(H,t)$$

De tal forma, aunque el AG no calcule explícitamente $\hat{u}(H,t)$, los incrementos o decrementos de las instancias de esquemas en la población dependen de esta cantidad.

12.2.2 Efecto de la Cruza

Haremos un análisis del efecto destructivo (o sea, el que decrementa el número de instancias de H) de la cruza de un punto.

Hagamos que Pc sea el porcentaje de cruza y supongamos que una instancia del esquema H se selecciona para ser padre. Se dice que el esquema H "sobrevive" bajo la cruza de un punto si uno de sus hijos es también una instancia del esquema H.

Podemos proporcionar un límite inferior de la probabilidad $S_c(H)$ de que H sobrevivirá la cruza de un punto:

$$S_c(H) \geq 1 - P_c\left[\frac{\delta(H)}{l-1}\right]$$

Esto es, las cruzas que ocurren dentro de la longitud de definición de H pueden destruir a H (osea, pueden producir hijos que no son instancias de H).

De tal forma, multiplicamos la fracción de la cadena que H ocupa por la probabilidad de cruza para obtener un límite superior de la probabilidad de que el esquema será destruído.

El valor es un límite superior porque algunas cruzas dentro de las posiciones definidas de un esquema no lo destruirán (p. ej. si dos cadenas idénticas se cruzan). Al sustraer este valor de 1 obtenemos un límite inferior.

NOTA: La probabilidad de supervivencia bajo cruza es alta, para esquemas con baja longitud de definición.

12.2.3 Efecto de la Mutación

Hagamos que Pm sea la probabilidad de mutación y que $S_m(H)$ sea la probabilidad de que el esquema H sobrevivirá bajo la mutación de una instancia de H:

$$S_m(H) = (1 - P_m)^{o(H)}$$

Esto es, para cada bit, la probabilidad de que éste no se mutará es $1 - P_m$. De tal manera, la probabilidad de que bits no definidos del esquema H se muten es esta cantidad multiplicada por sí misma $o(H)$ veces.

NOTA: La probabilidad de supervivencia bajo mutación es más alta para esquemas de bajo orden.

Combinando estos 3 efectos (selección, cruza y mutación), tenemos:

$$E(m, (H, t+1)) \geq \frac{\hat{u}(H,t)}{\bar{f}(t)} m(H,t) \left[1 - P_c \left(\frac{\delta(H)}{l-1}\right)\right] (1 - P_m)^{o(H)}$$

A esta expresión se le conoce como el **Teorema de los Esquemas** [127], y describe el crecimiento de un esquema de una generación a la siguiente. El Teorema de los Esquemas frecuentemente se interpreta de tal forma que implica que los esquemas cortos y de bajo orden cuya aptitud promedio se mantiene por encima de la media, recibirán un número de muestras que crece exponencialmente con el tiempo.

La razón por la que se dice que los esquemas cortos y de bajo orden reciben un número de muestras que se incrementa exponencialmente con el tiempo es porque el número de muestras de esos esquemas que no son perturbados y permanecen

sobre la aptitud promedio se incrementan en un factor de $\hat{u}(H,t)/\bar{f}(t)$ a cada generación.

El Teorema de los Esquemas es un límite inferior porque sólo lidia con los efectos destructivos de la cruza y la mutación. Sin embargo, se cree que la cruza es la fuente de mayor poder del AG, pues tiene la capacidad de recombinar las instancias de esquemas favorables para formar otros igualmente buenos o mejores. Esta suposición de que los AGs trabajan de esta manera se conoce como la **Hipótesis de los Bloques Constructores** [105].

El efecto de la selección es sesgar gradualmente el procedimiento de muestreo hacia instancias de esquemas cuya aptitud se estime estén sobre el promedio. Con el paso del tiempo, el estimado de la aptitud promedio de un esquema debiera, en principio, volverse cada vez más preciso puesto que el AG está muestreando más y más instancias de este esquema.

El Teorema de los Esquemas y la Hipótesis de los Bloques Constructores lidian primordialmente con el papel de la selección y la cruza en los AGs, pero ¿cuál es el papel de la mutación?

Holland [127] propuso que la mutación previene la pérdida de diversidad en una posición cualquiera. Es una especie de "póliza de seguro" contra fijaciones en una cadena cromosómica.

12.3 Críticas al Teorema de los Esquemas

Existe una enorme cantidad de críticas en torno a las interpretaciones que se le han dado al teorema de los esquemas. Estas críticas provienen de parte de los matemáticos y físicos que han modelado matemáticamente el comportamiento de un algoritmo genético (ver por ejemplo [222]). La primera crítica es el torno al hecho de que el "Teorema de los Esquemas" es realmente una desigualdad "débil", no un "teorema".

Adicionalmente, las siguientes afirmaciones sobre el teorema de los esquemas no son del todo demostrables:

- Los esquemas por arriba del promedio se incrementan exponencialmente con el tiempo.

- Los esquemas por arriba del promedio se exploran rápidamente en paralelo sin alentar de manera significativa la búsqueda.

- Aproximadamente se procesan N^3 esquemas de manera útil y en paralelo por cada generación.

12.4 No Free Lunch Theorem

Este famoso teorema fue formulado en un artículo (liberado originalmente como reporte técnico) escrito por David Wolpert y William MacReady, del Instituto Santa Fe, en Nuevo México, en 1995 [229, 230].

La principal implicación del *No Free Lunch Theorem* es que todas las técnicas de búsqueda heurística son matemáticamente equivalentes en general. Es decir, no hay una sola técnica que supere a las demás en todos los casos.

La moraleja es pues que el énfasis que suele ponerse en optimizar con técnicas heurísticas (como el AG) es erróneo.

¿Qué alternativa tenemos entonces?

Investigar el comportamiento emergente de una técnica heurística.

¿Cuál es el costo de esta alternativa?

Formalizar nuestro modelo heurístico y realizar demostraciones a partir de dicha formalización.

¿Qué ganamos?

Una comprensión conceptual de la técnica y una descripción a fondo de las circunstancias en las cuales un AG es la mejor alternativa.

12.5 Decepción

Hemos hablado anteriormente sobre un problema que ha preocupado a los teóricos de la computación evolutiva: la **decepción**.

Se llama **decepción** a la condición donde la combinación de buenos bloques constructores llevan a una reducción de aptitud, en vez de un incremento. Este fenómeno fue sugerido originalmente por Goldberg [105] para explicar el mal

desempeño del AG en algunos problemas. Veamos un ejemplo de un problema deceptivo:

Supongamos que tenemos una función de aptitud que nos devuelve los siguientes valores para las cadenas binarias de longitud 3:

Cadena	Aptitud
000	70
001	50
010	49
011	1
100	30
101	2
110	3
111	80

Las cadenas con mayor número de ceros tienen mayor aptitud, pero el óptimo global es la cadena de todos unos. En este caso, el AG tenderá a favorecer durante la selección a las cadenas con más ceros y encontrará la cadena de todos ceros (un óptimo local) en vez de llegar al óptimo global.

12.6 Areas abiertas de investigación

Algunas de las preguntas más importantes que se han planteado dentro de la comunidad de los algoritmos genéticos son las siguientes [167, 10]:

- ¿Qué leyes describen el comportamiento macroscópico de los AGs? En particular, ¿qué predicciones pueden hacerse acerca del cambio de aptitud en el tiempo y acerca de la dinámica de la población en un AG en particular.

- ¿Cómo dan lugar los operadores de bajo nivel (selección, cruza y mutación) al comportamiento macroscópico de los AGs?

- ¿En qué tipo de problemas es más probable que los AGs tengan un buen desempeño?

- ¿En qué tipo de problemas es más probable que los AGs tengan un mal desempeño?

- ¿Qué significa para un AG tener un "buen desempeño" o un "mal desempeño"? Esto es, ¿qué criterios de desempeño son apropiados para un AG?

- ¿Bajo qué condiciones (tipos de AGs y tipos de problemas) superará un AG a otras técnicas de búsqueda tales como escalando la colina y otros métodos de gradiente?

12.7 ¿Cuándo debe utilizarse un AG?

El AG suele considerarse una técnica que es buena para encontrar rápida-mente regiones prometedoras del espacio de búsqueda, pero para realizar verdaderamente optimización se ha demostrado que en muchas instancias los híbridos de un AG con otra técnica (por ejemplo, escalando la colina) parecen dar mejores resultados. Aunque los AGs pueden encontrar los óptimos globales de problemas de alta complejidad, la realidad es que muchas veces el costo computacional que requieren es prohibitivamente alto, y se prefieren para encontrar una solución razonable, ya que eso suelen poder hacerlo en un tiempo relativamente corto.

Como heurística, el AG no resulta muy adecuado para problemas críticos en los cuales el no encontrar una solución en un período de tiempo muy corto puede causar fallas irreversibles al sistema. Asimismo, no es apropiado para aplicaciones en tiempo real en las que la respuesta debe proporcionarse de manera inmediata conforme se interactúa con el ambiente.

Algunos autores (por ejemplo Kenneth De Jong [138]) han argumentado elocuentemente que, contrario a lo que se cree, los AGs no son optimizadores, sino que más bien son "satisfactores de metas" (o decisiones) secuenciales que pueden modificarse para actuar como optimizadores de funciones. Si bien en la práctica han tenido un gran éxito como optimizadores, la realidad es que los AGs suelen tener dificultades para encontrar óptimos globales en ciertas clases de problemas (como por ejemplo el del viajero) sumamente susceptibles a la representación o aquellos en los que la evaluación de la función de aptitud resulta sumamente costoso.

12.8 ¿Qué es lo que hace difícil un problema para un AG?

La teoría de los esquemas se ha usado por algunos en la comunidad de los AGs para proponer una respuesta a la pregunta: "¿qué hace que un problema sea difícil

para un AG?"

La panorámica actual es que la competencia entre los esquemas procede aproximadamente de particiones de esquemas de bajo orden a particiones de esquemas de orden alto. Bethke [22] infirió que sería difícil para un AG encontrar el óptimo de una función de aptitud si las particiones de bajo orden contenían información errónea acerca de las particiones de orden más alto.

Consideremos un ejemplo, un tanto extremo, de lo dicho en el punto anterior. Llamemos al esquema H "ganador" si su aptitud estática promedio es la más alta en su partición. Supongamos que cualquier esquema cuyos bits definidos sean todos unos es un ganador, excepto por el esquema de longitud L $1111...1$, y hagamos que $0000...0$ sea un ganador.

En principio, será difícil para un AG encontrar $0000...0$, porque toda partición de bajo orden da información equivocada sobre dónde es más probable encontrar el óptimo. A este tipo de función de aptitud se le llama "**con decepción total**".

Afortunadamente, nadie ha podido encontrar un problema del mundo real que exhiba **decepción total**. Sólo se han encontrado problemas reales con **decepción parcial**.

También es posible definir funciones de aptitud con menores cantidades de decepción (es decir, algunas particiones dan información correcta acerca de la localización del óptimo). Bethke [22] usó la "Transformada de Walsh" (similar a la transformada de Fourier) para diseñar funciones de aptitud con varios grados de aptitud. Posteriormente, un gran número de investigadores han profundizado en estos estudios, y hoy en día la decepción es uno de los problemas centrales de los teóricos en AGs. Debido a que los AGs se han usado extensamente para la optimización de funciones, es importante que los que desarrollan aplicaciones con ellos sepan al menos sobre este fenómeno para poder prevenirlo y atacarlo en caso de que se presente.

Varios investigadores han cuestionado la relevancia del análisis de los esquemas en la comprensión del funcionamiento verdadero de los AGs [117, 154, 177].

Por ejemplo, Grefenstette [117] afirma que mucho del trabajo efectuado en teoría de los AGs ha asumido un versión más fuerte de lo que él llama la "**Hipótesis Estática de los Bloques Constructores**" (HEBC):

> Dada cualquier partición de esquemas de bajo orden y reducida longitud de definición, se espera que un AG converja al hiperplano (en esa partición) con la mejor aptitud promedio estática (el 'ganador esperado').

Esta formulación es más fuerte que la original, puesto que dice que el AG convergerá a los "**verdaderos**" ganadores de cada competencia entre particiones cortas y de bajo orden en vez de que converja a esquemas con la mejor aptitud **observada**.

La HEBC no es lo que propuso Holland [127], y nunca se ha demostrado o validado empíricamente, pero implícitamente involucra la premisa de que las funciones de aptitud con decepción serán difíciles para un AG.

Grefenstette proporciona 2 razones posibles por las que la HEBC puede fallar:

1. **Convergencia Colateral**: Una vez que la población comienza a converger hacia algún punto, las muestras de algunos esquemas dejarán de ser uniformes.

   Por ejemplo, supongamos que las instancias de $111 * ... *$ son muy aptas y que la población ha convergido más o menos a esos bits (o sea, casi todos los miembros de la población son una instancia de ese esquema). Entonces casi todas las muestras de, por ejemplo, $***000 * ... *$, serán realmente muestras de $111000 * ... *$. Esto puede impedir que el AG haga una estimación precisa de $\hat{u}(***000 * ... *)$. Aquí la $\hat{u}(H)$ denota la aptitud promedio estática de un esquema H (el promedio sobre todas las instancias del esquema en el espacio de búsqueda).

2. **Elevada Varianza de la Aptitud**: Si la aptitud promedio estática de un esquema tiene una varianza alta, el AG puede no ser capaz de efectuar una estimación precisa de esta aptitud promedio estática.

   Consideremos, por ejemplo, la siguiente función de aptitud:

$$f(x) = \begin{cases} 2 & \text{si } x \in 111 * ... * \\ 1 & \text{si } x \in 0 * ... * \\ 0 & \text{de lo contrario} \end{cases}$$

   La varianza de $1 * ... *$ es alta, así que el AG converge a las subregiones de aptitud elevada de este esquema. Esto sesga a todas las muestras subsecuentes de este esquema, impidiendo que se puede obtener un estimado preciso de su aptitud estática.

Esto tiene relación directa con la importancia de la decepción en el comportamiento de los AGs, porque las funciones de aptitud con decepción se definen completamente en términos de las aptitudes estáticas promedio de los esquemas.

Para ilustrar su argumento, Grefenstette [117] da ejemplos de problemas con decepción que son fáciles de optimizar con un AG y de problemas que no tienen decepción y que son arbitrariamente difíciles para un AG. Su conclusión es de que la decepción no es una causa necesaria ni suficiente para que un AG tenga dificultades, y de que su relevancia en el estudio de los AGs debe ser demostrada todavía.

12.9 Las Funciones de la Carretera Real

El Teorema de los Esquemas, por sí mismo, no hace referencia a los efectos positivos de la selección (asignar muestras cada vez mayores de los esquemas que han mostrado tener un mejor desempeño), sino únicamente a los aspectos de perturbación de la cruza. Tampoco aborda la pregunta de cómo hace la cruza para recombinar los esquemas más aptos, aunque esta parece ser la mayor fuente de poder del AG.

La hipótesis de los bloques constructores dice que la cruza combina esquemas cortos y de alto desempeño demostrado para formar candidatos más aptos, pero no da una descripción detallada de cómo ocurre esta combinación.

Para investigar el procesamiento de esquemas y la recombinación en más detalle, Stephanie Forrest, John Holland y Melanie Mitchell [169, 168] diseñaron ciertos paisajes de aptitud llamados de la **Carretera Real** (*Royal Road*), que intentaron capturar la esencia de los bloques constructores en una forma idealizada.

La Hipótesis de los Bloques Constructores sugiere 2 características de los paisajes de aptitud que son particularmente relevantes a los AGs:

1. La presencia de esquemas cortos, de bajo orden y altamente aptos, y

2. La presencia de "escalones" intermedios (esquemas de orden intermedio de aptitud más alta que resultan de combinaciones de los esquemas de orden menor y que, en turno, pueden combinarse para crear esquemas de aptitud aún mayor).

Las funciones que diseñaron Mitchell et al. [169, 168] se esperaba que (siguiendo la hipótesis de los bloques constructores) tenderían una "carretera real" al AG de manera que pudiera llegar fácilmente a la cadena óptima. Asimismo, la hipótesis era que las técnicas escalando la colina tendrían dificultades con estas funciones, porque se necesitaba optimizar un gran número de posiciones de la cadena simultáneamente para moverse de una instancia de bajo orden a una de

orden intermedio o alto. Sin embargo, algunas cosas extrañas sucedieron. Uno de los **hillclimbers** que usaron (habían de 3 tipos) superó por mucho al algoritmo genético (encontró la solución 10 veces más rápido que el AG).

Tras efectuar un análisis de la función en la que el AG tuvo problemas, se determinó que una de las causas fueron los **hitchhikers**, que limita seriamente el paralelismo implícito del AG restringiendo los esquemas muestreados en ciertos lugares. Estos genes parásitos limitan el efecto de la cruza para recombinar bloques constructores, y hacen que converjan hacia esquemas equivocados en diversas particiones. Sin embargo, este fenómeno no debe resultar demasiado sorprendente si se considera que se ha observado en la genética real. Para lidiar con este problema se propuso un **Algoritmo Genético Idealizado**, y se concluyeron varias cosas importantes en torno a cómo debe diseñarse un algoritmo genético convencional para evitar el **hitchhiking**:

1. La población tiene que ser suficientemente grande, el proceso de selección debe ser suficientemente lento, y el porcentaje de mutación debe ser suficientemente alto para asegurar que no haya ninguna posición que permanezca fija con un solo valor en ninguna cadena.

2. La selección tiene que ser suficientemente fuerte como para preservar los esquemas deseados que se han descubierto, pero también tiene que ser suficientemente lenta (o, de manera equivalente, la aptitud relativa de los esquemas deseables no traslapados tiene que ser suficientemente pequeña) para prevenir que ocurra algún **hitchhiking** significativo en algunos esquemas altamente aptos que pueda eliminar esquemas deseables de otras partes de la cadena.

3. El porcentaje de cruza tiene que ser tal que el tiempo en que ocurra una cruza que combine dos esquemas deseados sea pequeño con respecto al tiempo de descubrimiento de los esquemas deseados.

Estos mecanismos no son compatibles entre sí. Por ejemplo, un alto porcentaje de mutación está en contraposición con una selección fuerte. Por lo tanto, debe cuidarse de que haya algún equilibrio de estos puntos a la hora de aplicarlos.

El teorema de los esquemas hace predicciones en torno al cambio esperado en las frecuencias de los esquemas de una generación a la siguiente, pero no hace predicciones directamente sobre la composición de la población, la velocidad de convergencia de la población o la distribución de aptitudes de la población con respecto al tiempo. Como un primer paso para tener una mejor comprensión del

comportamiento de los AGs, y para poder hacer mejores predicciones, varios investigadores han construido modelos matemáticos "exactos" de AGs simples [104, 221, 114, 139, 222, 171]. Estos modelos exactos capturan todos los detalles de un AG simple usando operadores matemáticos.

12.10 ¿Cuándo debe usarse un AG?

Es adecuado si el espacio de búsqueda es grande, accidentado, poco comprendido, o si la función de aptitud tiene mucho ruido, y si la tarea no requiere que se encuentre el óptimo global (encontrar una solución bastante buena con cierta rapidez resulta suficiente).

Si el espacio de búsqueda es muy pequeño, entonces el problema se puede resolver mediante búsqueda exhaustiva, y el AG no tiene mucha razón de ser.

Si el espacio de búsqueda no está accidentado y es unimodal, entonces una técnica de gradiente como "escalando la colina con ascenso pronunciado" será mucho más eficiente que un algoritmo genético.

Si el espacio de búsqueda se conoce bien (p. ej. alguna instancia pequeña del problema del viajero) es posible diseñar métodos de búsqueda que usen conocimiento especfico sobre el dominio para superar fácilmente a una técnica independiente del dominio como el AG. Si la función de aptitud tiene ruido (por ejemplo, si involucra tomar medidas sujetas a error de un proceso del mundo real tal como la visión de un robot), un método de búsqueda que use un solo candidato a la vez (como escalando la colina) será arrastrada inevitablemente por rutas erróneas debido al ruido, mientras que el AG tendrá un desempeño razonable porque trabaja mediante la acumulación de estadsticas de aptitud a través de las generaciones.

Los consejos anteriores deben tomarse con ciertas precauciones, porque no hay reglas universales sobre cuándo utilizar un AG para resolver un problema y cuándo no hacerlo. Su desempeño normalmente dependerá de detalles tales como el método de codificación de las soluciones candidatas, los operadores, los valores de los parámetros, y el criterio en particular para medir el éxito del algoritmo.

12.11 Diseño de Funciones Deceptivas

Problema deceptivo mínimo

La idea básica tras el diseño de funciones deceptivas para un algoritmo genético es violar de manera extrema la hipótesis de los bloques constructores. En otras palabras, buscaremos ahora que los bloques cortos y de bajo orden nos conduzcan a bloques constructores largos y de mayor orden que sean incorrectos (subóptimos).

El problema deceptivo más pequeño posible es de dos bits. Su descripción se presenta a continuación.

Supongamos que tenemos un conjunto de cuatro esquemas de orden 2 como se indica a continuación:

esquema	aptitud
0**0*	f_{00}
0**1*	f_{01}
1**0*	f_{10}
1**1*	f_{11}

f_{11} es la aptitud máxima posible (óptimo global).

$f_{11} > f_{00}; f_{11} > f_{01}; f_{11} > f_{10}$

Ahora procederemos a introducir el elemento deceptivo usando la idea siguiente: buscaremos que en nuestro problema uno de los esquemas subóptimos de orden 1 (o los dos) sea mejor que los esquemas de orden 1 del óptimo.

Para poner el problema en una perspectiva más adecuada, vamos a normalizar todas las aptitudes con respecto al complemento del óptimo global:

$$r = \frac{f_{11}}{f_{00}}; c = \frac{f_{01}}{f_{00}}; c' = \frac{f_{10}}{f_{00}}$$

Podemos re-escribir ahora la condición de globalidad en forma normalizada:

$$\frac{f_{11}}{f_{00}} > \frac{f_{00}}{f_{00}}; \frac{f_{11}}{f_{00}} > \frac{f_{01}}{f_{00}}; \frac{f_{11}}{f_{00}} > \frac{f_{10}}{f_{00}}$$

Re-escribamos ahora la condición deceptiva:

$$\frac{f_{00} + f_{01}}{2} > \frac{f_{10} + f_{11}}{2}$$

$$\frac{f_{00} + f_{01}}{f_{00}} > \frac{f_{10} + f_{11}}{f_{00}}$$

$$1 + \frac{f_{01}}{f_{00}} > \frac{f_{10}}{f_{00}} + \frac{f_{11}}{f_{00}}$$

$$1 + c > c' + r$$

$$r + c' < c + 1$$

$$r < c + 1 - c'$$

$$f(0*) > f(1*)$$

$$f(*0) > f(*1)$$

En estas expresiones estamos ignorando todos los demás alelos de las cadenas cromosómicas que no sean las 2 posiciones definidas antes indicadas y las expresiones anteriores implican un promedio sobre todas las cadenas contenidas en el subconjunto de similitud. De tal forma que deben cumplirse las siguientes expresiones:

$$\frac{f(00) + f(01)}{2} > \frac{f(10) + f(11)}{2}$$
$$\frac{f(00) + f(10)}{2} > \frac{f(01) + f(11)}{2}$$

Si embargo, estas 2 expresiones no pueden cumplirse simultáneamente, pues de hacerlo f_{11} no sería el óptimo global.

Sin pérdida de generalidad, supondremos que la primera expresión es cierta:

$$f(0*) > f(1*)$$

A partir de lo anterior, podemos concluir que:

$$c' < 1; c' < c$$

Tenemos entonces 2 clases de problemas deceptivos:

TIPO I: $f_{01} > f_{00}$ ($c > 1$)

TIPO II: $f_{00} \geq f_{01}$ ($c \leq 1$)

Figura 12.2: Representación gráfica de la clase de problemas deceptivos de Tipo I.

Gráficamente, podemos representar estos dos problemas en las figuras 12.2 y 12.3, respectivamente.

Estos 2 tipos de problemas son deceptivos y puede demostrarse que ninguno de ellos puede expresarse como una combinación lineal de los valores alélicos del individuo.

Ninguno de estos casos puede expresarse como:

$$f(x_1, x_2) = b + \sum_{i=1}^{2} a_i x_i$$

En términos biológicos, tenemos un problema epistático. Puesto que puede demostrarse que ningún problema de 1 bit puede ser deceptivo, el problema de 2 bits antes indicado es el problema deceptivo mínimo.

12.12 Estudios de Convergencia

Los Algoritmos Genéticos son usados a menudo para resolver problemas de optimización del tipo: $max\{f(b)|b \in \mathbb{B}^l\}$ asumiendo que $0 < f(b) < \infty$ para todo $b \in \mathbb{B}^l = \{0,1\}^l$ y $f(b) \neq const$.

Figura 12.3: Representación gráfica de la clase de problemas deceptivos de Tipo II.

12.12.1 Algoritmo Genético Sin Elitismo

En [195] Rudolph modela el Algoritmo Genético Simple (AGS) mediante una cadena de Markov finita homogénea.

Cada estado i de la cadena de Markov corresponde con una posible población del AGS de tal manera que el espacio de estados es $\mathcal{S} = \mathbb{B}^{nl}$ donde n es el número de individuos de la población y l es la longitud de cada individuo. Definimos a $\pi_k^t(i)$ como el individuo k de la población i en el paso t.

Dada la naturaleza del AGS, la matriz de transición $\mathbf{P}$ que lo representa queda definida como:

$$\mathbf{P} = \mathbf{CMS}$$

donde $\mathbf{C}$, $\mathbf{M}$ y $\mathbf{S}$ son las matrices de transición de los operadores de Cruza, Mutación y Selección respectivamente.

Cuando se usa mutación uniforme se tiene que:

$$m_{ij} = p_m^{H_{ij}}(1 - p_m)^{N - H_{ij}} > 0$$

en donde p_m es la probabilidad de mutación del AGS, H_{ij} es la distancia de Hamming entre los estados i y j, y $N = nl$. De lo anterior concluímos que $\mathbf{M}$ es positiva.

Por otra parte, dado que el operador de selección lo que hace es proporcionarnos pares de individuos con fines de que ya sea que pasen intactos a la población siguiente o que con una cierta probabilidad mediante el operador de cruza puedan dar lugar a otros dos individuos nuevos, la matriz de transición de este operador lo que hace es dejar "ordenados" a los individuos tal y como se irán tomando para dar lugar a la población siguiente.

El uso de un operador de selección proporcional o de torneo [196] determina la existencia de una probabilidad estrictamente positiva de que la población quede intacta, lo cual asegura que los elementos de la diagonal s_{ii} de la matriz de transición del operador son positivos, por lo que se concluye que la matriz $\mathbf{S}$ es columna-permisible.

Lema 12.12.1 *Sean* $\mathbf{C}$*,* $\mathbf{M}$ *y* $\mathbf{S}$ *matrices estocásticas, donde* $\mathbf{M}$ *es positiva y* $\mathbf{S}$ *es columna-permisible. Entonces la matriz producto* $\mathbf{CMS}$ *es positiva.*

En resumen tenemos que, dado que la matriz $\mathbf{M}$ es positiva y la $\mathbf{S}$ es columna-permisible, por el lema 12.12.1 la matriz $\mathbf{P} = \mathbf{CMS}$ es positiva y por lo tanto primitiva.

Para poder hablar de convergencia a continuación se muestra la definición correspondiente para el AGS [195]:

Definición 12.12.1 *Sea* $Z_t = max\{f(\pi_k^t(i))|k = 1, ..., n\}$ *una sucesión de variables aleatorias representando la mejor aptitud dentro de la población representada por el estado* i *en el paso* t*. Un algoritmo genético converge al óptimo global si y sólo si:*

$$lim_{t\to\infty} P\{Z_t = f^*\} = 1 \tag{12.1}$$

donde $f^* = max\{f(b)|b \in \mathbb{B}^l\}$.

De esta manera entenderemos que el AGS converge al óptimo global de función objetivo si la probabilidad de que éste se encuentre en la población tiende a 1 cuando el número de iteraciones tiende a infinito.

Teorema 12.12.1 *Sea P una matriz estocástica primitiva. Entonces* P^k *converge cuando* $k \to \infty$ *a una matriz estocástica positiva estable* $P^\infty = \mathbf{1}'p^\infty$*, donde* $p^\infty = p^0 \cdot lim_{k\to\infty} P^k = p^0 P^\infty$ *tiene entradas diferentes de cero y es única independientemente de la distribución inicial.*

Así pues, dada la definición anterior y usando el teorema 12.12.1 Rudolph demuestra que el AGS no converge:

Teorema 12.12.2 *El AGS con matriz de transición primitiva no converge al óptimo global.*

Demostración Sea $i \in \mathcal{S}$ cualquier estado en el que $max\{f(\pi_k^t(i))|k = 1, ..., n\} < f^*$ y p_i^t la probabilidad de que el AGS esté en tal estado i en el paso t. Claramente, $P\{Z_t \neq f^*\} \geq p_i^t \Leftrightarrow P\{Z_t = f^*\} \leq 1 - p_i^t$. Por el teorema 12.12.1 la probabilidad de que el AGS esté en el estado i converge a $p_i^\infty > 0$. Por lo tanto:

$$lim_{t\to\infty} P\{Z_t = f^*\} \leq 1 - p_i^\infty < 1$$

por lo tanto la condición **(12.1)** no se satisface.

El Teorema 12.12.2 muestra que dado que según el teorema 12.12.1 la matriz de transición $\mathbf{P}$ del AGS converge a una matriz positiva, la probabilidad de estar en un estado no-óptimo es estrictamente positiva conforme el número de iteraciones se incrementa por lo que la probabilidad de permanecer en un estado óptimo no es 1 en el límite.

12.12.2 Algoritmo Genético Elitista

En [195] Rudolph argumenta que en las aplicaciones del mundo real el AGS comúnmente mantiene a través del proceso evolutivo la mejor solución encontrada hasta el momento por lo que lo correcto es modelar el AGS de tal manera.

Así pues, consideraremos ahora agregar a la población del AGS un *súper individuo* que no tomará parte en el proceso evolutivo y que por facilidad en la notación será colocado en la primera posición a la izquierda, es decir, se podrá accesar a él mediante $\pi_0(i)$. Llamaremos a esta nueva versión Algoritmo Genético Elitista (AGE).

La cardinalidad del espacio de estados de la cadena de Markov correspondiente crece ahora de 2^{nl} a $2^{(n+1)l}$ debido a que tenemos 2^l posibles *súper individuos* y por cada uno de ellos tenemos 2^{nl} poblaciones posibles.

El operador de elitismo estará representado por la matriz $\mathbf{E}$ que lo que hará será actualizar un estado de tal manera que si éste contiene un individuo mejor que su actual *súper individuo* éste será reemplazado por aquél.

En particular, sea:

$$i = (\pi_0(i), \pi_1(i), \pi_2(i), ..., \pi_n(i)) \in \mathcal{S}$$

$\pi_0(i)$ es el súper individuo de la población (estado) i, ahora bien, sean $b = arg(max\{f(\pi_k(i))|k = 1, ..., n\}) \in \mathbb{B}^l$ el mejor individuo de la población i excluyendo el *súper individuo* y:

$$j \stackrel{\text{def}}{=} (b, \pi_1(i), \pi_2(i), ..., \pi_n(i)) \in \mathcal{S}$$

entonces:

$$e_{ij} = \begin{cases} 1 & \text{si } f(\pi_0(i)) < f(b) \\ 0 & \text{de otra manera.} \end{cases}$$

Nuestra nueva matriz de transición para el AGE resulta del producto de una matriz que está compuesta por 2^l matrices $\mathbf{P}$, una por cada posible *súper individuo* acomodadas de manera que entre mejor sea su súper individuo más alta será su posición, y la matriz $\mathbf{E}$ de operador de elitismo:

$$\mathbf{P}^+ = \begin{pmatrix} \mathbf{P} & & & \\ & \mathbf{P} & & \\ & & \ddots & \\ & & & \mathbf{P} \end{pmatrix} \begin{pmatrix} \mathbf{E_{11}} & & & \\ \mathbf{E_{21}} & \mathbf{E_{22}} & & \\ \vdots & \vdots & \ddots & \\ \mathbf{E_{2^l,1}} & \mathbf{E_{2^l,2}} & \cdots & \mathbf{E_{2^l,2^l}} \end{pmatrix}$$

$$= \begin{pmatrix} \mathbf{PE_{11}} & & & \\ \mathbf{PE_{21}} & \mathbf{PE_{22}} & & \\ \vdots & \vdots & \ddots & \\ \mathbf{PE_{2^l,1}} & \mathbf{PE_{2^l,2}} & \cdots & \mathbf{PE_{2^l,2^l}} \end{pmatrix}$$

La estructura mostrada de la matriz $\mathbf{P}^+$ se debe a que, como ya se mencionó, las poblaciones están ordenadas de manera descendente de acuerdo a la calidad de su súper individuo, de tal manera los espacios en blanco representan ceros puesto que no es posible pasar de un estado a otro con un súper individuo de menor calidad.

De lo anterior se concluye que $\mathbf{PE_{11}} = \mathbf{P}$ puesto que tales matrices corresponden con las poblaciones que tienen como súper individuo al óptimo f^*.

Por otra parte, haciendo las siguientes definiciones:

$$\mathbf{R} = \begin{pmatrix} \mathbf{PE_{21}} \\ \vdots \\ \mathbf{PE_{2^l,1}} \end{pmatrix} \mathbf{T} = \begin{pmatrix} \mathbf{PE_{22}} & & \\ \vdots & \ddots & \\ \mathbf{PE_{2^l,2}} & \cdots & \mathbf{PE_{2^l,2^l}} \end{pmatrix}$$

concluimos que la matriz $\mathbf{P}^+$ es reducible:

$$\mathbf{P}^{+} = \begin{pmatrix} \mathbf{P} & \mathbf{0} \\ \mathbf{R} & \mathbf{T} \end{pmatrix}$$

Teorema 12.12.3 *Sea* $\mathbf{P}$ *una matriz estocástica reducible, donde* $\mathbf{C} : m \times m$ *es una matriz estocástica primitiva y* $\mathbf{R}$, $\mathbf{T} \neq 0$. *Entonces:*

$$\mathbf{P}^{\infty} = lim_{k\to\infty}\mathbf{P}^{k} = \begin{pmatrix} \mathbf{C}^{k} & 0 \\ \sum_{i=0}^{k-1} \mathbf{T}^{i}\mathbf{R}\mathbf{C}^{k-i} & \mathbf{T}^{k} \end{pmatrix} = \begin{pmatrix} \mathbf{C}^{\infty} & 0 \\ \mathbf{R}_{\infty} & 0 \end{pmatrix}$$

es una matriz estocástica estable con $\mathbf{P}^{\infty} = \mathbf{1}'p^{\infty}$, *donde* $p^{\infty} = p^{0}\mathbf{P}^{\infty}$ *es única independientemente de la distribución inicia, y* p^{∞} *satisface:* $p_i^{\infty} > 0$ *para* $1 \leq i \geq m$ *y* $p_i^{\infty} = 0$ *para* $m < i \geq n$.

Como conclusión tenemos el siguiente Teorema:

Teorema 12.12.4 *El AGE converge al óptimo global.*

Demostración La submatriz $\mathbf{P}$ contiene las probabilidades de transición de estados óptimos globales. Puesto que $\mathbf{P}$ es una matriz estocástica primitiva y $\mathbf{R}$, $\mathbf{T} \neq 0$, el teorema 12.12.3 garantiza que la probabilidad de permanecer en un estado no-óptimo converge a cero. Por lo tanto la probabilidad de permanecer en un estado óptimo global converge a 1. Así pues, se ha demostrado la convergencia del AGE, es decir, de un Algoritmo Genético que usa elitismo.

12.13 Problemas Propuestos

1. Suponga que tenemos los siguientes esquemas: (a) $H_1 = 1 * * * * * **$, (b) $H_2 = 0 * * * *1 * 0$, (c) $H_3 = * * * * * * 01$, (d) $H_4 = *0 * 1 * 00*$, (e) $H_5 = 1 * * * * * 1*$ y (f) $H_6 = 1010 * 1 * *$.

   (a) Estime la probabilidad de supervivencia de cada esquema bajo mutación, suponiendo que la probabilidad de mutación es $p_m = 0.003$.

   (b) Estime la probabilidad de supervivencia de cada esquema bajo cruza, suponiendo que la probabilidad de cruza es $p_c = 0.65$.

Reporte sus resultados con al menos tres dígitos de precisión.

2. Basándose en el análisis que hicimos de la probabilidad de que un esquema sobreviva a la cruza de un punto, derive una expresión similar para la cruza de dos puntos.

3. Basándose en el análisis que hicimos de la probabilidad de que un esquema sobreviva a la cruza de un punto, derive una expresión similar para la cruza uniforme.

Capítulo 13

Operadores Avanzados

Además de los operadores tradicionales de cruza y mutación que hemos estudiado previamente, existen otros, más específicos, que aunque no suelen usarse con mucha frecuencia en la práctica, es importante conocer. Este capítulo se dedicará al estudio de ellos.

13.1 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.

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

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).

Figura 13.1: Ejemplo del uso de cromosomas diploides.

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

Suena lógico cuestionarse ¿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.

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.

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).

El ejemplo de diploides mostrado en la figura 4.6 se debe a Hillis [123, 122]. 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 la figura 4.6).

13.2 Inversión

Holland [127] 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.

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.

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.

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.

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 resulado sería:

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

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.

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 cruza de un punto lo preservará y esta permutación tenderá a diseminarse con el paso de las generaciones.

Debe advertirse que el uso de este operador introduce nuevos problemas cuando se combina con la cruza 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

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

Si el punto de cruza 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

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

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?

Holland propuso 2 soluciones posibles:

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

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.

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.

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.

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 cruza) que se requiere.

13.3 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).

Pero, ¿para qué tomarnos estas molestias?

Holland [127] 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.

13.3.1 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 cruza se efectúa igual que como vimos antes, pero cuando formamos un gameto, tenemos que seleccionar aleatoriamente uno de los cromosomas haploides.

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.

13.3.2 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.

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

La segregación y la traslocación no se han usado mucho en la práctica, excepto por algunas aplicaciones de aprendizaje de máquina [198, 208].

13.3.3 Duplicación y Borrado

Estos 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.

Holland [127] 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.

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.

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.

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).

13.4 Problemas Propuestos

1. Investigue la forma en la que se implementaría un cromosoma triploide y discuta una posible aplicación del mismo. El libro de Goldberg podría serle de ayuda:

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

2. Implemente los micro-operadores discutidos en este capítulo y desarrolle alguna aplicación para ellos. ¿Cuál de ellos consideraría como más difícil de implementar? ¿Cuál es el más fácil? Explique.

Capítulo 14

Aplicaciones Exitosas de la Computación Evolutiva

En este capítulo revisaremos brevemente algunas aplicaciones exitosas de la computación evolutiva a problemas del mundo real. La intención es mostrar el potencial de estas técnicas en la solución de problemas de norme complejidad.

14.1 Diseño de Péptidos

Un equipo de *Unilever Research* ha usado algoritmos genéticos combinados con redes neuronales para diseñar nuevos péptidos bactericidas para usarse en limpiadores anti-bacterianos y preservativos de alimentos.

Las redes neuronales se utilizaron para predecir la actividad bactericida en los péptidos, y posteriormente se combinaron con algoritmos genéticos para optimizar péptidos virtuales. El resultado fue la generación de más de 400 bactericidas virtuales potencialmente activos, de los cuales 5 fueron sintetizados.

14.2 Optimización de Estrategias de Producción

La empresa de software escocesa *Quadstone*, usó algoritmos genéticos para resolver un problema de optimización de estrategias de producción de British Petrol. El objetivo era maximizar el retorno financiero de un grupo de campos petrolíferos y de gas interdependientes. El problema es bastante complejo debido a los muchos compromisos posibles entre beneficios y penalizaciones. Sin embargo, aún

una mejora relativamente pequeña puede traer enormes ahorros a la larga, ya que su impacto es acumulativo.

El uso de algoritmos genéticos en este problema produjo retornos netos substancialmente mejores que los producidos previamente por cualquier planeador humano o por cualquier otra técnica de optimización.

14.3 Predicción

La empresa holandesa *Cap Gemini* y la empresa británica *KiQ Ltd* han desarrollado de forma conjunta un sistema llamado **Omega**,el cual usa algoritmos genéticos para resolver problemas de mercadotecnia, crédito y aplicaciones financieras relacionadas.

Omega usa como punto de partida un portafolio de comportamiento previo de un cliente, ya partir de él genera un modelo matemático que puede usarse posteriormente para predecir comportamientos de clientes que se encuentren fuera de los portafolios conocidos. Este software puede utilizarse para evaluar solicitudes de crédito, generar listas de correo (para publicidad), modelar lealtad de los clientes a un producto, y para detectar fraudes.

Recientemente, un banco holandés comparó **Omega** contra su sistema de asignación de crédito tradicional (basado en sistemas expertos), encontrando que **Omega** era substancialmente superior tanto en cantidad (ofrecía más préstamos) como en calidad (se reducía el riesgo en los créditos) de los préstamos evaluados.

14.4 Diseño de un Sistema de Suspensión

Investigadores del *KanGAL*, el "Laboratorio de Algoritmos Genéticos de Kanpur", en la India, han utilizado esta técnica para diseñar un sistema de suspensión para un automóvil que es más cómodo que el previamente utilizado por una empresa automotriz reconocida mundialmente.

Utilizando un modelo tridimensional de un automóvil, estos investigadores optimizaron el diseño de los amortiguadores y los resortes de rigidez del vehículo. Las simulaciones efectuadas muestran que el sistema generado por el algoritmo genético hace que los pasajeros sufran menor aceleración vertical, de manera que se disfruta de un mayor confort que con los sistemas utilizados previamente por el fabricante de automóviles en cuestión.

14.5 Programación de Horarios

Un grupo de investigadores del *Joszef Stefan Institute*, en Eslovenia, desarrollaron un sistema de programación de horarios basado en técnicas evolutivas. El sistema ha reducido sustancialmente los costos de energía en la planta de prensado de una fábrica de automóviles.

El sistema utiliza una heurística "avariciosa" (*greedy*, en inglés) para diseñar horarios iniciales que luego son utilizados como punto de partida por una técnica evolutiva que minimiza el consumo de energía durante las horas pico.

14.6 Diseño de una Red de Agua Potable

Tras un estudio en la región de York (en Ontario, Canadá), se determinó que la población de aquel lugar se duplicará en el período de 1996 a 2031. De tal manera, se hace necesario extender y reforzar la infraestructura de su red de suministro de agua potable.

Para tener una idea de la magnitud del problema, hay que tener en cuenta lo siguiente:

- Se usaron 300 nuevos tipos de tubos, cada uno de los cuales podía adoptar uno de 14 diámetros disponibles (de 300 mm a 2100 mm)
- Combinando las demás variantes del problema (p.ej. ubicación de las estaciones de bombeo, etc.) se estimó que el tamaño del espacio de búsqueda era de aproximadamente 10^{357}. Este número es superior a la cantidad de átomos en el universo.

Antes de intentar resolver este problema, se recurrió a un análisis que incluyó lo siguiente:

- Estudios de campo extensivos de la red y de las estaciones de bombeo
- Se construyó un modelo de todos los cauces principales de la región, usando **StruMap**, que es un sistema de información geográfica que tiene un módulo integrado para resolver redes hidráulicas.

Antes de intentar usar técnicas evolutivas en este problema, se recurrió al análisis efectuado por expertos humanos. Dicho estudio sólo reportó resultados

hasta el año 2011, y hubieron de extrapolarse los resultados para el año 2031. El costo estimado de la extensión y reforzamiento de la red de agua potable fue de $150 millones de dólares.

Usando la información disponible, se procedió entonces a utilizar GAnet, que es una biblioteca de clases para desarrollar algoritmos genéticos. GAnet incluye rutinas para simular redes hidráulicas y permite el manejo de restricciones duras y blandas.

GAnet propuso lo siguiente:

- Agregar 85 tuberías principales a las 750 ya existentes.
- Se propusieron 6 nuevas estaciones de bombeo y se sugirió expandir 3 de las ya existentes, totalizando 42 nuevas bombas.
- Se propuso sacar de circulación a 3 estaciones de bombeo.
- Se propusieron 7 nuevos tanques elevados y se sugirió sacar de circulación a 2 de los existentes.

El costo estimado de los cambios propuestos por GAnet fue de $102 millones de dólares. Esta solución resultó 35% más económica que la propuesta por diseñadores humanos expertos. Esto se tradujo en un ahorro estimado de unos $54 millones de dólares.

GAnet es uno de los 3 productos principales de software de la empresa *Optimal Solutions*. Los otros 2 son:

1. **GAser**: *Genetic Algorithms Sewer Analysis*, que se usa para optimizar redes de alcantarillado.
2. **GAcal**: Herramienta para calibrar modelos hidráulicos.

La empresa *Optimal Solutions* fue fundada en 1996. Su página web está en:

`http://www.ewan.co.uk/os.html`

Esta empresa surgió a partir de la investigación del Dr. Dragan Savic y sus colaboradores en la Universidad de Exeter (en Inglaterra).

Actualmente, *Optimal Solutions* tiene varios contratos dentro y fuera del Reino Unido y se les considera la vanguardia en cuanto a optimización hidráulica con técnicas no convencionales

Esta empresa está asociada con *Ewan Associates* en lo que constituye un ejemplo a seguir de vinculación entre una universidad y la industria.

Figura 14.1: Dragan Savic.

14.7 Optimización de Losas de Concreto Prefabricadas

El siguiente es un problema que fue abordado en el *Engineering Design Centre* de la Universidad de Plymouth, en Inglaterra.

La empresa británica *Redlands* produce losas prefabricadas de concreto en grandes volúmenes. El diseño de la topología (o sea la forma) de dichas losas ha sido optimizado por matemáticos usando técnicas tradicionales de optimización.

Desde el punto de vista de ingeniería civil, la losa se representa como una placa sujeta a diversos tipos de cargas, comenzando con una puntual (el caso más simple) hasta varias cargas distribuidas en diferentes partes de su superficie. El problema tiene una alta dimensionalidad (unas 400 variables de desición, las cuales son todas continuas). El costo computacional asociado con la solución de este problema también es muy alto. Una sola evaluación de la función de aptitud toma alrededor de 10 minutos en una estación de trabajo *Sun Sparc* con 6 procesadores

Para la solución del problema se recurrió al uso de ANSYS, que es un programa comercial para realizar análisis por medio del elemento finito. Debido al alto costo computacional asociado con el problema, resultó muy obvia la necesidad de desarrollar un esquema que redujera al mínimo posible la cantidad de evaluaciones de la función de aptitud. La pregunta fundamental era si se podrá diseñar un algoritmo genético el cual, con un número relativamente bajo de evaluaciones de la función de aptitud, pudiese mejorar las soluciones producidas por expertos humanos.

Para utilizar computación evolutiva se recurrió a un algoritmo genético con representació real implementado en FORTRAN. A fin de mantener dentro de un límite razonable el costo computacional del algoritmo, se recurrió a un esquema distribuido para evaluar la función de aptitud. La clave fundamental de la propuesta fue una representación con cambio de granularidad, en la cual se empezaba con una granularidad gruesa (es decir, un intervalo grande entre cada valor de las variables) y se hacía correr al algoritmo genético durante un cierto número de generaciones (pre-convenido, o hasta alcanzar la convergencia nominal[1]). Una vez hecho eso, se cambiaba la granularidad a otra más fina, y se procedía a continuar con las iteraciones del algoritmo genético. Esto permitió un acercamiento progresivo hacia la zona factible, manteniendo un costo computacional razonable para el algoritmo genético.

La mejor solución obtenida por el algoritmo genético resultó entre 3% y 5% más económica que la mejor encontrada por expertos humanos con técnicas tradicionales de optimización. Esto se tradujo en ahorros de aproximadamente 1.3 millones de libras esterlinas por año para *Redlands*. Si consideramos que esta empresa invirtió unas 50 mil libras esterlinas en total en esta inveestigación, el retorno de su inversión fue cuantioso.

14.8 Problemas Propuestos

1. Investigue aplicaciones de los algoritmos genéticos en las siguientes áreas:

    - Planeación de movimientos de robots
    - Diseño de armaduras (planas y espaciales)
    - Inferencia gramatical (aplicada a reconocimiento de voz)
    - Diseño de redes de telecomunicaciones
    - Diseño de circuitos eléctricos
    - Compresión de datos
    - Optimización de consultas en bases de datos

    Se le recomienda consultar:

[1]Hay varios criterios para definir convergencia nominal. Uno que suele adoptarse es el de la similitud genotípica. Cuando la mayoría de los individuos en la población son muy similares al nivel genotipo, puede decirse que se alcanzó convergencia nominal

- Mitsuo Gen and Runwei Cheng, *Genetic Algorithms & Engineering Optimization*, Wiley Series in Engineering Design and Automation. John Wiley & Sons, New York, 2000.
- Lawrence Davis (editor), *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, New York, New York, 1991.
- Thomas Bäck, David B. Fogel & Zbigniew Michalewicz (editor), *Handbook of Evolutionary Computation*, Institute of Physics Publishing and Oxford University Press, New York, 1997.

2. Busque más aplicaciones de la computación evolutiva a problemas del mundo real. Vea, por ejemplo, la página web de *Evonet* (hay diversos espejos de esta página en Europa):

   `http://ls11-www.informatik.uni-dortmund.de/evonet/Coordinator/evonet.html`

3. Investigue sobre posibles aplicaciones de la computación evolutiva a problemas de su comunidad. Identifique posibles problemas y la forma en que éstos se resolverían usando algoritmos genéticos.

Capítulo 15

AGs Paralelos

15.1 Nociones de Paralelismo

Podemos definir el **procesamiento en paralelo** como la ejecución concurrente (o simultánea) de instrucciones en una computadora. Dicho procesamiento puede ser en la forma de eventos que ocurran [152]:

1. durante el mismo intervalo de tiempo
2. en el mismo instante
3. en intervalos de tiempo traslapados

La motivación más obvia del paralelismo es el incrementar la eficiencia de procesamiento.

Existen muchas aplicaciones que demandan grandes cantidades de tiempo de procesamiento y que, por ende, resultan beneficiadas de contar con arquitecturas en paralelo.

Una de las frecuentes confusiones respecto al paralelismo es que se cree que al contar con una computadora que tenga n procesadores trabajando en el mismo problema, éste podrá resolverse n veces más rápido. Esto es falso.

Al usar varios procesadores para una misma tarea, debemos tomar en cuenta que existirán:

- Problemas de comunicación entre ellos.
- Conflictos al intentar accesar la memoria.

Figura 15.1: Michael J. Flynn.

- Algoritmos ineficientes para implementar el paralelismo del problema. paralelismo del problema.

Por tanto, si tenemos n procesadores, el incremento de velocidad normalmente no será de n veces.

Existe un límite inferior respecto del incremento de velocidad real al tener n procesadores. A este límite inferior se le conoce como la **Conjetura de Minsky**, y es de $\log_2 n$.

Aunque el límite superior depende realmente de si se considera a todo el programa (incluyendo la parte de entrada y salida, la cual suele ser secuencial), suele aceptarse que éste está definido por $\frac{n}{logn}$ [133].

De estos 2 límites puede inferirse fácilmente que no resulta útil agregar más y más procesadores, si lo que queremos es hacer más rápida a una computadora.

La eficiencia de un sistema de cómputo se mide en términos de sus capacidades tanto en hardware como en software. A dicha medida de eficiencia se le conoce como **rendimiento total** (*throughput*) y se define como la cantidad de procesamiento que puede realizarse en un cierto intervalo de tiempo.

Una técnica que ha conducido a incrementos notables del rendimiento total de un sistema de cómputo es el **proceso de encauzamiento** (*pipelining*).

Este proceso de encauzamiento es análogo a una línea de ensamblaje en una planta industrial. Una función a ejecutarse por una computadora es dividida en sub-funciones más pequeñas, y se diseña hardware separado (llamado **etapa**) para cada una de estas subfunciones. Estas etapas están conectadas entre sí, de manera que forman un solo cauce (o *pipeline*) que realiza la función original.

Michael J. Flynn [78] introdujo un esquema para clasificar la arquitectura de una computadora basado en la forma en la que la máquina relaciona sus instrucciones con los datos que procesa. Flynn definió el término **stream** (flujo) como una secuencia de elementos, ya sea datos o instrucciones, ejecutados u operados por un procesador.

La clasificación de Flynn es la siguiente:

- **SISD**: *Single Instruction Stream, Single Data Stream*
- **SIMD**: *Single Instruction Stream, Multiple Data Stream*
- **MISD**: *Multiple Instruction Stream, Single Data Stream*
- **MIMD**: *Multiple Instruction Stream, Multiple Data Stream*

Una computadora **SISD** es la computadora serial convencional que todos conocemos, en la cual las instrucciones se ejecutan una por una, y se usa una sola instrucción para lidiar con, cuando mucho, una operación sobre los datos. Aunque es posible introducir cierto nivel de paralelismo en estas computadoras (usando *pipelining*), la naturaleza secuencial de la ejecución de sus instrucciones la coloca en esta categoría.

En una computadora **SIMD**, una sola instrucción puede iniciar un gran número de operaciones. Estas instrucciones (llamadas **vectoriales**) se ejecutan de manera secuencial (una a la vez), pero son capaces de trabajar sobre varios flujos de datos a la vez. También en este caso es posible usar *pipelining* para acelerar la velocidad de procesamiento.

La clase **MISD** implica la ejecución de varias instrucciones operando simultáneamente sobre un solo dato. Este modelo es únicamente teórico, porque no existen computadoras que caigan dentro de esta categoría.

Una computadora **MIMD** se caracteriza por la ejecución simultánea de más de una instrucción, donde cada instrucción opera sobre varios flujos de datos. Ejemplos de esta arquitectura son los sistemas multiprocesadores. Para lo relacionado con AGs paralelos, sólo hablaremos de las computadoras SIMD y MIMD.

15.2 AGs Paralelos

Una vez revisados algunos conceptos básicos de paralelismo, procederemos a analizar los esquemas más comunes de paralelización de un algoritmo genético.

Figura 15.2: Esquema de paralelización global de un algoritmo genético.

15.2.1 Paralelización global

El método más simple de paralelizar un AG es la llamada **paralelización global**, la cual se ilustra en la figura 15.2. En este caso, sólo hay una población, como en el AG convencional, pero la evaluación de los individuos y los operadores genéticos se paralelizan de forma **explícita**.

Puesto que sólo hay una población, la selección considera a todos los individuos y cada individuo tiene oportunidad de aparearse con cualquier otro (o sea, hay **apareamiento aleatorio**). Por lo tanto, el comportamiento del AG simple permanece sin cambios.

La paralelización global es un método relativamente fácil de implementar y puede obtenerse un incremento significativo de velocidad si los costos de comunicación no dominan los costos de procesamiento. Una observación importante es que no debe confundirse el concepto de **paralelismo implícito** de un AG con el de paralelismo explícito.

A la paralelización global también se le conoce como **AG panmítico**, pues se cuenta con un solo depósito de material genético (*gene pool*), o sea con una sola

Figura 15.3: Esquema de funcionamiento de un algoritmo genético paralelo de grano grueso.

población.

Los AGs panmíticos son útiles cuando el costo de evaluar la función de aptitud es elevado (p.ej., una simulación). En el AG panmítico no se requieren nuevos operadores ni nuevos parámetros y la solución encontrada será la misma que la producida con un AG convencional (o sea, serial).

Es importante hacer notar que aunque el paralelismo global normalmente es **síncrono** (o sea, que el programa se detiene y espera a recibir los valores de aptitud de toda la población antes de proceder a producir la siguiente generación), puede implementarse también de forma **asíncrona**, aunque en ese caso, su funcionamiento ya no resultará equivalente al de un AG convencional.

Además de paralelizarse la evaluación de la función de aptitud, en el paralelismo global es posible incluir también los operadores genéticos, pero dada la simplicidad de éstos, no suelen paralelizarse, pues los costos de comunicación disiparían cualquier mejora en el desempeño del programa.

15.2.2 AGs de grano grueso

Una idea más sofisticada es usar los llamados **AGs de grano grueso**, cuyo funcionamiento se ilustra en la figura 15.3. En este caso, la población del AG se divide en múltiples subpoblaciones o **demes** que evolucionan de manera aislada la mayor parte del tiempo, aunque intercambian individuos ocasionalmente.

A este intercambio de individuos se le llama **migración**, y se considera como un nuevo operador genético. Además de requerirse parámetros adicionales en este caso, el comportamiento de un AG de grano grueso es diferente del de un AG convencional.

A los **AGs de grano grueso** se les suele llamar también **AGs distribuidos**, porque suelen implementarse en computadoras MIMD con memoria distribuida. Asimismo, algunos autores los llaman también **AGs de isla**, haciendo alusión a un modelo poblacional usado en genética en el cual se consideran **demes** relativamente aislados. A este modelo se le conoce como **modelo de isla**.

Los AGs de grano grueso son muy populares debido a varias razones:

- Son una extensión muy simple de los AGs seriales. Simplemente se toman unos cuantos AGs convencionales (seriales), se corre cada uno de ellos en un procesador diferente y, a ciertos intervalos de tiempo, se intercambian unos cuantos individuos entre ellos.

- Aunque no se tenga acceso a una arquitectura paralela, puede implementarse un AG de grano grueso a través de una simulación efectuada en una red de estaciones de trabajo, o incluso en una computadora con un solo procesador haciendo la simulación mediante software (usando por ejemplo MPI o PVM).

- Se requiere relativamente de poco esfuerzo para convertir un AG serial en un AG de grano grueso. La mayor parte de la programación permanece igual, y sólo se requieren ciertas rutinas adicionales para implementar la migración.

Los parámetros que requieren los AGs de grano grueso son:

- Número de *demes* (esto puede estar determinado por el hardware disponible).

- Tamaño de cada *deme*.

- Estructura de la interconexión (o sea, la topología).

- Intervalo de migración

- Tasa de migración

- Radio de selección

- Radio de migración

De entre estos parámetros, algunos como la **topología**, juegan un papel preponderante en el desempeño del AG. La topología determina qué tan rápido (o qué tan lentamente) se disemina una buena solución hacia los otros **demes**.

Si se usa una topología dispersamente conectada (con un diámetro grande), las soluciones se diseminarán más lentamente y los **demes** estarán más aislados entre sí, permitiendo la aparición de soluciones diferentes, favoreciendo probablemente la diversidad.

La topología juega también un papel preponderante en el costo de las migraciones. Por ejemplo, una topología densamente conectada puede promover una mejor mezcla de individuos, pero a un costo computacional más alto.

Se sabe, por ejemplo, que una topología densa tiende a encontrar soluciones globales con un menor número de evaluaciones de la función de aptitud que si se usa una topología dispersa.

También es posible usar topologías **dinámicas**, en las que los **demes** no están limitados a poder comunicarse sólo con un cierto conjunto predefinido de **demes**, sino que envía sus migrantes a aquellos **demes** que satisfacen ciertos criterios.

La idea de este esquema es que puedan identificarse los **demes** donde los migrantes tienen mayores probabilidades de producir algún efecto. Usualmente, se usa la diversidad como el criterio principal para definir qué tan adecuado es un cierto **deme**.

Es importante mantener en mente la idea de que una topología es una estructura lógica que puede diferir de la estructura de hardware disponible. Es decir, la topología de un AG paralelo no necesariamente debe coincidir con la de nuestra computadora. El problema de hacer esto, sin embargo, es que los costos de comunicación resultantes pueden ser muy elevados.

Relacionado con las topologías se encuentra el concepto de **vecindario**.

El vecindario se refiere al área dentro de la cual puede moverse un migrante de un cierto **deme**.

Asociado al vecindario se encuentra el concepto de **radio de selección**, que se refiere a la cantidad de vecinos entre los cuales se puede efectuar la selección. Es común usar un radio de selección de cero, o sea, efectuar la selección sólo dentro del mismo **deme**, aunque cualquier otro valor es válido.

Es común usar vecindarios compactos en computación evolutiva, motivados por el hecho de qe en la naturaleza, las poblaciones están limitadas geográficamente.

En AGs paralelos, es fácil definir vecindarios compactos, y de ahí que sean tan populares en computación evolutiva.

Figura 15.4: Algunas topologías posibles.

15.2.3 AG de grano fino

Otra forma de paralelizar un AG es usando un esquema de **grano fino**. En este caso, la población de un AG se divide en un gran número de subpoblaciones muy pequeñas. De hecho, el caso ideal sería tener sólo un individuo por cada unidad de procesamiento disponible.

Este modelo es adecuado para arquitecturas masivas en paralelo, aunque puede implementarse en cualquier tipo de multiprocesador.

El problema del paralelismo de grano fino es que el costo de comunicación entre los procesadores puede hacer que el desempeño del algoritmo se degrade con relativa facilidad.

Es común implementar este tipo de paralelismo colocando los individuos en una malla bidimensional, debido a que ésta es la topología usada en hardware para muchas arquitecturas masivas en paralelo.

Resulta difícil comparar de manera justa a un AG paralelo de grano fino con uno de grano grueso, y los pocos estudios al respecto suelen enfatizar sólo una cierta métrica (por ejemplo, la calidad de las soluciones encontradas). De tal forma, no hay un claro ganador entre estos 2 esquemas.

15.2.4 Esquemas híbridos

Otra posibilidad para implementar un AG paralelo es combinar los esquemas descritos anteriormente. Debe cuidarse, sin embargo, de que el esquema resultante no sea más complejo que los esquemas originales. A continuación veremos varios híbridos posibles.

Un posible híbrido consiste en usar un AG de grano fino a bajo nivel y otro de grano grueso a alto nivel, tal y como se muestra en la figura 15.5.

Un ejemplo de este tipo de híbrido es el AG propuesto por Gruau [118], en el cual la población de cada **deme** se coloca en una malla bidimensional y los **demes** se conectan entre sí en forma de toroide bidimensional. La migración entre los **demes** vecinos ocurre a intervalos regulares.

Otro posible esquema híbrido consiste en usar una forma de paralelización global en cada uno de los **demes** de un AG de grano grueso. Este esquema se ilustra en la figura 15.6.

En este caso, la migración ocurre entre los **demes** de manera similar a un AG de grano grueso, pero la evaluación de los individuos se maneja en paralelo. Esta técnica no introduce nuevos problemas analíticos, y puede ser muy útil cuando se

Figura 15.5: Ejemplo de un esquema híbrido en el que se combina un AG de grano grueso (a alto nivel) con un AG de grano fino (a bajo nivel).

Figura 15.6: Un esquema híbrido en el cual se usa un AG de grano grueso de alto nivel donde cada nodo es a su vez un AG con paralelismo global.

Figura 15.7: Un esquema híbrido en el que se usa un AG de grano grueso tanto a alto como a bajo nivel. A bajo nivel, la velocidad de migración es mayor y la topología de comunicaciones es mucho más densa que a alto nivel.

trabaja con aplicaciones en las cuales la mayor parte del tiempo de procesamiento lo consume la evaluación de la función de aptitud.

Un tercer método híbrido podría consistir en usar un AG de grano grueso tanto a bajo como a alto nivel. Un esquema de este tipo se ilustra en la figura 15.7.

En este caso, la idea es forzar el mezclado panmítico a bajo nivel usando una alta tasa de migración y una topología densa, y usar una baja tasa de migración a alto nivel. Este híbrido sería equivalente en complejidad a un AG de grano grueso, si consideramos a los grupos de subpoblaciones panmíticas como un solo **deme**. Según Cantú Paz [39], este esquema nunca ha sido implementado.

15.2.5 Tipos de Arquitecturas

Otra forma de hablar sobre AGs paralelos, es desde la perspectiva del tipo de arquitectura computacional a utilizarse. Desde este punto de vista, podemos hablar fundamentalmente de usar:

1. SIMD

2. MIMD

Figura 15.8: Un ejemplo de arquitectura SIMD.

15.2.5.1 SIMD

En este caso, cada elemento de procesamiento (EP) tiene su propia memoria y controla su propio espacio de direccionamiento, aunque también podría haber una sola memoria global compartida por todos los EPs.

- Las arquitecturas SIMD normalmente tienen una forma de malla (*mesh*) o de toroide.
- Esta arquitectura (SIMD) suele usarse para AGs de grano fino (o sea, para **demes** de tamaño reducido: 12 a 15 individuos cada uno).
- Sin embargo, puede implementarse un AG de grano grueso con la misma arquitectura si se usan tamaños mayores de **demes** (p.ej. 50 ó 100 individuos).
- El grado de conectividad es mucho más importante que la estructura de las interconexiones en una arquitectura SIMD. Se sabe que un grado de conectividad de alrededor de 6 es razonablemente bueno.
- El uso de una arquitectura SIMD está asociado con el uso de migración y dicho operador puede complicarse bastante, dependiendo del grado de conectividad de la arquitectura.

15.2.5.2 Migración

Dos son los parámetros principales relacionados con la migración:

1. **Vecindario de migración**: hacia qué demes podemos migrar un individuo.

2. **Probabilidad de migración**: ¿cuál es la probabilidad de aceptar a un migrante en un cierto **deme** (suelen usarse valores altos, p.ej. 0.8)?

Los puntos importantes relacionados con la migración son 2:

1. ¿A quién importar en una población?
2. ¿A quién reemplazar en una población?

Existen varios criterios para llevar a cabo estas 2 operaciones:

- Importar al azar y reemplazar al azar
- Importar al azar y reemplazar al peor individuo en el **deme**.
- Importar el mejor y reemplazar al azar.
- Importar el mejor y reemplazar el peor.
- Se sabe que la política de reemplazo no es muy importante (no parece tener un efecto significativo en el desempeño de un AG paralelo).
- Sin embargo, el criterio de importación sí es importante (importar al mejor parece funcionar bien).

15.2.5.3 MIMD

Las arquitecturas MIMD (ver figura 15.9) pueden ser de 2 tipos:

1. **Descentralizadas**: tienen poca o ninguna memoria global.
2. **Centralizadas**: cuentan con una memoria global compartida.

Las arquitecturas MIMD suelen asociarse con los AGs de grano grueso.

- En la arquitectura MIMD suele tenerse un número pequeño de **demes** (normalmente, menor a 40), pero el tamaño de cada uno de ellos suele ser grande.
- Es posible usar la misma representacióin para cada **deme**, o mezclar diferentes representaciones.

Figura 15.9: Un ejemplo de arquitectura MIMD.

Figura 15.10: Una topología que suele usarse con las arquitecturas MIMD es la de árbol.

Figura 15.11: Topología de anillo.

- Los **demes** suelen diferenciarse debido al particionamiento del espacio de búsqueda.
- Las políticas de migración, en este caso, están dictadas por el propósito de los **demes**.

La migración en este caso introduce 2 nuevos parámetros:

1. ¿Con qué frecuencia exportar? (siempre se exporta el mejor). Si se hace con mucha frecuencia, se produce disrupción. Si se hace con poca frecuencia, habrá poca recombinación y puede producirse convergencia prematura en ciertos **demes**.
2. ¿A qué **deme** exportar? Normalmente se usa una de las 2 s iguientes opciones:
   - Exportar el mejor individuo hacia el peor deme.
   - Exportar hacia el deme donde se tenga una mejor correspondencia (*matching*) con respecto al individuo elitista, medida usando la distancia de Hamming (en el genotipo).

Otra posibilidad para una arquitectura MIMD es usar una topología de anillo como la mostrada en la figura 15.11.

En este tipo de topología, la búsqueda puede hacerse más local cambiando la precisión de la representación.

Otra posibilidad es usar una topología de grafo con k interconexiones, o una topología de estrella, como las que se muestran en la figura 15.12.

Figura 15.12: Topología de grafo y de estrella.

En esta topología, se usan típicamente menos de 100 procesadores.

Algunas opciones interesantes de la topología de grafo con k interconexiones (o de estrella) son las siguientes:

1. Mientras que la arquitectura es MIMD, pueden usarse la misma estructura y las mismas opciones para migración que con la arquitectura SIMD.

2. **Pizarrones**: Usando datos globales, cada **deme** decide por sí mismo cuándo cambiar su "dirección" de búsqueda.

15.2.6 Métricas

Otro punto interesante relacionado con los AGs paralelos son las **métricas**.

Normalmente, se consideran 2 de las métricas usadas con AGs simples:

- **Velocidad de convergencia**: Tiempo (generaciones) en alcanzar el óptimo.

- **Precisión de la respuesta obtenida**: ¿Qué tan buena es la solución con respecto a la obtenida con otras técnicas?

- **Diversidad**: El grado en el cual los organismos (de una sola o de varias poblaciones) permanecen diferentes.

Sin embargo, hay al menos una métrica adicional que es exclusiva de los AGs paralelos:

- **Velocidad de propagación de los esquemas**: ¿qué tan bien distribuidos están los esquemas "aptos"? O sea, ¿qué tan útil resulta la migración?

15.2.7 Midiendo la diversidad

Existen expresiones estándar para medir la diversidad de un AG (serial o paralelo).

Consideremos por ejemplo la siguiente:

$$\delta = \frac{1}{l} \times \sum_{i=1}^{l} \left(1 - 2 \times \left| \frac{\sum_{j=1}^{m} \sum_{k=1}^{n} bit(i,k,j)}{m \times n} \right| \right)$$

l = Longitud cromosómica.
m = número de **demes**.
n = tamaño de cada **deme**.
$bit(.)$ = Valor del i-ésimo bit en el k-ésimo miembro del j-ésimo **deme**.
δ = diversidad.

En esta fórmula, $\delta \in [0, 1]$ representa la diversidad de una población (o conjunto de poblaciones).

Si las cadenas consisten de puros ceros,$\delta = 0$.
Si las cadenas consisten de puros unos, $\delta = 0$.
Si las cadenas son del tipo 101010...10, $\delta = 0$.

15.2.8 Velocidad de propagación de esquemas

¿A qué se refiere la velocidad de propagación de esquemas?

Se refiere no sólo al porcentaje de **demes** en los que un cierto esquema está presente, sino también al porcentaje en el cual dicho esquema está presente en un **deme** vecino.

¿Cómo medimos la propagación de esquemas?

- Idealmente, deberíamos conocer de antemano cuáles son los buenos esquemas.
- Esto, sin embargo, es imposible en la práctica.

Una alternativa viable es:

- Escoger varios esquemas de antemano.
- Hacer que la propagación de esquemas sea la fracción máxima de **demes** en la cual aparece un esquema.

Ejemplo del cálculo de la propagación de esquemas (SP):

Esquemas seleccionados	% de demes en los que aparecen
*1*10*	3/9
*110**	4/9
*10*0*	5/9

En este caso: SP = 5/9

15.3 Problemas Propuestos

1. ¿Tiene sentido usar migración en un algoritmo SIMD de grano fino? Explique.

2. ¿Por qué no tiene sentido tener las regiones de selección y migración en el mismo vecindario? Explique.

3. ¿Cuál sería la desventaja de organizar un algoritmo SIMD en una topología que no correspondiera a un *mesh* (es decir, en una topología distinta a un toroide)?

Capítulo 16

Técnicas Evolutivas Alternativas

En este capítulo discutiremos brevemente algunos de los paradigmas emergentes dentro de la computación evolutiva. Nuestra discusión se centrará fundamentalmente en las técnicas siguientes:

- Evolución Diferencial
- Modelos Probabilísticos
- Evolución Simulada

16.1 Evolución Diferencial

Se refiere a una rama de la computación evolutiva desarrollada por Rainer Storn y Kenneth Price [213, 214] para optimización en espacios continuos.

En la Evolución Diferencial (ED), las variables se representan mediante números reales (o sea, opera a nivel fenotípico). La población inicial se genera aleatoriamente, aunque se usan reglas de reparación que aseguren que cada variable se genere dentro de los límites requeridos. Posteriormente, se selecciona aleatoriamente un individuo para reemplazo y se seleccionan 3 individuos como padres. Uno de los 3 padres seleccionados es el "padre principal". Con alguna probabilidad, se cambia cada variable del padre principal, de tal forma que al menos una de sus variables sea modificada.

El cambio se efectúa agregando al valor de la variable una razón de la diferencia entre los dos valores de esta variable en los otros dos padres. En esencia, el vector del padre principal se perturba con el vector de los otros dos padres. Este

proceso representa el equivalente de la cruza en ED. Si el valor resultante es mejor que el elegido para reemplazo, entonces lo reemplaza. De lo contrario, se retiene el vector elegido para reemplazo.

La ED difiere entonces de los AGs en los aspectos siguientes:

1. La ED usa representación real, mientras el AG suele usar representación binaria.
2. En ED se usan 3 padres, en vez de los 2 que usa el AG.
3. En ED, se genera un solo hijo de la cruza y éste se produce a partir de la perturbación de uno solo de sus padres.
4. En ED, el nuevo padre reemplaza a un vector de la población elegido aleatoriamente sólo si es mejor que él. En el AG, siempre se reemplaza la población anterior.

16.2 Modelos Probabilísticos

Los EDAs (*Estimation of Distribution Algorithms*) [172] son algoritmos que usan un modelo probabilístico de soluciones promisorios para guiar la exploración posterior del espacio de búsqueda. Su motivación principal fue la limitante de los AGs en problemas en los cuales los bloques constructores no se encuentran fuertemente unidos. En dichos problemas, se vuelve necesario aprender la estructura del problema en tiempo real, de manera que se pueda usar esta información para realizar la recombinación de la manera más apropiada. Los EDAs son precisamente este tipo de algoritmos.

Los EDAs usan una población inicial generada aleatoriamente, al igual que el AG simple. La distribución de probabilidad verdadera se estima a partir de un conjunto selecto de soluciones de esta población. Se generan luego nuevas soluciones de acuerdo a esta estimación. Las nuevas soluciones se agregan a la población original, reemplazando algunas de las anteriores.

Este proceso continúa hasta alcanzar la condición de paro del algoritmo.

Los EDAs son, por tanto, similares a los AGs, excepto por el hecho de que no usan cruza ni mutación.

Los operadores genéticos se reemplazan por los dos pasos siguientes:

1. Se construye un modelo (un estimado de la verdadera distribución) de las soluciones prometedoras.

2. Se generan nuevas soluciones de acuerdo al modelo construido.

Aunque los EDAs procesan soluciones de manera diferente que los AGs simples, se ha demostrado teórica y empíricamente que los resultados de ambas técnicas son muy similares. Por ejemplo, el AG simple con cruza uniforme es asintóticamente equivalente al algoritmo de distribución marginal univariada que supone que las variables son independientes.

Una estimación de distribución puede capturar la estructura de los bloques constructores de un problema de forma muy precisa, de forma que se asegure su efectiva recombinación. Esto resulta en un desempeño lineal o subcuadrático de los EDAs en estos problemas.

De hecho, si se obtiene una estimación de distribución precisa que capture la estructura del problema a resolverse,los EDAs convergen de acuerdo a como lo indica la teoría de los AGs. El problema es que determinar dicha estimación es una tarea que dista de lo trivial. Por lo tanto, hay un compromiso entre precisión y eficiencia de la estimación probabilística adoptada.

16.3 Evolución Simulada

La evolución simulada (SE) es una heurística iterativa general que fue propuesta por Ralph Kling [143] en el contexto de optimización combinatoria.

La SE cae en la categoría de algoritmos que enfatizan la liga conductista entre padres e hijos o entre poblaciones reproductivas. En contraste, los AGs enfatizan la liga genética. La SE combina las mejoras iterativas con la perturbación constructiva, y evade los mínimos locales siguiendo un enfoque basado en perturbaciones estocásticas. Opera iterativamente una secuencia de 3 pasos (evaluación, selección y asignación) sobre una solución. La selección y la asignación constituyen un movimiento compuesto de la solución actual a otra solución factible en el espacio de estados del problema. La SE parte de una solución generada aleatoriamente o mediante un proceso determinístico. Esta solución inicial debe ser válida. Una solución es vista como un conjunto de objetos movibles (módulo). Cada elemento tiene una medida de bondad en el intervalo [0,1].

El ciclo principal del algoritmo consta de 3 pasos: evaluación, selección y asignación. Estos pasos se ejecutan repetitivamente hasta que se cumpla alguna condición de detención.

En el paso de **evaluación**, se estima la bondad de cada elemento. En el paso de **selección**, se selecciona un subconjunto de elementos de la solución actual.

A menor bondad de un cierto elemento, mayor será su probabilidad de ser seleccionado. Se usa un parámetro adicional para compensar las posibles imprecisiones de la medida de bondad.

Finalmente, el paso de **asignación** intenta asignar los elementos seleccionados a mejores posiciones.

Además de los 3 pasos antes mencionados, se fijan algunos parámetros de entrada para el algoritmo en un paso preliminar denominado **inicialización**.

16.4 El Futuro de la Computación Evolutiva

Algunas de las áreas futuras prometedoras que se contemplan actualmente en la computación evolutiva son las siguientes:

- **Metamerismo**: El proceso en el cual una unidad estructural es duplicada un cierto número de veces y durante ese proceso se reoptimiza para otros usos.

- **Auto-adaptación**: Evitar el uso de parámetros *ad-hoc* en los algoritmos evolutivos.

- **Técnicas que exploten arquitecturas paralelas**: Es importante explotar al máximo las arquitecturas paralelas mediante nuevos algoritmos evolutivos. Esto traerá importantes ganancias en términos de esfuerzo computacional, sobre todo al lidiar con problemas del mundo real.

- **Teoría**: Pruebas de convergencia, modelos matemáticos de los algoritmos evolutivos, epístasis, diversidad, etc.

- **Entender mejor la evolución natural**: Simulaciones computacionales que permitan entender las complejas interacciones que ocurren entre los seres vivos.

- **Coevolución**: Muchos investigadores han dirigido sus esfuerzos a estudiar mejor la coevolución como un modelo alternativo para resolver problemas en computación evolutiva.

- **El AG sin parámetros**: Es, sin lugar a dudas, el sueño de los expertos en computación evolutiva.

Para finalizar, vale la pena mencionar que hace algún tiempo, en la lista de distribución `GA-Digest` se discutieron ampliamente los temas de investigación considerados como más importantes en los años por venir. Tras un intenso debate, se concluyó que las 3 preguntas más importantes que debieran atacarse por los investigadores de computación evolutiva en los próximos años para lograr la madurez del área son las siguientes:

1. ¿A qué tipo de problemas deben aplicarse los algoritmos evolutivos?
2. ¿Cómo podemos mejorar nuestra comprensión del funcionamiento de los algoritmos evolutivos?
3. ¿Qué nuevas ideas pueden aplicarse a la computación evolutiva a fin de extender el paradigma (p.ej., inspiración biológica)?

Índice Analítico

Bibliografía

[1] David H. Ackley. *A Connectionist Machine for Genetic Hillclimbing*. Kluwer Academic Publishers, Boston, Massachusetts, 1987.

[2] C. A. Anderson, K. F. Jones, and J. Ryan. A two-dimensional genetic algorithm for the ising problem. *Complex Systems*, 5:327–333, 1991.

[3] Peter J. angeline and Jordan B. Pollack. Competitive Environments Evolve Better Solutions for Complex Tasks. In Stephanie Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*. Morgan Kaufmann Publishers, San Mateo, California, July 1993.

[4] Hendrik James Antonisse. A Grammar-Based Genetic Algorithm. In Gregory E. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 193–204. Morgan Kaufmann Publishers, San Mateo, California, 1991.

[5] Wirt Atmar. Notes on the Simulation of Evolution. *IEEE Transactions on Neural Networks*, 5(1):130–148, January 1994.

[6] Thomas Bäck. Self-adaptation in genetic algorithms. In F.J. Varela and P. Bourgine, editors, *Proceedings of the First European Conference on Artificial Life*, pages 263–271, Cambridge, Massachusetts, 1992. MIT Press.

[7] Thomas Bäck. Optimal Mutation Rates in Genetic Search. In Stephanie Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 2–8. Morgan Kaufmann Publishers, San Mateo, California, July 1993.

[8] Thomas Bäck. Selective Pressure in Evolutionary Algorithms: A Characterization of Selection Mechanisms. In *Proceedings of the First IEEE Conference on Evolutionary Computation*, pages 57–62, Orlando, Florida, 1994. IEEE.

[9] Thomas Bäck. *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, New York, 1996.

[10] Thomas Bäck, David B. Fogel, and Zbigniew Michalewicz, editors. *Handbook of Evolutionary Computation*. Institute of Physics Publishing and Oxford University Press, New York, 1997.

[11] Thomas Bäck and Frank Hoffmeister. Extended Selection Mechanisms in Genetic Algorithms. In Richard K. Belew and Lashon B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 92–99. Morgan Kaufmann Publishers, San Mateo, California, July 1991.

[12] Thomas Bäck, Frank Hoffmeister, and Hans-Paul Schwefel. A Survey of Evolution Strategies. In R. K. Belew and L. B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 2–9, San Mateo, California, 1991. Morgan Kaufmann Publishers.

[13] James Edward Baker. Adaptive Selection Methods for Genetic Algorithms. In John J. Grefenstette, editor, *Proceedings of the First International Conference on Genetic Algorithms*, pages 101–111. Lawrence Erlbaum Associates, Hillsdale, New Jersey, July 1985.

[14] James Edward Baker. Reducing Bias and Inefficiency in the Selection Algorithm. In John J. Grefenstette, editor, *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, pages 14–22. Lawrence Erlbaum Associates, Hillsdale, New Jersey, July 1987.

[15] James Mark Baldwin. *Development and Evolution: Including Psychophysical Evolution, Evolution by Orthoplasy and the Theory of Genetic Modes*. Macmillan, New York, 1902.

[16] Nils Aall Barricelli. Esempi numerici di processi di evoluzione. *Methodos*, pages 45–68, 1954.

[17] Nils Aall Barricelli. Symbiogenetic evolution processes realized by artificial methods. *Methodos*, IX(35–36):143–182, 1957.

[18] Nils Aall Barricelli. Numerical Testing of Evolution Theories. Part I: Theoretical Introduction and Basic Tests. *Acta Biotheoretica*, 16(1–2):69–98, 1962.

[19] Nils Aall Barricelli. Numerical Testing of Evolution Theories. Part II: Preliminary Tests of Performance, Symbiogenesis and Terrestrial Life. *Acta Biotheoretica*, 16(3–4):99–126, 1964.

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

[21] D. Beasley, D. Bull, and R. Martin. Reducing epistasis in combinatorial problems by expansive coding. In Stephanie Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 400–407, San Mateo, California, 1993. University of Illinois at Urbana-Champaign, Morgan Kauffman Publishers.

[22] A.D. Bethke. *Genetic Algorithms as Function Optimizers*. PhD thesis, Department of Computer and Communication Sciences, University of Michigan, Ann Arbor, Michigan, 1981.

[23] Peter Bienert. Aufbau einer optimierungsautomatik für drei parameter. Dipl.-Ing. thesis, 1967. (in German).

[24] Lashon B. Booker. *Intelligent Behavior as an Adaptation to the Task Environment*. PhD thesis, Logic of Computers Group, University of Michigan, Ann Arbor, Michigan, 1982.

[25] Lashon B. Booker. Improving Search in Genetic Algorithms. In Lawrence Davis, editor, *Genetic Algorithms and Simulated Annealing*. Morgan Kaufmann Publishers, San Mateo, California, 1987.

[26] George E. P. Box. Evolutionary operation: A method for increasing industrial productivity. *Applied Statistics*, 6(2):81–101, 1957.

[27] Hans J. Bremermann. The evolution of intelligence. The nervous system as a model of its environment. Technical Report 1, Contract No. 477(17), Department of Mathematics, University of Washington, Seattle, July 1958.

[28] Hans J. Bremermann. Optimization through evolution and recombination. In M. C. Yovits, G. T. Jacobi, and G. D. Goldstein, editors, *Self-Organizing Systems—1962*, pages 93–106. Spartan Books, Washington, D.C., 1962.

[29] Hans J. Bremermann. Numerical optimization procedures derived from biological evolution processes. In H. L. Oestreicher and D. R. Moore, editors, *Cybernetic Problems in Bionics*, pages 543–561. Gordon and Breach, New York, 1968.

[30] Hans J. Bremermann and M. Rogson. An evolution-type search method for convex sets. Technical Report Contracts No. 222(85) and 3656(58), ONR, Berkeley, California, May 1964.

[31] Hans J. Bremermann and M. Rogson. Global properties of evolution processes. In H. H. Pattee, E. A. Edlsack, L. Fein, and A. B. Callahan, editors, *Natural Automata and Useful Simulations*, pages 3–41. Spartan Books, Washington, D.C., 1966.

[32] A. Brindle. *Genetic Algorithms for Function Optimization*. PhD thesis, Department of Computer Science, University of Alberta, Edmonton, Alberta, 1981.

[33] Bill P. Buckles and Frederick E. Petry, editors. *Genetic Algorithms*. Technology Series. IEEE Computer Society Press, 1992.

[34] T. N. Bui and B. R. Moon. On multidimensional encoding/crossover. In Larry J. Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 49–56, San Fracisco, California, 1995. Morgan Kauffman Publishers.

[35] G. H. Burgin. On playing two-person zero-sum games against non-minimax players. *IEEE Transactions on Systems Science and Cybernetics*, SSC-5:369–370, 1969.

[36] A. W. Burks. Computation, behavior and structure in fixed and growing automata. In M. C. Yovits and S. Cameron, editors, *Self-Organizing Systems*, pages 282–309. Pergamon Press, New York, 1960.

[37] Eduardo Camponogara and Sarosh N. Talukdar. A Genetic Algorithm for Constrained and Multiobjective Optimization. In Jarmo T. Alander, editor, *3rd Nordic Workshop on Genetic Algorithms and Their Applications (3NWGA)*, pages 49–62, Vaasa, Finland, August 1997. University of Vaasa.

[38] W. D. Cannon. *The Wisdom of the Body*. Norton and Company, New York, 1932.

[39] Erick Cantú-Paz. A Survey of Parallel Genetic Algorithms. Technical Report 97003, Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana-Champaign, Urbana, Illinois, May 1997.

[40] R. Caruana and J. D. Schaffer. Representation and Hidden Bias: Gray vs. Binary Coding for Genetic Algorithms. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 132–161, San Mateo, California, 1988. Morgan Kauffman Publishers.

[41] Munirul M. Chowdhury and Yun Li. Messy Genetic Algorithm Based New Learning Method for Fuzzy Controllers. In *Proceedings of the IEEE International Conference on Industrial Technology*, page `http://eeapp.elec.gla.ac.uk/~chy/publications.html`, Shangai, China, December 1996. IEEE Service Center.

[42] Carlos A. Coello Coello, Alan D. Christiansen, and Arturo Hernández Aguirre. Automated Design of Combinational Logic Circuits using Genetic Algorithms. In D. G. Smith, N. C. Steele, and R. F. Albrecht, editors, *Proceedings of the International Conference on Artificial Neural Nets and Genetic Algorithms ICANNGA'97*, pages 335–338, Norwich, England, April 1997. University of East Anglia, Springer-Verlag.

[43] 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(4):401–414, 1998.

[44] 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):101–108, January 1997.

[45] Carlos A. Coello Coello. Constraint-handling using an evolutionary multiobjective optimization technique. *Civil Engineering and Environmental Systems*, 17:319–346, 2000.

[46] Carlos A. Coello Coello. Treating Constraints as Objectives for Single-Objective Evolutionary Optimization. *Engineering Optimization*, 32(3):275–308, 2000.

[47] Michael Conrad. Evolutionary learning circuits. *Journal of Theoretical Biology*, 46:167–188, 1974.

[48] Michael Conrad. Algorithmic specification as a technique for computing with informal biological models. *BioSystems*, 13:303–320, 1981.

[49] Michael Conrad. Natural selection and the evolution of neutralism. *BioSystems*, 15:83–85, 1982.

[50] Michael Conrad, R. R. Kampfer, and K. G. Kirby. Neuronal dynamics and evolutionary learning. In M. Kochen and H. M. Hastings, editors, *Advances in Cognitive Science: Steps Towards Convergence*, pages 169–189. American Association for the Advancement of Science, New York, 1988.

[51] Michael Conrad and H. H. Pattee. Evolution experiments with an artificial ecosystem. *Journal of Theoretical Biology*, 28:393–409, 1970.

[52] Michael Conrad and M. M. Rizki. The artificial worlds approach to emergent evolution. *BioSystems*, 23:247–260, 1989.

[53] Michael Conrad and M. Strizich. Evolve II: A computer model of an evolving ecosystem. *BioSystems*, 17:245–258, 1985.

[54] F. N. Cornett. An Application of Evolutionary Programming to Pattern Recognition. Master's thesis, New Mexico State University, Las Cruces, New Mexico, 1972.

[55] Nareli Cruz Cortés. Uso de emulaciones del sistema inmune para manejo de restricciones en algoritmos evolutivos. Master's thesis, Universidad Veracruzana, Xalapa, México, 2000.

[56] Nichal Lynn Cramer. A Representation for the Adaptive Generation of Simple Sequential Programs. In John J. Grefenstette, editor, *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, pages 183–187. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1985.

[57] J. L. Crosby. Computers in the study of evolution. *Science Progress Oxford*, 55:279–292, 1967.

[58] Charles Robert Darwin. *The Variation of Animals and Plants under Domestication*. Murray, London, second edition, 1882.

[59] Charles Robert Darwin. *On the Origin of Species by Means of Natural Selection Or the Preservation of Favoured Races in the Struggle for Life*. Cambridge University Press, Cambridge, UK, sixth edition, 1964. Originally published in 1859.

[60] Dipankar Dasgupta and Douglas R. McGregor. Nonstationary Function Optimization using the Structured Genetic Algorithm. In *Proceedings of Parallel Problem Solving from Nature (PPSN 2)*, pages 145–154, Brussels, September 1992. Springer-Verlag.

[61] Dipankar Dasgupta and Douglas R. McGregor. A more Biologically Motivated Genetic Algorithm: The Model and some Results. *Cybernetics and Systems: An International Journal*, 25(3):447–469, May-June 1994.

[62] Lawrence Davis. Job Shop Scheduling with Genetic Algorithms. In John J. Grefenstette, editor, *Proceedings of the First International Conference on Genetic Algorithms*, pages 136–140. Lawrence Erlbaum Associates, Hillsdale, New Jersey, July 1985.

[63] Lawrence Davis. Adapting Operator Probabilities In Genetic Algorithms. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 61–69, San Mateo, California, 1989. Morgan Kaufmann Publishers.

[64] Lawrence Davis, editor. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, New York, New York, 1991.

[65] 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.

[66] Kalyanmoy Deb. An Efficient Constraint Handling Method for Genetic Algorithms. *Computer Methods in Applied Mechanics and Engineering*, 1999. (in Press).

[67] Kalyanmoy Deb and Ram Bhushan Agrawal. Simulated Binary Crossover for Continuous Search Space. *Complex Systems*, 9:115–148, 1995.

[68] 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, June 1989. George Mason University, Morgan Kaufmann Publishers.

[69] D. Dumitrescu, B. Lazzerini, L.C. Jain, and A. Dumitrescu. *Evolutionary Computation*. CRC Press, Boca Raton, Florida, 2000.

[70] B. Dunham, D. Fridshal, and J. H. North. Design by natural selection. *Synthese*, 15:254–259, 1963.

[71] B. Dunham, H. Lewitan, and J. H. North. Introduction of artificial data to enable natural selection scoring mechanism to handle more complex cases. *IBM Technical Disclosure Bulletin*, 17(7):2193, 1974.

[72] B. Dunham, H. Lewitan, and J. H. North. Simultaneous solution of multiple problems by natural selection. *IBM Technical Disclusure Bulletin*, 17(7):2191–2192, 1974.

[73] N. Eldredge and S. J. Gould. Punctuated Equilibria: An Alternative to Phyletic Gradualism. In T. J. M. Schpf, editor, *Models of Paleobiology*, pages 82–15. W. H. Freeman, San Francisco, California, 1972.

[74] Larry J. Eshelman, Richard A. Caruana, and J. David Schaffer. Biases in the Crossover Landscape. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 10–19, San Mateo, California, 1989. Morgan Kaufmann Publishers.

[75] Larry J. Eshelman and J. David Schaffer. Preventing Premature Convergence in Genetic Algorithms by Preventing Incest. In Richard K. Belew and Lashon B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 115–122, San Mateo, California, July 1991. Morgan Kaufmann Publishers.

[76] Larry J. Eshelman and J. Davis Schaffer. Real-coded Genetic Algorithms and Interval-Schemata. In L. Darrell Whitley, editor, *Foundations of Genetic Algorithms 2*, pages 187–202. Morgan Kaufmann Publishers, San Mateo, California, 1993.

[77] Jose Ribeiro Filho. The GAME system (Genetic Algorithms Manipulation Environment). In *IEE Colloquium on Applications of Genetic Algorithms*, pages 2/1–2/4, London, UK, 1994. IEE.

[78] Michael J. Flynn. Very High-Speed Computing Systems. *Proceedings of the IEEE*, 54:1901–1909, 1966.

[79] Terence C. Fogarty. Varying the Probability of Mutation in the Genetic Algorithm. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 104–109, San Mateo, California, 1989. Morgan Kaufmann Publishers.

[80] David B. Fogel. *Evolutionary Computation. Toward a New Philosophy of Machine Intelligence*. The Institute of Electrical and Electronic Engineers, New York, 1995.

[81] David B. Fogel, editor. *Evolutionary Computation. The Fossil Record. Selected Readings on the History of Evolutionary Algorithms*. The Institute of Electrical and Electronic Engineers, New York, 1998.

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

[83] Lawrence J. Fogel. *On the organization of intellect*. PhD thesis, University of California, Los Angeles, California, 1964.

[84] Lawrence J. Fogel. *Artificial Intelligence through Simulated Evolution*. John Wiley, New York, 1966.

[85] Lawrence J. Fogel. *Artificial Intelligence through Simulated Evolution. Forty Years of Evolutionary Programming*. John Wiley & Sons, Inc., New York, 1999.

[86] Lawrence J. Fogel, A. J. Owens, and M. J. Walsh. On the Evolution of Artificial Intelligence. In *Proceedings of the 5th National Sympsium on Human Factors in Engineering*, pages 63–76. IEEE, San Diego, California, 1964.

[87] Lawrence J. Fogel, A. J. Owens, and M. J. Walsh. Artificial Intelligence through a Simulation of Evolution. In M. Maxfield, A. Callahan, and L. J. Fogel, editors, *Biophysics and Cybernetic Systems: Proceedings of the Second Cybernetic Sciences Symposium*, pages 131–155. Spartan Books, Washington, D.C., 1965.

[88] B.R. Fox and M.B. McMahon. Genetic Operators for Sequencing Problems. In Gregory J. E. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 284–300. Morgan Kaufmann, San Mateo, California, 1991.

[89] A. S. Fraser. Simulation of Genetic Systems by Automatic Digital Computers I. Introduction. *Australian Journal of Biological Sciences*, 10:484–491, 1957.

[90] A. S. Fraser. Simulation of Genetic Systems by Automatic Digital Computers II. Effects of Linkage on Rates of Advance Under Selection. *Australian Journal of Biological Sciences*, 10:492–499, 1957.

[91] A. S. Fraser. Simulation of Genetic Systems by Automatic Digital Computers VI. Epistasis. *Australian Journal of Biological Sciences*, 13:150–162, 1960.

[92] Alexander S. Fraser. The evolution of purposive behavior. In H. von Foerster, J. D. White, L. J. Peterson, and J. K. Russell, editors, *Purposive Systems*, pages 15–23. Spartan Books, Washington, D.C., 1968.

[93] Alexander S. Fraser and D. Burnell. *Computer Models in Genetics*. McGraw-Hill, New York, 1970.

[94] R. M. Friedberg. A Learning Machine: Part I. *IBM Journal of Research and Development*, 2(1):2–13, 1958.

[95] R. M. Friedberg, B. Dunham, and J. H. North. A Learning Machine: Part II. *IBM Journal of Research and Development*, 3:282–287, 1959.

[96] G. J. Friedman. Digital simulation of an evolutionary process. *General Systems: Yearbook of the Society for General Systems Research*, 4:171–184, 1959.

[97] George J. Friedman. Selective Feedback Computers for Engineering Synthesis and Nervous System Analogy. Master's thesis, University of California at Los Angeles, February 1956.

[98] C. Fujiki. An evaluation of Holland's genetic operators applied to a program generator. Master's thesis, University of Idaho, Moscow, Idaho, 1986.

[99] Mitsuo Gen and Runwei Cheng. *Genetic Algorithms & Engineering Optimization*. Wiley Series in Engineering Design and Automation. John Wiley & Sons, New York, 2000.

[100] John S. Gero, Sushil J. Louis, and Sourav Kundu. Evolutionary learning of novel grammars for design improvement. *AIEDAM*, 8(3):83–94, 1994.

[101] G. Gibson. *Application of Genetic Algorithms to Visual Interactive Simulation Optimisation*. PhD thesis, University of South Australia, 1995.

[102] Fred Glover and Manuel Laguna. *Tabu Search*. Kluwer Academic Publishers, Norwell Massachusetts, 1998.

[103] David E. Goldberg. Optimal initial population size for binary–coded genetic algorithms. Technical Report 85001, Tuscaloosa: University of Alabama, The Clearinghouse for Genetic Algorithms, 1985.

[104] David E. Goldberg. Simple Genetic Algorithms and the Minimal Deceptive Problem. In Lawrence Davis, editor, *Genetic Algorithms and Simulated Annealing*. Morgan Kaufmann Publishers, San Mateo, California, 1987.

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

[106] David E. Goldberg. Sizing Populations for Serial and Parallel Genetic Algorithms. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 70–79, San Mateo, California, 1989. Morgan Kaufmann Publishers.

[107] David E. Goldberg. A note on Boltzmann tournament selecion for genetic algorithms and population-oriented simulated annealing. *Complex Systems*, 4:445–460, 1990.

[108] David E. Goldberg. Real–Coded Genetic Algorithms, Virtual Alphabets and Blocking. Technical Report 90001, University of Illinois at Urbana-Champaign, Urbana, Illinois, September 1990.

[109] David E. Goldberg, K. Deb, and B. Korb. Messy genetic algorithms revisited: Studies in mixed size and scale. *Complex Systems*, 4:415–444, 1990.

[110] David E. Goldberg and Kalyanmoy Deb. A comparison of selection schemes used in genetic algorithms. In Gregory J. E. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 69–93. Morgan Kaufmann, San Mateo, California, 1991.

[111] David E. Goldberg, Kalyanmoy Deb, and Bradley Korb. Don't Worry, Be Messy. In Richard K. Belew and Lashon B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 24–30, San Mateo, California, July 1991. University of California, San Diego, Morgan Kaufmann Publishers.

[112] David E. Goldberg, B. Korb, and K. Deb. Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems*, 3:493–530, 1989.

[113] David E. Goldberg and Jr. Robert Lingle. Alleles, Loci, and the Traveling Salesman Problem. In John J. Grefenstette, editor, *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, pages 154–159. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1985.

[114] David E. Goldberg and Philip Segrest. Finite Markov Chain Analysis of Genetic Algorithms. In John J. Grefenstette, editor, *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, pages 1–8. Lawrence Erlbaum Associates, Hillsdale, New Jersey, July 1987.

[115] J. Grefenstette, R. Gopal, B. Rosmaita, and D. Gucht. Genetic algorithms for the travelling salesman problem. In John J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and Their Applications*, pages 160–168, Hillsdale, New Jersey, 1985. Lawrence Erlbaum Associates.

[116] John J. Grefenstette. Optimization of Control Parameters for Genetic Algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC–16(1):122–128, Janury/February 1986.

[117] John J. Grefenstette. Deception Considered Harmful. In L. Darrell Whitley, editor, *Foundations of Genetic Algorithms 2*, pages 75–91. Morgan Kaufmann, San Mateo, California, 1993.

[118] Frederic Gruau. *Neural Network Synthesis using Cellular Encoding and the Genetic Algorithm*. PhD thesis, Ecole Normale Supérieure de Lyon, Lyon, France, 1994.

[119] D. Halhal, G. A. Walters, D. Ouazar, and D. A. Savic. Water Network Rehabilitation with a Structured Messy Genetic Algorithm. *Journal of Water Resources Planning and Management, ASCE*, 123(3), May/June 1997.

[120] Jürgen Hesser and Reinhard Männer. Towards an Optimal Mutation Probability for Genetic Algorithms. In H.-P. Schwefel and R. Männer, editors, *Parallel Problem Solving from Nature. 1st Workshop, PPSN I*, pages 23–32. Springer-Verlag. Leture Notes in Computer Science Volume 496, Berlin, 1991.

[121] J. F. Hicklin. Application of the genetic algorithm to automatic program generation. Master's thesis, University of Idaho, Moscow, Idaho, 1986.

[122] W. D. Hillis. Co-evolving parasites improves simulated evolution as an optimization procedure. In C. Langton, C. Taylor, J. Farmer, and S. Rasmussen, editors, *Artificial Life II*, pages 313–324. Addison-Wesley, Reading, Massachusetts, 1992.

[123] W.D. Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. *Physica D*, 42:228–234, 1990.

[124] A. Hoffman. *Arguments on Evolution: A Paleontologist's Perspective*. Oxford University Press, New York, 1989.

[125] John H. Holland. Concerning efficient adaptive systems. In M. C. Yovits, G. T. Jacobi, and G. D. Goldstein, editors, *Self-Organizing Systems—1962*, pages 215–230. Spartan Books, Washington, D.C., 1962.

[126] John H. Holland. Outline for a logical theory of adaptive systems. *Journal of the Association for Computing Machinery*, 9:297–314, 1962.

[127] John H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, Michigan, 1975.

[128] R. B. Hollstien. *Artificial Genetic Adaptation in computer control systems*. PhD thesis, University of Michigan, Ann Harbor, Michigan, 1971.

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

[130] Ellis Horowitz and Sartaj Sahni. *Fundamentals of Computer Algorithms*. Computer Science Press, Rockville, Maryland, 1984.

[131] Reiner Horst and Hoang Tuy. *Global Optimization. Deterministic Approaches*. Springer, Berlin, third edition, 1996.

[132] Evan J. Hughes. Constraint Handling With Uncertain and Noisy Multi-Objective Evolution. In *Proceedings of the Congress on Evolutionary Computation 2001 (CEC'2001)*, volume 2, pages 963–970, Piscataway, New Jersey, May 2001. IEEE Service Center.

[133] K. Hwang and F.A. Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill, New York, 1984.

[134] F. Jiménez, A.F. Gómez-Skarmeta, and G. Sánchez. How Evolutionary Multiobjective Optimization can be used for Goals and Priorities based Optimization. In *Primer Congreso Español de Algoritmos Evolutivos y Bioinspirados (AEB'02)*, pages 460–465. Mérida España, 2002.

[135] Fernando Jiménez and José L. Verdegay. Evolutionary techniques for constrained optimization problems. In *7th European Congress on Intelligent Techniques and Soft Computing (EUFIT'99)*, Aachen, Germany, 1999. Springer-Verlag.

[136] 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.

[137] A. K. De Jong. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*. PhD thesis, University of Michigan, 1975.

[138] Kenneth A. De Jong. Genetic Algorithms are NOT Function Optimizers. In L. Darrell Whitley, editor, *Foundations of Genetic Algorithms 2*, pages 5–17. Morgan Kaufmann Publishers, San Mateo, California, 1993.

[139] Kenneth A. De Jong, William M. Spears, and Diana F. Gordon. Using Markov Chains to Aanalyze GAFOs. In L. Darrell Whitley and Michael D. Vose, editors, *Foundations of Genetic Algorithms 3*, pages 115–137. Morgan Kaufmann Publishers, San Mateo, California, 1994.

[140] Joe L. Blanton Jr. and Roger L. Wainwright. Multiple Vehicle Routing with Time and Capacity Constraints using Genetic Algorithms. In Stephanie Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 452–459, San Mateo, California, 1993. University of Illinois at Urbana-Champaign, Morgan Kauffman Publishers.

[141] Isamu Kajitani, Tsutomu Hoshino, Masaya Iwata, and Tetsuya Higuchi. Variable length chromosome GA for Evolvable Hardware. In *Proceedings of the 1996 IEEE International Conference on Evolutionary Computation*, pages 443–447, Nagoya, Japan, May 1996. Nagoya University, IEEE Service Center.

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

[143] Ralph Michael Kling. *Optimization by Simulated Evolution and its Application to Cell Placement*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1990.

[144] John R. Koza. Hierarchical genetic algorithms operating on populations of computer programs. In N. S. Sridharan, editor, *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pages 768–774. Morgan Kaufmann, San Mateo, California, 1989.

[145] John R. Koza. *Genetic Programming. On the Programming of Computers by Means of Natural Selection*. The MIT Press, Cambridge, Massachusetts, 1992.

[146] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. The MIT Press, Cambridge, Massachusetts, 1994.

[147] John R. Koza, Forrest H. Bennet III, David Andre, and Martin A. Keane. *Genetic Programming III: Darwinian Invention and Problem Solving*. Morgan Kaufmann Publishers, San Francisco, California, 1999.

[148] K. Krishnakumar. Mico-genetic algorithms for stationary and non-stationary function optimization. *SPIE Proceedings: Intelligent Control and Adaptive Systems*, 1196:289–296, 1989.

[149] Ting Kuo and Shu-Yuen Hwang. A Genetic Algorithm with Disruptive Selection. In Stephanie Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 65–69. Morgan Kaufmann Publishers, San Mateo, California, July 1993.

[150] Jean Baptiste Lamarck, editor. *Zoological Philosophy: An Exposition with Regard to the Natural History of Animals*. Chicago University Press, Chicago, 1984. (Publicado originalmente en 1809 en francés, como *Philosophie Zoologique*).

[151] Christopher G. Langdon, editor. *Artificial Life*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

[152] Gideon Langholz, Joan Francioni, and Abraham Kandel. *Elements of Computer Organization*. Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[153] B. R. Levin. Simulation of genetic systems. In N. E. Morton, editor, *Computer Applications in Genetics*, pages 38–48. University of Hawaii Press, Honolulu, 1969.

[154] Andrew J. Mason. Crossover non-linearity ratios and the genetic algorithm : Escaping the blinkers of schema processing and intrinsic parallelism. Technical Report 535b, School of Engineering, University of Auckland, Private Bag 92019 New-Zealand, September 1993.

[155] Keith E. Mathias and L. Darrel Whitley. Transforming the search space with Gray coding. In J. D. Schaffer, editor, *Proceedings of the IEEE International Conference on Evolutionary Computation*, pages 513–518. IEEE Service Center, Piscataway, New Jersey, 1994.

[156] Keith E. Mathias and L. Darrell Whitley. Changing Representations During Search: A Comparative Study of Delta Coding. *Evolutionary Computation*, 2(3):249–278, 1994.

[157] Gregor Johann Mendel. Versuche ber pflanzen-hybriden. In *Verhandlungen des naturforschenden Vereines in Brünn*, volume 1, pages 3–47. Verlag des Vereines, Brünn, Februar–März 1865.

[158] Gregor Johann Mendel. Experiments in Plant Hybridisation. *Journal of the Royal Horticultural Society*, 26:1–32, 1901. (Traducción al inglés de un artículo publicado originalmente en 1865).

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

[160] Zbigniew Michalewicz. A Survey of Constraint Handling Techniques in Evolutionary Computation Methods. In J. R. McDonnell, R. G. Reynolds, and D. B. Fogel, editors, *Proceedings of the 4th Annual Conference on Evolutionary Programming*, pages 135–155. The MIT Press, Cambridge, Massachusetts, 1995.

[161] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, New York, third edition, 1996.

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

[163] Zbigniew Michalewicz and G. Nazhiyath. Genocop III: A co-evolutionary algorithm for numerical optimization with nonlinear constraints. In David B. Fogel, editor, *Proceedings of the Second IEEE International Conference on Evolutionary Computation*, pages 647–651, Piscataway, New Jersey, 1995. IEEE Press.

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

[165] Marvin L. Minsky. Steps toward artificial intelligence. *Proceedings of the IRE*, 49(1):8–30, 1961.

[166] Marvin L. Minsky and O. G. Selfridge. Learning in random nets. In C. Cherry, editor, *Proceedings of the 4th London Symposium on Information Theory*, London, 1961. Butterworths.

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

[168] Melanie Mitchell and Stephanie Forrest. What makes a Problem Hard for a Genetic Algorithm? Some Anomalous Results and Their Explanation. *Machine Learning*, 13:285–319, 1993.

[169] Melanie Mitchell, Stephanie Forrest, and John H. Holland. The Royal Road for Genetic Algorithms: Fitness Landscapes and GA Performance. In Francisco J. Varela and Paul Bourgine, editors, *Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life*. MIT Press, Cambridge, Massachusetts, 1992.

[170] Tom M. Mitchell. The Need for Biases in Learning Generalizations. Technical Report CBM-TR-117, Department of Computer Science, Rutgers University, 1980.

[171] Angel Kuri Morales. *A Comprehensive Approach to Genetic Algorithms in Optimization and Learning. Volume 1 : Foundations*. Centro de Investigación en Computación—Instituto Politécnico Nacional, Mexico City, 1999.

[172] Pedro Larra naga and José A. Lozano, editors. *Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation*. Kluwer Academic Publishers, New York, 2001.

[173] A. Newell, J.C. Shaw, and H.A. Simon. The processes of creative thinking. In H.E. Gruber, G. Terrell, and M. Wertheimer, editors, *Contemporary approaches to creative thinking*, pages 63–119. Atherton Press, New York, 1962.

[174] I.M. Oliver, D.J. Smith, and J.R.C. Holland. A Study of Permutation Crossover Operators on the Traveling Salesman Problem. In John J. Grefenstette, editor, *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, pages 224–230. Lawrence Erlbaum Associates, Hillsdale, New Jersey, July 1987.

[175] C.C. Palmer. *An approach to a problem in network design using genetic algorithms*. PhD thesis, Polytechnic University, Troy, New York, 1994.

[176] I. C. Parmee and G. Purchase. The development of a directed genetic search technique for heavily constrained design spaces. In I. C. Parmee, editor, *Adaptive Computing in Engineering Design and Control-'94*, pages 97–102, Plymouth, UK, 1994. University of Plymouth.

[177] C. Peck and A.P. Dhawan. A Review and Critique of Genetic Algorithm Theories. *Evolutionary Computation*, 3(1):39–80, 1995.

[178] Nicholas J. Radcliffe. Forma Analysis and Random Respectful Recombination. In Richard K. Belew and Lashon B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 222–229. Morgan Kaufmann Publishers, San Mateo, California, July 1991.

[179] S. Ranji Ranjithan, S. Kishan Chetan, and Harish K. Dakshima. Constraint Method-Based Evolutionary Algorithm (CMEA) for Multiobjective Optimization. In Eckart Zitzler, Kalyanmoy Deb, Lothar Thiele, Carlos A. Coello Coello, and David Corne, editors, *First International Conference on Evolutionary Multi-Criterion Optimization*, pages 299–313. Springer-Verlag. Lecture Notes in Computer Science No. 1993, 2001.

[180] Singiresu S. Rao. *Engineering Optimization. Theory and Practice*. John Wiley & Sons, Inc., third edition, 1996.

[181] Tapabrata Ray. Constraint Robust Optimal Design using a Multiobjective Evolutionary Algorithm. In *Proceedings of the Congress on Evolutionary Computation 2002 (CEC'2002)*, volume 1, pages 419–424, Piscataway, New Jersey, May 2002. IEEE Service Center.

[182] Tapabrata Ray, Tai Kang, and Seow Kian Chye. An Evolutionary Algorithm for Constrained Optimization. In Darrell Whitley, David Goldberg, Erick Cantú-Paz, Lee Spector, Ian Parmee, and Hans-Georg Beyer, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'2000)*, pages 771–777, San Francisco, California, 2000. Morgan Kaufmann.

[183] Thomas S. Ray. An approach to the synthesis of life. In C. G. Langton, C. Taylor, J. D. Farmer, and S. Rasmussen, editors, *Artificial Life II*, pages 371–408. Addison-Wesley, Reading, Massachusetts, 1992.

[184] Ingo Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Frommann–Holzboog, Stuttgart, Alemania, 1973.

[185] J. Reed, R. Toombs, and Nils Aall Barricelli. Simulation of biological evolution and machine learning. I. Selection of self-reproducing numeric patterns by data processing machines, effects of hereditary control, mutation type and crossing. *Journal of Theoretical Biology*, 17:319–342, 1967.

[186] Colin B. Reeves, editor. *Modern Heuristic Techniques for Combinatorial Problems*. John Wiley & Sons, Great Britain, 1993.

[187] 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.

[188] 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.

[189] M. M. Rizki and Michael Conrad. Evolve III: A discrete events model of an evolutionary ecosystem. *BioSystems*, 18:121–133, 1985.

[190] George G. Robertson. Population size in classifier systems. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 142–152. Morgan Kaufmann Publishers, 1988.

[191] S. Ronald. *Genetic algorithms and permutation-encoded problems: Diversity preservation and a study of multimodality*. PhD thesis, The University of South Australia, 1995.

[192] S. Ronald. Robust encodings in genetic algorithms. In D. Dasgupta & Z. Michalewicz, editor, *Evolutionaty Algorithms in Engineering Applications*, pages 30–44. Springer-Verlag, 1997.

[193] R. Root. An investigation of evolutionary programming. Master's thesis, New Mexico State University, Las Cruces, New Mexico, 1970.

[194] Franz Rothlauf. *Representations for Genetic and Evolutionary Algorithms*. Physica-Verlag, New York, 2002.

[195] Günter Rudolph. Convergence Analysis of Canonical Genetic Algorithms. *IEEE Transactions on Neural Networks*, 5:96–101, January 1994.

[196] Günter Rudolph and Alexandru Agapie. Convergence properties of some multi-objective evolutionary algorithms. In *Proceedings of the 2000 Conference on Evolutionary Computation*, pages 1010–1016, Piscataway, New Jersey, 2000. IEEE.

[197] Stuart J. Russell and Peter Norvig. *Artificial Intelligence. A Modern Approach*. Prentice Hall, Upper Saddle River, New Jersey, 1995.

[198] J. David Schaffer. *Multiple Objective Optimization with Vector Evaluated Genetic Algorithms*. PhD thesis, Vanderbilt University, 1984.

[199] J. David Schaffer. Multiple Objective Optimization with Vector Evaluated Genetic Algorithms. In *Genetic Algorithms and their Applications: Proceedings of the First International Conference on Genetic Algorithms*, pages 93–100. Lawrence Erlbaum, 1985.

[200] J. David Schaffer, Richard A. Caruana, Larry J. Eshelman, and Raharshi Das. A Study of Control Parameters Affecting Online Performance of Genetic Algorithms for Function Optimization. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 51–60, San Mateo, California, 1989. Morgan Kaufmann Publishers.

[201] J. David Schaffer and Amy Morishima. An Adaptive Crossover Distribution Mechanism for Genetic Algorithms. In John J. Grefenstette, editor, *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, pages 36–40. Lawrence Erlbaum Associates, Hillsdale, New Jersey, July 1987.

[202] Hans-Paul Schwefel. Kybernetische evolution als strategie der experimentellen forschung in der strömungstechnik. Dipl.-Ing. thesis, 1965. (in German).

[203] Hans-Paul Schwefel. *Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie*. Birkhäuser, Basel, Alemania, 1977.

[204] Hans-Paul Schwefel. *Numerical Optimization of Computer Models*. Wiley, Chichester, UK, 1981.

[205] Greg W. Scragg. *Computer Organization. A Top-Down Approach*. McGraw-Hill, New York, 1992.

[206] Anthony V. Sebald and Jennifer Schlenzig. Minimax design of neural net controllers for highly uncertain plants. *IEEE Transactions on Neural Networks*, 5(1):73–82, January 1994.

[207] O. G. Selfridge. Pandemonium: A paradigm for learning. In *Proceedings of the Symposium on Mechanization of Thought Processes*, pages 511–529, Teddington, England, 1958.

[208] S.F. Smith. *A learning system based on genetic adaptive algorithms*. PhD thesis, University of Pittsburgh, 1980.

[209] William M. Spears and Kenneth A. De Jong. An Analysis of Multi-Point Crossover. In Gregory E. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 301–315. Morgan Kaufmann Publishers, San Mateo, California, 1991.

[210] M. Srinivas and L. M. Patnaik. Adaptive probabilities of crossover and mutation in genetic algorithms. *IEEE Transactions on Systems, Man and Cybernetics*, 24(4):656–667, April 1994.

[211] M. Srinivas and L. M. Patnaik. Adaptive Probabilities of Crossover and Mutation in Genetic Algorithms. *IEEE Transactions on Systems, Man and Cybernetics*, 24(4):656–667, April 1994.

[212] Edward J. Steele, Robyn A. Lindley, and Robert V. Blanden. *Lamarck's Signature. How Retrogenes are Changing Darwin's Natural Selection Paradigm*. Perseus Books, Reading, Massachusetts, 1998.

[213] Rainer Storn and Kenneth Price. Differential Evolution: A Simple and Efficient Adaptive Scheme for Global Optimization over Continuous Spaces. Technical Report TR-95-012, International Computer Science Institute, Berkeley, California, March 1995.

[214] Rainer Storn and Kenneth Price. Differential Evolution—A Fast and Efficient Heuristic for Global Optimization over Continuous Spaces. *Journal of Global Optimization*, 11:341–359, 1997.

[215] Patrick D. Surry and Nicholas J. Radcliffe. Real Representations. In Richard K. Belew and Michael D. Vose, editors, *Foundations of Genetic Algorithms 4*, pages 343–363. Morgan Kaufmann, San Mateo, California, 1997.

[216] Patrick D. Surry, Nicholas J. Radcliffe, and Ian D. Boyd. A Multi-Objective Approach to Constrained Optimisation of Gas Supply Networks : The COMOGA Method. In Terence C. Fogarty, editor, *Evolutionary Computing. AISB Workshop. Selected Papers*, Lecture Notes in Computer Science, pages 166–180. Springer-Verlag, Sheffield, U.K., 1995.

[217] 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, 1989. Morgan Kaufmann Publishers.

[218] Gilbert Syswerda. Schedule Optimization using Genetic Algorithms. In Lawrence Davis, editor, *Handbook of Genetic Algorithms*, chapter 21, pages 332–349. Van Nostrand Reinhold, New York, New York, 1991.

[219] Alan Mathison Turing. Computing Machinery and Intelligence. *Mind*, 59:94–101, 1950.

[220] G. Vignaux and Z. Michalewicz. Genetic algorithms for the transportation problem. *Methodologies for Intelligent Systems*, 4:252–259, 1989.

[221] Michael D. Vose. Generalizing the Notion of Schema in Genetic Algorithms. *Artificial Intelligence*, 50(3):385–396, 1991.

[222] Michael D. Vose. *The Simple Genetic Algorithm. Foundations and Theory*. The MIT Press, Cambridge, Massachusetts, 1999.

[223] Hugo De Vries. *Species and Varieties: Their Origin by Mutation*. Open Court, Chicago, 1906.

[224] August Weismann, editor. *The Germ Plasm: A Theory of Heredity*. Scott, London, UK, 1893.

[225] A. Wetzel. *Evaluation of the effectiveness of genetic algorithms in combinatorial optimization*. PhD thesis, University of Pittsburgh, Pittsburgh, Philadelphia, USA, 1983.

[226] D. Whitley and J. Kauth. GENITOR: A different Genetic Algorithm. In *Proceedings of the Rocky Mountain Conference on Artificial Intelligence*, pages 118–130, Denver, Colorado, 1988.

[227] D. Whitley, S. Rana, and R. Heckendorn. Representation Issues in Neighborhood Search and Evolutionary Algorithms. In D. Quagliarella, J. Périaux, C. Poloni, and G. Winter, editors, *Genetic Algorithms and Evolution Strategies in Engineering and Computer Science. Recent Advances and Industrial Applications*, chapter 3, pages 39–57. John Wiley and Sons, West Sussex, England, 1998.

[228] Darrell Whitley. The GENITOR Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials is Best. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 116–121. Morgan Kaufmann Publishers, San Mateo, California, July 1989.

[229] D. H. Wolpert and W. G. Macready. No Free Lunch Theorems for Search. Technical Report SFI-TR-95-02-010, Santa Fe Institute, New Mexico, 1995.

[230] David H. Wolpert and William G. Macready. No Free Lunch Theorems for Optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, April 1997.

[231] Alden H. Wright. Genetic Algorithms for Real Parameter Optimization. In Gregory J. E. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 205–218. Morgan Kaufmann Publishers, San Mateo, California, 1991.

[232] Jin Wu and Shapour Azarm. On a New Constraint Handling Technique for Multi-Objective Genetic Algorithms. In Lee Spector, Erik D. Goodman, Annie Wu, W.B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon, and Edmund Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'2001)*, pages 741–748, San Francisco, California, July 2001. Morgan Kaufmann Publishers.

[233] J. Yoo and P. Hajela. Enhanced GA Based Search Through Immune System Modeling. In *3rd World Congress on Structural and Multidisciplinary Optimization*, Niagara Falls, New York, May 1999.