La Importancia de la Representación en los Algoritmos Genéticos (Parte II)

Carlos A. Coello Coello
LANIA, A.C.
Rébsamen 80, Apartado Postal 696
Xalapa, Veracruz 91090, México
ccoello@xalapa.lania.mx

Resumen

La segunda parte de este artículo analiza la representación de árbol utilizada por la programación genética y un esquema híbrido de representación al que se denomina "algoritmo genético estructurado". Además, se mencionan brevemente otras propuestas de uso menos común y algunas de las tendencias futuras en esta área de investigación.

5. Representación de árbol

Una de las metas originales de la Inteligencia Artificial (IA) fue la generación automática de programas de computadora que pudiesen efectuar una cierta tarea. Durante varios años, sin embargo, esta meta pareció demasiado ambiciosa puesto que normalmente el espacio de búsqueda se incrementa exponencialmente conforme extendemos el dominio de un cierto programa y, consecuentemente, cualquier técnica tenderá a producir programas no válidos o altamente ineficientes.

Algunas técnicas de computación evolutiva intentaron lidiar con el problema de la programación automática desde su mera concepción, pero sus notables fallas aún en dominios muy simples, hicieron que otros investigadores de la comunidad de IA no se tomaran en serio estos esfuerzos [1]. Sin embargo, Holland desarrolló el concepto moderno del AG dentro del entorno del aprendizaje de máquina [2], y todavía existe una cantidad considerable de investigación dentro de esa área, aunque la programación automática fue hecha a un lado por los investigadores de IA durante varios años. Una de las razones por las que esto sucedió fue el hecho de que el AG tiene algunas limitaciones (obvias) cuando se pretende usar para programación automática, particularmente en términos de la representación.

Codificar el conjunto de instrucciones de un lenguaje de programación y encontrar una forma de combinarlas que tenga sentido no es una tarea simple, pero si usamos una estructura de árbol junto con ciertas reglas para evitar la generación de expresiones no válidas, podemos construir un evaluador de expresiones primitivo que pueda producir programas simples. Este fue precisamente el enfoque tomado por John Koza [3] para desarrollar la denominada "programación genética", en la cual se usó originalmente el lenguaje de programación LISP para aprovechar su evaluador de expresiones integrado al intérprete.

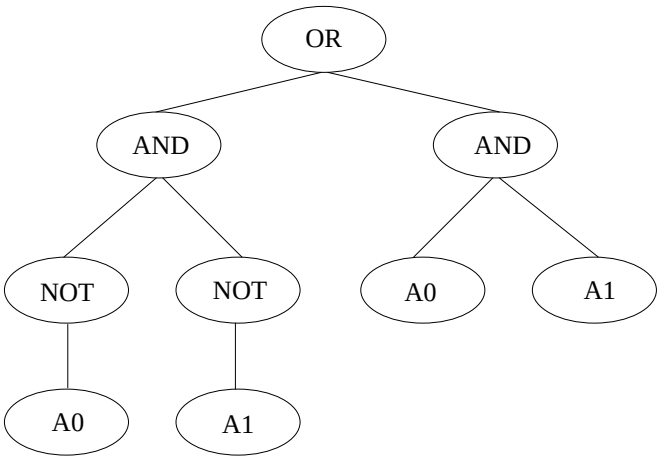

Figura 9 : Un ejemplo de un cromosoma usado en programación genética.

La representación de árbol adoptada por Koza requiere obviamente de alfabetos diferentes y operadores especializados para evolucionar programas generados aleatoriamente hasta que éstos se vuelvan 100% válidos para resolver cierta tarea predefinida, aunque los principios básicos de esta técnica pueden generalizarse a cualquier otro dominio. Los árboles se componen de funciones y terminales. Las funciones usadas normalmente son las siguientes [3]:

1. Operaciones aritméticas (por ejemplo: +, -, ×, ÷)

2. Funciones matemáticas (por ejemplo: seno, coseno, logaritmos, etc.)

3. Operaciones Booleanas (por ejemplo: AND, OR, NOT)

4. Condicionales (IF-THEN-ELSE)

5. Ciclos (DO-UNTIL)

6. Funciones recursivas

7. Cualquier otra función definida en el dominio utilizado

Las terminales son típicamente variables o constantes, y pueden verse como funciones que no toman argumentos. Un ejemplo de un cromosoma que usa las funciones F={AND, OR, NOT} y las terminales T={A0, A1} se muestra en la figura 9.

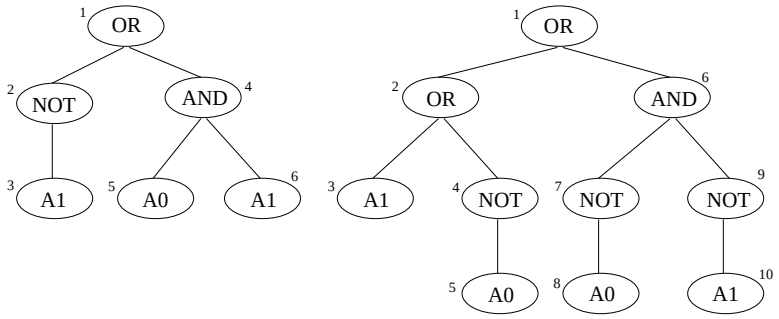

Figura 10 : Los nodos del árbol se numeran antes de aplicar el operador de cruza.

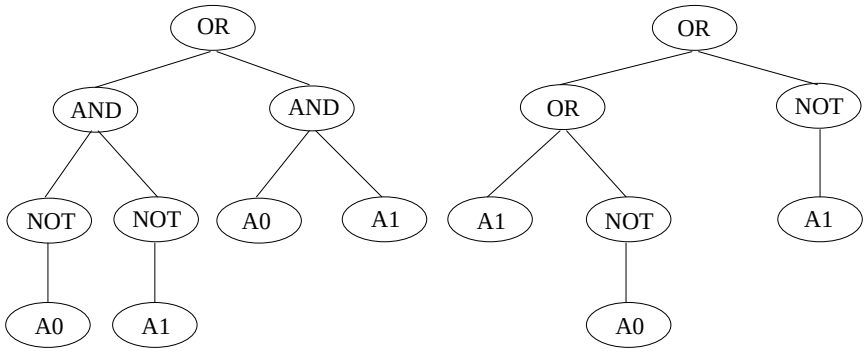

Figura 11 : Los dos hijos generados después de efectuar la cruza.

La cruza puede efectuarse numerando los nodos de los árboles correspondientes a los 2 padres elegidos (ver figura 10) y seleccionando (al azar) un punto en cada uno de ellos de manera que los sub-árboles por debajo de dicho punto se intercambien (ver figura 11, donde suponemos que el punto de cruza para el primer padre es 2 y para el segundo es 6).

Típicamente, los tamaños de los 2 árboles padres será diferente, como se muestra en el ejemplo anterior. También debe observarse que si el punto de cruza es la raíz de uno de los dos árboles padres, entonces todo ese cromosoma se volverá un sub-árbol del otro padre, lo cual permite la incorporación de subrutinas en un programa. También es posible que las raíces de ambos padres sean seleccionadas como puntos de cruza. En ese caso, no se efectúa la cruza, y los hijos pasan a ser idénticos a sus padres. Normalmente, la implementación de la programación genética impone un límite en cuanto a la máxima profundidad que puede alcanzar un árbol, a fin de evitar la generación (al azar y producto del uso de la cruza y la mutación) de árboles de gran tamaño y complejidad.

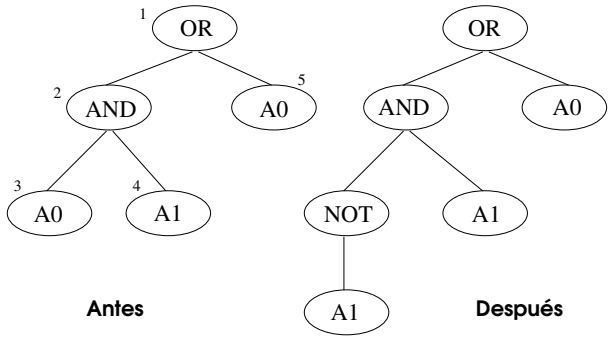

Figura 12 : Un ejemplo de mutación en la programación genética.

La mutación se efectúa mediante la selección (aleatoria) de un cierto punto de un árbol. El sub-árbol que se encuentre por debajo de dicho punto es reemplazado por otro árbol generado al azar. La figura 12 muestra un ejemplo del uso de este operador (el punto de mutación en este ejemplo es el 3).

La permutación es un operador asexual que emula el efecto del operador de inversión que se usa en los algoritmos genéticos [4]. Este operador reordena las hojas de un sub-árbol ubicado a partir de un punto elegido al azar, y su finalidad es fortalecer la unión de combinaciones de alelos con buen desempeño dentro de un cromosoma [2].

La figura 13 muestra un ejemplo del uso del operador de permutación (el punto seleccionado en este caso es el 4). En la figura 13, el '*' indica multiplicación y el '%' indica "división protegida", refiriéndose a un operador de división que evita que nuestro programa genere un error de sistema cuando el segundo argumento sea cero.

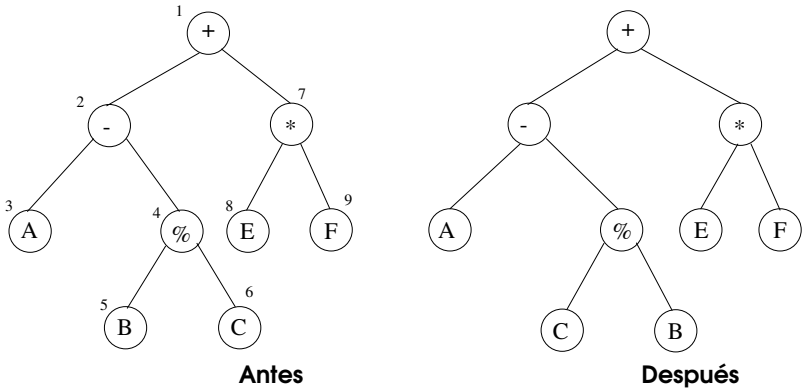

Figura 13 : Un ejemplo de permutación en la programación genética.

En la programación genética es posible también proteger o "encapsular" un cierto sub-árbol que sepamos constituye un buen bloque constructor, a fin de evitar que sea destruido por los operadores genéticos. El sub-árbol seleccionado es reemplazado por un nombre simbólico que apunta a la ubicación real del sub-árbol, y dicho sub-árbol es compilado por separado y enlazado al resto del árbol de forma similar a las clases externas de los lenguajes orientados a objetos. La

figura 14 muestra un ejemplo de encapsulamiento en el cual el sub-árbol de la derecha es reemplazado por el nombre (**E0**).

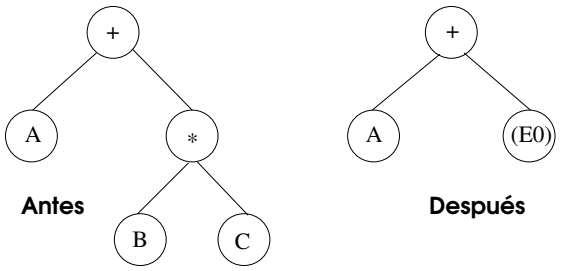

Figura 14 : Un ejemplo de encapsulamiento en programación genética.

Normalmente, también es necesario editar las expresiones generadas a fin de simplificarlas, aunque las reglas para llevar a cabo este proceso dependen generalmente del problema. Por ejemplo, si estamos generando expresiones Booleanas, podemos aplicar reglas como las siguientes:

$$(\textbf{AND}\ X\ X) \rightarrow X$$
$$(\textbf{OR}\ X\ X) \rightarrow X$$
$$(\textbf{NOT}\ (\textbf{NOT}\ X)) \rightarrow X$$

Finalmente, la programación genética también proporciona mecanismos para destruir un cierto porcentaje de la población de manera que podamos renovar el material cromosómico después de un cierto número de generaciones. Este mecanismo, llamado ejecución, es muy útil en dominios de alta complejidad en los cuales nuestra población puede no contener ni un solo individuo factible aún después de un gran número de generaciones.

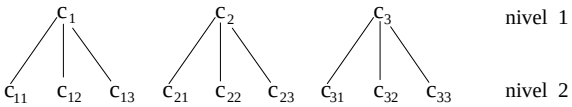

Figura 15 : Un ejemplo de estructura de dos niveles de un AG estructurado.

6. Algoritmo Genético Estructurado

Dasgupta [5] propuso una representación que es un compromiso entre el cromosoma lineal tradicional de longitud fija y la codificación de árbol usada en la programación genética. Su técnica, denominada *Algoritmo Genético Estructurado* (stGA por sus siglas en inglés), usa una representación jerárquica con un mecanismo similar a los diploides[1] [2], en la cual ciertos genes actúan como

[1] Un **diploide** en biología se refiere a una célula que contiene 2 copias de cada cromosoma. En algoritmos genéticos, la definición es análoga pues en los diploides existen 2 cadenas (o cromosomas) asociadas a cada individuo. Normalmente, los individuos utilizados en computación evolutiva suelen ser **haploides**, ya que sólo se asocia una cadena cromosómica con cada individuo.

operadores de cambio (o dominancia) que encienden o apagan ciertos genes, a los cuales se les llama activos o pasivos, respectivamente [5].
El stGA usa una cadena cromosómica lineal, pero realmente codifica una estructura genética de varios niveles (un grafo dirigido o un árbol) tal y como se indica en la figura 15. Los genes en cualquier nivel pueden ser activos o pasivos, pero los genes de alto nivel activan o desactivan conjuntos de genes de más bajo nivel, lo que significa que cualquier cambio pequeño a un alto nivel se magnifica en los niveles inferiores [6]. La idea es que los genes de alto nivel deben explorar las áreas potenciales del espacio y los de bajo nivel deben explotar ese sub-espacio.

$$(\; c_1 \;\; c_2 \;\; c_3 \qquad c_{11} \;\; c_{12} \;\; c_{13} \;\; c_{21} \;\; c_{22} \;\; c_{23} \;\; c_{31} \;\; c_{32} \;\; c_{33} \;)$$

Figura 16 : Una representación cromosómica de la estructura de 2 niveles del AG estructurado.

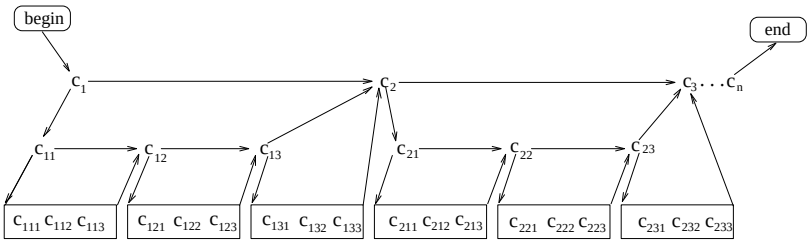

Figura 17 : Ejemplo de una estructura de datos usada para implementar un AG estructurado.

La estructura jerárquica usada por el stGA es, sin embargo, codificada como un cromosoma lineal de longitud fija, tal y como se muestra en la figura 16. No obstante, la estructura de datos que se requiere para implementar un stGA es ligeramente más complicada que el arreglo unidimensional que requiere un AG tradicional. La figura 17 muestra un ejemplo de dicha estructura de datos. En esta figura, cada gene en los niveles superiores actúa como un puntero cambiable que tiene dos estados posibles: cuando el gene está activo (encendido), apunta a su gene de más bajo nivel y cuando es pasivo (apagado), apunta al gene del mismo nivel en que se encuentre [6].

7. Otras propuestas

Las representaciones anteriores no son las únicas alternativas que existen en la literatura especializada. Por ejemplo, Antonisse [7] y Gero et al. [8] han propuesto el uso de gramáticas en el contexto de lenguajes de programación y diseño en ingeniería, respectivamente. De hecho, Antonisse asegura qe su representación es más poderosa que la de Koza [3], porque define gramáticas sensibles al contexto, las cuales son más generales (en la jerarquía de lenguajes propuesta por Noam Chomsky) que las expresiones-S usadas por Koza en LISP [7].

Algunas otras representaciones más dependientes del problema, tales como la matricial [9,10], la multi-dimensional [11,15] y la de permutaciones [12,16] han sido propuestas también por algunos investigadores.

8. Conclusiones y tendencias futuras

En este artículo hemos intentado proporcionar un panorama general de algunas de las principales técnicas de representación que han sido propuestas como alternativas a la representación binaria tradicional del algoritmo genético.

Inicialmente, mostramos algunas de las principales motivaciones para explorar el uso de alfabetos de mayor cardinalidad y luego examinamos algunas de las propuestas más interesantes reportadas en la literatura técnica.

Las limitaciones de la representación binaria en algunas aplicaciones ha hecho de ésta una ruta interesante de investigación, sobre todo en el contexto de problemas de manufactura y definición de rutas, en los cuales la respuesta se expresa en forma de permutaciones. Idealmente, debiera existir un tipo de representación suficientemente flexible como para permitir su utilización en una amplia gama de problemas de forma sencilla y natural. Filho [13] desarrolló un sistema llamado GAME (*Genetic Algorithms Manipulation Environment*), que constituye un paso importante en esta dirección. GAME usa una codificación de árbol en la que las hojas pueden ser enteros, caracteres, números reales o cadenas binarias.

Gibson [13] también propuso una codificación híbrida similar a la de Filho. Su propuesta se basa en el uso de componentes de diferentes tipos en el contexto de modelado de procesos industriales.

Existe amplia evidencia en la literatura especializada [8,12,16,17,18] de que el usar una representación adecuada puede simplificar tremendamente el proceso de búsqueda de un algoritmo genético, pero a pesar de eso, muchos investigadores suelen pasar por alto este importante hecho.

Por lo tanto, es vital no únicamente reconocer que se requieren codificaciones más poderosas, sino también saber más acerca de las representaciones (distintas a la binaria tradicional) que actualmente existen, pues su uso adecuado puede ahorrar mucho trabajo innecesario y permitirnos concentrarnos en la aplicación misma del algoritmo genético a nuestro problema en vez de canalizar todas nuestras energías en el diseño de operadores genéticos especiales.

Referencias Bibliográficas

[1] Fogel, D. B. (1995*), Evolutionary Computation. Toward a New Philosophy of Machine Intelligence*, The Institute of Electrical and Electronic Engineers, New York.

[2] Holland, J. H. (1975), *Adaptation in Natural and Artificial Systems*, Ann Harbor : University of Michigan Press.

[3] Koza, J. R. (1992), *Genetic Programming. On the Programming of Computers by Means of Natural Selection*, The MIT Press.

[4] Goldberg, D. E. (1989), *Genetic Algorithms in Search, Optimization and Machine Learning*, Reading, Massachusetts : Addison-Wesley Publishing Co.

[5] Dasgupta, D. & McGregor, D. R. (1994) , 'A more Biologically Motivated Genetic Algorithm: The Model and some Results', *Cybernetics and Systems: An International Journal* **25**(3), pp. 447-469.

[6] Dasgupta, D. & McGregor, D. R. (1992), Nonstationary Function Optimization using the Structured Genetic Algorithm, *en* 'Proceedings of Parallel Problem Solving from Nature (PPSN 2)', Springer-Verlag, Brussels, pp. 145-154.

[7] Antonisse, H. J. (1991), A Grammar-Based Genetic Algorithm, *en* G.E. Rawlins, ed., 'Foundations of Genetic Algorithms', Morgan Kaufmann Publishers, San Mateo, California, pp. 193-204.

[8] Gero, J. S., Louis, S. J. & Kundu, S. (1994) , 'Evolutionary learning of novel grammars for design improvement', *AIEDAM* **8**(3), pp. 83-94.

[9] Vignaux, G. & Michalewicz, Z. (1989), 'Genetic algorithms for the transportation problem', *Methodologies for Intelligent Systems* **4**, pp. 252-259.

[10] Beasley, D.; Bull, D. & Martin, R. (1993), Reducing epistasis in combinatorial problems by expansive coding, *en* S. Forrest, ed., 'Proceedings of the Fifth International Conference on Genetic Algorithms', University of Illinois at Urbana-Champaign, Morgan Kauffman Publishers, San Mateo, California, pp. 400-407.

[11] Bui, T. N. & Moon, B. R. (1995), On multidimensional encoding/crossover, en Larry J. Eshelman, editor, 'Proceedings of the Sixth International Conference on Genetic Algorithms', Morgan Kauffman Publishers, San Francisco, California, pp. 49-56.

[12] Grefenstette, J.; Gopal, R.; Rosmaita, B. & Gucht, D. (1985) Genetic algorithms for the travelling salesman problem, *en* John J. Grefenstette, ed., 'Proceedings of an International Conference on Genetic Algorithms and Their Applications', Lawrence Erlbaum Associates, Hillsdale, New Jersey, pp. 160-168.

[13] Filho, R. (1994) The GAME system (Genetic Algorithms Manipulation Environment), IEE Colloquium on Applications of Genetic Algorithms, pp. 2/1-2/4, IEE, London, UK.

[14] Gibson, G. (1995) *Application of Genetic Algorithms to Visual Interactive Simulation Optimisation*, PhD Thesis, University of South Australia.

[15] Anderson, C. A.; Jones, K. F. & Ryan, J. (1991) A two-dimensional genetic algorithm for the ising problem, *Complex Systems*, Vol. 5, pp. 327-333.

[16] Michalewicz, Z. (1992), *Genetic Algorithms + Data Structures = Evolution Programs*, second edition, Springer-Verlag.

[17] Mathias, K. E. & Whitley, L. D. (1994*b*), 'Changing Representations During Search: A Comparative Study of Delta Coding', *Evolutionary Computation* **2**(3),249-278.

[18] Ronald, S. (1997) Robust Encodings in Genetic Algorithms, *en* D. Dasgupta & Z. Michalewicz, editors, Evolutionary Algorithms in Engineering Applications, Springer-Verlag, pp. 30-44.