Universidad de Chile
Facultad de Cs. Físicas y Matemáticas
Departameto de Ciencias de la Computación
CC5114 - Redes Neuronales y Programación Genética

Tarea 4

Programación genética y AST

Autor: Matías Meneses C.
Profesor: Alex Bergel
3 de diciembre de 2017
Santiago, Chile.

Índice

1. Introducción

El objetivo de esta tarea es implementar un algoritmo de programación genética para árboles de operaciones binarias, con el objetivo de estudiar la implementación práctica de programación genética.

1.1. Software utilizado

Se utilizó el lenguaje de programación Scala para la implementación del AST y los test unitarios. Se eligió este lenguaje debido a la facilidad para realizar Pattern Matching sobre clases, y para variar de las tareas anteriores en donde se utilizó Python. Se utilizó SBT como gestor de paquetes y compilación.

2. Estructura de Árbol

Scala permite el uso de *case classes* para poder realizar pattern matching de forma intuitiva y funcional.

```
trait ArithmeticNode {

    def evaluate(): Int = this match {
      case Number(n) => n
      case Plus(n1, n2) => n1.evaluate() + n2.evaluate()
      case Minus(n1, n2) => n1.evaluate() - n2.evaluate()
      case Mult(n1, n2) => n1.evaluate() * n2.evaluate()
      // se establece que x / 0 es x
      case Div(n1, n2) =>
        if (n2.evaluate() == 0) n1.evaluate()
        else n1.evaluate() / n2.evaluate()
    }

    ...
}

case class Number(n: Int) extends ArithmeticNode
case class Plus(l: ArithmeticNode, r: ArithmeticNode) extends ArithmeticNode
case class Minus(l: ArithmeticNode, r: ArithmeticNode) extends ArithmeticNode
case class Mult(l: ArithmeticNode, r: ArithmeticNode) extends ArithmeticNode
case class Div(l: ArithmeticNode, r: ArithmeticNode) extends ArithmeticNode
```

Se definió además un trait (interfaz) AlgebraicNode, cuyas terminales son identificadores en lugar de números.

Se implementaron métodos para generar árboles aleatorios, y reemplazar de forma aleatoria un subárbol de un árbol por otro árbol, lo cual es útil para la implementación de los algoritmos genéticos. Además, se implementaron métodos para obtener la altura del árbol y para imprimirlo en pantalla de forma legible.

3. Algoritmos Genéticos

3.1. Selection

La función de selección recibe como parámetro una lista de árboles, la cantidad de árboles a seleccionar y una función de fitness $Node \to \mathbb{Z}$, y retorna una lista con los árboles seleccionados, ordenados por mejor fitness.

3.2. Crossover

La función de crossover recibe como parámetro una lista de árboles (padres) y el número de hijos a generar, y retorna una lista con los hijos generados. La función toma dos padres al azar de la lista de padres, y luego reemplaza un subárbol aleatorio del segundo padre en el primer padre, para formar a un hijo, el cual tiene una probabilidad de 1 % de ser mutado. Se repite este proceso hasta generar todos los hijos.

3.3. Mutation

La función de mutación recibe como parámetro un árbol y retorna un árbol mutado. El árbol mutado es el resultado del reemplazo de un subárbol aleatorio del árbol por un árbol generado aleatoriamente.

4. Experimentos

En esta oportunidad, los experimentos se implementaron como test unitarios.

4.1. Primer Experimento

En el primer experimento se utiliza el árbol aritmético (las terminales son números hasta 100). La idea es que a partir de 20 árboles aleatorios, evolucionen para formar un árbol que evalúe a un número entre 40 y 50.

La función de fitness para este problema es la distancia al punto medio del intervalo. A menor fitness, más cerca de la respuesta.

```
val fitness = (n: ArithmeticNode) => Math.abs(n.evaluate() - 45.0)
```

Se itera siguiendo los pasos del algoritmo genético, hasta entontrar un árbol que solucione el problema. Se varió la profundidad de los árboles generados aleatoriamente para analizar su impacto.

4.2. Segundo Experimento

En el segundo experimento se utiliza el árbol algebraico (las terminales son variables x, y, z). La idea es que a partir de 20 árboles aleatorios, evolucionen para formar un árbol que, dado $x = 1, y = 2, z = 3$, evalúe a un número entre $40 y 50$.

La función de fitness es similar al experimento anterior.

```
val fitness = (n: AlgebraicNode) => Math.abs(n.evaluate(1,2,3) - 45.0)
```

Se itera siguiendo los pasos de algoritmo genético, hasta encontrar un árbol que solucione el problema.

5. Resultados

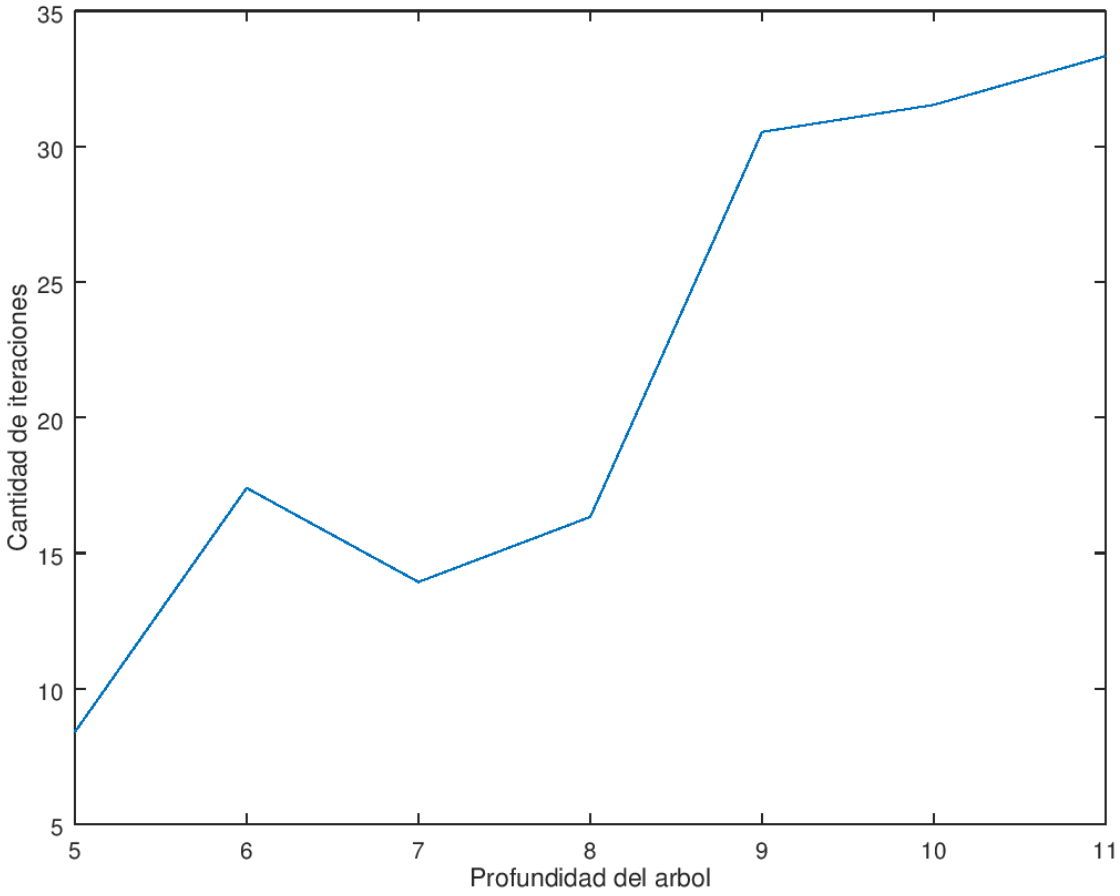

Figura 1: Cantidad de iteraciones hasta encontrar la solución por altura del árbol

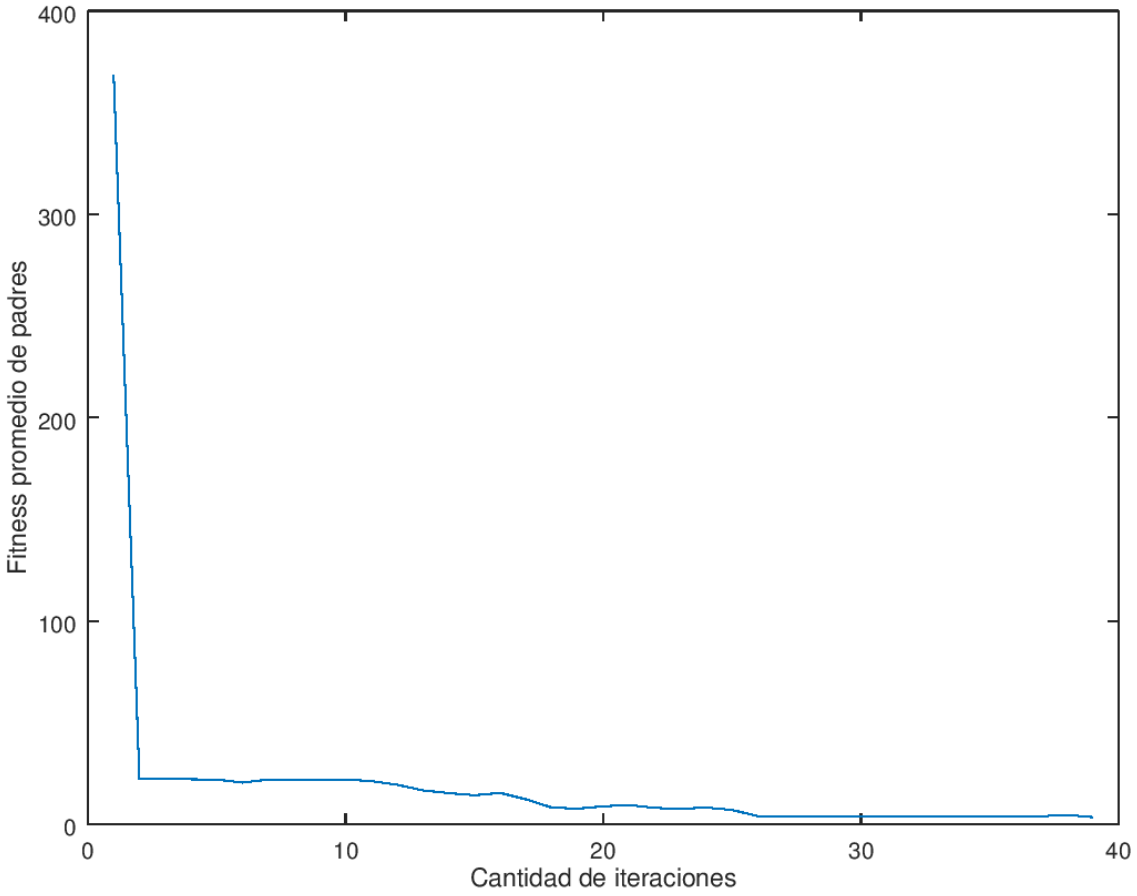

Figura 2: Fitness promedio de padres por cantidad de iteraciones

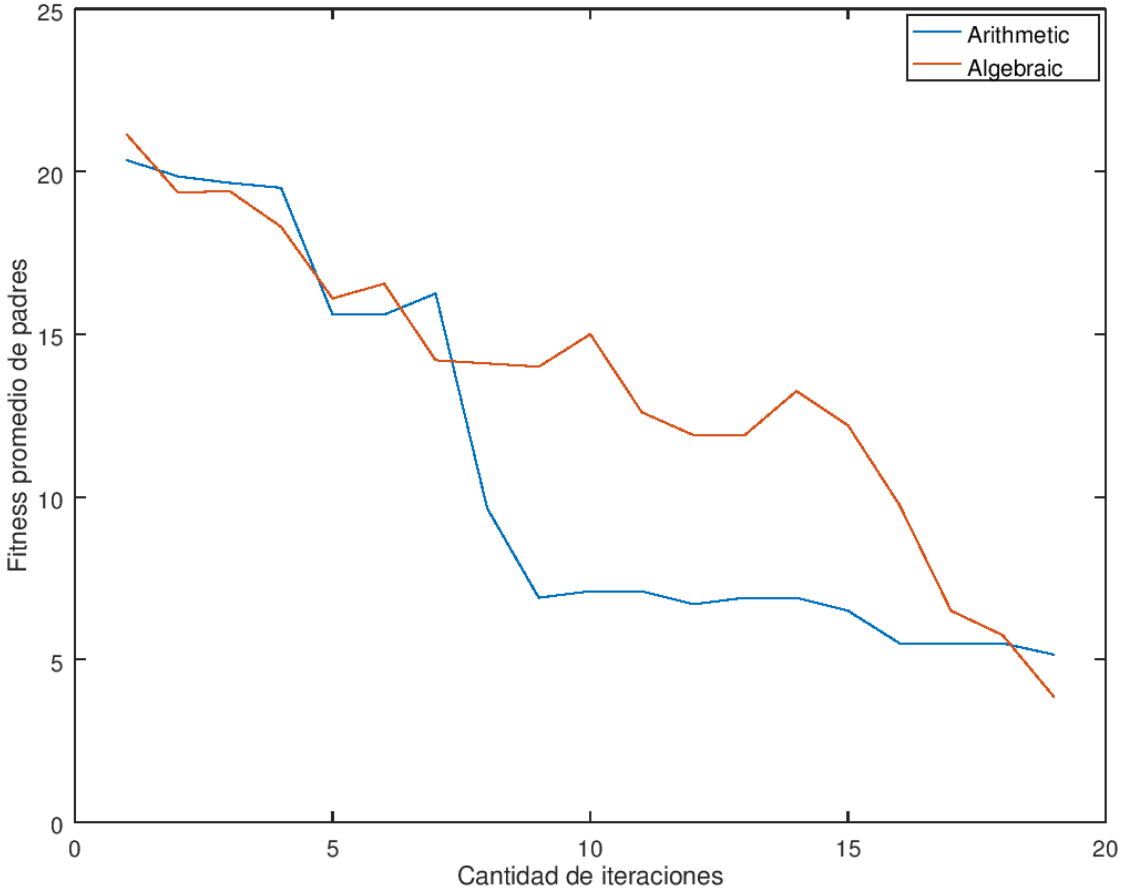

Figura 3: Fitness promedio de padres por cantidad de iteraciones (sin singularidad)

6. Análisis de Resultados

Se puede ver de la figura 1, que mientras mayor es la altura de los árboles generados, mayor es la cantidad de iteraciones requerida para encontrar una solución, lo cual era de esperar, puesto que mientras más complejos son los programas, mayor es la cantidad de outputs que puede generar, por lo que será más difícil acotar la solución.

Se puede ver de las figuras 2 y 3 que el algoritmo genético logra aproximar las soluciones en cada iteración. En todas las ejecuciones, el fitness promedio de la primera iteración siempre era un valor muy alejado de la solución, por lo que se removió en la figura 3 para visualizar de mejor forma la variación. Esta figura también compara las iteraciones para los dos tipos de árboles implementados, no habiendo diferencias significativas.

7. Conclusión

Se concluye que el algoritmo genético fue capaz de generar programas que solucionan las ecuaciones objetivo, en ambos ejemplos que utilizan terminales distintas. Se espera que, ante un AST con mayor cantidad de tipos de nodos, también se puedan generar programas que aproximen a una solución.

Se cumplió el objetivo propuesto para esta tarea.