Capítulo 5

ALGORITMOS EVOLUTIVOS

La historia de los algoritmos que tratan de simular un proceso evolutivo es paralela a la historia de los primeros computadores. Uno de los éxitos de esa época fue un *software* para jugar a las damas, desarrollado por Arthur Samuel en 1959, donde ya hay algunos indicios de comportamiento adaptativo. Sin embargo, es a otros investigadores a los que se les reconocen los méritos de las mejores propuestas, seguramente porque son las más generales, y también por su más amplia divulgación. En Estados Unidos el primero fue John Holland, a quien se le considera el padre de los algoritmos genéticos y su aplicación a animales artificiales que se adaptaban a un entorno artificial (*Animats*). Uno de sus estudiantes, John Koza, diseñó una variante que permitía fabricar programas automáticamente. Luego, de forma independiente, se hicieron otros trabajos también en Estados Unidos, entre ellos los de Lawrence Fogel (programación evolutiva para predecir series de datos temporales), y en Europa destacan los de Rechenberg, Schwefel y Bienert, que desarrollaron las estrategias evolutivas como un método para diseñar perfiles de alas de aviones. De ellos se derivan todos los trabajos posteriores.

También vamos a ver *Simulated Annealing*, que es un algoritmo inspirado en la física pero, curiosamente, muy similar a los algoritmos evolutivos basados en la biología, por lo que se explicará también aquí. Este hecho es otro indicador de que la evolución no solo es una cuestión de animalitos, sino que es un algoritmo general que el universo usa prácticamente para todo.

Todos estos algoritmos cumplen con las cuatro características que debe haber para que haya evolución: población de entes, que se reproducen, con errores en las copias, y sometidos a una presión selectiva. Sin embargo, al-

gunos de ellos nacieron con restricciones producto de la poca capacidad de cómputo en aquella época, como tener poblaciones muy reducidas (a veces de apenas dos entes); o tener poblaciones de tamaño fijo (debido a la imposibilidad de crear matrices dinámicas en los lenguajes de programación de entonces); o usar codificaciones binarias (definitivamente un error); o usar operadores de reproducción equivocados (el estándar de hoy es tener al menos dos: la mutación y el cruce); o aplicar una presión selectiva determinista (elegir siempre a los mejores, lo cual es, definitivamente, un error).

Las cuatro condiciones para que haya evolución son muy precisas, aunque se pueden resumir un poco más debido a que si hay un proceso de copia es inevitable tener una población. Por otro lado, en el universo físico en que vivimos siempre podemos contar con que haya errores, aunque conviene advertir que en universos digitales ello no tiene que ser así: el ruido es prácticamente inexistente, por lo que debe ser introducido a propósito. La presión selectiva sí es importante en cualquier caso, porque si se elimina nos quedamos con un proceso de crecimiento exponencial nada más, donde todo es posible.

En un mundo con recursos finitos, como en el que vivimos, la presión selectiva es fuerte: solo podrán reproducirse los que logren adquirir y usar esos recursos de forma más rápida y eficiente que los demás. Pero en universos digitales no tiene por qué ser así. En un mundo con infinitos recursos, la evolución produce cualquier tipo de entes, pues la presión selectiva es suave (nada está prohibido y se acepta el derroche). Todo es posible, en mayor o menor medida. En un mundo donde todo es posible, nada se puede entender, clasificar ni predecir. La inteligencia no puede existir. No valdría para nada. No tendría ninguna importancia ni utilidad.

Por ello y como resumen, se puede decir que los motores de la evolución son tres: la reproducción —que produce más de lo mismo—, las mutaciones —que producen variedad—, y la presión selectiva —que aumenta la complejidad del sistema—. Y teniendo en cuenta los matices que acabo de señalar, podemos decir que la evolución es un algoritmo que emerge cuando hay reproducción, que a su vez emerge cuando hay suficiente complejidad.

La evolución existe en biología y Darwin lo descubrió, pero no está limitada a ella. Puede ocurrir y ocurre en sistemas mecánicos, hidráulicos, electrónicos, económicos, sociales o artísticos. En particular donde es más fácil implementar el algoritmo de la evolución es con herramientas informáticas. Allí aparecen dos grandes grupos de algoritmos evolutivos, la evolución de objetos y la evolución de programas, que veremos en detalle a continuación.

Cuando los objetos son estructuras de datos:

- **Algoritmos genéticos.** Lo que evoluciona son vectores de números enteros, booleanos, símbolos o cadenas de caracteres.
- **Evolución diferencial.** Una variante del anterior, con operadores de reproducción distintos.
- **Algoritmos genéticos híbridos de Taguchi.** Otra variante del anterior, con un operador de cruce más sofisticado.
- **Estrategias evolutivas.** Lo que evoluciona son vectores de números flotantes.
- **Enfriamiento simulado.** Lo que evoluciona son cualquier tipo de dato o estructura de datos.

Cuando los objetos son programas de computador:

- **Programación genética.** Lo que evoluciona son programas de computador.
- **Gramáticas evolutivas.** Lo que evoluciona son programas de computador.
- **Programación por expresión genética.** Lo que evoluciona son programas de computador.

Y cuando tenemos realizaciones mixtas, donde se pueden interpretar los objetos como datos o como programas:

- **Sistemas clasificadores.** Lo que evoluciona es un conjunto de reglas IF – THEN – ELSE.
- **Programación evolutiva.** Lo que evoluciona son típicamente máquinas de estado, que se pueden considerar estructuras de datos o también reconocedores de lenguajes regulares.

En la práctica no hay por qué mantener estas distinciones históricas. Nada impide mezclar distintos tipos de datos e incluso datos con programas y hacerlos evolucionar a ambos.

Algoritmos genéticos

Los algoritmos genéticos[28] fueron ideados por John Holland en los años 70 y popularizados por uno de sus estudiantes, David Goldberg. Los libros y artículos de Holland son muy simples, y se recomienda mejor leer los de Goldberg, que contienen además varias aplicaciones interesantes. Los algoritmos genéticos se usan principalmente en optimización paramétrica: queremos

28 *Genetic Algorithms.*

encontrar la mejor solución para un problema que se modela con un conjunto de parámetros. Los algoritmos de optimización se pueden clasificar de varias formas:

Clasificación según el tipo de solución:

- Numérico: la solución son *N* parámetros.
- Combinatorio: la solución son *N* parámetros ordenados.

Clasificación según el grado de uso del azar:

- Determinista.
- Estocástico.
- Orientado.

Clasificación según la dirección de búsqueda:

- Primero en profundidad (explotador).
- Primero en anchura (explorador).

Clasificación según los candidatos a solución:

- Simple.
- Múltiple.

Clasificación según use información específica:

- General.
- Heurística.

En este sentido, los algoritmos genéticos se emplean en problemas numéricos. Son orientados, es decir, ni completamente deterministas ni completamente estocásticos. La búsqueda la hacen simultáneamente en profundidad y en anchura, o sea, son exploradores-explotadores. Manejan múltiples candidatos a solución simultáneamente. Y son generales.

Adicionalmente, pueden modificarse ligeramente para resolver problemas combinatorios y también permiten fácilmente incorporar heurísticas, pero entonces se vuelven menos generales. Hay que tener en cuenta (Figura 103) que los algoritmos más generales son menos eficientes y viceversa.

Los algoritmos deterministas no son buenos cuando se enfrentan a problemas multimodales (con varios óptimos, ver figura 104), pues se van a quedar atrapados en el primer óptimo local que encuentren.

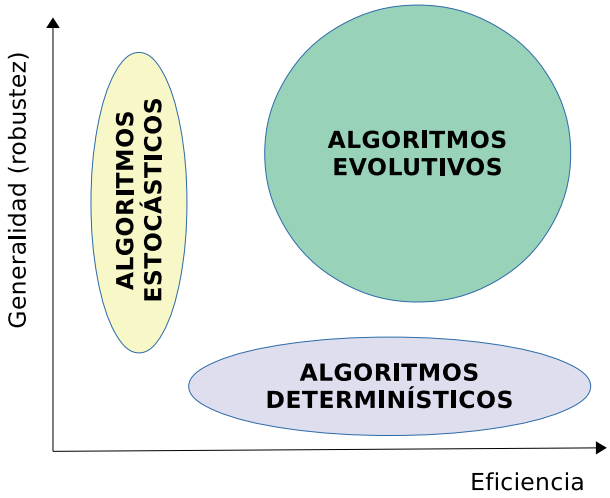

Figura 103. Generalidad versus eficiencia en cualquier tipo de algoritmo de optimización

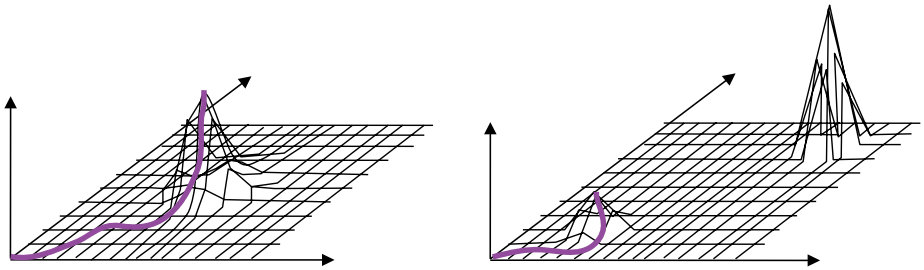

Figura 104. Problema con un solo máximo y con varios máximos

Los algoritmos genéticos exploran varias soluciones simultáneamente y lo hacen de forma no determinista, por lo que suelen ser mejores que otros algoritmos cuando el problema tiene muchos óptimos locales.

Diagrama de flujo de datos

Un algoritmo genético está conformado por una población de cromosomas (típicamente 100, 1000 o más). Cada cromosoma representa una posible solución (buena o mala) a mi problema. Al enfrentar cada cromosoma al problema se obtiene una aptitud que indica lo bueno o malo que es. Después se hace una selección de unos cuantos cromosomas que son los que pasarán al *mating pool* para reproducirse. Los cromosomas hijos resultantes se inyectan a la población. Y todo ello se vuelve a repetir. Cada repetición constituye una generación. El diagrama de flujo de datos puede verse en la figura 105.

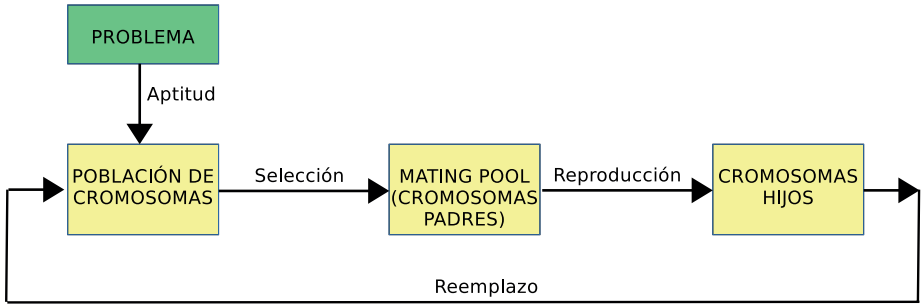

Figura 105. Flujo de datos de todo algoritmo evolutivo

El número de generaciones puede ir desde unas pocas decenas hasta miles de millones o más, dependiendo del tiempo de cómputo disponible y de la calidad de las soluciones encontradas.

A continuación, en los siguientes apartados se explican los detalles del algoritmo.

Población de cromosomas

Se requiere una población de objetos que, en la jerga computacional, se llaman cromosomas (robando términos alegremente a la biología). Cada cromosoma está formado por un conjunto de parámetros (llamados genes). Todos los cromosomas tienen los mismos genes y en el mismo orden. Cada gen puede tomar distintos valores llamados alelos. En general, los alelos que toman los genes de un cromosoma diferirán de los alelos de otros cromosomas en el mismo gen. Y los genes tienen un significado posicional, es decir, si los cromosomas son vectores de 35 elementos, existirá el gen 0, el gen 1, y así hasta el gen 34. Por otro lado, cada gen puede tener un conjunto de alelos distinto al de otro gen.

Se suele diferenciar el cromosoma del objeto que representa: el cromosoma es un conjunto de parámetros que está en el espacio de los genotipos, mientras que el objeto es físico, tiene forma, está en el espacio de los fenotipos. En la figura 106 destacamos como ejemplos un tucán, un circuito electrónico y un programa de computador. La traducción de un cromosoma a su objeto físico se llama expresión.

A cada cromosoma se le calcula su aptitud para saber lo bueno o lo malo que es resolviendo un problema. Cada cromosoma codifica, de alguna manera, una solución posible al problema (puede ser una solución buena, mala o malísima, pero solución a fin de cuentas).

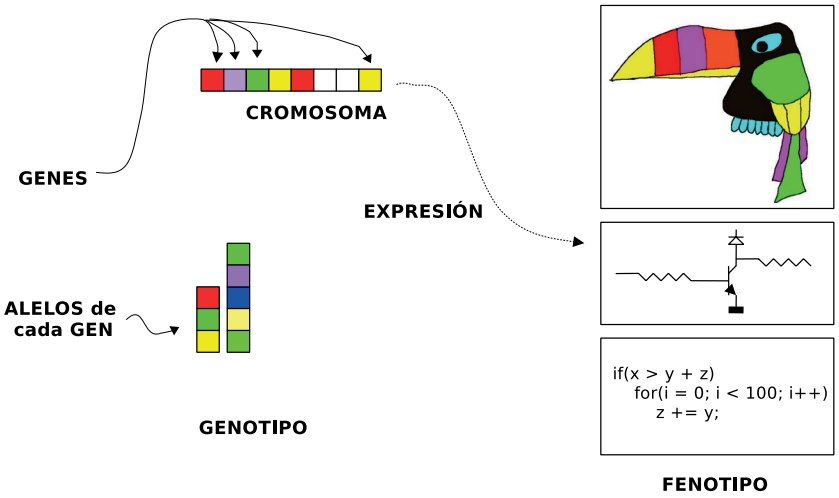

Figura 106. El fenotipo se expresa, creando el genotipo

Esto es lo importante: cada cromosoma contiene información de un objeto que representa la solución a mi problema. Por ejemplo, si quiero diseñar un automóvil que sea veloz, que consuma poco, que sea seguro y que sea fácil de aparcar, cada cromosoma debe describir un auto de entre todos los autos posibles. La forma de codificar el objeto es muy variada. En el caso de los algoritmos genéticos se logra con un vector de parámetros (cilindrada del motor, radio de las ruedas, tipo de combustible), pero nada impide usar una estructura de datos más elaborada (listas, matrices, árboles).

Habitualmente se emplean algoritmos genéticos cuando el espacio de búsqueda es enorme (por ejemplo, hay trillones de formas de fabricar un automóvil), mientras que la población de cromosomas suele tener un tamaño limitado (dependiendo de la capacidad de cómputo disponible), pero es razonable tener apenas 100, 1000 o 10 000 cromosomas. Los cromosomas se generan inicialmente al azar, de modo que es de esperar que al principio implementen soluciones muy malas (un auto con ruedas gigantes pero con motor muy pequeño, o cosas más disparatadas).

Es muy importante que la codificación de objetos en cromosomas cumpla con dos propiedades:

- **Completitud:** todo objeto factible debe tener un cromosoma que lo represente.
- **Cerradura:** todo cromosoma debe representar un objeto factible.

En la figura 107 se muestran dos contraejemplos.

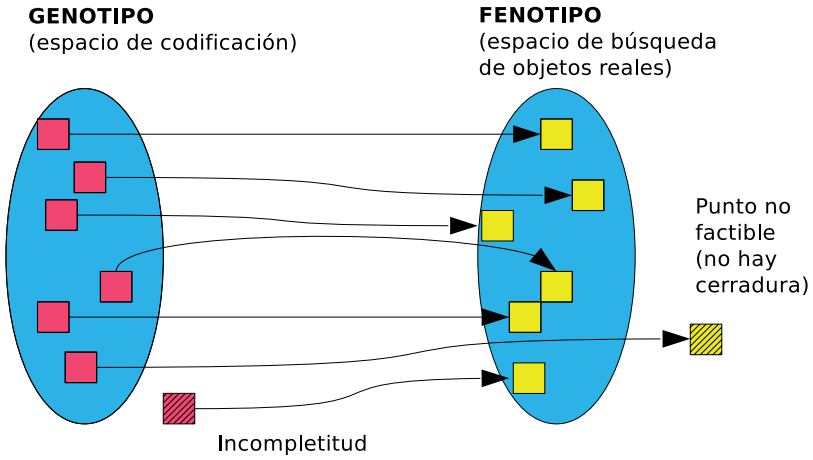

Figura 107. Ejemplo de incompletitud y de no cerradura

Entonces como decíamos, hay que disponer de una población de estos cromosomas. ¿Cuántos? Cuantos más, mejor. Sin embargo, dependemos de las limitaciones de memoria de nuestro equipo de cómputo de modo que podemos pensar en un número entre 100 y 10 000. Inicialmente rellenaremos cada gen de estos cromosomas con cualquier alelo válido, elegido al azar (Figura 108).

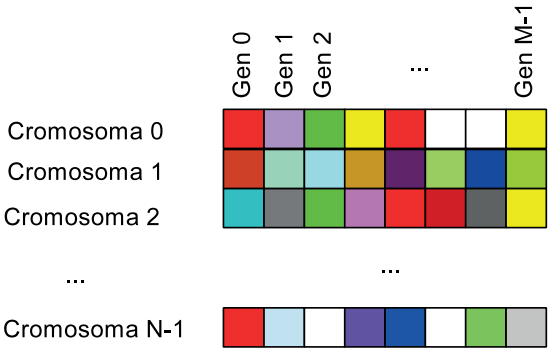

Figura 108. Población de N cromosomas con M genes cada uno inicializada al azar

Cálculo de la aptitud

Cada cromosoma se expresa en el objeto que representa y luego se le enfrenta con el problema que tratamos de resolver. En el ejemplo anterior, debemos tomar cada cromosoma y fabricar el automóvil que representa o, al menos, construir una simulación computacional de este. Y el problema a resolver puede ser un circuito de pruebas, es decir, una carretera con rectas, curvas y obstáculos, bien sea real o simulada. Esto último es más barato,

en materiales y en tiempo. La única desventaja de las simulaciones es que pueden perderse detalles menores que sean importantes. Por ello es común comenzar con simulaciones y después expresar los mejores cromosomas en objetos reales, para continuar la evolución allí.

Al enfrentar el objeto así construido al problema, debemos calcular su aptitud, que es un número que indica lo bien o mal que se desempeñó. Las aptitudes se suelen mantener normalizadas (por ejemplo, entre 0% y 100%) de modo que si el auto circula por la pista de pruebas pero se choca o se sale en las curvas o no llega al destino le asignaré 0% y conforme vea que avanza por la carretera, le iré asignando una aptitud cada vez mayor, hasta llegar al 100% si lo hace muy bien y cumple con todos mis requerimientos de velocidad, consumo y seguridad.

No se requiere que la aptitud sea una función, matemáticamente hablando, y mucho menos que sea continua, derivable o repetible. Es simplemente un indicador de lo buena o mala que es una solución. Solo se le exige que sea monótona, en el sentido de que soluciones mejores tengan aptitudes mejores. Pero incluso si esto no se logra del todo, el algoritmo genético funcionará también, aunque irá degradando la calidad de las soluciones.

Muchas veces se llama función de evaluación a la función de aptitud cuando está sin normalizar. Pero ello no tiene mayor importancia. Dependiendo del método de selección que utilicemos, que vamos a ver a continuación, se requerirá que la aptitud esté normalizada. Pero no siempre es necesario hacerlo. El proceso de normalización se lleva a cabo siguiendo el sentido común, es decir, buscando (teórica o prácticamente) el valor máximo y dividiendo cada aptitud de cada cromosoma por este valor. A veces no hay un valor máximo teórico, pero siempre podremos obtener el mayor valor de todos los cromosomas existentes en un momento dado. La normalización en ocasiones se hace de forma no lineal, como exponencial o logarítmica, aunque ello rara vez aportar alguna ventaja. Más adelante veremos que sí hay una forma de normalizar, llamada *ranking*, que tiene muchas ventajas prácticas.

En la figura 109 observamos un ejemplo sencillo de espacio de búsqueda unidimensional con una población de 3 cromosomas generados al azar. Aunque hemos dibujado la aptitud como una función continua, para darnos una idea de cómo es el espacio de búsqueda, lo cierto es que solo conocemos los 3 puntos calculados para los cromosomas.

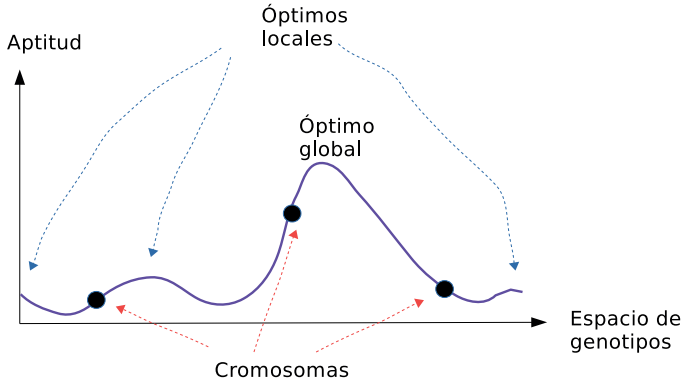

Figura 109. Ejemplo de espacio de búsqueda, con 3 cromosomas

Selección

A continuación se hace un proceso de selección que consiste en elegir a los *K* mejores cromosomas con mayor probabilidad. Eso significa que los peores tienen menos probabilidad, pero no se les descarta de antemano. Es importante que la selección no sea determinista (como los algoritmos de escalada, que eligen siempre las mejores soluciones) porque en caso contrario se corre más riesgo de quedar atrapado en óptimos locales, como veíamos en figura 104.

Hay varios algoritmos para hacer la selección. Los más usados son:

Sorteo. La probabilidad de ser seleccionado es directamente proporcional a la aptitud. Es como si en un sorteo de lotería se repartieran 100 billetes entre los cromosomas concursantes. Hay un cromosoma de aptitud 3 al que se le asignan 3 billetes, otro de aptitud 25 al que se le asignan 25 billetes y otro de aptitud 72 al que se le asignan 72 billetes. Después se juega la lotería, seleccionando al azar uno de los 100 billetes. Está claro que con este esquema los cromosomas de mayor aptitud tienen ventaja, pero los cromosomas de menor aptitud también pueden ser seleccionados eventualmente.

Entonces, para cada uno de los *N* cromosomas con aptitud u_i se calcula su probabilidad de ser seleccionado, así:

$$p_i = \frac{u}{\sum_{i=1}^{N} u_i} \quad \forall i=1,...N \qquad \text{Ec. 28}$$

Se calculan luego las probabilidades acumuladas:

$$q_0=0$$
$$q_i=p_1 + ... + p_i \quad \forall i=1,...N \qquad \text{Ec. 29}$$

Por último, se generan K números aleatorios r_j y se seleccionan los K cromosomas tales que:

$$q_{i-1} < r_j \leq q_i \quad \forall j=1,...K \qquad \text{Ec. 30}$$

Pongamos un ejemplo, con una población de N=8 cromosomas y un *mating pool* de K=3 cromosomas.

Tabla 5. Ejemplo de cromosomas

Cromosoma	Evaluación e_i	Aptitud normalizada (%) u_i	Probabilidad de ser seleccionado p_i	Probabilidad acumulada q_i
C1	40.5	15	0.15	0.15
C2	13.5	5	0.05	0.2
C3	27	10	0.1	0.3
C4	40.5	15	0.15	0.45
C5	81	30	0.3	0.75
C6	27	10	0.1	0.85
C7	13.5	5	0.05	0.9
C8	27	10	0.1	1

Al evaluar los cromosomas, la suma de todas sus evaluaciones da 270. Entonces, dividiendo cada evaluación por 270 y multiplicando por 100, salen sus respectivas aptitudes en porcentajes. El cálculo de probabilidades de selección y probabilidades acumuladas es directo siguiendo las fórmulas anteriores y sus resultados pueden verse en la figura 110.

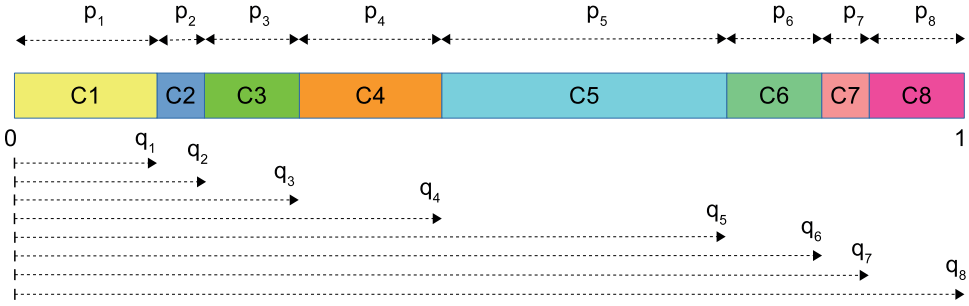

Figura 110. Ejemplo de cálculo de probabilidades de selección y probabilidades acumuladas

Si, por ejemplo, al generar los 3 números aleatorios salen 0.58, 0.14 y 0.69, entonces los cromosomas seleccionados serán C5, C1 y C5 (Figura 111).

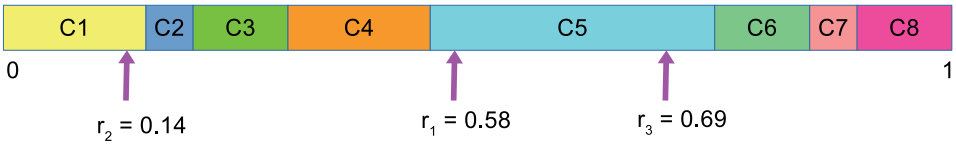

Figura 111. Ejemplo de selección de 3 cromosomas por sorteo

Ruleta. Es igual al anterior, con la única diferencia de que no hace falta generar *K* números aleatorios (recordemos que ello es costoso computacionalmente), sino que basta generar uno solo (*r*). Si pensamos en la regla anterior como si fuera una ruleta circular, los demás números se sitúan de forma equiespaciada, a partir del número aleatorio generado.

$$r_j = \frac{r+j-1}{k} \quad \forall j=1,\ldots k \qquad \text{Ec. 31}$$

Continuando con el mismo ejemplo, si el número aleatorio sale r=108 grados, como son 3 cromosomas a seleccionar entonces hay que sumar 360/3=120 grados para ir obteniendo los demás, con lo cual saldrán r1=108, r2=228 y r3=348 grados (Figura 112). Ello implicaría que se seleccionan los cromosomas C4, C5 y C8. En la práctica no hace falta usar un círculo y grados sino que podemos continuar en el segmento real [0,1] y hacer las operaciones de suma módulo 1.

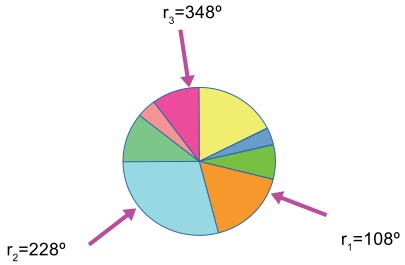

Figura 112. Ejemplo de selección de 3 cromosomas por ruleta

Muestreo universal por restos. Cada cromosoma *i* se selecciona directamente *floor*(K^*p_i) veces. Las vacantes que queden hasta completar las *K* muestras se reparten por el método del sorteo o de la ruleta. Este sistema garantiza no perder a los mejores cromosomas. De hecho, garantiza una distribución de los nuevos cromosomas proporcional a su aptitud, sin intervención del azar.

Volviendo al ejemplo anterior, con K=3 no se seleccionaría ningún cromosoma directamente y los tres habría que elegirlos por sorteo o ruleta. Para hacer el ejemplo más didáctico, supongamos que hay que seleccionar

K=7 cromosomas. Entonces, en la tabla 6 podemos ver que saldrían directamente seleccionados un cromosoma C1, un C4 y dos C5, que suman cuatro cromosomas. Los otros tres, para completar *K=7*, se seleccionarán usando cualquiera de los otros métodos.

Tabla 6. Ejemplo de muestreo universal por restos con K=7

Cromosoma	Evaluación e_i	Aptitud normalizada (%) u_i	Probabilidad de ser seleccionado p_i	$K*p_i$	floor($K*p_i$)
C1	40.5	15	0.15	1.05	1
C2	13.5	5	0.05	0.35	0
C3	27	10	0.1	0.7	0
C4	40.5	15	0.15	1.05	1
C5	81	30	0.3	2.1	2
C6	27	10	0.1	0.7	0
C7	13.5	5	0.05	0.35	0
C8	27	10	0.1	0.7	0

Torneo. Es el sistema más rápido. Se toman al azar 2 o 3 cromosomas de la población, y se selecciona el que tenga mayor función de aptitud. Se repite este procedimiento *K* veces.

A pesar de que este método tiene malas propiedades matemáticas (por ejemplo, no garantiza que los mejores sean seleccionados nunca), en la práctica ello no suele ser un problema. Como el número de generaciones suele ser muy alto, la probabilidad de que nunca seleccione a los mejores es muy baja.

Otra ventaja es que no se requiere normalizar la aptitud.

Y una tercera ventaja es que ni siquiera se requiere calcular la aptitud de toda la población. Basta con ir calculando la de los cromosomas que salen elegidos para hacer el torneo.

Todo ello hace que sea el sistema de selección más simple y efectivo para implementarse en *software*.

Reproducción

Ahora hay que reproducir los cromosomas seleccionados que se encuentran en el *mating pool*, para generar los cromosomas hijos. Existen muchos operadores de reproducción y podemos inventar muchos más. Nos detendremos en los más básicos y comunes.

Mutación. Se elige al azar un cromosoma del *mating pool*, se elige al azar uno de sus genes y se cambia su alelo por otro generado al azar (Figura 113).

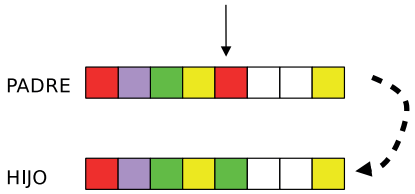

Figura 113. Mutación

Existen otros operadores de mutación y podemos inventar más, pero lo importante es que cumplan con una propiedad: al aplicar este operador infinitas veces sobre cualquier cromosoma deben generarse todos los cromosomas posibles. Con ello nos aseguramos de que, en el caso peor, esté realizando una búsqueda exhaustiva probabilista. Y, desde luego, que genere variedad en la población, recuperando alelos que podrían haberse perdido con los procesos de selección y de cruce. Habitualmente la mutación produce un cambio pequeño y ello significa que explota las soluciones encontradas hasta el momento en el espacio de búsqueda, es decir, hace una búsqueda en profundidad.

Cruce uniforme. También se llama recombinación uniforme. Se necesitan dos padres, que generarán un hijo. Cada gen del hijo se selecciona al azar entre el correspondiente gen de los padres (Figura 114).

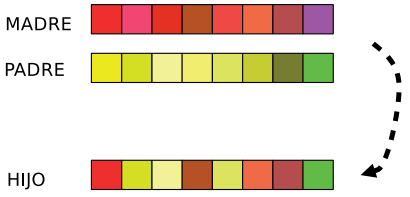

Figura 114. Cruce uniforme

Nada impide obtener de una vez otro hijo, con los genes complementarios, no usados, de los padres.

La misión del cruce es menos crítica que la mutación, pero el objetivo es realizar cambios grandes para explorar el espacio de búsqueda, usando a los padres (que si han sido seleccionados es probable que sean buenos) como base para buscar combinaciones mejores, es decir, hace una búsqueda en anchura.

En la figura 115 vemos algunos ejemplos de generación de hijos (puntos azules): el cromosoma *A* lo hizo por mutación, al igual que el cromosoma *B*. Sus hijos son muy parecidos a los respectivos padres, pues los cambios son pequeños, por lo que están haciendo búsqueda local, en profundidad (explotación de soluciones conocidas). *A* su vez, *B* y *C* se cruzaron generan-

do un hijo que se parece algo a *B* y algo a *C*, porque está entremedias de los dos. Los cambios son mayores y hace búsqueda en anchura (exploración de soluciones desconocidas).

La mutación de *A* produce un hijo con mejor aptitud, lo cual implica que tiene una gran probabilidad de ser seleccionado para futuras generaciones. La mutación de *B* empeoró su aptitud y, por tato, es menos probable que salga seleccionado. Este es el sustento de los algoritmos genéticos. Los operadores de reproducción generan variedad mientras que la selección presiona en la búsqueda de los mejores, de los más aptos para solucionar mi problema.

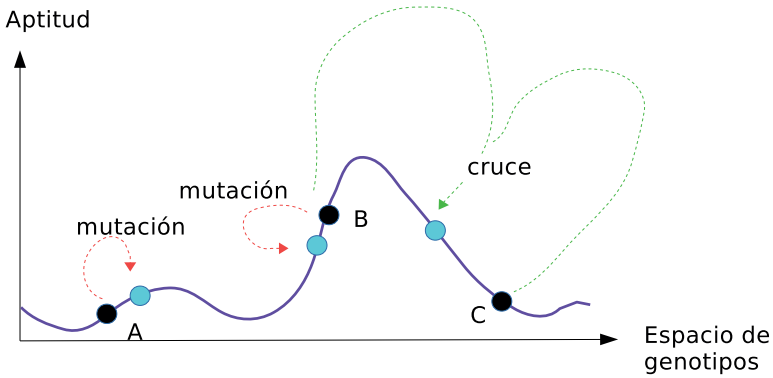

Figura 115. Ejemplos de mutación y cruce

Los operadores de reproducción se pueden aplicar en paralelo sobre el *mating pool* de forma porcentual (un porcentaje del *mating pool* se muta y otro porcentaje se cruza) o probabilista (para cada cromosoma del *mating pool* hay una cierta probabilidad de que participe en una mutación o en un cruce). Pero también se pueden aplicar secuencialmente, o sea, primero se cruzan dos cromosomas y el resultado se muta. En general, los detalles de implementación son irrelevantes, salvo uno (el mencionado en la mutación): siempre debe haber algún operador que produzca pequeños cambios, de modo que se generen todos los cromosomas posibles si se deja pasar suficiente tiempo.

En algunos libros los porcentajes o probabilidades se expresan relativos al número de cromosomas o de genes de la población o del *mating pool*. Todo ello es irrelevante, aunque suele confundir a quien se inicia programando en estos algoritmos. Lo que sí suele ser sano es correr el algoritmo genético dos veces (o más): una con mucha mutación y poco cruce, y otra con poca mutación y mucho cruce. Obviamente, al final se elige la mejor solución de entre todas las generadas.

Hay trabajos donde se ha tratado de caracterizar los porcentajes de mutación y cruce más adecuados, pero ello en general es muy dependiente del problema, por lo que más vale hacer las dos pruebas mencionadas. Hay otros trabajos donde se incorporan los porcentajes de mutación y de cruce como genes adicionales en los cromosomas, tratando de que la misma evolución encuentre los valores más adecuados. Esta es una buena idea, aunque complica un tanto el algoritmo.

Reemplazo

Finalmente hay que inyectar los hijos en la población. Aunque es una operación sencilla, hay varias formas de hacerlo.

En los inicios del uso de estas técnicas se trataba de mantener la población constante, de modo que si se inyectaban digamos 15 hijos, había que eliminar 15 cromosomas en la población original. Imagino que se hacía así porque las matrices de los lenguajes de programación de la época eran estáticas, de tamaño fijo. Hoy día se puede simplemente añadir los hijos a la población.

De todos modos, si se desea eliminar cromosomas de la población (para evitar consumir mucha memoria, que conlleva a la vez a mayores tiempos de cómputo) se pueden seguir estas estrategias:

- **Eliminar cromosomas al azar.** Es lo más rápido y recomendable.
- **Eliminar a los peores.** Es razonable, pero se gasta tiempo ordenando los cromosomas por aptitud.
- **Eliminar a los peores con mayor probabilidad.** Es muy razonable. También se llama proceso de selección inversa. Esto es lo que ocurre en biología. Hay un proceso de selección para reproducción y hay otro (selección inversa) para sobrevivir. Si se emplea selección inversa por torneo, es un proceso rápido.
- **Asignar a cada cromosoma una edad.** Debe ser igual a cero en el momento de ser creado, y debe incrementarse en uno con cada generación que pase. Después hay que eliminar los cromosomas según vayan superando un cierto valor, por ejemplo 100.

En cualquiera de los casos anteriores, conviene no eliminar al mejor de los cromosomas. A esto se le llama elitismo, y es conveniente usarlo porque en cualquier caso el mejor no conviene perderlo. Tampoco es bueno que haya mucho elitismo (mantener más de uno o dos de los mejores cromosomas) porque eso perjudica la exploración de nuevas zonas en el espacio de búsqueda.

Hay también otra forma de simplificar el algoritmo genético que consiste en que el *mating* pool tiene un tamaño K=1. En cada generación solo se produce un hijo y se elimina un padre. A esta variante se la llama de "estado estacionario", porque casi no cambia la población de una generación a la siguiente. Es la más rápida y cómoda de implementar en *software* porque nos quitamos de encima la duda de cuál es el tamaño óptimo del mating pool.

Al respecto, téngase en cuenta que, *grosso modo*, da igual tener una población de 200 cromosomas que se ejecuten por 3000 generaciones a usar 300 cromosomas por 2000 generaciones. En ambos casos, se efectuarán aproximadamente 600 000 evaluaciones de aptitud, o sea, se van a explorar 600 000 puntos del espacio de búsqueda. De modo que con la variante de estado estacionario debemos aumentar el número de generaciones para compensar que en cada generación solo se evalúan dos o tres de cromosomas (los que producen el hijo que va a llegar al *mating pool*). Por ejemplo, 150 000 generaciones y una población de cualquier tamaño de donde se seleccionen al azar 4 cromosomas que compitan entre sí por torneo para generar dos cromosomas padres en el *mating pool*. El resultado es aproximadamente el mismo y se evitan ineficiencias (por ejemplo, con el algoritmo genético básico podrían llegar al *mating pool* bastantes cromosomas que luego no produzcan ningún hijo, de modo que se perdió tiempo evaluándolos y seleccionándolos). Este será el algoritmo genético que implementaremos al final de este capítulo.

Ejemplo 1: salas de cine

Heredé de un tío una cadena de 4 salas de cine, pero no entiendo nada del negocio. Mi objetivo es maximizar los ingresos. Y tengo varias preguntas cuya respuesta desconozco:

- ¿El precio de la entrada debería ser 2000, 3000, 5000, 8000 o 12 000?, ya que eso es lo que he visto que cobran otros cines.
- ¿Debería haber películas comerciales, subtituladas o en blanco y negro?
- ¿El horario debería ser en la mañana, en la tarde o en la noche?

En este caso, cada sala es un cromosoma, y cada una de mis preguntas es un gen. Con ello me aseguro que puedo implementar salas con cualquiera de las características enunciadas. Mi población es muy reducida, pues solo tiene 4 cromosomas. Esto no es lo habitual, pero no tengo elección pues es una limitación física del problema. Como hemos dicho, cada cromosoma tiene 3 genes con los alelos que se muestran en la tabla 7.

Tabla 7. Genes y alelos del ejemplo de las salas de cine

Gen	Alelos
Gen 0: Precio	2000, 3000, 5000, 8000, 12 000
Gen 1: Calidad	Comercial, Subtitulada, Blanco y Negro
Gen 2: Horario	Mañana, Tarde, Noche

Una vez definido el cromosoma, debo generar al azar la población inicial, programar de esa manera las salas y esperar un día a ver cuánto recaudo en cada una. Imaginemos, por ejemplo, que es como dice la tabla 8.

Tabla 8. Población inicial del ejemplo de las salas de cine y aptitudes resultantes

Sala	Cromosoma	Genes			Dinero recaudado	Aptitud normalizada
		Precio	Calidad	Horario		
S1	C1	3000	Comercial	Mañana	70 000	26.3%
S2	C2	3000	Comercial	Tarde	150 000	56.4%
S3	C3	5000	Subtitulada	Noche	40 000	15.0%
S4	C4	2000	Blanco y negro	Noche	6000	2.3%

Vistos los resultados, he obtenido las mayores ganancias en la sala 2. De este modo, podría limitarme a explotar esa combinación, programando todas las salas con entradas a 3000, películas comerciales y horario en la tarde. Aparentemente, explotando esta solución conocida obtendré ganancias muy altas, pero ¿no será que existen otras combinaciones que me pueden dar más dinero? Es decir, ¿me conviene explorar más? Para quien sepa de negocios, bajar la entrada a 2000 y poner horario nocturno podría darme todavía más ganancias, pero yo no sé nada de eso. Por lo tanto, me toca poner en marcha un algoritmo genético. El proceso que viene ahora es hacer una selección de los mejores con mayor probabilidad.

Supongamos entonces que hacemos selección por torneo de dos padres, con el algoritmo de estado estacionario, para generar un único hijo resultante de cruzar los dos padres y mutar el resultado, inyectando este hijo a la población y eliminando el cromosoma peor.

Entonces elegimos al azar dos cromosomas (supongamos que salen C2 y C3) y por torneo nos quedamos con el mejor (C2). Ahora elegimos al azar otros dos padres (por ejemplo, salieron C4 y C4) con lo cual el mejor es C4. Ya tenemos los dos padres (C2 y C4). Para generar el hijo (llamémoslo C5) por cruce uniforme seleccionamos al azar los respectivos genes de la madre y del padre, por ejemplo {padre, padre, madre}, con lo cual el hijo sale con genes {3000, Comercial, Noche}. Ahora vamos a mutarlo para lo cual elegi-

mos al azar un gen (supongamos que salió el de "precio") y elegimos al azar uno de los alelos (supongamos que salió 8000). Por último, inyectamos este cromosoma C5 a la población, eliminando el peor que hay hasta el momento, que es C4. Con ello nos queda la población indicada en la tabla 9, después de transcurrir la primera generación. Además, dejamos pasar un día para ver cuánto recauda cada sala y calculamos las nuevas aptitudes.

Se observa que en este problema hay inevitablemente ruido en la evaluación: no tenemos garantías de recaudar lo mismo si programamos la misma película en idénticas condiciones. Lo que sí podemos observar es una serie de preferencias del público, y eso es lo importante para el cálculo de la aptitud.

Tabla 9. Generación 1 del ejemplo de las salas de cine y nuevo cálculo de la aptitud

Sala	Cromosoma	Genes			Dinero recaudado	Aptitud normalizada
		Precio	Calidad	Horario		
S1	C1	3000	Comercial	Mañana	60 000	14.3%
S2	C2	3000	Comercial	Tarde	180 000	42.9%
S3	C3	5000	Subtitulada	Noche	20 000	4.7%
S4	C5	8000	Comercial	Noche	160 000	38.1%

Y así repetiríamos generación tras generación, lo cual nos lleva a una pregunta: ¿cuándo detenerse? La respuesta es muy obvia: cuando tengamos una solución aceptable, cuando veamos que no aparecen mejores soluciones (es decir, el algoritmo ha convergido al óptimo global, o se ha quedado atascado en un óptimo local) o cuando ya no nos quede más tiempo disponible.

Ejemplo 2: invertir una matriz

Supongamos que no sabemos nada de álgebra y queremos invertir una matriz A, es decir, calcular una matriz A^{-1} tal que $A * A^{-1} = I$ siendo I la matriz identidad (todas los elementos de la diagonal principal igual a 1, y 0 en el resto). Por ejemplo:

$$A = \begin{pmatrix} 4 & -1 & 2 \\ 3 & 5 & -1 \\ 2 & -2 & 1 \end{pmatrix} \qquad \text{Ec. 32}$$

Queremos calcular:

$$A^{-1} = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \qquad \text{Ec. 33}$$

Entonces el cromosoma será un vector de 9 genes con todos sus alelos números flotantes (Figura 116).

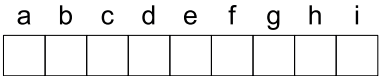

Figura 116. Cromosoma para invertir una matriz de 3x3

Y la función de aptitud a minimizar será:

$$error = |A \times A^{-1} - I| \qquad \text{Ec. 34}$$

Se puede, entonces, tomar como aptitud el error cambiado de signo. De este modo, al maximizar la aptitud se minimizará el error.

$$aptitud = -error = -|A \times A^{-1} - I| \qquad \text{Ec. 35}$$

El resto es ya conocido.

Problemas que pueden aparecer

Los algoritmos genéticos son fantásticos, muy sencillos, potentes y generales. Pero no están exentos de problemas, que veremos a continuación junto a las posibles soluciones.

Como hemos visto en los ejemplos, solo hay dos cosas difíciles cuando se quiere aplicar un algoritmo genético a la solución de un problema de optimización:

- Diseñar el cromosoma.
- Definir cómo se va a calcular la aptitud.

El resto es bastante automático y general. Solo estas dos partes del algoritmo dependen del problema concreto que se quiera solucionar y a ellas debe gastarse suficiente tiempo pensando en cuáles van a ser las mejores alternativas.

Diseñar el cromosoma. Recordemos que es importante lograr la completitud y la cerradura en el diseño del cromosoma (Figura 107). Sin completitud quizás no alcancemos buenas soluciones porque no existe la forma de codificarlas en los cromosomas. Sin cerradura quizás el algoritmo genético gaste mucho tiempo generando soluciones inválidas (también llamadas puntos no factibles) cuya aptitud es cero por lo que, si son muchas, no producirán ninguna presión evolutiva en busca de mejores soluciones.

La completitud suele ser más fácil de lograr (y de detectar si no se cumple), y se soluciona añadiendo más genes que codifiquen adecuadamente todo el espacio de soluciones posibles. Por contra, la cerradura suele ser difícil de lograr porque habitualmente lo que significa es que hay restricciones

duras en el enunciado del problema. En caso de no lograrse, se suelen aplicar técnicas de penalización, que veremos enseguida.

Sin embargo, incluso aunque cumplamos con la completitud y la cerradura, puede haber codificaciones de cromosomas francamente malas. El algoritmo genético funcionará, pero requerirá mucho más tiempo para encontrar soluciones. Veamos un ejemplo usando el conocido problema computacional de las N-Damas: el objetivo es situar *N* damas de ajedrez en un tablero de *NxN* casillas de modo que ninguna ataque a otra. Se sabe que es un problema *NP* y que existen soluciones para todo $N \geq 4$. En la figura 117 podemos ver algunas soluciones para $N=4$, para $N=5$ y para $N=6$.

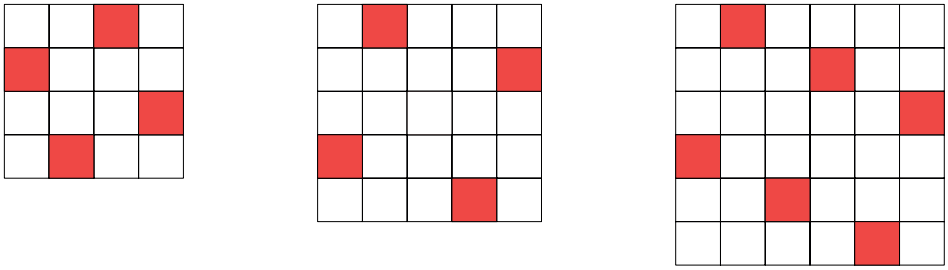

Figura 117. Solución a las N-Damas para N={4, 5, 6}

El cromosoma que rápidamente se nos ocurre lo podemos ver en la figura 118 para el problema de las 4-damas: consiste en estirar el tablero de 4x4 para formar un vector de 16 casillas, donde todos los alelos son binarios e indican si existe o no existe una dama en esa casilla.

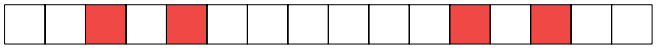

Figura 118. Cromosoma ingenuo, para las 4-damas

Dado que hay 16 genes binarios, el espacio de búsqueda es de 2^{16} posibilidades. No parece muy grande, pero para el caso general de N-Damas es de 2^{NxN}, lo cual es prohibitivo. Obviamente el problema es NP, de modo que sabemos que el espacio de soluciones crecerá exponencialmente, pero ¿será que el exponente de la exponencial se puede reducir? La respuesta es afirmativa si pensamos en la codificación de la figura 119, donde ahora se necesita un cromosoma de N genes únicamente. El gen va a indicar la columna donde se encuentra la dama, y el alelo del gen va a indicar la fila. O sea, hay una dama en la columna 0, fila 1; otra en la columna 1, fila 3; otra en la columna 2, fila 0; y otra en la columna 3 fila 2.

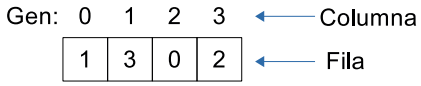

Figura 119. Un cromosoma mejor pensado, para el problema de las 4-damas

Cada gen tiene N alelos, y hay N genes, por lo que el espacio de búsqueda se reduce a N^N (lo que es igual a $2^{Nxlog2(N)}$ que es menor que 2^{NxN} de la primera codificación).

Además, en la primera codificación, los operadores de mutación y cruce pueden añadir o quitar damas, de modo que no sean N, con lo cual estaremos explorando soluciones no factibles y es más probable que el algoritmo genético se atasque. Mientras que en la segunda codificación todas las soluciones son factibles y, como añadido, es imposible que tengan cruces en las columnas. Es decir, en cada columna solo hay una dama por lo que no hace falta verificar ataques en la misma columna. Lo único que resta por verificar son los ataques en las filas y en las diagonales.

Todavía se puede hacer una mejora, en este caso, al operador de mutación. Si nos damos cuenta, con este nuevo cromosoma el problema ya no es paramétrico sino que se ha convertido en combinatorio, pues la solución consiste en ofrecer todas las filas en los genes, pero en un orden específico. En el ejemplo de la figura 119 las filas aparecen en el orden 1,3,0,2. Pero otras soluciones posibles podrían ser 2,0,3,1, y otras más. Tienen que aparecer todos los alelos pues es obligatorio que haya una dama en cada fila. Y manejando adecuadamente el orden de esos alelos lograremos evitar ataques en las diagonales. El operador de mutación típico para problemas combinatorios como este lo veremos más adelante, en la figura 126, donde se eligen dos genes al azar y se intercambian sus alelos.

El cálculo de la aptitud se limitará entonces a contar el número de ataques en las diagonales porque, por diseño, este cromosoma no tiene ataques en las filas ni en las columnas. El problema se vuelve así más rápido de explorar para un algoritmo genético.

Diseñar la función de aptitud. Habitualmente es trivial hacerlo debido a los pocos requerimientos que tiene. La aptitud no tiene por qué ser una función matemática, ni continua, ni derivable. Puede obtenerse como resultado de la evaluación de un modelo, de la votación de personas, de un experimento físico, o cosas similares. Lo único que importa es que arroje un número que sea mayor cuanto mejor sea la solución.

Sin embargo, también puede haber problemas. El más importante ocurre para situaciones de tipo "aguja en un pajar" donde hay algunas soluciones aisladas, rodeadas de un vasto espacio sin ninguna solución. Un ejemplo

es el problema computacional de satisfabilidad. En la figura 120 vemos un ejemplo para la función lógica

$$F(a, b, c, d)= a.(a+b).(a+b+d) \qquad \text{Ec. 36}$$

Esta función tiene 4 variables y, por tanto, su espacio es de 2^4=16 posibilidades, muy pequeño para poderlo visualizar bien. Pero conforme aumenta el número de variables, el espacio de búsqueda lo hace exponencialmente. Los problemas de satisfabilidad son NP-completos. El cromosoma se implementaría con 4 genes binarios (uno para cada variable *a,b,c,d*) y la aptitud se calcularía aplicando la ecuación 36.

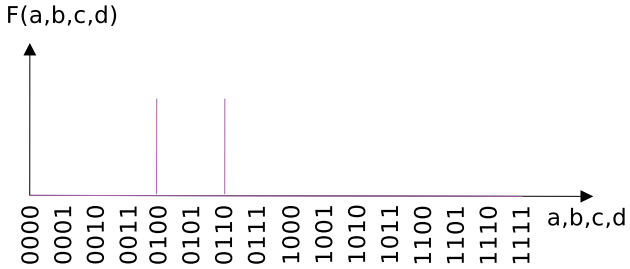

Figura 120. Problema de tipo "aguja en un pajar"

En este ejemplo hay dos soluciones, en 0100 y 0110. Los problemas de tipo "aguja en un pajar" se caracterizan porque no importa cuán cerca esté a una solución. La aptitud saldrá de todos modos 0%, excepto en el momento de llegar a esa solución donde saltará a 100%. De modo que no hay ninguna presión selectiva que ayude a orientar la búsqueda en una u otra dirección.

Conviene advertir que los problemas de tipo "aguja en un pajar" son los más difíciles para cualquier tipo de algoritmo de búsqueda y optimización, no solo para los algoritmos genéticos.

La única forma de abordarlos es tratando de buscar algún truco que me ayude a suavizar la función de aptitud. En este caso, una idea podría ser contar el número de términos que se activan en la ecuación 36. Son 3 términos multiplicativos, de modo que la función valdrá 1 cuando los 3 términos valgan 1. Esto es simplemente una heurística que funcionará en algunos casos, pero no en general, pues el problema, como dijimos, es NP.

Los problemas de tipo "aguja en un pajar" son muy frecuentes en criptografía. Digamos que se me ha olvidado la clave de entrada a mi computador. Pruebo con una, y el computador me responde "frío, frío". Pruebo con otra y el computador me dice "vas mejorando, ya has acertado las tres primeras letras". Pruebo con otra y me dice "caliente, caliente, ya solo tienes una letra equivocada". Sería bueno que el computador respondiera así (para los crac-

kers), pero la realidad es muy distinta. Al teclear una contraseña el computador solo da dos respuestas: "te equivocaste" o "acertaste". De modo que no da ninguna información adicional para orientar la búsqueda.

Y entonces, todo depende del ingenio humano para lograr suavizar la función de aptitud. Con los primeros computadores, no se usaba criptografía realmente sino que se comparaba la clave almacenada con la que introducía el usuario, letra a letra. En cuanto una letra fallaba, de inmediato comenzaba a imprimir el mensaje de acceso denegado. Midiendo el tiempo de demora entre la introducción del *ENTER* de la contraseña y la salida del mensaje, se podía deducir cuántas letras del principio de la contraseña eran correctas. De modo que un problema *NP* se volvía $O(N)$. Solo había que probar con la primera letra, hasta que el tiempo de respuesta aumentaba bruscamente, lo cual significaba que habíamos acertado con ella. Luego, dejando esa primera letra fija, había que probar con la segunda letra, y así sucesivamente.

Esto fue corregido rápidamente, y la industria del *software* produjo un programa de verificación de contraseñas que antes de imprimir el mensaje de acceso denegado se quedase en un bucle sin hacer nada, solo para esperar un tiempo equivalente al del acierto de la contraseña completa. Con ello se lograba que el tiempo de respuesta fuera siempre el mismo. Este sistema también fue violado si se tenía acceso físico a la CPU que ejecutaba el programa, midiendo su corriente de consumo: en los computadores, cuando se hacen cálculos reales el consumo de corriente es alto, mientras que cuando simplemente se está esperando sin hacer nada el consumo es menor. Midiendo el consumo de corriente se podía averiguar de nuevo cuántas letras desde el principio de la contraseña eran correctas. Todo ello al final desembocó en el empleo de criptografía DES y RSA, y así este hueco de seguridad también fue eliminado.

Con esto no quiero animar a nadie a *hackear*. Lo que quiero contar es que, dado cualquier problema teóricamente irresoluble, puede haber una forma ingeniosa de plantearlo donde la solución sea trivial. Y esa forma ingeniosa no está especificada de ninguna manera en el planteamiento del problema. Requiere inteligencia genuina.

La aptitud puede tener también otras dificultades, como que se formen superindividuos, definidos como unos pocos cromosomas muy malos pero mucho mejores que el resto. En la figura 121 y tabla 10, la aptitud del superindividuo B es apenas 4, cuando el óptimo global está en 100. Pero 4 es mucho mayor que las aptitudes de los demás cromosomas.

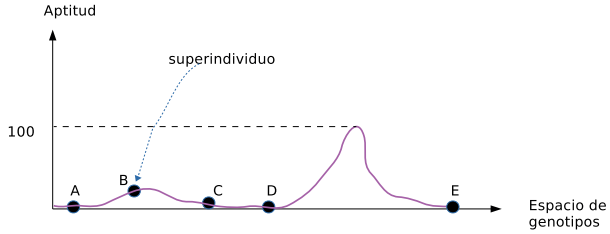

Figura 121. Ejemplo de superindividuo

Tabla 10. Aptitudes cuando hay un superindividuo

Cromosoma	Aptitud
A	0.10
B	4.00
C	0.20
D	0.08
E	0.05

Si utilizamos selección por sorteo, por ruleta (Figura 122) o muestreo universal por restos, esos superindividuos en unas pocas generaciones van a extinguir a los demás cromosomas, haciendo que desaparezca la variedad en la población y provocando una convergencia prematura a una mala solución.

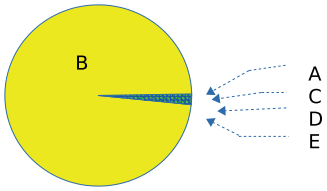

Figura 122. Ruleta con un superindividuo

La selección por torneo no padece este problema, al menos no es tan acusado. Otra alternativa es hacer selección por *ranking*, donde primero se ordenan los cromosomas según su aptitud (primero el de peor aptitud) y el número de este *ranking* se toma como el valor sobre el que realizar la selección, es decir, el número de orden es el área en la ruleta (Tabla 11 y Figura 123).

Tabla 11. Ranking

Cromosoma	Aptitud	Ranking
A	0.10	3
B	4.00	5
C	0.20	4
D	0.08	2
E	0.05	1

De esta forma los superindividuos no aplastan a los demás, y se preserva la variedad en la población.

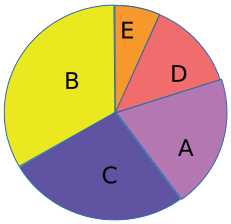

Figura 123. Ruleta usando ranking

Paralelización

Otra característica positiva de los algoritmos genéticos es que son fáciles y efectivos de paralelizar, usando hilos, núcleos o computadores. Hay varias formas de hacerlo:

- Paralelizar la evaluación y la reproducción, manteniendo la población centralizada. Se puede hacer, pero no es muy útil pues se pierde demasiado tiempo en las comunicaciones.
- De grano fino. Se emplea hardware específico, típicamente usando FPGA, donde hay muchas CPU en forma de matriz 2D, y cada CPU está conectada únicamente a sus vecinas de arriba, abajo, a la derecha y a la izquierda. Cada CPU mantiene una población muy pequeña de cromosomas, quizás apenas 2, que cruza y muta e intercambia los hijos con las CPU vecinas.
- Modelo de islas con migración. Se implementa en una red de computadores, todos ellos corriendo el mismo algoritmo genético pero con distintas poblaciones de cromosomas. De vez en cuando los mejores cromosomas se envían a los otros computadores. En este caso el paralelismo es muy eficiente porque casi no se pierde tiempo en comunicaciones entre computadores. La única precaución a tener en cuenta es que las poblaciones deben ser distintas en cada computador, y eso no es obvio de lograr porque hoy día los computadores mantienen sus relojes sincronizados gracias a servicios de red, y resulta que el inicializador del generador de números aleatorios (el famoso *srand(time(0))* en *C++* o similar en otros lenguajes) usa la fecha y hora actuales para generar la semilla inicial, de modo que todos los computadores generarán la misma semilla y, de allí, las mismas secuencias, los mismos alelos y las mismas mutaciones. Para evitarlo, es recomendable ejecutar el *srand(time(0))* en un único computador, y desde allí generar unos cuantos números aleatorios que sirvan como semillas para los demás computadores. En

EVALAB hemos realizado una implementación del modelo de islas (Pineda y Estacio, 2003) usando todos los computadores del laboratorio.

- Arquitectura de inyección. También se implementa en una red de computadores, pero las comunicaciones tienen forma de árbol binario. Se usa cuando la función de aptitud es muy compleja conteniendo varios objetivos. En todos los computadores se ejecuta el mismo algoritmo, pero la función de aptitud se va refinando: en los computadores-hoja es una función sencilla, y conforme se va avanzando por las ramas se van añadiendo más requerimientos a la función de aptitud, hasta llegar al computador-raíz, donde la función de aptitud es la completa. Inicialmente los computadores-hoja comienzan a ejecutar el algoritmo genético y cuando van encontrando soluciones a su función de aptitud, van inyectándolas al siguiente nivel. Al final, en el computador-raíz habrá una población de cromosomas que han ido superando todos los niveles, por lo que será más fácil encontrar allí la solución al problema total. Un ejemplo de aplicación puede ser en el diseño de circuitos impresos, que tienen varios requerimientos: en el primer nivel únicamente se busca que las conexiones entre componentes sean correctas y que no haya cortocircuitos; en el segundo nivel se añade el grosor de las pistas; en el tercer nivel se añaden requerimientos de minimización de interferencias electromagnéticas; y en el cuarto y último nivel se añaden requerimientos de disipación de calor. En cada nivel se busca cumplir un nuevo objetivo manteniendo también los objetivos anteriores.

Aplicaciones de los algoritmos genéticos a problemas combinatorios

En los problemas combinatorios tenemos un conjunto de objetos y la solución a mi problema consiste en entregarlos en un orden concreto. El más conocido es el del vendedor viajero. Un vendedor tiene que pasar por un conjunto de ciudades para vender sus productos. ¿En qué orden debe visitar las ciudades para minimizar la distancia recorrida? (ver un ejemplo con 9 ciudades en la figura 124). Sin embargo, hay muchos otros problemas combinatorios. Como acabamos de ver, las N-Damas se puede convertir a combinatorio.

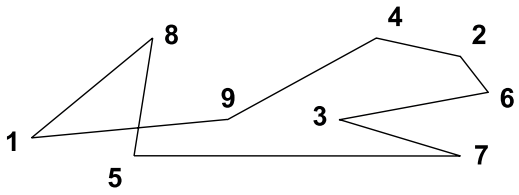

Figura 124. Problema del vendedor viajero

Más formalmente, el problema del vendedor viajero se enuncia así: dadas N ciudades y los costes de viajar de una a otra *{cij}* con *i,j* $\in$ *{1,...N}*, se trata de calcular una trayectoria completa (que visite todas las ciudades), cerrada (que empiece y termine en la misma), conexa (conectada continuamente) y cuya distancia recorrida total sea menor o igual a una constante *D*. Esta es la versión *NP*-completa del problema. Si se busca la trayectoria más corta posible, el problema es *NP*-hard. El espacio de búsqueda es el de las permutaciones de las *N* ciudades.

No existe una forma práctica de codificar una solución a este problema en un cromosoma, para ser usada con los operadores de cruce y mutación habituales, pues estas operaciones dejan de ser cerradas. Típicamente el cromosoma será como el de la figura 125, donde los genes ya no tienen ningún significado posicional, sino solo de orden. En los alelos están las ciudades a visitar.

1	8	5	7	3	6	2	4	9

Figura 125. Cromosoma que implementa una posible solución al problema del vendedor viajero

Y la representación no es única pues también valdrían 8,5,7,3,6,2,4,9,1 o 4,2,6,3,7,5,8,1,9, y muchas otras.

El operador de mutación debe cambiarse por el indicado en la figura 126 para evitar que desaparezcan ciudades.

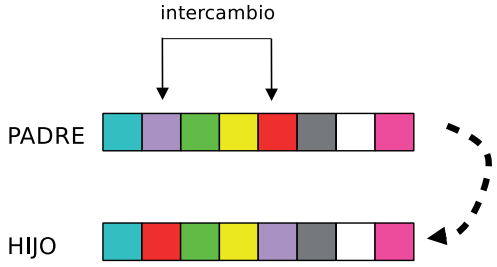

Figura 126. Mutación por intercambio

Observemos que esta mutación produce el cambio más pequeño posible en un cromosoma para problemas combinatorios. Y que si iteramos infinitas veces se producirán todos los cromosomas posibles, de modo que cumplimos con la condición esencial de la mutación.

Desgraciadamente el cruce no es tan sencillo. El cruce uniforme también producirá puntos no factibles, donde se visitarán varias veces la misma ciudad y habrá ciudades que no se visiten nunca. Hay varias propuestas de

nuevos operadores de cruce donde se eviten estos problemas (cruce por orden, cruce por emparejamiento parcial) pero todas son insatisfactorias pues los hijos dejan de parecerse a los padres. La recomendación es no usar cruce en estos casos pues en últimas, no es tan importante. En su lugar, si se desea, se puede utilizar un operador de mutación que haga cambios más grandes.

Aplicaciones de los algoritmos genéticos a problemas multiobjetivo

Lo que se va a contar en este capítulo se aplica a cualquier algoritmo de optimización y no únicamente a los algoritmos genéticos. Lo que ocurre es que en los algoritmos genéticos es muy fácil resolver estos problemas.

Es raro tener un problema con un único objetivo. Lo habitual es que haya varios objetivos que pueden ser incluso contradictorios. Por ejemplo, quiero diseñar una bicicleta eléctrica que pese poco, pero que el motor sea muy potente, que la batería ofrezca mucha autonomía y que el precio sea asequible.

Cuando se desean optimizar varios objetivos simultáneamente, es intuitivo definir una función de aptitud como combinación lineal de los objetivos y luego optimizar esa función:

$$f(x) = \alpha_1 f_1(x) + \alpha_2 f_2(x) + \alpha_3 f_3(x) + ... \qquad \text{Ec. 37}$$

Sin embargo, esta forma de trabajar es ingenua e incorrecta, pues la mayoría de las veces el óptimo de la función linealizada proporciona soluciones mediocres, que no son buenas bajo ninguno de los objetivos individuales. Enseguida veremos por qué.

Otras posibilidades:

- Utilizar un solo objetivo (el más importante). Una vez encontrado el óptimo, se añade a la función de aptitud el siguiente objetivo. Y así sucesivamente. Esto es lo que se hace en la arquitectura de inyección ya comentada.
- Utilizar especiación (diversas especies de cromosomas). Cada una de ellas busca un objetivo. La selección se efectúa dentro de una especie, mientras que el cruce se efectúa entre especies.
- Usar óptimos de Pareto. Esta es la mejor y se explicará a continuación en detalle.

Primero daremos unas sencillas definiciones:

Dominancia de Pareto. Dado un conjunto de N funciones $f_k(x)$ con $x,x_1,x_2 \in X$, y $k=\{1,2,3...N\}$ se dice que x_1 domina a x_2 si y solo si:

$$f_k(x_1) \geq f_k(x_2) \; \forall k \qquad \text{Ec. 38}$$

Óptimos de Pareto. Dados $x_0, x \in X$ se dice que x_0 es óptimo de Pareto cuando no existe ninguna solución mejor en ningún objetivo. Es decir:

$$f_k(x_0) \geq f_k(x)\ \forall k,\ \forall x \in X,\ x_0 \neq x \qquad \text{Ec. 39}$$

Corolario. Si un punto domina a todos los demás puntos, se dice que es óptimo de Pareto.

Veamos unos ejemplos con solo dos objetivos (f_1, f_2) para que sean fáciles de visualizar. En la figura 127, el punto *x1* domina a *x2*. Y no hay ninguna relación de dominancia entre *x1* y *x3* ni entre *x2* y *x3*.

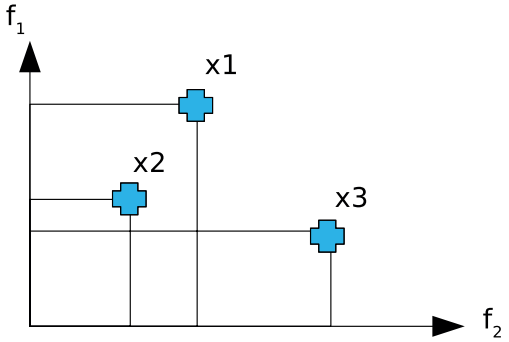

Figura 127. Ejemplo de dominancia de Pareto: x1 domina a x2

En la figura 128, *x1* domina a *x2* y a *x3*, por lo que x1 es óptimo de Pareto. No hay ninguna relación de dominancia entre *x2* y *x3*.

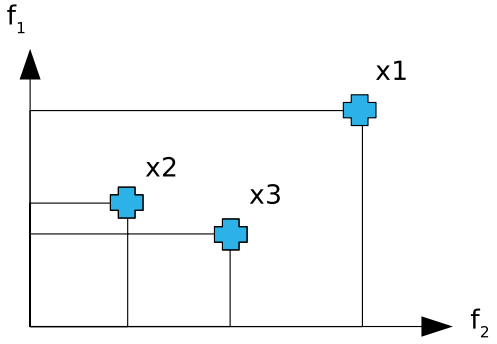

Figura 128. Ejemplo de dominancia: x1 es óptimo de Pareto

Y en la figura 129 puede verse un ejemplo donde *x1* domina a *x2* que a su vez domina a *x3*. Obviamente, *x1* también domina a *x3*, por lo que *x1* es óptimo de Pareto.

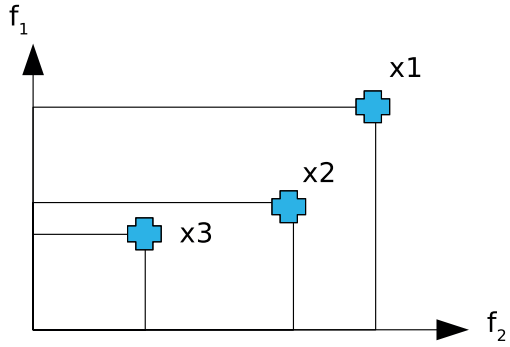

Figura 129. Ejemplo de dominancia: x1 domina a x2 que a su vez domina a x3

¿Para qué nos sirve este concepto de dominancia? Veámoslo con un ejemplo: quiero comprar una bicicleta eléctrica y me ofrecen las que vemos en la tabla 12:

Tabla 12. Ejemplo con bicicletas eléctricas

Modelo	Autonomía (km)	Peso (kg)	Precio
B1	40	33	1000
B2	50	20	1200
B3	50	20	1100

Este es un problema de 3 objetivos donde quiero maximizar la autonomía a la vez que minimizar el peso de la bicicleta y su precio. Por supuesto, donde quiero minimizar puedo cambiarlo de signo para convertirlo en maximizar. Está claro que la bicicleta B3 domina a B2 porque tienen la misma autonomía y peso, pero el precio de B3 es mejor. Eso significa que, al ser B2 una solución dominada, no tiene ningún sentido seguir considerándola como una opción. Cualquier persona racional la tacharía de inmediato de la lista. Nos queda entonces considerar B1 y B3, y vemos que ninguna es óptimo de Pareto. B1 domina en un objetivo (precio) mientras que B3 domina en dos objetivos (autonomía y peso).

En principio cuando tenemos varias propuestas y ninguna domina a la otra (o sea, no hay óptimo de Pareto), todas ellas forman lo que se llama la "frontera de Pareto" (ver otro ejemplo en la figura 130), y son soluciones válidas al problema. La frontera de Pareto puede variar conforme visito más tiendas y obtengo más cotizaciones. Claro que al final, en un caso real, hay que comprar una bicicleta, por lo que tendremos que decantarnos bien sea por B1, bien sea por B3. Si no hay información adicional (preferencias, importancia de los objetivos, dinero disponible) lo más razonable es elegir B3 porque domina en más objetivos que B1.

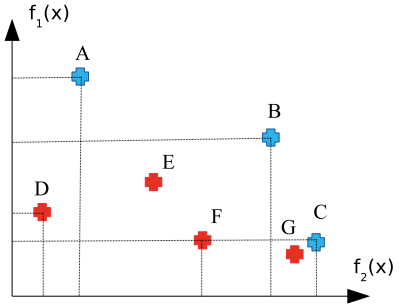

Figura 130. A, B y C forman parte de la frontera de Pareto, mientras que D, E, F y G están dominados

Hagamos lo mismo con algoritmos genéticos. Cada cromosoma representa un punto en un espacio multidimensional de objetivos (Figura 130). La idea es determinar cuál es la frontera de Pareto y darle algún tipo de privilegio, por ejemplo, eliminar los cromosomas dominados, reproducir con mayor probabilidad los que están en la frontera, o algo similar. De este modo, la frontera irá expandiéndose conforme pasan las generaciones (Figura 131).

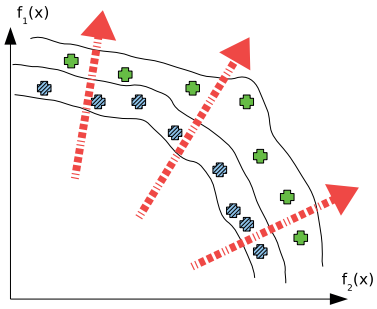

Figura 131. Evolución de la frontera de Pareto

A veces se admite que la frontera de Pareto tenga un cierto grosor, es decir, se acepta allí algunos cromosomas dominados pero que están muy cerca de cromosomas dominantes, con el objetivo de aumentar la variedad. Si vamos a hacerlo, en el ejemplo de la figura 130 se admitiría al cromosoma G dentro de la frontera de Pareto.

Otra forma de aplicar Pareto a los algoritmos genéticos es, en el momento de hacer la selección, utilizar el número de veces que domina cada cromosoma. En la figura 132 se muestra un ejemplo con 3 funciones objetivo a maximizar. En las abscisas están 5 cromosomas que vamos a suponer son los que conforman la población en un momento dado.

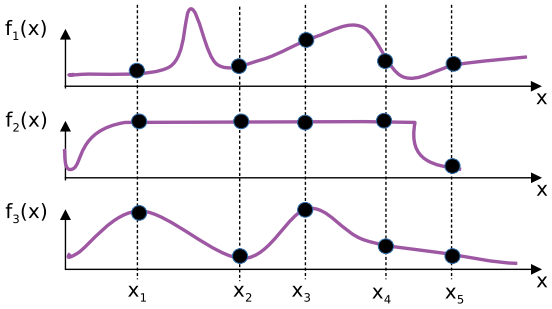

Figura 132. Selección por dominancia

Contemos, para cada cromosoma, cuántos objetivos domina. El resultado puede verse en la tabla 13.

Tabla 13. Dominancias de cada cromosoma

Cromosoma	Número de objetivos en los que domina
x1	2
x2	1
x3	3
x4	1
x5	0

Es importante entender que x3 es dominante en $f_1(x)$ ya que es mejor o igual a todos los demás cromosomas. Aunque entre *x1* y *x2* hay un pico de la función $f_1(x)$ que sobrepasa a $f_1(x3)$, ello no cuenta porque allí no hay ningún cromosoma. Es decir, hemos pintado las tres funciones en trazo continuo por razones pedagógicas, pero la verdad es que desconocemos el valor de esas funciones salvo en los sitios donde hay cromosomas (o sea, en los puntos negros).

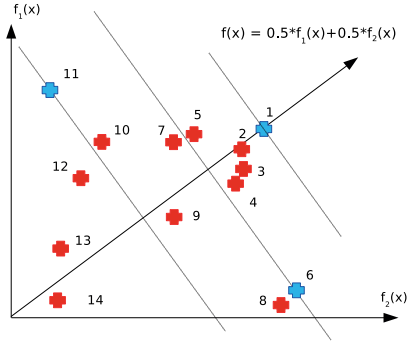

Figura 133. Linealización con 50% y 50%

Para terminar, vamos a explicar por qué linealizar todos los objetivos en uno solo no es una buena idea. Supongamos que tenemos dos objetivos y la población de soluciones dada en la figura 133 donde los cromosomas azules son la frontera de Pareto, y los rojos son los dominados. Si hallamos el promedio de ambos objetivos para resumirlos en uno solo, esto es, $f(x) = 0.5*f_1(x)+0.5*f_2(x)$, con ello creamos un nuevo eje (en diagonal) respecto al cual hay que maximizar. El mejor punto según este criterio lo hemos marcado con el número 1, el siguiente con el 2, siendo el último el 14.

Aquí ya se evidencian problemas: el punto 11, a pesar de estar en la frontera de Pareto, es superado por muchos puntos mediocres {2, 3, 4, 5, 7, 8, 9 y 10}. Lo mismo le ocurre al punto 6, que es superado por los mediocres {2, 3, 4 y 5}.

Pero la injusticia no termina allí. Supongamos ahora que las ponderaciones para linealizar los dos objetivos sean 20% y 80% respectivamente. El resultado lo tenemos en la figura 134 donde hemos vuelto a asignar un número de orden a cada solución (1 a la mejor, 14 a la peor). Allí podemos comprobar que la mejor solución ahora es la que antes quedó en sexto lugar, y la siguiente es la que quedó antes en primer lugar.

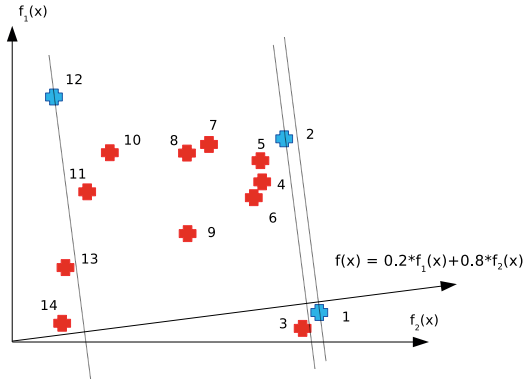

Figura 134. Linealización con 20% y 80%

Al cambiar las ponderaciones cambia el orden radicalmente y, además, hay de nuevo una injusticia con el punto 12 que, estando en la frontera de Pareto, queda superado por los mediocres {3, 4, 5, 6, 7, 8, 9, 10 y 11}.

Repitiendo de nuevo el experimento con ponderaciones 80% y 20%, cuyo resultado podemos ver en la figura 135, observamos que ahora el mejor cromosoma es el que quedó en lugar 11 en la primera clasificación. Otro cambio de orden radical y nuevas injusticias.

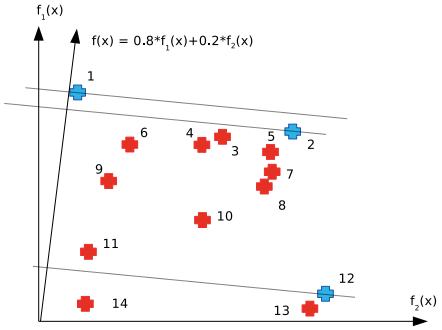

Figura 135. Linealización con ponderaciones 80% y 20%

Lo único que se puede demostrar es que al linealizar el primer punto siempre pertenece a la frontera de Pareto. Pero el orden cambia con los pesos asignados, y no de forma leve, sino radical. Además, se cometen injusticias con puntos mediocres que quedan mejor clasificados que puntos que pertenecen a la frontera de Pareto, es decir, los realmente mejores.

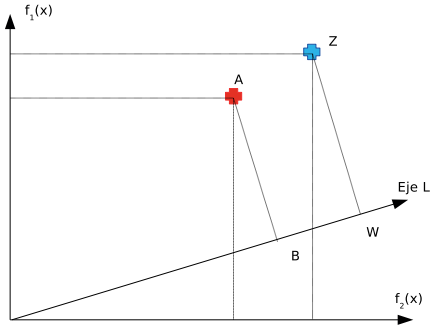

Figura 136. El mejor punto de cualquier ponderación siempre está en la frontera de Pareto

Teorema. Cuando se hace un *ranking* con ponderaciones positivas, la mejor solución siempre está en la frontera de Pareto, independientemente de las ponderaciones que se asignen a cada eje para hacer el *ranking*.

Demostración. Supongamos que no es así, es decir, que existe un punto dominado A (o sea, que no está en la frontera de Pareto) que queda primero en un cierto *ranking*. Ese *ranking* se caracteriza por unos pesos multiplicativos positivos sobre los ejes, que producen una transformación lineal, o sea, un nuevo eje *L*, dentro del primer cuadrante (Figura 136).

Entonces la perpendicular a ese eje *L* que pasa por ese punto *A*, interseca al eje *L* en un cierto punto *B*.

Sin embargo, al ser dominado existirá un punto *Z* que domine a *A*, es decir, que sea mejor o igual a él en todas las coordenadas.

Ello implica que la recta que pase por *Z* siendo perpendicular al eje *L*, intersecará a *Z* en un punto *W* tal que se cumplirá *W*>*B* respecto al eje *L*, al ser todas las ponderaciones positivas (primer cuadrante en la figura 136).

De modo que ello contradice que *A* sea el primero en el ranking respecto a un eje *L*.

Por tanto la suposición es falsa, es decir, no existe ningún *ranking* (eje *L*) en donde la proyección *B* de un punto dominado *A* quede en primer lugar.

De ahí se deduce que en cualquier *ranking*, independientemente del peso que se dé a cada eje, la primera solución siempre pertenece a la frontera de Pareto.

La mejor solución siempre es una excelente solución, lo cual es bueno. Desgraciadamente no hay garantías para el resto de puntos que conformen la frontera de Pareto, pues pueden quedar en cualquier orden en un *ranking*, en función de los pesos que se den a cada eje.

Una última reflexión: realmente solo tiene sentido linealizar objetivos cuando hay un factor de conversión. Por ejemplo, quiero maximizar las ganancias de mis inversiones en pesos colombianos, dólares y euros. En realidad no son tres dimensiones sino solo una porque existen factores de conversión entre monedas. Es lo mismo que si quiero maximizar la carga en un camión de dos compartimentos y en un lado mido en kilos y en otro en gramos. Dado que 1000 g = 1 kg entonces no son dos dimensiones distintas, sino solo una. Pero, en el ejemplo de la bicicleta, no hay ningún factor que convierta kg a km de modo que son dimensiones distintas, cuyos objetivos no se pueden linealizar.

A pesar de que este tópico es bien conocido, sigue siendo muy común encontrar problemas multiobjetivo donde se linealiza para encontrar una solución. Lo vemos continuamente en las clasificaciones de las mejores universidades y lo hemos criticado (Delgado, 2017) pues este error suele tener repercusiones sociales importantes.

Recomendaciones

Si son tan buenos, ¿debo usar siempre un algoritmo genético para problemas de optimización? Definitivamente no, y en el segundo libro veremos las razones teóricas para ello (el teorema de *no-free-lunch*). De momento plantearemos los escenarios donde conviene o no conviene usar algoritmos genéticos.

Si se conoce un algoritmo determinista que solucione el problema en un tiempo razonable, utilice ese algoritmo. Los algoritmos genéticos van a ser más lentos y menos precisos. Ejemplo: si necesita calcular el máxi-

mo de una función continua, derivable y con uno o pocos picos, utilice el algoritmo del gradiente.

Si el problema tiene muchas restricciones duras (esto es, valores de los parámetros que producen soluciones inválidas), entonces no suele ser conveniente utilizar algoritmos genéticos, pues no van a encontrar la solución o van a requerir demasiado tiempo. Un ejemplo con restricciones duras puede ser programar horarios y aulas para una universidad. Aquí hay varias restricciones duras: no se pueden dar dos clases en la misma aula a la misma hora, un profesor no puede dar dos clases en el mismo horario, el número de estudiantes en un aula debe ser menor o igual al número de sillas, y algunas otras. Los algoritmos genéticos no van a funcionar aquí debido a que los operadores de reproducción producirán muchas soluciones inviables. Conforme aumentan las restricciones, la mayor parte de los cromosomas (probablemente todos) van a representar soluciones inviables. La aptitud de esas soluciones es cero y la presión selectiva no operará con ellas. No hay forma de orientar el algoritmo para que, a partir de esas soluciones, encuentre otras mejores. El algoritmo genético degenerará en una búsqueda al azar. En este caso es mejor emplear otro tipo de algoritmo, como los basados en restricciones.

La única alternativa es convertir de alguna forma las restricciones duras en restricciones blandas, y allí sí pueden funcionar los algoritmos genéticos espléndidamente. En el ejemplo anterior podríamos aceptar soluciones con más estudiantes que sillas, penalizando progresivamente la función de aptitud (si nos pasamos poco, penalizamos poco, y si nos pasamos mucho, penalizamos mucho). Como consecuencia, el algoritmo sigue operando con la selección presionando para lograr mayores aptitudes, lo que implica menores penalizaciones, es decir, que tratará de buscar aulas donde quepan realmente los estudiantes. Y, por otro lado, si la solución final es definitivamente inviable (por ejemplo, un grupo de 45 alumnos programado en un aula de 40 sillas), el rector de la universidad puede decidir comprar las 5 sillas faltantes y acomodarlas de alguna manera para evitar dejar de dictar una asignatura. En este sentido, los algoritmos genéticos pueden ofrecer aproximaciones razonables a soluciones, aunque sean levemente inviables. Los algoritmos basados en restricciones no pueden hacerlo porque podan inmediatamente el árbol de búsqueda cuando no se cumple alguna restricción, impidiendo acercarse a soluciones levemente inviables.

Si el problema no tiene propiamente una función de aptitud, o la función no es derivable o no es continua, o el problema está mal definido, o tiene mucho ruido, o cambia con el tiempo, o el espacio de búsqueda es enorme o no existe ningún algoritmo que lo resuelva en tiempo razonable (típicamente en problemas NP o peores), entonces los algoritmos genéticos son una

buena opción, probablemente la única. Un ejemplo es el diseño artístico. Es imposible definir matemáticamente si una pintura o una canción es más bonita que otra. No hay posibilidad de sacar la derivada de la belleza de una escultura. E incluso el concepto de originalidad, creatividad y belleza en el arte cambia con el tiempo y con la persona que observa. Por otro lado, el espacio de diseño es enorme. Piense en cuantos cuadros distintos se pueden pintar. En estos casos, los algoritmos genéticos suelen funcionar muy bien. Habitualmente lo que se hace es tomar como función de aptitud el criterio subjetivo de muchos observadores humanos, exponiendo los objetos artísticos fabricados por los cromosomas a través de una aplicación web. Los objetos más votados por los humanos se reproducirán más. En EVALAB hemos realizado dos trabajos de grado en este sentido para diseño de logotipos (Camayo, 2009) y para creación de nuevos sonidos (Barona, 2013).

Otras aplicaciones

Una de las primeras aplicaciones que reavivó el interés por estos algoritmos, y que fundó el área de *hardware* evolutivo, la realizó Adrian Thompson en 1996 usando *FPGA*[29]. El objetivo era distinguir entre dos tonos, uno de 1 kHz y otro de 10 kHz (usado para reconocer los dígitos a partir de las pulsaciones en los actuales teléfonos multitono).

Una *FPGA* es un chip que contiene millones de puertas lógicas *AND*, *OR*, *NOT* y bits de memoria, que son las puertas básicas con las que se puede construir cualquier circuito digital, incluso un computador, aunque la tecnología de aquella época solo integraba unos pocos cientos. Lo interesante de la *FPGA* es que las conexiones entre las puertas son programables por *software*. Hay millones de conmutadores que permiten conectar o desconectar las salidas de cualquier puerta con las entradas de otras. Cada conmutador contiene un bit de memoria que puede ser programado (0=desconexión; 1=conexión). En la figura 137 puede verse un esbozo de ello, aunque las arquitecturas reales de las *FPGA* suelen ser más complejas. Lo relevante es que en los chips convencionales esas conexiones las hace el fabricante y quedan para siempre, mientras que en las *FPGA* se pueden hacer y deshacer tantas veces como se quiera, enviando la correspondiente secuencia de ceros y unos desde el exterior de la *FPGA* (por ejemplo, desde un computador) hacia su matriz de conmutadores.

29 *Field Programmable Gate Array*, o sea, matriz de puertas lógicas programable después de fabricar e incluso vender el circuito.

Thompson diseñó en su computador un algoritmo genético donde los cromosomas eran secuencias completas para programar los conmutadores de su *FPGA*. Por mutación y cruce se generaban nuevas secuencias. Para evaluar un cromosoma se enviaba esa secuencia de ceros y unos a la *FPGA*, se le inyectaban tonos de 1 kHz y 10 kHz y se veía si los conseguía identificar, en función de lo cual se le asignaba una aptitud. Y ya tenemos la evolución en marcha. Al principio, todos los cromosomas eran inútiles, pero con el paso de las generaciones fueron mejorando, hasta alcanzar el éxito completo en la generación 4000.

La primera conclusión es que los algoritmos genéticos pueden diseñar complejos circuitos electrónicos sin saber nada del tema.

La segunda conclusión fue más interesante aún. Cuando Thompson revisó el interior de la *FPGA* para averiguar qué clase de circuito había diseñado la evolución, no entendió nada. Había puertas completamente desconectadas del resto, pero que si se eliminaban, el circuito dejaba de funcionar. Había otras puertas con entradas al aire, lo cual está completamente prohibido a los ingenieros electrónicos, pues esas entradas captan todo tipo de ruidos electromagnéticos volviendo impredecibles sus salidas. Había cortocircuitos. En fin, si este circuito hubiera sido diseñado por un estudiante, habría recibido un cero como calificación. Y, sin embargo, funcionaba. La conclusión a la que se llegó es que el circuito operaba correctamente porque las puertas se comunicaban entre sí no solo por los cables internos y los conmutadores, sino también por ondas electromagnéticas. Y de allí la importancia de las entradas sin conexión, que eran antenas receptoras, y los cortocircuitos, que eran emisores. La evolución empleó todo lo que encontró para lograr su objetivo. No se limitó a la electrónica digital sino que también usó la física electromagnética. Igual que ocurre en la evolución biológica que, como decía Jacobs, no hace ingeniería sino *bricolage*.

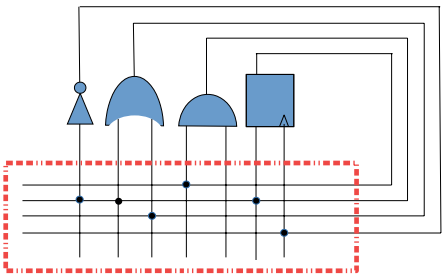

Figura 137. Lo básico de una FPGA: en azul, de izquierda a derecha hay una puerta NOT, una OR, una AND y un bit de memoria, con las entradas abajo y las salidas arriba. En el recuadro rojo está la matriz de conmutadores, donde un punto negro significa que hay conexión

En EVALAB hemos usado estos algoritmos para resolver varios problemas interesantes: en optimización paramétrica, para minimizar el desperdicio en el corte de telas por medio de maquinaria automática (Guzmán, 2008); en búsqueda combinatoria para asignar espacios y horarios en consultorios médicos (Castrillón, 2015), en universidades (Villegas, 2001; Barón, 1999), esta última usando otra técnica evolutiva, llamada *simulated annealing*, que veremos enseguida; en la búsqueda de estrategias óptimas en juegos como el dilema del prisionero (Gómez, 2001), el juego del Otelo (Villate, 2012), el Tetris (Triana, 2013) o juegos de guerra (Posada, 2014); y también los hemos empleado para búsqueda de imágenes etiquetadas (Lourido y Solano, 2004).

En el dilema del prisionero, aparte de confirmar otros experimentos sobre el origen de la cooperación, obtuvimos uno de nuestros primeros resultados sorprendentes. La teoría de juegos (que se verá en el correspondiente capítulo) dice que no hay ninguna estrategia que tenga garantías de ganar siempre, aunque se pueden diseñar estrategias adaptativas que suelen ganar en muchos casos. Sin embargo, al correr nuestro algoritmo genético, obteníamos una estrategia que ganaba siempre. Al revisar el código nos dimos cuenta de un pequeño error: la variable *ultimaJugadaRealizada* debería ser un atributo de los objetos *Jugador*, pero erróneamente la habíamos declarado como atributo de la clase. Eso significa que era común a todos los jugadores. El jugador evolutivo se aprovechaba de ello para mirar cuál era la última jugada realizada por el jugador contrario y sacar ventaja de ello. Como dice la segunda regla de Orgel: "¡la evolución es más lista que tú!". Ello también nos hizo reflexionar sobre las potencialidades de los algoritmos evolutivos para descubrir errores en *software*. Algo similar acaba de ocurrir con los *chatbots* de *Facebook*, que los pusieron a dialogar entre sí y descubrieron por si solos un *bug* en el *software* que les permitió crear su propio idioma, para lograr una comunicación más eficiente.

También aplicamos algoritmos genéticos para mejorar el filtro de entrada a la carrera de ingeniería de sistemas (*computing science*) de la Universidad del Valle (Torres, 2013). La idea era analizar la base de datos de estudiantes que entró a esta carrera y su nota promedio de salida (o sí había salido expulsado). Se empleaba un examen clasificatorio de entrada, que evaluaba diferentes áreas de conocimiento (matemáticas, español, inglés, física, química y ciencias sociales). El conjunto de parámetros a optimizar estaba constituido por las ponderaciones de cada una de estas áreas, lo cual indica su importancia en la carrera. La idea era buscar las características de los estudiantes que maximizaban la probabilidad de terminar la carrera y con una buena nota promedio. Con ello esperamos orientar mejor a los estudiantes al elegir su carrera, evitando la frustración de salir expulsados a la mitad. Como curio-

sidad, antes de realizar este estudio el componente de matemáticas era el que tenía mayor peso, pero al finalizar este trabajo nos dimos cuenta que era una equivocación, pues estudiantes con buenos conocimientos en matemáticas no necesariamente tenían buenas notas en computación.

¿El final de las ingenierías?

Estas aplicaciones nos obligan a reflexionar sobre el futuro que nos espera. Porque, como es bien sabido, las ciencias tratan de descubrir cómo funciona el mundo, es decir, lo estudian para sacar sus ecuaciones de funcionamiento. Mientras que las ingenierías usan esas ecuaciones de funcionamiento de la materia sólida, de las ondas, de los fluidos y de la electricidad para diseñar vigas, circuitos electrónicos, tuberías y cables. La ciencia hace análisis mientras que la ingeniería hace síntesis, principalmente.

Pues bien, los algoritmos evolutivos hacen también síntesis a partir del análisis, como puede verse en la figura 138, de modo que tenderán a sustituir a los ingenieros de todas las áreas.

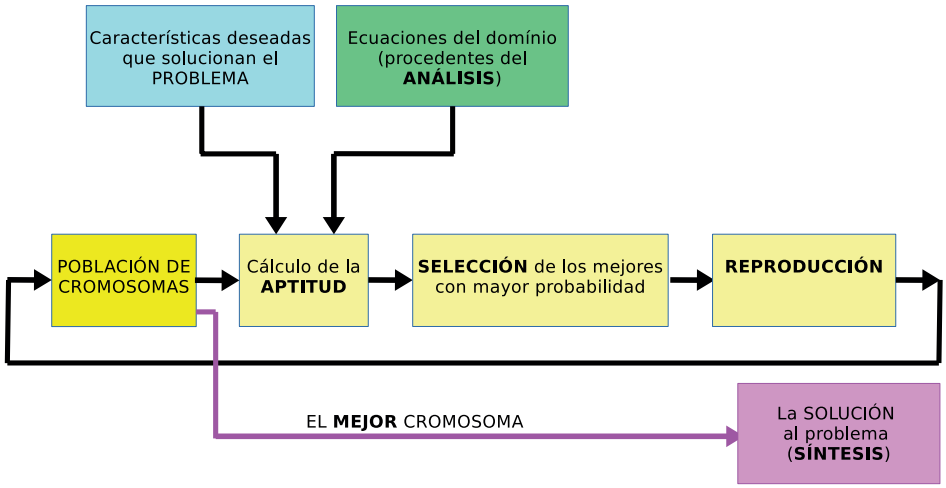

Figura 138. Síntesis, vía análisis

Esta es otra forma de ver la figura 105, donde se hace más explícito como se calcula la aptitud de cada cromosoma a partir las características que deseo como solución a mi problema y las ecuaciones que intervienen allí (que nos las da el análisis que realiza la ciencia). De la automatización no se salvan ni la ingeniería, ni la educación, ni la ciencia, pero aquí estamos viendo lo que va a ocurrir con la primera.

Respecto a la ciencia, ya hay muchos éxitos en automatizar la tarea del descubrimiento científico usando series de datos procedentes de experi-

mentos, de donde emergen las ecuaciones subyacentes a un fenómeno físico, aunque sean no lineales (Bongard y Lipson, 2007; Koza, 1992). Con ellas se pueden redescubrir las leyes de la física conocidas, como las leyes de Kepler, y descubrir otras nuevas.

Y no solo sustituirán a los ingenieros, sino también a cualquier proceso creativo, como el diseño industrial, la música, la poesía, la arquitectura, la pintura, por nombrar algunas. Hoy día muchos artistas reconocen que no pueden firmar sus obras, pues realmente fueron realizadas por algoritmos evolutivos. Quiero mencionar solo a dos de los más antiguos artistas que emplean esta tecnología con resultados excepcionales: Scott Draves (2017) y William Latham (2017). En los respectivos enlaces de la bibliografía de este capítulo, se puede conseguir *software* gratuito para desarrollar nuestros propios experimentos artísticos. El resultado suele ser sorprendente porque uno se da cuenta de que allí hay arte, a la vez que reconoce con asombro que no parece arte generado por un ser humano.

En el caso de las disciplinas mixtas entre ingeniería y arte, como el diseño industrial y la arquitectura, hay también resultados sorprendentes, algunos de los cuales pueden verse en Bentley (1999).

Respecto a la ingeniería de *software*, está claro que la programación evolutiva y sus variantes generan programas, así sean cortos debido a las limitaciones tecnológicas actuales. Aunque ya se comienza a vislumbrar cómo hacer programas más largos utilizando la misma técnica de la figura 138 y la metodología BDD. Básicamente, el programador se limitará a diseñar las pruebas que desea que pase el *software* (el bloque de color azul de la mencionada figura). En *software* no hay "leyes de la física", de modo que el bloque de color verde desaparece. Cada cromosoma es un programa generado al azar. Y el cálculo de la aptitud consiste en contar cuantas pruebas pasa el cromosoma. A mayor número de pruebas correctas, mayor es su aptitud. Esta idea la planteamos como tesis doctoral en EVALAB, aunque desgraciadamente no se logró concretar, pero ya ha sido llevado a cabo por Arcuri y Yao (2014), de modo que la ingeniería de *software* también está *ad portas* de desaparecer.

Mitos ampliamente divulgados sobre los algoritmos genéticos

Hay muchos mitos y son difíciles de erradicar porque en algunos casos quienes los promueven son los mismos creadores de los algoritmos, quizás como una estrategia para diferenciarse unos de otros. Vamos a mencionar los más importantes:

Genes binarios. En la literatura, especialmente la más antigua, se menciona que los cromosomas deben usar genes binarios, es decir, los alelos

deben ser {0,1}. Esta idea obliga a codificar en binario las soluciones a cualquier tipo de problema. Y hay argumentos a favor de esa idea. Quizás su mayor ventaja es que simplifica el análisis teórico de las mutaciones y cruces, pero al final, no produce ninguna mejora en la eficiencia de las implementaciones. Lo más sensato es emplear una codificación directa y adecuada al tipo de problema que vayamos a resolver. Si en un gen debemos codificar si un automóvil va a llevar frenos ABS o no, es perfecto que ese gen sea binario. Pero si vamos a codificar el diámetro de las ruedas en milímetros, es mejor usar un número entero.

El cruce debe ser de un punto de corte. En la literatura, especialmente en la más antigua, se menciona que el cruce debe hacerse de uno o dos puntos. Recordemos rápidamente como es el cruce de un punto: se seleccionan dos cromosomas (los padres) y un punto de corte para ambos que los divide en dos trozos cada uno. Se toma la parte derecha del padre y se junta con la izquierda de la madre. Se toma la parte izquierda de la madre y se junta con la derecha del padre. Con ello se obtienen dos hijos. Hay mucha literatura que reconoce sus ventajas, y mucha más que destaca sus defectos. Se menciona la "hipótesis de los bloques constructivos", que hace tiempo se ha demostrado que es falsa, y que dice que gracias a ese cruce, cierto bloque de parámetros que dan buen resultado con el padre se podrán juntar con otro bloque de la madre, produciendo un hijo con lo mejor de ambos. Esto solo es cierto para un limitadísimo tipo de problemas. Lo que es peor, en el momento de diseñar el cromosoma es importante poner juntos los genes que van a codificar esos bloques, para que el cruce no los separe, y esa información no suele ser obvia. Se han diseñado esquemas alternativos para que los genes puedan moverse de sitio y agruparse por sí mismos, pero lo único que se logra es complicar el algoritmo y volverlo mucho más lento. Por otro lado, el cruce uniforme no sufre de esos problemas, y si efectivamente existen bloques constructivos los encontrará independientemente de la posición de los genes en el cromosoma.

El cruce es lo importante. En el mismo orden de ideas, en la literatura, especialmente la más antigua, se dice que el operador de cruce es el fundamental en los algoritmos genéticos, y que el operador de mutación es accesorio, que debe usarse con baja probabilidad y que incluso se puede suprimir. Nada más lejos de la realidad, pues el operador de mutación es el fundamental, ya que sirve para generar variabilidad, cosa que el de cruce no puede hacer. Una vez perdido un alelo en un gen de la población, solo la mutación puede volverlo a crear. Y sirve para explorar todos los casos posibles si se le da suficiente tiempo, cosa que el cruce tampoco puede hacer. La mutación es un operador de cambios pequeños, por lo que explota las mejo-

res soluciones. Es importante que haya un operador explorador que realice cambios grandes, y efectivamente eso se puede lograr con el operador de cruce, pero también con muchos otros operadores incluyendo mutaciones grandes. Además, en biología puede apreciarse que las mutaciones están presentes en todos los seres vivos, mientras que el cruce es un operador de alto nivel que solo funciona en organismos superiores y que requiere de una considerable infraestructura previa, como la diferenciación sexual.

Reflexionemos un poco más sobre este aspecto. El cruce es un operador de muy alto nivel que, en biología, funciona solo entre organismos de la misma especie, esto es, los que tienen muchísimos genes idénticos. De modo que lo que realmente hace es una búsqueda local en un espacio muy pequeño. Genera variedad, pero sin salirse de una frontera que enmarca a la especie. Por el contrario, la mutación no tiene límites. En biología hace una búsqueda global, y es la única que realmente puede generar una nueva especie.

El cruce requiere datos fuertemente estructurados. Por ejemplo, sería útil en programación orientada a componentes, para sustituir un componente por otro funcionalmente equivalente. La mutación es de bajo nivel y, por ello, la mayoría de las veces es destructiva. Pensemos, por ejemplo, si hacemos un cambio al azar en un programa en Ruby, lo más probable es que produzca algún error de sintaxis (claro que si se hacen millones de mutaciones, de vez en cuando se producirá alguna mejora al código). Pero en muchos otros problemas no existen esos datos fuertemente estructurados, de modo que la mutación es el único operador razonable.

Tienen soporte teórico y garantía de funcionamiento. Nada de ello es cierto. Holland desarrolló la "teoría de los esquemas" para dar un soporte teórico a los algoritmos genéticos, pero aunque contiene ideas muy llamativas, también estaba incompleta y adolecía de varios errores. Los errores se repararon y se terminó de desarrollar, pero el resultado es una fórmula que no vamos a presentar aquí por su inutilidad práctica, donde la mayoría de los términos son desconocidos y de donde no sale ninguna recomendación ingenieril a seguir. Otros investigadores han probado algunas propiedades límite, por ejemplo, cuando la población es infinita, lo cual no sirve de consuelo a quien vaya a usar estos algoritmos en la práctica. Y también se han utilizado otros métodos como los modelos ocultos de Markov, con un éxito limitado y poca aplicabilidad práctica. De la misma manera, no hay garantías de que los algoritmos genéticos converjan, ni tampoco de que si lo hacen, lo hagan al óptimo global. Y si los ejecutamos varias veces sobre el mismo problema, lo habitual es que ofrezcan cada vez una solución distinta. Esta falta de garantías y ausencia de determinismo puede sonar muy

desanimadora, pero la verdad es que es una de las grandes fortalezas de los algoritmos genéticos: son algoritmos creativos gracias a su aleatoriedad. En el segundo libro veremos que el teorema de Gödel limita la capacidad de los sistemas deterministas.

Sistemas clasificadores evolutivos

Los sistemas clasificadores evolutivos[30] también fueron ideados por John Holland como una aplicación de los algoritmos genéticos para robots artificiales (*Animats* = *Animal*+*robot*), que debían moverse en un cierto entorno, interactuar con él y aprender para sobrevivir. Hoy día forman parte de lo que se ha dado en llamar Aprendizaje de Máquina[31].

Se emplea cuando se desea controlar un entorno caracterizado por cambios constantes, y eventos importantes mezclados con ruido, donde es necesario responder a esos eventos en tiempo real y donde los objetivos son implícitos o están mal definidos o hay un objetivo general difícil de cumplir como, simplemente, sobrevivir.

Es similar a un sistema experto, pues está basado en reglas de tipo *IF-THEN*, pero el conocimiento lo va adquiriendo sobre la marcha. Esa es la gran ventaja, porque no necesita un experto humano. Los expertos humanos son caros y poco colaborativos (después de todo, el objetivo es sustituirles por un *software*). Y el producto final es frágil, tanto si el experto comete algún error con las reglas que enuncia, o posee conocimientos que no logra hacer explícitos o tiene lagunas en esos conocimientos. También es frágil en el sentido de que su ámbito de aplicación es estrictamente aquel donde fue diseñado, no permitiéndose ningún cambio. Por el contrario, un sistema clasificador evolutivo siempre está aprendiendo, por lo que puede remediar sus propios errores y lagunas y se adapta a nuevos entornos con facilidad.

Los sistemas clasificadores evolutivos están basados en una población de reglas de aprendizaje sencillas, también llamadas clasificadoras:

IF <condición> THEN <mensaje>

A ello se le llama "sistema de producción", que es computacionalmente completo, y se implementa por medio de un algoritmo genético, donde cada regla es un cromosoma.

Tanto las condiciones como los mensajes utilizan un código binario {0,1,#}, donde "#" significa "no importa".

30 *Learning Classifier Systems.*

31 *Machine Learning.*

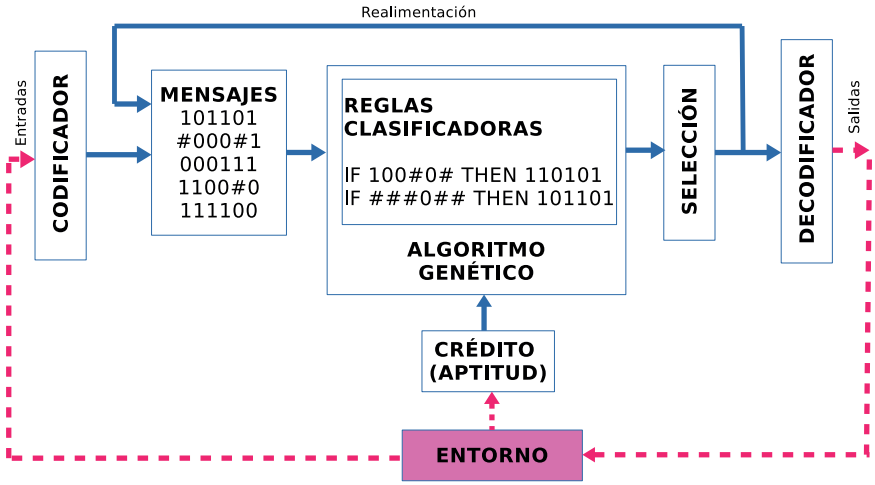

Figura 139. Diagrama de bloques de un sistema clasificador evolutivo

Como puede verse en la figura 139, está constituido por los siguientes módulos en color azul (los de color rojo corresponden al entorno):

- Módulos de codificación/decodificación del entorno en mensajes internos.
- Módulo del algoritmo genético (donde los cromosomas son las reglas clasificadoras).
- Módulo de selección.
- Módulo de crédito (recompensa, fuerza o aptitud).

Las entradas procedentes del entorno se codifican en una o más cadenas binarias, que se inyectan a un tablón de mensajes.

Las reglas clasificadoras chequean el tablón de mensajes. Si hay alguna cadena que coincida con su respectiva condición IF, entonces se activa la respectiva regla. A continuación, el tablón de mensajes se vacía.

Cada regla clasificadora tiene asociada una aptitud. Las que estén activas compiten por ser seleccionadas (ruleta, torneo...), en función de su aptitud. Las que sean seleccionadas, generan su respectivo mensaje, que se inyecta al tablón de mensajes y que también puede ser decodificado para producir una salida hacia el entorno.

Todas las reglas activadas deben pagar un "impuesto", que se les descuenta de la aptitud. Y el entorno debe ofrecer una recompensa a las reglas que generaron salidas buenas y que se suma a la aptitud. Esa recompensa se distribuye a todas las reglas que contribuyeron a producir esa salida.

Hay varios algoritmos de distribución, pero uno muy usado (y también muy criticado) es el de la "brigada de baldes" (Figura 140), que funciona así:

la regla que envió al entorno la salida, recibe una recompensa. Una parte proporcional de esa recompensa la cede a la regla que produjo el mensaje anterior que la activó, y así hacia atrás, siguiendo la trayectoria inversa que generó el mensaje de salida.

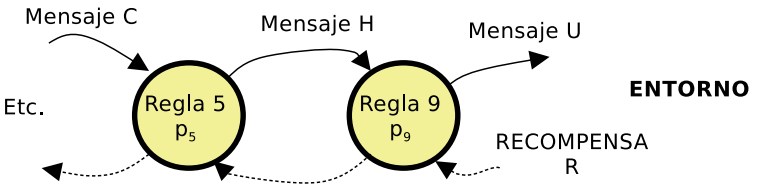

Figura 140. Reparto de recompensas con algoritmo brigada de baldes

La recompensa se reparte usualmente con la siguiente fórmula:

$$p \leftarrow p+\beta^*(R-p) \text{ con } 0<\beta\leq 1 \qquad \text{Ec. 40}$$

Siendo p la aptitud, R la recompensa del entorno y β la tasa de aprendizaje.

El principal problema de los sistemas clasificadores evolutivos es que cuando los mensajes de salida son producidos por cadenas muy largas de reglas clasificadoras, el reparto de la recompensa hace que a cada una le toque muy poco. Y entonces, el sistema aprende muy despacio o no aprende.

El objetivo del algoritmo genético es ir creando nuevas reglas que reciban mayores recompensas, o sea, que estén mejor adaptadas al entorno.

Hay dos variantes de estos algoritmos:

- **Variante Michigan** (de Holland, que es la que acabamos de ver). Cada una de las reglas clasificadoras es un cromosoma del algoritmo genético.
- **Variante Pittsburgh** (de S. F. Smith). Todo el conjunto de reglas clasificadoras es un cromosoma del algoritmo genético.

Estos algoritmos son un campo de investigación muy activo porque la verdad es que no funcionan bien, aunque deberían hacerlo. El problema está en el bucle de realimentación de la figura 139. Si no existiera, ambas variantes serían equivalentes y funcionarían muy bien. Sin embargo, su existencia implica, en la variante Michigan, la colaboración entre cromosomas dentro del algoritmo genético. Eso es difícil de lograr porque las reglas reproductivas hablan de competencia, no de colaboración (recordemos que se seleccionan los mejores con mayor probabilidad). Entonces, la colaboración no va a surgir, si surge es por casualidad y además es inestable.

La variante Pittsburgh lo soluciona haciendo que todas las reglas formen parte de un cromosoma. Los mejores cromosomas serán seleccionados con

mayor probabilidad, por lo que ahora sí hay una presión para que emerja la cooperación dentro de cada cromosoma. Pittsburgh funciona bien pero, a cambio, no puede ser entrenado en tiempo real con el entorno real ya que cada conjunto de reglas querrá hacer una cosa distinta con el entorno, incompatible y potencialmente irreversible. Por ejemplo, si el robot es un coche automático y el entorno es la carretera, un conjunto de reglas puede querer girar a la derecha, mientras que el otro puede querer girar a la izquierda. Con uno de los cromosomas va a haber un accidente, y entonces el otro cromosoma tampoco va a poder seguir conduciendo. Lo ideal es que el entorno sea *reseteable* a su estado inicial (lo que llamaremos "problemas clonables" en el capítulo "Inteligencia" del segundo libro), pero ello rara vez se da.

La variante más interesante es la de Michigan, aunque no funcione bien todavía. Esta variante modela muy bien lo que ocurre en entornos humanos, donde cada cromosoma sería una persona. Pensemos en un equipo de fútbol formado por once personas. Simplificando diríamos que el equipo gana si marca muchos goles. En un momento dado hay un delantero que marca gol. La recompensa del entorno la recibe él directamente en forma de premios, el apoyo de los *fans*, elogios, y contratos de publicidad, pero el mérito no es solo de él. Seguramente un mediocampista le hizo un pase perfecto que le sirvió para marcar el gol. A su vez, el mediocampista recibió el balón de un defensa que cortó una jugada peligrosa del equipo contrario. El algoritmo de brigada de baldes debería funcionar aquí, asignando a cada jugador una parte de la recompensa. Pero este algoritmo es inestable: puede asignar recompensas a jugadores que no hayan hecho nada importante, ni bueno ni malo, simplemente estaban allí para poner el pie. Y es inequitativo: cuanto más lejos en la cadena, menos recompensa se recibe. Quizás el mayor mérito fue del defensa su acción pues tomó a todos por sorpresa, y el mediocampista y el delantero lo tuvieron fácil, pero el defensa recibirá siempre poca recompensa, mientras que el delantero recibirá mucha.

Por otro lado, también se puede pensar que la recompensa real se la lleva todo el equipo, que gana o pierde partidos y que quizás llegue a ser campeón en su torneo. En este caso, nos podemos preguntar por el reparto de recompensas interno (el salario, básicamente): ¿cómo se hace? Hay muchos factores que inciden, como el prestigio, el apoyo de los *fans*, los comentarios de la prensa, las jugadas clave donde participó, siendo todos ellos bastante difusos y subjetivos. ¿Cómo evitar que a un equipo muy bueno entren jugadores que no quieran trabajar? Dawkins demuestra que la colaboración de grupo no es evolutivamente estable en un ambiente genético. Quedaría por ver en ambientes distintos al genético. Un club de fútbol no funciona como los genes porque los hijos de los mejores jugadores no entran al equipo directamente,

e incluso es más probable que se conviertan en ingenieros de software. Por ello tendríamos que considerar en qué ambientes y con qué estrategias se puede lograr que emerja la cooperación. Más adelante lo analizaremos en el capítulo "Teoría de Juegos".

El problema de la falta de convergencia de la variante Michigan es muy parecido al problema de la falta de convergencia en las redes neuronales cuando tienen demasiadas capas: es difícil atribuir la recompensa por un buen resultado a alguna capa concreta. Dado que en redes neuronales ese problema ya está solucionado con los *autoencoders* conectados en cascada, es bastante posible que se pueda diseñar una técnica similar para la variante de Michigan.

Programación evolutiva

La programación evolutiva[32] fue ideada por Lawrence J. Fogel en 1966. En aquella época él trabajaba con una población de solamente 2 cromosomas, posiblemente por las limitaciones de cómputo, y hacía la selección de forma determinista (entre el padre y el hijo, se quedaba con el mejor, borrando el otro y generando un nuevo hijo). Esto, como ya sabemos, conduce a óptimos locales, especialmente con poblaciones tan pequeñas, por lo que lo recomendable hoy día es usar poblaciones grandes y un método de selección probabilista como los presentados para los algoritmos genéticos.

Los cromosomas pueden ser cualquier cosa, y realmente él no hablaba de separar el espacio de genotipos y el de fenotipos, sino que trabajaba directamente en este último. Las aplicaciones típicas que muestra son para entrenamiento de redes neuronales y para diseño de máquinas de estados finitos.

No hay operador de cruce y, a cambio, hay varios de mutación. Lo importante de estos operadores es que produzcan pequeños cambios la mayoría de las veces y grandes cambios pocas veces. Esto es consistente con la propiedad que debe cumplir el operador de mutación. También produce simultáneamente búsqueda en profundidad y en anchura. Por último, suena muy bien ya que —como se ve en el capítulo de "Leyes de Potencias"— la mayoría de los objetos naturales y de sus problemas asociados tienen pocos grandes aspectos y muchos pequeños detalles.

A continuación veremos un ejemplo de diseño de una máquina de estados finitos para la predicción de una secuencia temporal. El proceso con una población de 2 máquinas es:

- Generar al azar una población de máquinas de estados finitos, donde tanto los símbolos de entrada como los de salida son el conjunto de

32 *Evolutionary Programming.*

símbolos de la secuencia temporal a predecir. Y los estados y sus transiciones son arbitrarios (al azar), así como el estado inicial.

- A cada máquina, hacerla recibir sucesivamente cada símbolo de entrada de la secuencia temporal para ver cómo predice el siguiente símbolo. Su salida en cada instante temporal discreto es la predicción del siguiente símbolo de la entrada. La aptitud de la máquina se calcula contando el número de aciertos.
- Se hace una selección de las mejores con mayor probabilidad (como decíamos, Fogel seleccionaba únicamente la mejor).
- A partir de las seleccionadas se generan otras máquinas hijas aplicándoles varias mutaciones. Las mutaciones posibles son:
  - Cambiar un símbolo de entrada.
  - Cambiar una transición de estado.
  - Cambiar (añadir o quitar) el número de estados.
  - Cambiar el estado inicial.
- Estas máquinas hijas se vuelven a inyectar a la población, borrando el mismo número de originales si queremos mantener un tamaño fijo.

Veamos un ejemplo: queremos predecir la producción en toneladas de un cultivo de tomates orgánico. Tenemos los datos de las últimas temporadas, que fueron 2,2,1,0,1,3,3,0,3,0,1. Entonces creamos al azar una población de máquinas de estados finitos. Una de ellas la podemos ver en la figura 141, que tiene 3 estados *{A,B,C}*, siendo *B* el estado inicial, y cuyas entradas y salidas pueden valer *{0,1,2,3}*.

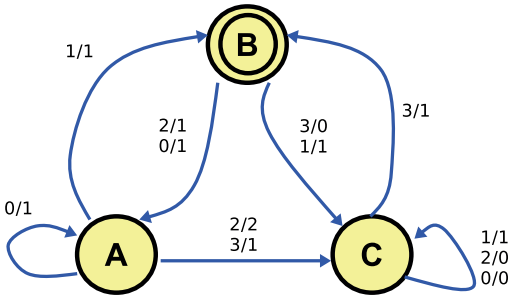

Figura 141. Máquina de estados finitos

Estos diagramas se leen así: si estando en el estado A llega una entrada 0, continúo en el estado A y emito una salida 1. Si estando en el estado A llega una entrada 1, paso al estado B y emito una salida 1. Si estando en el estado B llega una entrada 3, paso al estado C y emito una salida 0. Etcétera.

Si a esta máquina le inyectamos la secuencia de entrada conocida, obtendremos los resultados indicados en la tabla 14.

Tabla 14. Transiciones de la máquina de estados

Estado actual	B	A	C	C	C	C	B	C	C	B	A
Símbolo de entrada	2	2	1	0	1	3	3	0	3	0	1
Símbolo de salida		1	2	1	0	1	1	0	0	1	1
Aciertos		0	0	0	0	0	0	1	0	0	1

Esta tabla se lee así: estando en el estado B y recibiendo la entrada 2, pasa al estado A y genera la salida 1. Como la entrada siguiente es 2, entonces no ha acertado. Estando ahora en A y con entrada 2 se pasa a C generando la salida 2. Como la entrada siguiente es 1, tampoco ha acertado. Etcétera.

El total de aciertos de esta máquina es 2, por tanto esa es su aptitud, que sirve para realizar la selección.

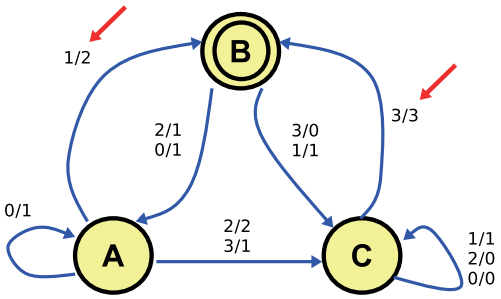

Figura 142. Máquina de estados finitos después de sufrir dos mutaciones

Supongamos que esa misma máquina fue seleccionada para la reproducción y que el azar decide realizar con ella dos mutaciones, en los símbolos de salida indicados con flechas rojas en la figura 142.

Entonces, para evaluarla se vuelven a inyectar en secuencia todos los símbolos de entrada, obteniéndose las transiciones que vemos en la tabla 15.

Tabla 15. Transiciones en la máquina de estados hija

Estado actual	B	A	C	C	C	C	B	C	C	B	A
Símbolo de entrada	2	2	1	0	1	3	3	0	3	0	1
Símbolo de salida		1	2	1	0	1	3	0	0	3	1
Aciertos		0	0	0	0	0	1	1	0	0	1

Podemos observar que ahora acertó en 3 símbolos de salida, por lo que su aptitud es 3. El resto del proceso es igual al de los algoritmos genéticos, realizando sucesivas generaciones hasta lograr una cantidad de aciertos aceptable. En ese momento podremos usar la mejor máquina de estados finitos de la población para predecir el futuro.

Con el paso del tiempo dispondremos de más datos para la secuencia de entrada, que se deberán usar para seguir generando mejores soluciones.

Estrategias evolutivas

Las estrategias evolutivas[33] fueron ideadas por Rechenberg, Schwefel y Bienert en Berlín en 1963-1972, tratando de diseñar un ala de avión en un túnel de viento, introduciendo cambios al azar sobre un ala que tenía varias bisagras para permitir rotaciones y cambio de su perfil. Esto se pudo generalizar para llevarlo actualmente a aplicaciones *software*:

- La población es un vector de cromosomas y cada cromosoma es un vector de genes, exactamente igual que en algoritmos genéticos. Pero aquí, cada gen es un número flotante.
- La reproducción se logra con mutación y cruce, que son distintos a los de los algoritmos genéticos:
  - La mutación se hace sumando ruido *gaussiano* a cada uno de los genes. Esto es una gran idea, porque significa añadir cambios pequeños muy frecuentemente y cambios grandes con poca frecuencia.
  - El cruce se hace promediando los genes respectivos.
- La desviación típica del ruido *gaussiano* se puede cambiar dinámicamente de tres maneras para mejorar la búsqueda:
  - Haciendo que decrezca en el tiempo. Esto tiene sentido porque al principio de la evolución es bueno explorar fuertemente el espacio de búsqueda (búsqueda en anchura), mientras que al final se requiere afinar las soluciones encontradas (búsqueda en profundidad).
  - De forma adaptativa. Cuando la frecuencia de las mutaciones que tienen éxito después de varias generaciones es alta se disminuye la desviación típica (con el objetivo de hacer búsqueda local). Y cuando es baja, se aumenta (con el objetivo de generar variedad y salir del óptimo local en que se encuentre el algoritmo).
  - Añadiendo al cromosoma más genes que representen la desviación típica de cada uno de los genes originales. Y dejar que la evolución se encargue de encontrar los valores más adecuados en cada momento.

En estas estrategias se introduce una nueva notación para explicar cómo es la población y la selección, de la forma siguiente:

- Se llama una estrategia (μ, λ) a aquella en la que μ padres generan λ hijos, los μ padres se descartan y los λ hijos compiten entre ellos.

33 *Evolutionary Strategies.*

- Se llama una estrategia (μ + λ) a aquella en la que μ padres generan λ hijos, y los μ padres y los λ hijos compiten entre ellos.

Los primeros trabajos fueron (1+1), seguramente por la baja potencia de cómputo de aquella época. Por ejemplo, si me dicen que hay una estrategia evolutiva de tipo (200+5) significa que tenemos una población de 200 individuos, que generan 5 hijos. Y de los 205 individuos se seleccionan los 200 mejores (de manera determinista) para formar la siguiente generación. Recordemos aquí de nuevo que, aunque históricamente algunos algoritmos hagan selección determinista, es más conveniente hacerla probabilista para escapar de óptimos locales.

Enfriamiento simulado

El enfriamiento simulado[34] fue ideado por Kirpatrick y Cerny en 1985. Está inspirado en la forma como los átomos van agregándose para pasar de líquido a sólido cristalino cuando se va bajando suavemente la temperatura. A pesar de su fundamento físico-químico, se le considera equivalente a una estrategia evolutiva de tipo (1+1).

La metáfora puede verse en la figura 143, donde los átomos "buscan" situarse en un lugar del retículo donde su energía potencial sea mínima. Pero pueden "equivocarse", dado que se depositan al azar. Las equivocaciones se corrigen gracias a la energía cinética de los otros átomos, que arrancarán a aquellos que estén mal situados.

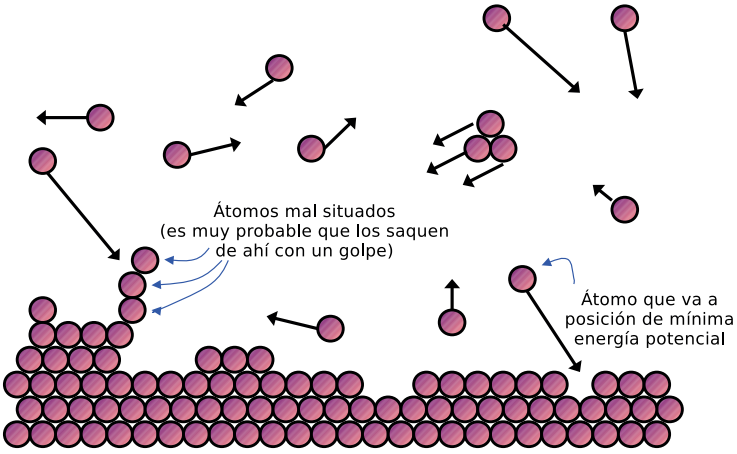

Figura 143. Fenómeno físico que usa como analogía el enfriamiento simulado

34 *Simulated Annealing*, también conocido como temple, cristalización, recocido o solidificación simulados.

Es importante ir bajando poco a poco la temperatura para disminuir la probabilidad de arrancar de su posición a los átomos que sí están bien situados, y forzar la solidificación de todos ellos en sus lugares de mínima energía potencial.

Entonces, el algoritmo trata de replicar este proceso en términos generales. La diferencia más importante respecto a otros algoritmos evolutivos es que ahora hay un parámetro llamado temperatura, que comienza en un valor alto y debe ir decrementándose hasta un valor mínimo. Y esa temperatura se usa en el proceso de selección: cuando la temperatura sea alta, se aceptarán con mayor probabilidad soluciones malas. Y conforme la temperatura disminuya, la probabilidad de aceptar soluciones malas también disminuye.

El algoritmo completo lo podemos ver en el diagrama de flujo de ejecución de la figura 144. Estos diagramas ya casi no se usan, pero para este caso resulta muy apropiado[35]. En color rojo vemos los procesos relacionados con el nuevo parámetro de la temperatura, que hay que explicar en más detalle.

El criterio de parada es similar al de los algoritmos genéticos. Puede ser:

- Si la aptitud llegó al 100% o a un valor aceptable para el usuario.
- Si se ha alcanzado el valor final de la temperatura.
- Si se han ejecutado un cierto número prefijado de iteraciones.
- Si se ha agotado el tiempo de cómputo disponible.

Además, hay que decidir una temperatura inicial ($T_{INICIAL}$) y una temperatura final (T_{FINAL}), así como una fórmula de enfriamiento.

- La temperatura inicial debe ser lo suficientemente alta para que la probabilidad de aceptar cualquier solución sea 1. Valores mucho más altos no mejoran el algoritmo, mientras que valores muy bajos lo empeoran, siendo 100 el valor típico.
- La temperatura final debe ser lo suficientemente baja para que el sistema pueda converger, siendo el valor típico de 0.1.

A su vez, el cambio de temperatura puede hacerse de varias formas:

- Enfriamiento geométrico. No se suele recomendar por ser demasiado rápido:

$$T \leftarrow \alpha^{*} T \; con \; \alpha \in [0.9,0.99] \qquad \text{Ec. 41}$$

- Enfriamiento Lundy & Mess. Es bastante bueno.

$$T \leftarrow \frac{T}{1+\beta^{*}T_N} \text{ con } T_N = \frac{T}{T_{INICIAL}} \; y \; \beta \in (0,1) \qquad \text{Ec. 42}$$

35 En muchos libros se habla de costo en vez de aptitud: mucha aptitud significa costo bajo.

- Enfriamiento Dowsland: es adaptativo, de modo que hace calentamiento en vez de enfriamiento, si la última solución fue rechazada. Con ello logra aumentar la diversidad de las soluciones cuando no parecen mejorar. A cambio, podría no terminar jamás:

$$T \leftarrow \begin{cases} \dfrac{T}{1+\beta * T_N} & \text{si la nueva solución se aceptó} \\ \dfrac{T}{1-\alpha * T_N} & \text{si la nueva solución se rechazó} \end{cases} \qquad \text{Ec. 43}$$

con valores típicos $\alpha \approx 0.2$ y $\beta \approx 0.02$

Por otro lado, el criterio de aceptación de la nueva solución es:

- Si la nueva solución es mejor que la antigua ($APTITUD_{NUEVA} > APTITUD_{ACTUAL}$), se acepta la nueva.
- Si es peor, se genera un número aleatorio (*rand*) entre 0 y 1 y solo se acepta la nueva solución si se cumple que:

$$rand < e^{\frac{APTITUD_{ANTIGUA} - APTITUD_{ACTUAL}}{T}} \qquad \text{Ec. 44}$$

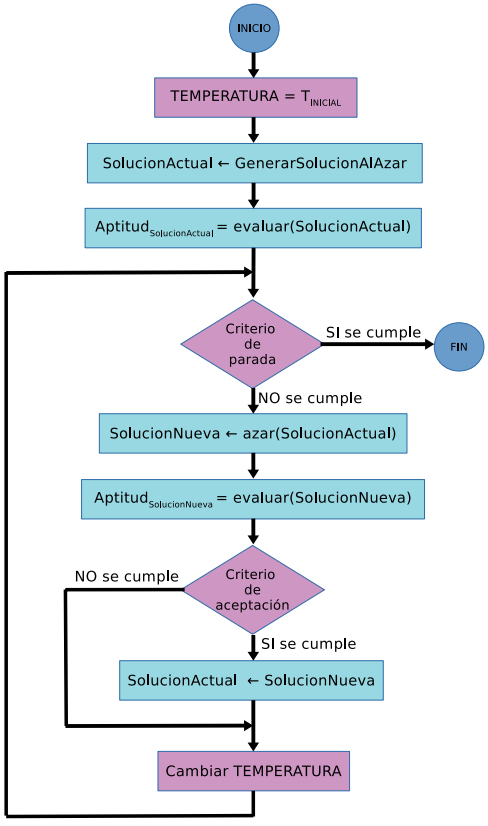

Figura 144. Diagrama de flujo de ejecución del enfriamiento simulado

Si analizamos esta fórmula (que viene dada por analogía con el proceso físico de solidificación[36]), vemos que si la temperatura es alta entonces hay una alta probabilidad de aceptar malas soluciones. Y si la temperatura es baja, la probabilidad también es baja. La probabilidad de aceptación de una solución disminuye cuanto peor sea la aptitud de esa solución. Todo es razonable y adecuado respecto a lo que hemos visto en los demás algoritmos evolutivos.

La nueva solución se genera a partir de la solución actual introduciendo algún pequeño cambio al azar y equivale a las mutaciones en los algoritmos evolutivos.

Tanto en enfriamiento simulado como en estrategias evolutivas, bajo ciertas condiciones está garantizada teóricamente la convergencia al óptimo, aunque las garantías son probabilistas, y se cumplen cuando el tiempo de búsqueda tiende a infinito, es decir, hay convergencia asintótica. De hecho, estos dos algoritmos son tan similares (a pesar de estar uno inspirado en la física y el otro en la biología), que se dice que el enfriamiento simulado es una estrategia evolutiva de tipo (1+1).

Evolución diferencial

La evolución diferencial[37] fue creada por Kenneth Price y Rainer Storn en 1997. Como en cualquier algoritmo evolutivo, hay una población de *NC* cromosomas, y cada cromosoma es un vector de *N* números flotantes que modelan las posibles soluciones. En cada generación se intentan reproducir todos los cromosomas. La principal diferencia respecto a otros algoritmos evolutivos es que no hay mutación y el operador de cruce es de la siguiente manera: para cada cromosoma x_i con $i=\{1,2...NC\}$ de la población se eligen al azar otros tres cromosomas x_a, x_b, x_c distintos entre sí y distintos a x_i. Se crea un nuevo cromosoma $x_z=x_a+F^*(x_b-x_c)$ siendo *F* una constante (las operaciones matemáticas se hacen como si cada cromosoma fuera un vector, usando sus genes como componentes). Y se hace un cruce uniforme entre x_i y x_z. Este cruce no es equiprobable, sino que la probabilidad de elegir un gen de x_z es p_{CR} y la de elegir un gen de x_i es $(1-p_{CR})$.

36 En Física, $p=e^{-\Delta E/KT}$ siendo p la probabilidad de que un cambio de estados se consolide (se acepte), ΔE la diferencia de energías entre el estado nuevo y el actual (que, en nuestra fórmula, se cambia de signo para reflejar que las aptitudes son lo contrario que las energías potenciales de los estados físicos), K es la constante de Boltzman (que no tiene interés en un algoritmo como este) y T es la temperatura absoluta a la que se encuentra el material que se está enfriando.

37 *Differencial Evolution*.

Por último, si la aptitud del nuevo cromosoma así fabricado es mejor que la del original x_i, se elimina x_i y se añade el nuevo a la población.

Este algoritmo depende de 3 constantes que hay que ajustar, *NC*, *F* y p_{CR}. Los valores típicos son:

- *NC= 10*n*
- *F= 0.8* Otra táctica es seleccionar en cada generación un valor de F al azar en el intervalo *[0.5, 1.0]*.
- *pCR= 0.9*

Al no haber mutación, no hay quien lo rescate si converge a un óptimo local, cuando todos los cromosomas sean similares. En cualquier caso, no cuesta nada añadir mutación, así no lo hayan planteado los autores del algoritmo. De la misma manera, en vez de hacer una selección determinista (elegir el mejor entre el padre y el hijo), se puede hacer probabilista.

La idea principal de la evolución diferencial es generar cromosomas variando los genes donde todavía no ha habido convergencia. Ello se logra eligiendo dos cromosomas al azar y restándolos como si fueran un vector. La resta dará cero en las componentes del vector (genes) que tengan valores muy parecidos, es decir, donde los alelos ya hayan convergido.

Algoritmo genético híbrido de Taguchi

El algoritmo genético híbrido de Taguchi38 fue creado por Jinn-Tsong Tsai, Tung-Kuan Liu, y Jyh-Horng Chou en 2004, aunque tuvo otros antecesores menos eficientes y a partir de él se han creado otras variantes, como el de Yang en 2013. Está basado en el método Taguchi que se emplea en control de calidad y optimización de procesos. Cuando el problema tiene pocas variables el algoritmo genético básico funciona bien, aunque con muchas variables es ineficiente. Para mejorarlo se puede usar este otro tipo de algoritmos cuya principal diferencia es la forma de hacer el cruce, usando dos nuevos conceptos: las matrices ortogonales de Taguchi y la relación señal-ruido (*SNR*).

La mayoría de los cruces al azar van a dar resultados muy pobres, especialmente cuando el cromosoma es muy largo. Entonces lo que se hace es una serie de experimentos con cruces y a partir de ellos, usando como criterio la *SNR*, se determina cuánto contribuye cada gen a la aptitud del cromosoma resultante. De este modo se elige el mejor de los cruces posibles.

$$SNR = (a_j - a_{MEJOR})^2 \qquad \text{Ec. 45}$$

38 Hybrid Taguchi-Genetic Algorithm.

Siendo a_j la aptitud del cromosoma en el j-ésimo experimento y a_{MEJOR} la mejor aptitud conocida hasta el momento. Hay otras fórmulas alternativas, pero esta es la más razonable, pues no presupone nada sobre la función a optimizar.

En función del número de genes del cromosoma (n) se fabrica una matriz llamada "latina" de tamaño n+1 filas por n columnas, que se identifica con la notación:

$$L_{n+1}(Q^n) \qquad \text{Ec. 46}$$

Las matrices son ortogonales en el sentido de que sus columnas son linealmente independientes. Aquí no se va a mostrar cómo se construyen las matrices latinas, pero se puede encontrar la forma de hacerlo en el trabajo de grado de EVALAB de los estudiantes Cristhian Fuertes y Óscar Tigreros (2017), donde también incluyen una comparación de estos algoritmos respecto a los tradicionales. En el trabajo emplean BDD como metodología de desarrollo, usando *Cucumber* para las pruebas.

Por ejemplo, si tenemos 7 genes se requieren 7 columnas de factores, y la matriz latina es $L_8(2^7)$ que se muestra en la figura 145.

Experimento número (j)	Factores (i)						
	1	2	3	4	5	6	7
1	L	L	L	L	L	L	L
2	L	L	L	M	M	M	M
3	L	M	M	L	L	M	M
4	L	M	M	M	M	L	L
5	M	L	M	L	M	L	M
6	M	L	M	M	L	M	L
7	M	M	L	L	M	M	L
8	M	M	L	M	L	L	M

Figura 145. Matriz latina para n=7

Donde:

- $Q=2$ es el número de cromosomas a cruzar. En algoritmos genéticos es siempre 2, y los vamos a llamar cromosoma L y cromosoma M.
- n es el número de genes del cromosoma (las variables que intervienen en la función a optimizar), y tiene que tener la forma $n=2^k$-1 siendo k un número natural. Si el número de genes no coincide con ningún valor 2^k-1, entonces se elige el k inmediatamente mayor y se rellena el cromosoma con genes inútiles. O, lo que es equivalente, se eliminan columnas de la matriz latina (da igual las que se eliminen, pues todas son ortogonales entre sí). En la figura anterior n=7.
- n+1 es el número de experimentos de cruce a realizar.

Las matrices latinas que se usan son $L_8(2^7)$, $L_{16}(2^{15})$, $L_{32}(2^{31})$, $L_{64}(2^{63})$, $L_{128}(2^{127})$, $L_{256}(2^{255})$, $L_{512}(2^{511})$, $L_{1024}(2^{1023})$ y sucesivas, para problemas de hasta 7, 15, 31, 63, 127, 255, 511 y 1023 genes, respectivamente.

Supongamos que la función a minimizar sea la siguiente (es importante aclarar que nosotros sabemos que su mínimo está en $x_i=0\ \forall i$, pero el algoritmo no lo sabe):

$$f(x) = \Sigma_{i=1}^{7} x_i \qquad \text{Ec. 47}$$

Y que los cromosomas seleccionados para cruzar son los de la figura 146, donde las variables (los genes) son números enteros.

	Genes (i)						
	1	2	3	4	5	6	7
Cromosoma L	2	14	0	8	7	5	4
Cromosoma M	4	5	12	6	8	6	5

Figura 146. Cromosomas que van a cruzarse

Entonces deben realizarse los experimentos de cruce indicados en la figura 145, cuyo resultado podemos ver en la figura 147.

Experimento número (j)	Genes (i)						
	1	2	3	4	5	6	7
1	2	14	0	8	7	5	4
2	2	14	0	6	8	6	5
3	2	5	12	8	7	6	5
4	2	5	12	6	8	5	4
5	4	14	12	8	8	5	5
6	4	14	12	6	7	6	4
7	4	5	0	8	8	6	4
8	4	5	0	6	7	5	5

Figura 147. Resultado de los 8 experimentos de cruce

Después se calculan las aptitudes del cromosoma resultante en cada experimento (a_j) y los correspondientes SNR_j (Figura 148). Vamos a suponer que la mejor aptitud que se tenía hasta este momento es $a_{MEJOR}=39$.

$$E_{ik} = \sum_{\forall j:\ MatrizLatina\ ij = k} SNR_j \qquad \text{Ec. 48}$$

Es decir, para cada gen i se hallan dos coeficientes E_{jL} y $E_{jM.}$ El primero con la suma de los SNR_j donde intervenga el cromosoma L, y el segundo donde intervenga el cromosoma M (Figura 149).

Experimento	Genes (i)							Aptitud	
número (j)	1	2	3	4	5	6	7	a_i	SNR_i
1	2	14	0	8	7	5	4	40	1
2	2	14	0	6	8	6	5	41	4
3	2	5	12	8	7	6	5	45	36
4	2	5	12	6	8	5	4	42	9
5	4	14	12	8	8	5	5	56	289
6	4	14	12	6	7	6	4	53	196
7	4	5	0	8	8	6	4	35	16
8	4	5	0	6	7	5	5	32	49

Figura 148. Cálculo de la aptitud y el SNR de cada experimento (suponiendo a_{MEJOR}=39)

	E_{ik}						
Cromosoma (k)	1	2	3	4	5	6	7
L	50	490	70	342	282	348	222
M	550	110	530	258	318	252	378

Figura 149. Coeficientes de sensibilidad de cada gen a cada cromosoma

Por ejemplo, E_{5M} es el efecto que tiene el cromosoma *M* en los SNR del gen 5, y se calcula así:

$$E_{5M} = \Sigma_{\forall j:\, MatrizLatina\ 5j=M}\ SNR_j = SNR_2+SNR_4+SNR_5+SNR_7 = 4+9+289+16 = 318 \quad \text{Ec. 49}$$

Por último, para cada gen se elige cuál es el coeficiente mayor (si se desea maximizar la función) o menor (si se desea minimizarla) como se ve en la figura 150. Y en función de ello se determina si el gen óptimo proviene del cromosoma L o del M (Figura 151) para generar el cromosoma óptimo como resultado final del cruce. Allí también se ha calculado su aptitud para que veamos que, efectivamente, se produce una mejora.

	E_{ik}						
Cromosoma (k)	1	2	3	4	5	6	7
L	50	490	70	342	282	348	222
M	550	110	530	258	318	252	378
Min $\{E_{jL}, E_{jM}\}$	50(L)	110(M)	70(L)	258(M)	282(L)	252(M)	222(L)
Máx $\{E_{jL}, E_{jM}\}$	550(M)	490(L)	530(L)	342(L)	318(M)	348(L)	378(M)

Figura 150. Minimizar o maximizar

	Genes (i)							
	1	2	3	4	5	6	7	Aptitud
Cromosoma óptimo (minimizando)	2	5	0	6	7	6	4	30
Cromosoma óptimo (maximizando)	4	14	12	8	8	5	5	56

Figura 151. Cromosoma óptimo

Como resultado de ello se tiene que para 7 genes se han realizado solamente 8 experimentos de cruce con los que se ha podido determinar el cruce óptimo. Teniendo en cuenta que existen 2^7=128 cruces posibles, esto representa una mejoría sustancial.

Otra variante sobre este HTGA es el CCHTGA[39] propuesta por Poorjandaghi en el 2014, donde en cada generación los cromosomas se dividen al azar en trozos más o menos iguales (de un tamaño elegido al azar), y todos los subcromosomas homólogos forman parte de una subpoblación que se intenta maximizar (o minimizar) por separado usando un HTGA. Después se juntan los subcromosomas resultantes para formar de nuevo los cromosomas con la estructura original. Esto hace que el espacio de búsqueda disminuya, y con ello el tiempo de ejecución[40].

Programación genética

La programación genética (*genetic programming*) fue ideada por John Koza, alumno de Holland, en 1990. Koza publicó una serie de tres libros que se encuentran en la bibliografía, pero que considero muy repetitivos a excepción de las primeras páginas donde explica este algoritmo.

En la programación genética, el diagrama de flujo de datos es el mismo que el de los algoritmos genéticos, y la principal diferencia es que el cromosoma contiene un programa ejecutable, en forma de árbol sintáctico. La mutación y el cruce cambian un poco para adaptarse a ello, y la evaluación de la aptitud pasa por ejecutar el programa representado en el cromosoma.

De modo que usando esta técnica podemos dejar que la evolución escriba programas de manera automática, sin intervención humana. Realmente no son programas muy largos. Son más bien sentencias. Y se requiere mucha potencia de cómputo para poder evaluarlas, por ejemplo, un cluster de computadores.

Veamos los detalles:

39 *Cooperative Coevolutionary Hybrid Taguchi Genetic Algorithm.*

40 En el siguiente enlace se puede encontrar una hoja de cálculo para hacer pruebas con este algoritmo: https://goo.gl/EB4gTq

Koza utilizó el lenguaje LISP para construir los cromosomas, aunque ciertamente se puede hacer con cualquier otro lenguaje. Tiene ventajas usar un lenguaje interpretado, pues así es más fácil evaluar el cromosoma, llamando al intérprete para que lo ejecute.

Como el objetivo es construir un programa que cumpla con ciertos requerimientos, lo primero que se debe hacer es definir cuáles van a ser los datos de entrada de ese programa (llamados terminales) y cuáles las operaciones permitidas sobre esos datos (llamadas funciones). En los terminales hay que incluir también las constantes que se puedan requerir (como π, e y 46) así como las funciones que entreguen salidas pero no reciban entradas como por ejemplo ocurre en un robot con los sensores de distancia a obstáculos.

Pongamos un ejemplo sencillo: queremos fabricar un sistema de alarma para la casa. Tenemos como entradas tres datos booleanos {*D0, D1, D2*} que representan si la puerta está abierta, si la ventana está abierta y si el propietario está en casa. Queremos que la salida sea una alarma, que indique sonoramente una condición de riesgo (típicamente que haya una ventana o puerta abierta cuando el propietario no está en casa). Como los datos son booleanos, es razonable asumir que las funciones también deben serlo. Definamos entonces:

- Conjunto de terminales = *{D0, D1, D2}*
- Conjunto de funciones = *{AND, OR, NOT}*

En este escenario, una posible expresión simbólica podría ser *(AND (OR D1 D0) (NOT (AND D0 D0)))* cuyo cromosoma vemos en la figura 152 Obsérvese que los terminales están en las hojas del árbol, y que las funciones son los nodos que tienen arcos hacia abajo (que son sus argumentos de entrada).

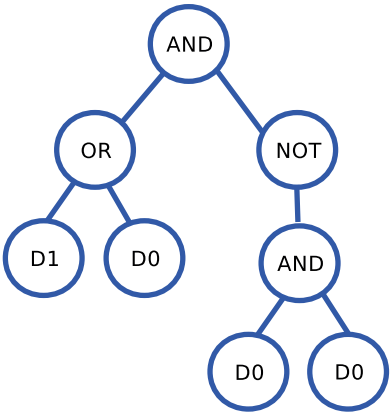

Figura 152. Ejemplo de cromosoma en un programa genético

Al igual que con los algoritmos genéticos, aquí también deben cumplirse las condiciones de completitud y cerradura. Pero ahora, la cerradura tiene nuevas implicaciones, pues se requiere que cada una de las funciones sea capaz de recibir como argumento cualquier dato (tanto en valor como en tipo) que pueda retornar cualquier otra función. Por ejemplo, el álgebra de Boole es cerrada ya que cualquier función recibe como entrada valores booleanos y retorna como salida valores booleanos:

- Conjunto de funciones = *{AND, OR, NOT}*
- Conjunto de terminales = *{true, false}*

Lo que no ocurre con la aritmética de números flotantes, ya que la división por cero y la raíz cuadrada de números negativos generan resultados que en la mayoría de lenguajes de programación no son válidos y no se pueden usar como entradas para otras funciones. Muchas veces lanzan excepciones y abortan la ejecución del programa. Para evitarlo se aconseja redefinir estas funciones retornando algún valor arbitrario que evite la excepción (ver un ejemplo en la figura 153).

```
def dividir(numerador, denominador)
   if denominador == 0
      return 1000000
   else
      return numerador / denominador
   end
end


def sqrt2(numero)
   return sqrt(abs(numero))
end
```

Figura 153. Operaciones aritméticas protegidas

Ocurre algo similar cuando se necesita usar operaciones aritméticas y lógicas a la vez. En estos casos, en vez de utilizar {*false, true*} se pueden definir arbitrariamente como equivalentes a {*=0, !=0*}, por ejemplo. Así, los resultados booleanos pueden interoperar con funciones aritméticas y viceversa (Figura 154).

De este modo, el cromosoma de la figura 155 es válido y ejecutable. Hay que tener en cuenta que esta mezcla de operaciones va a surgir inevitablemente, debido a la naturaleza aleatoria de los operadores de mutación y cruce.

```
def mayorQue(a, b)
   if a > b
      return 1
   else
      return 0
   end
end


def and(a, b)
   If a != 0 && b != 0
      return 1
   else
      return 0
   end
end
```

Figura 154. Algunas operaciones aritmético-logicas interoperables

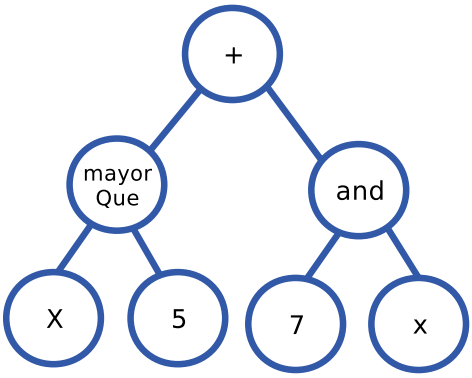

Figura 155. Cromosoma válido, con mezcla de operaciones aritméticas y lógicas

La población inicial de cromosomas se crea al azar. Para cada cromosoma, se elige al azar entre funciones y terminales y se van llenando los arcos hacia abajo con más funciones o terminales elegidos también al azar (Figura 156).

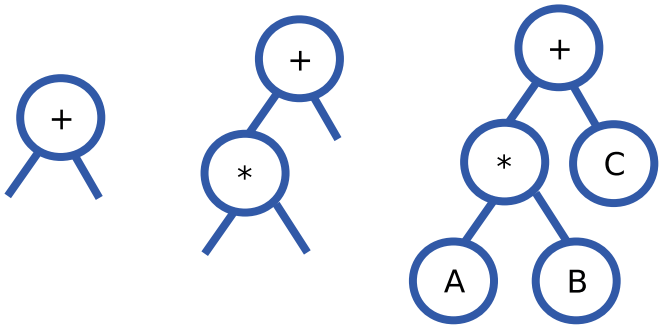

Figura 156. Creación de un cromosoma al azar

Hay tres estrategias que nos permiten completar un cromosoma:

- **Total** (*full*)**.** Crear los cromosomas con todos los caminos de longitud *L*. Para ello, en los nodos intermedios se seleccionan aleatoriamente solo funciones, y cuando se llega a la longitud *L* deseada, se seleccionan aleatoriamente solo terminales.
- **Creciente (***grow***).** Crear los cromosomas con caminos de longitud variable, menor que *L*. Para ello, en los nodos intermedios se seleccionan aleatoriamente funciones y terminales, y cuando se llega a la longitud *L* deseada, se seleccionan aleatoriamente solo terminales.
- **Progresivo** (*ramped half and half*)**.** Crear un número igual de cromosomas con una profundidad especificada entre 2 y el máximo *L*. Por ejemplo, si se desea L=6, el 20% de la población tendrá profundidad 2, el 20% 3, el 20% 4, el 20% 5 y el 20% tendrá profundidad 6. Este método es el que genera más variedad.

La aptitud de un programa se calcula ejecutándolo y midiendo lo mal o bien que cumple con sus objetivos. Ello indica el punto débil de la programación genética: requiere mucha potencia de cómputo.

Los operadores de reproducción son:

- Cruce. Intercambiar dos subárboles al azar de dos individuos (ver ejemplo en la figura 157).
- Mutación. Eliminar al azar un subárbol y generar allí otro al azar (ver ejemplo en la figura 158).
- Permutación. Similar al cruce, pero dentro del mismo individuo.
- Edición. Eliminar bloques inútiles tales como una expresión multiplicada por cero (sustituirla por cero) y las operaciones con constantes, sustituirlas por el resultado. Koza propone hacer esta edición de los cromosomas a mano, pero eso es laborioso. Por otro lado, hacerla automáticamente es imposible en el caso general, y solo puede aspirarse a hacer en casos sencillos como los mencionados. La única utilidad de la edición es la reducción del tiempo de cómputo al calcular la aptitud ejecutando el cromosoma, por lo que no es realmente algo muy importante hoy día.
- Encapsulación. Seleccionar un trozo de código al azar (un subárbol del cromosoma) y encapsularlo en una función, que habrá que definir.

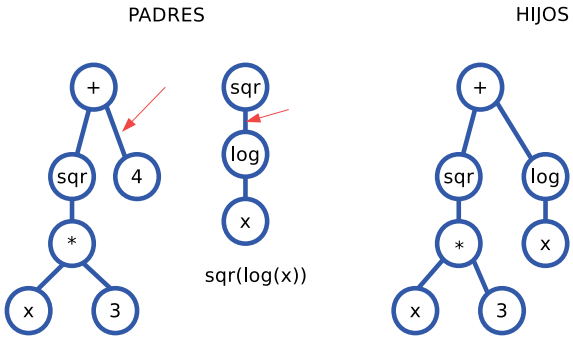

Figura 157. Cruce. Las flechas indican los puntos de corte elegidos al azar

El usuario debe saber cuáles funciones son más apropiadas para un determinado problema, y lo mismo con los terminales. Si hay funciones o terminales superfluos aumenta mucho el tiempo para encontrar la solución. Y si faltan, no encontrará la solución. Las constantes (como π, 1, 2 y 46) se pueden dar como terminales o también se pueden generar al azar.

Algo interesante es que durante la evolución suelen aparecer muchos intrones[41], es decir, fragmentos de código que no hacen nada, por ejemplo x-x, 0*(log(x)+x*3). Es interesante porque en la evolución biológica también aparecen intrones, aunque en la programación genética son simplemente un desperdicio de memoria y de tiempo de ejecución.

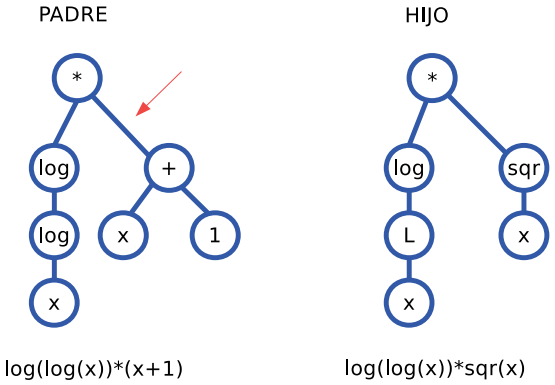

Figura 158. Mutación. La flecha indica el punto de corte elegido al azar

41 Término tomado de la biología: un intrón es una región del ADN que no se expresa, que no sirve para nada. Lo opuesto es un exón, una región del ADN que se expresa, típicamente para participar en la fabricación de proteínas. Hay varias teorías que tratan de explicar por qué existen los intrones si aparentemente no realizan ninguna función. Una de ellas dice que antes eran exones que dejaron de tener importancia. Otra, que ayudan a realizar cambios evolutivos a largo plazo. La más reciente explica que ayudan indirectamente en la expresión de los exones. Pero quizás lo más espectacular para nosotros es que también aparecen cuando hacemos evolución de programas.

Las aplicaciones de la programación genética son muy numerosas:

- Integración, derivación e inversión simbólicas. Predicción de secuencias. Regresión simbólica. Todas son similares. Si tenemos una serie temporal (por ejemplo, los litros de lluvia por metro cuadrado en Cali, día a día) y queremos predecir qué ocurrirá mañana, lo que muchas veces se hace es tratar de interpolar linealmente los puntos (por "mínimos cuadrados") para ver por dónde va a pasar esa recta en días futuros. Pero el fenómeno físico puede no corresponderse con una ecuación lineal, de modo que la predicción puede fallar bastante. Es mejor plantear un cromosoma donde puedan salir otro tipo de ecuaciones (polinómicas, con funciones trigonométricas o logaritmos) y dejar que la evolución encuentre la más adecuada. La aptitud de cada cromosoma se calcula como la sumatoria de los errores de ajuste (cambiado de signo) de la ecuación del cromosoma respecto a todos los puntos conocidos. En la figura 159 podemos ver que el cromosoma de color púrpura implementa la ecuación $f(t)=t*log(t+1)+10*cos(100*t)+5$, que es la que tiene el menor error cuadrático. La interpolación lineal (en verde) es bastante mala en este ejemplo.
- Descubrimiento de leyes a partir de datos empíricos. Por ejemplo, las tres leyes de Kepler han sido redescubiertas por estos algoritmos a partir del conjunto de datos astronómicos obtenidos por Tycho Brahe.
- Compresión de datos con pérdidas (imágenes).
- Diseño en ingeniería civil, arquitectura, muebles, arte.
- Estrategias en robots (hormigas artificiales).
- Diseño de circuitos digitales.
- Comportamiento emergente (hormigas artificiales, agentes, robots).
- Diseño de controladores y automatismos (péndulo invertido, aparcar un camión largo).
- Diseño de estrategias óptimas en juegos (*Pac-Man, Otelo,* damas, y muchos otros).
- Generación de secuencias seudoaleatorias.
- Clasificación de datos (*Clustering*, *Data Mining*).

Para concluir, es bueno mostrar los problemas que se presentan en programación genética con el fin de entender las dos variantes expuestas en los siguientes capítulos.

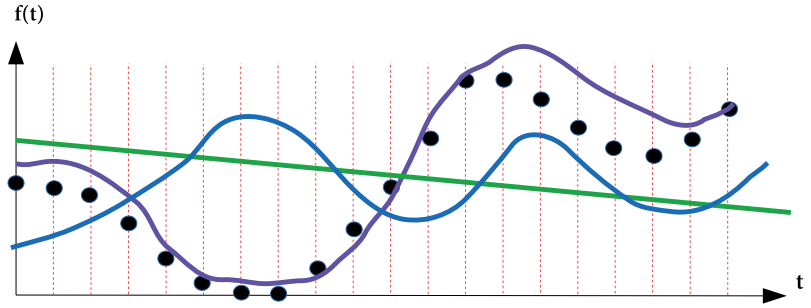

Figura 159. Interpolación simbólica de una secuencia de puntos (en negro). Se muestran dos cromosomas: en púrpura el que se aproxima más a la solución, y en azul otro que no se acerca tanto. En verde está el resultado de interpolar linealmente

- Manejar árboles es engorroso.
- Se requiere una gran capacidad de cómputo debido a que hay que ejecutar los programas que producen los cromosomas, y puede haber miles de cromosomas y millones de generaciones.
- La ejecución de código arbitrario en un cromosoma puede producir problemas de seguridad. Para evitarlo, hay que crear un ambiente cerrado adecuado (un *sandbox*) donde el malfuncionamiento de un programa procedente de un cromosoma no afecte a nadie más. Además, es recomendable lanzar a la vez un hilo con una temporización, para que cuando expire se mate el programa, pues probablemente se haya quedado atascado en algún bucle. Si eso ocurre se le asigna la aptitud más baja.
- La mutación casi nunca produce cambios pequeños, lo cual no es bueno. Viendo el ejemplo de la figura 158, si el arco seleccionado al azar está muy arriba del árbol (muy cerca del nodo raíz) la mutación hará un cambio muy grande. Y si el arco seleccionado está cerca de las hojas, la mutación realizará un cambio muy pequeño. Aunque, de todos modos, sí cumple la propiedad esencial de que produciendo infinitas mutaciones a cualquier cromosoma se generarán todos los cromosomas posibles.

Evolución gramatical

La evolución gramatical[42] fue ideada por Michael O'Neill, J. J. Collins y Conor Ryan en 1997. Sirve para generar automáticamente programas, pero los cromosomas ya no son árboles sino vectores de enteros, como en los algoritmos genéticos. Su expresión en programas se hace por medio de una gramática BNF (*Backus-Naur Form*), en la que hay que definir:

42 Grammatical Evolution.

- Un conjunto de símbolos terminales (+, -, *1,2,3...*) y un conjunto de símbolos no-terminales.
- Unas reglas de producción que permiten generar los símbolos no-terminales en función de ellos mismos y de los terminales.

Una gramática puede expresarse por medio de una tupla {*N,T,P,S*} donde N es el conjunto de símbolos no-terminales, T el conjunto de símbolos terminales, P el conjunto de reglas de producción que mapean los elementos de N en T, y S es el símbolo inicial, que debe ser miembro de N.

Es habitual no usar toda una especificación BNF de un lenguaje (pues suele ser extremadamente larga), sino solo un subconjunto de interés.

La traducción del cromosoma al programa se consigue leyendo secuencialmente los genes (números enteros) y usándolos para tomar decisiones sobre la regla de producción a elegir en la gramática BNF. En la figura 160 hay un ejemplo de regla que tiene 3 opciones numeradas a la derecha como 0, 1 y 2.

```
<sentencia> :=   if (<expresion>) { <sentencia>; } else { <sentencia>; }   [0]
                 |      <sentencia> ; <sentencia>                           [1]
                 |      <variable> = <expresion>;                           [2]
```

Figura 160. Ejemplo de regla de producción en BNF

La regla 0 nos dice que una sentencia puede ser un *if-else* con sus tres correspondientes sentencias (la condición, lo que se hace si es verdad, lo que se hace si es falso). La regla 1 nos dice que una sentencia puede expandirse a dos sentencias consecutivas, una detrás de otra. La regla 2 nos dice que una sentencia puede estar conformada por una expresión cuyo resultado se asigna a una variable. En las tres reglas, todos los tags encerrados entre < > son no-terminales, que deben expandirse subsecuentemente.

Para elegir cuál de las 3 reglas se selecciona en el ejemplo, debemos extraer un gen del cromosoma (que va a ser un número entre 0 y 255) y, ya que hay 3 posibilidades, calculamos su módulo 3. El resultado puede ser 0, 1 o 2 y, según salga, se elegirá una de las tres reglas de producción.

Y así sucesivamente hasta lograr un programa completo (con todas las reglas convertidas a nodos terminales) o hasta que se acaben los genes.

Si se acaban los genes y todavía quedan símbolos no-terminales sin resolver, hay tres opciones:

- Se puede volver a comenzar, extrayendo genes desde el principio del cromosoma, reusándolos. Esta no suele ser una idea recomendable porque se crean dependencias artificiales entre distintas partes del programa.

- Se dice que el programa resultante no es ejecutable, y se le asigna el menor valor de adaptación posible.
- Se alarga el cromosoma, para poder disponer de más genes. Enseguida veremos cómo se hace, pero solo se logra una solución parcial pues como las reglas de producción se eligen al azar, potencialmente podría ocurrir que sigan saliendo reglas con no-terminales y, como todo tiene un límite, al final habrá que aplicar la opción anterior.

En estas gramáticas evolutivas se han introducido dos nuevos operadores, aparte de la mutación y el cruce que son iguales que en los algoritmos genéticos:

- **Duplicación.** Se seleccionan al azar unos cuantos genes consecutivos y se copian al final del cromosoma. Sirve para resolver parcialmente el problema mencionado.
- **Poda.** Si un cromosoma no necesitó usar todos sus genes para generar un programa, se aplica este operador con una cierta probabilidad. La poda consiste en quitar los genes no usados. De esta manera se eliminan los intrones (genes que no se expresan). La única ventaja de este operador es ahorrar memoria.

Debido a estos operadores, cada cromosoma puede tener una longitud distinta. Y los genes dejan de tener significado posicional. Esto hay que tenerlo en cuenta para hacer correctamente la mutación y el cruce. Por ejemplo, si se cruza un cromosoma de 100 genes (el padre) con otro de 120 (la madre), el resultado será un cromosoma de 120 genes donde los primeros 100 proceden al azar del padre o de la madre, y los últimos 20 proceden directamente de la madre.

A continuación podemos ver un ejemplo más realista donde la gramática se muestra en la figura 161, un posible cromosoma en la figura 162 y la expresión del cromosoma en un programa en la figura 163.

En las gramáticas evolutivas hay problemas de dependencias similares a los de la programación genética: cuánto más lejos de la raíz está un gen, más probable es que su expresión se vea alterada por una mutación, y potencialmente pueden salir cromosomas incompletos a los que hay que dar aptitud nula. Como ventajas se puede mencionar que los cromosomas son más sencillos de almacenar y manejar —pues son vectores de enteros en vez de árboles—, y que la mutación y el cruce son sencillos.

```
N = {expresion, op, pre_op, var}
T = {sin, cos, tan, log, +, -, /, *, X, () }
S = <expresion>
P =
(1) <expresion> ::=     <expresion> <op> <expresion>                [0]
                    |   ( <expresion> <op> <expresion> )            [1]
                    |   <pre_op> ( <expresion> )                    [2]
                    |   <var>                                       [3]

(2) <op> ::=            +           [0]
                    |   -           [1]
                    |   *           [2]
                    |   /           [3]
                    |   %           [4]

(3) <pre_op> ::=        sin         [0]
                    |   cos         [1]
                    |   log         [2]

(4) <var> ::=           X           [0]
                    |   Y           [1]
```

Figura 161. Ejemplo de una gramática BNF

6	34	12	15	7	7	0	36

Figura 162. Ejemplo de un cromosoma para la gramática BNF

```
Iniciamos con S=                    <expresion>
6%4=2  => regla [2]:              <pre_op> ( <expresion> )
34%3=1  => regla [1]:               cos( <expresion> )
12%4=0 => regla [0]:      cos(<expresion> <op> <expresion>)
15%4=3 => regla [3]:            cos(<var> <op> <expresion>)
7%5=2 => regla [2]:               cos(<var> * <expresion>)
7%4=3 => regla [3]:                cos(<var> * <var>)
0%2=0 => regla [0]:                cos( X * <var>)
36%2=0 => regla [0]:                 cos( X * X )
```

Figura 163. Expresión del cromosoma

Programación por expresión genética

La programación por expresión genética (*gene expression programming*) fue creada por Cándida Ferreira en el año 2000. Resuelve muchos de los problemas anteriores y es, posiblemente, el algoritmo evolutivo más general.

Una advertencia: los cromosomas propuestos por Ferreira son mucho más complejos que antes, y la notación de gen es un poco distinta a lo que hemos visto en el resto del libro. El nuevo gen tiene estructura interna y el nuevo cromosoma también. Veremos todo esto en detalle a continuación. El resto (población, evaluación de aptitud, selección, reproducción y reemplazo) es similar a los algoritmos genéticos.

El cromosoma sigue siendo un vector lineal de genes y cada gen codifica una sentencia de un programa usando símbolos terminales y no-terminales. El último gen es ligeramente distinto pues sirve para agrupar los anteriores

en un único programa. Para ello usa símbolos no-terminales y los genes anteriores como terminales.

Todos los genes tienen la misma longitud y cada gen está formado por tres dominios: cabeza, cola e índices de constantes. En la cabeza puede haber terminales y no-terminales mientras que en la cola solo puede haber no-terminales. Los índices a constantes se explican después. El número de símbolos en la cabeza (h) se decide en función de lo complejo que se sospeche sea el problema, mientras que el número de símbolos en la cola (t) se calcula con la ecuación 50:

$$t = h * (n-1) + 1 \qquad \text{Ec. 50}$$

Donde *n* es el número máximo de argumentos que tengan las funciones no-terminales.

Usando este tamaño de cola se garantiza que cualquier cromosoma pueda ser completado, independientemente de los no-terminales que tenga (lo cual es una ventaja respecto a las gramáticas evolutivas).

La forma de transformar un gen en una sentencia de un programa es a través de las llamadas expresiones-k. En ellas se rellena el árbol sintáctico con los símbolos del gen, de arriba a abajo y de izquierda a derecha. Por ejemplo, si los no-terminales son {sqr, log, +, -, *, /} y los terminales son las variables {a, b, c} entonces n=2, ya que el número de argumentos de entrada de la suma, resta, multiplicación y división es 2, mientras que la raíz cuadrada y el logaritmo solo necesitan 1.

Supongamos que decidimos h=4. Eso hace —aplicando la ecuación 50— que t=5. En la figura 164 podemos ver un ejemplo de un gen con cabeza (genes 0, 1, 2 y 3) y cola (genes 4, 5, 6, 7, y 8) y en la figura 165 su correspondiente árbol sintáctico rellenado, como dijimos, de arriba a abajo y de izquierda a derecha.

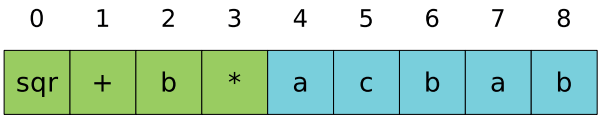

Figura 164. Gen con cabeza (en color verde) y cola (en color azul)

Es importante remarcar que:

- En la cola no hay funciones, es decir, solo hay terminales.
- No es obligatorio usar toda la cola. En este caso solo se emplea el primer símbolo.

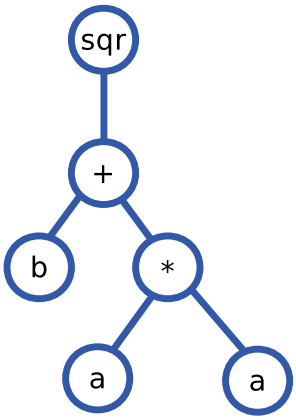

Figura 165. Árbol sintáctico

La sentencia correspondiente es *sqrt(b+a*a)*.

Aparte de la población de cromosomas, existe un vector lineal conteniendo solo constantes, que pueden ser útiles para la ejecución de programas. Este vector de constantes sufre de vez en cuando mutaciones, independientemente de la población general. En los genes existe un tercer dominio que hace referencia a esas constantes por medio de índices a ese vector. Y hay un nuevo símbolo terminal que es la "?" y que puede aparecer, como cualquier otro terminal, en los otros dos dominios (cabeza y cola). En la figura 166 (la zona de color magenta son los índices que se refieren al vector de constantes) podemos ver un ejemplo, y en la figura 167 su conversión a árbol sintáctico en dos pasos: primero se genera la expresión-k y luego se sustituyen los símbolos "?" por constantes: la primera "?" se cambia por el índice 7 que nos lleva a la constante -8 en el vector de constantes; y la segunda "?" se cambia por el índice 6 que nos lleva a la constante 4.4 en el vector de constantes. Si hubiera una tercera constante se cambiaría por el índice 2 que nos lleva a la constante 0.9.

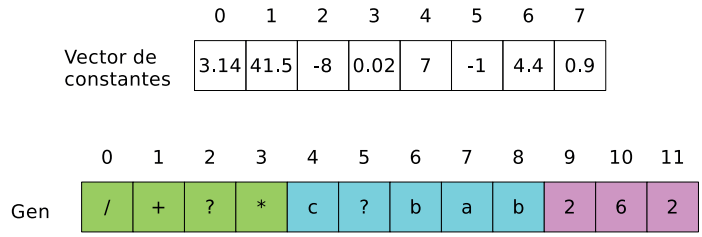

Figura 166. Vector de constantes, común a todos los genes de todos los cromosomas. Y un ejemplo de un gen con cabeza (color verde) cola (color azul) e índices a constantes (color magenta)

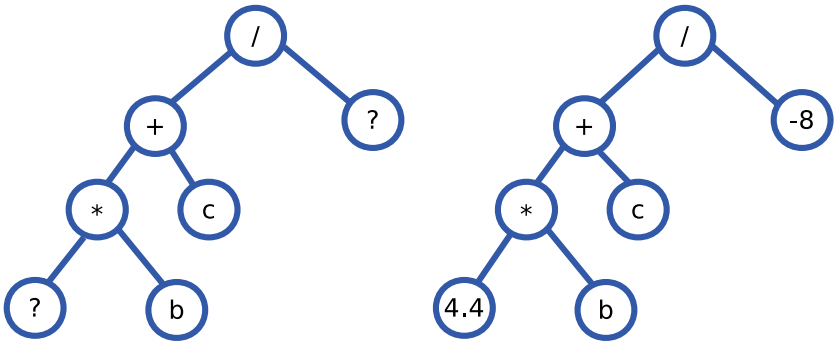

Figura 167. Conversión a árbol sintáctico en dos pasos

La sentencia que resulta es *(4.4*b+c)/(-8)*.

Esta es una muy buena idea, aunque la implementación es un poco deficiente porque no está claro cómo actúa la presión selectiva sobre estas constantes, ya que solo existe un vector de constantes y no hay una función de selección sobre ellas.

La forma del cromosoma en programación por expresión genética es un vector de genes, como se ve en la figura 168.

Figura 168. Cromosoma como vector de genes con cabeza, cola e índices a constantes

Otra novedad muy interesante es que uno o más de los últimos genes pueden incluir referencias a los genes anteriores dentro de sus terminales. Con ello se logra la construcción de programas complejos, y no simples sentencias (Figura 169).

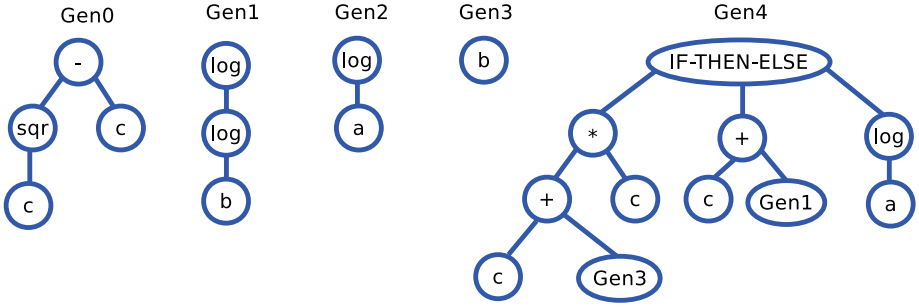

Figura 169. Ejemplo más complejo, donde el último gen puede llamar a otros genes, como si fueran funciones

El programa correspondiente es el de la figura 170.

```
if(c+(b)*c)
    c+(log(log(b)))
else
    log(a)
```

Figura 170. Programa

Coevolución

Cualquiera de los algoritmos evolutivos anteriores se puede usar en un escenario coevolutivo, que significa que el problema no es estático sino que también evoluciona.

En biología prácticamente todo es coevolutivo, porque cuando un ser vivo evoluciona lo hace adaptándose al comportamiento de sus predadores y de sus presas. Incluso en el caso de las plantas, donde su comida es inerte (luz, agua, aire, minerales del suelo), también coevolucionan conforme todos estos factores cambian (desaparecen minerales, cambia la composición del aire, aumenta la humedad, por mencionar algunos). Y dado que los seres vivos también producen cambios en la atmósfera, en la humedad de su entorno, en la disponibilidad de minerales y nutrientes, hay que considerar que todo el planeta Tierra es una red compleja de sistemas que se realimentan y coevolucionan unos con otros. Esto es lo que James Lovelock denominó la teoría de Gaia.

Esto se conoce también como las carreras de armamentos entre predador y presa (que veremos en el capítulo "Teoría de Juegos") donde, a lo largo de generaciones, el predador desarrolla nuevas estrategias para capturar a la presa, mientras la presa desarrolla nuevas estrategias para huir del depredador. También se le conoce como la carrera de la Reina Roja: en *Alicia a través del espejo*, Lewis Carroll nos relata el encuentro de Alicia con un personaje de ajedrez, la Reina Roja, que vive en un extraño país donde al caminar, cada vez que da un paso hacia adelante el suelo se mueve la misma distancia hacia atrás. La Reina Roja comenta que hay que correr mucho para mantenerse en el mismo lugar. La coevolución es igual.

La coevolución puede usarse de dos formas principales:

- En juegos, en sentido amplio, podemos tener un jugador evolutivo tratando de superar a otros jugadores con estrategias bien conocidas. Una vez que el jugador evolutivo ha logrado superarlos, el nuevo algoritmo resultante se añade a los jugadores con estrategias conocidas, y se rearranca el proceso de evolución. Como resultado se irán obteniendo estrategias de juego cada vez más sofisticadas. Esto es lo que emplea rutinariamente Google (por ejemplo, con Alpha Go Zero) y otras empresas para mejorar sus algoritmos de inteligencia artificial. Aquí hacemos coevolución de soluciones.

- En problemas muy complejos, se puede partir de una versión muy simplificada del problema y, conforme el algoritmo evolutivo la vaya solucionando, se le puede añadir más complejidad hasta convertirlo en la versión real y completa. Aquí hacemos coevolución de problemas y soluciones.

Algoritmo evolutivo general básico

Ya que existen tantas variantes de algoritmos evolutivos, se propone aquí una lo suficientemente general para servir en un amplio tipo de problemas. Como ejemplo de aplicación, se usa para resolver el problema de las N-Damas, especificándose primero los criterios de aceptación en Cucumber, para pasar luego al programa propiamente dicho en lenguaje *Ruby*[43].

```
# language: es
# encoding: utf-8
# Archivo: VerificarNDamasGA.feature
# Autor: Ángel García Baños
# Email: angarciaba@gmail.com
# Fecha creación: 2015-05-20
# Fecha última modificación: 2015-05-20
# Versión: 0.1
# Licencia: GPL

Característica: Verificar que funciona la evaluación de un Cromosoma en el algoritmo genético para la N-Damas. Nota: los tableros aquí indicados no deben tener conflictos en filas ni en columnas, pues eso no se verifica.
  Escenario: Ningún conflicto
    Cuando el tablero es
|   | X |   |   |
|   |   |   | X |
| X |   |   |   |
|   |   | X |   |
  Entonces al evaluarlo debe indicar 0 conflictos
```

43 Disponible en: https://goo.gl/EB4gTq

Escenario: Un conflicto en diagonal principal
    Cuando el tablero es
X			
	X		
Entonces al evaluarlo debe indicar 1 conflicto

Escenario: Un conflicto en diagonal principal
    Cuando el tablero es
	X		
		X	
Entonces al evaluarlo debe indicar 1 conflicto

Escenario: Un conflicto en diagonal secundaria
    Cuando el tablero es
		X	
	X		
Entonces al evaluarlo debe indicar 1 conflicto

Escenario: Un conflicto en diagonal secundaria
    Cuando el tablero es
	X		
X			
Entonces al evaluarlo debe indicar 1 conflicto

Escenario: Dos conflictos
    Cuando el tablero es
			X
	X		
		X	
Entonces al evaluarlo debe indicar 2 conflictos

```
Escenario: Dos conflictos
    Cuando el tablero es
| X |   |   |   |
|   | X |   |   |
|   |   | X |   |
|   |   |   |   |
Entonces al evaluarlo debe indicar 2 conflictos

Escenario: Tres conflictos
    Cuando el tablero es
| X |   |   |   |
|   | X |   |   |
|   |   | X |   |
|   |   |   | X |
Entonces al evaluarlo debe indicar 3 conflictos

Escenario: Tres conflictos
    Cuando el tablero es
|   |   |   | X |
| X |   |   |   |
|   | X |   |   |
|   |   | X |   |
Entonces al evaluarlo debe indicar 3 conflictos

Escenario: Cuatro conflictos
    Cuando el tablero es
|   |   | X |   |
|   |   |   | X |
| X |   |   |   |
|   | X |   |   |
Entonces al evaluarlo debe indicar 4 conflictos
# language: es
# encoding: utf-8
# Archivo: VerificarNDamasCromosoma.feature
# Autor: Ángel García Baños
# Email:  angarciaba@gmail.com
# Fecha creación: 2014-11-08
# Fecha última modificación: 2015-03-24
# Versión: 0.2
# Licencia: GPL
```

```
  Característica: Verificar el correcto funcionamiento de los Cromosomas.
  Antecedentes: Crear unos cuantos cromosomas para poder trabajar con
ellos
      Dado que los Cromosomas van a ser de 50 genes
      Y tengo un Cromosoma1 con todos los genes distintos
      Y copio el Cromosoma1 al Cromosoma2

  Escenario: El Cromosoma debe estar bien formado
  Entonces todos los genes del Cromosoma1 deben ser distintos

  Escenario: La mutación funciona
      Cuando muto el Cromosoma1 10 veces
      Entonces todos los genes del Cromosoma1 deben ser distintos
      Y el Cromosoma1 debe ser distinto al Cromosoma2

  Escenario: La mutación funciona
      Cuando muto el Cromosoma1 1 veces
      Entonces todos los genes del Cromosoma1 deben ser distintos
      Y la diferencia entre el Cromosoma1 y el Cromosoma2 deben ser 2
genes

  Escenario: El cruce funciona
      Cuando copio el Cromosoma1 al Cromosoma3
      Y muto el Cromosoma3 50 veces
      Y cruzo el Cromosoma1 con el Cromosoma3 dando como resultado
      el Cromosoma4
      Entonces el Cromosoma4 debe tener los genes del Cromosoma1 o el
      Cromosoma3
      Y el Cromosoma4 debe ser distinto al Cromosoma1
      Y el Cromosoma4 debe ser distinto al Cromosoma3

  # encoding: utf-8
  # Archivo: VerificarNDamas_steps.rb
  # Autor: Ángel García Baños
  # Email:  angarciaba@gmail.com
  # Fecha creación: 2015-05-20
  # Fecha última modificación: 2015-05-20
  # Versión: 0.1
  # Licencia: GPL
  Cuando(/^el tablero es$/) do |tabla|
```

```
    @cromosoma = Cromosoma.new(0)
    tablero.each_index do |fila|
      tablero[fila].each_index { |columna|  @cromosoma[columna] = fila if
tablero[fila][columna] and not tablero[fila][columna].empty? }
    end
  end

  Entonces(/^al evaluarlo debe indicar (\d+) conflictos?$/) do |numero-
Conflictos|
    algoritmoGeneticoTest = AlgoritmoGeneticoTest.new
    aptitud = algoritmoGeneticoTest.evaluar(@cromosoma)
    expect(-aptitud).to eq(numeroConflictos.to_i)
  end

  # encoding: utf-8
  # Archivo: VerificarNDamas_steps.rb
  # Autor: Ángel García Baños
  # Email:  angarciaba@gmail.com
  # Fecha creación: 2014-11-08
  # Fecha última modificación: 2015-05-19
  # Versión: 0.1
  # Licencia: GPL

  Dado /^que los Cromosomas van a ser de (\d+) genes$/  do |numeroDe-
Genes|
    @numeroDeGenes = numeroDeGenes.to_i
    @cromosomas = Hash.new(Cromosoma.new(@numeroDeGenes))
  end

  Y /^tengo un Cromosoma(.+?) con todos los genes distintos$/ do |nom-
breDeUnCromosoma|
     @cromosomas[nombreDeUnCromosoma] = Cromosoma.new(@nu-
meroDeGenes)
  end

  Y /^copio el Cromosoma(.+?) al Cromosoma(.+?)$/ do |nombreDeUn-
Cromosoma, nombreDelOtroCromosoma|
     @cromosomas[nombreDelOtroCromosoma] = @cromosomas[nom-
breDeUnCromosoma].clone
  end
```

```
Cuando /^muto el Cromosoma(.+?) (\d+) veces$/ do |nombreDeUnCromosoma, numeroDeVeces|
    numeroDeVeces.to_i.times { @cromosomas[nombreDeUnCromosoma].mutar! }
end

Entonces /^todos los genes del Cromosoma(.+?) deben ser distintos$/ do |nombreDeUnCromosoma|
  vecesQueEstaCadaGen = Hash.new(0)
  @cromosomas[nombreDeUnCromosoma].each { |gen| vecesQueEstaCadaGen[gen] += 1 }
  vecesQueEstaCadaGen.each_value { |veces| expect(veces).to eq(1) }
end

Y /^el Cromosoma(.+?) debe ser distinto al Cromosoma(.+?)$/ do |nombreDeUnCromosoma, nombreDelOtroCromosoma|
    iguales = @cromosomas[nombreDeUnCromosoma].zip(@cromosomas[nombreDelOtroCromosoma]).reduce(true) { |acumulado, genes| genes[0] == genes[1] ? acumulado : false }
  expect(iguales).not_to eq(true)
end

Entonces(/^la diferencia entre el Cromosoma(.+?) y el Cromosoma(.+?) deben ser (.+?) genes$/) do |nombreDeUnCromosoma, nombreDelOtroCromosoma, numeroDeGenesDistintos|
    diferencia = @cromosomas[nombreDeUnCromosoma].zip(@cromosomas[nombreDelOtroCromosoma]).reduce(0) { |acumulado, genes| genes[0] == genes[1] ? acumulado : acumulado+1 }
  expect(diferencia).to eq(numeroDeGenesDistintos.to_i)
end

Cuando(/^cruzo el Cromosoma(.+?) con el Cromosoma(.+?) dando como resultado el Cromosoma(\d+)$/) do |nombreDeUnCromosoma, nombreDelOtroCromosoma, nombreDelCromosomaHijo|
  @cromosomas[nombreDelCromosomaHijo] = @cromosomas[nombreDeUnCromosoma].cruzar(@cromosomas[nombreDelOtroCromosoma])
end
```

```
Entonces(/^el Cromosoma(.+?) debe tener los genes del Cromosoma(.+?) o el Cromosoma(.+?)$/) do |nombreDelCromosomaHijo, nombreDeUnCromosoma, nombreDelOtroCromosoma|
  correcto = @cromosomas[nombreDeUnCromosoma].zip(@cromosomas[nombreDelOtroCromosoma]).zip(@cromosomas[nombreDelCromosomaHijo]).reduce(true) { |acumulado, genes| genes[1] == genes[0][0] or genes[1] == genes[0][1] ? acumulado : false }
  expect(correcto).to eq(true)
end

#!/usr/bin/env ruby
# encoding: utf-8
# Programa: NDamas.rb
# Autor: Ángel García Baños
# Email: angarciaba@gmail.com
# Fecha creación: 2014-11-07
# Fecha última modificación: 2014-11-08
# Versión: 0.3

##########################################
# Utilidad: Enseñar a programar en Ruby. Hacer un algoritmo genético para resolver las N-Damas
##########################################
# VERSIONES
# 0.3 rdoc
# 0.2 Se simplificaron las clases a solo dos (Cromosoma y AlgoritmoGeneticoNDamas), para que fuera un ejemplo sencillo y didáctico.
# 0.1 La primera. Con clases Gen, Cromosoma, AlgoritmoGenetico y NDamas. Demasiado complicado.
########################################## Para ayudar a depurar:
def dd(expresion,entorno,mensaje="")
  p "#{expresion}=#{entorno.eval(expresion)}  #{mensaje}"
end
# Ejemplo de uso:
# a="Hola"
# dd("a",binding)
########################################## Para que funcione bundler (que traiga las gemas especificadas en Gemfile):
require 'rubygems'
```

```
require 'bundler/setup'
##########################################=begin
require 'ruby-prof'
RubyProf.start
$result = RubyProf.stop
printer = RubyProf::FlatPrinter.new($result)
printer.print(STDOUT)
=end
##################################################

# El Gen es un entero, por lo que no merece la pena hacer una clase para ello

# El Cromosoma es un Array de Genes (enteros)
class Cromosoma < Array
 # El Cromosoma tiene una aptitud que se puede leer y escribir
 attr_accessor :aptitud

 # Se define cuantosGenes va a tener el Cromosoma
  # Este constructor depende del problema a resolver, en este caso, las NDamas. Por ello, lo que hace es
 # construir un Cromosoma de genes enteros en un Array, cuya posición indica la columna y cuyos alelos
 # codifican el número de la fila donde se ubica cada reina
 def initialize(cuantosGenes=0)
  super
   cuantosGenes.times { |n| self[n] = n }  # Para asegurar que no haya alelos (valores de genes) repetidos
  sort_by! {rand}  # Se permutan los genes al azar
 end

 # La mutación se hace intercambiando dos genes cualesquiera
 def mutar!
  cualGen1, cualGen2 = rand(size), rand(size)
  self[cualGen1], self[cualGen2] = self[cualGen2], self[cualGen1]
  @aptitud = nil
  self
 end
```

```
    # El cruce se hace uniforme, eligiendo al azar un gen del padre o de la
madre, para cada posición
    def cruzar(otroCromosoma)
      hijo = Cromosoma.new
      self.zip(otroCromosoma).each { |genPadre, genMadre| hijo << (rand <
0.5 ? genPadre : genMadre) }
      hijo
    end

    # No se puede usar el operador = porque hace una copia superficial (por
referencia), y lo que queremos
    # es una copia profunda (por valor).
    # Ojo: El operador = no se puede redefinir porque es global (se puede
usar con cualquier tipo de dato), por
    # lo que no pertenece a ninguna clase específica.
    def clone
      otroCromosoma = Cromosoma.new
      otroCromosoma.replace(self)
      otroCromosoma.aptitud = self.aptitud
      otroCromosoma
    end
  end

  # Esta clase implementa un algoritmo genético para resolver el problema
de las NDamas. Es un Array de Cromosomas.
  class AlgoritmoGeneticoNDamas < Array
    attr_reader :mejorCromosoma, :numeroDeEvaluaciones

    # En el constructor hay que especificar cuantas damas tiene el problema
y cuantos Cromosomas se desea tener
    # en la población
    def initialize(cuantasDamas, cuantosCromosomas=100)
      cuantosCromosomas.times { self << Cromosoma.new(cuantasDamas) }
      @mejorCromosoma = Cromosoma.new
      @numeroDeEvaluaciones = 0
    end

    # Se ejecuta el número de generaciones que se especifique aquí
    def ejecutar(cuantasGeneraciones=1000)
      cuantasGeneraciones.times do
```

```
        cromosoma1 = seleccionarPorTorneo
        cromosoma2 = seleccionarPorTorneo
    #     cromosomaHijo = cromosoma1.cruzar(cromosoma2)  # El cruce no
se debe usar, porque genera cromosomas
    # inválidos
        cromosomaHijo = cromosoma1
        cromosoma1.mutar!.mutar!
        cromosoma2.mutar!
        reemplazar(cromosoma1)
        reemplazar(cromosoma2)
        reemplazar(cromosomaHijo)
       end
       self
      end

      private

      # Evalúa un Cromosoma, retornando su aptitud.
      # La función de evaluación depende del problema concreto a resolver.
En este caso, las NDamas, se
      # puntúa negativamente cada colisión entre damas, en los dos tipos de
diagonales
      # En las filas y en las columnas es imposible que haya colisiones debido
a la forma de codificar el Cromosoma
      # Dos damas A y B están en la misma diagonal si la recta que definen
tiene pendiente +1 o -1, o sea:
      # (yA-yB)/(xA-xB) = +-1
      #   ==>  (yA-yB)=+-(xA-xB)  ==>   (yA+-xA) = (yB+-xB)
      def evaluar(cromosoma)
       return cromosoma.aptitud if cromosoma.aptitud
       @numeroDeEvaluaciones += 1

       coordenadas = []
       cromosoma.each_index do |fila|
        coordenadas[fila] = []
        coordenadas[fila][0] = fila+cromosoma[fila] if cromosoma[fila]
        coordenadas[fila][1] = fila-cromosoma[fila] if cromosoma[fila]
       end
        repetidos = coordenadas.inject([Hash.new(0),Hash.new(0)]) { |hash,
item| hash[0][item[0]] += 1; hash[1][item[1]] += 1; hash}
```

```
        conflictos = 0
        repetidos.each do |r|
         conflictos += r.inject(0) { |acumulador, item| acumulador+item[1]-1 }
        end

        cromosoma.aptitud = -conflictos
         @mejorCromosoma = cromosoma.clone if not @mejorCromosoma.
aptitud or @mejorCromosoma.aptitud < cromosoma.aptitud # Copia pro-
funda
        cromosoma.aptitud
      end

      # La selección por torneo elige dos Cromosomas al azar y retorna el que
tenga mejor aptitud
      def seleccionarPorTorneo
        cromosoma1 = sample
        cromosoma2 = sample
        if evaluar(cromosoma1) > evaluar(cromosoma2) then
          cromosoma1
        else
          cromosoma2
        end
      end

      # El reemplazo recibe un nuevo Cromosoma y lo inserta en un lugar al
azar en la población, eliminando al
      # cromosoma que estuviera allí
      def reemplazar(cromosoma)  # Eliminando otro al azar
        self[rand(size)] = cromosoma.clone  # Copia profunda
      end
    end

    if $0 == __FILE__
      ######################################
      # INPUTS
      numeroCromosomas=200
      numeroGeneraciones=100000
      numeroDamas=12
      ######################################
```

```
    espacioBusqueda=1
    1.upto(numeroDamas) { |i| espacioBusqueda *= i }
     nDamas = AlgoritmoGeneticoNDamas.new(numeroDamas,numero-
Cromosomas)
    nDamas.ejecutar(numeroGeneraciones)
     p "====== SOLUCION PERFECTA ======" if nDamas.mejorCro-
mosoma.aptitud == 0
      p "El mejor cromosoma es #{nDamas.mejorCromosoma} que tiene
#{-nDamas.mejorCromosoma.aptitud} conflictos."
    p "El espacio de búsqueda es #{espacioBusqueda} y se hicieron #{nDa-
mas.numeroDeEvaluaciones} evaluaciones (#{(100.0*nDamas.numero-
DeEvaluaciones)/espacioBusqueda} %)"
  end
```

Resumen

La evolución fue descubierta por Darwin en biología, pero puede ocurrir en cualquier otro ambiente, incluyendo el computacional, siempre y cuando se den cuatro condiciones: que haya una población de entes, que saquen copias de sí mismos (reproducción), pero que las copias no sean exactas (variabilidad) y que estén sometidos a una presión selectiva.

Los algoritmos evolutivos sirven para hacer búsqueda, optimización y diseños creativos. Son algoritmos muy simples y generales, y de allí su potencia. A cambio, son lentos. Los usaremos como última opción, es decir, cuando no se conozca ningún otro algoritmo razonable, cuando las entradas estén mal especificadas, cuando los requerimientos cambien con el tiempo o cuando el objetivo a lograr no esté muy claro.

A lo largo de los años se han diseñado varios tipos de algoritmos evolutivos, dependiendo de cuál sea el objeto que se quiera hacer evolucionar. El más sencillo es el algoritmo genético, que hace evolucionar un conjunto de datos simples como enteros, bits o *strings*. Las estrategias evolutivas hacen evolucionar un conjunto de parámetros flotantes, y el enfriamiento simulado un conjunto de estructuras de datos cualesquiera. Por otro lado, también se pueden hacer evolucionar programas de computador, y eso es lo que hace la programación genética, las gramáticas evolutivas y la programación por expresión genética. La programación evolutiva hace evolucionar máquinas de estado, que se pueden considerar un tipo limitado de programas. Y los sistemas clasificadores hacen evolucionar un sistema de reglas que se pueden ver también como datos o como programas.

Los algoritmos evolutivos se han estudiado en mayor detalle, porque muchas de sus partes se aplican a los demás algoritmos. En particular hemos visto la selección, la reproducción y el reemplazo, así como los principales problemas que suelen aparecer. Recordemos que lo importante de la selección es que sea probabilista, dando mayor prioridad a los mejores pero sin descuidar a los peores. Y que el operador de reproducción más importante es la mutación, que debe producir cambios pequeños y que, si se aplica infinitas veces, debe poder generar todos los cromosomas posibles.

También hemos visto la búsqueda multiobjetivo, usando varios criterios de dominancia de Pareto.

Hay otras variantes de los algoritmos anteriores que se muestran porque son populares: la evolución diferencial y los algoritmos híbridos de Taguchi.

Además, se explica lo que es el concepto de la coevolución, cuando evolucionan simultáneamente tanto el problema como el algoritmo que lo soluciona.

Al final se incluye el código en *Ruby* de un algoritmo genético optimizado, junto con su especificación de pruebas en *Cucumber*, aplicado a resolver el problema de las N-Damas.

Para saber más

- Anil Menon et ál. (2004). *Frontiers of evolutionary computation*. New York: Kluwer Academic Publishers.

Un resumen de las técnicas evolutivas, donde se hace énfasis en los aspectos a mejorar y en los que desconocemos por completo.

- David E. Goldberg (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Massachusetts: Addison-Wesley Publishing Company Inc.

Escrito por un alumno de Holland, es el primer libro que habla de algoritmos evolutivos y sus numerosas aplicaciones. En particular muestra su uso para optimizar las tuberías de distribución de gas.

- Melanie Mitchel (1999). *An introduction to genetic algorithms*. Cambridge: The MIT Press.

Cuenta lo básico de los algoritmos genéticos, donde se incluyen distintas variantes y ejemplos. Es de particular interés un tipo de algoritmo coevolutivo, diseñado por Hillis.

- Richard J. Bauer Jr. (1994). *Genetic Algorithms and Investment Strategies*. New York: John Willey & Sons Inc.

Habla de algoritmos genéticos, caos, redes neuronales, y cómo analizar series de datos financieras en busca de patrones fractales. Explica muchos detalles sobre las estrategias de inversión. Independientemente del título, el libro no devela el secreto de cómo hacerse rico, pero sí propone ideas para investigar y está repleto de citas bibliográficas para consultar estudios más detallados.

- Una-May O'Reilly, Tina Yu, Rick Riolo y Bill Worzel (2005). *Genetic Programming: Theory and practice II*. Boston: Springer.

Se centra en la programación genética, es decir, generar programas de forma evolutiva. Contiene varios artículos individuales y muchos ejemplos.

Referencias

Libros, artículos y enlaces web

Arcuri, A. y Yao, X. (2014). Co-evolutionary automatic programming for software development. *Information Sciences, 259*(20), pp. 412-432. Elsevier. DOI: https://doi.org/10.1016/j.ins.2009.12.019

Bentley, P. J. (1999). *Evolutionary Design by Computers*. San Francisco: Morgan Kaufmann.

Bongard, J. y Lipson, H. (2007). Automated reverse engineering of nonlinear dynamical systems. *PNAS, 104*(24), pp. 9943–9948. DOI: https://doi.org/10.1073/pnas.0609476104

Carroll, L. (1992). *Alicia en el país de las maravillas: A través del espejo.* Traducción de Ramón Buckley. Madrid: Ediciones Cátedra.

Delgado, C. A., García, Á. y Bucheli, V. A. (2017). *Perception of rankings: risks and opportunities*. Enviado a publicación.

Draves, S. (2017). *Scott Draves – Software Artist*. Recuperado el 17 de agosto de 2017. Disponible en: https://goo.gl/W7iDYu

Ferreira, C. (2002). Combinatorial Optimization by Gene Expression Programming: Inversion Revisited. *Proceedings of the Argentine Symposium on Artificial Intelligence,* pp. 160-174. Argentina.

Fogel, D. B. (1995). *Evolutionary Computation. Toward a New Philosophy of Machine Intelligence*. Piscataway, New Jersey: IEEE Press.

Fogel, L. J., Owens, A. J. y Walsh, M. J. (1967). *Artificial Intelligence Through Simulated Evolution*. New York: John Willey & Sons.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Boston: Addison-Wesley.

Jacob, F. (1977). Evolution and Tinkering. *Science, 196*(4295), pp. 1161–1166.

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge, Massachusetts: MIT Press.

Koza, J. R., Bennett, F. H., Andre, D. y Keane, M. A. (1999). *Genetic Programming III: Darwinian Invention and Problem Solving*. San Francisco: Morgan Kaufmann.

Latham, W. (2017). *William Latham – Software*. Recuperado el 17 de agosto de 2017. Disponible en: https://goo.gl/YVvbtM

Lovelock, J. (2000). *Las edades de Gaia*. Barcelona: Tusquets Editores.

Maes, P. (1996). *From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior*. Cambridge, Massachusetts: MIT Press.

Otten, R. H. J. M. y Van Ginneken, L. P. P. P. (1989). *The Annealing Algorithm*. Boston: Kluwer Academic Publishers.

Poorjandaghi, S. S. y Afshar, A. (2014). A Robust Evolutionary Algorithm for Large Scale Optimization. Proceedings from the *19th IFAC World Congress*, pp. 7037-7042. Cape Town: Elsevier Ltd.

Smith, S. F. (1980). *A Learning System Based on Genetic Adaptive Algorithms*. [PhD Thesis]. Department of Computer Science, University of Pittsburgh.

Sorkin, G. B. (1990). Simulated Annealing on Fractals: Theoretical Analysis and Relevance for Combinatorial Optimization. In *Dally J.W., 6th MIT Conference on Advanced Research in VLSI,* pp. 331-351. Cambridge, Massachusetts: The MIT Press.

Spector, L. (ed.) (2001). Proceedings from the 10th Genetic and Evolutionary Computation Conference - GECCO. San Francisco, California: Morgan Kaufmann.

Torres, A. R. y García, A. (2013). *Optimización de puntajes de admisión a una carrera universitaria, el caso de Ingeniería de Sistemas*. CLEI-2013, Venezuela: IEEE.

Tsai, J., Liu, T. y Chou, J. (2004). Hybrid Taguchi-genetic algorithm for global numerical optimization. *IEEE Transactions on Evolutionary Computation*, *8*(4), pp. 365-377.

Whitley, L. D., Goldberg, D. E., Cantú-Paz, E. *et ál.* (ed.) (2000). Proceedings from the *9th Genetic and Evolutionary Computation Conference - GECCO*. Las Vegas, Nevada: Morgan Kaufmann.

Yang, Ch. I., Jyh-Horng Ch. y Ching-Kao Ch. (2013). Hybrid Taguchi-based genetic algorithm for flowshop scheduling problem. *International Journal of Innovative Computing, Information and Control*, *9*(3), pp. 1045-1063.

Películas y videos

ABCUPM (2013). *Araña robótica con aprendizaje de la marcha mediante Algoritmo Genético*. Recuperado el 2 de septiembre de 2017. Disponible en: https://goo.gl/SgSq63

ArmoredSandwich (2010). *Genetic Algorithm and Robocode*. Recuperado el 2 de septiembre de 2017. Disponible en: https://goo.gl/Q2F2FR

Bacalov, D. (2010). *Demo de Algoritmos Genéticos*. Recuperado el 2 de septiembre de 2017. Disponible en: https://goo.gl/ZgZ2uQ

CornellCCSL (2014). *Evolved Electrophysiological Soft Robots*. Recuperado el 2 de septiembre de 2017. Disponible en: https://goo.gl/3fZLYi

Mann, C. (2010). *Pirouette - Evolved Virtual Creature*. Recuperado el 2 de septiembre de 2017. Disponible en: https://goo.gl/RyKMdD

OptimalEnergetics (2011). *Evolutionary Algorithms and Building Design*. Recuperado el 2 de septiembre de 2017. Disponible en: https://goo.gl/giKapZ

pɹɐddəɥS (2013). *Genetic evolution of a wheeled vehicle with Box2d*. Recuperado el 2 de septiembre de 2017. Disponible en: https://goo.gl/qnjHYS

Seč, V. (2012). *Karl Sims - Evolving Virtual Creatures With Genetic Algorithms*. Recuperado el 2 de septiembre de 2017. Disponible en: https://goo.gl/D9xpXb

Virtualspecies (2012). *Genetic Programming - "Santa Fe Trail" problem*. Recuperado el 2 de septiembre de 2017. Disponible en: https://goo.gl/7nvUE9

Tesis y trabajos de grado en EVALAB

Arias, C. G. (2010). *Descubrimiento automático de equivalencias entre esquemas de bases de datos relacionales*. [Tesis Maestría]. Cali: Universidad del Valle.

Barón, R. (1999). *Planificador de horarios de clases automático*. [Tesis Maestría]. Cali: Universidad del Valle.

Barona, M. A. (2013). *Música evolutiva*. Cali: Universidad del Valle.

Cabezas, I. M. (2004). *Algoritmos genéticos híbridos distribuidos*. Cali: Universidad del Valle.

Camayo, J. M. (2009). *Software para diseño evolutivo usando Ruby orientado a web 2.0*. Cali: Universidad del Valle.

Castrillón, J. (2015). *Módulo de asignación de agendas basado en algoritmos genéticos*. Cali: Universidad del Valle.

Chaparro, J. C. (2009). *Objetos virtuales para la enseñanza de la computación evolutiva -OVACE-*. Cali: Universidad del Valle.

Cossio, O. (2011). *Objeto virtual de aprendizaje para programación por expresión genética*. Cali: Universidad del Valle.

Cruz, M. A. (2014). *Búsqueda de solapamiento en clusters, usando técnicas de computación evolutiva*. Cali: Universidad del Valle.

Fuertes, C. E. y Tigreros, Ó. I. (2017). *Análisis, Implementación y Comparativas entre un Algoritmo Genético Tradicional, el Algoritmo Híbrido Genético-Taguchi y el Algoritmo Híbrido Genético-Taguchi Cooperativo-Coevolutivo para Problemas de Optimización Numérica Global*. Cali: Universidad del Valle.

Gómez, J. A. (2001). Desarrollo de una Máquina de Estados Finitos (FSM) evolutiva mediante algoritmos genéticos. Cali: Universidad del Valle.

Guzmán, J. (2008). *Desarrollo de una aplicación para la optimización del aprovechamiento de telas en la actividad de trazado en la industria de la confección mediante algoritmos genéticos*. [Tesis Meritoria]. Cali: Universidad del Valle.

Lourido, A. K. y Solano, C. (2004). *Desarrollo de una aplicación que colabora en la búsqueda de imágenes usando un algoritmo genético*. Cali: Universidad del Valle.

Pineda, D. L. y Estacio, I. (2003). *Desarrollo de una herramienta para la implementación de algoritmos evolutivos paralelos utilizando modelo de islas o celular*. Cali: Universidad del Valle.

Posada, J. M. (2014). *Aplicación de algoritmos evolutivos en un videojuego competitivo usando un lenguaje de muy alto nivel*. Cali: Universidad del Valle.

Posada, L. E. y Benítez, A. (2006). *Diseño e implementación de la interfaz de usuario para la interacción web con la biblioteca evolutiva*. Cali: Universidad del Valle.

Triana, J. (2013). *Desarrollo de un agente para el videojuego Tetris basado en técnicas de inteligencia artificial*. Cali: Universidad del Valle.

Villate, D. F. (2012). *Comparación de técnicas de inteligencia artificial aplicadas al juego Otelo*. Cali: Universidad del Valle.

Villegas, J. A. (2001). *Asignación de aulas empleando algoritmos genéticos paralelos*. Cali: Universidad del Valle.