GRUPO 13

3 de mayo de 2023 -
Programación Evolutiva

INTEGRANTES

Andrés Cardenal Antón
Rafael Alonso García

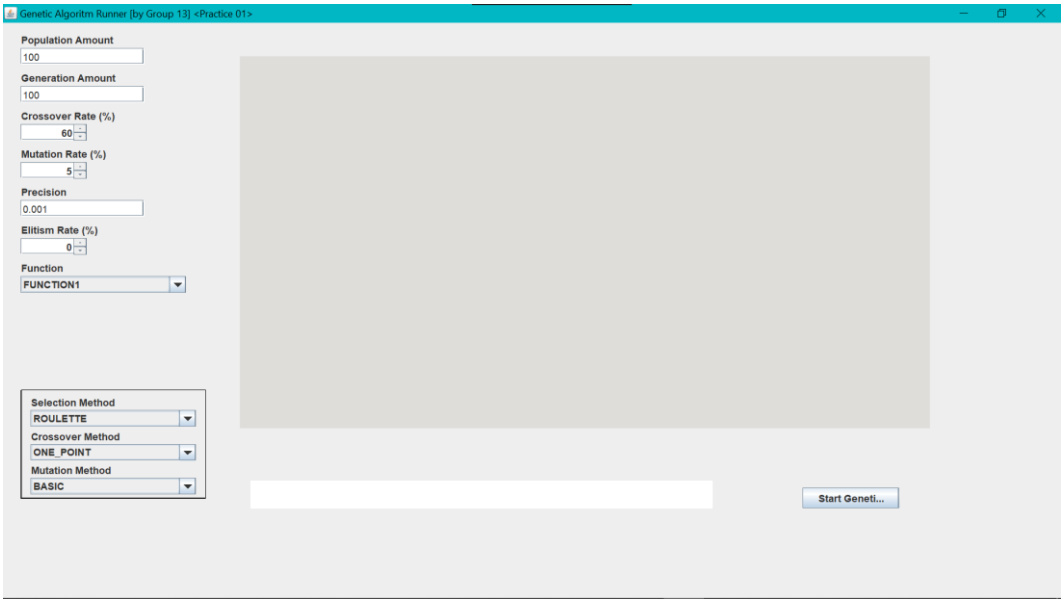
CONTENIDO

1. Guía de Uso

EJECUCIÓN DEL PROGRAMA

Para probar el programa se arrancará el ejecutable o tras haber importado el proyecto, desde el IDE Eclipse.

Se introducirán los parámetros de la izquierda antes de empezar la prueba, tal como funcionaban las prácticas anteriores. Se podrá ver la gráfica de la población con el fitness del mejor individuo, el mejor generacional y la media de cada generación. También está disponible una gráfica que compara la función de máximo fitness ***x^4 + x^3 + x^2 + x + 1*** con la función del mejor individuo obtenido.



Menú de Inicio

2. Arquitectura de la Aplicación

Debido a que en los comentarios de la entrega anterior se pedía que no se implementasen las funcionalidades exclusivas de las prácticas anteriores, hemos borrado y simplificado la lógica de esta última entregan para sólo incluir lo necesario con respecto a los últimos requisitos.

En la clase ***TreeChromosome*** se implementan los métodos del individuo. En este caso, los genes forman una estructura de árbol, los cuales todos surgen del nodo principal, *raíz* de la clase ArithmeticNode. Cuando se trata de un nodo no terminal esta clase contiene sus ramas izquierda y derecha al igual que el valor del propio *nudo* implementado por un *ArithmeticEnum.* Por el contrario, cuando se trata de un nodo *hoja* todos estos anteriores atributos serán nulos y solo tendrá el registro *fruit* del nodo que está implementada por la clase *TerminalEnum*. Para obtener el valor de un individuo, se pasará por parámetro el valor de la *x* que se desea usar. La implementación general realizará la operación correspondiente entre la rama izquierda y derecha, aunque si el nodo es una hoja, simplemente se devolverá su valor terminal o el valor pasado como *x*.

Al igual que en prácticas anteriores desde el *Launcher* se crea la ***Window,*** *l*a cual almacenará todos los campos que es necesario introducir para que se ejecute el algoritmo y que a través de la interfaz

RequestMaker y de la clase *Request* nos aseguramos de que la entrada cumple los requisitos. A continuación, el *Controller* llama al *Builder* con la configuración actual donde se inicializa la función ***AdaptativeFunction*** que hereda de *Fitness* y también se inicializa el *Mold* que es una referencia a parámetros de inicialización que poseen todos los cromosomas; el *Mold* también almacena la variable ***k*** que se obtiene mediante la fórmula *k = Covarianza (l, f) / Varianza(l)* perteneciente al método de **Control del Bloating,** *Penalización bien fundamentada (Poli and McPhee, 2008b).*

El *Executer* se lleva a cabo la iteración general de los algoritmos genéticos y a través de las clases *GrowPopulationInitializer*, *FullPopulationPopulationInitializer* y *RampedAndHalfPopulationInitializer* se inicializan los árboles *cuando se ha seleccionado* ***GenType*** de *TREE,* que difiere del utilizado por las gramáticas evolutivas. La clase *TreeCrossover* implementa el cruce entre árboles de intercambio de nodos y por su parte las clases *ContractionTreeMutation*, *FunctionalTreeMutation*, *HoistTreeMutation*, *PermutationTreeMutation* (todas heredan de TreeMutation que a su vez implementa la interfaz *MutationI*) se encargan de la mutación.

La penalización del *bloating* se incorpora en el método *evaluate()* del *TreeChromosome*.

Cuando se selecciona ***GenType*** de *BINARY se utiliza el método de mutación de BasicBinaryMutation y los métodos de selección seleccionables son Monopunto (ONE_POINT) y Uniforme (UNIFORM).*

En el paquete model.util podemos observar a las clases *Pair*, *Cast* para usar el tipo *TreeChromosome* en todo momento, *Covariance* y *Variance* en el caso de usar *bloating*.

Los nuevos métodos de cruce implementan a la clase abstracta *Crossover* que se encarga de los emparejamientos y los nuevos métodos de mutación implementan a ***TreeMutationI*** que a su vez extiende a *MutationI*. Las clases *InitializerBuilder*, *MutationBuilder* y *CrossoverBuilder* se han refactorizado para incluir a los nuevos métodos, que serán generado por el *Builder*, antes de que el ***Controller***, ejecute el programa a través del *Executor*.

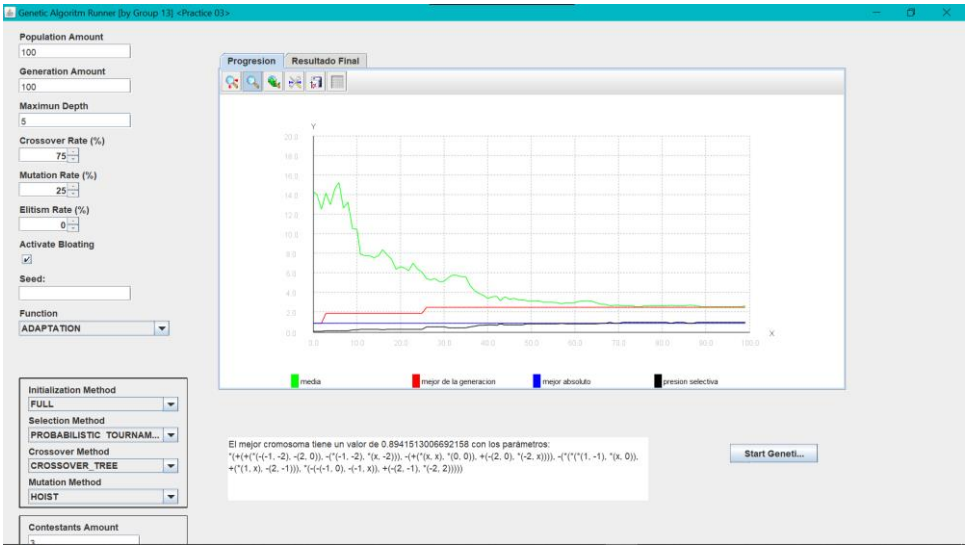
3. Cruces y mutaciones

A continuación, se presentarán ejemplos de ejecución para los nuevos métodos de mutación e inicialización. En general, se ha intentado que la ejecución sea representativa al comportamiento del método. Se ha utilizado una probabilidad de cruce del 60%, una probabilidad de mutación del 5% en todas las ejecuciones en orden de facilitar la comparación y una población de 100 individuos.

2.1 Inicialización

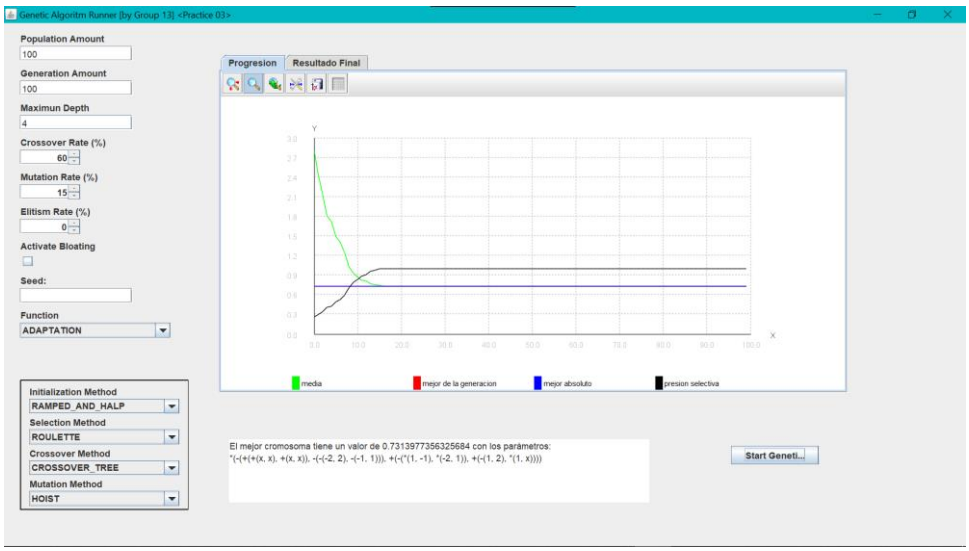
Para las inicializaciones hemos decidido utilizar el método de mutación *hoist*.

FULL



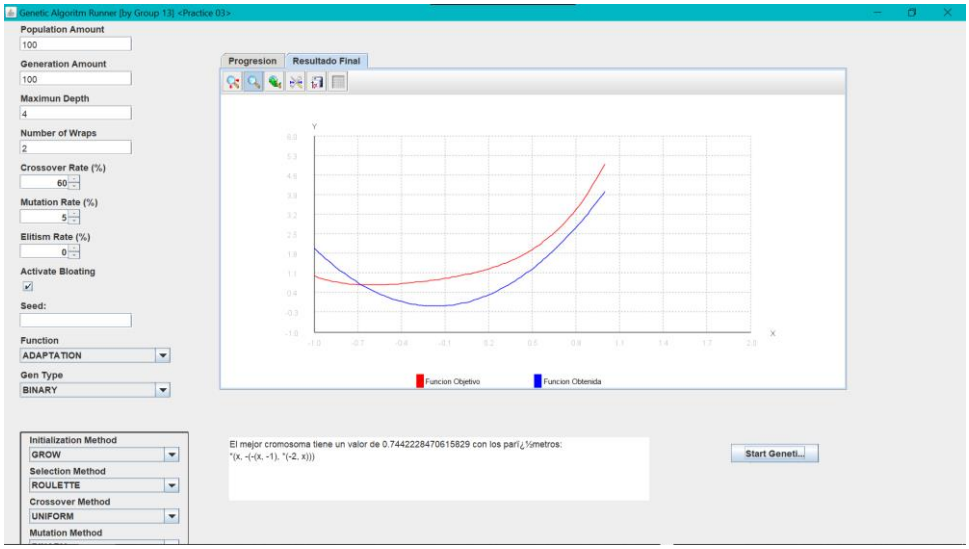
Inicialización Full

Ramped And Half



Inicialización Ramped & Half

Grow

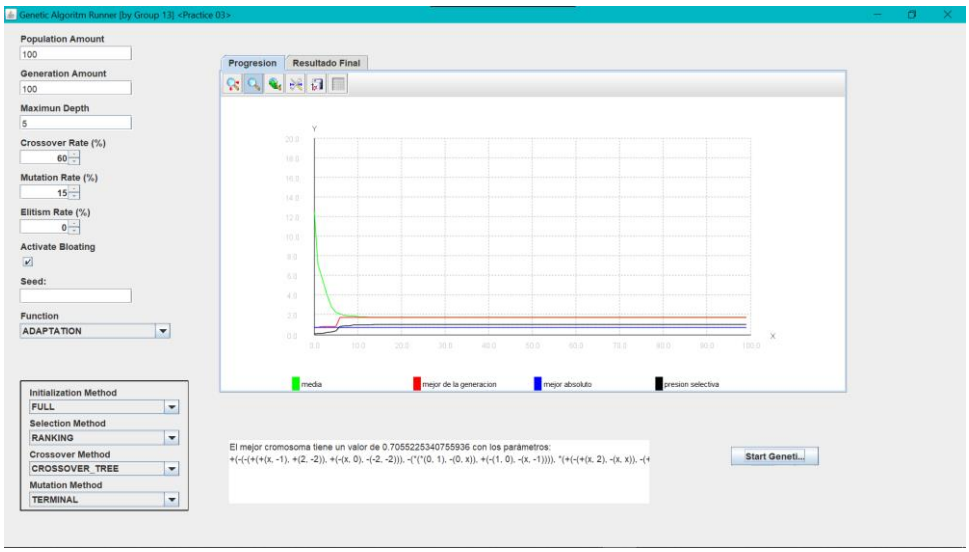


Inicialización Grow

2.1 Mutaciones

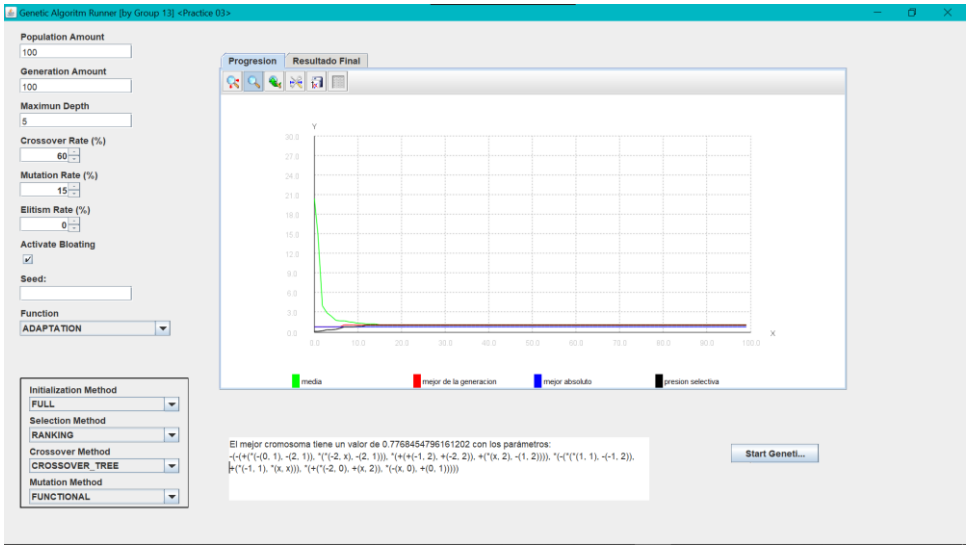
Para las mutaciones hemos utilizado al método de full como constante.

Terminal



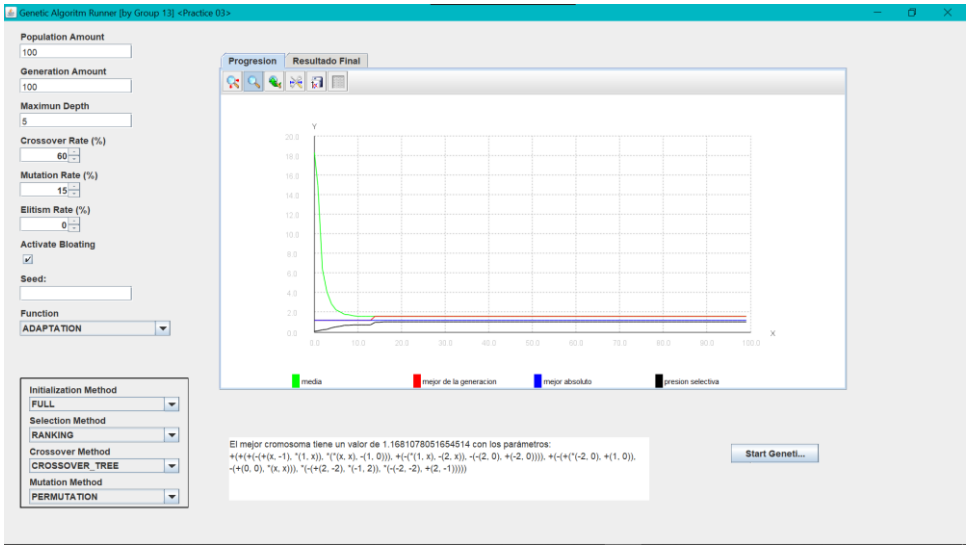
Terminal

Functional



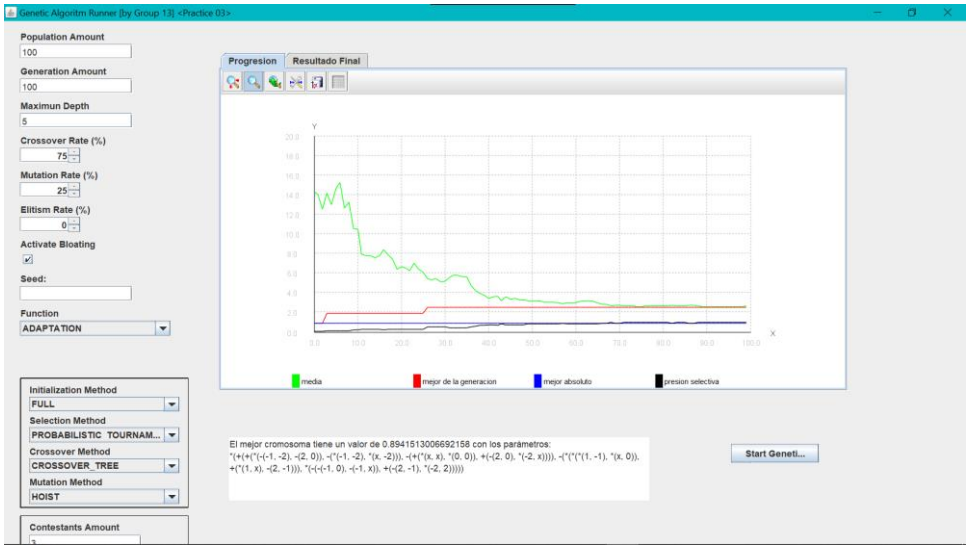
Intercambio

Inversión



Inversión

Hoist



Hoist

4. Bloating

Hemos utilizado el método de bloating de penalización bien fundamente para asegurarnos de penalizar correctamente los árboles extremadamente largos. De esta forma, se ha evitado la construcción de ramas inútiles, mejorando la eficacia del algoritmo pese a que las soluciones finales apenas hayan variado.

Además, esta técnica mejora la solución del algoritmo dado que se disminuye de forma natural la cantidad de intrones en la programación genética.

5. ¿Qué métodos de inicialización y mutación son mejores?

Tras varias pruebas probando los distintas inicializaciones, cruces y mutaciones tanto de la parte obligatoria como la gramática evolutiva, los mejores resultados se obtuvieron con las inicializaciones **Full** y **Ramped And Half**. Mientras que no ha habido un cruce que haya destacado, por el lado de las mutaciones, las permutaciones o hoist han permitido obtener una curva más parecida a la ideal

6. Gramática evolutiva

El flujo de información de la parte de **Gramática Evolutiva** sería el siguiente: Al principio se inicializan los genes (cada uno codifica un codón) y se evalúan, primero traduciendo los genes a un *ArithmeticNode* mediante *TreeBuilder*, que a través del método *evaluate()* del padre de *CodonChromosome* (portador de los genes que representan a los codones), *TreeChromosome se obtiene el fitness.*

La gramática que hemos diseñado es la siguiente (BNF):

<op> ::= |*| or |+| or |-|

<sym> ::= |-2| or |-1| or |0| or |1| or |2| or |x|

<exp> ::= <op>(<exp>, <exp>) or <sym>

La fórmula que hemos implementado para la longitud de los codones es *(Math.pow(2, maxHeight - 1) - 1) * 2 + Math.pow(2, maxHeight - 1)*. Al utilizar wrapping la longitud real se dividirá por el número de *wraps*.

Ejecución de la gramática evolutiva (GenType = BINARY) con el método de crossover Uniforme

*Nótese que cuando está seleccionado GenType = Binary, el método de inicialización seleccionado **no aplica,** sino que se utilizará un conjunto de genes que codifican los codones aleatorios.*

7. Problemas / Curiosidades encontrados

Para mejorar la reutilización de codones mediante el wrapping que de forma normal se leería **EOEOESESEOESES** donde E es la regla de la expresión que diferencia entre nodo nudo o nodo hoja, O es la operación del nodo nudo y S es el símbolo del nodo hoja, hemos decidido aplicar un **retardo** cuando se realiza una nueva lectura del mismo codón (técnica del *wrapping*). Este retardo se aplica para evitar una excesiva similitud entre la primera lectura y las segunda, de la segunda a la tercera, etc. La idea es que si cuando aplicamos una segunda lectura el primer codón se lee como E se replicará todo el árbol anterior como subárbol de sí mismo, pero al aplicar este retardo evitamos está redundancia ya que se tenderá a leer siempre de forma diferente. El retardo consiste en saltarse un codón cuando se itera, de forma que en la vuelta 1 no se salte ningún codón en la vuelta 2 se salte 2 codones, en la vuelta 3, 3 codones.... De esta forma con una solución muy simple, evitamos que siempre se lea un codón para la misma regla de producción.

8. Reparto de tareas

Rafael Alonso: Métodos de cruce y mutación, bloating, cambios en la vista, restructuración de los métodos de mutación uniforme y monopunto para la parte de gramática evolutiva.

Andrés Cardenal: Métodos de inicialización, restructuración de métodos de mutación, gramática evolutiva incluida traducción de binario a árbol y el gen correspondiente, rediseño de builders.

Ambos (laboratorio y google meet): ArithmeticNode, método de inicialización de grow, diseño del wrapping.