RISI 10(1), 65 - 73 (2013)
REVISTA DE INVESTIGACIÓN DE SISTEMAS E INFORMÁTICA
FACULTAD DE INGENIERÍA DE SISTEMAS E INFORMÁTICA
UNIVERSIDAD NACIONAL MAYOR DE SAN MARCOS

ISSN 1815-0268 (versión impresa)
ISSN 1816-3823 (versión electrónica)

Revisión de las técnicas existentes en Programación Genética para evolucionar las subrutinas

Revision of the existing techniques in genetic programming to evolve the subroutines

Ana María Huayna Dueñas

Universidad Nacional Mayor de San Marcos
Facultad de Ingeniería de Sistemas e Informática

ahuaynad@gmail.com

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. La 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 DLGP. Este trabajo tiene como propósito desarrollar el Estado de Arte de las variadas técnicas existentes en PG para evolucionar subrutinas.

Palabras claves: Programacion genetica, definicion automatica de funciones

ABSTRACT

The genetic programming (GP) is a machine learning technique based on the evolution of computer programs by a genetic algorithm. The version, called ADF (automatic function definition) to reuse a subroutine several times within the same individual. However, there is the possibility that the same subroutine may be reused by several individuals from the same population. There are several systems that, in principle, allow subroutines discover valid for many individuals in a population. One of the most advanced is the DLGP dynamic network. This work aims to develop the State of Art of PG varied techniques to evolve existing subroutines.

Keywords: Genetic algorithm, evolution of subroutines, genetic programming.

1. INTRODUCCIÓN

La Programación Genética (PG) es un método de búsqueda en el espacio de programas de ordenador inventado por John Koza [1]. Esta metodología está inspirada en las teorías de la evolución de Darwin y la genética de Mendel y se clasifica dentro del área de aprendizaje en la inteligencia artificial. Este método pertenece al conjunto genérico de métodos de computación evolutiva, uno de cuyos más conocidos ejemplos son los algoritmos genéticos de Holland [2]. El paradigma de búsqueda clásico que se parece más a la PG es el de "beam search" de Tackett [3], aunque posee ciertas peculiaridades propias. Como cualquier algoritmo de búsqueda, la PG busca en un espacio de posibles soluciones representadas de determinada manera, dispone de operadores de búsqueda y de una función heurística que la guía.

Al igual que "beam search", la PG mantiene una población finita de posibles buenas soluciones o candidatos a solución (denominadas individuos). Dichos candidatos a solución suelen ser programas de ordenador funcionales codificados en forma de árboles. Sin embargo, se ha usado PG para evolucionar otros tipos de estructuras (árboles de decisión, conjuntos de reglas prolog, grafos ,secuencias de instrucciones , etc).

Aunque la investigación en PG ha tomado muchos rumbos, una de sus ideas directrices es que la PG pueda utilizar los mismos recursos que cualquier programador. En particular, aquellos programadores que utilizan subrutinas en sus códigos que puedan ser utilizadas en diferentes lugares del mismo programa. Koza encontró que aprendiendo las subrutinas adecuadas permitía a la PG resolver los problemas de manera más rápida y encontrando programas más cortos [4]. A la técnica anterior se le 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 lo tanto, parece buena idea separar la evolución de los programas principales de la evolución de las subrutinas. Esta fue aplicada de diversas maneras en [5] y [6] con buenos resultados.

En 1998, [7] propuso un método similar llamado DLGP Dynamic Lattice Genetic Programming: Red Dinámica DLGP, 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 DLGP tenga que funcionar necesariamente o de que lo haga de manera eficiente.

En tal sentido, el presente artículo hace una revisión bibliográfica del uso de las técnicas en PG para evolucionar las subrutinas, y está organizado en secciones. En la sección 2 se presenta toda la fundamentación teórica de PG, en la sección 3 la evolución de las subrutinas en donde son presentadas las diversas técnicas existentes en Programación Genética para evolucionar las subrutinas, y finalmente las conclusiones en la sección 4.

2. FUNDAMENTACIÓN TEÓRICA

2.1. Definición

La programación genética es una técnica de aprendizaje automático que se basa en la evolución de programas de ordenador mediante un algoritmo genético [1].

2.2. Orígenes

La PG no es idea nueva pero en la actualidad sigue evolucionando surgiendo nuevas variantes y aplicaciones [10].

Aquí se presenta un breve resumen histórico:

1. 1948 – Turing – propone proceso de búsqueda evolutiva.
2. 1964 – Rechenberg – Estrategias Evolutivas (EE).
3. 1965 – Fogel – Programación Evolutiva (PE).
4. 1975 – Holland – Algoritmos Genéticos (AG).
5. 1992 – John R. Koza – Programación Genética (PG).

2.3. Características

1. Consumen muchos recursos (computación paralela ó distribuida)

2. El usuario debe saber cuales funciones primitivas y variables son más apropiadas para un determinado problema. Si hay funciones y/o variables superfluas aumenta mucho el tiempo para encontrar la solución. Y si faltan funciones y/o variables, no encontrará la solución.
3. Afronta problemas no computables en tiempo polinómico (NP).
4. Son paralelizables y distribuibles.
5. No se estanca en máximos locales.
6. Necesitan de un lenguaje dinámico ó la escritura de un pequeño compilador.
7. Si la función de aptitud cambia suavemente en el tiempo, el GP puede encontrar la solución óptima para cada momento.
8. Puede diseñar la estructura de un programa :
   - ADF -- Funciones automáticamente definidas por medio de operadores de encapsulamiento, borrado de una función, adición de un argumento, borrado de un argumento.
   - Generación automática de bucles (ADL) y recursiones (ADR).
   - Variables internas automáticamente definidas (arrays, colas, listas,...)
   - Constantes automáticamente generadas [10], [11].

2.4. Aplicaciones

1. Síntesis automática de circuitos eléctricos analógicos
2. Síntesis automática de controladores
3. Problemas en biología molecular computacional, incluyendo rutas metabólicas y redes genéticas.
4. Entrenamiento de redes neuronales celulares
5. Análisis y predicción de la estructura de las proteínas
6. Autómatas Celulares
7. Estrategias en Robots (Hormigas artificiales)
8. Sistemas Multiagentes
9. Investigación Operativa – Problemas de optimización
10. Demostraciones de identidades matemáticas.
11. Áreas de Diseño
12. Clasificación de Datos (Clustering, Data Mining..)
13. Uso de PG como una "Máquina Automática de Invenciones" para crear nuevas invenciones útiles patentables [10]. [12].

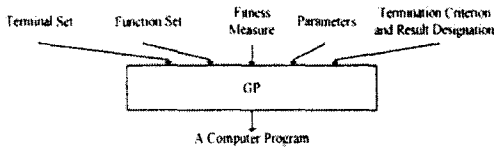

Figura N.º 1. Componentes de una solución PG.

2.5. Componentes de una Solución PG

1. Conjunto de símbolos terminales.
2. Conjunto de funciones primitivas.
3. Medida de aptitud.
4. Parámetros de control.
5. Criterio de terminación.

Se genera un programa que satisface los parámetros de entrada [10], [14].

2.6. Representación y creación de programas

1. Clásicamente los programas se representan en árboles, que presentan una correspondencia directa con la notación prefija ó polaca.
2. La familia de lenguajes de Programación que se adapta a esta representación son la familia LISP (Lisp-like), como por ejemplo: Lisp o Scheme. Así como para el siguiente programa en LISP [10], [15].

(+ (* x y) (IFLTE x y 3.2 0.4)) (* 6.2 (+ x (- y x)))

Se tienen la siguiente representación en forma de árbol.

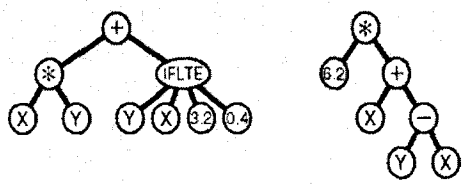

Figura N.º 2. Representación en forma de árbol.

2.7. Algoritmo

Se considera a la PG como un paradigma de búsqueda bastante genérico que, de acuerdo a la discusión an-

terior, puede representarse de forma abstracta por la Figura N.° 3. Dicho gráfico muestra el diagrama de flujo de un sistema genérico de PG en el que se selecciona a los individuos de acuerdo a la función de evaluación, los cuales son modificados por operadores genéticos que también han sido seleccionados de acuerdo a algún criterio, y finalmente son evaluados con un subconjunto de casos de prueba previamente seleccionado. Finalmente, los nuevos individuos son introducidos en la población [9], [10], [15],[18].

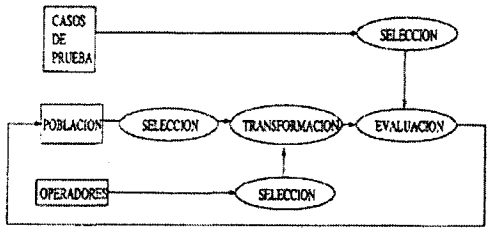

Figura N.° 3. Esquema genérico de la programación genética.

A continuación se describe el algoritmo de la variante de la PG conocido como modelo generacional [11].

1. Crear aleatoriamente una población de individuos.
2. Seleccionar un conjunto de casos de prueba y evaluar la población con ellos.
3. Repetir hasta que algún criterio de terminación sea satisfecho:

a. Repetir hasta que se complete una nueva población:

i. Seleccionar un operador genético del conjunto de operadores (normalmente reproducción, cruce y mutación). Cada operador tiene una probabilidad de ser seleccionado.

ii. Seleccionar tantos individuos como requiera el operador seleccionado (por ejemplo, el cruce requiere dos). La selección es estocástica. Mejores individuos deberían tener una probabilidad más alta de ser seleccionados.

iii. Aplicar el operador genético, obtener un nuevo individuo y añadirlo a la nueva población.

b. Seleccionar un conjunto de casos de prueba (normalmente se seleccionan todos ellos) y evaluar a la nueva población con ellos.

c. Reemplazar la población antigua con la nueva.

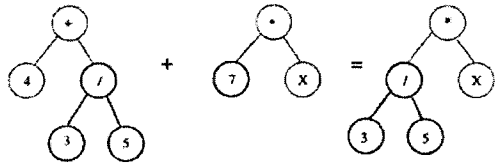

Figura N.° 4. Componentes de una solución PG.

2.8. Operadores Genéticos

Operador de Reproducción

Simplemente crea un individuo exactamente igual al que se le pasa como argumento utilizando funciones y terminales.

Operador de Cruce (Crossover)

Escoge dos individuos, selecciona aleatoriamente un nodo en cada uno de los individuos progenitores (representados como árboles) e intercambia los dos subárboles correspondientes[1][16].

Hay que hacer una salvedad: en los campos derivados de los algoritmos genéticos se le da gran importancia al operador de cruce, puesto que es capaz de combinar buenos fragmentos de dos candidatos a solución, lo que puede llevar a obtener mejores soluciones que cualquiera de los progenitores [3]. A la hipótesis de que un algoritmo genético funciona a base de ir combinando progresivamente buenos fragmentos se la denomina la hipótesis de los bloques de construcción [2]. Sin embargo, estudios recientes muestran que en la PG estandard, este efecto podría no conseguirse [19]. Las razones principales podrían ser dos. Primero, si el sistema ha conseguido encontrar un buen fragmento de programa que es grande, el operador de cruce tenderá a romperlo. En segundo lugar, un fragmento de programa tendrá efectos diversos según el lugar del otro programa donde se lo inserte. Es decir, el operador de cruce es contextual; es decir, a diferencia de los algoritmos genéticos en los que los elementos que se intercambian tienen su significado fijado por su posición, los subárboles de la PG pueden tener significados distintos según el contexto.

1 A diferencia del ejemplo de cruce mostrado en la figura, el operador de cruce estándar genera dos árboles hijos a partir de dos árboles padres.

Operador de Mutación

Selecciona un nodo en el árbol progenitor, cortar el subárbol que depende de ese nodo, y lo substituye por un subárbol generado aleatoriamente.

Otros operadores

1. Operador de Encapsulación [6]

El operador de Encapsulamiento hace la creación de una función. automáticamente a través del ADF.

2. Operador de Inversión [9]

Escoge un individuo de la población, selecciona dos puntos de fractura (análogos a los puntos de cruce en las representaciones genéticas en cadenas utilizadas por los AG) en el individuo, generando un nuevo individuo conmutando los símbolos delimitados y solapando las sublistas con la lista principal.

3. Operador de Alteracion estructural [9]

Se aplica una operación a cierta parte de un programa para posteriormente copiarlo en la nueva población

4. Operador de Comprensión [6]

Dicho operador toma un subárbol de un individuo de manera aleatoria, lo parametriza y lo guarda en una librería. Selecciona un nodo en el árbol progenitor, cortar el subárbol que pende de ese nodo, y lo substituye por un subárbol generado aleatoriamente.

2.9. Medida de Aptitud - Fitness

La PG está guiada por una función heurística denominada función de adecuación o de "fitness". Dicha función evalúa la bondad de cada uno de los individuos de la población. Normalmente, la función de adecuación devuelve un valor numérico o también un vector de valores, si se está haciendo optimización multi-objetivo. Dicho valor sirve para determinar la probabilidad de seleccionar cada individuo de la población, que normalmente es proporcional al valor devuelto por la función de adecuación. Una vez seleccionado, dicho individuo (o individuos, en el caso de que el operador genético sea el cruce) será transformado por alguno de los operadores genéticos. La función de adecuación puede evaluar tanto aspectos dinámicos del individuo (tras haberlo ejecutado) como estáticos. Aspectos dinámicos son, por ejemplo, el error cometido en la salida del programa, el tiempo que ha tardado, etc. Aspectos estáticos podrían ser el tamaño en nodos del individuo, la cantidad de determinadas funciones que contiene su código, etc.

La población de individuos, los operadores genéticos, la función de adecuación y los casos de prueba, son los ingredientes fundamentales de cualquier sistema de PG. Se ha hecho hincapié en que, aunque tradicionalmente se empleen determinadas estructuras y determinados operadores genéticos, no existen realmente razones poderosas para restringirse a las elecciones estándar, y en cambio sí existen algunas razones para examinar otras posibilidades [9], [10],[15], [18].

2.10. Diferencias de AG y PG

1. Los AG crean cadenas de cromosomas que codifican (representan) la solución buscada
2. La PG crea Programas (código)
3. Es decir: en PG se transforman Programas que buscan la solución, en vez de la representación de la solución [18].

3. EVALUACIÓN DE SUBRUTINAS

3.1. 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 [4]. 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 rutina 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.

3.2. Iteracciones Definidas Automáticamente Loops Definidas Automáticamente, Recursiones Definidas Automáticamente y Almacenamiento Definidas Automáticamente

Iteracciones Definidas Automáticamente (ADIs), Loops Definidas Automáticamente (ADLs) y Recursiones Definidas Automáticamente (ADRs) proveen significado (en adición a ADFs) para reutilizar código.

Almacenamiento definidas Automáticamente (ADSs) significa reutiizar el resultado del código ejecutable.

Iteracciones Definidas Automáticamente (ADIs), Loops Definidas Automáticamente (ADLs), Recursiones Definidas Automáticamente (ADRs) y Almacenamiento definidas Automáticamente son descritos por John Koza [8], [9].

3.3. Constructor de Librerías Genéticas (GLiB)

Otro enfoque para la explotación de la modularidad es el de Angeline [9], [19], donde 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 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 ésta disminuye.

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.

3.4. Representaciones Adaptativas (ARL)

Rosca y Ballard describen el método de representaciones adaptativas mediante aprendizaje para la inducción de subrutinas. 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 [21].

3.5. Funciones evolutivas (EDF)

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 continúen 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.

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.

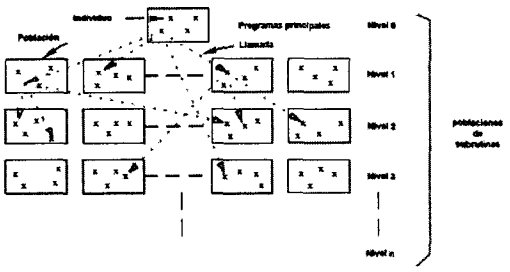

Figura N.º 5. Topología en red. Existen una población de programas principales (nivel 0) que pueden llamar a subrutinas de niveles inferiores (novel 1,2, ..n)

Ahluwalia [5] estudió 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).

En un trabajo posterior, Ahluwalia combina las ideas de GliB y la co-evolución de rutinas a la que denomina EDF (Evolution Defined Functions) [5], [9]. La cual 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 continúan evolucionando por separado. Dichas subrutinas pueden ser llamadas por los programas principales. En el 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 se crea unas pocas poblaciones de subrutinas.

En un trabajo presentado por Aler [6] 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.

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.

3.6. Red dinámica DLGP (Dynamic Lattice Genetic Programming)

Racine [7] propone un sistema para la co-evolución de subrutinas y programas a la que denomina red dinámica DLGP. Al igual que las funciones co-evolutivas de Ahluwalia, Racine separa las poblaciones de programas y subrutinas y jerarquiza éstas 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, éste 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.

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. Además del operador de cruce estándar y varios operadores de mutación, DLGP 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.

DLGP 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.

Tras la inicialización aleatoria de cada una de las poblaciones, comienza el llamado ciclo evolutivo, que funciona de arriba a abajo en la jerarquía. Así, DLGP 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.

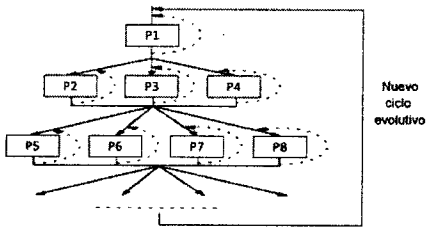

Figura N.° 6. Ciclo co-evolutivo de la topología en red. Se realiza un ciclo evolutivo de la población de programas principales, seguido de ciclos evolutivos en las poblaciones de subrutinas.

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.

3.7. Encapsulación de Subárboles

Cada uno de los métodos anteriores se ocupa de producir módulos y funciones en paralelo con los individuos que las van a utilizar. Un enfoque diferente fue tomada por [20] con su Método de encapsulación de subárbol. En lugar de producir módulos sobre la marcha, se completan corridas para módulos útiles que luego son utilizados posteriormente para aumentan el conjunto de terminales de otra corrida.

Para lograr esto, una base de datos de todos los subárboles generadas durante una corrida se mantiene. En ésta base de datos se mantiene un conteo de la utilización de cada árbol, que luego se utiliza para decidir qué subárboles debe añadirse al conjunto terminal; los más utilizados con frecuencia son encapsulados como átomos.

Se ha demostrado que su método estándar GP superó tanto en términos de la velocidad a buenas soluciones y a la calidad general de la solución generada.

Tal vez sorprendentemente, también mostraron que la elección aleatoria de los subárboles les dio mejor rendimiento que el enfoque basado en el uso.

3.8. Corridas Secuenciales

Ha habido varios intentos para pasar información desde una ejecución a otra, aunque éstos han estado con el fin de resolver un solo problema, en lugar de tratar a un sistema de que puede ser aplicado sucesivamente a problemas más difíciles. La más conocida de estas técnicas es simplemente inicializar una población con el mejor individuo de la ejecución anterior. Esto se ha utilizado con éxito por muchos investigadores, como algunos han encontrado que esto puede conducir a una convergencia prematura en ese individuo [20], [21].

4. CONCLUSIÓN

Dentro de las contribuciones principales de este artículo es la extensa revisión del estado actual sobre la evolución de subrutinas en PG. En ADF un método para la descomposición de problemas donde los ADFs son funciones locales que los individuos pueden llamar que también están sujetas al tiempo de evolución. Las Subrutinas en GLiB se conocen como módulos que son creados dinámicamente durante una ejecución, a través del uso del operador de compresión. Este se aplica probabilísticamente a un subárbol de un individuo, y lo comprime en un módulo, que es un nodo con la misma funcionalidad. Esto tiene un doble efecto de aumentar la expresividad del lenguaje y la disminución del tamaño promedio de los individuos en la población. A diferencia del ADF, sin embargo, debido al proceso de extracción, los módulos GLiB no puede volver a utilizar argumentos en los módulos; por lo que impide su desempeño. La otra diferencia del GliB con el ARL es que selecciona aquellos individuos basados en el fitness que son mejores que sus padres. ARL ha demostrado un buen rendimiento con respecto a los otros métodos, pero ha sido criticado de exigir muchos parámetros de control, y aún no ha encontrado una amplia aceptación en el campo. En cambio con el EDF solo se crea unas pocas poblaciones de subrutinas a ser guardadas; haciendo esa diferencia con Glib que podría guardar cientos de ellas. En particular, DLGP un sistema co-evolutivo jerárquico en donde 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. Es posible que DLGP sólo sea útil para problemas realmente complicados, como ocurre con las ADF de Koza.

5. REFERENCIAS BIBLIOGRÁFICAS

[1] J.R. Koza, "Genetic Programming: On the programming of Computers by Means of Natural Selection", eds. MIT Press, Cambridge, MA,USA,1992. (Estilo Libro)

[2] J.H. Holland ,"Adaptation in Natural and Artificial System", The University of Michigan Press,1975

[3] W.A. Tackett, "Recombination, Selection y The Genetic Construction of Computer Programs",PhD Tesis, Dept. of Electrical Eng Systems. Southern Univ.,California., 1994 (Tesis)

[4] J.R. Koza, "Genetic Programming: II : Automatic Discovery of Reusable Programs ," eds. MIT Press, Cambridge, MA,USA,1994

[5] M. Ahluwalia, L. Bull y T. C. Fogarty,"Co-evolving Functions in Genetic Programming: A Comparison in ADF Selection Strategies," Proc. of the Genetic Programming Conference GP'97, 1997. (Conference proceedings)

[6] R.Aler, " Immediate transfer of global improvements to all individuals in a population compared to Automatically Defined Functions for the EVEN-5,6-PARITY problems," Proc. of the EuroGP'98 Workshop. Paris. 1998. (Conference proceedings)

[7] R.Aler, F.Blázquez, D. Camacho, "Experimentación en Programación Genética Multinivel," Inteligencia Artificial: Revista Iberoamericana de Inteligencia Artificial, 2001, vol. 5, n. 13, p. 10-22 , URI: http://hdl.handle.net/10016/5292

[8] J.R. Koza, Bennet, Andre y Keane, "Genetic Programming: III : Darwinian Invention and Problem Solving ",eds. MIT Press, Cambridge, MA,USA ,pp. 87-166, 1999. (Estilo Libro)

[9] J.R. Koza, "Genetic Programming IV: Routine Human-Competitive Machine Intelligence," pp 29-42, eds. MIT Press, Cambridge, MA,USA,2003 . (Estilo Libro)

[10] A. Rios, "Introducción a la Programación Genética," http://code.google.com/p/rubygp/wiki/IntroduccionProgramacionGenetica.html. 2008

[11] C. A. Coello, "Introducción a la Computación Evolutiva," Departamento de Ingeniería Eléctrica, Sección de Computación, Av. Instituto Politécnico Nacional No. 2508 Col. San Pedro Zacatenco, México, D.F pp 62-64,. Enero, 2004

[12] R..Poli, W..B. .Langdon, N..F..McPhee,, .R.Koza, "A field Guide to Genetic Programming," pp 37-109, Marzo 2008

[13] S. Luke, "Issues in Scaling Genetic Programming: Breeding Strategies, Tree Generation, and Code Bloat.," PH Tesis , Facultad de la Escuela de Graduados, Universidad de Maryland, 2000 (Tesis)

[13] M. Hernández , " Programación Genética, " Instituto Universitario de Sistemas Inteligentes y Aplicaciones Numéricas en Ingeniería, 2008

[15] J. Baier, "Programación Genética," PUC, 2005

[16] R. Aler, "Programación Genética de Heurísticas para Planificación," PhD Tesis Facultad de Informática, Universidad Politécnica de Madrid, España, 1999 (Tesis)

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

[18] L. Araujo, C. Cervigón, "Algoritmos Evolutivos: Un Enfoque Práctico," ed. AlfaOmega, México, pp. 112-123, 2009.

[19] P. J. Angeline, "Subtree crossover: Building block engine or macromutation, " In J. R. Koza, et al., ed. Programación Genética 1997: Proc. Second Annual Conference, pages 9–17, Stanford University, CA, USA, 13-16 July 1997.

[20] S.C. Roberts, D. Howard, John R. Koza, "Evolving modules in genetic programming by subtree encapsulation". In Julian F. Miller, Marco Tomassini, Pier Luca Lanzi, Conor Ryan, Andrea G. B. Tettamanzi, and William B. Langdon, editors, Genetic Programming, Proc. EuroGP'2001, volume 2038 of LNCS, pages 160–175, Lake Como, Italy, 18-20 April 2001.

[21] M. Keijzer1, C. Ryan2, M. Cattolico, "Run Transferable Libraries — Learning Functional Bias in Problem Domains" , 2003