Ciencia Investigación Academia Desarrollo

Un enfoque para la evolución de programas en Lenguaje Estructurado mediante Programación Genética

Sergio A. Rojas[1]
Julián Olarte[2]
David Suárez[3]

RESUMEN

La Programación Genética (PG) es una técnica de computación evolutiva que permite el desarrollo de programas de computador de manera automática, simulando el proceso evolutivo natural que dio origen a la diversidad de especies orgánicas. Esta técnica ha sido ampliamente utilizada para diferentes aplicaciones, la mayoría de ellas empleando lenguajes especializados de procesamiento de listas (LISP). Nuestro objetivo es desarrollar experimentos de computación evolutiva, pero empleando lenguajes estructurados de más alto nivel y de mayor difusión. Con este fin, inicialmente se debe contar con un entorno computacional que permita simular el proceso evolutivo. En este artículo se presenta una primera aproximación a una plataforma de PG basada en lenguaje C.

Palabras clave: programación genética, computación evolutiva, lenguajes estructurados.

A Genetic Programming Approach for Structured Language Programs Evolution

ABSTRACT

Genetic Programming (GP) is an evolutionary computation technique for discovering automatically computer programs, resembling the evolutionary process that follow different species in the Earth. GP have been actively used in a number of applications, most of them using list-processing languages like LISP. Our goal is to run some evolutionary computation experiments using not specialized languages but higher level and more widely used structured languages. For this purpose, a proper computational environment must be developed, so in this paper we present an initial approach for a GP C-based framework.

Key words: genetic programming, evolutionary computation, structured language.

I. INTRODUCCIÓN

De acuerdo con la teoría de Darwin [1], el proceso evolutivo que han seguido las especies naturales en la Tierra, ha llevado al desarrollo de una desbordante complejidad en sus sistemas orgánicos, manifestada en la gran diversidad de mecanismos de defensa, protección, colaboración y reproducción. Asombrosamente, toda esta complejidad se ha obtenido gracias a una lucha por la supervivencia de las especies llevada a cabo durante millones de generaciones: cada especie intentando aprovechar de la mejor manera los recursos escasos que tiene disponible en su ambiente, esto es, adaptándose. Los mecanismos que se utilizan para buscar una mejor adaptación ocurren a nivel genético, al recombinar el genoma de los padres (reproducción) o cuando casualmente se introducen pequeños cambios en la configuración de los genes (mutaciones). Estos cambios se verán reflejados posteriormente en el fenotipo del nuevo individuo. El mismo entorno es el encargado de evaluar qué tan buenos resultan los cambios genéticos, favoreciendo a los mejores (la selección natural califica la aptitud de los individuos).

Recientemente, la computación evolutiva ha aplicado este mismo enfoque para la solución de problemas en diversas áreas de la ingeniería. La idea es que si la evolución natural ha obtenido sistemas tremendamente complejos (piénsese en una máquina tan asombrosa como el cerebro humano), un proceso artificial quizás conduzca al desarrollo automático de programas de computador aplicables a problemas de difícil resolución. Los individuos en este caso corresponden a estructuras computacionales cuyo genoma tiene implícito una representación de los parámetros del problema, y las operaciones genéticas de reproducción, cruce y mutación se realizan sobre bits e instrucciones de computador. La idoneidad de cada programa se determina mediante una función de aptitud definida para el problema particular, y el resto es cuestión de evolucionar durante muchas generaciones a una población de individuos.

Los individuos en este caso corresponden a estructuras computacionales cuyo genoma tiene implícito una representación de los parámetros del problema,

El enfoque ha sido aplicado exitosamente bajo los nombres de Algoritmos Genéticos (AG) [2] y Programación Genética (PG) [3]. En el caso de los AG, las operaciones genéticas son relativamente simples de realizar, puesto que el genoma se representa como una secuencia de bits (la implementación se puede hacer en cualquier lenguaje que maneje cadenas de caracteres, como C, Pascal, Java, VisualBasic). En el caso de la PG, el mecanismo no es directo, ya que el genoma se representa como un árbol de instrucciones (la mayoría de implementaciones se realizan en lenguajes orientados a listas, proclives a ser representados como árboles, como el LISP).

[1] Director del Grupo de Interés en Adaptación, Computación y Mente (ACME-UD), investigador principal del grupo LAMIC.
[2] Miembro grupo de investigación Interés en Adaptación, Computación & MEnte (ACME-UD).
[3] Miembro grupo de investigación Interés en Adaptación, Computación & MEnte (ACME-UD).

Artículo recibido en Abril de 2003
Aceptado en Junio 2003

Debido a que los lenguajes estructurados usan la notación prefija para representar invocaciones a funciones, sería conveniente utilizarla también para representar todo tipo de instrucción secuencial.

Ya que los lenguajes estructurados son ampliamente utilizados y preferidos en el ámbito ingenieril, sería interesante poder utilizarlos para desarrollar experimentos de Programación Genética. Para ello, inicialmente debería contarse con una plataforma o marco de trabajo que permita manipular los programas estructurados y configurar procesos evolutivos artificiales con ellos. En este artículo se describe una primera aproximación a este objetivo. En la siguiente sección se amplían los conceptos sobre PG. En la sección III se describe el enfoque propuesto para realizar PG en lenguajes estructurados. En la sección IV se presenta la plataforma desarrollada. En la sección V aparecen algunos resultados preliminares. Para finalizar se discuten ideas acerca del trabajo que se puede continuar en esta dirección.

II. EVOLUCIÓN DE PROGRAMAS

En la PG se tienen estructuras computacionales o programas de computador en continua adaptación, a los cuales se aplican operadores genéticos de cruce o mutación. Para abordar un problema de PG se debe tener una población inicial de programas de computador generada aleatoriamente, que es evolucionada repetidamente hasta satisfacer el criterio de finalización. Dentro de estas iteraciones, primero se evalúa cada una de las criaturas de acuerdo con su comportamiento para asignarles un valor de aptitud, para posteriormente crear una nueva población aplicando dos criterios de reproducción determinados por la aptitud: copia de una criatura (ocasionalmente mutada), y recombinación del genoma de dos criaturas existentes para crear otras dos nuevas criaturas.

Para ilustrar como actúa la recombinación, considérese los siguientes programas en LISP [4]:

```
(÷ (- v (* 2 a ) b ) ( * 2 a ) )

(÷(- v( - (* b b)(*(* 2 2)(* a c))) b)(*(+ 2 2)(* a c))
```

En la Figura 1, se muestran estos dos programas en forma de árbol.

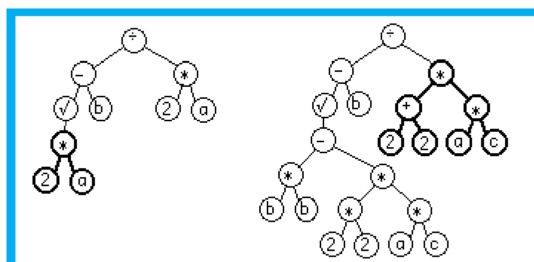

Figura 1. Dos programas LISP en forma de árbol.

La operación cruce (o recombinación) crea dos hijos intercambiando los subárboles resaltados en negrilla entre los dos padres. Los hijos resultantes del cruce se muestran en la Figura 2 (el árbol de la derecha corresponde a la conocida expresión para obtener las raíces de una ecuación cuadrática). La operación de mutación, actua de manera similar, introduciendo un cambio aleatorio en alguna de las ramas del árbol. De esta manera se crean nuevos programas de computadora que al estar sometidos a un proceso de "selección artificial", pueden resultar en formas aptas y coherentes para resolver algún problema.

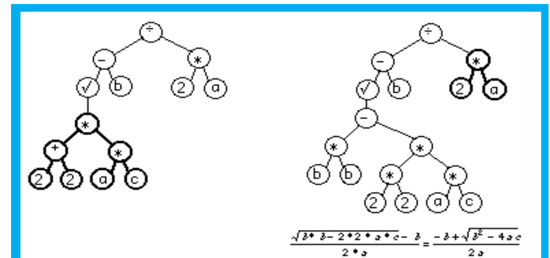

Figura 2. Programas originados mediante el cruce de los programas de la Fig. 1.

III. PG CON LENGUAJES ESTRUCTURADOS

En los lenguajes estructurados, se manejan estructuras computacionales más complejas que las trabajadas en lenguajes de programación basados en listas. Además de tener invocaciones a funciones con argumentos, se deben tener en cuenta las instrucciones condicionales, los bucles y las instrucciones de agrupamiento. Estas estructuras requieren de un manejo especializado para poder aplicar de una manera eficiente la PG en un lenguaje específico (como el lenguaje C).

3.1 CONSIDERACIONES SOBRE LENGUAJES ESTRUCTURADOS

Fortran, Pascal, C y otros lenguajes similares utilizan notación infija (donde los operadores se encuentran entre los operandos, por ejemplo, $A + B$), mientras que los lenguajes basados en listas emplean notación prefija (en la que el operador precede a su operando, por ejemplo, $+ A B$). Las sentencias escritas en infija permiten una fácil conversión a una estructura arbórea, la cual se genera mediante una análisis sintáctico [5], donde se toma primero el operador y se convierte en la raíz y luego los operandos en sus terminales.

El uso de PG en lenguajes estructurados requiere algunas restricciones en el manejo del lenguaje sobre el cual se quiere evolucionar. Una de estas restricciones es el uso de instrucciones con notación infija (como asignación de valores a variables u operaciones aritméticas). Debido a que los lenguajes estructurados usan la notación prefija para representar invocaciones a funciones, sería conveniente utilizarla también para representar todo tipo de instrucción secuencial. Para ello se debe generar una representación prefija de cada una de las posibles expresiones infijas en cada lenguaje. Por ejemplo, la ex-

presión “$A + B$” puede ser reemplazada por “*add (A , B)*”, donde “*add*” representa al operador matemático de la suma y “A , B” los elementos a los cuales se les va a operar. Otra restricción son las reducciones o facilidades en la sintaxis, permitidas por cada lenguaje. Estas son por ejemplo, múltiples asignaciones en una misma sentencia, la asignación condicional fuera de un *IF-ELSE* y las reducciones de ámbito cuando existe solo una instrucción, entre otras, dependiendo del lenguaje utilizado.

Otras instrucciones que se trabajan en los lenguajes estructurados son las de agrupamiento. Estas agrupan un conjunto de instrucciones secuenciales que se quiere repetir un número específico de veces o que se ejecuten sólo si una condición particular es verdadera. Este conjunto puede incluir instrucciones secuenciales, de agrupamiento e invocaciones a funciones. Un ejemplo de esta jerarquía de agrupamiento con una instrucción repetitiva, es el siguiente programa en lenguaje C:

```
void main(){
      int a=10;
      while(a>2){
              a—;
      }
}
```

En ella se observa que la función principal contiene dos instrucciones y la segunda instrucción, a su vez, tiene otras dos instrucciones, la primera es la condición y la segunda la que se repetirá mientras la primera sea verdadera.

Los otros tipos de instrucciones se refieren a los bucles y a las bifurcaciones. El bucle *WHILE* posee una instrucción para la condición y un agrupamiento de instrucciones para el conjunto de rutinas a ejecutar. El bucle *FOR* siendo similar al *WHILE* debe tener un agrupamiento y además, algunas instrucciones que delimiten un rango de repetición. La bifurcación *IF-ELSE* debe tener una instrucción para la condición y dos agrupamientos, uno en caso que la condición sea verdadera y otro en caso que ésta sea falsa.

3.2 PROGRAMAS ESTRUCTURADOS VISTOS COMO ÁRBOLES

Las expresiones escritas en forma funcional (prefija) pueden ser representadas por medio de árboles n-arios (es decir, todo nodo puede tener entre 0 y n hijos). Si además la secuencia de instrucciones se representa como un vector ordenado, es posible generar una estructura de árbol más compleja que pueda representar un programa escrito en un lenguaje estructurado. Por ejemplo, en la Figura 3, se puede ver un programa escrito en lenguaje C, generado aleatoriamente, en donde las instrucciones algebraicas han sido reemplazadas por expresiones funcionales. Las líneas 3 a 6 muestran invocaciones a funciones (instrucciones sencillas); en las líneas 7 a 15 se puede observar una instrucción *IF-ELSE*, con dos agrupaciones de instrucciones; las líneas 16 a 19 muestran una instrucción *FOR*, con un agrupamiento y con las tres instrucciones sencillas que limitan su rango de repetición.

```
1 void main()
2 {
3   create(int,a,1);
4   create(int,b,0);
5   assign(a,5);
6   assign(b,pow(a,2));
7   if(big(a,b))
8   {
9    assign(b,0);
10   assign(a,add(a,1)); //a++
11  }
12  else
13  {
14   inc(b); //b++
15  }
16  for(assign(i,0);small(i,a);inc(i))
17  {
18   addequ(a,i);
19  }
20  printf("param1",param2,param3);
21  return;
22}
```

Instrucciones sencillas (líneas 3–6)
Instrucción if - else (líneas 7–15)
for (líneas 16–19)
Instrucciones compuestas (líneas 7–19)
Llamados a funciones (líneas 20–21)

Figura 3. Programa en C con instrucciones secuenciales, repetitivas y de bifurcación.

La Figura 4, muestra un vector de instrucciones numeradas, donde cada etiqueta corresponde al número de línea de código mostrado en la Figura 3. Cada cuadro del vector señala la instrucción correspondiente representada en forma de árbol. Por ejemplo, en el cuadro del vector con etiqueta 5, se representa en forma de árbol la instrucción de la línea número 5: (*assign(a , 5)*).

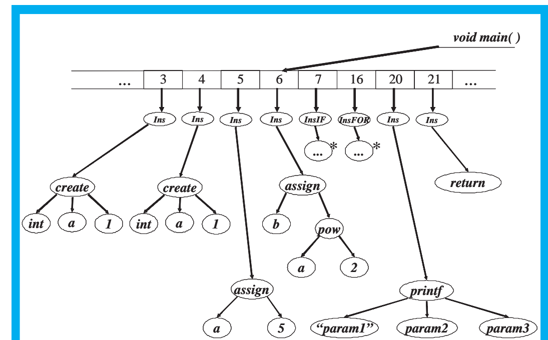

Figura 4. Árbol que muestra las instrucciones secuenciales del programa de la Fig. 3.

En la Figura 5, aparece la representación para la instrucción *IF-ELSE*. Se observa además las instrucciones de agrupamiento para la parte verdadera en las líneas 9 y 10, y para la parte falsa en la línea 14. El árbol generado para la instrucción *FOR* se puede apreciar en la Figura 6, donde las tres primeras ramas representan las instrucciones necesarias para limitar las repeticiones y la última representa el agrupamiento de las instrucciones a ser iteradas.

3.3 CARACTERÍSTICAS DE PG EN UN LENGUAJE ESTRUCTURADO

Dentro de la PG se tienen ciertas características importantes que se deben analizar al ser usadas en un lenguaje de programación específico, a saber, el

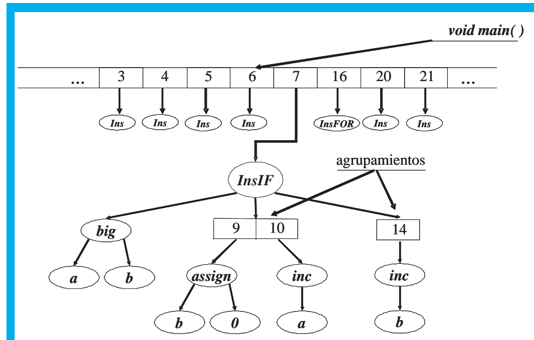

Figura 5. Árbol que muestra las instrucciones de bifurcación del programa de la Fig. 3.

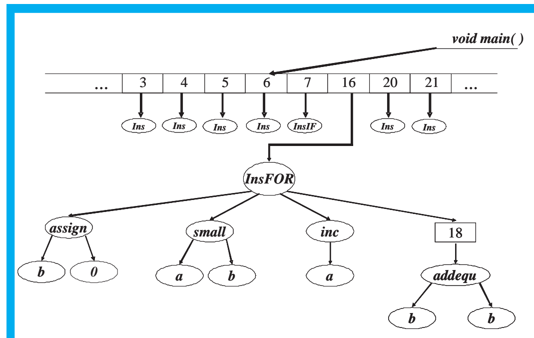

Figura 6. Árbol que muestra las instrucciones repetitivas del programa de la Fig. 3.

conjunto de terminales, el conjunto de funciones primitivas, el criterio de aptitud (*fitness*) y el criterio de finalización [4].

El conjunto de terminales que pueden ser utilizadas en un lenguaje estructurado, se limita a las posibles constantes y variables que pertenezcan a los tipos básicos de datos proporcionados (*int*, *float*, *char*, entre otros).

Las funciones primitivas que podrían ser empleadas en PG son aquellas determinadas por el mismo lenguaje (operaciones matemáticas, booleanas, bit a bit, etc.). Además, es posible incluir funciones implementadas por el desarrollador del software con el fin de ejecutar tareas especiales, si se agregan por medio de librerías adicionales. Una vez definido el conjunto de terminales y funciones primitivas es posible construir una estructura en forma de árbol, a partir de cualquier segmento de código de un programa.

El objetivo de la computación evolutiva es reproducir los procesos de optimización de los sistemas naturales en modelos computacionales.

El criterio de aptitud se define por medio del correcto funcionamiento de tal programa. Teniendo en cuenta que este no debe tener errores que impidan su ejecución, se puede precisar la medida de aptitud de un programa de acuerdo al número de errores de sintaxis que pueda llegar a tener por línea de código y luego, si se puede llegar a ejecutar, evaluando su respuesta conforme al problema planteado.

Por último, el criterio de finalización depende en gran medida del tipo de solución requerida y de la carga computacional necesaria para llegar a ella.

IV. PROLE: PLATAFORMA DE PG EN C

PROLE, es una plataforma para la evolución de programas escritos en lenguaje C, basada en el enfoque descrito en la sección anterior. Utiliza los conceptos de la computación evolutiva y esta orientada a la optimización de código escrito en lenguaje C, mediante PG. Se siguió el paradigma orientado por objetos para su desarrollo y se implementó la herramienta VisualC++ 6.0. A continuación, se describe parte del modelado de clases y seguido, algunos detalles de implementación.

El objetivo de la computación evolutiva es reproducir los procesos de optimización de los sistemas naturales en modelos computacionales. La abstracción de estos sistemas naturales es el punto de partida para el desarrollo de cualquier plataforma que se base en un prototipo evolutivo. La primera parte en el diseño de un modelo que represente un sistema natural, consiste en analizar y modelar un ecosistema con sus partes y relaciones. El paso a seguir, es definir el número y tipo de poblaciones que componen éste ecosistema. Cada una de estas poblaciones está integrada por un conjunto de individuos o criaturas, las cuales poseen información genética que las caracteriza. Además, el ecosistema se encuentra gobernado por leyes naturales como las operaciones genéticas y mecanismos como la selección natural.

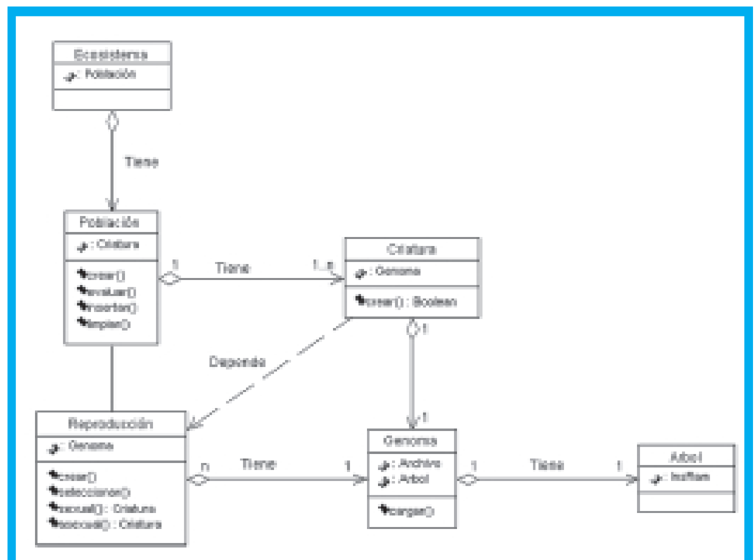

Figura 7. Modelo de clases del ecosistema.

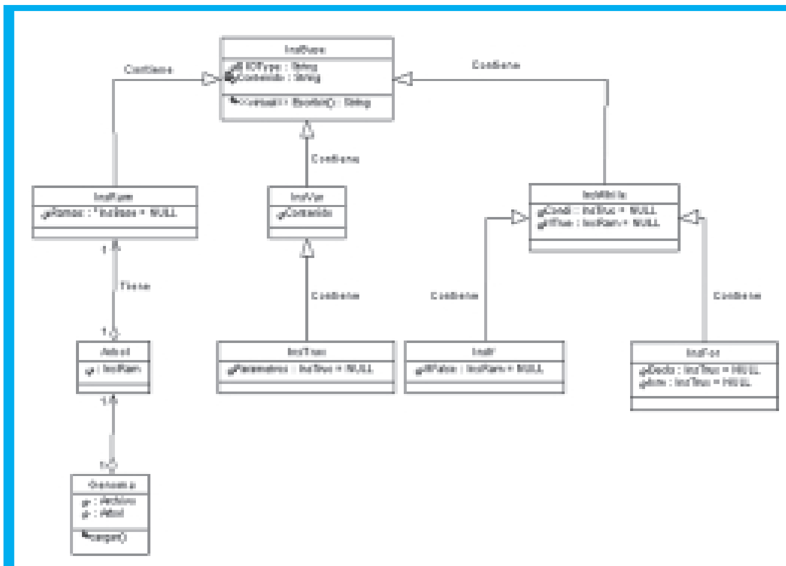

Figura 8. Modelo de clases de la jerarquía de instrucciones.

Partiendo de la anterior descripción se planteó un modelo básico de clases (ver Figura 7 y 8) dividido en dos partes principales. La primera donde se representan las clases *ecosistema, población, criatura, genoma y reproducción*. La segunda representa la composición interna del genoma de cada criatura, donde se observan clases que representan las instrucciones de programas en C, de acuerdo con la descripción dada en la sección 3.

Así pues, el genoma de cualquier individuo tiene un vector o agrupamiento principal de instrucciones que se ha denominado *InsRam* (rama de instrucciones) el cual contiene un arreglo de instrucciones de cualquier tipo (invocación a función, sentencias *if-else, while* o *for*). Todas estas instrucciones se han categorizado en una clase genérica denominada *InsBase* (instrucción básica); cuando se hace una referencia a un objeto *InsBase* éste puede tomar la forma de cualquiera de las instrucciones anteriores (aplicando el concepto de polimorfismo).

Dado que las instrucciones *if* y *for* tienen atributos en común con la instrucción *while*, ambas heredan de ésta. Por otro lado, las terminales (variables) se han descrito mediante la clase *InsVar*. Ésta posee un atributo denominado contenido, que es heredado por la clase *InsTruc* que representa una instrucción sencilla. Por último, la clase *Arbol* es la encargada de generar y manipular la estructura computacional del árbol.

V. RESULTADOS PRELIMINARES

Los primeros experimentos de PG que se describen a continuación, se han limitado a la realización de pruebas con individuos de una misma generación, es decir, aún no se han considerado las operaciones genéticas que se pueden efectuar durante varias generaciones del proceso evolutivo artificial. Esto con el fin de validar por una parte el enfoque propuesto para manipular los programas como árboles, y por otra, el modelo de clases y la plataforma desarrollada. En un futuro se reportarán los resultados correspondientes a experimentos donde se realice evolución y generación automática de programas.

Inicialmente se crearon individuos de forma manual, utilizando el modelo de clases propuesto. Después de crearlo, a cada individuo se le permitió producir un archivo con las instrucciones correspondientes en código C. Posteriormente se procedió a generar varios individuos pero de manera aleatoria, es decir, el número y tipo de instrucciones, y el nivel de profundidad se escogían automáticamente al azar. Para ello se diseñaron funciones recursivas que creaban terminales y ramas para que fueran siendo añadidas al árbol del programa generado. Se realizaron varias pruebas variando la longitud (número de instrucciones) y nivel de profundidad (número de subrayas del árbol) de los individuos generados. Utilizando un compilador de C, se midió la aptitud de cada individuo al calcular el número de errores de sintaxis por línea de código. En la Figura 9 se muestra la gráfica de la aptitud para una generación de individuos generada aleatoriamente.

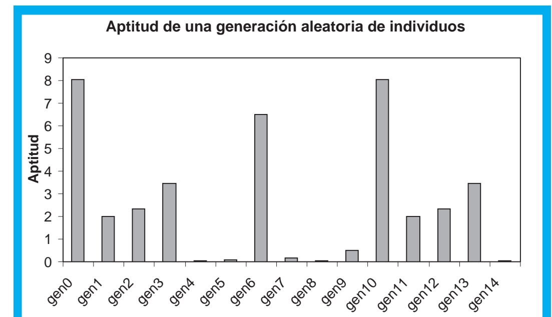

Figura 9. Aptitud de una generación aleatoria de 14 individuos.

Con el ánimo de visualizar la variación de la aptitud promedio de una generación de individuos, se procedió a ejecutar el experimento varias veces (es decir, se generaron poblaciones independientes cada una con 14 individuos). En la Figura 10 se observa el resultado obtenido. En futuros experimentos se espera comprobar de qué forma esta medida de aptitud va mejorando si se deja evolucionar cada población sujeta al proceso evolutivo de PG. Finalmente, a manera de ejemplo, en la Figura 11 aparece el listado de uno de los programas con mejor aptitud generados en el experimento. Este programa tiene 24 líneas de código y 18 errores, la mayoría de ellos por que no se incluyeron instrucciones para declaración de variables.

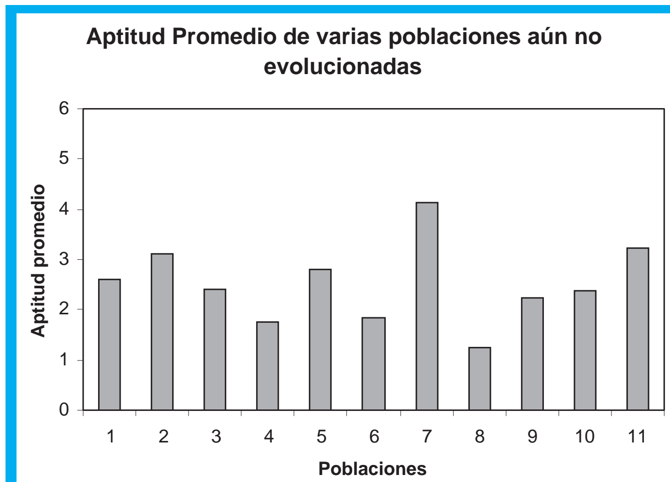

Figura 10. Aptitud promedio de varias poblaciones aún no evolucionadas.

```
void main(){
if ( mult(y,y))
{
while (greatere(i,j) )
{
greater(divi(i,j),add(z,i));
assign(sub(j,x),mod(j,z));
divi(mult(k,y),less(x,i));
greater(create(y,j,i),lesse(y,z));
greater(mult(i,z),lesse(k,j));
greater(lesse(k,i),x);
dif(mod(i,z),create(i,x,z));
};
}else {
divi(add(create(i,i,x),lesse(i,i)),create(divi(x,i),assign(y,x),add(i,z)));
assign(k,k);
add(mod(divi(y,j),lesse(x,k)),y);
while (assign(i,y) )
;
less(create(x,assign(z,i),sub(j,i)),divi(greater(y,z),lesse(y,j)));
sub(add(dif(j,x),assign(i,z)));
dif(less(greatere(i,z),greater(y,i)));
create(assign(lesse(assign(i,k),add(k,y)));
};
```

Figura 11.Código fuente de uno de los mejores programas generados aleatoriamente.

VI. DISCUSIÓN Y TRABAJO FUTURO

Se ha presentado una plataforma en la que programas de computador en lenguaje estructurado pueden ser manejados como estructuras arbóreas, con lo cual es posible aplicar el enfoque de PG para evolucionarlos. En ella se pueden crear programas estructurados que cuenten con instrucciones secuenciales, de bifurcación y de repetición. Además, estos programas pueden ser considerados criaturas de una población sujeta a un proceso evolutivo artificial.

Hasta el momento las pruebas realizadas han incluido la generación de una población inicial aleatoria y una población a partir de programas prototípicos. La medida de aptitud que se ha utilizado corresponde al número de errores de sintaxis al evaluarlo con un compilador de C. Sin embargo, aún no se ha considerado dentro de su aptitud, la funcionalidad de cada individuo, es decir, que no solo no tenga errores de sintaxis sino que además ejecute un algoritmo útil. En este sentido se podrían realizar experimentos para resolver problemas conocidos (por ejemplo, ordenamiento) en donde se configuren poblaciones iniciales basadas en algoritmos ya probados (como el *quicksort*), y evolucionarlos para encontrar nuevos algoritmos, quizás con un mejor desempeño computacional.

Otra de las aplicaciones es buscar nuevas soluciones o soluciones desconocidas a problemas, comenzando con poblaciones completamente aleatorias. En este caso se deben diseñar adecuadamente las funciones de aptitud y probablemente afinar los operadores genéticos para guiar el proceso de optimización.

Finalmente, aunque los lenguajes estructurados son todavía ampliamente utilizados, también es cierto que en la actualidad son más populares otros paradigmas como la programación orientada por objetos. Un siguiente paso en la PG puede ser la PG orientada por objetos, en donde se evolucionen objetos, relaciones y colaboraciones entre ellos. Esta será uno de nuestras posibles áreas de exploración en futuros proyectos.

REFERENCIAS BIBLIOGRÁFICAS

[1] Darwin, C. R. (1859). *The origin of species.* Londres: John Murrap.

[2] Goldberg, D.E. (1986). *Genetic Algorithms in Search, Optimization, and Machine Learning.* Cambridge: MIT Press.

[3] Kinnear, K.E.(1994). *Advances in Genetic Programing.* MIT Press.

[4] Koza, J.R. (1992) *Genetic programming: on the programing of coputers by mean of natural selection.* MIT Press.

[5] Lemone, K.A.(1999). *Fundamentos de compiladores.* Mexico: CECSA.Autores

Sergio A. Rojas
Ing. de Sistemas, U. Nacional de Colombia. Especialista en Ing. de Software, U. Distrital. Profesor/investigador de la Facultad de Ingeniería, Universidad Distrital FJC. **srojas@udistrital.edu.co**

Julián Y. Olarte
Estudiante de Ing. de Sistemas, U. Distrital Francisco José de Caldas. **julian_olarte@hotmail.com**

David E. Suárez
Estudiante de Ing. de Sistemas, U. Distrital Francisco José de Caldas. **davidsua@hotmail.com**