Experimentación en Programación Genética Multinivel

Ricardo Aler, Francisco Blázquez, David Camacho

Universidad Carlos III
Avda. de la Universidad, 30
Madrid, 28911
aler@inf.uc3m.es, dcamacho@ia.uc3m.es

Resumen

La programación genética (PG) es una técnica de aprendizaje automático que se basa en la evolución de programas de ordenador mediante un algoritmo genético. Una versión avanzada de la PG intenta aprovechar las regularidades de los dominios a resolver aprendiendo simultáneamente subrutinas que codifiquen dichas regularidades. Dicha versión, denominada ADF (definición automática de funciones) permite reutilizar una subrutina varias veces dentro de un mismo individuo. Sin embargo, existe la posibilidad de que la misma subrutina pueda ser reaprovechada por varios individuos de la misma población. Existen varios sistemas que, en principio, permiten descubrir subrutinas válidas para muchos individuos de una población. Uno de los más avanzados es el de red dinámica DL^{GP} propuesto por Racine, Schoenauer y Dague en 1998. Desgraciadamente, hasta el momento no existe ninguna evaluación experimental de DL^{GP}. El objetivo de este artículo es presentar una extensa experimentación de algunos aspectos de DL^{GP} y su análisis posterior. Además, los autores pretenden realizar una revisión de las variadas técnicas existentes en PG para evolucionar subrutinas.

Palabras clave: Programación Genética, Definición Automática de Funciones (ADF), Evolución de Subrutinas.

1. Introducción

La programación genética (PG) es una técnica de aprendizaje automático que se basa en la evolución de programas de ordenador mediante un algoritmo genético [Koza1992]. Aunque la investigación en PG ha tomado una gran multitud de rumbos, una de las ideas directrices de dicha investigación es que la PG pueda utilizar los mismos recursos que cualquier programador. Esta es desde luego la línea seguida por Koza. En particular, los programadores humanos utilizan subrutinas en sus códigos que pueden ser utilizadas en muchos lugares distintos del mismo programa. Desde un punto de vista abstracto, las subrutinas codifican regularidades del dominio a resolver. Utilizando dicho mecanismo, Koza encontró que, en efecto, aprender las subrutinas adecuadas permitía a la PG resolver los problemas de manera mucho más rápida y encontrando programas más cortos [Koza1994]. A la técnica anterior se la denomina ADF ("Automatic definition of Functions": definición automática de subrutinas). Sin embargo, las subrutinas no sólo se pueden reutilizar dentro de un mismo programa, sino que distintos programas se pueden beneficiar de la misma subrutina. Aplicando esta idea a la PG, varios individuos dentro de la misma población se podrían beneficiar de una misma subrutina recién descubierta, sin necesidad de tener que esperar a que la misma subrutina se redescubra de manera independiente en varios individuos, que es lo que debería ocurrir con las ADF. Por tanto, parece buena idea separar la evolución de los programas principales de la evolución de las subrutinas. Esta idea fue aplicada de diversas maneras en [Ahluwalia1997] y [Aler1998] con buenos resultados. En 1998, [Racine1998] propuso un método similar llamado DL^{GP}, en el que se disponía de una población de programas principales y varias poblaciones de subrutinas, organizadas en distintos niveles, de manera que una subrutina sólo puede llamar a subrutinas de nivel inferior. Desgraciadamente, hasta el momento los autores no han publicado una evaluación experimental de la idea, a pesar de que no es obvio que la arquitectura DL^{GP} tenga que funcionar necesariamente o de que lo haga de manera eficiente. El objetivo de este artículo es realizar una evaluación experimental de dicha arquitectura. Desafortunadamente, los resultados parecen ser negativos, al menos con la variante de DL^{GP} implementada en este artículo.

3. Estado de la Cuestión

Esta sección resume el estado de la cuestión relativo a la evolución de subrutinas. En primer lugar, se introducirán de manera breve las ideas básicas de la PG. A continuación, y de manera más extensa, se revisarán los distintos modos experimentados hasta la fecha de evolucionar subrutinas. Para terminar, se describirá de manera exhaustiva las ideas básicas de DL^{GP} (Dynamic Lattice Genetic Programming) en las que se basa la experimentación desarrollada en este artículo [Racine1998].

3.1. Programación Genética

No se pretende explicar en este artículo todos los detalles de la PG. Para ello, se remite al lector a [Koza1992]. Aquí tan solo se resumirán los puntos principales. Los elementos principales de la PG son tres:

- Una población P de individuos $i \in I$, que son programas de ordenador. Dichos programas están constituidos por funciones primitivas elegidas de un conjunto F y terminales elegidos de un conjunto T. Por ejemplo, *i*=`IF X<5 THEN X=(X+5)` sería un ejemplo de individuo donde *F*=`{IF-THEN,<,=,+}` y *T*=`{X,5}`.
- Una función de adecuación (o *fitness*) $f(x): I \rightarrow \Re$ que mide la idoneidad de cada individuo. Para poder realizar esta medida, la PG ejecuta al individuo en distintos contextos (o sea, distintos valores de entrada) y se comprueba como de correcta es la salida en cada uno de esos contextos.
- Unos operadores genéticos $O=\{m,c,r\}$ que son capaces de transformar unos programas de ordenador en otros. Los operadores que se suelen usar son la mutación de un individuo $m: I \rightarrow I$, el cruce de dos individuos (o recombinación o *crossover*) $c: IxI \rightarrow IxI$ y la reproducción sin modificación $r: I \rightarrow I$. Cada operador tiene una probabilidad asociada $P(o)$ de ser utilizado. Estos operadores permiten explorar el espacio de los programas de ordenador.

A continuación se describe el algoritmo estándar de la PG:

1.Comenzar en la generación $g=0$
2.Generar $P(0)$ compuesta de individuos con elementos de F y T elegidos de manera aleatoria.
3.Realizar, iterativamente, los subpasos siguientes hasta que se encuentre una solución o se llegue a un determinado número de generaciones:
  3.1.$\forall i \in P(g)$ calcular $f(i)$
  3.2.Realizar hasta que se llene la población $P(g+1)$
    3.2.1.Seleccionar $o \in O$ de acuerdo a $P(o)$
    3.2.2.Si $o=m \vee o=r$
      Elegir i de acuerdo a $f(i)$
      Calcular sucesor $s=o(i)$
      Introducir s en $P(g+1)$
    3.2.3.Si $o=c$
      Elegir i_1 e i_2 de acuerdo a $f(i_1)$ y $f(i_2)$
      Calcular sucesores $(s_1,s_2)=c(i_1,i_2)$
      Introducir s_1,s_2 en $P(g+1)$
  3.3.$g=g+1$
4.Devolver el mejor individuo encontrado

3.2. Evolución de Subrutinas

3.2.1. Funciones Definidas Automáticamente (ADF)

Para mejorar el rendimiento de la PG convencional, Koza desarrolló un paradigma nuevo donde un individuo contiene, a la vez, un programa principal y un conjunto de subrutinas que pueden ser llamadas desde el cuerpo principal [Koza1994]. El operador de recombinación está modificado para que el cruce sólo ocurra entre partes homogéneas de los programas. Por ejemplo, la subrutina 1 de un programa sólo se puede recombinar con la subrutina 1 de otro programa.

En este paradigma, la única manera que tiene una subrutina (o partes de ella) de propagarse en la población es a través del operador de cruce. Esto es lo que ocurre en PG estándar con cualquier otro subárbol de un individuo. Esto quiere decir que no es posible que otros programas puedan reutilizar una subrutina inmediatamente que es descubierta. Por tanto, no es sencillo para el paradigma de ADFs el descubrir subrutinas que sean útiles para resolver un problema a muchos individuos distintos: cada subrutina co-evoluciona con su programa correspondiente. Un detalle a tener en cuenta es que las ADFs sólo son útiles a partir de determinado grado de complejidad de un problema. Es decir, para problemas simples es mejor no utilizarlas.

Dentro de este paradigma, [Spector1995] describe un sistema, similar a las ADF de Koza, llamado Macros Definidas Automáticamente (ADM). La diferencia es que las macros no evalúan sus argumentos hasta que se los necesita, mientras que las funciones los evalúan inmediatamente. Usar ADMs es la única manera en que un programa puede evolucionar sus propias estructuras de control de manera eficiente. Por ejemplo, para la típica estructura *if-then-else* es preferible que la parte *else* sólo se evalue en caso de que la condición no sea cierta, especialmente si dicha parte else produce efectos colaterales (tales como cambiar el valor de una variable global). Spector muestra que en algunos problemas, el uso de ADMs reduce el esfuerzo computacional requerido por la PG.

3.2.2. Constructor de librerías genéticas (GLiB)

El objetivo de GLiB es el de proteger buenos subárboles almacenándolos en una librería, de manera que no puedan ser destruidos por las operaciones genéticas [Angeline1992,1993,1994]. Para ello, GLiB utiliza el llamado operador de compresión. Dicho operador toma un subárbol de un individuo de manera aleatoria, lo parametriza y lo guarda en una librería. En este momento, el subárbol se ha convertido en una nueva función primitiva que puede ser utilizada por cualquier otro programa en la población. La manera de transferir esta nueva primitiva a nuevos programas es bien a través de cruce con el individuo comprimido, bien a través de mutación. Puesto que la compresión elimina subárboles de la población, la diversidad genética de esta disminuye, por lo que los autores utilizan también el operador de expansión, el cual substituye una llamada a una primitiva almacenada en la librería por el subárbol de dicha primitiva. El sistema se comportar bien en el dominio del tres en raya y el problema del “camino de comida”.

GLiB permite la transferencia de subárboles inalterados de unos individuos a otros, con mucha más facilidad que las ADFs, gracias a la compresión. Una vez transferidos, cada individuo comprueba (a través de la evaluación de su idoneidad) si la nueva primitiva le es útil o no, de manera que aquellas nuevas primitivas útiles se extenderán rápidamente en la población. Por tanto, el sistema aprenderá rutinas que son útiles no sólo a un individuo, sino a toda la población. Sin embargo, puesto que los subárboles a comprimir se eligen sin ningún criterio, muchos de los que son almacenados carecerán de valor, lo que resta eficiencia al sistema.

GLiB está basado en el operador de encapsulación definido en [Koza1992], con la diferencia de que la encapsulación almacenaba subárboles sin parametrizar. Es decir, generaba subrutinas sin parámetros.

Las subrutinas almacenadas en la librería no compiten entre sí. Teller y Veloso definen un sistema híbrido llamado PADO que también utiliza una librería de subrutinas [Teller1996]. A diferencia de GLiB, las subrutinas compiten mediante sus idoneidades que están basadas en una media ponderada de las de los programas que las llaman. En cada generación, se borra el peor individuo de la librería y se sustituye por las ADF del mejor programa principal.

3.2.3. Representaciones adaptativas (ARL)

Rosca y Ballard describen el método de representaciones adaptativas mediante aprendizaje (ARL: Adaptive Representation through Learning) para la inducción de subrutinas [Rosca1994a,Rosca1994b,Rosca1995,Rosca1996a, Rosca1996b]. ARL también intenta encapsular en subrutinas los buenos subárboles presentes en la población, pero a diferencia de GLiB, no selecciona dichos árboles de manera aleatoria. El operador de creación busca subárboles en aquellos individuos que son mejores que sus padres. Además, selecciona los subárboles de entre aquellos que se ejecutan un mayor número de veces. Esto tiene sentido, puesto que la evaluación de un individuo en PG consiste o bien en ejecutar un mismo individuo en diversos contextos para evaluar su generalidad, o bien en ejecutar dicho individuo varias veces, en caso de que el dominio sea no determinista. Una restricción adicional es que, por razones de eficiencia, ARL se concentra en subárboles de pequeño tamaño. Una vez que un bloque útil ha sido detectado, se le da un nombre de subrutina y se generaliza (algunos de sus nodos terminales se convierten en parámetros de la nueva subrutina). Por último, se guarda en una librería. Cada cierto tiempo, se “mata” a un determinado porcentaje de programas y se crean otros tantos nuevos, de manera que las nuevas subrutinas tienen la oportunidad de entrar en la población. Por último, las subrutinas de la librería pueden desaparecer de la misma si son poco útiles, es decir, si la media de las idoneidades de los programas que la usan es demasiado baja. Cuando se borra una subrutina, la llamada es sustituida por su código equivalente en cualquier programa que la llame. ARL ha sido experimentado extensivamente en el PAC-MAN, con muy buenos resultados.

ARL permite descubrir subrutinas útiles generalizando el uso en la población de bloques eficientes de código. Sin embargo, una vez que han sido extraídas, las subrutinas dejan de evolucionar, a pesar de que no tienen porque estar completamente optimizadas.

3.2.4. Funciones coevolutivas

Al igual que GLiB y ARL, las funciones co-evolutivas separan los programas de sus subrutinas en poblaciones diferentes. Pero a diferencia de ellos, las funciones co-evolutivas permiten que las subrutinas continuen evolucionando en sus propias poblaciones. De hecho, co-evolucionan al mismo tiempo que los programas principales. La co-evolución no es una idea nueva en el campo de la computación evolutiva. Los principales proponentes dentro de la PG son Ahluwalia, Bull y Fogarty quienes las introdujeron en [Ahluwalia1997].

Puesto que los programas principales y las subrutinas están separadas, es necesario resolver dos problemas. El primero ocurre a la hora de ejecutar el programa principal y encontrar una llamada a subrutina. ¿Cuál de las subrutinas de la población de subrutinas correspondiente debería ser llamada?. El segundo problema es cómo asignar idoneidad a las propias subrutinas, puesto que lo único que es evaluable directamente es el programa principal. Este último problema se puede resolver de dos maneras. O bien se dispone de una función de idoneidad para cada una de las poblaciones de subrutinas (distinta de la función de idoneidad del programa principal), o bien se le asigna a cada subrutina la idoneidad del programa principal que la usa. Para aplicar esta segunda opción es necesario resolver el primer problema, es decir, cómo enlazar individuos -subrutinas y programas principales- que están en poblaciones distintas.

[Ahluwalia1997] estudia diversas maneras de asociar programas principales y subrutinas. La selección puede ser aleatoria, o bien estar basada en la idoneidad (el programa principal selecciona con preferencia aquella subrutina que tiene una idoneidad alta, aunque la selección suele ser no determinista). Obsérvese que esto permite que se compruebe la generalidad de las subrutinas, puesto que en distintas generaciones pueden estar asociadas a programas distintos. Sin embargo, Ahluwalia advierte que en ocasiones los sistemas co-evolutivos tienen un comportamiento no convergente. En efecto, si un programa es evaluado en distintas ocasiones con muy distintas subrutinas, es posible que la evaluación del programa sea prácticamente aleatoria, puesto que su comportamiento depende en gran medida de con qué subrutinas está asociado. Para solventar este problema, en aquellos casos en los que se produzca, Ahluwalia propone que los programas principales se vinculen con las subrutinas de manera definitiva. Desgraciadamente, es difícil ver en qué difiere esto de las ADFs de Koza [Koza1994].

En un trabajo posterior, Ahluwalia combina las ideas de GliB y la coevolución de rutinas. Denomina su sistema EDF (Evolution Defined Functions) [Ahluwalia1998]. Funciona de la siguiente manera. Cada cierto tiempo, un subárbol es seleccionado aleatoriamente de un individuo. Pero a diferencia del operador de compresión, este subárbol es mutado repetidas veces, dando lugar a una población entera de subrutinas, las cuales continuan evolucionando por separado. Dichas subrutinas pueden ser llamadas por los programas principales. Caso de que el número total de llamadas a las subrutinas sea muy bajo, se aplica el operador de expansión, que en este caso consiste en eliminar la población de subrutinas y substituir las llamadas en los programas principales por subrutinas elegidas aleatoriamente de esa misma población. Las diferencias con GliB son por tanto dos. La primera es que el número de clases de subrutinas a guardar es mucho menor. En GliB, se podían guardar cientos de ellas, mientras que EDF sólo crea unas pocas poblaciones de subrutinas. Cada población de subrutinas contiene muchas de ellas, pero los programas principales las usan como si se tratara de la misma rutina. Por tanto, hay muchas subrutinas pero pocos tipos de ellas. La segunda diferencia es que las subrutinas almacenadas en la librería pueden continuar evolucionando y adaptándose a los programas principales. Además de EDF, Ahluwalia aplica EDF(auto), en el que el número de poblaciones de subrutinas se decide automáticamente: caso de que una población de subrutinas sea muy usada, se crea una nueva población. Tanto EDF como EDF(auto) se comportan muy bien experimentalmente en los dominios seleccionados.

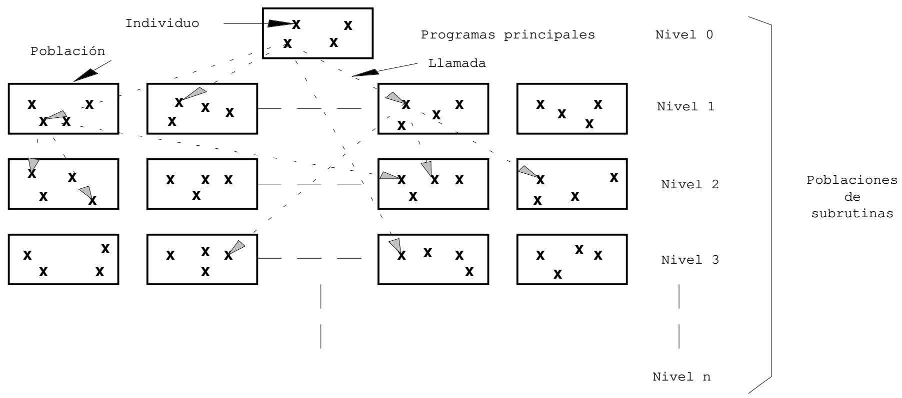

Figura 1 Topología en red. Existe una población de programas principales (Nivel 0) que pueden llamar a subrutinas de niveles inferiores (Nivel 1, 2, …, n).

El trabajo presentado en [Aler1998] también se basa en la co-evolución separada de programas principales y de subrutinas. El estudio intenta averiguar si la transferencia de buenas subrutinas encontradas por una población de subrutinas a otra población que las usa, es útil o no. Básicamente, esta idea consistiría en vincular a cada individuo de una población con los mejores individuos encontrados por las otras poblaciones. Es como si una población de subrutinas transfiriera de manera inmediata las mejoras encontradas a los individuos de las otras poblaciones, de una manera similar a como hacen los científicos. Los resultados muestran que el esfuerzo computacional necesario para obtener los mismos resultados que las ADF es algo menor y que retrasa el problema de la convergencia prematura de la PG.

3.3. Red dinámica DL^{GP} (Dynamic Lattice Genetic Programming)

En [Racine1998] se propone un sistema para la co-evolución de subrutinas y programas a la que denomina red dinámica DL^{GP}. Al igual que las funciones co-evolutivas de Ahluwalia, Racine separa las poblaciones de programas y subrutinas y jerarquiza estas últimas. Es decir, hay diversas poblaciones de subrutinas, pero las de un nivel sólo pueden llamar a las de nivel inferior. Además, la vinculación entre programas y subrutinas es fija: cuando un programa llama a una subrutina, este especifica a qué individuo de qué población está llamando realmente. De esta manera, distintos programas pueden estar vinculados a la misma subrutina simultáneamente, por lo que dicha subrutina puede ser probada en distintos contextos. La hipótesis básica de DL^{GP} es que cada población debería concentrarse en resolver una subtarea particular del problema durante el proceso de evolución.

La estructura de DL^{GP} puede verse en la Figura 1[1]. Las cajas representan poblaciones que se organizan a distintos niveles. Las líneas discontinuas representan llamadas desde el cuerpo de un programa principal o una subrutina a otras subrutinas. Dichas llamadas sólo ocurren hacia niveles inferiores. Cada individuo puede llamar a uno o varios individuos, debido a lo cual el gráfico se muestra como una especie de red. Pero debido a la evolución, cada individuo puede cambiar, por lo que la red cambiará con el tiempo. De ahí el nombre de Red Dinámica. Puede observarse también que cada nivel tiene varias poblaciones. Cada una de esas poblaciones corresponde al número de argumentos de las subrutinas que hay en ella. Así por ejemplo, el nivel 1 tiene tres poblaciones que corresponderían a subrutinas con cero, uno y dos argumentos, respectivamente. En principio no hay ninguna objeción para que en un mismo nivel haya varias poblaciones de subrutinas correspondientes al mismo número de argumentos, pues cada una de ellas podría especializarse en una tarea distinta.

De entrada, esta estructura presenta ciertas posibles ventajas frente a sistemas vistos anteriormente. En ADFs, ADMs y otros, cada subrutina es poseída por un único programa principal, por lo que es complicado transferir y evaluar la utilidad de dicha subrutina para otros programas principales. Otras técnicas, como GLiB no distinguen subrutinas y cuerpo principal en el uso de los operadores genéticos. Por lo tanto, la recombinación es anárquica y la convergencia hacia una habilidad específica no es estable.

[1] *Figura extraída de [Racine1998]*

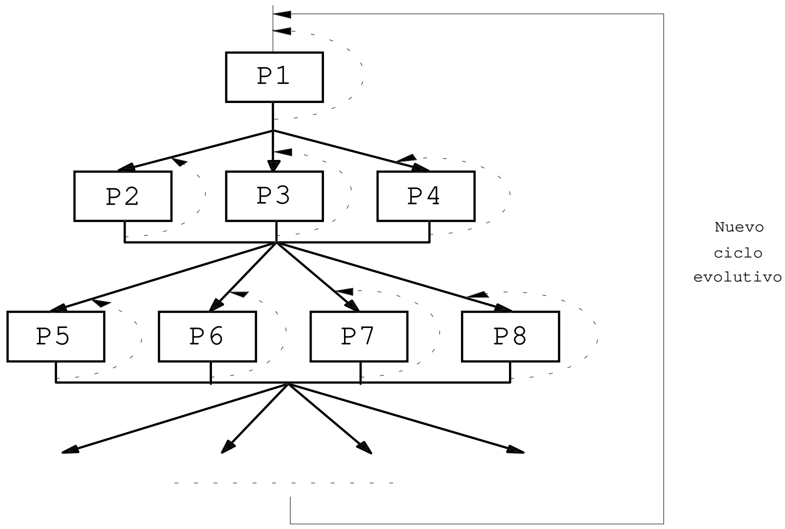

Figura 2 Ciclo co-evolutivo de la topología en red. Se realiza un ciclo evolutivo de la población de programas principales, seguido de ciclos evolutivos en las poblaciones de subrutinas.

Además del operador de cruce estándar y varios operadores de mutación, DL^{GP} define un operador de mutación específico denominado *Mutación de subrutinas*. En este operador se substituye una llamada a subrutina por una llamada a otra subrutina con el mismo número de parámetros. La función de este operador es permitir al programa experimentar con otras subrutinas. En realidad, otras mutaciones pueden producir el mismo efecto pero de manera más azarosa.

DL^{GP} propaga de arriba a abajo la idoneidad. Primero, se ejecutan todos los programas de la población principal tras lo cual se puede obtener su idoneidad. Ahora, las poblaciones de segundo nivel pueden calcular su idoneidad a partir de la de los programas principales, y así sucesivamente. Obsérvese que los únicos individuos que pueden obtener su idoneidad directamente son los programas principales. Las subrutinas calculan la idoneidad a partir de la de los programas principales y la de otras subrutinas. Pueden definirse muchas maneras de calcular dicha idoneidad. [Racine1998] propone la siguiente fórmula para calcular la idoneidad de las subrutinas, si se desea minimizar:

$$\begin{cases} \dfrac{1+\alpha * \overline{F_{x\%bcf}} + \beta * F_{best}}{1+\lambda * N_C}, \text{si } N_C \neq 0 \\ F_{max}, \text{ en otro caso} \end{cases}$$

donde:

- $\overline{F_{x\%bcf}}$ es la media de las idoneidades de las x% mejores funciones llamadoras.
- F_{best} es la idoneidad de la mejor función llamadora.
- N_c es el número de llamadas hacia la subrutina actual.
- F_{max} es la idoneidad maximal (correspondiente al individuo peor).
- α, β, λ son pesos especificados por el usuario.

Tras la inicialización aleatoria de cada una de las poblaciones, comienza el llamado ciclo co-evolutivo[1], que funciona de arriba a abajo en la jerarquía. Así, DL^{GP} puede dividirse en dos partes: población principal (o superior) y poblaciones de nivel inferior. Cada población del mismo nivel sufre uno o varios ciclos evolutivos locales. Esto incluye las tres etapas básicas: (1) Cálculo de las idoneidades (2) selección de acuerdo a la idoneidad, y (3) reproducción y aplicación de los operadores genéticos para crear/modificar la nueva población. Después, el proceso comienza otra vez con poblaciones que pertenecen al siguiente estrato. Este proceso se describe en la Figura 2[2]. Las líneas de

[1] *Puesto que la evolución de una población influencia la evolución de las otras. Por ejemplo, el cómo evolucione una subrutina influye en la evolución de los programas principales que las llaman, y viceversa.*

[2] *Figura extraída de [Racine1998]*

puntos corresponden a los ciclos locales.

De todo el proceso co-evolutivo, merece la pena destacar el modo de manejar la descendencia, una vez que se han aplicado los operadores genéticos. En la población de programas principales, cada ciclo evolutivo renueva la población por completo. Esto es lo que se denomina modelo generacional. Sin embargo, hacer esto mismo en las poblaciones de subrutinas conduciría a un sistema muy inestable. En efecto, el comportamiento de los programas superiores depende del de las subrutinas. Por tanto, si en cada ciclo se renueva por completo una población de subrutinas, el comportamiento de gran parte de los programas llamadores cambiará de golpe. Es por tanto necesario una renovación más gradual de las poblaciones de subrutinas. Para ello, se sigue el modelo de estado estable (*steady-state*), en el que en cada ciclo sólo cambian un pequeño porcentaje de individuos de la población. En este caso, además de seleccionar que individuos se van a reproducir, también es necesario seleccionar que individuos van a ser sustituidos por los nuevos descendientes. Según Racine, esto puede hacerse de dos maneras:

- O bien seleccionando uno de los peores individuos en la población
- O bien substituyendo una de las rutinas padre[3] por la rutina descendiente, siempre que esta última sea mejor que la rutina padre. El problema de esta alternativa es que para conocer la idoneidad de la rutina descendiente es necesario re-evaluar todos los programas principales que directa o indirectamente llamen a la rutina padre, habiendo sustituido previamente la rutina padre por la descendiente. En suma, por cada substitución, es necesario realizar un gran número de re-evaluaciones.

4. Descripción del sistema utilizado para la experimentación

El sistema de *programación genética multinivel* desarrollado aquí pretende ser una particularización del sistema multipoblación en *red dinámica* (DL^{GP}) de Racine, Schoenauer y Dague según el corto artículo publicado al respecto en [Racine1998]. El sistema de *programación genética multinivel* tiene exactamente la misma topología jerárquica y practica la misma evolución local que DL^{GP}. Ha sido desarrollado en Common Lisp, a partir de la implementación producida por Koza [Koza1992] y de código desarrollado por uno de los autores [Aler1998].

El sistema se ha estructurado como un conjunto de 1 a *n* niveles, cada uno de los cuales está compuesto por un conjunto de *p* poblaciones y, a su vez, cada población se compone de *i* individuos. El nivel *n-1* es el inferior y está compuesto de *p* poblaciones de *i* individuos cuyas subrutinas solamente pueden llamar a las funciones primitivas definidas por el usuario. El nivel 0 contiene una sola población de programas principales. Dicha población principal sigue el modelo de evolución generacional (en cada generación se reemplaza la población completa) mientras que las poblaciones de subrutinas utilizan el modelo *steady-state*, donde en cada ciclo evolutivo sólo cambia un individuo de la población. Para ello hay que seleccionar qué individuo de la población va a ser substituido. En los experimentos se han permitido dos posibilidades que se determinan por medio de un parámetro: que los individuos padres del descendiente sean substituidos o que lo sean los individuos peores. Los efectos de cada una de estas políticas se estudiarán posteriormente. Hay que destacar que nunca se comprueba que la rutina descendiente sea mejor que la rutina progenitora, debido al gasto computacional que conllevaría (habría que re-evaluar todos los programas principales que directa o indirectamente llamaran a la rutina padre, substituyendo esta por la rutina hija, para comprobar el efecto del cambio de una por la otra). Además, tampoco se ha implementado la mutación que cambia una llamada a una subrutina por una llamada a otra subrutina, aunque este cambio se puede conseguir por la mutación y el cruce estándares. Se ha hecho así porque se quería comprobar como se comportaba un sistema simple, antes de introducirle nuevas complicaciones.

Una llamada a una subrutina aparece en un programa como una lista cuyo primer elemento es otra lista de tres elementos que indica el nivel, población e individuo donde está almacenado su código y el resto de elementos son los argumentos que se le pasan al programa o subrutina. Antes de poder evaluar un programa principal es necesario expandirlo, es decir, cambiar las llamadas a subrutinas por las subrutinas mismas. Aunque esta forma es la menciona en [Racine1998], pueden encontrarse otras opciones aplicables, como por ejemplo, que el programa principal se ejecute y cuando se encuentre una llamada a una subrutina

[3] *La rutina padre es aquella a la que se le aplica la operación genética (mutación o cruce) y produce al descendiente.*

simplemente se salte al código de la misma, sin necesidad de expandir el programa principal. En cualquier caso, una vez que el programa ha sido expandido, se debería compilar para que su ejecución sea más eficiente.

La función de adecuación utilizada es la propuesta por Racine [1998] y descrita en la sección anterior:

$$\frac{1+\alpha * m+\beta * j}{1+\gamma * l}$$

Los valores de α, β, λ son el inverso de los valores máximos que pueden alcanzar las magnitudes ponderadas por dichos coeficientes, para normalizar dichas magnitudes. Estos coeficientes se calculan de manera dinámica, de manera que pueden ser diferentes en generaciones diferentes. Sin embargo, para una generación y una población particulares, los tres coeficientes son los mismos para todos los individuos en la población.

El método de selección utilizado es el de ***torneo***. Este método se utiliza tanto para elegir que individuos se van a reproducir y modificar mediante los operadores genéticos, como para elegir a aquellos individuos que van a ser substituidos por los descendientes (en el caso de las poblaciones de subrutinas). El método de torneo consiste en seleccionar al mejor (o al peor, según el caso) de un pequeño subconjunto de individuos elegidos aleatoriamente.

5. Experimentación

5.1 Introducción

El propósito de esta sección es exponer los experimentos realizados para estudiar el comportamiento del sistema de programación genética multinivel.

Las pruebas iniciales mostraron que una configuración que muestra un buen comportamiento se compone dos o tres niveles, tres o cuatro poblaciones de subrutinas por nivel, 150 subrutinas por población, una población de programas de 4000 individuos y una profundidad máxima inicial de 3 para los árboles que representan a programas y subrutinas. Esta configuración constituye el punto de arranque de los experimentos a los que se ha sometido el sistema.

Para realizar experimentos se ha utilizado el conocido problema de la regresión simbólica de la paridad par de orden 4. Este problema consiste en evolucionar programas de ordenador que determine si en sus cuatro argumentos binarios (0 ó 1) hay un número par de unos. Este problema es difícil de resolver para la PG debido a la gran extensión del espacio de búsqueda. Es también un problema muy utilizado, por lo que se pueden comparar los resultados obtenidos con los de otros modelos existentes, especialmente con las ADF de Koza.

Se han realizado 5 experimentos, cada uno de los cuales consiste en una serie de 17 ejecuciones del sistema[4] durante 51 generaciones con un mismo grupo de parámetros que las gobiernan exceptuando, claro está, la semilla para el generador de números aleatorios que permite la obtención de poblaciones aleatorias iniciales distintas.

5.2. Experimento de control

En primer lugar, se ha realizado un experimento, con el código propuesto en [Koza1994] para tener una referencia del comportamiento de sistemas similares, en este caso programación genética con funciones definidas automáticamente (ADF). La Figura 3 muestra la evolución de la media (los puntos) y la desviación típica (líneas verticales) de las 17 ejecuciones de esta configuración. Los resultados son bastante buenos, pues el sistema siempre encuentra un programa que resuelve el problema de la paridad par de orden 4, como muy tarde en la generación 17. Estos serían los resultados a superar por la programación genética multinivel.

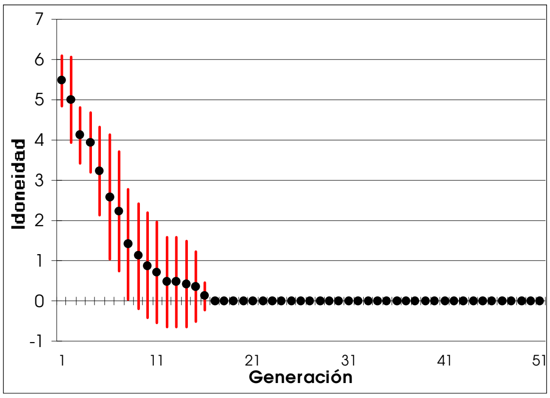

Figura 3 Idoneidad media de 17 ejecuciones del problema de paridad-4 con ADFs.

[4] *La PG es un método estocástico, por lo que para evaluar correctamente los resultados, es necesario ejecutar el sistema varias veces.*

5.2. Ciclo continuo vs. Ciclo alterno

Tal como se ha programado nuestra versión de DL^{GP}, el ciclo co-evolutivo genera una nueva población de programas y después genera un nuevo individuo por cada una de las poblaciones de subrutinas. Sin embargo, la modificación de la primera población de subrutinas, cambia de manera indirecta algunos de los programas de la población principal (de aquellos que directa o indirectamente llamen a dicha subrutina). Sin embargo, la idoneidad de dicha población principal no se vuelve a calcular hasta que se han modificado todas las poblaciones de subrutinas. Es decir, hasta que se ha producido un ciclo co-evolutivo completo. Es posible que este retardo en la actualización de la idoneidad tenga un efecto pernicioso. Para ello se han realizado dos experimentos. Uno en el que la idoneidad de los programas principales no se re-evalúa hasta que se realizado el ciclo co-evolutivo completo (ciclo continuo) y otro en el que la idoneidad de los programas afectados se re-evalúa tras la modificación de cada población de subrutinas (ciclo alternado). Hay que destacar que el esfuerzo computacional en ambos experimentos es más o menos el mismo, puesto que se ha forzado a que el número de veces que se evalúa la población principal para calcular su idoneidad coincida con el número de generaciones. Eso significa que en el ciclo alternado, los operadores genéticos se aplicarán un número menor de veces a las subrutinas, aunque el resultado de aplicar una operación genética se conocerá de inmediato.

Los resultados pueden verse en la Figura 4 para el ciclo continuo y la Figura 5 para el ciclo alterno. El primer resultado a observar es decepcionante: ninguna de las dos configuraciones lo hace tan bien como la configuración con ADFs (ver Figura 3). El ciclo continuo encuentra soluciones en 3 de las 17 ejecuciones y existe una gran variación de resultados en cada una de las ejecuciones (gran desviación típica), incluso en las últimas generaciones. El ciclo alternado encuentra una solución en sólo una de las ejecuciones. Sin embargo, su comportamiento parece más consistente, pues su desviación típica es bastante menor y la media a lo largo de las generaciones muestra una mayor tendencia de progreso. En efecto, el ciclo continuo se estanca prácticamente en la generación 11, mientras que el ciclo alterno progresa hasta la generación 40. Posiblemente, la información suministrada por las evaluaciones de idoneidad extra proporciona una guía más fiable que en el caso continuo. Por tanto, sí parece que el tipo de ciclo tiene cierta influencia en el comportamiento del sistema. A partir de este momento, todos los experimentos tendrán el ciclo continuo.

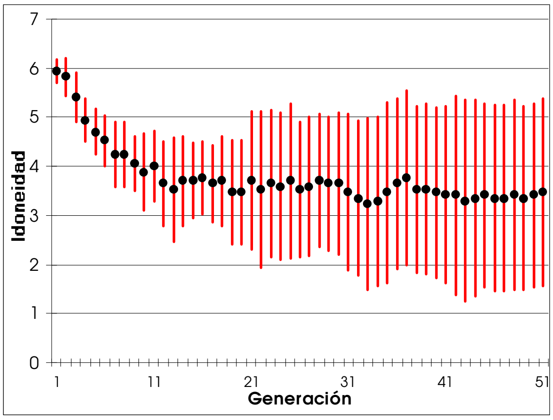

Figura 4 Idoneidad media para el problema de paridad-4 con ciclo contínuo.

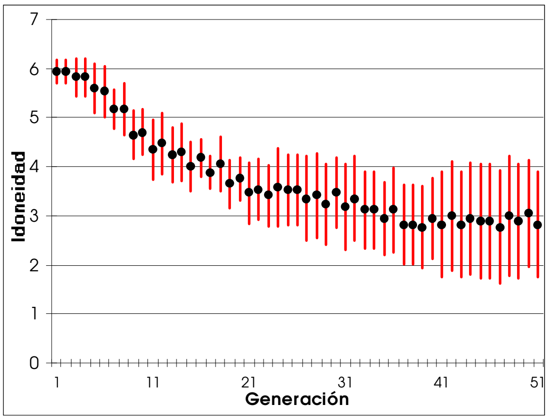

Figura 5 Idoneidad media para el problema de paridad-4 con ciclo alterno.

5.3. Políticas de substitución

En los experimentos anteriores, al generar nuevos individuos para las poblaciones de subrutinas, es necesario elegir qué individuos serán substituidos por los nuevos descendientes. En los experimentos anteriores, las rutinas a substituir son siempre las rutinas padre (es decir, la rutina a la que se le aplicó mutación, o una de las dos rutinas a las que se les aplicó cruce). La situación es que puede que hubiera muchos programas principales que, directa o indirectamente, llamaran a dicha rutina padre. Si la rutina descendiente incorporara mejoras, todos los programas principales dependientes se beneficiarán inmediatamente de dicha mejora [Aler1992]. Pero puede ocurrir que algunos o todos los programas principales dependientes empeoren debido al cambio de subrutina. Esto haría que el sistema diera saltos bastante aleatorios en el espacio de programas. Esto explicaría que la desviación típica de las ejecuciones de los dos experimentos anteriores sea tan elevada. Para comprobar este fenómeno, se ha realizado otro experimento en el que las rutinas que se substituyen no son las rutinas padre, sino la rutina a substituir se elige mediante torneo: se eligen unos pocos (7 en este caso) individuos al azar y "gana" aquel que tiene peor idoneidad. La Figura 6 muestra los resultados de este experimento. En efecto, la desviación típica es algo menor. Además, el comportamiento en media es bastante mejor si lo comparamos con el de la Figura 5: el valor final obtenido cuando no se substituye a los padres (1.5) es menor que cuando se los substituye (3). Por tanto, parece necesario no substituir las rutinas padre. Otra alternativa sería la mencionada en [Racine1998], en la que antes de substituir el padre por el hijo, se re-evaluan los programas necesarios para comprobar que el cambio es beneficioso a la mayoría de ellos. Sin embargo, como los propios autores comentan, el esfuerzo computacional en este caso sería mucho mayor y posiblemente poco eficaz (el esfuerzo no compensaría los beneficios). Se deja esta alternativa para estudiarla en el futuro.

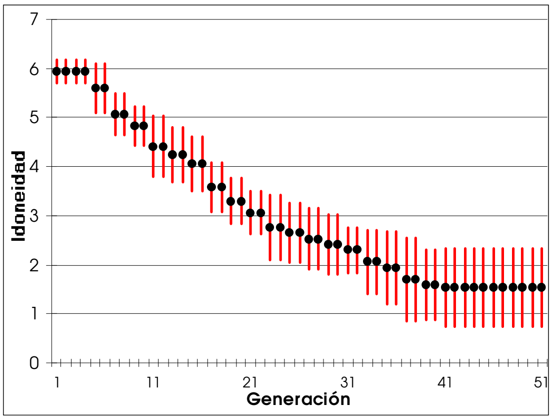

Figura 6 Idoneidad media para el problema de paridad-4 con ciclo alterno sin substitución.

5.4. Influencia del número de niveles

En este apartado se quiere comprobar si el tener una jerarquía de poblaciones profunda tiene alguna influencia en el comportamiento del sistema. En principio, puesto que se usa una profundidad de individuo pequeña, para que el tamaño del individuo expandido no crezca en demasía, la única manera que tiene un individuo de ser profundo es llamando a subrutinas de nivel inferior. Si el número de niveles se decrementa, es posible que los individuos no puedan ser lo suficiente profundos para resolver el problema. La Figura 7 muestra que esto es lo que parece ocurrir al disminuir el número de niveles de 3 a 2. El comportamiento en media de la Figura 7 es ligeramente peor que su equivalente con tres niveles que se puede ver en la Figura 6. Esto parecería indicar que los programas no pueden llegar a ser tan complejos como se necesiten. La conclusión es que será necesario elegir correctamente la relación entre el número de niveles y la profundidad máxima permitida para un individuo. Si disminuimos el número de niveles, habrá que incrementar la profundidad máxima permitida a los individuos.

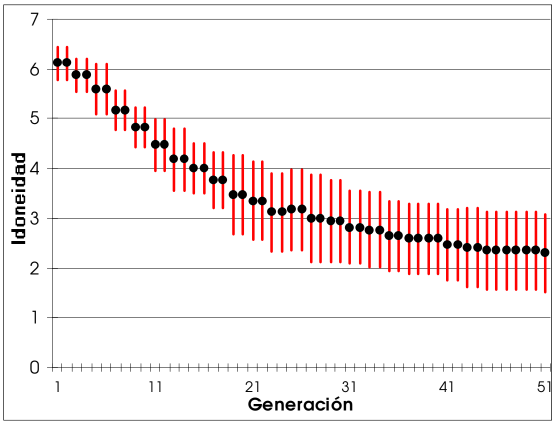

Figura 7 Idoneidad media para el problema de paridad-4 con dos niveles de poblaciones.

6. Discusión

El primer resultado es decepcionante. En ninguna de las configuraciones experimentadas se consigue superar los resultados del experimento de control (ADFs únicamente), a pesar de que DL^{GP} introduce una complejidad notable. Es perfectamente posible que, al igual que las ADFs de Koza [Koza1994], esta arquitectura sólo sea útil para problemas muy complicados. Si el problema es demasiado simple, no compensa el esfuerzo extra a realizar. En cualquier caso, queda fuera de nuestras posibilidades el experimentar estar arquitectura en problemas mucho más complicados, por lo que nos centraremos en los aspectos dinámicos de DL^{GP} en el problema de paridad 4.

El primer aspecto a observar es que DL^{GP} se beneficia de conocer los efectos sobre los programas principales que tiene modificar una subrutina. Esto se puede conseguir re-evaluando los programas afectados. Si no se hace así, existe una gran varianza entre distintas ejecuciones de la PG, lo que indica que el sistema tiende a convertirse en un paseo aleatorio. La guía proporcionada por las re-evaluaciones hace que, efectivamente, se reduzca dicha varianza. Por tanto, parece recomendable que DL^{GP} tenga un ciclo alterno. Desgraciadamente, el ciclo alterno disminuye el número de veces que se pueden aplicar los operadores genéticos si queremos mantener el mismo número de evaluaciones que en el ciclo continuo. Esto disminuye la exploración en el espacio de programas, por lo que es posible que el mejor comportamiento del sistema se obtenga en algún lugar entre el ciclo alterno y el ciclo continuo. Por ejemplo, en lugar de modificar una población de subrutinas y re-evaluar inmediatamente, se podrían modificar sólo alguna de ellas (pero no todas, pues eso sería el ciclo continuo).

Pero incluso con el ciclo alterno, los resultados en media no son demasiado buenos. Parece que las políticas de substitución son responsables de este comportamiento. En este caso, tras una operación genética, la subrutina hija substituye a alguna de las subrutinas padre. Si en lugar de eso, la subrutina hija substituye a alguna de las peores subrutinas en la población, el sistema pasa a mejorar en media de una manera notable. Además, su varianza también disminuye ligeramente. Parece que lo que esta ocurriendo es que al modificar de repente una subrutina, muchos programas principales se ven afectados. De hecho, las subrutinas que tienen descendencia se eligen de entre las mejores, y estas tienen tan alta valoración debido a que los padres que las llaman son también muy buenos. Así, en un instante, muchos programas que funcionaban muy bien pueden pasar a funcionar mal, porque se han cambiado algunas de sus subrutinas. Esto podría explicar en parte la alta varianza encontrada en las diversas ejecuciones. Parece que es así, pues al cambiar la política de substitución, la varianza disminuye.

Aunque pueda parecer obvio que la política correcta consiste en no substituir a la rutina padre, también podría darse el siguiente comportamiento: incluso aunque buena parte de las substituciones del padre por el hijo den malos resultados, algunas podrían representar mejoras. Y en ese caso, la mejora se transferiría inmediatamente a todos los programas que llamen a la subrutina padre, sin necesidad de esperar a que el proceso de cruce hiciera el trabajo de una manera más lenta y menos fiable. Además, hay que tener en cuenta que las poblaciones de programas principales y subrutinas co-evolucionan, por lo que tras varias generaciones, cada población debería producir subrutinas que realizan, más o menos la misma tarea y que están co-adaptadas a los programas principales. Por supuesto, lo mismo ocurre con los programas principales, los cuales se adaptan a aceptar subrutinas de ese tipo. Además, al mutar o cruzar rutinas similares, las rutinas hijas deberían parecerse en muchos casos a las rutinas padre, por lo que cambiar un padre por el hijo no tiene porqué producir efectos tan drásticos. Estos eran los resultados observados en otro sistema co-evolutivo [Aler1998], pero parece que son no lo suficientemente generales como para ser transferidos a DL^{GP}.

Si a pesar de este resultado se pretende seguir substituyendo al padre por el hijo, habría que seguir la sugerencia de Racine de comprobar que el efecto de la substitución es positivo, antes de realizarla definitivamente [Racine1998]. Esto llevaría a encontrar el conjunto de programas principales afectados por la substitución, re-evaluarlos y propagar hacia abajo las idoneidades. Si el hijo mejora a muchos de los programas, el cambio podría ser aceptado. Sin embargo, esto lleva a realizar muchas re-evaluaciones de programas. En efecto, si como parece, la mayor parte de las rutinas hijo no consiguen aportar alguna mejora, la mayor parte de las re-evaluaciones no tendrían ninguna utilidad. Que esta es la manera más eficiente de gastar el esfuerzo computacional habrá que determinarlo en futuros experimentos.

Por último, si DL^{GP} tiene muchos niveles, es necesario disminuir la profundidad permitida a los individuos. Esto es así porque si expandimos un programa principal (o sea, cambiamos sus llamadas a subrutinas por el código de las mismas) y existen muchos niveles, y cada el árbol representativo de cada individuo de cada nivel tiene una gran profundidad, el individuo expandido completamente podría ser gigantesco. Incluso aunque la expansión no se haga de manera explícita, la ejecución de un programa con tantos nodos puede resultar prohibitiva en tiempo. Por tanto, será necesario limitar la profundidad máxima. Sin embargo, será conveniente incrementar dicha profundidad si decrementamos el número de niveles. De otra manera, los programas no podrían alcanzar el nivel de complejidad requerido.

7. Conclusiones

Las contribuciones principales de este artículo son dos. La más importante es la evaluación empírica de DL^{GP} , un sistema co-evolutivo jerárquico. Esta idea fue propuesta en [Racine1998], pero transcurridos más de dos años, todavía no han aparecido resultados experimentales de la misma. Puesto que uno de los autores de este artículo ya estaba trabajando en ideas similares ([Aler1992]), se decidió realizar una implementación de esas ideas extendiendo un código del que ya disponíamos. La experimentación ha demostrado que no es trivial mejorar el modelo, más simple, de las ADFs [Koza1992]. Sin embargo, nuestros experimentos muestran algunas de las propiedades que debería tener un sistema similar a DL^{GP} . En particular, parece necesario que el sistema tenga cierto conocimiento de los efectos de los cambios de las poblaciones de subrutinas, puesto que estos cambios pueden afectar potencialmente a muchos programas principales. Además, es necesario ser cauteloso con la política de substitución de rutinas padre por rutinas hija, a pesar de que en principio esta política podría tener ciertos beneficios. Será necesario seguir investigando para determinar bajo qué condiciones se puede utilizar dicha política.

En segundo lugar, el presente artículo realiza una extensa revisión del estado actual sobre la evolución de subrutinas en PG.

8. Trabajos Futuros

- En primer lugar, habría que determinar en que punto de la gradación ciclo alterno-ciclo continuo es necesario comprobar los efectos de los cambios realizados a las subrutinas. Es posible que dicha comprobación pueda ser realizada después de unos pocos cambios, con lo que, con el mismo número de evaluaciones, se podrían realizar más operaciones genéticas, y por tanto llevar a cabo una exploración más completa.
- Como se comentaba en las conclusiones, habría que determinar bajo que condiciones es posible realizar la substitución de padre por hijo. Habría que determinar experimentalmente cuando merece la pena el esfuerzo computacional de comprobar que el hijo es mejor que el padre.
- Es posible que DL^{GP} sólo sea útil para problemas realmente complicados, como ocurre con las ADF de Koza. Al igual que este último, habría que considerar un problema parametrizado de manera que al incrementar el parámetro se incremente la complejidad del problema. Una vez echo esto, se podría determinar si hay algún valor del parámetro tras el cual DL^{GP} es un método que proporciona beneficios. Por supuesto, el esfuerzo computacional necesario para la experimentación sería bastante grande.
- Introducir la mutación sugerida en [Racine1998], en la que se cambia una llamada a subrutina por otra llamada a una subrutina de la misma población que la primera. El efecto de esta mutación puede conseguirse por medio de varias operaciones genéticas estándar, pero sería interesante ver si la nueva mutación mejora el comportamiento de manera cualitativa.

Referencias

[Ahluwalia1998] Manu Ahluwalia and Larry Bull. Conocimiento-evolving Functions in Genetic Programming: Dynamic ADF Creation Using GLiB. Evolutionary Programming VII 1998. Proceedings of the Evolutionary Programming VII International Conference 7th International Conference EP'98. San Diego (USA). 1998.

[Ahluwalia1997] Manu Ahluwalia and Larry Bell and Terence C. Fogarty. Co-evolving Functions in Genetic Programming: A Comparison in ADF Selection Strategies. Genetic Programming 1997: Proceedings of the Genetic Programming Conference GP'97. 1997.

[Aler1998] Ricardo Aler. Immediate transfer of global improvements to all individuals in a population compared to Automatically Defined Functions for the EVEN-5,6-PARITY problems. Proceedings of the EuroGP'98 Workshop. Paris. 1998

[Angeline1992] Angeline, P. J. and Pollack, J. B. (1992) The Evolutionary Induction of Subroutines, In The Proceedings of the 14th Annual Conference of the Cognitive Science Society. Hillsdale (USA). 1992.

[Angeline1993] Angeline, P. J. and Pollack, J. B. (1993) Evolutionary Module Acquisition, In Proceedings of the Second Annual Conference on Evolutionary Programming. 1993.

[Angeline1994] Angeline, P. J. and Pollack, J. B. (1994) Coevolving High-Level Representations, In Artificial Life III, C. Langton (ed.), Addison-Wesley: Reading MA, pp. 55-71.

[Koza1992] Koza, John R. 1992. Genetic Programming: On the Programming of Computers by Means of Natural Selection. The MIT Press.

[Koza1994] Koza, John R. 1994. Genetic Programming II: Automatic Discovery of Reusable Programs. The MIT Press.

[Rosca1994a] Justinian P. Rosca, Dana H. Ballard. "Hierarchical Self-Organization in Genetic Programming", Proceedings of the Eleventh International Conference on Machine Learning, Morgan Kaufman Publishers, 1994.

[Rosca1994b] Justinian P. Rosca, Dana H. Ballard. "Hierarchical Self-Organization in Genetic Programming", Proceedings of the Eleventh International Conference on Machine Learning, Morgan Kaufman Publishers, 1994.

[Rosca1995] Justinian P. Rosca. "Towards automatic discovery of building blocks in genetic programming." Proc. of the AAAI Symposium on Genetic Programming, Technical Report AAAI Press, 1995.

[Rosca1996a] Justinian P. Rosca, Dana H. Ballard. "Discovery of Subroutines in Genetic Programming." In Advances in Genetic Programming II, Edited by P. Angeline and K. Kinnear Jr. MIT Press, 1996.

[Rosca1996b] Justinian P. Rosca, Dana H. Ballard. "Evolution-based discovery of hierarchical behaviors." Proc. of the Thirteenth National Conference on Artificial Intelligence (AAAI-96), AAAI / The MIT Press, 1996.

[Racine1998] Alain Racine and Marc Schoenauer and Philippe Dague. A dynamic Lattice to Evolve Hierarchically Shared Subroutines: DL'GP Proceedings of the First European Workshop on Genetic Programming, LNCS, Vol. 1391, pp. 220-232, Springer-Verlag, 14-15 April 1998.

[Spector1995] Spector, L. 1995. Evolving Control Structures with Automatically Defined Macros. *Working Notes of the AAAI Fall Symposium on Genetic Programming*. The American Association for Artificial Intelligence. pp. 99-105

[Teller1996] Teller, A. and Veloso M. SYMBOLIC VISUAL LEARNING. 1996. PADO: A new Learning Architecture for Object Recognition. Oxford University Presss. pp 81-116