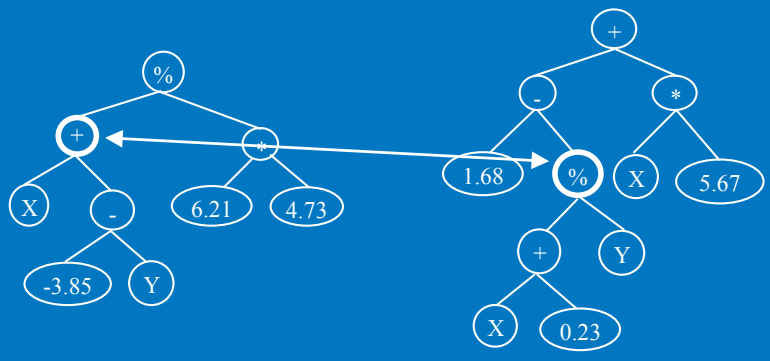
Introducción a los Algoritmos Genéticos y la Programación Genética

Marcos Gestal
Daniel Rivero
Juan Ramón Rabuñal
Julián Dorado
Alejandro Pazos



UNIVERSIDADE DA CORUÑA

This page intentionally left blank

Introducción a los Algoritmos Genéticos y la Programación Genética

Marcos Gestal
Daniel Rivero
Juan Ramón Rabuñal
Julián Dorado
Alejandro Pazos

A Coruña 2010

Universidade da Coruña
Servizo de Publicacións

Introducción a los Algoritmos Genéticos y la Programación Genética

Gestal, Marcos; Rivero, Daniel; Rabuñal, Juan Ramón; Dorado, Julián; Pazos, Alejandro

A Coruña, 2010
Universidade da Coruña, Servizo de Publicacións

Monografías, nº 140

Nº de páxinas: 76
Índice, páxinas: 5-6

ISBN: 978-84-9749-422-9
Depósito legal: C 3193-2010

CDU: 519.7 Cibernética matemática. Algortimos genéticos. Programación genética

Electronic version
published by

Edición
Universidade da Coruña, Servizo de Publicacións
http://www.udc.es/publicaciones

Segundo as normas da colección de *Monografías*, a presente publicación foi revisada e avaliada positivamente por dous expertos non pertencentes á UDC

Distribución

CONSORCIO EDITORIAL GALEGO. Estrada da Estación 70-A, 36818, A Portela. Redondela (Pontevedra). Tel. 986 405 051. Fax: 986 404 935. Correo electrónico: pedimentos@coegal.com

BREOGÁN. C/ Lanuza, 11. 28022, Madrid. Tel. 91-725 90 72. Fax: 91- 713 06 31.
Correo electrónico: breogan@breogan.org.
Web: http://www.breogan.org

Deseño da cuberta: Servizo de Publicacións da UDC
Imprime: Agencia Gráfica Gallega

Índice

Índice de ilustraciones

This page intentionally left blank

Prólogo

Desde un punto de vista general las técnicas de Computación Evolutiva, como los Algoritmos Genéticos o la Programación Genética, pueden considerarse como un conjunto de técnicas computacionales más ligadas en sus conceptos a los procesos biológicos que a las técnicas computacionales tradicionales. Es por ello que la aproximación a este tipo de técnicas en ocasiones puede convertirse en un camino plagado de espinas, en donde cada paso dado parece alejarnos más y más del oasis prometido. Sólo al final del camino, una vez introducidos los términos y conceptos básicos y asimiladas las analogías entre el mundo biológico y el mundo computacional, se alcanza a ver el gran potencial que ofrecen estas técnicas.

Con el objetivo de guiar los pasos a lo largo de ese camino surge la idea del presente libro. Para ello se plantea una introducción general a las técnicas de Computación Evolutiva más usuales, como son los Algoritmos Genéticos y la Programación Genética.

Por lo tanto, este libro no debería ser entendido como un documento exhaustivo acerca de las diferentes técnicas aquí mostradas. Debe considerarse más bien una referencia que sirva para introducir la terminología, los conceptos clave y una bibliografía de base, quedando en manos del lector profundizar en aquellos aspectos que considere de mayor interés.

Bajo la premisa expuesta, este libro está especialmente dirigido a aquellos estudiantes interesados en nuevas técnicas de resolución de problemas, así como a aquellos investigadores que pretenden comenzar su trabajo en estas o similares líneas de investigación.

Por último, deseamos al lector una amena lectura de estas páginas y que encuentre de utilidad el trabajo que se ha invertido en la elaboración de este libro.

M. Gestal, D. Rivero, J. R. Rabuñal, J. Dorado, A. Pazos
Laboratorio de Redes de Neuronas Artificiales y Sistemas Adaptativos
Facultad de Informática. Universidade da Coruña

This page intentionally left blank

Capítulo 1
Algoritmos Genéticos

1.1. Introducción

Los Algoritmos Genéticos son métodos adaptativos, generalmente usados en problemas de búsqueda y optimización de parámetros, basados en la reproducción sexual y en el principio de supervivencia del más apto (Fogel, 2000) (Fogel, 2006).

Más formalmente, y siguiendo la definición dada por Goldberg, "los Algoritmos Genéticos son algoritmos de búsqueda basados en la mecánica de selección natural y de la genética natural. Combinan la supervivencia del más apto entre estructuras de secuencias con un intercambio de información estructurado, aunque aleatorizado, para constituir así un algoritmo de búsqueda que tenga algo de las genialidades de las búsquedas humanas" (Goldberg, 1989).

Para alcanzar la solución a un problema se parte de un conjunto inicial de individuos, llamado población, generado de manera aleatoria. Cada uno de estos individuos representa una posible solución al problema. Estos individuos evolucionarán tomando como base los esquemas propuestos por Darwin sobre la selección natural, y se adaptarán en mayor medida tras el paso de cada generación a la solución requerida (Darwin, 2007).

1.2. Orígenes

Si algo funciona bien, ¿por qué no imitarlo?. La respuesta a esta pregunta nos lleva directamente a los orígenes de la computación evolutiva. Durante

millones de años las diferentes especies se han adaptado para poder sobrevivir en un medio cambiante. De la misma manera se podría tener una población de potenciales soluciones a un problema, de las que se irían seleccionando las mejores hasta que se adaptasen perfectamente al medio, en este caso el problema a resolver (Michalewicz & Fogel, 2000) (Bäck, 1996) (Whitley, 1994). En términos muy generales se podría definir la computación evolutiva como una familia de modelos computacionales inspirados en la evolución.

Más formalmente, el término de computación evolutiva se refiere al estudio de los fundamentos y aplicaciones de ciertas técnicas heurísticas basadas en los principios de la evolución natural (Tomassini, 1995). Estas técnicas heurísticas podrían clasificarse en 3 grandes categorías o grupos, dando lugar a la ecuación evolutiva recogida en la Figura 1.1.

Computación Evolutiva = Algoritmos Genéticos + Estrategias de Evolución + Programación Evolutiva

Figura 1.1: Ecuación Evolutiva

A continuación se detallarán un poco más los orígenes de cada una de las disciplinas participantes en la ecuación.

El desarrollo de los Algoritmos Genéticos se debe en gran medida a John Holland, investigador de la Universidad de Michigan. A finales de la década de los 60 desarrolló una técnica que imitaba en su funcionamiento a la selección natural. Aunque originalmente esta técnica recibió el nombre de "planes reproductivos", a raíz de la publicación en 1975 de su libro *"Adaptation in Natural and Artificial Systems"* (Holland, 1975) se conoce principalmente con el nombre de Algoritmos Genéticos. A grandes rasgos un Algoritmo Genético consiste en una población de soluciones codificadas de forma similar a cromosomas. Cada uno de estos cromosomas tendrá asociado un ajuste, valor de bondad o *fitness*, que cuantifica su validez como solución al problema. En función de este valor se le darán más o menos oportunidades de reproducción. Además, con cierta probabilidad se realizarán mutaciones de estos cromosomas (Goldberg, 2002).

Este proceso hará posible que los individuos genéticos tiendan hacia las soluciones a un problema dado, aunque las condiciones del espacio de búsqueda varíen con el transcurso del tiempo (Grefenstette, 1992)

Las bases de las Estrategias de Evolución fueron apuntadas en 1973 por Rechemberg en su obra *"Evolutionsstrategie: Optimierung Technisher Systeme nach Prinzipien der Biologischen Evolution"* (Rechenberg, 1973). Las

dos Estrategias de Evolución más empleadas son la (μ+λ)-ES y la (μ,λ)-ES. En la primera de ellas un total de μ padres producen λ descendientes, reduciéndose nuevamente la población a μ individuos (los padres de la siguiente generación) por selección de los mejores individuos. De esta manera los padres sobreviven hasta que son reemplazados por hijos mejores que ellos. En la (μ,λ)-ES la descendencia reemplaza directamente a los padres, sin hacer ningún tipo de comprobación.

La Programación Evolutiva surge principalmente a raíz del trabajo *"Artificial Intelligence Through Simulated Evolution"* de Fogel, Owens y Walsh, publicado en 1966 (Fogel, Ewens & Walsh, 1966). En este caso los individuos, conocidos aquí como organismos, son máquinas de estado finito. Los organismos que mejor resuelven alguna de las funciones objetivo obtienen la oportunidad de reproducirse. Antes de producirse los cruces para generar la descendencia se realiza una mutación sobre los padres.

A su vez la computación evolutiva puede verse como uno de los campos de investigación de lo que se ha dado en llamar *Soft Computing* (ver Figura 1.2).

Soft Computing	=	Computación Evolutiva	+	Redes Neuronas Artificiales	+	Lógica Difusa

Figura 1.2: Soft Computing

Tal y como se ha comentado anteriormente, la computación evolutiva tiene una fuerte base biológica. En sus orígenes los algoritmos evolutivos consistieron en copiar procesos que tienen lugar en la selección natural. Este último concepto había sido introducido, rodeado de mucha polémica, por Charles Darwin (Darwin, 1859). A pesar de que aún hoy en día no todos los detalles de la evolución biológica son completamente conocidos, existen algunos hechos apoyados sobre una fuerte evidencia experimental:

- La evolución es un proceso que opera, más que sobre los propios organismos, sobre los cromosomas. Estos cromosomas pueden ser considerados como herramientas orgánicas que codifican la vida o, visto al revés, una criatura es 'creada' decodificando la información contenida en los cromosomas.
- La selección natural es el mecanismo que relaciona los cromosomas (genotipo) con la eficiencia respecto al entorno de la entidad (fenotipo) que representan. Otorga a los individuos más adaptados al entorno un mayor número de oportunidades de reproducirse.

- Los procesos evolutivos tienen lugar durante la etapa de reproducción. Aunque existe una larga serie de mecanismos que afectan a la reproducción, los más comunes son la mutación, causante de que los cromosomas en la descendencia sean diferentes a los de los padres, y el cruce o recombinación, que combina los cromosomas de los padres para producir la descendencia.

Precisamente, sobre estos hechos se sustenta el funcionamiento de la Computación Evolutiva en general, y de los Algoritmos Genéticos en particular (Michalewicz, 1999).

1.3. Bases biológicas

En la naturaleza, los individuos de una población compiten constantemente con otros por recursos tales como comida, agua y refugio. Los individuos que tienen más éxito en la lucha por los recursos tienen mayores probabilidades de sobrevivir y generalmente una descendencia mayor. Al contrario, los individuos peor adaptados tienen un menor número de descendientes, o incluso ninguno. Esto implica que los genes de los individuos mejor adaptados se propagarán a un número cada vez mayor de individuos de las sucesivas generaciones.

La combinación de características buenas de diferentes ancestros puede originar, en ocasiones, que la descendencia esté incluso mejor adaptada al medio que los padres. De esta manera, las especies evolucionan adaptándose más y más al entorno a medida que transcurren las generaciones (Beasley, Bull & Martin, 1993).

Pero la adaptación de un individuo al medio no sólo está determinada por su composición genética. Influyen otros factores como el aprendizaje, en ocasiones adquirido por el método de prueba y error, en ocasiones adquirido por imitación del comportamiento de los padres, la cultura, la religión, etc. aspectos no contemplados en el Algoritmo Genético clásico, pero sí en otras variantes.

1.4. Codificación de problemas

Cualquier solución potencial a un problema puede ser presentada dando valores a una serie de parámetros. El conjunto de todos los parámetros (*genes* en la terminología de Algoritmos Genéticos) se codifica en una cadena de valores denominada *cromosoma* (ver Figura 1.3).

El conjunto de los parámetros representado por un cromosoma particular recibe el nombre de *genotipo*. El genotipo contiene la información necesaria para la construcción del organismo, es decir, la solución real al problema, denominada *fenotipo*. Por ejemplo, en términos biológicos, la información genética contenida en el ADN de un individuo sería el genotipo, mientras que la expresión de ese ADN (el propio individuo) sería el fenotipo.

Desde los primeros trabajos de John Holland la codificación suele hacerse mediante valores binarios. Se asigna un determinado número de bits a cada parámetro y se realiza una discretización de la variable representada por cada gen. El número de bits asignados dependerá del grado de ajuste que se desee alcanzar. Evidentemente no todos los parámetros tienen por qué estar codificados con el mismo número de bits. Cada uno de los bits pertenecientes a un gen suele recibir el nombre de *alelo.*

En la Figura 1.3 se muestra un ejemplo de un individuo binario que codifica 3 parámetros.

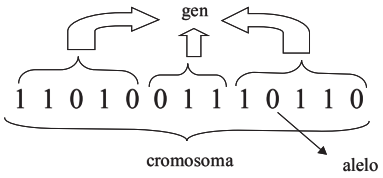
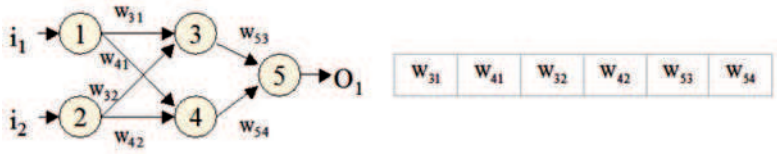


Figura 1.3: Individuo genético binario

Sin embargo, también existen representaciones que codifican directamente cada parámetro con un valor entero, real o en punto flotante. A pesar de que se acusa a estas representaciones de degradar el paralelismo implícito de las representaciones binarias, permiten el desarrollo de operadores genéticos más específicos al campo de aplicación del Algoritmo Genético.

En la Figura 1.4, se muestra un ejemplo típico de la aplicación de los Algoritmos Genéticos: cómo representar una red de neuronas artificiales para posteriormente realizar el proceso de optimización de los pesos sinápticos.

Codificar una red de neuronas artificiales en forma de cromosoma es tan sencillo como asignar un gen del cromosoma a cada uno de los pesos de la red. También se podrían añadir genes que indiquen el número de capas y el número de elementos de procesado en cada una.



Red de Neuronas Artificiales Codificación Individuo Genético

Figura 1.4: Codificación de una red de neuronas Artificiales

1.5. Algoritmo Principal

Los Algoritmos Genéticos trabajan sobre una población de individuos. Cada uno de ellos representa una posible solución al problema que se desea resolver. Todo individuo tiene asociado un ajuste de acuerdo a la bondad con respecto al problema de la solución que representa (en la naturaleza el equivalente sería una medida de la eficiencia del individuo en la lucha por los recursos).

Una generación se obtiene a partir de la anterior por medio de los operadores de reproducción. Existen 2 tipos:

- Cruce. Se trata de una reproducción de tipo sexual. Se genera una descendencia a partir del mismo número de individuos (generalmente 2) de la generación anterior. Existen varios tipos que se detallarán en un punto posterior.
- Copia. Se trata de una reproducción de tipo asexual. Un determinado número de individuos pasa sin sufrir ninguna variación directamente a la siguiente generación.

```
Inicializar población actual aleatoriamente
MIENTRAS no se cumpla el criterio de terminación
        crear población temporal vacía
        SI elitismo: copiar en población temporal mejores individuos
        MIENTRAS población temporal no llena
        seleccionar padres
        cruzar padres con probabilidad Pc
        SI se ha producido el cruce
                mutar uno de los descendientes (prob. Pm)
                evaluar descendientes
                añadir descendientes a la población temporal
        SINO
                añadir padres a la población temporal
        FIN SI
        FIN MIENTRAS
        aumentar contador generaciones
        establecer como nueva población actual la población temporal
FIN MIENTRAS
```

Figura 1.5: Funcionamiento de un Algoritmo Genético

El funcionamiento genérico de un Algoritmo Genético puede apreciarse en el pseudocódigo, reflejado en la Figura 1.5.

Si desea optarse por una estrategia elitista, los mejores individuos de cada generación se copian siempre en la población temporal, para evitar su pérdida.

A continuación comienza a generarse la nueva población en base a la aplicación de los operadores genéticos de cruce y/o copia. Una vez generados los nuevos individuos se realiza la mutación con una probabilidad *Pm*. La probabilidad de mutación suele ser muy baja, por lo general entre el 0.5% y el 2%.

Se sale de este proceso cuando se alcanza alguno de los criterios de parada fijados. Los más usuales suelen ser:

- Los mejores individuos de la población representan soluciones suficientemente buenas para el problema que se desea resolver.
- La población ha convergido. Un gen ha convergido cuando el 95% de la población tiene el mismo valor para él, en el caso de trabajar con codificaciones binarias, o valores dentro de un rango especificado en el caso de trabajar con otro tipo de codificaciones. Una vez que todos los genes alcanzan la convergencia se dice que la población ha convergido. Cuando esto ocurre la media de bondad de la población se aproxima a la bondad del mejor individuo.
- Se ha alcanzado el número de generaciones máximo especificado.

Sobre este algoritmo inicialmente propuesto por Holland se han definido numerosas variantes.

Quizás una de las más extendidas consiste en prescindir de la población temporal de manera que los operadores genéticos de cruce y mutación se aplican directamente sobre la población genética. Con esta variante el proceso de cruces varía ligeramente. Ahora no basta, en el caso de que el cruce se produzca, con insertar directamente la descendencia en la población. Puesto que el número de individuos de la población se ha de mantener constante, antes de insertar la descendencia en la población se le ha de hacer sitio. Es decir, para ubicar a los descendientes generados previamente se han de eliminar otros individuos de la población genética. Existen para ello diversas opciones, que se comentarán con más detalle en un punto posterior.

Evidentemente, trabajando con una única población no se puede decir que se pase a la siguiente generación cuando se llene la población, pues siempre está llena. En este caso el paso a la siguiente generación se producirá una vez que se hayan alcanzado cierto número de cruces y mutaciones. Este número dependerá de la tasa de cruces y mutaciones especificadas por el usuario y del tamaño de la población. Así, con una tasa de cruces del 90%, una tasa de

mutaciones del 2% y trabajando con 100 individuos se pasará a la siguiente generación cuando se alcanzasen 45 cruces (cada cruce genera 2 individuos con lo que se habrían insertado en la población 90 individuos, esto es el 90%) o 2 mutaciones.

Otra variación común consiste en la modificación del esquema de selección de los individuos que serán mutados. En el esquema mostrado, sólo los descendientes originados a partir de un cruce son mutados (proceso que imita los errores de transcripción del ADN que tienen lugar en la naturaleza); otra opción habitual es la selección aleatoria del individuo a mutar entre todos los que forman parte de la población genética.

1.6. Operadores genéticos

Para el paso de una generación a la siguiente se aplican una serie de operadores genéticos. Los más empleados son los operadores de selección, cruce, copia y mutación. En el caso de no trabajar con una población intermedia temporal también cobran relevancia los algoritmos de reemplazo. A continuación se verán en mayor detalle.

1.6.1. Selección

Los algoritmos de selección serán los encargados de escoger qué individuos van a disponer de oportunidades de reproducirse y cuáles no. Puesto que se trata de imitar lo que ocurre en la naturaleza, se ha de otorgar un mayor número de oportunidades de reproducción a los individuos más aptos. Por lo tanto, la selección de un individuo estará relacionada con su valor de ajuste. No se debe, sin embargo, eliminar por completo las opciones de reproducción de los individuos menos aptos, pues en pocas generaciones la población se volvería homogénea.

En cuanto a algoritmos de selección se refiere, estos pueden ser divididos en dos grandes grupos: probabilísticos y determinísticos. Ambos tipos de algoritmos basan su funcionamiento en el principio indicado anteriormente (permitir escoger una mayor cantidad de veces a los más aptos). Sin embargo, como su nombre indica, el primer tipo adjudica estas posibilidades con un importante componente basado en el azar. Es en este grupo donde se encuentran los algoritmos de selección por ruleta o por torneo que, dado su importancia por ser los más frecuentemente utilizados, se describen con detalle en esta sección. El segundo grupo engloba una serie de algoritmos que, dado el ajuste conocido de cada individuo, permite asignar a cada uno el número de veces que será

escogido para reproducirse. Esto puede evitar problemas de predominancia de ciertos individuos y cada uno de estos algoritmos presentan variaciones respecto al número de veces que se tomarán los mejores y peores y, de esta forma, se impondrá una presión en la búsqueda en el espacio de estados en la zona donde se encuentra el mejor individuo (en el caso de que se seleccionen más veces los mejores), o bien que se tienda a repartir la búsqueda por el espacio de estados, pero sin dejar de tender a buscar en la mejor zona (caso de repartir más la selección). Algunos de estos algoritmos son sobrante estocástico (Brindle, 1981) (Booker, 1982), universal estocástica (Baker, 1987) o muestreo determinístico.

Una opción bastante común consiste en seleccionar el primero de los individuos participantes en el cruce mediante alguno de los métodos expuestos en esta sección y el segundo de manera aleatoria.

1.6.1.1. Selección por ruleta

Propuesto por DeJong, es posiblemente el método más utilizado desde los orígenes de los Algoritmos Genéticos (Blickle & Thiele, 1995).

A cada uno de los individuos de la población se le asigna una parte proporcional a su ajuste de una ruleta, de tal forma que la suma de todos los porcentajes sea la unidad. Los mejores individuos recibirán una porción de la ruleta mayor que la recibida por los peores. Generalmente, la población está ordenada en base al ajuste, por lo que las porciones más grandes se encuentran al inicio de la ruleta. Para seleccionar un individuo basta con generar un número aleatorio del intervalo [0..1] y devolver el individuo situado en esa posición de la ruleta. Esta posición se suele obtener recorriendo los individuos de la población y acumulando sus proporciones de ruleta hasta que la suma exceda el valor obtenido.

Es un método muy sencillo pero ineficiente a medida que aumenta el tamaño de la población (su complejidad es $O(n^2)$). Presenta además el inconveniente de que el peor individuo puede ser seleccionado más de una vez.

1.6.1.2. Selección por torneo

La idea principal de este método de selección consiste en escoger a los individuos genéticos en base a comparaciones directas entre sus genotipos.

Existen dos versiones de selección mediante torneo, el torneo determinístico y el torneo probabilístico, que a continuación pasan a detallarse.

En la versión determinística se selecciona al azar un número p de individuos (generalmente se escoge p=2). De entre los individuos seleccionados se selecciona el más apto para pasarlo a la siguiente generación.

La versión probabilística únicamente se diferencia en el paso de selección del ganador del torneo. En vez de escoger siempre el mejor se genera un número aleatorio del intervalo [0..1], si es mayor que un parámetro *p* (fijado para todo el proceso evolutivo) se escoge el individuo más alto y en caso contrario el menos apto. Generalmente p toma valores en el rango $0.5 < p \leq 1$

Variando el número de individuos que participan en cada torneo se puede modificar la presión de selección. Cuando participan muchos individuos en cada torneo, la presión de selección es elevada y los peores individuos apenas tienen oportunidades de reproducción. Un caso particular es el *elitismo global.* Se trata de un torneo en el que participan todos los individuos de la población, con lo cual la selección se vuelve totalmente determinística. Cuando el tamaño del torneo es reducido, la presión de selección disminuye y los peores individuos tienen más oportunidades de ser seleccionados.

Elegir uno u otro método de selección determinará la estrategia de búsqueda del Algoritmo Genético. Si se opta por un método con una alta presión de selección se centra la búsqueda de las soluciones en un entorno próximo a las mejores soluciones actuales. Por el contrario, optando por una presión de selección menor se deja el camino abierto para la exploración de nuevas regiones del espacio de búsqueda.

Existen muchos otros algoritmos de selección. Unos buscan mejorar la eficiencia computacional, otros el número de veces que los mejores o peores individuos pueden ser seleccionados. Algunos de estos algoritmos son muestreo determinístico, escalamiento sigma, selección por jerarquías, estado uniforme, sobrante estocástico, brecha generacional, etc.

1.6.2. Cruce

Una vez seleccionados los individuos, éstos son recombinados para producir la descendencia que se insertará en la siguiente generación. Tal y como se ha indicado anteriormente, el cruce es una estrategia de reproducción sexual.

Su importancia para la transición entre generaciones es elevada puesto que las tasas de cruce con las que se suele trabajar rondan el 90%.

Los diferentes métodos de cruce podrán operar de dos formas diferentes. Si se opta por una estrategia destructiva los descendientes se insertarán en la población temporal aunque sus padres tengan mejor ajuste (trabajando con una única población esta comparación se realizará con los individuos a reemplazar). Por el contrario, utilizando una estrategia no destructiva la descendencia pasará a la siguiente generación únicamente si supera la bondad del ajuste de los padres (o de los individuos a reemplazar).

La idea principal del cruce se basa en que, si se toman dos individuos correctamente adaptados al medio y se obtiene una descendencia que comparta genes de ambos, existe la posibilidad de que los genes heredados sean precisamente los causantes de la bondad de los padres. Al compartir las características buenas de dos individuos, la descendencia, o al menos parte de ella, debería tener una bondad mayor que cada uno de los padres por separado. Si el cruce no agrupa las mejores características en uno de los hijos y la descendencia tiene un peor ajuste que los padres no significa que se esté dando un paso atrás. Optando por una estrategia de cruce no destructiva garantizamos que pasen a la siguiente generación los mejores individuos. Si, aún con un ajuste peor, se opta por insertar a la descendencia, y puesto que los genes de los padres continuarán en la población –aunque dispersos y posiblemente levemente modificados por la mutación–, en posteriores cruces se podrán volver a obtener estos padres, recuperando así la bondad previamente perdida.

Existen multitud de algoritmos de cruce. Sin embargo los más empleados son los que se detallarán a continuación:

- Cruce de 1 punto
- Cruce de 2 puntos
- Cruce uniforme

1.6.2.1. Cruce de 1 punto

Es la más sencilla de las técnicas de cruce. Una vez seleccionados dos individuos se cortan sus cromosomas por un punto seleccionado aleatoriamente para generar dos segmentos diferenciados en cada uno de ellos: la cabeza y la cola. Se intercambian las colas entre los dos individuos para generar los nuevos descendientes. De esta manera ambos descendientes heredan información genética de los padres.

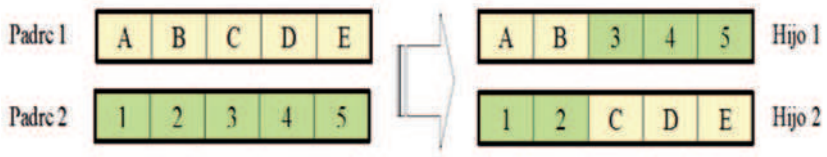


Figura 1.6: Cruce de un punto

En la Figura 1.6 se puede ver con claridad el proceso descrito anteriormente.

En la bibliografía suele referirse a este tipo de cruce con el nombre de SPX (Single Point Exchange)

1.6.2.2. Cruce de 2 puntos

Se trata de una generalización del cruce de 1 punto. En vez de cortar por un único punto los cromosomas de los padres, como en el caso anterior, se realizan dos cortes. Deberá tenerse en cuenta que ninguno de estos puntos de corte coincida con el extremo de los cromosomas para garantizar que se originen tres segmentos. Para generar la descendencia se escoge el segmento central de uno de los padres y los segmentos laterales del otro padre.

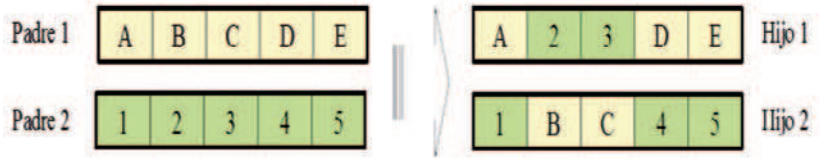


Figura 1.7: Cruce de dos puntos

Generalmente, es habitual referirse a este tipo de cruce con las siglas DPX (Double Point Crossover). En la Figura 1.7 se muestra un ejemplo de cruce por dos puntos.

Generalizando, se pueden añadir más puntos de cruce dando lugar a algoritmos de cruce multipunto. Sin embargo existen estudios que desaprueban esta técnica (DeJong & Spears, 1999). Aunque se admite que el cruce de 2 puntos aporta una sustancial mejora con respecto al cruce de un solo punto, el hecho de añadir un mayor número de puntos de cruce reduce el rendimiento del Algoritmo Genético. El problema principal de añadir nuevos puntos de cruce radica en que es más fácil que los segmentos originados sean corrompibles, es decir, que por separado quizás pierdan las características de bondad que poseían conjuntamente. Hay que evitar, por lo tanto, romper dichos segmentos, denominados bloques constructivos.

Sin embargo no todo son desventajas y añadiendo más puntos de cruce se consigue que el espacio de búsqueda del problema sea explorado con más intensidad.

1.6.2.3. Cruce uniforme

El cruce uniforme es una técnica completamente diferente de las vistas hasta el momento. Cada gen de la descendencia tiene las mismas probabilidades de pertenecer a uno u otro padre.

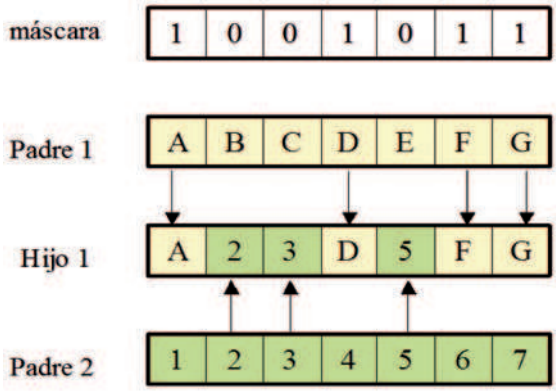


Figura 1.8: Cruce uniforme

Aunque se puede implementar de muy diversas formas, la técnica implica la generación de una máscara de cruce con valores binarios. Si en una de las posiciones de la máscara hay un 1, el gen situado en esa posición en uno de los descendientes se copia del primer padre. Si por el contrario hay un 0 el gen se copia del segundo padre. Para producir el segundo descendiente se intercambian los papeles de los padres, o bien se intercambia la interpretación de los unos y los ceros de la máscara de cruce.

Tal y como se puede apreciar en la Figura 1.8, la descendencia contiene una mezcla de genes de cada uno de los padres. El número efectivo de puntos de cruce es fijo pero será por término medio L/2, siendo L la longitud del cromosoma (número de alelos en representaciones binarias o de genes en otro tipo de representaciones).

La máscara de cruce puede no permanecer fija durante todo el proceso evolutivo. Se genera de manera aleatoria para cada cruce. Se suele referir a este tipo de cruce con las siglas UPX (Uniform Point Crossover).

1.6.2.4. Cruces específicos de codificaciones no binarias

Los tres tipos de cruce vistos hasta el momento son válidos para cualquier tipo de representación del genotipo. Si se emplean genotipos compuestos por valores enteros o reales pueden definirse otro tipo de operadores de cruce:

- Media: el gen de la descendencia toma el valor medio de los genes de los padres. Tiene la desventaja de que únicamente se genera un descendiente en el cruce de dos padres.
- Media geométrica: cada gen de la descendencia toma como valor la raíz cuadrada del producto de los genes de los padres. Presenta el problema añadido de qué signo dar al resultado si los padres tienen signos diferentes.
- Extensión: se toma la diferencia existente entre los genes situados en las mismas posiciones de los padres y se suma al valor más alto o se resta del valor más bajo. Solventa el problema de generar un único descendiente. Una variante basada en este cruce son los Algoritmos genéticos de evolución diferencial.

1.6.3. Algoritmos de Reemplazo

Cuando en vez de trabajar con una población temporal se hace con una única población, sobre la que se realizan las selecciones e inserciones, deberá tenerse en cuenta que para insertar un nuevo individuo deberá de eliminarse previamente otro de la población. Existen diferentes métodos de reemplazo:

- Aleatorio: el nuevo individuo se inserta en un lugar escogido de manera aleatoria en la población.
- Reemplazo de padres: se obtiene espacio para la nueva descendencia liberando el espacio ocupado por los padres.
- Reemplazo de similares: una vez obtenido el ajuste de la descendencia se selecciona un grupo de individuos (entre seis y diez) de la población con un ajuste similar. Se reemplazan aleatoriamente los que sean necesarios.
- Reemplazo de los peores: de entre un porcentaje de los peores individuos de la población se seleccionan aleatoriamente los necesarios para dejar sitio a la descendencia.

1.6.4. Copia

La copia es la otra estrategia reproductiva para la obtención de una nueva generación a partir de la anterior. A diferencia del cruce, se trata de una estrategia de reproducción asexual. Consiste simplemente en la copia de un individuo en la nueva generación.

El porcentaje de copias de una generación a la siguiente es relativamente reducido, pues en caso contrario se corre el riesgo de una convergencia prema-

tura de la población hacia ese individuo. De esta manera el tamaño efectivo de la población se reduciría notablemente y la búsqueda en el espacio del problema se focalizaría en el entorno de ese individuo.

Lo que generalmente se suele hacer es seleccionar dos individuos para el cruce y, si éste finalmente no tiene lugar, se insertan en la siguiente generación los individuos seleccionados.

1.6.5. Elitismo

El elitismo es un caso particular del operador de copia consistente en copiar siempre al mejor, o en su caso mejores, individuos de una generación en la generación siguiente. De esta manera se garantiza que el proceso de búsqueda nunca dará un paso atrás en cuanto a la calidad de la mejor solución obtenida, sino que un cambio en ésta siempre implicará una mejora.

Una variación de este proceso consiste en copiar al mejor o mejores individuos de una generación en la siguiente, únicamente cuando tras el paso de una generación no se haya mejorado con los operadores de cruce o mutación la mejor solución de la generación actual.

1.6.6. Mutación

La mutación de un individuo provoca que alguno de sus genes, generalmente uno sólo, varíe su valor de forma aleatoria.

Aunque se pueden seleccionar los individuos directamente de la población actual y mutarlos antes de introducirlos en la nueva población, la mutación se suele utilizar de manera conjunta con el operador de cruce. Primeramente se seleccionan dos individuos de la población para realizar el cruce. Si el cruce tiene éxito entonces uno de los descendientes, o ambos, se muta con cierta probabilidad *Pm*. Se imita de esta manera el comportamiento que se da en la naturaleza, pues cuando se genera la descendencia siempre se produce algún tipo de error, por lo general sin mayor trascendencia, en el paso de la carga genética de padres a hijos.

La probabilidad de mutación es muy baja, generalmente menor al 1%. Esto se debe sobre todo a que los individuos suelen tener un ajuste menor después de mutados. Sin embargo se realizan mutaciones para garantizar que ningún punto del espacio de búsqueda tenga una probabilidad nula de ser examinado.

Tal y como se ha comentado, la mutación más usual es el reemplazo aleatorio. Este consiste en variar aleatoriamente un gen de un cromosoma. Si se trabaja con codificaciones binarias, consistirá simplemente en negar un bit.

También es posible realizar la mutación intercambiando los valores de dos alelos del cromosoma. Con otro tipo de codificaciones no binarias existen otras opciones:

- Incrementar o decrementar a un gen una pequeña cantidad generada aleatoriamente.
- Multiplicar un gen por un valor aleatorio próximo a 1.

Aunque no es lo más común, existen implementaciones de Algoritmos Genéticos en las que no todos los individuos tienen los cromosomas de la misma longitud. Esto implica que no todos ellos codifican el mismo conjunto de variables. En este caso existen mutaciones adicionales como puede ser añadir un nuevo gen o eliminar uno ya existente.

1.7. Evaluación

Para el correcto funcionamiento de un Algoritmo Genético se debe de poseer un método que indique si los individuos de la población representan o no buenas soluciones al problema planteado. Por lo tanto, para cada tipo de problema que se desee resolver deberá derivarse un nuevo método, al igual que ocurrirá con la propia codificación de los individuos.

De esto se encarga la función de evaluación, que establece una medida numérica de la bondad de una solución. Esta medida recibe el nombre de ajuste. En la naturaleza el ajuste (o adecuación) de un individuo puede considerarse como la probabilidad de que ese individuo sobreviva hasta la edad de reproducción y se reproduzca. Esta probabilidad deberá estar ponderada con el número de individuos de la población genética.

En el mundo de los Algoritmos Genéticos se empleará esta medición para controlar la aplicación de los operadores genéticos. Es decir, permitirá controlar el número de selecciones, cruces, copias y mutaciones llevadas a cabo.

La aproximación más común consiste en crear explícitamente una medida de ajuste para cada individuo de la población. A cada uno de los individuos se le asigna un valor de ajuste escalar por medio de un procedimiento de evaluación bien definido. Tal y como se ha comentado, este procedimiento de evaluación será específico del dominio del problema en el que se aplica el Algoritmo Genético. También puede calcularse el ajuste mediante una manera ‘co-evolutiva’. Por ejemplo, el ajuste de una estrategia de juego se determina aplicando esa estrategia contra la población entera (o en su defecto una muestra) de estrategias de oposición.

Se pueden diferenciar cuatro tipos de ajuste o fitness (Koza, 1992):

- **Fitness Puro:** *r(i,t)*

  Es la medida de ajuste establecida en la terminología natural del propio problema. Por ejemplo, supóngase una población de hormigas que deben llenar la despensa de cara al invierno. La bondad de cada hormiga será el número de piezas de comida llevadas por ella hasta el hormiguero.

$$r(i,t) = \sum_{j=1}^{N_c} \left| s(i,j) - c(i,j) \right|$$

  R(i,t) : bondad del individuo i en la generación t

  S(i,j) : valor deseado para individuo i en el caso j

  C(i,j) : valor obtenido por el individuo i en el caso j

  Nc : número de casos

  En los problemas de maximización, como sería el de las hormigas mencionado anteriormente, los individuos con un fitness puro elevado serán los más interesantes. Al contrario, en los problemas de minimización interesarán los individuos con un fitness puro reducido.

- **Fitness Estandarizado**. *s(i,t)*

  Para solucionar esta dualidad ante problemas de minimización o maximización se modifica el ajuste puro de acuerdo a la siguiente fórmula:

$$s(i,t) = \begin{cases} r(i,t) & \text{minimización} \\ r_{max} - r(i,t) & \text{maximización} \end{cases}$$

  En el caso de problemas de minimización se emplea directamente la medida de fitness puro. Si el problema es de maximización se resta de una cota superior r_{max} del error el fitness puro.
  Empleando esta métrica la bondad de un individuo será menor cuanto más cercano esté a cero el valor del ajuste. Por lo tanto, dentro de la generación t, un individuo i siempre será mejor que uno j si se verifica que s(i,t) < s(j,t).

- **Fitness Ajustado**. a*(i,t)*

  Se obtiene aplicando la siguiente transformación al fitness estandarizado:

$$a(i,t) = \frac{1}{1 + s(i,t)}$$

De esta manera, el fitness ajustado tomará siempre valores del intervalo (0,1]. Cuando más se aproxime el fitness ajustado de un individuo a 1 mayor será su bondad.

- **Fitness Normalizado**. *n(i,t)*

  Los diferentes tipos de fitness vistos hasta ahora hacen referencia únicamente a la bondad del individuo en cuestión. El fitness normalizado introduce un nuevo aspecto: indica la bondad de una solución con respecto al resto de soluciones representadas en la población. Se obtiene de la siguiente forma (considerando una población de tamaño *M*):

$$n(i,t) = \frac{a(i,t)}{\sum_{k=1}^{M} a(k,t)}$$

  Al igual que el fitness ajustado, siempre tomará valores del intervalo [0,1], con mejores individuos cuanto más próximo esté a la unidad. Pero a diferencia de antes, un valor cercano a 1 no sólo indica que ese individuo represente una buena solución al problema, sino que además es una solución destacadamente mejor que las proporcionadas por el resto de la población.
  La suma de los valores de fitness normalizado de una población da siempre 1.
  Este tipo de ajuste es empleado en la mayoría de los métodos de selección proporcionales al fitness.

1.8. Ejemplos prácticos

1.8.1. Resolución de un sistema de ecuaciones

Un ejemplo sencillo, pero en el que se aprecia la potencia de resolución de los algoritmos genéticos, es el relativo a la resolución de un sistema de ecuaciones.

1.8.1.1. Descripción del problema

En la Figura 1.9 se muestra un sistema con n ecuaciones y un total de m incógnitas (junto con su representación matricial). Los coeficientes de las incógnitas X_i están representados por los valores $a_{i,j}$, mientras que el término independiente de cada ecuación se representa con el valor b_i.

$$a_{11} \cdot X_1 + a_{12} \cdot X_2 + \cdots + a_{1m} \cdot X_m = b_1$$
$$a_{21} \cdot X_1 + a_{22} \cdot X_2 + \cdots + a_{2m} \cdot X_m = b_2 \quad \begin{pmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nm} \end{pmatrix} \cdot \begin{pmatrix} X_1 \\ \vdots \\ X_m \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}$$
$$\vdots$$
$$a_{n1} \cdot X_1 + a_{n2} \cdot X_2 + \cdots + a_{nm} \cdot X_m = b_n$$

Figura 1.9: Sistema de n-ecuaciones y m incógnitas

1.8.1.2. Codificación del problema

La codificación de un sistema de ecuaciones es realmente sencilla mediante algoritmos genéticos. Deberá tenerse en cuenta que cada individuo ha de proporcionar una solución válida al problema. Por lo tanto, si el sistema de ecuaciones presenta m incógnitas, cada individuo genético deberá codificar cada una de esas incógnitas. Podría optarse por diferentes codificaciones pero, para mostrar paso a paso la resolución del problema propuesto, puede definirse que cada gen represente, mediante valores reales, cada uno de los valores asignados a las incógnitas del sistema en la solución. Así, en la Figura 1.10 se muestra un individuo genético válido para la resolución del citado problema.

X_1= 1.25	X_2= -5.87	X_3= 15.68	…	X_m= 258.6

Figura 1.10: Individuo Genético

Tal y como se ha comentado, cada uno de los genes representará el valor asignado a cada incógnita para la resolución del sistema planteado. Asimismo, durante la fase de codificación se podrá establecer un límite máximo y mínimo para el valor de los genes.

1.8.1.3. Función de Evaluación

Una vez codificado el problema, será necesario proporcionar una función de evaluación que permita determinar cómo de buena es la solución proporcionada por cada uno de los individuos de la población genética.

A medida que se incrementa el número de ecuaciones y el número de incógnitas, la resolución por métodos puramente matemáticos de este tipo de problemas se complica considerablemente. Sin embargo, trabajando con al-

goritmos genéticos debe tenerse en cuenta que no es necesario proporcionar un método de resolución, sino un método que permita determinar si una solución es buena o no, y en qué medida. Así, una aproximación perfectamente válida para evaluar la bondad de un individuo consistiría en lo siguiente. En primer lugar, deberán reemplazarse las incógnitas del sistema de ecuaciones por los valores proporcionados por cada uno de los genes. A continuación se realizarían los cálculos especificados en la parte izquierda de la ecuación para obtener un valor numérico. Dicho valor numérico podrá compararse con los términos independientes especificados en el sistema de ecuaciones. Cuanto más cercano esté (en valor absoluto) el valor obtenido al valor del término independiente, mejor solución representa el individuo genético. Evidentemente, la solución ideal consistiría en un conjunto de valores (las soluciones del sistema) tales que, si las incógnitas se substituyesen por ellos, los valores obtenidos fuesen idénticos a los términos independientes.

Por lo tanto, la función de evaluación podría definirse como el sumatorio de las diferencias en valor absoluto entre cada uno de los términos independientes del sistema y los términos independientes obtenidos al sustituir los valores del genotipo en la ecuación correspondiente. Visto en forma de pseudocódigo se tendría la siguiente función:

```
coef ← Matriz Coeficientes a_ij
b ← Matriz de Términos independientes b_i

fitness ← 0

Para cada ecuación i
hacer
    aux ← 0

    Para cada incógnita j de la ecuación i
    hacer
    aux ← aux + genotipo(j) · coef(i)(j)
    finPara

    fitness ← fitness + abs(b(i) – aux)

finPara

devolver fitness
```

1.8.1.4. Ejemplo de resolución paso a paso

Siguiendo la codificación y método de evaluación anteriormente comentados, a continuación se mostrará paso a paso la resolución del sistema de ecuaciones planteado en la Figura 1.11.

$$3 \cdot x + 8 \cdot y + 2 \cdot z = 25$$

$$x - 2 \cdot y + 4 \cdot z = 12$$

$$-5 \cdot x + 3y + 11 \cdot z = 4$$

Solución:

x = 4.643

y = 0.821

z = 2.250

Figura 1.11: Sistema de ecuaciones de partida

Tal y como se ha comentado anteriormente, la fortaleza de los algoritmos evolutivos reside en la evolución en paralelo de múltiples soluciones. Sin embargo, con el objetivo de hacer comprensible la simulación se escogerán valores fuera del rango normal de operación de este tipo de algoritmos, como puede ser el tamaño de población. Así, en este caso los parámetros de configuración del algoritmo genético serán los siguientes:

- Tamaño de población: 10 individuos
- Elitismo: 2 individuos
- Algoritmo selección: torneo determinístico+aleatorio
- Algoritmo cruce: 1 punto
- Tasa de Cruce (P_c): 90%
- Tasa de Mutación (P_m): 2.5%
- Algoritmo mutación: puntual

El primer paso en la ejecución del algoritmo genético consistirá en la inicialización de la población genética. En este caso, por simplicidad podemos suponer que se restringen los valores de los genes al rango [-10..10] y que se permiten únicamente 2 valores decimales. Así, la población inicial junto con sus valores de aptitud (fitness puro y fitness normalizado) podría ser la siguiente:

	Individuo			Fitness Puro	Fitness Normalizado
1	-2,68	-7,68	6,53	3,57	0,0046
2	1,36	0,02	1,62	14,64	0,0187
3	-6,36	5,67	0,74	28,97	0,0370
4	2,5	-3,18	2,36	32	0,0408
5	-5,20	2,69	-1,27	33,18	0,0808
6	2,79	9,54	1,25	63,32	0,0423
7	9,89	1,28	7,67	91,02	0,1161
8	4,5	0,69	-4,58	117,15	0,1495
9	-4,08	6,25	8,63	166,04	0,2119
10	4,68	-8,72	-6,45	233,81	0,2983

Población inicial y valores de aptitud. Generación 0

Obsérvese que la población aparece ordenada, tal y como suele ser habitual, en función del valor de bondad de cada individuo.

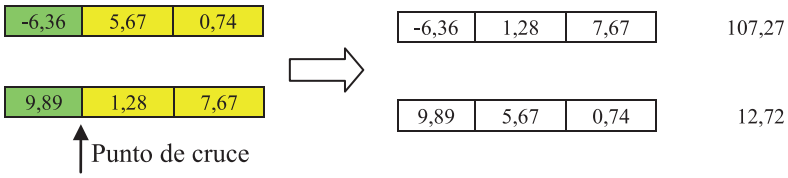
Siguiendo las pautas mostradas en el pseudocódigo de la Figura 1.5 y, puesto que se ha establecido un valor de elitismo de 2, los dos primeros individuos de la población pasarían a formar parte de la población temporal.

Individuo			Fitness Puro	Fitness Normalizado
-2,68	-7,68	6,53	3,57	0,0046
1,36	0,02	1,62	14,64	0,0187

Población temporal. Paso 1

El resto de la población temporal se rellenará como resultado de los operadores de cruce, copia y/o mutación. El siguiente paso consistirá en seleccionar dos individuos para la realización del cruce. El primero de ellos se seleccionará mediante el algoritmo de torneo determinístico (p=2), por lo que previamente deberán seleccionarse al azar dos individuos de la población, supóngase que sean los individuos 3 y 8 de la población. De entre estos, el que presenta mejor fitness es el individuo 3, por lo que será seleccionado como primer padre para el algoritmo de cruce. El segundo de los padres del algoritmo de cruce se seleccionará al azar, por ejemplo el individuo número 7.

Una vez seleccionados los padres se deberá determinar si se procede o no a la aplicación del algoritmo de cruce. Para ello se genera un número al azar en el intervalo [0..1] y, si este es menor que la tasa de cruce P_c, se procede a la realización del cruce. Supóngase que así ocurre en este caso. En este caso el algoritmo de cruce escogido es el cruce en un punto, por lo que deberá escogerse un punto de cruce dentro del genotipo de los individuos. Suponiendo que el punto escogido es el 1, el resultado del cruce sería el siguiente:



Obsérvese como el segundo descendiente originado a través de la operación de cruce posee un mejor fitness que cualquiera de los dos descendientes.

Una vez generada la descendencia es hora de aplicar la operación de mutación. Existen varias alternativas a la hora de escoger el individuo a mutar:

seleccionar al azar uno de los individuos o bien, tal y como aquí se ilustra, mutar uno de los descendientes originados tras el cruce. En este caso, supóngase que se va a aplicar la mutación sobre el primer descendiente. El proceso que determina si la mutación tiene o no lugar es análogo al seguido con la operación de cruce. Así, se genera un valor aleatorio en el intervalo [0..1] y, si este es menor que la tasa de mutación P_m, se procede a la realización de la operación de mutación. Supóngase que en este caso se genera un valor de 0.013 que, al ser menor que P_m=0.025, implicará que el individuo seleccionado vea alterado al azar uno de sus genes. Por ejemplo, suponiendo que se mute el segundo gen, el resultado de dicha mutación podría ser el siguiente:

-6,36	3,65	7,67	128,6

En este caso, al contrario de lo ocurrido con la operación de cruce, el nuevo individuo presenta un peor ajuste que el original. Esta situación no es extraña ni tiene porqué ser perjudicial, puesto que el objetivo primordial de la operación de mutación consiste en introducir nueva información en la población genética.

Una vez aplicados los operadores de cruce y mutación, los descendientes deberán ser insertados en la población temporal, quedando así:

Individuo			Fitness Puro
-2,68	-7,68	6,53	3,57
9,89	5,67	0,74	12,72
1,36	0,02	1,62	14,64
-6,36	3,65	7,67	128,6

Población temporal. Paso 2

El proceso continúa con la selección de nuevos individuos. Supóngase que son los individuos número 9 y 8 los escogidos para la realización del torneo, resultando ganador por lo tanto el individuo 8 por presentar un mejor fitness. El otro individuo participante en el cruce se escogerá al azar. Supóngase que se escoge el individuo número 3. Nótese que este individuo ya había sido escogido previamente, hecho que no lo invalida para una nueva re-elección.

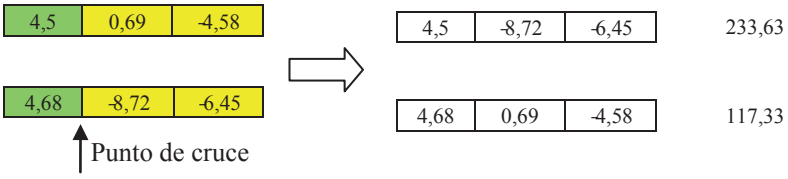
Una vez escogidos los individuos es hora de determinar si la operación de cruce se aplica o no. En este caso, se genera un valor aleatorio de 0.97 que, al ser mayor que P_C, conlleva que la operación de cruce no tenga lugar. En este

caso, los individuos seleccionados pasan, a través del operador genético de copia, directamente a la población temporal.

Individuo			Fitness Puro
-2,68	-7,68	6,53	3,57
9,89	5,67	0,74	12,72
1,36	0,02	1,62	14,64
-6,36	5,67	0,74	28,97
4,5	0,69	-4,58	117,15
-6,36	3,65	7,67	128,6

Población temporal. Paso 3

Nuevamente es necesario seleccionar los individuos para la operación de cruce, en este caso escogiendo los individuos 8 y 9. Al igual que en los casos anteriores será el mejor individuo (el 8) el que finalmente participe en el cruce junto con otro escogido al azar, en este caso el 10. Una vez generado el número al azar (p.e. 0,62) se comprueba si es menor que la tasa de cruce. Al ser así en este caso, se procede a la realización del cruce:



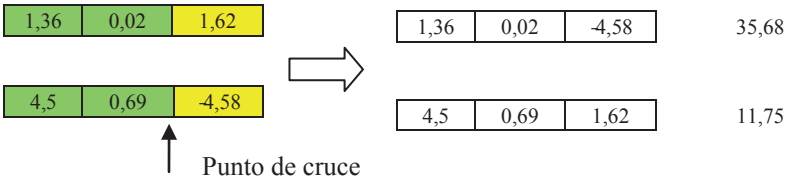
En este caso supóngase que tras la generación del número aleatorio correspondiente y su comprobación con P_m no es necesario aplicar el operador de mutación, con lo que los descendientes generados a partir del operador de cruce se insertan directamente en la población temporal.

Individuo			Fitness Puro
-2,68	-7,68	6,53	3,57
9,89	5,67	0,74	12,72
1,36	0,02	1,62	14,64
-6,36	5,67	0,74	28,97
4,5	0,69	-4,58	117,15
4,68	0,69	-4,58	117,33
-6,36	3,65	7,67	128,6
4,5	-8,72	-6,45	233,63

Población temporal. Paso 4

Una vez más deberán seleccionarse los individuos que formarán parte de la operación de cruce si ésta llega a realizarse. Supóngase que en esta ocasión se seleccionan para la participación en el torneo los individuos 2 y 6 (con lo que el ganador será el 2) y el individuo 8 es seleccionado aleatoriamente.

Para determinar si finalmente se aplica o no la operación de cruce, nuevamente ha de generarse un valor aleatorio en el rango [0..1]. Supóngase que se genera el valor 0,27 que, al ser menor que P_C, determinará la realización del cruce, en este caso seleccionándose al azar como punto de cruce el segundo gen.



Una vez realizado el cruce deberá comprobarse si es o no necesario realizar la operación de mutación. En este caso supóngase que se genera un valor 0,02 que, al ser menor que $P_{m,}$ determinará la necesidad de realizar la mutación. En esta ocasión se mutará de manera aleatoria el tercer gen del primer descendiente generado en la operación de cruce.

4,5	0,69	2,13	3,08

Ambos individuos deberán ser insertados en la población temporal que, llegado este punto, ya se encuentra completa. Por lo tanto, la población genética actual se descarta y se establece como población de la siguiente generación, en este caso la 1, la actual población temporal.

	Individuo			Fitness Puro	Fitness Normalizado
1	4,5	0,69	2,13	3,08	0,0046
2	-2,68	-7,68	6,53	3,57	0,0053
3	4,5	0,69	1,62	11,75	0,0175
4	9,89	5,67	0,74	12,72	0,0189
5	1,36	0,02	1,62	14,64	0,0218
6	-6,36	5,67	0,74	28,97	0,0431
7	4,5	0,69	-4,58	117,15	0,1745
8	4,68	0,69	-4,58	117,33	0,1747
9	-6,36	3,65	7,67	128,6	0,1915
10	4,5	-8,72	-6,45	233,63	0,3480

Población genética y valores de aptitud. Generación 1

Obsérvese cómo tras esta primera iteración el individuo más adaptado obtenido ya mejora al mejor individuo previo. Es decir, la población genética se va adaptando paulatinamente al medio, en este caso, a la resolución del sistema de ecuaciones.

Ahora sería el momento de determinar si la ejecución del algoritmo genético continúa o bien si ha de detenerse. Esta comprobación suele realizarse en base al número de generaciones transcurridas o en base a la aptitud del mejor individuo. También puede hacerse en base a criterios algo más complejos, como el hecho de que la mejor solución proporcionada por el algoritmo genético varíe en un porcentaje menor a uno prefijado durante determinado número de generaciones.

1.8.1.5. Implementación en Matlab

Para la implementación de la solución propuesta mediante Matlab se empleará la versión 2.1 de su toolbox de algoritmos genéticos. En las dos siguientes secciones se recoge el código necesario para la inicialización del algoritmo genético y para la definición de la función de evaluación.

1.8.1.5.1. *Inicialización algoritmo genético*

El fichero de inicialización se encarga de la configuración de los diferentes parámetros del algoritmo genético: tamaño y tipo de población, definición de los algoritmos de selección, cruce y mutación, establecimiento de los criterios de parada, etc.

Además, en este fichero se establecen los coeficientes y términos independientes del sistema de ecuaciones que desea resolverse (variable sistemaEcuaciones). Esta variable se declara global para que sea directamente accesible desde el fichero de evaluación.

```
% ******************************************************
    % Sistema Ecuaciones : Resolución mediante AAGG
% ******************************************************

close all;
clear all;

warning('off');

global sistemaEcuaciones;

sistemaEcuaciones = [3   8    2   25;
                     1  -2    4   12;
                    -5   3   11    4];
```

```
 % ******************************************************
          % Configuración Algoritmo Genético
 % ******************************************************

options = gaoptimset;

% Población Genética
options = gaoptimset(options,'PopulationSize' , 500);
options = gaoptimset(options,'PopulationType' ,
                             'doubleVector');

options = gaoptimset(options,'PopInitRange' ,
                              [-5 -5 -5; 5 5 5]);

options = gaoptimset(options, 'CreationFcn',
                              @gacreationuniform);

% Criterios de parada

options = gaoptimset(options, 'Generations', 1000);
options = gaoptimset(options, 'FitnessLimit', 0.001);

options = gaoptimset(options,'TolFun', 1e-12);
options = gaoptimset(options,'StallTimeLimit', 1000);
options = gaoptimset(options,'StallGenLimit', 1000);


 % ******************************************************
               % Operadores Genéticos
 % ******************************************************

% Elitismo
options = gaoptimset(options, 'EliteCount', 2);

% Operador de selección
options = gaoptimset(options, 'SelectionFcn',
                               @selectiontournament);

% Algoritmo de cruce
options = gaoptimset(options, 'CrossoverFcn',
                               @crossoversinglepoint);

options = gaoptimset(options, 'CrossoverFraction', 0.9);

% Algoritmo de mutación
options = gaoptimset(options, 'MutationFcn',
                               @mutationuniform);
```

```
% ******************************************************
                % Configuracion Salida
% ******************************************************

options = gaoptimset(options, 'Display', 'off');
options = gaoptimset(options, 'PlotInterval', 10);

options = gaoptimset(options,'PlotFcns',[@gaplotbestf]);


% Ejecución algoritmo
[x, fval, reason, output, population, scores]= ...
 ga(@fitness,length(sistemaEcuaciones)-1,options);

disp('');
disp('Mejor individuo: ');
disp(x);
disp('Ajuste: ');
disp(fval);
```

1.8.1.5.2. *DEFINICIÓN DE LA FUNCIÓN DE EVALUACIÓN*

La función evaluación es una traducción directa a código Matlab del pseudocódigo mostrado anteriormente.

Así, el valor de ajuste de un individuo sería el sumatorio de las diferencias en valor absoluto entre cada uno de los términos independientes del sistema y los términos independientes obtenidos al sustituir los valores del genotipo en la ecuación correspondiente.

El bucle interno reemplaza en cada una de las ecuaciones del sistema los valores proporcionados por el individuo para cada incógnita, mientras que el bucle externo acumula las diferencias en valor absoluto entre el valor obtenido tras dicha substitución y el término independiente del sistema de ecuaciones.

```
function [ FitnessValue ] = fitness(individuo)
global sistemaEcuaciones;
[f, c] = size(sistemaEcuaciones);
diferencia = 0;
for i=1:f
```

```
resultado = 0;

for j=1:(c-1)
       resultado = resultado +
              sistemaEcuaciones(i,j)*individuo(j);
end

diferencia = diferencia +
              abs(sistemaEcuaciones(i, c) - resultado);

end

FitnessValue = diferencia;
```

1.8.2. N Reinas

A continuación se muestra un ejemplo práctico resuelto mediante la aplicación de algoritmos genéticos. El problema de las N-Reinas, debido a sus peculiares características, permitirá observar un proceso de codificación y configuración especial de un algoritmo genético para proceder a su resolución. Una vez mostrados los pasos necesarios para abordar la solución del problema, se muestra su implementación mediante el lenguaje de programación MATLAB (Mathworks, 2005)

1.8.2.1. Descripción del problema

A grandes rasgos, el problema de las N Reinas consiste en situar N reinas sobre un hipotético tablero de ajedrez de NxN celdas, de tal manera que ninguna reina pueda atacar la posición del resto de reinas. Es decir, colocar N reinas sobre el tablero NxN sin que ninguna comparta fila, columna o diagonal de dicho tablero.

Una búsqueda secuencial de todas las posibles combinaciones implicaría la búsqueda de $N \cdot (N-1) \cdot \ldots \cdot 1 = N!$ posibilidades. Concretamente, para un tablero tradicional de 8x8 casillas, existirían $8! = 40.320$ posibilidades, de las cuales únicamente 92 serían soluciones válidas. En la Figura 1.12 se muestra una de estas soluciones válidas, con la ubicación de las 8 reinas sobre el tablero de ajedrez.

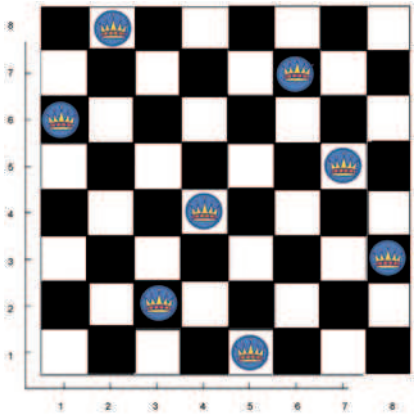


Figura 1.12: Solución válida para el problema 8-Reinas

1.8.2.2. Codificación del problema

Si desea abordarse el problema mediante el empleo de algoritmos genéticos, en primer lugar será necesario codificarlo en los términos adecuados.

Una primera aproximación a la codificación del problema se podría plantear como una matriz de NxN elementos binarios. En este caso un valor 1 significaría que se encuentra una reina en esa posición, y un 0 que esa casilla del tablero está vacía. Se trataría de una codificación claramente ineficiente ya que se está almacenando mucha más información de la necesaria, puesto que el hecho de conocer en que posiciones se colocan las reinas implica directamente el conocimiento acerca de las casillas que se encuentran vacías.

Por lo tanto puede plantearse otra codificación, mucho más eficiente. Para ello, hay que recordar que cada reina ha de estar en una fila y columna diferente. Partiendo de que en cada fila existirá una reina, el problema se puede replantear de tal manera que consista en determinar la columna dentro de la fila correspondiente en la que haya que colocar la reina, de tal manera que ninguna se haga jaque. Por lo tanto, la solución al problema podrá codificarse en forma de una tupla de n valores $S = (X_1, X_2, \ldots, X_n)$ en la que cada X_i representa la columna en la que se coloca la reina de la fila i. Para evitar situar más de una reina en la misma columna, los valores de S serán una permutación de la tupla (1, 2, …, N).

Retomando la solución mostrada en la Figura 1.12, ésta se codificaría según la siguiente tupla S = (5, 3, 8, 4, 7, 1, 6, 2), de tal manera que la reina de

la primera fila se sitúa en la columna 5, la reina de la segunda fila se sitúa en la columna 3, y así sucesivamente.

1.8.2.3. Función de Evaluación

Una vez determinada la codificación del problema, será necesario proporcionar una función de fitness que permita determinar la validez de cada uno de los individuos generados por el algoritmo genético.

Para ello deberá tenerse en cuenta que una solución será válida cuando ninguna reina amenaza al resto. Es decir, cuando ninguna comparta fila, columna o diagonal. En la codificación escogida, cada posición *i* del vector se corresponde con la posición de la reina dentro de la fila *i* del tablero, con lo cual no podrá haber dos reinas compartiendo fila. Por otra parte, el vector de posiciones representa posiciones de columnas, con lo que si en dicho vector no existen valores repetidos, tampoco habría conflictos en cuanto a situar más de una reina en la misma columna. Restaría por determinar si dos reinas comparten diagonal. Ello ocurrirá cuando para algún par de posiciones del vector (X_1, X_2, …, X_n) se cumple que tienen el mismo valor para (fila-columna) o bien para (fila+columna). Es decir, expresado de manera más formal, existirá un conflicto cuando se cumpla la siguiente condición:

$$\exists\, i,j \left\{(i + X_i) = \left(j + X_j\right) \text{ ó } (i - X_i) = (j - X_j)\right\} \; con \;\; 1 \leq i,j \leq N$$

Esta expresión puede simplificarse de la siguiente manera con el objetivo de determinar si existe o no colisión en diagonal para dos reinas situadas en las columnas i y j:

$$\exists\, i,j \left\{\|(i - j)\| = \left\|X_i - X_j\right\|\right\} \; con \;\; 1 \leq i,j \leq N$$

Tal y como se ha indicado, la propia codificación del problema elimina la existencia de problemas como que una reina sea atacada por otra de la misma fila.

Utilizando convenientemente las operaciones de cruce y mutación es posible también eliminar la probabilidad de que en una misma columna se ubique más de una reina. Ambas operaciones deben tener en cuenta que el cromosoma es realmente una permutación de los valores 1 a N, y que por lo tanto, tras la aplicación de las operaciones genéticas, el cromosoma debe mantener dicha propiedad. En cuanto a la mutación, simplemente con aplicar la mutación por intercambio de 2 valores comentada anteriormente se mantendría el cromosoma correctamente formado. El operador de cruce es ligeramente

distinto a los explicados anteriormente. La idea es construir una operación de cruce que mantenga el orden relativo en el que aparecen los valores dentro del vector de columnas. Una aproximación válida sería la siguiente, basada en el cruce en un punto:

1. Escoger un punto aleatorio dentro del primer padre
2. Copiar la primera parte del cromosoma del progenitor al primer descendiente
3. Copiar en el primer descendiente los valores no incluidos en esta primera parte, manteniendo el orden en el que aparecen en el segundo progenitor
4. Proceder de igual manera, pero intercambiado los papeles de los progenitores, para generar el segundo descendiente.

Aún con la codificación escogida y los operadores genéticos definidos, lo que no se elimina es la posibilidad de conflictos con respecto a que una reina sea atacada por otra que esté en su misma diagonal. La función de fitness deberá ser capaz de determinar lo cerca que está de una solución válida cada una de las soluciones proporcionadas por el algoritmo genético. Por lo tanto, una opción será considerar el valor de ajuste de un individuo como el número de conflictos que se presentan en la diagonal. Así, cuantos más conflictos haya, más errónea será la solución, mientras que si el número de conflictos es 0, el individuo genético representará una solución válida.

1.8.2.4. Implementación en Matlab

Para la implementación de la solución propuesta mediante Matlab se empleará la versión 2.1 de su toolbox de algoritmos genéticos. En las siguientes secciones se recogen todos los pasos necesarios. Así, se incluye la implementación de la función de creación de la población genética (para forzar a que los individuos sean una permutación de valores), la implementación de los algoritmos de cruce y mutación que tengan en cuenta esta peculiaridad de los individuos, y, por último, la función de evaluación, que calculará el número de colisiones que se producen sobre la diagonal para determinar la bondad de cada solución.

1.8.2.4.1. *Creación de la población genética*

La función de creación inicializa la población mediante individuos consistentes en una permutación de valores, manteniendo los individuos proporcionados por el usuario (mediante el parámetro options.InitialPopulation) en caso de que estos existan.

```
function Population = ...
       PermCreation(GenomeLength, FitnessFcn, options)

       totalPopulation = options.PopulationSize;
       initPopulation = size(options.InitialPopulation,
1);

       individualsToCreate=totalPopulation-initPopula-
tion;

       Population=zeros(individualsToCreate, Genome-
Length);

if (initPopProvided > 0)
       Population(1:initPopulation,:)=...
       options.InitialPopulation;

end

for individual=1:individualsToCreate
       Population(initPopulation+individual,:)= ...
              randperm(GenomeLength);
end
```

1.8.2.4.2. *OPERADOR DE MUTACIÓN*

El operador de mutación simplemente realiza el intercambio entre dos genes escogidos al azar dentro del cromosoma de los individuos genéticos seleccionados para la mutación.

El vector *parents* empleado en la implementación del operador recoge los índices de los individuos seleccionados para la mutación. El propio funcionamiento determina que la aplicación del operador de mutación no se hace de manera individual, sino en conjunto a todos aquellos individuos seleccionados para la mutación genética.

```
function mutationChildren = PermMutation(parents, ...
   options, GenomeLength, FitnessFcn, state, ...
   thisScore, thisPopulation)

mutationChildren = zeros(length(parents),GenomeLength);

   for i=1:length(parents)

        child = thisPopulation(parents(i),:);
        position1 = ceil(GenomeLength*rand());
```

```
        equalPosition = true;
        while (equalPosition)
            position2=ceil(GenomeLength*rand());
            if (position1 ~= position2)
               equalPosition = false;
            end
        end

        aux = child(position1);
        child(position1) = child(position2);
        child(position2) = aux;

        mutationChildren(i,:) = child;

    end
```

1.8.2.4.3. OPERADOR DE CRUCE

El operador de cruce recoge el comportamiento explicado con anterioridad (copia de un segmento inicial de los padres y copia de los valores no incluidos en dicho segmento conservando el orden del segundo progenitor).

En este caso, el funcionamiento del propio toolbox de Matlab fuerza a que se genere un único descendiente por cada dos progenitores. Asimismo, los cruces se realizan también en bloque a partir de los índices de los progenitores (parents) previamente seleccionados.

```
function xoverKids = PermCrossover(parents, options,...
GenomeLength, FitnessFcn, unused, thisPopulation)

% Se produce un descendiente por cada dos padres
nKids = length(parents)/2;

% Reserva de espacio para la descendencia
xoverKids = zeros(nKids,GenomeLength);

pIndex = 1;

for i=1:nKids

% Obtencion de los padres
parent1 = thisPopulation(parents(pIndex), :);
pIndex = pIndex + 1;
parent2 = thisPopulation(parents(pIndex), :);
pIndex = pIndex + 1;
```

```
% Paso 1: obtener punto de cruce
   crossoverPoint = ceil((GenomeLength-1)*rand());

% Paso 2: copiar primera parte del primer padre
   % a la descendencia
   xoverKids(i, 1:crossoverPoint) = ...
   parent1(1:crossoverPoint);

% Paso 3: copiar en descendencia valores no
   % incluidos en el primer segmento, manteniendo
   % el orden del segundo progenitor. Estos nuevos
   % valores se insertan a partir de la posición
   % de cruce.

initialIndex = crossoverPoint + 1;

for j=1:GenomeLength
   if (isempty(...
          find(parent2(j)==...
                 xoverKids(i,1:crossoverPoint))))
                 % gen del padre no pasado previamente a
                 % la descendencia
                 xoverKids(i, initialIndex) = parent2(j);
   initialIndex = initialIndex + 1;
              end
          end
   end
end
```

1.8.2.4.4. FUNCIÓN DE EVALUACIÓN

Tal y como se comentó anteriormente, la bondad de un individuo se determina en base al número de colisiones, o posibles ataques entre reinas, que se producen en la diagonal del tablero. Para ello es necesario comprobar cada reina con cada una de las restantes para verificar si pueden atacarse entre sí.

```
function [ FitnessValue ] = fitness(individuo)

   colisiones = 0;

   for i=1:(length(individuo)-1)

          for j=(i+1):length(individuo)

          if (abs(i-j)==abs(individuo(i)-individuo(j)))
                 colisiones = colisiones + 1;
          end
   end
end

FitnessValue = colisiones;
```

1.8.2.4.5. CONFIGURACIÓN DEL ALGORITMO GENÉTICO

Una vez definidos los diferentes operadores, sólo resta configurar el algoritmo genético para que haga uso de ellos. Además, es necesario configurar las dimensiones del tablero (numeroReinas), el tamaño de la población genética (PopulationSize), el número máximo de generaciones (Generations), etc.

```
% ****************************************************
% N-Reinas : Resolución mediante Algoritmos Genéticos
% ****************************************************

%            Configuración Algoritmo Genético
% ****************************************************
options = gaoptimset;
% Población Genética
options = gaoptimset(options,'PopulationSize' , 25);
options = gaoptimset(options,'CreationFcn', ...
                             @PermCreation);

% Criterios de parada
options = gaoptimset(options, 'Generations', 100);
options = gaoptimset(options, 'FitnessLimit', 0);
options = gaoptimset(options,'TolFun', 1e-12);
options = gaoptimset(options,'StallTimeLimit', 100);
options = gaoptimset(options,'StallGenLimit', 100);


%                 Operadores Genéticos
% ****************************************************

% Elitismo
options = gaoptimset(options, 'EliteCount', 1);


% Ajuste del operador de selección
options = gaoptimset(options, 'SelectionFcn',...
                              @selectionroulette);


% Ajuste del algoritmo de cruce:
options = gaoptimset(options, 'CrossoverFcn', ...
                               @PermCrossover);

options = gaoptimset(options, 'CrossoverFraction', 0.8);

% Ajuste del algoritmo de mutación:
options = gaoptimset(options, 'MutationFcn', ...
                             @PermMutation);
```

```
                % Configuración de Salida
% ******************************************************

options = gaoptimset(options, 'Display', 'diagnose');
options = gaoptimset(options, 'PlotInterval', 1);

options = gaoptimset(options,'PlotFcns', ...
          [@gaplotbestindiv, @gaplotbestf]);

% Ejecución algoritmo

[x, fval, reason, output, population, scores]= ...
   ga(@fitness,numeroReinas,options);

disp('');
disp('Mejor individuo: ');
disp(x);
disp('Número de colisiones: ');
disp(fval);
```

1.8.8.4.6. *Ejemplo de Ejecución*

Una vez mostrado el código necesario para la resolución del problema de las N-Reinas, se muestra una ejecución de ejemplo. Ésta corresponde a la resolución del problema para un tablero de 8x8 casillas.

Empleando un tamaño de población típico (del orden de 100 individuos) la resolución es prácticamente instantánea, del orden de 3 o 4 iteraciones. Sin embargo, para que pueda mostrarse la evolución del error de la Figura 1.13 se ha reducido este número hasta los 25 individuos. En esta figura, puede observarse tanto la evolución del error del mejor individuo y la media de la población, como la mejor solución finalmente encontrada (5, 8, 4, 1, 7, 2, 6, 3).

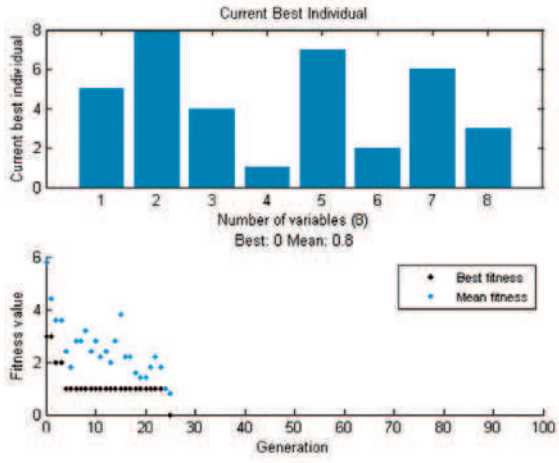


Figura 1.13: Problema de las 8-Reinas: evolución del fitness

Referencias

Bäck, T. (1996). *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*: Oxford University Press, USA.

Beasley, D., Bull, D. R., Martin, R. R. (1993). An overview of genetic algorithms: Part 1, fundamentals. University Computing, 15(2):58-69.

Blickle, T., Thiele, L. (1995). A comparison of selection schemes used in genetic algorithms. Technical Report 11, Computer Engineering and Comunnication Network Lab (TIK), Gloriastrasse 35, 8092 Zurich, Switzerland.

Darwin, C. (1859). On the Origin of Species by Means of Natural Selection. John Murray, London.

Darwin, C. (2007). Descent of Man: Nuvision Publications.

De Jong, K. A. (1975). An analysis of the behavior of a class of genetic adaptive systems. PhD thesis, University of Michigan.

De Jong, K. A., & Spears, W. M. (1992). A formal analysis of the role of multi-point crossover in genetic algorithms. Annals of Mathematics and Artificial Intelligence, 5(1), 1-26.

Fogel, D. B. (2000). What is evolutionary computation? Spectrum, IEEE, 37(2), 26, 28-32.

Fogel, D. B. (2006), Evolutionary Computation: Toward a New Philosophy of Machine Intelligence, IEEE Press, Piscataway, NJ. Third Edition

Fogel, L. J., Owens, A. J., and Walsh, M. J. (1966). Artificial Intelligence through Simulated Evolution. John Wiley & Sons, New York.

Goldberg, D. E. (1989). Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Goldberg, D. E. (2002), The Design of Innovation: Lessons from and for Competent Genetic Algorithms, Addison-Wesley, Reading, MA.

Grefenstette, J. J. (1992). Genetic algorithms for changing environments. Paper presented at the Parallel Problem Solving from Nature, Bruselas, Bélgica.

Holland, J. H. (1975). Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor. Republished by the MIT press, 1992.

Koza, J. R. (1992). Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems). The MIT Press.

Mathworks. (2005). Genetic Algorithm and Direct Search Toolbox User's Guide version 2. Disponible online en: http://www.mathworks.com/access/helpdesk/help/pdf_doc/gads/gads_tb.pdf

Michalewicz, Z. (1999), Genetic Algorithms + Data Structures = Evolution Programs, Springer-Verlag.

Michalewicz, Z., & Fogel, D. B. (2000). How to Solve It: Modern Heuristics: Springer Verlag.

Rechenberg, I. (1973). Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution. Frommann-Holzboog, Stuttgart.

Tomassini, M. (1995). A survey of genetic algorithms. Annual Reviews of Computational Physics, III:87-118.

Whitley, D. (1994). A genetic algorithm tutorial. Statistics and Computing 4, 65–85.

This page intentionally left blank

Capítulo 2
Programación Genética

2.1. Introducción

La Programación Genética surge como una evolución de los algoritmos genéticos tradicionales, manteniendo el mismo principio de selección natural. Lo que ahora se pretende es resolver los problemas mediante la inducción de programas y algoritmos que los resuelvan. Es en esta posibilidad donde reside toda la potencialidad de la Programación Genética, puesto que permite desarrollar de forma automatizada programas, entendiéndose éstos de una forma amplia, es decir, como algoritmos en los que, a partir de una serie de entradas, se genera una serie de salidas. De esta manera, por ejemplo, una ecuación matemática podría ser inducida mediante el uso de Programación Genética.

La base biológica de la Programación Genética es exactamente la misma que la de los Algoritmos Genéticos. Por esta razón, el funcionamiento es similar. La diferencia entre una técnica y otra consiste en la forma de codificación de problemas, lo que permite su utilización en una serie de entornos donde anteriormente los Algoritmos Genéticos no podían ser aplicados.

2.2. Orígenes

A pesar de que oficialmente se data la creación de la Programación Genética en 1992 tras la aparición del libro titulado "Genetic Programming" por John Koza (Koza, 1992), donde se acuñó el término y se sentaron las bases formales de esta técnica, existen trabajos previos que, sin usar explícitamente el nombre de Programación Genética, pueden ser considerados como precursores de la materia.

Ya en el primer congreso sobre Algoritmos Genéticos, celebrado en la Universidad de Carnegie Mellon en 1985, se presentó un artículo titulado "A Representation for the Adaptive Generation of Simple Sequential Programs" (Cramer, 1985), en el que se plantea un sistema adaptativo para la generación de pequeños programas secuenciales. Con este objetivo, en dicho trabajo se hace uso de dos lenguajes: JB, que representa los programas en forma de cadenas de números, y TB. Este último es una versión evolucionada del anterior, pero con estructura de árbol para representar programas, que es la estructura que utiliza la Programación Genética como forma de codificación. El objetivo principal del artículo es conseguir una forma de representar programas que, por un lado, permita la aplicación de los operadores genéticos clásicos (mutación, cruce, etc.) con la restricción de que estos programas generados automáticamente estén "bien formados". En dicho artículo no es importante que todos los programas que se puedan obtener sean útiles, puesto que de ello se encargaran los criterios de selección; sólo importa que estén dentro del espacio de programas sintácticamente correctos.

Este trabajo tiene una gran importancia puesto que se constata la importancia que tiene la representación de los programas para su manipulación. Los problemas que plantea el lenguaje JB pueden ser eliminados si se usa TB y su estructuración en forma de árboles, que es la forma de codificación de la Programación Genética.

Otro trabajo, titulado "Using the Genetic Algorithm to Generate Lisp Source Code to solve the Prisoner's Dilemma" (Fujiki & Dickinson, 1987) y presentado en el Segundo congreso sobre Algoritmos Genéticos, celebrado en el *Massachussetts Institute of Technology* en 1987, trata sobre la resolución de un problema clásico: el dilema del prisionero. En este problema, dos prisioneros son interrogados por separado sin posibilidad de comunicación entre ellos. Cada uno debe decidir si delatar o no al otro y, en función de la declaración de cada uno, ambos obtienen más o menos puntos. Esta situación se repite un número determinado de interrogatorios, en los cuales cada prisionero únicamente sabe cuál ha sido el resultado de las anteriores (es decir, su decisión y la del otro prisionero y, por tanto, el número de puntos de cada uno). Al finalizar, el que obtenga más puntos gana. El objetivo es desarrollar una estrategia que, al ser aplicada de forma iterativa en cada interrogatorio, permita obtener el mayor número de puntos.

El interés del artículo se centra en el apartado dedicado a la representación del conocimiento y de los programas o estrategias generados. Concretamente, se utilizó un conjunto restringido de expresiones LISP, que demostraron ser de

gran utilidad y funcionalidad. Más adelante, en 1992, el trabajo que sentaría las bases de la Programación Genética, titulado "Genetic Programming" se basaría casi exclusivamente en el lenguaje LISP para realizar la generación automática de programas (Koza, 1992).

Es precisamente este libro el que se suele tomar como referente de la Programación Genética, donde se acuñó el término y donde se sentaron las bases formales de una técnica que estaba empezando a surgir, a pesar de que el mismo autor, John Koza, dos años antes había publicado un informe técnico en la Universidad de Stanford citando explícitamente el término *Programación Genética*, y describiendo el paradigma (Koza, 1990). Sin embargo, no fue hasta la aparición del libro en 1992 cuando se dotó a esta técnica de la formalidad necesaria.

2.3. Codificación de problemas

La mayor diferencia entre los algoritmos genéticos y la programación genética es la forma de codificación de la solución al problema. En programación genética la codificación se realiza en forma de árbol, de forma similar a como los compiladores interpretan los programas según una gramática prefijada.

2.3.1. Elementos del árbol

Al haber una representación de árbol, existirán dos tipos de nodos:

- Terminales, u hojas del árbol. Son aquellos que no tienen hijos. Normalmente se asocian con valores constantes o variables.
- Funciones. Son aquellos que tienen uno o más hijos. Generalmente se asocian con operadores del algoritmo que se quiere desarrollar.

En la figura 2.1 se puede ver un ejemplo de árbol, que representa el programa f(x) = 2*(3+x). Tiene como funciones los correspondientes al producto y a la suma, y como nodos terminales los correspondientes a los valores 2 y 3 y la variable x.

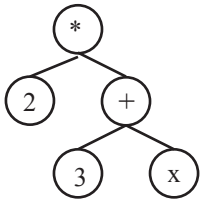


Figura 2.1: Árbol para la expresión 2*(3+x)

Una parte fundamental del funcionamiento de la Programación Genética es la especificación del conjunto de funciones y terminales antes del inicio del proceso evolutivo. Con los nodos que se le especifique el algoritmo construirá los árboles. Por tanto, es necesario un mínimo proceso de análisis del problema para configurar el algoritmo, puesto que hay que decirle qué operadores puede utilizar. Como regla general, es conveniente ajustar el número de operadores sólo a los necesarios, puesto que la adición de elementos que no sean necesarios no provocará que no se encuentre la solución, pero sí que el algoritmo tarde más en encontrarla.

A la hora de especificar los conjuntos de elementos terminales y funciones, es necesario que estos conjuntos posean dos requisitos, que son *cerradura* y *suficiencia* (Koza, 1992). El requisito de suficiencia dice que la solución al problema debe poder ser especificada con el conjunto de operadores especificados. El requisito de cerradura dice que cualquier árbol que se construya con estos operadores debe ser correcto.

Dado que el proceso de construcción de árboles es un proceso estocástico o basado en el azar, muchos de los árboles construidos no serán correctos, no por no seguir las reglas de la gramática, sino por la aplicación de operadores (funciones) a elementos que no están en su dominio. Por esta razón no se aplican estos operadores directamente, sino una modificación de los mismos en la que se amplía su dominio de aplicación. El ejemplo más claro es el operador de división, cuyo dominio es el conjunto de números reales excepto el valor cero. Ampliando su dominio, se define un nuevo operador (%), el cual por norma general asigna el valor unidad al resultado de dividir cualquier valor entre 0:

$$\%\,(a,b) \begin{cases} 1 & \text{si } b = 0 \\ a/b & \text{si } b \neq 0 \end{cases}$$

A esta nueva operación se denomina operación de división protegida, y en general, cuando se crea una nueva operación que extiende el dominio de otra, se denomina operación protegida.

2.3.2. Restricciones

El algoritmo de Programación Genética clásico establece muy pocas restricciones a los árboles que se generen. En general, la condición de cerradura

implica que cualquier árbol construido con los operadores especificados (y protegidos) debe de ser correcto, lo cual quiere decir que la mayor restricción se impone en el momento de la elección del conjunto de terminales y funciones.

Sin embargo, una restricción habitual es la de altura máxima de los árboles. Esta restricción evita la creación de árboles demasiado grandes, y fuerza una búsqueda en soluciones cuyo tamaño se acota de antemano. Con esta restricción se evita que los árboles posean mucho código redundante y el crecimiento excesivo de los árboles, en un fenómeno llamado *bloat* (Soule & Foster, 1997) (Soule, 1998). Este efecto se provoca por el crecimiento desmesurado de los árboles, puesto que una gran parte de estos árboles es código inútil, es decir, que no tiene influencia en el resultado final. Este tipo de código surge espontáneamente como resultado del proceso evolutivo, como una forma que tienen los árboles de protegerse ante posibles modificaciones originadas por los operadores de cruce y mutación. Por lo tanto, es necesario poner un límite al tamaño que pueden alcanzar los árboles, lo cual se suele realizar imponiendo una restricción de altura máxima.

Además, a pesar de que no es habitual utilizarla, es posible especificar reglas de tipado con el objetivo de establecer reglas sintácticas en la creación de árboles (Montana, 1995). Para ello, se establece el tipo de cada nodo (terminales y funciones), y para las funciones el tipo que debe tener cada hijo. De esta forma se especifica la estructura que deben seguir los árboles.

Al especificar el tipo de cada nodo, se está especificando una gramática que va a ser la que siga el algoritmo para la construcción de árboles; esta gramática permitirá que los árboles tengan la estructura deseada.

La inmensa mayoría de las aplicaciones de la Programación Genética se basan en el desarrollo de expresiones matemáticas. Un ejemplo puede verse en la Figura 2.1. En este caso, no es necesario especificar ninguna regla de tipado, puesto que todas las operaciones se realizan sobre valores reales. En general, a pesar de su gran potencialidad, el sistema de tipado no es muy utilizado en las aplicaciones de Programación Genética.

2.4. Algoritmo principal

El funcionamiento es similar al de los algoritmos genéticos: se basa en la generación de sucesivas generaciones a partir de las anteriores. Este algoritmo se puede ver en la Figura 2.2 (Koza, 1992).

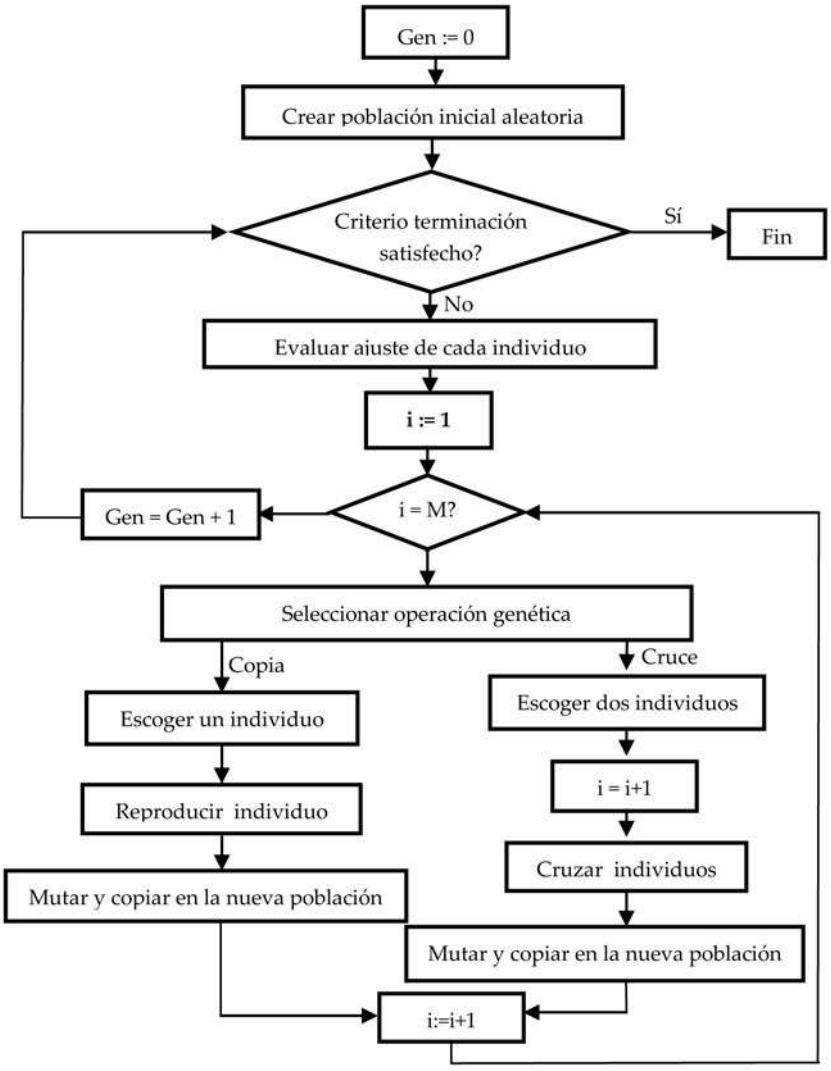


Figura 2.2: Diagrama de flujo de programación genética.

Tras la creación inicial de árboles (generalmente serán aleatorios), se construyen sucesivas generaciones a partir de copias, cruces y mutaciones de los individuos de cada generación anterior.

2.5. Generación inicial de árboles

El primer paso en el funcionamiento del algoritmo es la generación de la población inicial. En la creación de la generación 0, cada árbol se creará de forma más o menos aleatoria, dependiendo del algoritmo, teniendo en cuenta las restricciones que existen en los árboles. Dado que los árboles son aleatorios, los individuos de esta población en general representan soluciones malas al problema.

Para la creación de un árbol existe una gran variedad de algoritmos, pero son tres los más utilizados (Koza, 1992): parcial, completo e intermedio.

El algoritmo de creación parcial genera árboles cuya altura máxima no supera la especificada. El algoritmo es el siguiente, dada la altura máxima y los conjuntos de elementos terminales (T) y funciones (F):

```
GeneraÁrbol (altura, F, T)
begin
      if altura=1
      then
          Asignar como raíz del árbol como un elemento aleatorio de T
      else
          Asignar como raíz del árbol como un elemento aleatorio de F ∪ T
          Para cada hijo de la raíz,
          Asignar a la raíz como hijo el subárbol generado con GeneraÁrbol (altura-1, F, T)
end
```

En este algoritmo, cada hoja tendrá como máximo la profundidad especificada.

El algoritmo completo genera árboles cuyas hojas están todas a un determinado nivel, pues genera árboles completos. Este algoritmo es muy similar al anterior:

```
GeneraÁrbol (altura, F, T)
begin
      if altura=1
      then
          Asignar como raíz del árbol como un elemento aleatorio de T
      else
          Asignar como raíz del árbol como un elemento aleatorio de F
      Para cada hijo de la raíz,
      Asignar a la raíz como hijo el subárbol generado con GeneraÁrbol (altura-1, F, T)
end
```

El algoritmo de creación intermedio es una mezcla de los dos anteriores, creado para que exista mayor variedad en la población inicial, y con ello mayor diversidad genética. Este algoritmo se basa en ejecutar los anteriores alternándolos y tomando distintas alturas para crear todos los elementos de la población. El algoritmo es el siguiente: dado un tamaño de población M y una altura máxima A:

```
for i:=2 to A do begin
        Generar M/(2*(A-1)) árboles de altura i con el método parcial
        Generar M/(2*(A-1)) árboles de altura i con el método completo
end
```

Este método genera un porcentaje de 100/(A-1) % árboles nuevos de altura variando entre 2 y A, de forma alternativa completos y parciales.

En general se evita que se generen árboles de altura 1, es decir, árboles que contengan sólo un elemento terminal, y en la práctica se modifican los algoritmos para que se evite esta posibilidad.

Estos algoritmos se basan fuertemente en el azar, y la única intervención del usuario está en la introducción de los elementos terminales y funciones. Sin embargo, existen muchos más algoritmos en los que la creación de árboles no es tan aleatoria. Por ejemplo, en (Luke, 2000) se asigna una probabilidad de aparición a cada nodo y de esta forma se reduce el carácter aleatorio de la creación y se orientan los árboles a que contengan más nodos de una clase que otra. Los algoritmos serán muy similares, con la salvedad de que la elección de los elementos seguirá siendo aleatoria pero estará ponderada por esa probabilidad asignada.

2.6. Operadores genéticos

Como se ha mostrado, el funcionamiento de la Programación Genética es similar al de los Algoritmos Genéticos: tras la creación de una población inicial aleatoria se desencadena un proceso evolutivo en el que, a partir de combinaciones de distintos individuos se van formando nuevos que darán lugar a una nueva generación. Estos individuos son seleccionados de una población de forma probabilística dando más posibilidades a los más aptos para ser seleccionados, o de forma determinística, siendo los mejores seleccionados un mayor número de veces. Por lo tanto, y dado que el funcionamiento global es el mismo para ambos paradigmas, los operadores genéticos también siguen

la misma filosofía. De hecho, algunos de estos operadores son exactamente iguales a los de los algoritmos genéticos, como es el caso del operador de selección, descrito en la sección 1.6.1, y que, por esa razón, no se describe nuevamente en ésta.

Sin embargo, dada la diferencia en la forma de codificación, es necesario modificar ciertos otros operadores para adaptarlos a la forma de codificación de la Programación Genética, es decir, en forma de árbol.

2.6.1. Cruce

El principal operador es el de cruce. En él, dos individuos de la población se combinan para crear otros dos individuos nuevos.

Después de seleccionar a dos individuos como padres, se selecciona un nodo al azar en el primero y otro en el segundo de forma que su intercambio no viole la restricción de altura máxima. El cruce entre los dos padres se efectúa mediante el intercambio de los subárboles seleccionados en ambos padres. A continuación se ilustra un ejemplo de cruce entre dos árboles.

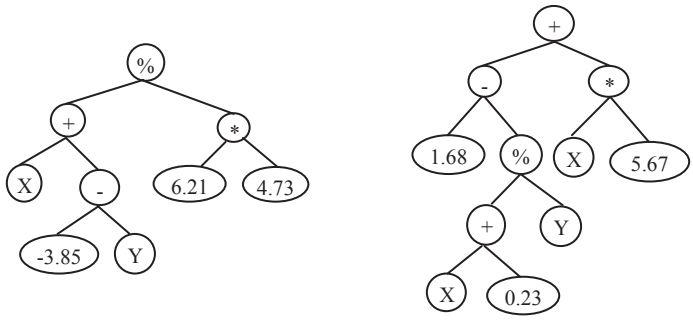


Figura 2.3: Árboles seleccionados para cruce

Se toman dos árboles de la generación anterior. En este ejemplo, los árboles tendrán como altura máxima 5 (Figura 2.3).

Se selecciona un nodo al azar del primer árbol. En este caso el nodo "+". En el segundo árbol se descartan aquellos nodos que llevarían a violar la restricción de altura tras el intercambio. Se descarta la raíz porque llevaría a un árbol demasiado alto en el primer padre y los nodos de altura 4 y 5 porque llevarían a un árbol demasiado alto en el segundo padre. En la Figura 2.4 aparecen destacados los nodos que no han sido descartados por llevar a violar la restricción de altura máxima.

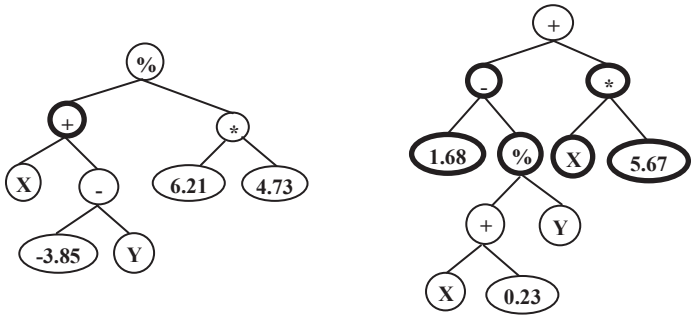


Figura 2.4: Sub-Árboles seleccionados para cruce

En el segundo árbol se selecciona al azar un nodo de los restantes, en este caso el nodo "%", y se intercambian los nodos (Figura 2.5 y Figura 2.6).

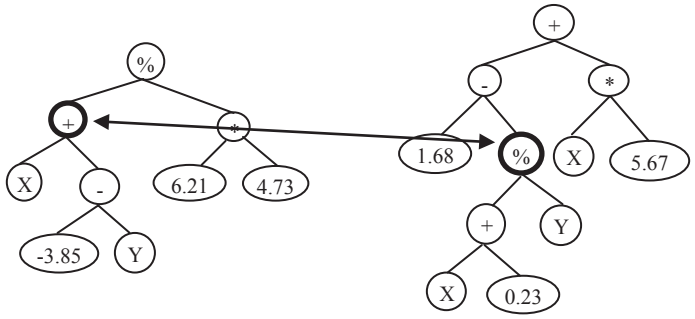


Figura 2.5: Nodos seleccionados para cruce

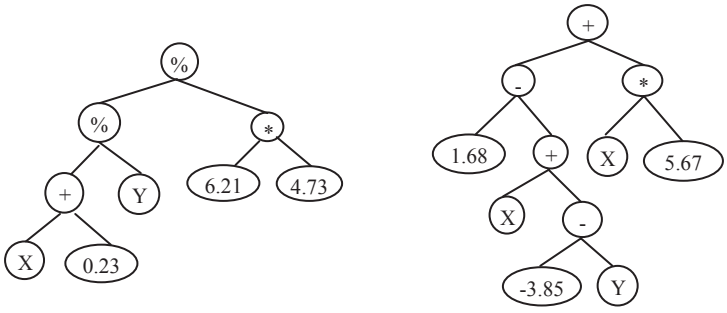


Figura 2.6: Resultado del cruce

Finalmente, se introducen los árboles nuevos en la nueva generación.

La operación de cruce es sexual en el sentido de que se necesitan dos individuos para generar individuos nuevos.

Este operador es el auténtico motor del funcionamiento de la Programación Genética y provoca la combinación de resolución de subproblemas (cada subárbol puede interpretarse como una forma de resolver un subproblema) para la resolución del problema principal.

Se ha observado (Soule, 1998) que la mayoría de los cruces provocan la generación de individuos peores, así como la aparición de mucho código redundante dentro de los árboles. Este código redundante previene los posibles efectos nocivos de otros operadores más destructivos como el de mutación, pero provoca un crecimiento exagerado de los árboles en poco tiempo. Por ello, una solución es utilizar *cruces no destructivos*: los árboles generados por la operación de cruce son insertados en la nueva generación si son mejores que sus padres. En caso contrario, se insertan en la nueva generación copias de los padres.

Una ventaja de este tipo de cruces sobre los cruces de los algoritmos genéticos tradicionales es que al cruzar dos padres iguales, los hijos en general son distintos a los padres (y distintos entre ellos). Esto no ocurría en el cruce en los algoritmos genéticos, en los, que en este caso, cuando se cruzaban padres idénticos, los hijos eran iguales a los padres.

Por ejemplo, si se cruzan los nodos señalados de los árboles iguales presentes en la Figura 2.7, se obtienen los árboles mostrados en la Figura 2.8.

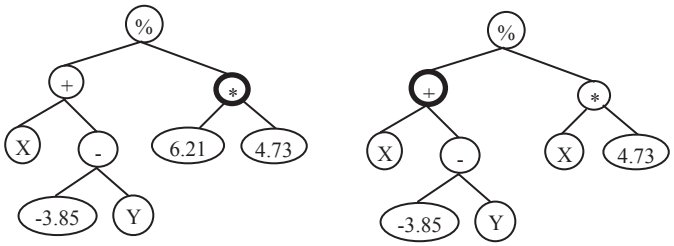


Figura 2.7: Árboles seleccionados para cruce

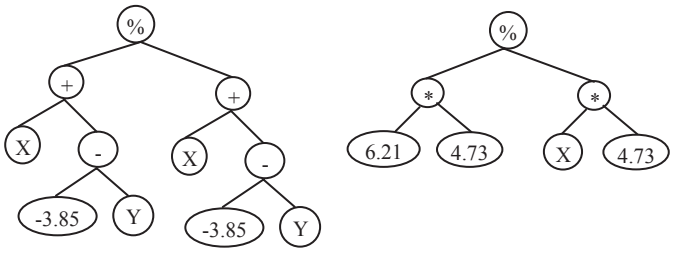


Figura 2.8: Resultado del cruce entre dos árboles iguales

Al realizar un cruce, el nodo escogido en ambos árboles no se suele tomar al azar: generalmente se asigna una probabilidad de que el nodo tomado sea no terminal. Esta probabilidad suele ser alta (sobre 0.9), puesto que en un árbol la mayoría de los nodos son terminales, y si no se toma esta probabilidad la mayoría de los cruces no son más que permutaciones de elementos terminales entre distintos árboles (Angeline, 1996a).

El operador de cruce aquí descrito se realiza de forma igual para todos los individuos. Sin embargo, existen variantes adaptativas de estos operadores (Angeline, 1996b) en las que el propio algoritmo se modifica, así como numerosas variantes que tienen como base este algoritmo (Aguirre, Tanaka & Sugimura, 1999) (Pereira, Machado, Costa & Cardoso, 1999).

2.6.2. Mutación

El operador de mutación provoca la variación de un árbol de la población. Este operador suele usarse con probabilidad muy baja (menos que 0.1) antes de introducir un individuo en la nueva generación.

Existen dos tipos principales de mutación: mutación en la que se varía un solo nodo y mutación en la que se varía una rama entera del árbol.

En el primer caso, conocida por mutación puntual, la mutación actúa de la siguiente manera:

1. Se escoge un nodo al azar del árbol.
2. Se escoge al azar un nodo del conjunto de terminales o funciones, del mismo tipo que el seleccionado, con el mismo número de hijos y de forma que sus hijos sean del mismo tipo.
3. Se intercambia el nodo antiguo del árbol por el nuevo, manteniendo los mismos hijos que el antiguo.

Dado que cada rama del árbol representa una solución a un subproblema y el nodo función que las une representa la forma de combinar esas soluciones, si se realiza este tipo de mutación sobre un elemento no terminal se estará provocando que las soluciones se combinen de distinta forma. Este tipo de mutación apenas se usa. Un ejemplo es el mostrado en la Figura 2.9.

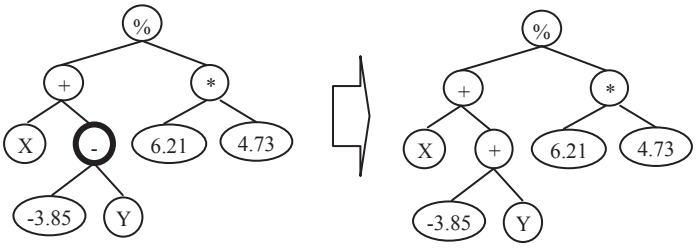


Figura 2.9: Ejemplo de mutación puntual

En este ejemplo, se ha mutado el árbol cambiando el nodo destacado por otro del mismo número de hijos.

En el segundo tipo de mutación, se efectúan cambios mayores en el árbol. La operación es la siguiente: se escoge un nodo al azar del árbol, se elimina todo el subárbol que cuelga de ese nodo, se crea un nuevo subárbol del tipo y altura adecuados y se pone en su lugar. Al cambiar una rama entera, lo que ahora se cambia del árbol es la forma de resolver el subproblema. La Figura 2.10 muestra un ejemplo de mutación, sobre un árbol con altura máxima 5.

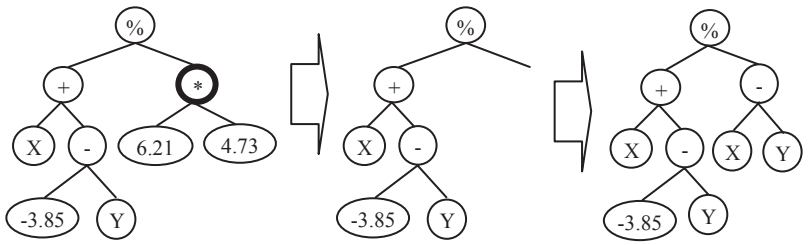


Figura 2.10: Ejemplo de mutación de subárbol

En este ejemplo se escoge un nodo al azar, en este caso se cogió la función de producto de la Figura 2.10. Posteriormente, se elimina ese nodo y se crea un subárbol nuevo. Para no violar la restricción de altura, el subárbol nuevo deberá tener una altura máxima de 4. Finalmente, se coloca el subárbol en el hueco dejado por el nodo eliminado.

El operador de mutación provoca en ese individuo un salto en el espacio de estados, comenzando una búsqueda distinta en otra zona. La mayoría de las mutaciones son destructivas, es decir, el individuo empeora, y por eso se utilizan con una probabilidad muy baja, para conseguir variedad genética. Existen estudios sobre la evolución sin el uso de cruces, en los que la mutación juega un papel fundamental (Chellapilla, 1997), en los que se utilizan distintos tipos de mutaciones, pero los resultados siguen siendo peores que utilizando cruces.

2.7. Evaluación

Al igual que ocurría en los Algoritmos Genéticos, la cuantificación de la bondad de un determinado individuo se realiza por medio del ajuste de ese individuo. Este valor representa lo bien que el árbol soluciona el problema

actual, y, para valorar esta medida, se utilizan los mismos tipos que en el caso de los Algoritmos Genéticos, descritos en la sección 1.7.

Sin embargo, es muy común modificar el valor de ajuste para penalizar árboles excesivamente grandes. Como ya se ha dicho en la sección 2.3.2, un problema bastante común en la Programación Genética es el denominado *bloat*, consistente en el crecimiento excesivo de los árboles. Una forma de controlarlo, como se ha comentado, es mediante el establecimiento de una restricción de altura máxima. Otra forma es la que aquí se describe, mediante una penalización que se suma al ajuste.

Para evitar redundancia en el código y un crecimiento excesivo de éste, puede incluirse un factor de parsimonia en el cálculo del ajuste (Soule & Foster, 1997) (Soule, 1998). Esta técnica se puede usar para reducir la complejidad del cromosoma que está siendo evaluado, y funciona mediante la penalización en el ajuste del individuo i de la siguiente forma:

$$f(i) = P(i) + \alpha \cdot s_i$$

Donde P(i) es una medida de la bondad del individuo (en este caso, peor cuanto más positivo), α es el nivel o coeficiente de parsimonia y s_i es el tamaño (número de nodos) del individuo. Con este coeficiente se está penalizando el número de nodos de un árbol, y su valor máximo suele ser de 0.1. Con este valor, se necesitará que el árbol tenga 10 nodos para incrementar en una unidad el valor de ajuste. Sin embargo, un valor tan alto ya es muy dañino en la evolución, y se suelen tomar valores menores (0.05, 0.01, etc), dependiendo del rango de valores en los que se espera que estén los ajustes de los individuos.

2.8. Parámetros

La Programación Genética tiene una serie de parámetros que han de ser configurados para poder ser ejecutada. Algunos de ellos, lógicamente, son idénticos a los existentes en los Algoritmos Genéticos. Otros, sin embargo, son propios de la Programación Genética. Los principales parámetros que se pueden configurar para variar la ejecución son:

- **Tamaño de la población.** Este parámetro indica el número de individuos que va a tener la población. En general este parámetro se ajusta de forma proporcional a la complejidad del problema, tomando valores altos cuanto mayor sea ésta. De esta forma, cuanto más complicado es un problema, habrá más opciones de conseguir mejores resultados en un menor número de generaciones, puesto que se generan más

individuos nuevos. Sin embargo, un tamaño alto puede no ser siempre la mejor solución: es posible tomar un tamaño menor y confiar más en la evolución durante mayor número de generaciones (Gathercole & Ross, 1997) (Fuchs, 1999).

- **Altura máxima del árbol.** Indica la altura máxima que van a poder alcanzar los árboles durante el proceso evolutivo. Este parámetro se ajusta también de forma proporcional a la complejidad del problema, de forma similar al anterior: un tamaño de población elevado exige elevar la altura máxima, pues si no existirá un gran número de individuos repetidos en la población y ello conlleva a problemas de dominio de un individuo, con los problemas que ello acarrea (fundamentalmente, pérdida de variedad genética y convergencia prematura del algoritmo). Para ajustar este parámetro y el anterior, y valorar la complejidad del problema, es conveniente tener en cuenta el número de elementos que tienen los conjuntos de elementos terminales y funciones, así como el número de variables y analizar el problema a solucionar.
- **La altura máxima que tendrán los árboles iniciales.**
- **La altura máxima que tendrán los subárboles** creados por una mutación de subárbol. Para una de estas mutaciones, el subárbol generado tendrá una altura máxima que será la mínima entre este parámetro y la altura permitida por la restricción de altura máxima en el nodo seleccionado.
- La **tasa de cruces**, es decir, el porcentaje de individuos de la siguiente generación que serán creados a partir de cruces de individuos de la anterior. Esta tasa suele ser alta, generalmente supera el 90%.
- La **probabilidad de mutación**. Esta probabilidad suele ser muy baja (menor de 0.05).
- **Algoritmos de creación, selección y mutación utilizados**.
- En los cruces, se utiliza también una **probabilidad de selección de nodo no terminal**, que se usa para escoger nodos del árbol: con esa probabilidad se escogerá un no terminal antes que un terminal (Koza, 1992) (Angeline, 1996a). Si no se utilizase este parámetro, la mayoría de los cruces serían solo intercambios entre elementos terminales de los árboles padre, puesto que en un árbol el número de terminales siempre es mayor o igual que el de no terminales, siendo a menudo mucho mayor.
- El **coeficiente de parsimonia** para penalizar árboles grandes.

- El **criterio de parada** del algoritmo. Al igual que con Algoritmos Genéticos, éste puede ser:
  - Número máximo de generaciones.
  - Haber alcanzado un valor de ajuste aceptable.
  - Convergencia de la población.

2.9. Ejemplo práctico

En esta sección se muestra un ejemplo de aplicación clásica de Programación Genética. Dada la capacidad que tiene de generar expresiones, la principal aplicación de la Programación Genética ha sido el desarrollo de expresiones matemáticas que permiten modelizar fenómenos físicos a partir de distintas mediciones, o realizar tareas de clasificación, como es el caso del ejemplo aquí descrito.

2.9.1. Descripción del problema

El problema a resolver consiste en clasificar mediciones de radar de la capa de la ionosfera. Estos datos de radar fueron tomados por un sistema en Goose Bay, Labrador, Canadá. Este sistema consiste en un conjunto de 16 antenas de alta frecuencia con un poder de transmisión total del orden de 6.4 kilowatios. Los objetivos fueron los electrones libres en la ionosfera. Las mediciones "buenas" son aquellas que muestran evidencia de algún tipo de estructura en la ionosfera. Por su parte, las "malas" son aquellas que no muestran tal evidencia, es decir, las señales pasan a través de la ionosfera. Las señales que se recibieron fueron procesadas utilizando una función de autocorrelación cuyos argumentos fueron el momento de un pulso y el número de este pulso. A partir de 17 pulsos distintos, de cada uno se extrajeron 2 atributos, correspondientes a los valores complejos devueltos por esta función para cada pulso. Por lo tanto, en total, se contaron con 34 atributos, a partir de los cuales se quiere predecir este tipo de presencia o no de estructuras. En este caso se cuenta con 351 instancias, es decir, 351 mediciones (Sigillito, Wing, Hutton & Baker, 1989).

Esta base de datos está disponible en internet, en el UCI (Asuncion & Newman, 2007), un repositorio de bases de datos para su aplicación con distintas técnicas de aprendizaje máquina.

Resolver este problema equivale a encontrar un modelo el cual, a partir de un conjunto de datos de entrada, produzca una salida. En este caso, los datos de entrada serán cada uno de los 34 atributos de la base de datos, y la salida será el valor de clasificación: "buena" o "mala". Para cada una de las

instancias de la base de datos se tiene el valor de la salida que debería de dar el sistema (conocido como salida deseada). Por lo tanto, a partir de las 351 instancias de pares entradas-salida deseada es necesario construir el modelo.

El primer paso en la construcción de un modelo siempre es el procesado de los datos. En este caso, la base de datos con la que se trabaja ya está depurada, y todos los datos están en un rango comprendido entre -1 y 1. De no ser así, sería necesaria una labor de análisis de la base de datos que incluyera tareas como eliminación de valores atípicos o normalización de los datos.

2.9.2. Codificación del problema

Para construir este modelo con Programación Genética, el primer paso es seleccionar cuáles son los conjuntos de terminales y funciones. Dado que se trabaja con 34 variables, obviamente habrá que introducir dichas variables en el conjunto de operadores terminales. Además, será necesario introducir números reales para que el sistema pueda operar con ellos. Esto se hace introduciendo un operador aleatorio, que cada vez que se introduce en un árbol creará un número real de forma aleatoria que, a partir de ese momento, mantendrá su valor. Estos valores aleatorios serán tomados en el rango entre -1 y 1.

Como conjunto de funciones se introducirán solamente operadores que permitan construir árboles en los que se opere con los terminales especificados. En general, en este punto es conveniente realizar un análisis del problema a resolver y de los datos que se poseen, para ver si existe alguna relación entre los mismos que pueda llevar a pensar que sería interesante introducir operadores, por ejemplo trigonométricos. Sin embargo, sin realizar este análisis es recomendable introducir, como mínimo, operadores aritméticos. Por lo tanto, los conjuntos de terminales y funciones son los mostrados en la tabla 2.1.

	Operadores	Núm. hijos
Terminales	X_1, …, X_{34}, [-1, 1]	-
Funciones	+, -, *, %	2

Tabla 2.1: Conjuntos de terminales y funciones

De esta manera, es posible construir árboles que, utilizando los operadores especificados, relacionen las variables para dar lugar a un modelo matemático. Un ejemplo de un posible árbol generado con estos operadores es el mostrado en la figura 2.11, el cual corresponde a la expresión $X_{13}/(-3.85*X_{21}) + X_2 - X_8$

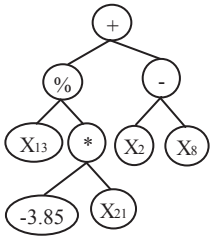


Figura 2.11: Árbol de ejemplo

Sin embargo, estos árboles producen una salida real, cuando lo que se desea es una salida que determine la clase en la que se clasifica cada instancia a partir de los valores que toman las variables. Para solucionar esto, es suficiente con establecer un umbral. Por ejemplo, si el resultado de evaluar un árbol es mayor o igual a cero, se toma que ese árbol clasifica esa instancia como "buena". En caso contrario, si el resultado de la evaluación es menor que cero, se clasifica como "mala".

2.9.3. Función de Evaluación

Respecto a la función de ajuste, se tomará una que evalúe el error que se comete en la clasificación. Por este motivo, un ajuste mayor implica un error mayor, y un ajuste igual a cero implica una precisión en la clasificación del 100%. Por lo tanto, se está tratando con un problema de minimización de un error.

El siguiente pseudocódigo muestra un ejemplo de cómo podría ser la función de ajuste, que se llama para evaluar cada árbol:

```
error = 0
para cada fila de la base de datos hacer,
      establecer los valores de las variables del árbol X1, ..., X34 como los valores de los
            atributos de esa fila
      valor = evaluar el árbol
      si (valor>=0),
            clase = "buena"
      si no,
            clase = "mala"
      fin si
      si clase no coincide con la clase deseada de esa fila de la base de datos,
           error = error + 1
      fin si
fin para
error = error / (número de filas de la base de datos)
devolver error
```

2.9.4. Ejemplo de resolución paso a paso

Una vez establecidos los conjuntos de terminales y funciones y la función de ajuste, sólo es necesario configurar los parámetros y proceder a la ejecución. Una configuración típica sería la siguiente:

- Tamaño de población: 1000 individuos.
- Algoritmo de selección: Torneo de 2 individuos.
- Altura máxima del árbol: 9.
- Altura máxima inicial: 6.
- Probabilidad de mutación: 0.03
- Probabilidad de selección de no terminal: 95%
- Tasa de cruces: 95%

Con esta configuración, se ejecuta el algoritmo, y, como resultado, se obtiene una expresión que tiene una determinada precisión en la clasificación. Es necesario notar que, dado el carácter estocástico del proceso, cada vez que se ejecute se obtendrá una expresión distinta. Un ejemplo es la siguiente expresión, que ofrece una precisión en la clasificación de un 91.99%:

(((((((-0.90963972) - X14) - X7) + (X3 - (X2 % X2))) - ((((-0.90963972) - (-0.90963972)) + ((-0.90963972) - (X2 % X3))) % (-0.90963972))) - ((X7 - (-0.90963972)) % (X7 - (X14 % X2)))) + (((X14 + ((-0.90963972) - ((X5 + X7) % ((-0.90963972) - X2)))) - ((((X3 + (0.47022715 + X5)) - X2) + X5) % ((((X2 % X3) % X7) - X14) - X7))) - (((((X3 + (0.47022715 + X5)) - X2) + (X2 + X14)) % (((X2 % X7) - (X5 - X2)) - X7)) - (-0.90963972))))

Sin embargo, para ilustrar el funcionamiento de la Programación Genética, se mostrará aquí un ejemplo de funcionamiento utilizando un tamaño de población de solamente 10 individuos. A pesar de que es un tamaño demasiado bajo como para obtener buenos resultados, sí que permite ver cómo se realiza el proceso evolutivo y cómo la población, a partir de una situación inicial en la que existen grandes diferencias entre los individuos, tiende con el tiempo hacia un mínimo del espacio de búsqueda.

La tabla 2.2 muestra un ejemplo de una generación inicial obtenida con un tamaño de 10 individuos. Como se puede ver, en esta población hay individuos de distintas alturas y complejidades.

Ajuste	Individuo
0.280000	X5
0.302857	X12 + X15
0.434286	((((X33+X16) * (X27+X21)) * ((X34 + X1) % (0.29157*X3))) - (((X26 * X11) - (X4 * X26)) + ((X3 - X13) * (X10 - X34))))
0.437143	X17
0.454286	X6
0.460000	(((((X23 * X2) + (X12 * X3)) + ((X20 - X33) * (X27 - 0.79946))) + (((X1 * 0.87272) + (X18 - X24)) + ((X27 % X3) % (X5 % 0.84773)))) + ((((X22 % X33) % (X18 + X19)) + ((X23 % X30) % (X5 + X25))) % (((0.16723 + X33) + (X11 * X10)) - ((X9 - X11) - (X23 - X5)))))
0.554286	((X32 + X23) - (X7 - X26)) % (X22 + X13 + (X5 * X2)
0.568571	(X9 * X26) - (0.54693 - X6)
0.594286	X34
0.594286	X34

Tabla 2.2: Generación inicial con 10 individuos

Es de destacar que, en estos individuos iniciales, han aparecido muy pocas constantes. Esto es debido a que en el conjunto de terminales existe una cantidad mucho mayor de variables (34) que de constantes (1), por lo que es mucho más probable la ocurrencia de las variables que de las constantes.

Ajuste	Individuo
0.257143	X12 + X5
0.280000	X5
0.357143	((((X33 + X34) * (X27 + X21)) * ((X34 + X1) % (0.29157 * X3))) - (((X26 * X11) - (X4 * X26)) + ((X3 - X13) * (X10 - X34))))
0.391429	((((X33 + X16) * (X27 + X21)) * ((X34 + X1) % (0.29157*X3))) - (((X26 * X11) - (X4 * X26)) + ((X3 - X13) * (X17 - X34))))
0.408571	X15
0.417143	X17 + X15
0.422857	X12
0.454286	X6
0.462857	X10
0.594286	X34

Tabla 2.3: Población tras una generación

La tabla 2.3 muestra el estado de la población tras hacerla evolucionar durante una sola generación. Como resultado de las operaciones de cruce, ha aparecido una gran cantidad de individuos que son solamente terminales. Si se deja evolucionar la población durante 5 generaciones más, se puede obtener una población similiar a la mostrada en la tabla 2.4, en la que se puede ver una clara tendencia hacia una expresión predominante. Si se dejase evolucionar unas cuantas generaciones más, seguramente la población convergería y todos los individuos serían idénticos. Esto no quiere decir que se haya alcanzado la mejor expresión, sino que se ha alcanzado un mínimo que, en esta ejecución del algoritmo, no se puede mejorar.

Ajuste	Individuo
0.217143	X28 + X5 + X5
0.231429	X18 + X5 + X5
0.242857	X28 + X5
0.242857	X28 + X5
0.251429	X12 + X5 + X5
0.251429	X12 + X5 + X5
0.262857	X28 + X5 + X18
0.262857	X10 + X5 + X5
0.302857	0.55628 + X5
0.440000	X28 + X10

Tabla 2.3: Población tras cinco generaciones más

En este caso se habría alcanzado un mínimo en un número muy bajo de generaciones. Sin embargo, esto es causado por el bajo tamaño de la población (10 individuos). Para poblaciones con tamaños mayores (100 – 1000 individuos) se necesitaría un número mayor de generaciones para lograr que la población se estanque, puesto que habría mayor variedad genética, por lo que se realizaría una exploración mucho más amplia del espacio de búsqueda. Además, un tamaño de población mayor reduciría las posibilidades de caer en un mínimo local y parar de forma prematura el algoritmo.

2.9.5. Implementación en Matlab

Para la resolución de este problema en Matlab, se utilizará un toolbox de Programación Genética denominado GPLAB, de libre distribución y disponible en internet (Silva, 2010).

Para utilizarlo, por lo tanto, será necesario descargárselo, descomprimirlo e incluirlo dentro del entorno de trabajo de Matlab mediante el comando *addpath*.

Antes de poder ejecutar el algoritmo, será necesario configurarlo convenientemente. Esto se realiza estableciendo los valores de los parámetros correspondientes. Para ello, será necesario crear una estructura con los parámetros para modificar los que sean pertinentes.

Algunos de los parámetros más importantes se refieren a qué función de ajuste se utiliza y si el algoritmo debe minimizar o maximizar el ajuste. Además, este toolbox permite especificar dónde están los archivos de patrones para cargarlos automáticamente. Todas estas acciones se pueden realizar con los siguientes comandos:

```
% Se toman los parametros por defecto
Parametros = resetparams;

% Como se cargaran los datos
Parametros = setparams(Parametros,'files2data=xy2inout');
Parametros = …
      setparams(Parametros,'datafilex=''.\entradas.txt''');
Parametros = …
      setparams(Parametros,'datafiley=''.\salidas.txt''');

% Funcion de Ajuste
Parametros.calcfitness='funcionAjuste';
% Se indica que se desea minimizar el ajuste
Parametros = setparams(Parametros,'lowerisbetter=1');
```

De esta forma se ha especificado, por ejemplo, que la función de ajuste se encuentra en un archivo llamado ‘ajuste.m’, que se describirá más adelante.

Para establecer los conjuntos de terminales y funciones es necesario modificar también esta estructura con los parámetros. Para realizarlo, se hace una llamada a las funciones pertinentes, como se puede ver en el siguiente ejemplo. Es importante tener en cuenta que, a la hora de introducir los operadores, estos deberán ser funciones de Matlab. Por esa razón, por ejemplo, la operación de división protegida será necesario definirla. Por suerte, ya viene definida en el toolbox, bajo el nombre de ‘mydivision’. Además, la inserción de las variables en el conjunto de terminales la puede realizar el toolbox de forma automatizada, a partir del análisis del archivo de patrones de entrada. De esta forma, se introducirá una variable por cada columna que tenga ese archivo.

Un ejemplo de código que configura estos conjuntos es el siguiente:

```
% Se introducen las funciones
Parametros = setfunctions(Parametros, 'plus', 2, 'minus', 2,
'times', 2);
% Se introduce tambien la division protegida
Parametros = addfunctions(Parametros, 'mydivide', 2);

% Se introducen los terminales (valores aletorios)
Parametros = setterminals(Parametros, 'rand', '1');

% Y se introducen en el conjunto de terminales las variables
Parametros = setparams(Parametros, 'numvars=[]');
Parametros = setparams(Parametros, 'autovars=1');
% El numero de variables es el numero de columnas
%  del archivo de entradas, estas variables se introducen
%  de forma automatica
```

Además, es posible configurar el resto de los parámetros del algoritmo, como por ejemplo la altura máxima, utilizando la misma estructura. El siguiente código muestra un ejemplo de configuración de una serie de parámetros del algoritmo:

```
% Algoritmo de creacion
% Algoritmo de creacion 'Ramped Half & Half'
Parametros = …
      setparams(Parametros,'initpoptype=rampedinit');
% Altura maxima de creacion
Parametros = …
      setparams(Parametros,'inicmaxlevel=6');

% Algoritmo de seleccion
Parametros = setparams(Parametros,'sampling=lexictour');
Parametros = setparams(Parametros,'tournamentsize=2');
      % 2 individuos

% Su usa altura maxima
Parametros = setparams(Parametros,'fixedlevel=1');
Parametros = setparams(Parametros,'realmaxlevel=9');

% Cuantos individuos seran creados en cada generacion
Parametros = setparams(Parametros, 'gengap=0.95');

% Elitismo
Parametros = setparams(Parametros, 'elitism=replace');
      % Los hijos reemplazan a los padres

% Condiciones de parada
Parametros = setparams(Parametros, 'hits=[100 0]');
      % Se para cuando se consiga un error de 0 en el 100%
      % de los patrones
```

Finalmente, para ejecutar el algoritmo será necesario especificar el tamaño de la población y el número máximo de generaciones que se ejecutará, en caso de no producirse la condición de parada anteriormente, además de los parámetros que se haya ejecutado. Un ejemplo de ejecución podría ser el siguiente, con un tamaño de población de 100 individuos y un límite de 30 generaciones:

```
TamanoPoblacion = 100;
NumGeneraciones = 30;
[EstadoAlgoritmo, Best] = …
      gplab(NumGeneraciones, TamanoPoblacion, Parametros);
```

Mediante este último comando comenzaría la ejecución del algoritmo, parándose cuando se verifique la condición de parada, o cuando se alcance el número máximo de generaciones prefijado.

Durante esta ejecución del algoritmo, se llamará repetidas veces a la función de ajuste, para evaluar cada uno de los individuos que se generen. Dicha función de ajuste, en este caso, es la transcripción al lenguaje de Matlab del pseudocódigo mostrado anteriormente, con algunas operaciones añadidas por exigencias del lenguaje de programación y del toolbox.

Un ejemplo de función de ajuste, correspondiente al pseudocódigo anterior, sería el siguiente:

```
function ind = funcionAjuste(ind,params,data,terminals,varsvals)

% Se establece primero los valores de las variables
X=data.example;
outstr=ind.str; % En outstr estara el string del individuo
% Para cada variable, se establece su valor
for i=params.numvars:-1:1
    % Se hace reemplazar en el string del individuo cada
    % aparicion de cada variable por su correspondiente valor
outstr=strrep(outstr,strcat('X',num2str(i)),strcat('X(:,',num2st
r(i),')'));
end

% Se evalua el individuo
try
    res=eval(outstr);
catch
    res=str2num(evaluate_tree(ind.tree,X));
end
```

```
if length(res)<length(data.result)
   res=res*ones(length(data.result),1);
end

% Se aplica un umbral en 0
res(find(res>=0))=1; res(find(res<0))=0;

% Se calcula el error medio y se asigna al individuo
sumdif=(sum(abs(res-data.result))) / length(data.result);
ind.result=res;

% Y el ajuste sera ese error
ind.fitness=sumdif;
```

Referencias

Aguirre, H. E., Tanaka, K. and Sugimura, T. (1999). "Cooperative crossover and mutation operators in genetic algorithms". Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-99. Morgan Kaufmann. Orlando, Florida. pp 772.

Angeline, P. J. (1996). "An investigation into the sensitivity of genetic programming to the frequency of leaf selection during subtree crossover". Genetic Programming 1996: Proceedings of the First Annual Conference GP-96. MIT Press. Stanford University, California, USA. pp 21-29.

Angeline, P. J. (1996) "Two self-adaptive crossover operators for genetic programming". Advances in Genetic Programming 2. MIT Press. Cambridge, MA, USA. pp 89-110. 1996.

Asuncion, A. & Newman, D.J. (2007) "UCI Machine Learning Repository". Irvine, CA: University of California, School of Information and Computer Science.

Chellapilla, K. (1997) "Evolutionary programming with tree mutations: Evolving computer programs without crossover". Genetic Programming 1997: Proceedings of the Second Annual Conference. Morgan Kaufmann. Stanford University, CA, USA. pp 431-438.

Cramer, M. L. (1985). "A Representation for the Adaptive Generation of Simple Sequential Programs". Proceedings of an International Conference on Genetic Algorithms and their Applications. Erlbaum.

Fuchs, M. (1999). "Large Populations Are Not Always The Best Choice In Genetic Programming". Proceedings of the Genetic and Evolutionary

Computation Conference (GECCO-99). Morgan-Kauffman. Orlando, Florida, USA. pp 1033-1038. 1999.

Fujiki, C. and Dickinson, J. (1987). "Using the Genetic Algorithm to Generate LISP Source Code to Solve the Prisoner's Dilemma". Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms. Erlbaum.

Gathercole, C. and Ross, P. (1997). "Small populations over many generations can beat large populations over few generations in genetic programming". Proceedings of the Second Annual Conference on Genetic Programming. Morgan-Kauffmann. pp 111-118.

Koza, J. R. (1990). Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems. Technical Report No. STAN-CS-90-314, Computer Sciences Department, Stanford University.

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

Luke, S. (2000). "Two Fast Tree-Creation Algorithms for Genetic Programming". IEEE Transactions on Evolutionary Computation.

Montana, D. J. (1995). "Strongly Typed Genetic Programming". Evolutionary Computation. The MIT Press. Cambridge, MA. pp 199-200.

Pereira F. B., Machado P., Costa E. and Cardoso A. (1999). "Graph based crossover-A case study with the busy beaver problem". Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-99. Morgan Kaufmann. Orlando, Florida, USA. pp 1149-1155.

Sigillito, V. G., Wing, S. P., Hutton, L. V., & Baker, K. B. (1989) "Classification of radar returns from the ionosphere using neural networks". Johns Hopkins APL Technical Digest, 10. pp. 262-266.

Silva, S. "GPLAB: A Genetic Programming Toolbox for MATLAB" [en línea] [fecha de consulta: 14 de abril del 2010] Disponible en <http://gplab.sourceforge.net/>

Soule, T. and Foster, J. A. (1997). "Code Size and Depth Flows in Genetic Programming". Genetic Programming 1997: Proceedings of the Second Annual Conference. Morgan Kauffmann. San Francisco, CA. pp 313-320.

Soule, T. (1998). Code Growth in Genetic Programming. PhD thesis. University of Idaho.