Informática evolutiva

J. J. Merelo Guervós
jmerelo@kal-el.ugr.es
http://kal-el.ugr.es/~jmerelo

1 Búsqueda, optimización y aprendizaje

Esta terna define los problemas a los que se pueden aplicar los algoritmos evolutivos, en cualquiera de sus formas. Por eso parece conveniente, antes que nada, describir el tipo de problemas a los que nos podremos enfrentar, mejor que presentarlos como la panacea que es capaz de encontrar la respuesta a la Pregunta Última sobre la Vida, el Universo y todo lo demás (además, todo el mundo sabe que es 46).

En realidad, los algoritmos de búsqueda abarcan prácticamente todo. Habitualmente, en Informática se habla de búsqueda cuando hay que hallar información dentro de un conjunto de datos almacenados siguiendo un determinado criterio; sin embargo, aquí nos referiremos a otro tipo de algoritmos de búsqueda, a saber, aquellos que, dado el espacio de todas las posibles soluciones a un problema, y partiendo de una solución inicial, son capaces de encontrar la solución mejor o la única. El ejemplo clásico de este tipo de problemas se encuentra en los rompecabezas y juegos que se suelen abordar en **inteligencia artificial**, como el *problema de las 8 reinas*, en el cual se deben de colocar 8 reinas en un tablero de ajedrez de forma que ninguna amenace a otra, o las *torres de Hanoi,* un problema en el que, dada una serie de discos de radio decreciente apilados, hay que apilarlos en otro sitio teniendo en cuenta que no se puede colocar ningún disco encima de otro de radio inferior.

Este tipo de problemas es fácil de abordar usando algoritmos clásicos,

algoritmos recursivos o de tipo **voraz (*greedy*)**, sin embargo, hay otro tipo de problemas mucho más complicados, sobre todo los NP-completos (aquellos cuya complejidad crece con el tamaño del problema de forma exponencial) que no pueden ser abordados de esta forma. Algunos ejemplos de estos problemas serían los siguientes:

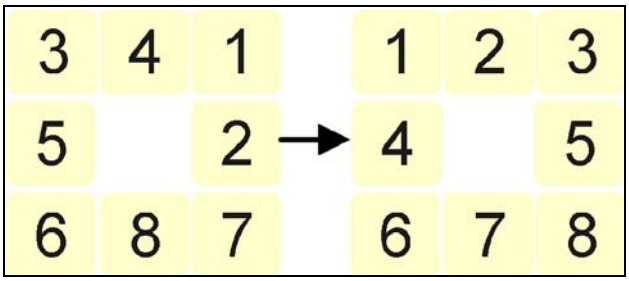


Figura 1: Problema del 8 puzzle.

- **8-puzzle**, en el cual, como se muestra en la ilustración 1, 4, a partir de una configuración inicial donde hay 8 cuadros desordenados, hay que llegar a otra configuración donde están ordenados, usando para el intercambio cualquier posición que se halle vacía. El problema de búsqueda en este caso consiste en encontrar un camino que vaya desde la configuración inicial hasta la final.
- **Problema del viajante**, en el cual, dadas una serie de ciudades separadas por diferentes distancias, hay que calcular un camino tal que la distancia total recorrida sea mínima, y no se tenga que pasar por una ciudad más de una vez. Este problema es NP-completo, y es paradigmático de este tipo de problemas.
- **Mastermind:** En este juego, que se muestra en la figura 2, un jugador debe de averiguar una combinación de chinchetas de colores oculta por el otro jugador, y lo hace haciendo suposiciones sobre la combinación, y siendo contestado con una chincheta pequeña y blanca por cada acierto de color, y una negra por cada acierto de color o posición. Solo una solución es la correcta, pero el jugador con la combinación oculta va orientando la búsqueda mediante las chinchetas blancas o negras. El problema se hace exponencialmente más difícil cuando se aumenta el número de colores, y la longitud de la combinación.

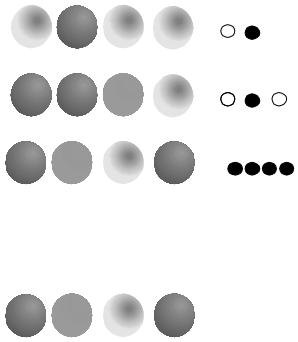


Figura 2 Juego del MasterMind. La combinación inferior se supone que está oculta.

En muchos casos, la búsqueda está guiada por una función que indica lo buena que es esa solución, o el coste de la misma, o lo cerca que se está de la solución final, si es que se conoce; el problema se convierte entonces en un problema de optimización, es decir, encontrar la solución que maximiza la función *objetivo*, *de evaluación* o *fitness* o minimiza el coste. En términos formales, dada una función *F* de *n* variables x_1, x_2, ..., x_n, optimizar la función consiste en encontrar la combinación de valores de x_i tales que $F(x_1, x_2, ..., x_n)$ = Máximo. Se puede hablar de maximizar en vez de minimizar sin perder generalidad, ya que maximizar *F* equivale a minimizar -*F.*

Generalmente, el problema de optimización es tratado por la rama de las matemáticas denominada **Investigación Operacional**, aunque prácticamente todas las ramas de la ciencia y la ingeniería necesitan tratar con problemas de optimización en algún momento. Por ejemplo, en *teoría de juegos* se trata de maximizar la probabilidad de ganar, y en *reconocimiento de patrones* de minimizar el error de clasificación de un patrón desconocido (como una imagen digitalizada).

El problema de optimización no siempre está formulado de una forma tan clara. En muchos casos, la forma de la función *F* no se conoce, y debe de aproximarse mediante polinomios o combinaciones de funciones conocidas; en este caso se trata de hallar los coeficientes de los polinomios o funciones que hacen que la función calculada se acerque lo más posible a la función objetivo. En este caso, en el que el problema se reduce a calcular una serie de coeficientes, se habla de *optimización paramétrica*.

En control industrial se plantean también problemas de optimización: como mantener el funcionamiento de una máquina dentro de su régimen óptimo, por ejemplo. Cada máquina suele tener una serie de parámetros variables, y su salida es habitualmente la calidad del producto final o la rapidez a la hora de producirlo.

En algunos casos, la función de evaluación ni siquiera existe, o no es estática, sino que viene dada por el entorno de la solución. Por ejemplo, en un programa para jugar al **Othello** o **reversi**, el *fitness* vendrá dada por su puntuación a la hora de jugar con los demás jugadores. En este caso se suele enfrentar unos jugadores con otros, de forma que según van evolucionando, el fitness va variando. Este tipo de optimización se suele encontrar en problemas de vida artificial y el *Dilema del Prisionero*, usado para modelizar interacciones sociales.

El *dilema del prisionero iterado* es un juego en el cual se enfrentan dos jugadores, que representan dos presos en una cárcel, que se han puesto de acuerdo para fugarse; en cada iteración, cada jugador decide si se chiva al director de la cárcel, o coopera y se escapa. Si los dos cooperan, reciben una recompensa de 3; si uno de los dos se chiva, recibe todos los privilegios de un preso bueno en la forma de una recompensa de 5, y si los dos se chivan, reciben una recompensa, pero mucho menor. Si este juego se repite por un número finito y conocido de iteraciones, la estrategia óptima es siempre chivarse, porque consigues una recompensa asegurada de *1*número de jugadas*, sin embargo, a largo plazo la estrategia óptima es cooperar, porque la recompensa es de 3. Con este juego se han hecho muchas variantes; un algoritmo debería de crear una estrategia de forma que maximice la recompensa de un jugador.

Respuesta jugador 1 ☞ Respuesta jugador 2 ↓	Cooperar	Chivarse
Cooperar	3,3	0,5
Chivarse	5,0	1,1

1 Tabla de recompensas en el *dilema del prisionero*

En algunos se trata de optimizar F(C), donde C es una combinación de diferentes elementos que pueden tomar un número finito de valores; pueden ser combinaciones con o sin repetición, o incluso permutaciones, como en el caso del problema del viajante; en este caso se denominan problemas de optimización

combinatoria.

No siempre, el espacio de búsqueda completo contiene soluciones válidas; en algunos casos, los valores de las variables se sitúan dentro de un rango, más allá del cual la solución es inválida. Se trata entonces de un problema de optimización con restricciones. En este caso, el problema consiste en maximizar $F(x_i)$ dentro del subespacio $R=\{\{x_1,...x_n\} / g_i(x_1,...x_n) \geq 0 \ \forall i =1,...,N_r \}$.

Hay muchas formar de abordar problemas de optimización. Algunas de ellas se verán en las siguientes secciones

1.1Método analítico

Si existe la función *F*, es de una sola variable, y tiene dos derivadas en todo su rango, se pueden hallar todos sus máximos, sean locales o globales. Sin embargo, la mayoría de las veces no se conoce la forma de la función *F*, y si se conoce, no tiene porqué ser diferenciable. Incluso el tratamiento analítico para funciones de más de 1 variable es complicado.

1.2 Métodos exhaustivos, aleatorios y heurísticos

Los métodos exhaustivos recorren todo el espacio de búsqueda, quedándose con la mejor solución, y los heurísticos utilizan reglas para eliminar zonas del espacio de búsqueda consideradas “poco interesantes”. Algunos algoritmos de búsqueda, como el MiniMax, son de este tipo; se suelen utilizar en juegos para examinar y podar el árbol de posibilidades a partir de la jugada actual; *Deep Blue*, por ejemplo, juega de esta forma.

En los métodos aleatorios, se va muestreando el espacio de búsqueda acotando las zonas que no han sido exploradas; se escoge la mejor solución, y además se da el intervalo de confianza de la solución encontrada.

1.3 Subiendo al cerro,

En estos métodos, también denominados de *hillclimbing*, se va evaluando la función en uno o varios puntos, pasando de un punto a otro en el cual el valor de la evaluación es superior. La búsqueda termina cuando se ha encontrado el punto con un valor máximo. En general, un algoritmo escalador funciona de la forma siguiente

Algoritmo escalador

1. Escoger una solución inicial $(x_i,...,x_n)$
2. Mientras que siga subiendo el valor de F, hacer
    3. Alterar la solución $(x'_i,...,x'_n) = (x_i,...,x_n) + (y_i,...,y_n)$, y evaluar F.
    4. Si $F(x'_i,...,x'_n) > F(x_i,...,x_n)$, hacer $(x_i,...,x_n) = (x'_i,...,x'_n)$.
    5. Volver a 2.

Estos algoritmos toman muchas formas diferentes, según el número de dimensiones del problema solución, el valor del incremento y en la dirección que se tiene que dar. En algunos casos se utiliza el llamado **Método Montecarlo** (por el casino), en el cual se escoge la nueva solución de forma aleatoria.

El principal problema de este tipo de algoritmos es que se quedan en el pico más cercano a la solución inicial; además, no son válidos para problemas *multimodales*, en los cuales la función de coste tiene varios óptimos posibles.

1.4 Recocimiento simulado

Conocido como *Simulated Annealing*, en inglés, el nombre viene de la forma como se consiguen ciertas aleaciones en forja; una vez fundido el metal, se va enfriando poco a poco, para conseguir finalmente la estructura cristalina correcta.

Este algoritmo se podría calificar como escalador estocástico, y su principal objetivo es evitar los mínimos locales en los que suelen caer los escaladores, para

ello, no siempre acepta la solución más óptima, sino que a veces puede escoger una solución menos óptima, siempre que la diferencia entre ambos tenga un nivel determinado, que depende de un parámetro denominado *temperatura* (seguimos con la metáfora). El algoritmo de recocimiento simulado es el siguiente:

Algoritmo de recocimiento simulado

1. Inicializar la temperatura T, y la solución inicial $(x_i,...,x_n)$ y evaluar $F(x_i,...,x_n)$.
2. Repetir los pasos siguientes, hasta que la temperatura sea nula o el valor de F converja:
   3. Disminuir la temperatura.
   4. Seleccionar una nueva solución $(x'_i,...,x'_n)$ en la vecindad de la anterior (*mutar la solución*), y evaluarla.
   5. Si $F(x'_i,...,x'_n) > F(x_i,...,x_n)$, hacer $(x_i,...,x_n) = (x'_i,...,x'_n)$, si no, generar un número aleatorio R entre 0 y 1. Si $\exp(\frac{F(x'_1,...,x'_n)-F(x_1,...,x_n)}{T})>R$, entonces $(x_i,...,x_n) = (x'_i,...,x'_n)$.

1.5 Técnicas basadas en poblacion

Este tipo de técnicas pueden ser versiones de cualquiera de las anteriores, pero en vez de tener una sola solución, que se va alterando hasta obtener el óptimo, se persigue el óptimo cambiando varias soluciones; de esta forma es más fácil escapar de los mínimos locales tan temidos. Entre estas técnicas se hallan la mayoría de los algoritmos de informática evolutiva.

2 La evolución

Cambiemos totalmente de tercio, y después de ver tipo de problemas a los que se pueden aplicar los algoritmos evolutivos, vamos a estudiar qué es lo que inspira dichos algoritmos, la Naturaleza y ese fenómeno natural denominado evolución.

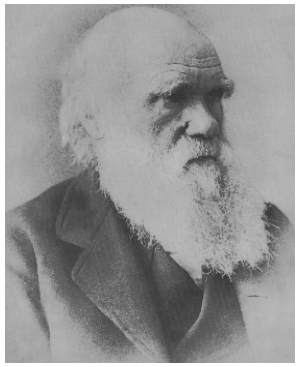


Figura 3 Charles Darwin.

La teoría de la evolución (que no es tal teoría, sino una serie de hechos probados), fue descrita por Charles Darwin (ilustración ?), después de su viaje por las islas Galápagos en el *Beagle*, en el libro *Sobre el Origen de las Especies por medio de la Selección Natural*. Este libro fue bastante polémico en su tiempo, y en cualquier caso es una descripción incompleta de la evolución. La hipótesis de Darwin, presentada conjuntamente con Wallace, que llegó a las mismas conclusiones independientemente, es que pequeños cambios heredables en los seres vivos y la selección son los dos hechos que provocan el cambio en la Naturaleza y la generación de nuevas especies. Sin embargo, Darwin pensaba que los rasgos de un ser vivo eran como un fluido, y que los "fluidos" de los dos padres se mezclaban en la descendencia; esto provocaba el problema de que al cabo de cierto tiempo, un población tendría los mismos rasgos intermedios.

Figura 4 Gregor Mendel, de abad.

Fue **Mendel** (ilustración ?) quien descubrió que los caracteres se heredaban de forma discreta, y que se tomaban del padre o de la madre, dependiendo de su carácter dominante o recesivo. A estos caracteres que podían tomar diferentes valores se les llamó *genes*, y a los valores que podían tomar, *alelos*. En realidad, las teorías de Mendel, que trabajó en total aislamiento, se olvidaron y no se volvieron a redescubrir hasta principios del siglo XX. Además, hasta 1930 el geneticista inglés **Robert Aylmer** no relacionó ambas teorías, demostrando que los genes mendelianos eran los que proporcionaban el mecanismo necesario para la evolución.

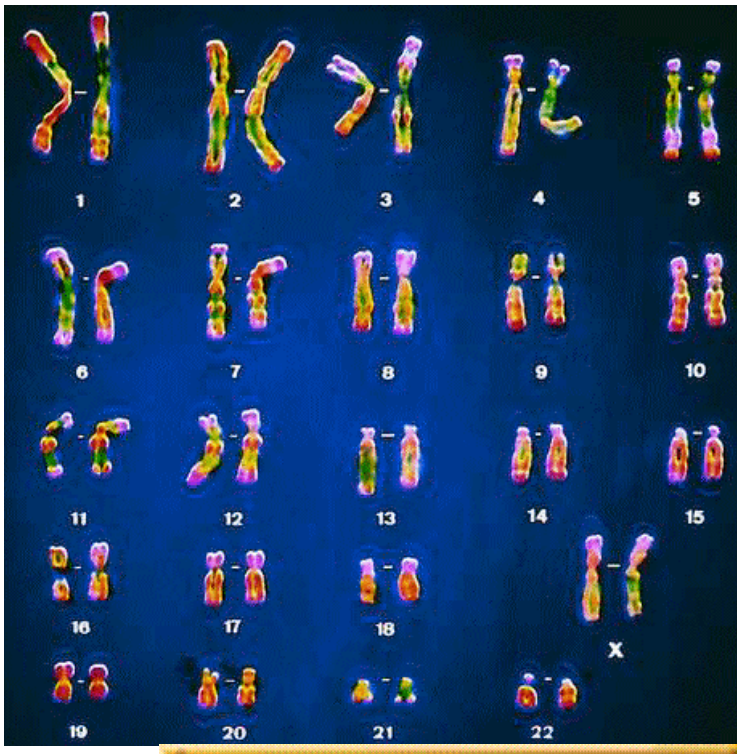


Figura 5: C
(obsérvese l
Enciclopedia

Más o menos por la misma época, el biólogo alemán **Walther Flemming** describió los cromosomas, como ciertos filamentos en los que se agregaba la cromatina del núcleo celular durante la división; poco más adelante se descubrió que las células de cada especie viviente tenía un número fíjo y característico de cromosomas.

Y no fue hasta los años 50, cuando Watson y Crick descubrieron que la base molecular de los genes está en el ADN, ácido desoxirribonucleico, Los cromosomas están compuestos de ADN, y por tanto los genes están en los cromosomas.

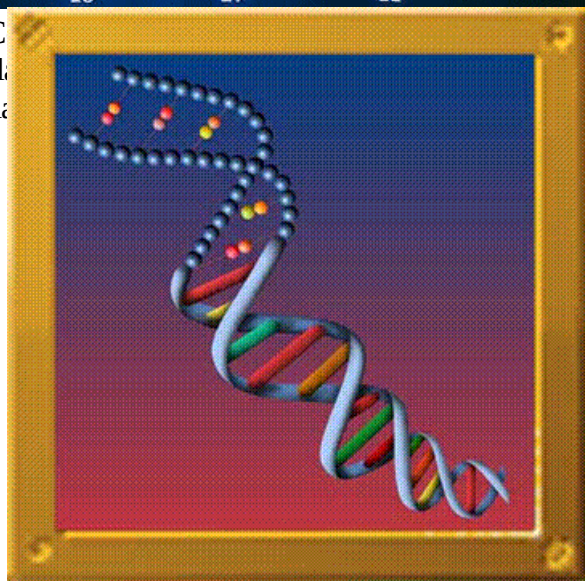


Figura 6: representación simbólica de la hélice de ADN.

La macromolécula de ADN está compuesta por bases *púricas y pirimidínicas,* la adenina, citosina, guanina y timina. La combinación y la secuencia de estas bases forma el **código genético**, único para cada ser vivo. Grupos de 3 bases forman un *codon*, y cada codon codifica una proteina; el código genético codifica todas las proteinas que forman parte de un ser vivo. Mientras que al codigo genético se le llama *genotipo*, al cuerpo que construyen esas proteinas, modificado por la presión ambiental y la historia vital, se llama *fenotipo*.

No toda la cadena de ADN codifican proteinas; las zonas que codifican proteinas se llaman *introne*s, las zonas que no lo hacen, *exones*. Un gen comienza con el sitio 3' o *aceptor* y termina con el sitio *5'* o donante. Proyectos como el del Genoma Humano tratan de identificar cuáles son estos genes, sus posiciones, y sus posibles alteraciones, que habitualmente conducen a enfermedades.

Todos estos hechos forman hoy en día la teoría del neo-darwinismo, que afirma que la historia de la mayoría de la vida está causada por una serie de procesos que actúan en y dentro de las poblaciones: reprodución, mutación, competición y selección. La evolución se puede definir entonces como cambios en el *pool* o conjunto genético de una población.

Un tema polémico, con opiniones variadas dependiendo de si se trata de informáticos evolutivos o de biólogos o geneticistas, es si la evolución optimiza o no. Según los informáticos evolutivos, la evolución optimiza, puesto que va creando seres cada vez más perfectos, cuyo culmen es el hombre; además, indicios de esta optimización se encuentran en el organismo de los animales, desde el tamaño y tasa de ramificación de las arterias, diseñada para maximizar flujo, hasta el metabolismo, que optimiza la cantidad de energía extraida de los alimentos. Sin embargo, los geneticistas y biólogos evolutivos afirman que la evolución no optimiza, sino que cambia y optimiza localmente en el espacio y el tiempo; evolución no significa progreso. Un organismo más evolucionado puede estar en desventaja competitiva con uno de sus ancestros, si se colocan en el ambiente del último.

2.1 Mecanismos de cambio en la evolución

Estos mecanismos de cambio serán necesarios para entender los algoritmos evolutivos, pues se trata de imitarlos para resolver problemas de ingeniería. Los mecanismos de cambio alteran la proporción de alelos de un tipo determinado en una población, y se dividen en dos tipos: los que disminuyen la variabilidad, y los que la

aumentan.

El principal mecanismo que disminuye la variabilidad son los siguientes:

- **Selección natural**: los individuos que tengan algún rasgo que los haga menos válidos para realizar su tarea de seres vivos, no llegarán a reproducirse, y por tanto su patrimonio genético desaparecerá del pool; algunos no llegarán ni siquiera a nacer. Esta selección sucede a muchos niveles: competición entre miembros de la especie (intraespecífica), competición entre diferentes especies, y competición predador-presa, por ejemplo. También es importante la selección sexual, en la cual las hembras eligen el mejor individuo de su especie disponible para reproducirse, y que da lugar a vistosos despliegues, luchas y documentales del National Geographic.
- **Deriva génica**: el simple hecho de que un alelo sea más común en la población que otro, causará que la proporción de alelos de esa población vaya aumentando en una población aislada, lo cual a veces da lugar a fenómenos de especiación.

Otros mecanismos aumentan la diversidd, y suceden generalmente a nivel molecular. Los más importantes son:

- **Mutación**: la mutación es una alteración del código genético, que puede suceder por múltiples razones. En muchos casos, las mutaciones las elimina la ADN-polimerasa, la navaja del ejército suizo del ADN, que igual duplica, que corrije, que desinvierte un cacho genético mal colocado, también puede sucer que ocurran en zonas no codificadoras. En muchos otros casos, las mutaciones, que cambian un nucleótido por otro, son letales, y los individuos ni siquiera llegan a desarrollarse, pero a veces se da lugar a la producción de una proteina que aumenta la supervivencia del individuo, y que por tanto es pasada a la descencencia. Las mutaciones son totalmente aleatorias, y son el mecanismo básico de generación de variedad genética. A pesar de lo que se piensa habitualmente, la mayoría de las mutaciones ocurren de forma natural, aunque

existen *sustancias mutagénicas* que aumentan su frecuencia.

- **Poliploidía**: mientras que las células normales poseen dos copias de cada cromosoma, y las células reproductivas 1 (haploides), puede suceder por accidente que alguna célula reproductiva tenga 2 copias; si se logra combinar con otra celula diploide o haploide dará lugar a un ser vivo con varias copias de cada cromosoma. La mayoría de las veces, la poliploidía da lugar a individuos con algún defecto (por ejemplo, el tener 3 copias del cromosoma 21 da lugar al mongolismo), pero en algunos casos se crean individuos viables. Un caso conocido de mutación fue el que sufrió (o disfrutó) el mosquito *Culex pipiens*, en el cual se duplicó un gen que generaba una enzima que rompía los organofosfatos, componentes habituales de los insecticidas;
- **Recombinación**: cuando las dos células sexuales, o gametos, una masculina y otra femenina se combinan, los cromosomas de cada una también lo hacen, intercambiándose genes, que a partir de ese momento pertenecerán a un cromosoma diferente. A veces también se produce traslocación dentro de un cromosoma; una secuencia de código se elimina de un sitio y aparece en otro sitio del cromosoma, o en otro cromosoma.
- **Flujo genético**: o intercambio de material genético entre seres vivos de diferentes especies. Normalmente se produce a través de un vector, que suelen ser virus o bacterias; estas incorporan a su material genético genes procedentes de una especie a la que han infectado, y cuando infectan a un individuo de otra especie pueden transmitirle esos genes a los tejidos generativos de gametos.

En resumen, la selección natural actúa sobre el fenotipo y suele disminuir la diversidad, haciendo que sobrevivan solo los individuos más aptos (aunque esta frase, bien mirada, es una redundancia: ¿Sobreviven los mejores o son los mejores porque sobreviven?); los mecanismos que generan diversidad y que combinan características actúan habitualmente sobre el genotipo.

3 Evolución de la informática evolutiva

Ya que se han descrito cuales son los mecanismos de la evolución, y una amplia gama de problemas que pueden o no tener relación entre sí, llega la hora de contar, desde sus principios, como evolucionó la idea de simular o imitar la evolución con el objeto de resolver problemas humanos.

Las primeras ideas, incluso antes del descubrimiento del ADN, vinieron de **Von Neumann,** uno de los mayores científicos de este siglo, que afirmó que la vida debía de estar apoyada por un código que a la vez describiera como se puede construir un ser vivo, y tal que ese ser creado fuera capaz de autoreproducirse; por tanto, un autómata o máquina autorreproductiva tendría que ser capaz, aparte de contener las instrucciones para hacerlo, de ser capaz de copiar tales instrucciones a su descendencia.

Sin embargo, no fue hasta mediados de los años cincuenta, cuando el rompecabezas de la evolución se había prácticamente completado, cuando **Box** comenzó a pensar en imitarla para, en su caso, mejorar procesos industriales. La técnica de Box, denominada EVOP (Evolutionary Operation), consistía en elegir una serie de variables que regían un proceso industrial. Sobre esas variables se creaban pequeñas variaciones que formaban un hipercubo, variando el valor de las variables una cantidad fija. Se probaba entonces con cada una de las esquinas del hipercubo durante un tiempo, y al final del periodo de pruebas, un comité humano decidía sobre la calidad del resultado. Es decir, se estaba aplicando mutación y selección a los valores de las variables, con el objeto de mejorar la calidad del proceso. Este procedimiento se aplicó con éxito a algunas industrias químicas.

Un poco más adelante, en 1958, **Friedberg** y sus colaboradores pensaron en

mejorar usando técnicas evolutivas la operación de un programa. Para ello diseñaron un código máquina de 14 bits (2 para el código de operación, y 6 para los datos y/o instrucciones); cada programa, tenía 64 instrucciones. Un programa llamado *Herman*, ejecutaba los programas creados, y otro programa, el *Teacher* o profesor, le mandaba a Herman ejecutar otros programas y ver si los programas ejecutados habían realizado su tarea o no. Los programas recibían su entrada en una posición de memoria, y debían depositar el resultado en otra posición de memoria, que era examinada al terminarse de ejecutar la última instrucción.

Para hacer evolucionar los programas, Friedberg hizo que en cada posición de memoria hubiera dos alternativas; para cambiar un programa, alternaba las dos instrucciones (que eran una especie de alelos), o bien reemplazaba una de las dos instrucciones con una totalmente aleatoria.

En realidad, lo que estaba haciendo es usar mutación para generar nuevos programas; al parecer, no tuvo más éxito que si hubiera buscado aleatoriamente un programa que hiciera la misma tarea. El problema es que la mutación solo, sin ayuda de la selección, hace que la búsqueda sea prácticamente una búsqueda aleatoria.

Más o menos simultáneamente, **Bremmerman** trató de usar la evolución para "entender los procesos de pensamiento creativo y aprendizaje", y empezó a considerar la evolución como un proceso de aprendizaje. Para resolver un problema, codificaba las variables del problema en una cadena binaria de 0s y 1s, y sometía la cadena a mutación, cambiando un bit de cada vez; de esta forma, estableció que la tasa ideal de mutación debía de ser tal que se cambiara un bit cada vez. Bremmerman trató de resolver problemas de minimización de funciones, aunque no está muy claro qué tipo de selección usó, si es que usó alguna, y el tamaño y tipo de la población. En todo caso, se llegaba a un punto, la "trampa de Bremmerman", en el cual la solución no mejoraba; en intentos sucesivos trató de añadir entrecruzamiento entre soluciones, pero tampoco ebtuvo buenos resultados. Una vez más, el simple uso de operadores

que creen diversidad no es suficiente para dirigir la búsqueda genética hacia la solución correcta.

El primer uso de procedimientos evolutivos en inteligencia artificial se debe a **Reed, Toombs y Baricelly**, que trataron de hacer evolucionar un tahur que jugaba a un juego de cartas simplificado. Las estrategias de juego consistían en una serie de 4 probabilidades de apuesta alta o baja con una mano alta o baja, con cuatro parámetros de mutación asociados. Se mantenía una población de 50 individuos, y aparte de la mutación, había intercambio de probabilidades entre dos padres. Es de suponer que los perdedores se eliminaban de la población (tirándolos por la borda). Aparte de, probablemente, crear buenas estrategias, llegaron a la conclusión de que el entrecruzamiento no aportaba mucho a la búsqueda.

Los intentos posteriores, ya realizados en los años 60, ya corresponden a los algoritmos evolutivos modernos, y se han seguido investigando hasta nuestros días. Algunos de ellos son simultáneos a los algoritmos genéticos, pero se desarrollaron sin conocimiento unos de otros. Uno de ellos, la programación evolutiva de **Fogel**, se inició como un intento de usar la evolución para crear máquinas inteligentes, que pudieran prever su entorno y reaccionar adecuadamente a él. Para simular una máquina pensante, se utilizó un *autómata celular*. Un autómata celular es un conjunto de estados y reglas de transición entre ellos, de forma que, al recibir una entrada, cambia o no de estado y produce una salida. En la figura 7, se muestra un autómata con estados etiquetados por letras mayúsculas y representados por círculos, las reglas de transición, con líneas que los unen, los símbolos de entrada son 0 y 1, y los símbolos de salida con letras mayúsculas.

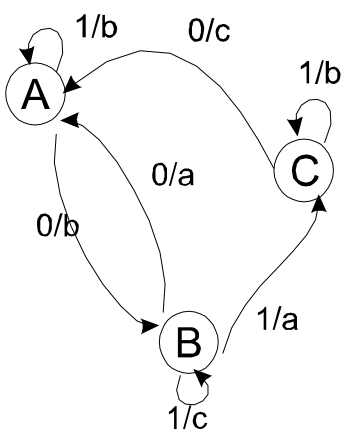


Figura 7: autómata celular como los usados por Fogel, con tres estados. Los caracteres encima de las líneas que unen los estados indican la entrada y la salida correspondiente.

Fogel trataba de hacer aprender a estos autómatas a encontrar regularidades en los símbolos que se le iban enviando. Como método de aprendizaje, usó un algoritmo evolutivo: una población de diferentes autómatas competían para hallar la mejor solución, es decir, predecir cual iba a ser el siguiente símbolo de la secuencia con un mínimo de errores; los peores 50% eran eliminados cada generación, y sustituidos por otros autómatas resultantes de una mutación de los existentes.

De esta forma, se lograron hacer evolucionar autómatas que predecían algunos números primos (por ejemplo, uno, cuando se le daban los números más altos, respondía siempre que no era primo; la mayoría de los números mayores de 100 son no primos). En cualquier caso, estos primeros experimentos demostraron el potencial de la evolución como método de búsqueda de soluciones novedosas.

Más o menos a mediados de los años 60, **Rechenberg** y **Schwefel** describieron las estrategias de evolución. Las estrategias de evolución son métodos de optimización paramétricos, que trabajan sobre poblaciones de cromosomas que son números reales. Hay diversos tipos de estrategias de evolución, que se verán más adelante, pero en la más común, se crean nuevos individuos de la población añadiendo un vector mutación a los cromosomas existentes en la población; en cada generación, se elimina un porcentaje de la población (especificado por los parámetros λ y μ), y los restantes generan la población total, mediante mutación y crossover. La magnitud del vector mutación se calcula adaptativamente.

Y, por supuesto, surgieron también los algoritmos genéticos. Pero esa es otra historia.

4 Algoritmo genético simple.

4.1 Introducción

John Holland desde pequeño, se preguntaba cómo logra la naturaleza, crear seres cada vez más perfectos, teniendo en cuenta además que todo se lleva a cabo a base de interacciones locales entre individuos, y entre estos y lo que les rodea. No sabía la respuesta, pero tenía una cierta idea de como hallarla: tratando de hacer pequeños modelos de la naturaleza, que tuvieran alguna de sus características, y ver cómo funcionaban, para luego extrapolar sus conclusiones a la totalidad. De hecho, ya de pequeño hacía simulaciones de batallas célebres con todos sus elementos: mapas que copiaba y que cubría luego de pequeños ejércitos que se enfrentaban entre sí.

En los años 50 entró en contacto con los primeros ordenadores, donde pudo llevar a cabo algunas de sus ideas, aunque no se encontró con un ambiente intelectual fértil para propagarlas. Fue a principios de los 60, en la Universidad de Michigan en Ann Arbor, donde, dentro del grupo *Logic of Computers,* sus ideas comenzaron a desarrollarse y a dar frutos. Y fue, además, leyendo un libro escrito por un biólogo evolucionista, R. A. Fisher, titulado *La teoría genética de la selección natural,* como comenzó a descubrir los medios de llevar a cabo sus propósitos de comprensión de la naturaleza. De ese libro aprendió que la evolución era una forma de adaptación más potente que el simple aprendizaje, y tomó la decisión de aplicar estas ideas para desarrollar programas bien adaptados para un fin determinado.

En esa universidad, Holland impartía un curso titulado *Teoría de sistemas adaptativos.* Dentro de este curso, y con una participación activa por parte de sus estudiantes, fue donde se crearon las ideas que más tarde se convertirían en los algoritmos genéticos.

Por tanto, cuando Holland se enfrentó a los algoritmos genéticos, los objetivos de su investigación fueron dos:

- imitar los procesos adaptativos de los sistemas naturales, y
- diseñar sistemas artificiales (normalmente programas) que retengan los mecanismos importantes de los sistemas naturales.

Unos 15 años más adelante, **David Goldberg,** actual delfín de los algoritmos genéticos, conoció a Holland, y se convirtió en su estudiante. Golberg era un ingeniero industrial trabajando en diseño de *pipelines*, y fue uno de los primeros que trató de aplicar los algoritmos genéticos a problemas industriales, en concreto al diseño de pipelines. Aunque Holland trató de disuadirle, porque pensaba que el problema era excesivamente complicado como para aplicarle algoritmos genéticos, Goldberg consiguió lo que quería, escribiendo un algoritmo genético en un ordenador personal Apple II. Estas y otras aplicaciones creadas por estudiantes de Holland convirtieron a los algoritmos genéticos en un campo suficientemente aceptado como para celebrar la primera conferencia en 1985, ICGA´85. Tal conferencia se sigue celebrando bianualmente.

4.2 Anatomía de un algoritmo genético simple

Los *algoritmos genéticos* son métodos sistemáticos para la resolución de problemas de búsqueda y optimización que aplican a estos los mismos métodos de la evolución biológica: selección basada en la población, reproducción sexual y mutación.

Los algoritmos genéticos son métodos de optimización, que tratan de resolver el mismo conjunto de problemas que se ha contemplado anteriormente, es decir, hallar $(x_i,...,x_n)$ tales que $F(x_i,...,x_n)$ sea máximo. En un algoritmo genético, $(x_i,...,x_n)$ se codifican en un cromosoma, que tiene toda la información necesaria para resolverlo (o toda la que no está implícita). Todas los operadores utilizados por un algoritmo genético se aplicarán sobre estos cromosomas, o sobre poblaciones de ellos. En el algoritmo genético va implícito el método para resolver el problema; son solo

parámetros de tal método los que están codificados, a diferencia de otros algoritmos evolutivos como la programación genética.

Las soluciones codificadas en un cromosoma *compiten* para ver cuál constituye la mejor solución (aunque no necesariamente la mejor de todas las soluciones posibles). El *ambiente,* constituido por las otras camaradas soluciones, ejercerá una presión selectiva sobre la población, de forma que sólo los mejor adaptados (aquellos que resuelvan mejor el problema) sobrevivan o leguen su material genético a las siguientes generaciones, igual que en la evolución de las especies. La diversidad genética se introduce mediante mutaciones y reproducción sexual.

En la Naturaleza lo único que hay que optimizar es la supervivencia, y eso significa a su vez maximizar diversos factores y minimizar otros. Un algoritmo genético, sin embargo, se usará para optimizar habitualmente para optimizar sólo una función, no diversas funciones relacionadas entre sí simultáneamente. Este tipo de optimización, denominada optimización multimodal, también se suele abordar con un algoritmo genético especializado.

Por lo tanto, un algoritmo genético consiste en lo siguiente: hallar de qué parámetros depende el problema, se codifican estos en un cromosoma, y se aplican los métodos de la evolución: selección y reproducción sexual con intercambio de información y alteraciones que generan diversidad. En las siguientes secciones se verán cada uno de los aspectos de un algoritmo genético.

4.3 Codificación de las variables

Los algoritmos genéticos requieren que el conjunto se codifique en un *gen* o *cromosoma* (en realidad, un cromosoma contiene varios genes, pero en este caso van a ser sinónimos). Para poder trabajar con estos genes en el ordenador, es necesario codificarlos en una *cadena*, es decir, una ristra de símbolos (números o letras) que

generalmente va a estar compuesta de 0s y 1s.

Por ejemplo, en esta cadena de bits, el valor del parámetro p_1 ocupará las posiciones 0 a 2, el p_2 las 3 a 5, etcétera, como aparece en la tabla 1. El número de bits usado para cada parámetro dependerá de la precisión que se quiera en el mismo o del número de opciones posibles que tenga ese parámetro. Por ejemplo, si se codifica una combinación del Mastermind, cada gen tendrá tantas opciones como colores halla, el número de bits elegido será el $\log_2$(número de colores).

p_1	p_2	p_3
001	010	10011

2 Codificación binaria de los parámetros de un problema.

Hay otras codificaciones posibles, usando alfabetos de diferente cardinalidad; sin embargo, uno de los resultados fundamentales en la teoría de algoritmos genéticos, el *teorema de los esquemas*, afirma que la codificación óptima, es decir, aquella sobre la que los algoritmos genéticos funcionan mejor, es aquella que tiene un alfabeto de cardinalidad 2.

Aquí se está codificando cada parámetro como un número entero de *n* bits. En realidad, se puede utilizar cualquier otra representación interna: BCD, código *Gray* y codificación en forma de números reales, por ejemplo.

La mayoría de las veces, una codificación correcta es la clave de una buena resolución del problema. Generalmente, la regla heurística que se utiliza es la llamada *regla de los bloques de construcción*, es decir, parámetros relacionados entre sí deben de estar cercanos en el cromosoma. Por ejemplo, si queremos codificar los pesos de una red neuronal, una buena elección será poner juntos todos los pesos que salgan de la misma neurona de la capa oculta (también llamada codificación *fregona*), como se indica en la figura. En esta, todos los pesos señalados con trazo doble se codificación mediante grupos de bits o bytes sucesivos en el cromosoma.

En todo caso, se puede ser bastante creativo con la codificación del problema, teniendo en cuenta la siguiente regla: variables del problema que estén juntas en el espacio-problema, deben de estar juntas en la codificación. Esto puede llevar a usar cromosomas bidimensionales, o tridimensionales, o con relaciones enre genes que no sean puramente lineales. En algunos casos, cuando no se conoce de antemano el número de variables del problema, caben dos opciones: codificar también el número de variables, o bien, lo cual es mucho más natural, crear un cromosoma que pueda variar de longitud. Para ello, claro está, se necesitan operadores genéticos que alteren la longitud.

4.4 Algoritmo genético propiamente dicho

Para comenzar la competición, se generan aleatoriamente una serie de cromosomas. El algoritmo genético procede de la forma siguiente:

Algoritmo genético

1. Evaluar la puntuación (*fitness*) de cada uno de los genes.
2. Permitir a cada uno de los individuos reproducirse, de acuerdo con su puntuación.
3. Emparejar los individuos de la nueva población, haciendo que intercambien material genético, y que alguno de los bits de un gen se vea alterado debido a una *mutación* espontánea.

Cada uno de los pasos consiste en una actuación sobre las cadenas de bits, es decir, la aplicación de un *operador* a una cadena binaria. Se les denominan, por razones obvias, *operadores genéticos*, y hay tres principales: *selección, crossover* o recombinación y *mutación;* aparte de otros operadores genéticos no tan comunes. Los veremos a continuación.

Un algoritmo genético tiene también una serie de parámetros que se tienen que fijar para cada ejecución, como los siguientes:

- **Tamaño de la población**: debe de ser suficiente para garantizar la diversidad de las soluciones, y además tiene que crecer más o menos con el número de bits del cromosoma, aunque nadie ha aclarado cómo tiene que hacerlo. Por supuesto, depende también del ordenador en el que se esté ejecutando.
- **Condición de terminación:** lo más habitual es que la condición de terminación sea la convergencia del algoritmo genético o un número prefijado de generaciones.

4.5 Evaluación y selección

Durante la evaluación, se decodifica el gen, convirtiéndose en una serie de parámetros de un problema, se halla la solución del problema a partir de esos parámetros, y se le da una puntuación a esa solución en función de lo cerca que esté de la mejor solución. A esta puntuación se le llama *fitness*.

Por ejemplo, supongamos que queremos hallar el máximo de la función $f(x)=1-x^2$, una parábola invertida con el máximo en x=1. En este caso, el único parámetro del problema es la variable x. La optimización consiste en hallar un x tal que f(x) sea máximo. Crearemos, pues, una población de cromosomas, cada uno de los cuales contiene una codificación binaria del parámero x. Lo haremos de la forma siguiente: cada byte, cuyo valor está comprendido entre 0 y 255, se transformará para ajustarse al intervalo [-1,1], donde queremos hallar el máximo de la función.

Valor binario	Decodificación	Evaluación f(x)
10010100	21	0,9559
10010001	19	0,9639
101001	-86	0,2604
1000101	-58	0,6636

El fitness determina siempre los cromosomas que se van a reproducir, y aquellos que se van a eliminar, pero hay varias formas de considerarlo para seleccionar la población de la siguiente generación:

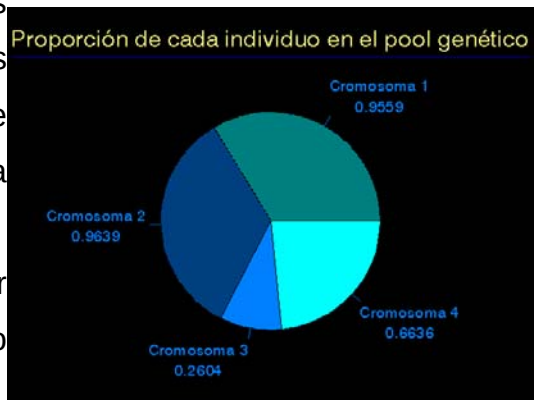


Figura 8: gráfico de tarta, con los fitness alcanzados por cada cromosoma y la proporción del pool genético destinado a la misma.

- Usar el orden, o rango, y hacer depender la probabilidad de permanencia o evaluación de la posición en el orden.
- Aplicar una operación al fitness para escalarlo.
- En algunos casos, el fitness no es una sola cantidad, sino diversos números, que tienen diferente consideración. Basta con que tal fitness forme un orden parcial, es decir, que se puedan comparar dos individuos y decir cuál de ellos es mejor. Esto suele suceder cuando se necesitan optimizar varios objetivos.

Una vez evaluado el fitness, se tiene que crear la nueva población teniendo en cuenta que los *buenos* rasgos de los mejores se transmitan a esta. Para ello, hay que seleccionar a una serie de individuos encargados de tan ardua tarea. Y esta selección, y la consiguiente reproducción, se puede hacer de dos formas principales:

- *Estado estacionario*: en este esquema se mantiene un porcentaje de la población para la siguiente generación. Se coloca toda la población por orden de fitness, y los *M* menos dignos son eliminados y sustituidos por la descendencia de alguno de los *M* mejores con algún otro individuo de la población. A este esquema se le pueden aplicar otros criterios; por ejemplo, se crea la descendencia de uno de los paladines/amazonas, y esta sustituye al más parecido entre los perdedores. Esto se denomina *crowding*, y fue introducido por **DeJong.** En nuestro caso, se eliminaría el cromosoma número 3, y se sustituiría por un descendiente del cromosoma número 2 y otro aleatorio,

escogido entre el 1 y el 4. En realidad, para este esquema se escoge un *crowding factor, CF*. Cuando nace una nueva criatura, se seleccionan CF individuos de la población, y se elimina al más parecido a la nueva criatura.

- *Rueda de ruleta*: se crea un *pool* genético formado por cromosomas de la generación actual, en una cantidad proporcional a su fitness. Si la proporción hace que un inviduo domine la población, se le aplica alguna operación de escalado. Dentro de este *pool*, se cogen parejas aleatorias de cromosomas y se emparejan, sin importar incluso que sean del mismo progenitor (para eso están otros operadores, como la mutación). Hay otras variantes: por ejemplo, en la nueva generación se puede incluir el mejor representante de la generación actual. En este caso, se denomina método *elitista*.

4.6 Crossover

Consiste en el intercambio de material genético entre dos cromosomas (a veces más, como el *operador orgía* propuesto por Eiben et al.). El *crossover* es el principal operador genético, hasta el punto que se puede decir que no es un algoritmo genético si no tiene *crossover*, y sin embargo puede serlo perfectamente sin mutación, según descubrió Holland. El *teorema de los esquemas* confía en él para hallar la mejor solución a un problema.

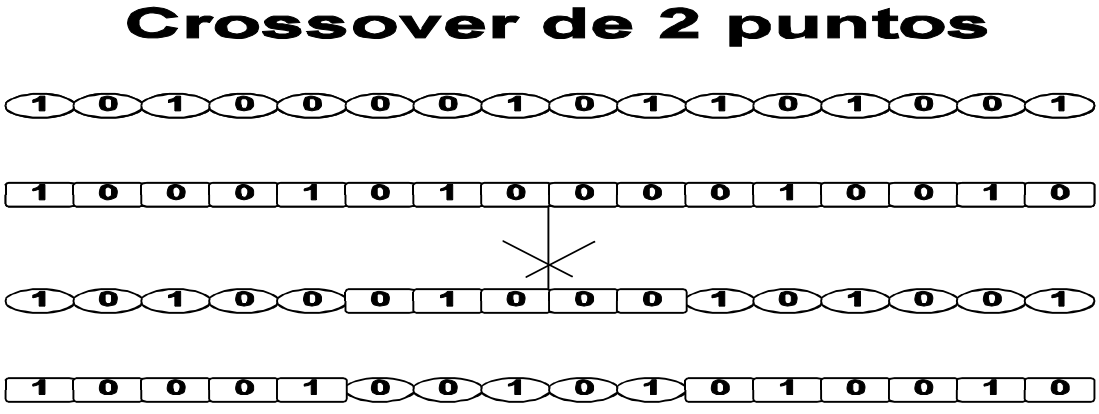
Para aplicar el crossover, entrecruzamento o recombinación, se escogen aleatoriamente dos miembros de la población. No pasa nada si se emparejan dos descencientes de los mismos padres; ello garantiza la perpetuación de un individuo con buena puntuación (y además, algo parecido ocurre en la realidad; es una práctica utilizada, por ejemplo, en la cría de ganado, llamada *inbreeding*, y destinada a potenciar ciertas características frente a otras). Sin embargo, si esto sucede demasiado a menudo, puede crear problemas: toda la población puede aparecer dominada por los descendientes de algún gen, que además puede tener caracteres no

deseados. Esto se suele denominar en otros métodos de optimización *atranque en un mínimo local*, y es uno de los principales problemas con los que se enfrentan los que aplican algoritmos genéticos.

En cuanto al teorema de los esquemas, se basa en la noción de *bloques de construcción*. Una buena solución a un problema está constituida por unos buenos bloques, igual que una buena máquina está hecha por buenas piezas. El crossover es el encargado de mezclar bloques buenos que se encuentren en los diversos progenitores, y que serán los que den a los mismos una buena puntuación. La presión selectiva se encarga de que sólo los buenos bloques se perpetúen, y poco a poco vayan formando una buena solución. El *teorema de los esquemas* viene a decir que la cantidad de *buenos bloques* se va incrementando con el tiempo de ejecución de un algoritmo genético, y es el resultado teórico más importante en algoritmos genéticos.

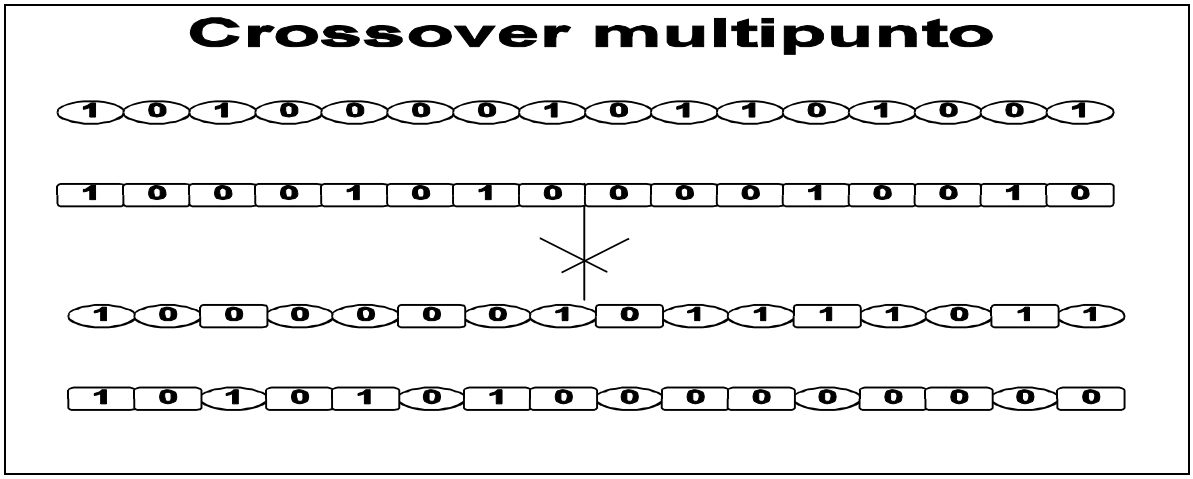
El intercambio genético se puede llevar a cabo de muchas formas, pero hay dos grupos principales

- *Crossover n-puntos*: los dos cromosomas se cortan por n puntos, y el material genético situado entre ellos se intercambia. Lo más habitual es un crossover de un punto o de dos puntos.



- *Crossover uniforme*: se genera un patrón aleatorio de 1s y 0s, y se intercambian

los bits de los dos cromosomas que coincidan donde hay un 1 en el patrón. O bien se genera un número aleatorio para cada bit, y si supera una determinada probabilidad se intercambia ese bit entre los dos cromosomas.



- *Crossover especializados*: en algunos problemas, aplicar aleatoriamente el crossover da lugar a cromosomas que codifican soluciones inválidas; en este caso hay que aplicar el crossover de forma que genere siempre soluciones válidas. Un ejemplo de estos son los operadores de crossover usados en el problema del viajante.

4.7 Mutación

En la Evolución, una mutación es un suceso bastante poco común (sucede aproximadamente una de cada mil replicaciones), como ya se ha visto en la sección 2.1. En la mayoría de los casos las mutaciones son letales, pero en promedio, contribuyen a la diversidad genética de la especie. En un algoritmo genético tendrán el mismo papel, y la misma frecuencia (es decir, muy baja).

Una vez establecida la frecuencia de mutación, por ejemplo, uno por mil, se examina cada bit de cada cadena cuando se vaya a crear la nueva criatura a partir de sus padres (normalmente se hace de forma simultánea al crossover). Si un número

generado aleatoriamente está por debajo de esa probabilidad, se cambiará el bit (es decir, de 0 a 1 o de 1 a 0). Si no, se dejará como está. Dependiendo del número de individuos que haya y del número de bits por individuo, puede resultar que las mutaciones sean extremadamente raras en una sola generación.

No hace falta decir que no conviene abusar de la mutación. Es cierto que es un mecanismo generador de diversidad, y por tanto la solución cuando un algoritmo genético está estancado, pero también es cierto que reduce el algoritmo genético a una búsqueda aleatoria. Siempre es más conveniente usar otros mecanismos de generación de diversidad, como aumentar el tamaño de la población, o garantizar la aleatoriedad de la población inicial.

Esta operación, junto con la anterior y el método de selección de ruleta, constituyen un *algoritmo genético simple,* SGA, introducido por Goldberg en su libro.

4.8 Otros operadores

4.8.1 Cromosomas de longitud variable

Hasta ahora se han descrito cromosomas de longitud fija, donde se conoce de antemano el número de parámetros de un problema. Pero hay problemas en los que esto no sucede. Por ejemplo, en un problema de clasificación, donde dado un vector de entrada, queremos agruparlo en una serie de clases, podemos no saber siguiera cuantas clases hay. O en diseño de redes neuronales, puede que no se sepa (de hecho, nunca se sabe) cuántas neuronas se van a necesitar. Por ejemplo, en un perceptrón hay reglas que dicen cuantas neuronas se deben de utilizar en la capa oculta; pero en un problema determinado puede que no haya ninguna regla heurística aplicable; tendremos que utilizar los algoritmos genéticos para hallar el número óptimo de neuronas. En este caso, si utilizamos una *codificación fregona*, necesitaremos un locus para cada neurona de la capa oculta, y el número de locus variará dependiendo

del número de neuronas de la capa oculta.

En estos casos, necesitamos dos operadores más: *añadir* y *eliminar*. Estos operadores se utilizan para añadir un locus, o eliminar un locus del cromosoma. La forma más habitual de añadir un locus es *duplicar* uno ya existente, el cual sufre mutación y se añade al lado del anterior. En este caso, los operadores del algoritmo genético simple (selección, mutación, crossover) funcionarán de la forma habitual, salvo, claro está, que sólo se hará crossover en la zona del cromosoma de menor longitud.

Estos operadores permiten, además, crear un *algoritmo genético de dos niveles*: a nivel de cromosoma y a nivel de alelo. Supongamos que, en un problema de clasificación, hay un alelo por clase. Se puede asignar una puntuación a cada alelo en función del número de muestras que haya clasificado correctamente. Al aplicar estos operadores, se duplicarán los alelos con mayor puntuación, y se eliminarán aquellos que hayan obtenido menor puntuación, o cuya puntuación sea nula.

Por ejemplo, en un problema de clasificación en el que hay que clasificar los puntos del cuadrado [0,10]x[0,10] en dos clases, *1* y *2*, que no son linealmente separables. Inicialmente no sabemos cuantos vectores son necesarios para clasificar estas clases. El algoritmo genético es capaz de hallar un número óptimo de vectores, a cada uno de los cuales se asigna una etiqueta de clase, tales que el error se hace mínimo, en este caso 4 vectores para la primera clase y 5 para la 2ª. Cada cromosoma estará compuesto por un diccionario o conjunto de vectores, cada uno de los cuales tienea asignada uan etiqueta de clase.

4.8.2 Operadores de nicho (ecológico)

Otros operadores imporantes son los operadores de *nicho*. Estos operadores están encaminados a mantener la diversidad genética de la población, de forma que cromosomas similares sustituyan sólo a cromosomas similares, y son especialmente

útiles en problemas con muchas soluciones; un algoritmo genético con estos operadores es capaz de hallar todos los máximos, dedicándose cada especie a un máximo.

Uno de las formas de llevar esto a cabo ya ha sido nombrada, la introducción del *crowding*. Otra forma es introducir una función de compartición, que indica cuán similar es un cromosoma al resto de la población. La puntuación de cada individuo se dividirá por esta función de compartición, de forma que se facilita la diversidad genética y la aparición de individuos diferentes.

También se pueden restringir los emparejamientos. Se puede, por ejemplo, restringir los emparejamientos a aquellos cromsomas que sean similares. Para evitar las malas consecuencias del inbreeding que suele aparecer en poblaciones pequeñas, estos periodos se intercalan con otros periodos en los cuales el emparejamiento es libre.

4.8.3 Operadores especializados

En una serie de problemas hay que restringir las nuevas soluciones generadas por los operadores genéticos, pues no todas las soluciones generadas van a ser válidas, sobre todo en los problemas con restricciones. Por ello, se aplican operadores que mantengan la estructura del problema. Otros operadores son simplemente generadores de diversidad:

- **Zap**: en vez de cambiar un solo bit de un cromosoma, cambia un gen completo de un cromosoma.
- **Creep:** este operador aumenta o disminuye en 1 el valor de un gen; sirve para cambiar suavemente y de forma controlada los valores de los genes.
- **Transposición:** similar al crossover y a la recombinación genética, pero dentro de un solo cromosoma; dos genes intercambian sus valores, sin afectar al resto del cromosoma.

4.8.4 Aplicando operadores genéticos

En toda ejecución de un algoritmo genético hay que decidir con qué frecuencia se va a aplicar cada uno de los algoritmos genéticos; en algunos casos, como en la mutación, se debe de añadir algún parámetro adicional, que indique con qué frecuencia se va a aplicar dentro de cada gen del cromosoma. La frecuencia de aplicación de cada operador estará en función del problema; teniendo en cuenta los efectos de cada operador, tendrá que aplicarse con cierta frecuencia o no. Generalmente, la mutación y otros operadores que generen diversidad se suele aplicar con poca frecuencia; la recombinación se suele aplicar con frecuencia alta.

En general, la frecuencia de los operadores no varía durante la ejecución del algoritmo, pero hay que tener en cuenta que cada operador es más efectivo en un momento de la ejecución. Por ejemplo, al principio, en la fase denominada de *exploración*, los más eficaces son la mutación y la recombinación; posteriormente, cuando la población ha convergido en parte, la recombinación no es útil, pues se está trabajando con individuos bastante similares, y es poca la información que se intercambia. Sin embargo, si se produce un estancamiento, la mutación tampoco es útil; y hay que aplicar otros operadores. En todo caso, se pueden usar operadores especializados.

Por ejemplo, en el algoritmo genético para jugar al MasterMind (http://kal-el.ugr.es/mastermind), se usan varios operadores genéticos: transposición, mutación y entrecruzamiento. Sin embargo, la mutación y el crossover deja de ser efectivo en el momento que la combinación que se ha jugado tiene los colores correctos, y en cualquier caso la tasa de mutación tendrá que ser mayor cuanto s menos colores haya averiguados; por eso las tasas varían durante la ejecución, convirtiéndose eventualmente en 0.

4.9 El zen y los algoritmos genéticos

Este es el título de un artículo que publicó **Goldberg** en la conferencia sobre algoritmos genéticos celebrada en el año 89 (ICGA 89), en donde da una serie de consejos para que se apliquen los algoritmos genéticos debidamente, y avisa a aquellos que se quieren apartar de la ortodoxia. Estos consejos son los siguientes

- ✓ *Deja que la Naturaleza sea tu guía*: dado que la mayoría de los problemas a los que se van a aplicar los algoritmos genéticos son de naturaleza no lineal, se mejor actuar como lo hace la naturaleza, aunque intuitivamente pueda parecer la forma menos acertada. *Si queremos desarrollar sistemas no lineales que busquen y aprendan, mejor que comencemos (como mínimo) imitando a sistemas que funcionan*. Y estos sistemas se hallan en la naturaleza.

- ✓ *Cuidado con el asalto frontal*: a veces se plantea el problema de pérdida de diversidad genética en una población de cromosomas. Hay dos formas de resolver este problema: aumentar el ritmo de mutación, lo cual equivale a convertir un algoritmo genético en un algoritmo de búsqueda aleatoria, o bien introducir mecanismos como el *sharing*, por el cual el fitness de un individuo se divide por el número de individuos similares a él. Este segundo método, más parecido al funcionamiento de la naturaleza, en la cual cada individuo, por bueno que sea, tiene que compartir recursos con aquellos que hayan resuelto el problema de la misma forma, funciona mucho mejor.

- ✓ *Respeta la criba de esquemas*: para ello, lo ideal es utilizar alfabetos con baja cardinalidad (es decir, con pocas letras) como el binario.

- ✓ *No te fíes de la autoridad central*: la Naturaleza actúa de forma distribuida, por tanto se debe de minimizar la necesidad de operadores que "vean" a da la población. Ello permite, además, una fácil paralelización del algoritmo genético. Por ejemplo, en vez de comparar el fitness de un individuo con todos los demás, se puede comparar sólo con los *vecinos*, es decir,

aquellos que estén, de alguna forma, situados cerca de él.

5 Práctica 1: Algoritmos genéticos con el programa `gwin2`

Gwin2, que aparece en casi todos sitios como WinGA, es un programa que permite ejecutar algoritmos genéticos simples, cambiar sus parámetros, y que incluso admite ampliaciones mediante la programación de nuevas funciones en Pascal. Fue realizado por I.R. Munro, de la Universidad de Hertfordshire. Está disponible en la página web *Zooland*, y en el sitio de ftp del autor.

WinGA es un programa que funciona en Windows 3.1 y Windows 95; según el autor necesita como mínimo 4 megas para funcionar. La práctica consistirá en ver los efectos de los diferentes parámetros en la ejecución de un algoritmo genético simple; en este caso, los únicos operadores admitidos son mutación y crossover, y dos tipos de selección diferentes. Existen unas 10 funciones ya programadas, pero, teóricamente, se pueden programar más.

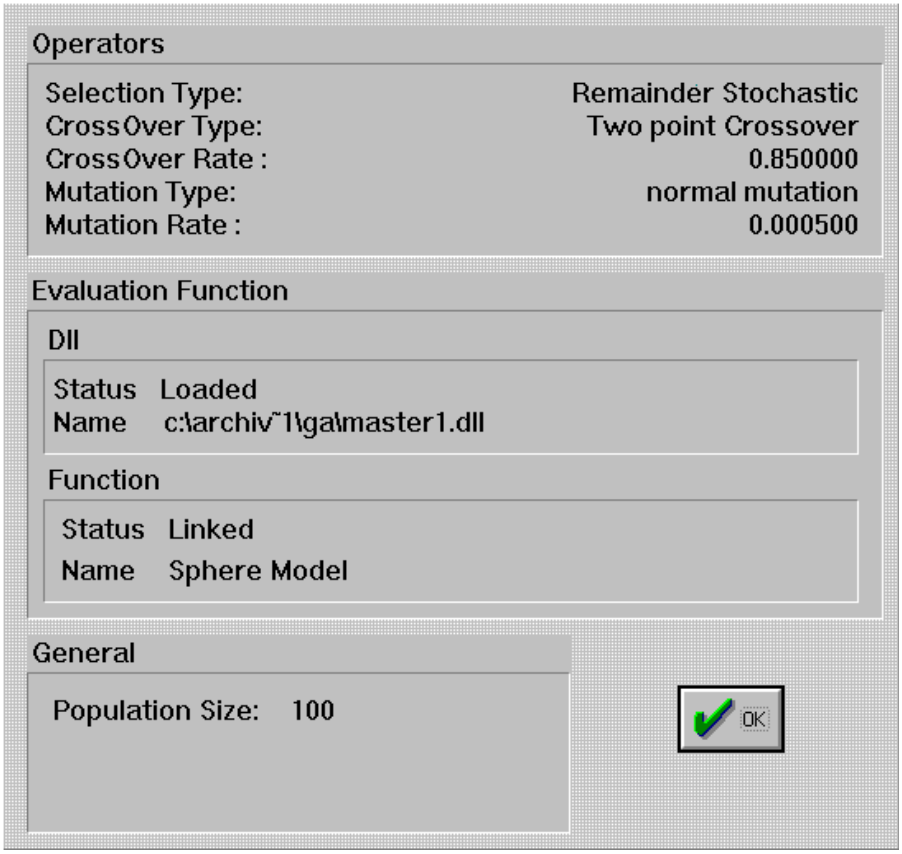
1. En el primer paso, se carga un fichero `.dll` usando la opción **Functions→Load DLL.** Cada `.dll` que contiene el código que evalúa las funciones; hay dos: `example1.dll`, y `master1.dll`, se puee escoger el segundo, que contiene más funciones. De ellas, se puede escoger **Sphere Model**, por ejemplo, que maximiza el cuadrado de una suma de parámetros; el número de parámetros es el que pide en el cuadro de diálogo.

2. Añadir *views,* diferentes ventanas que contienen información sobre el algoritmo

genético que se está ejecutando. Para ello, se elige la opción **Views** del menú y sucesivamente se van abriendo las cuatro ventanas.

3. Seleccionar los parámetros genéticos, y comprobar posteriormente el estado de lo seleccionado. Esto se hace desde el menú **Setup**. Se puede elegir, por ejemplo, **Selection→Remainder Stochastic, Combinations→One point crossover,** con la misma tasa, que indica la cantidad de la población a la que afecta esa operación, y **Normal Mutation** con la misma tasa.

Eligiendo **Setup→Status** aparece un cuadro que indica los parámetros elegidos.



4. Establecer el fichero de registro, en el cual se guardarán los datos de la ejecución actual, usando la opción **Reports→Log file.** En este fichero se puede guardar, por ejemplo, el mejor cromosoma, en formato binario, y darle un nombre cualquiera, como `primero.log`.

5. En este momento ya se puede ejecutar el algoritmo genético, eligiendo la opción **Run**, y se pueden ver el efecto de los diferentes parámetros sobre la ejecución del algoritmo. Probar, por ejemplo, lo siguiente:
   - Cambiar la mutación, y ver el efecto sobre la diversidad, es decir, el

número de cromosomas diferentes que aparece en el gráfico de **Current Distribution**.

- Reducir el crossover, y comprobar su efecto sobre la velocidad de convergencia. Probar también a cambiar el tipo de crossover, que en este tipo de problema no tendrá mucho efecto sobre el resultado.
- Cambiar el tamaño de la población, hasta encontrar el mínimo necesario para que el algoritmo converja en un número de generaciones razonable; cambiar también el número de generaciones.
- Cambiar el tipo de selección; en la selección de **Tournament** o torneo siempre se seleccionan los mejores, sin embargo, en la estocástica pueden desaparecer de la población.

***Ejercicio**: Encontrar la combinación de parámetros que halla el mínimo de la función anterior, y de alguna otra función del ejemplo, en un mínimo de tiempo. Tener en cuenta que, hasta cierto punto, se pueden intercambiar número de generaciones por el tamaño de la población.*

6 Práctica 2: Algoritmo genético interactivo en Java

En esta página Web, situada en el sitio denominado Evolvica (universidad de Erlangen), al cual se puede acceder desde la *página de aplicaciones evolutivas en Java,* `http://www.systemtechnik.tu-ilmenau.de/~pohlheim/EA_Java/ea_java.html`, se ejecuta un algoritmo genético interactivo. Para ello, es necesario acceder a la dirección Web -- mediante un browser que admita Java, como el Netscape Navigator 2 o el Internet Explorer 3.

En esta página, se hacen evolucionar formas bajo control del usuario; el usuario elige primero una forma hacia la cual tiende la evolución, y luego, en cada generación elige las formas que mutarán para dar finalmente, con un poco de suerte, la que se ha elegido inicialmente.

Se pueden modificar los parámetros, como por ejemplo, la tasa de mutación y el radio de mutación, y ver como varían las formas generadas.

***Ejercicio**: Por supuesto, conseguir generar la forma inicial en un mínimo de generaciones.*

7 Práctica 3: Algoritmos genéticos simples en Java

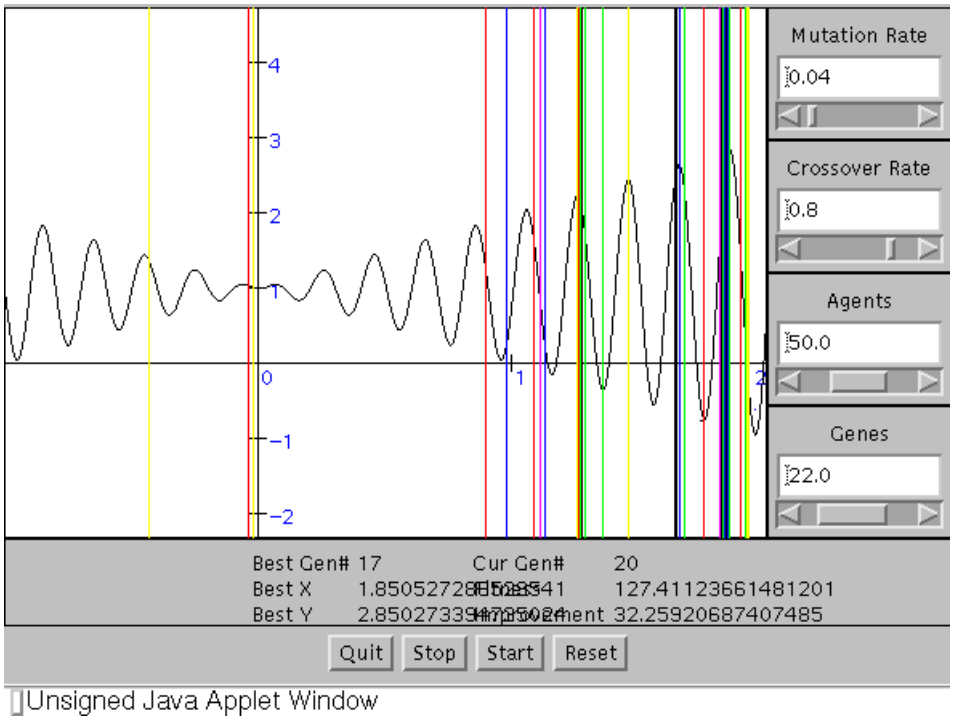


Figura 13 Algoritmo genético en Java de Ramsey et al.

En este applet, programado por **Ramsey et al.** de la Universidad de Arizona, que se halla en la dirección Web `http://ai.bpa.arizona.edu/~mramsey/ga.html`, se muestra un algoritmo genético simple que trata de hallar el máximo global de una función con muchos máximos y una sola variable. El valor de esa variable para los diferentes elementos de la población aparece como líneas verticales de color. Se puede variar, por ejemplo, la tasa de mutación; en problemas tan pequeños el crossover no tiene tanta importancia.

8 Práctica 4: Programación de un algoritmo genético simple

En esta práctica, se trata de programar un algoritmo genético simple, que incluya selección de tipo rueda de ruleta, mutación y entrecruzamiento de dos puntos. Utilizar cualquier lenguaje de programación (PERL, Pascal, C, C++, Tcl/Tk), y usar como función de evaluación alguna función simple, como la suma total de los componentes del cromosoma. Comparar la representación binaria con la representación de números reales, y comprobar la eficiencia de cada uno de ellos.

9 Los otros paradigmas: Estrategias de evolución y programación genética

Estos dos paradigmas siguen teniendo sus seguidores, aunque hoy en día se encuadran en general dentro de los algoritmos genéticos, difiriendo solo en una serie de detalles. Las estrategias de evolución se presentan principalmente en las conferencias Parallel Solving From Nature, PPSN, y los algoritmos de programación evolutiva en los Evolutionary Programming. Las primeras suelen ser en Europa (o Israel), y las segundas en algoritmos genéticos.

9.1 Estrategias de evolución

Las estrategias de evolución son algoritmos de optimización numérica. Lo que tratan de hallar es una serie de parámetros, $(x_1, x_2, ..., x_n)$, tales que $F(x_1, x_2, ..., x_n)$ sea mínimo. Las estrategias de evolución mantienen una población de vectores, que evolucionan mediante:

- **Mutación**: a cada $(x_1, x_2, ..., x_n)$ se añade un vector aleatorio de distribución normal con una varianza σ, que varía durante el algoritmo.
- **Recombinación:** se intercambia parte del vector.

Las estrategias de evolución, dependiendo del método de selección, reciben diferentes denominaciones con las letras μ y λ:

- (1,1) En esta EE, solo hay un padre y un hijo; el hijo sustituye al padre si su fitness es mejor.
- (μ + 1) Se mantienen en la población μ padres, y se genera un hijo de cada vez. Si tiene un fitness más alto, sustituye a uno de los padres.
- (μ,λ) λ hijos sustituyen a μ padres, caiga quien caiga. Al parecer, esta es la preferida por Schwefel.
- (μ + λ) En esta, los padres y los hijos conviven: se crean μ padres, a partir de ellos se crean λ hijos; se evalúan juntos, y quedan los μ mejores para la siguiente generación.

9.2 Programación evolutiva

Ultimamente, Fogel padre y Fogel hijo han formado una empresa, *Natural Selection*, que comercializa los productos de su investigación. Las principales diferencias con respecto a los demás algoritmos evolutivos es que desecha el crossover (por tanto, en puridad, no sería un algoritmo genético), ya que afirma que prácticamente no sirve para generar nuevas estructuras que sean adecuadas.

Otra diferencia es que se utiliza un método de selección estocástica: al fitness se le añada un ruido aleatorio, para que no siempre se seleccione el mejor, sino algún otro componente no-tan-bueno.

La programación evolutiva se aplica, aparte de a la evolución de autómatas, a otros muchos campos, como la evolución de redes neuronales.

10 Práctica 5: Estrategias de evolución en Java

Esta serie de *applets*, situados como la práctica 2 en el sitio Web denominado **Evolvica**, está en alemán, pero más o menos se puede apañar uno. Se trata de

optimizar los parámetros de funciones de 1 y 2 variables, hasta que alcancen el máximo; para ello se varían 1 o 2 parámetros utilizando estrategias de evolución.

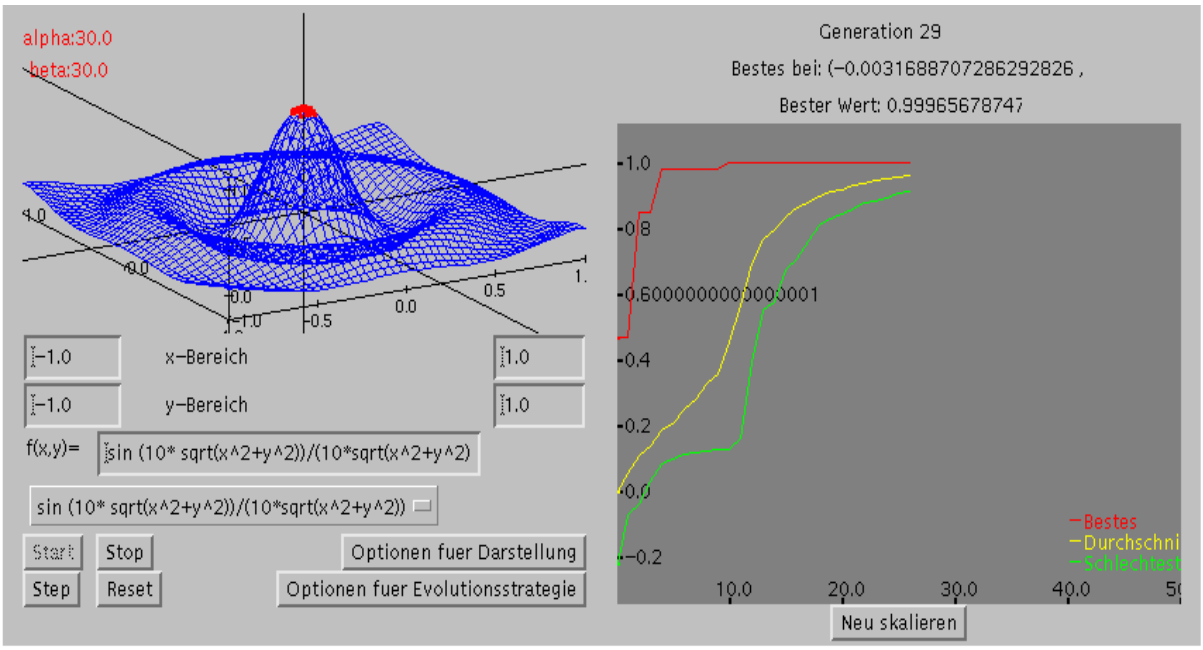


Figura 14 Ejemplo de estrategias evolutivas en Java (y en alemán).Los puntos rojos representan los valore de las individuos de la población.

En la función de dos variables, representada por una superficie, hay que hallar el punto *x,y* tal que el valor de la función sea máximo; los diversos puntos sobre la superficie representan los puntos que componen la población. En el panel de la derecha se representan el mejor, la media y el peor fitness de los miembros de la población.

Dos parámetros que se pueden modificar son el radio de mutación, que es la cantidad que se añade o sustrae a cada una de las variables, y los parámetros λ y μ, que son el número de padres y el número de "hijos" en una estrategia de evolución de tipo (λ+μ).

***Ejercicio**: Encontrar la combinación de parámetros que halla el máximo de la función en un mínimo de tiempo. Tener en cuenta que, hasta cierto punto, se pueden intercambiar número de generaciones por el tamaño de la población.*

11 Programación genética

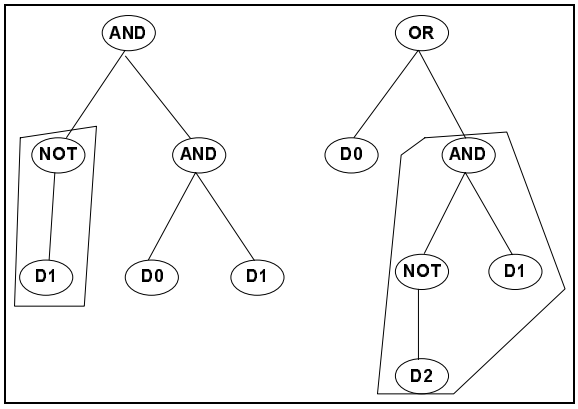
Dado que por reproducción, mutación y crossover se puede optimizar casi todo, a alguien (**John Koza**) se le ocurrió la idea de aplicar estos operadores a programas de ordenador. Para hacer esto hay una serie de problemas: los programas de ordenador tienen una sintaxis muy rígida, y si se aplica una mutación a algún programa, lo más probable es que de otro programa no válido. Si se hace en código máquina, la probabilidad de mutación letal es casi del 100%, dado que si se modifican las posiciones de memoria a las que se accede, el programa entrará probablemente en un bucle infinito o se colgará irremisiblemente.

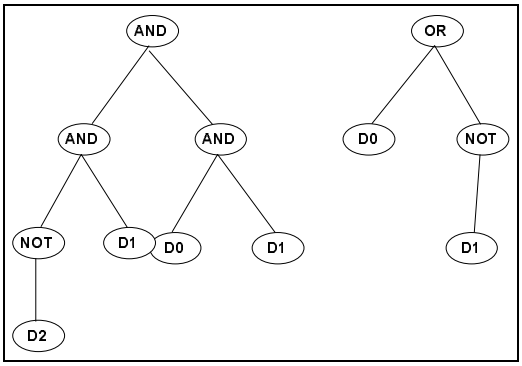
Según Koza, el representar soluciones a un programa mediante programas tiene una serie de ventajas:

- La solución al problema lleva ya implícita el algoritmo que lo soluciona; no es necesario fijar el algoritmo y optimizar los parámetrros del mismo.
- El representar una solución mediante un programa es más natural que usar una serie de número reales o una cadena binaria.
- Las soluciones no están predeterminadas ni en forma ni en tamaño. No todos los algoritmos genéticos usan cromosomas de longitud fija, pero la interpretación de esos cromosomas sí está fija de antemano.

Por lo tanto, para evitar los problemas de la representación clásica, y hacer posible la evolución de programas, como Friedberg había intentado hacer con anterioridad (véase capítulo 3) se hacía necesario crear, diseñar o adaptar un lenguaje de programación de tal forma que todas las formas posibles en las que pudiera cambiar por causa de la evolución fueran válidas, es decir, que no hubiera errores. La vía principal para hacer esto es tomar un número limitado de operadores y de variables a las que se puede acceder, de forma que cualquier mutación, al actuar sobre estas variables, dé otras variables u otros operadores. Esta es la opción tomada, por ejemplo, en el sistema **Tierra** de **Tom Ray.** Tierra trabaja en código máquina, pero si se desea utilizar lenguajes de alto nivel, las mutaciones, además de conservar el nivel más bajo (léxico), tienen que conservar la sintaxis, es decir, la forma como se ordenan los símbolos entre sí. Eso hace que halla que tratar las mutaciones de forma especial

en la programación genética, o simplemente eliminarlas. Lo más probable es que un crossover rompa la sintaxis de una frase. Se tiene que utilizar, por tanto, un tipo especial de crossover que conserve esta estructura.





La programación genética usa el lenguaje LISP, que tiene las siguientes ventajas para este tipo de trabajo:

- Hay una correspondencia directa entre la sintaxis (con miles de paréntesis) y la estructura en árbol que resulta del análisis sintáctico de la expresión, lo cual hace fácil pasar de una a la otra.
- LISP trata programas y datos de la misma forma; un programa puede crear y manipular otro programa fácilmente, y ejecutarlo mediante la orden `eval`.
- LISP permite manipular estructuras dinámicas, y alterar la forma del programa en tiempo de ejecución.

Los pasos seguidos para hacer evolucionar programas mediante programación genética son

Programación genética

1. Generar una población inicial de programas mediante composición aleatoria de las funciones y terminales del problema; previamente se tienen que haber elegido.
2. Llevar a cabo los siguientes pasos hasta que se satisfaga el criterio de terminación
    3. Ejecutar cada programa de la población y asignarle un valor de fitness

dependiendo de lo bien que resuelva el problema.

4. Crear una nueva población aplicando las siguientes operaciones primarias
   - Copiar programas existentes a la nueva población.
   - Crear nuevos programas recombinando genéticamente partes elegidas aleatoriamente de dos programas ya existentes.

5. Elegir como resultado el mejor programa que ha aparecido en cualquiera de las generaciones.

Hay varias formas de crear las estructuras iniciales; dado que la mutación prácticamente no se usa, toda la diversidad necesaria para resolvel el problema debe de estar en esta población inicial:

- *Full*: se crean programas cuyos árboles tienen todos la máxima profundidad permitida, y están completos, es decir, poseen 2^profundidad símbolos terminales (si todas las funciones son binarias).
- *Grow*: se generan árboles de programa, con cualquier tamaño menor o igual que el máximo, los árboles no tienen porque estar equilibrados.
- *Ramped half&half:* crea árboles completos, pero para cada una de las profundidades posibles. Al parecer, esta opción es la que da mejores resultados.

La selección que usa la programación genética es similar a los algoritmos genéticos, salvo que se suele preferir una selección basada en el orden, y la reproducción “estado estacionario”, donde se mantiene una parte de la población (por el uso del operador de copia).

Otros operadores que se usan ocasionalmente en programación genética son:

- **Mutación**: consiste en cambiar un nodo del árbol y todo lo que desciende de él por otro subárbol generado aleatoriamente.
- **Permutación:** consiste en intercambiar le posición de los subárboles de un nodo.
- **Editado**: se usa para simplificar los programas resultantes; consiste en

simplificar analíticamente las expresiones que aparecen en el programa; por ejemplo, (NOT (NOT X)) se convertirá en X, (AND X X) en (X), y así sucesivamente. Permite que las soluciones sean más compactas, aunque no está muy clara la ventaja de hacerlo durante la ejecución del algoritmo frente a hacerlo cuando acabe.

- **Encapsulado**: consiste en sustuir un subárbol por solo símbolo, de forma que el subárbol aparezca siempre junto en las operaciones de recombinación. Viene a ser un seguro ante ruptura de un subárbol beneficioso

Las aplicaciones de la programación genética son muy amplias; en realidad, todo lo que se pueda describir con un programa en LISP se puede programar genéticamente. Sin embargo, no está muy claro cuál es la ventaja frente a la simple programación de un algoritmo. El enfoque de la programación genética es similar al enfoque conexionista: se trata de encontrar un programa, es decir, la implementación de un algoritmo, que resuelva un problema para el que no se conoce. De hecho, la mayoría de las aplicaciones están en este campo. En este sentido, se parece más a las redes neuronales que a los algoritmos genéticos.

12 Práctica 6: Programación genética en Java

En este applet, situado en la Escuela Técnica Superior de Zürich, se trata de un problema de regresión simbólica, es decir, de encontrar una función que se acerque lo más posible

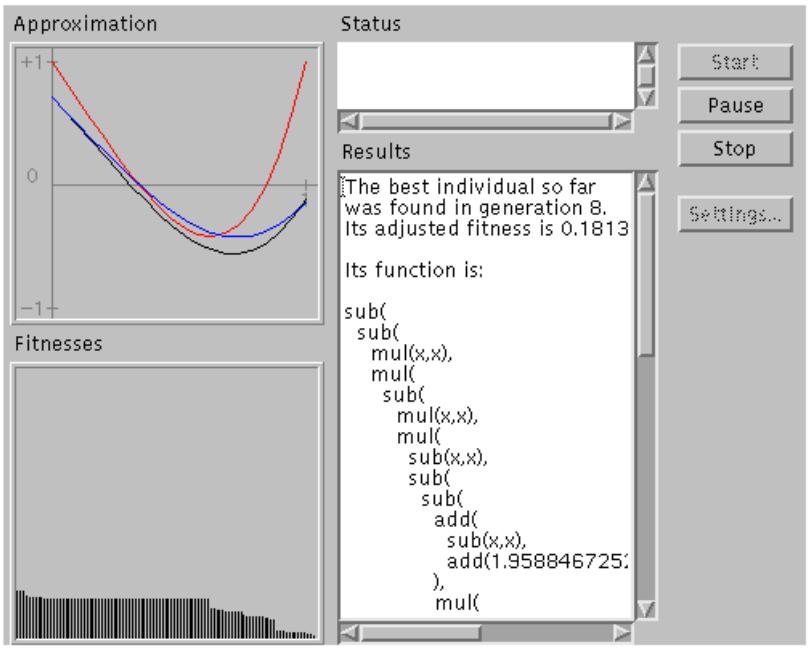


Figura 17 Programación genética en Java. Los paneles, en sentido de las agujas del reloj y empezando por arriba a la izquierda, representan la función a aproximar, el estado del algoritmo, el mejor programa evaluado hasta el momento, y el histograma de fitness para toda la población.

a la función problema; para ello se tratará de hacer evolucionar un programa que dé los mismos resultados de la función. Dicha función aparece en rojo, la mejor función hallada hasta el momento en azul, y el programa que se está evaluando en ese momento en negro. La evolución del fitness aparece en el gráfico de la derecha.

Como de costumbre, se pueden alterar los *settings* o parámetros de ejecución para ver como afecta al algoritmo. Por ejemplo, se puede cambiar el modo de selección, los símbolos terminales, es decir, los símbolos que se van a utilizar en los programas que evolucionan y la tasa de mutación.

***Ejercicio**: Encontrar la combinación de parámetros que halla el máximo de la función en un mínimo de tiempo. Probar diferentes símbolos terminales, para ver cual da una mejor aproximación funcional.*

13 Algunas aplicaciones de la computación evolutiva

Echando un vistazo a cualquiera de las conferencias específicas dedicadas al tema de algoritmos genéticos, o afines, como *Simulation of adaptive behavior*, *Parallel problem solving from nature*, o *Artificial life*, así como otras como *Evolutionary Programming*, se pueden encontrar miles de aplicaciones comerciales e industriales. Estas son algunas de ellas:

- *Elaboración de retratos robots de sospechosos*. Hace tiempo, **Richard Dawkins,** un biólogo evolucionista, incluyó con su libro *El relojero ciego* un programa, llamado *biomorfos*, donde se presentaban una serie de formas parecidas a insectos, y uno podía seleccionar con el ratón aquellas formas que se parecieran más a un insecto. Estas formas se reproducían, y el proceso se repetía unas cuantas generaciones. El resultado era que, en la generación final, aparecían unas formas sorprendentemente parecidas a insectos reales. Unos investigadores de la Universidad de Nuevo Mexico, **Caldwell & Johnston,** utilizando procedimientos policiales habituales, parametrizaron caras, de forma que podían ser almacenadas en un cromosoma. A la víctima se le presentaban

varias caras, y esta seleccionaba cuál o cuales de ellas se parecían más al sospechoso, asignándoles una puntuación. Utilizando procedimientos genéticos, los cromosomas correspondientes a esas caras se reproducían, obteniendo en una serie de generaciones un retrato robot bastante fiel del sospechoso.

- *Diseño de redes neuronales*: hay muchos enfoques diferentes para este problema, casi tantos como paradigmas neuronales. Nosotros hemos estado aplicando algoritmos genéticos para optimizar, principalmente, los pesos iniciales de una red, su tamaño y sus constantes de aprendizaje. El hallar unos buenos pesos iniciales es esencial en redes neuronales; unos malos pesos pueden provocar que el algoritmo de aprendizaje se quede parado en un mínimo local. La optimización de las constantes de aprendizaje también es importante; una constante grande puede hacer que olvide demasiado rápido, una pequeña que no aprenda en el tiempo necesario. Además, el tamaño debe de ser el ideal para el tipo de problema, si hay demasiadas neuronas el entrenamiento y la explotación serán muy lentas; si hay muy pocas el porcentaje de acierto puede no ser el apropiado. Para resolver este problema, se crea una población de redes neuronales, se entrenan, se prueban, y se asigna como fitness la exactitud en la clasificación de las muestras de entrada. En este caso, el aprendizaje neuronal se mezcla con el genético para dar una combinación ideal.

- *Aprendizaje basado en algoritmos genéticos*: este método, aunque tuvo bastante expansión a mediados de los 80, va empezando a desaparecer en las conferencias internacionales. Es un método singularmente parecido a las redes neuronales, pero en cierto sentido mucho más complicado. Habitualmente se aplica a clasificadores; se trata de que un clasificador sea capaz de clasificar correctamente los patrones de entrada que se le van introduciendo. Para ello, este clasificador tiene un sistema interno de paso de mensajes, mensajes que son recogidos por una serie de *reglas*, que a su vez pueden emitir mensajes. Sobre estas reglas, en función del número de veces que hayan sido útiles, se aplica el algoritmo genético. Hoy en día, por su mayor facilidad, este tipo de enfoques ha sido sustituido por las redes neuronales. Su representante más

notable es el ANIMAT de Wilson, un pequeño bicho artificial que aprendía a moverse en un bosque (virtual, claro).

14 Bibliografía comentada

La mayoría de la bibliografía está escrita en inglés, sin embargo, mientras que no haya traducciones o manuales en español disponibles, son la mejor referencia para el aprendiz del mester de algoritmos genéticos. Por supuesto, ahora también existe este tutorial (☺).

1. *Introducción a la Ciencia*, de Isaac Asimov, Biblioteca de Divulgación Científica Muy Interesante, Isaac Asimov. Este libro es imprescindible como introducción a todas las ciencias. Hay capítulos dedicados a la teoría de la Evolución, los componentes de la célula y el ADN, explicados de esa forma que solo Asimov sabe hacer.

2. *Handbook of Genetic Algorithms*: Lawrence Davis, ed. VNR Computer Library, New York.

3. *FAQ (preguntas frecuentemente preguntadas) del grupo de USENET* `talk.origins`, por diversos autores. Se puede encontrar en diversos sitios en Internet. Esta FAQ contiene documentos bastante completos con una introducción a la Biología evolutiva, sucesos de especiación observados en el laboratorio, y muchos otros documentos relacionados con la evolución.

4. *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*, de David B. Fogel, hijo del L. Fogel creador de la programación genética original. Es un libro más enfocado a demostrar, primero lo basados en la Naturaleza que está la programación evolutiva, y luego sus aplicaciones en problemas de inteligencia artificial.

5. *Genetic algorithms in search, optimization and machine learning*, David E. Goldberg, Addison Wesley, 1989.

6. *Genetic programming: On the programming of Computers by means of Natural Selection,* por John R. Koza. Nótese la paráfrasis del título con respecto al libro de Darwin. Un tocho considerable, pero útil para aquellos que practiquen la programación genética. Tiene también algunas secciones dedicadas a los algoritmos genéticos, pero se dedica principalmente a las aplicaciones.

7. *Artificial Life*, de Steven Levy. En este libro se dedica especial atención a las personas que han dedicado su vida a emular o simular en un ordenador la Vida, con mayúsculas; entre ellos, Von Neumann y Holland.

8. *An introduction to Genetic Algorithms,* por Melanie Mitchell. Uno de los últimos, está enfocado sobre todo a las aplicaciones de los algoritmos genéticos a la Vida Artificial. Puede servir también como libro de texto, porque tiene ejercicios y prácticas para realizar.

9. *Introducción a los algoritmos genéticos*, de Anselmo Pérez Serrada. Una introducción bastante completa, disponible en los sitios ENCORE, como `http://krypton.ugr.es/~encore`.