PROGRAMACIÓN EVOLUTIVA

Universidad Complutense de Madrid

Ingeniería del Software e Inteligencia Artificial

Pablo Villapún Martín

Sandra Mondragón Lázaro

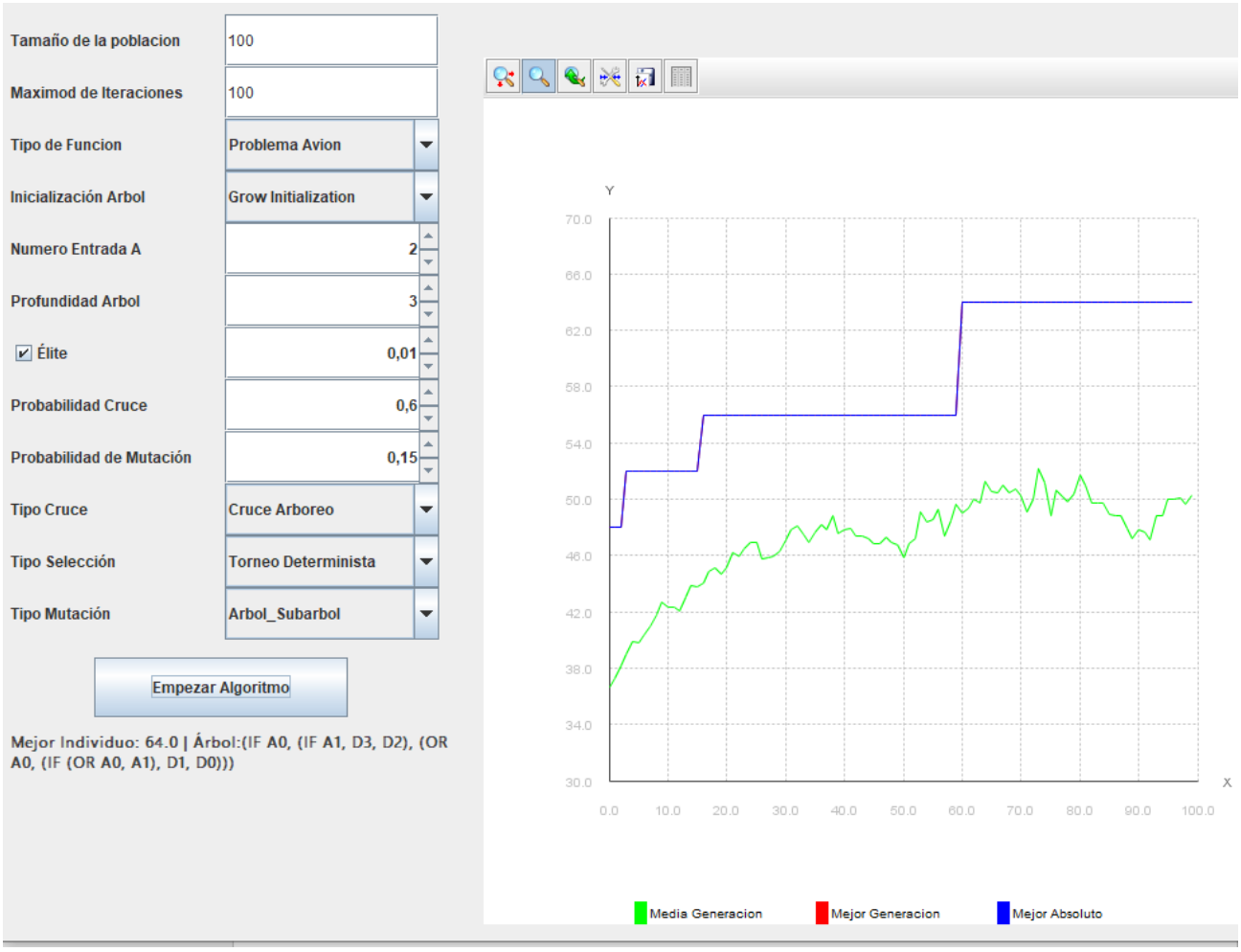
Programación genética con árboles

En esta práctica se ha realizado un multiplexor ayudándonos de la programación genética.

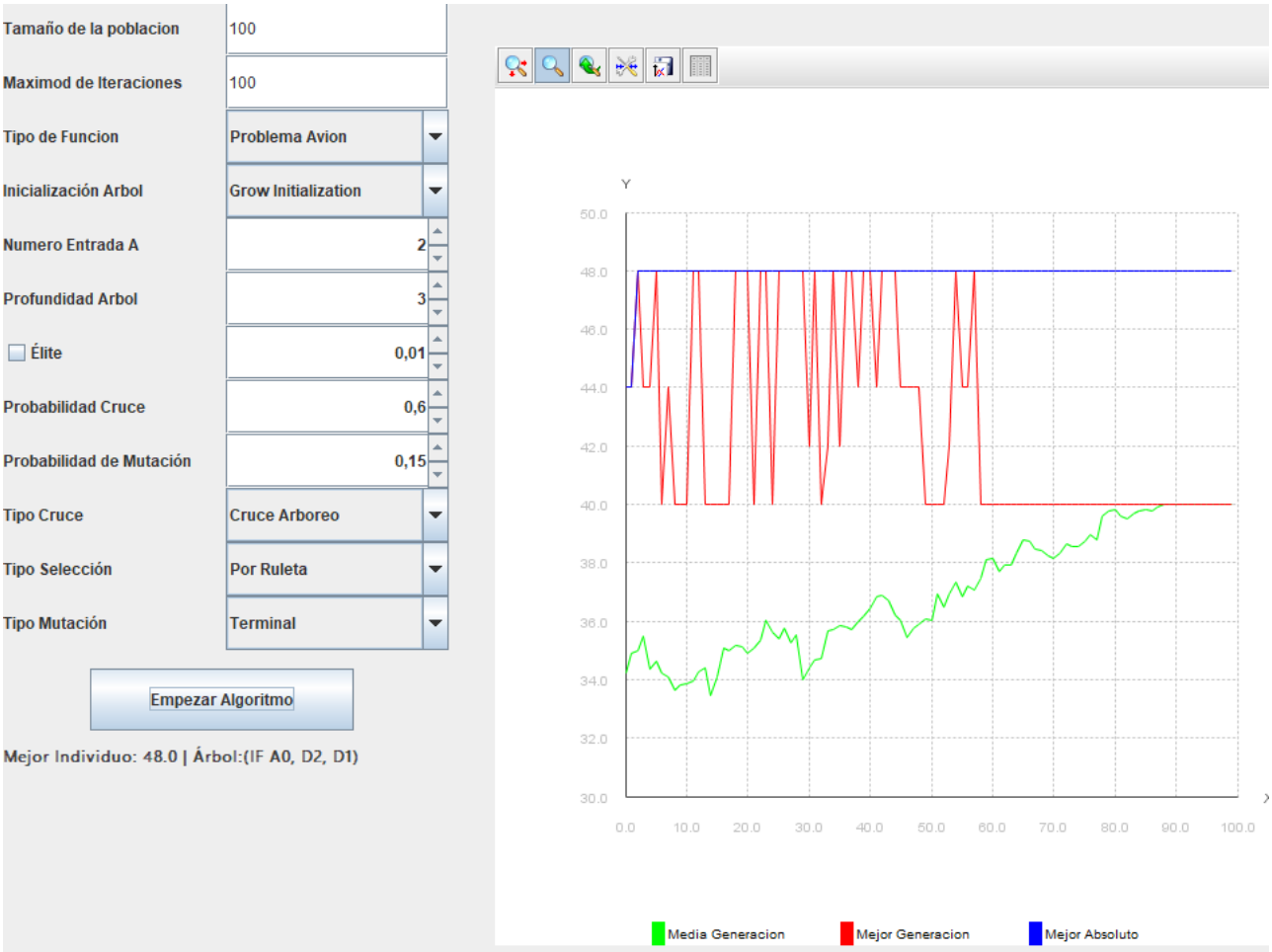
Parte 1

El mejor individuo (64.0) encontramos que se puede hallar con el Torneo Determinista o con Truncamiento como operador de Selección y con una mutación árbol-subárbol.

La media de ejecuciones es de 60.52 probando con distintos operadores de selección o de mutación.



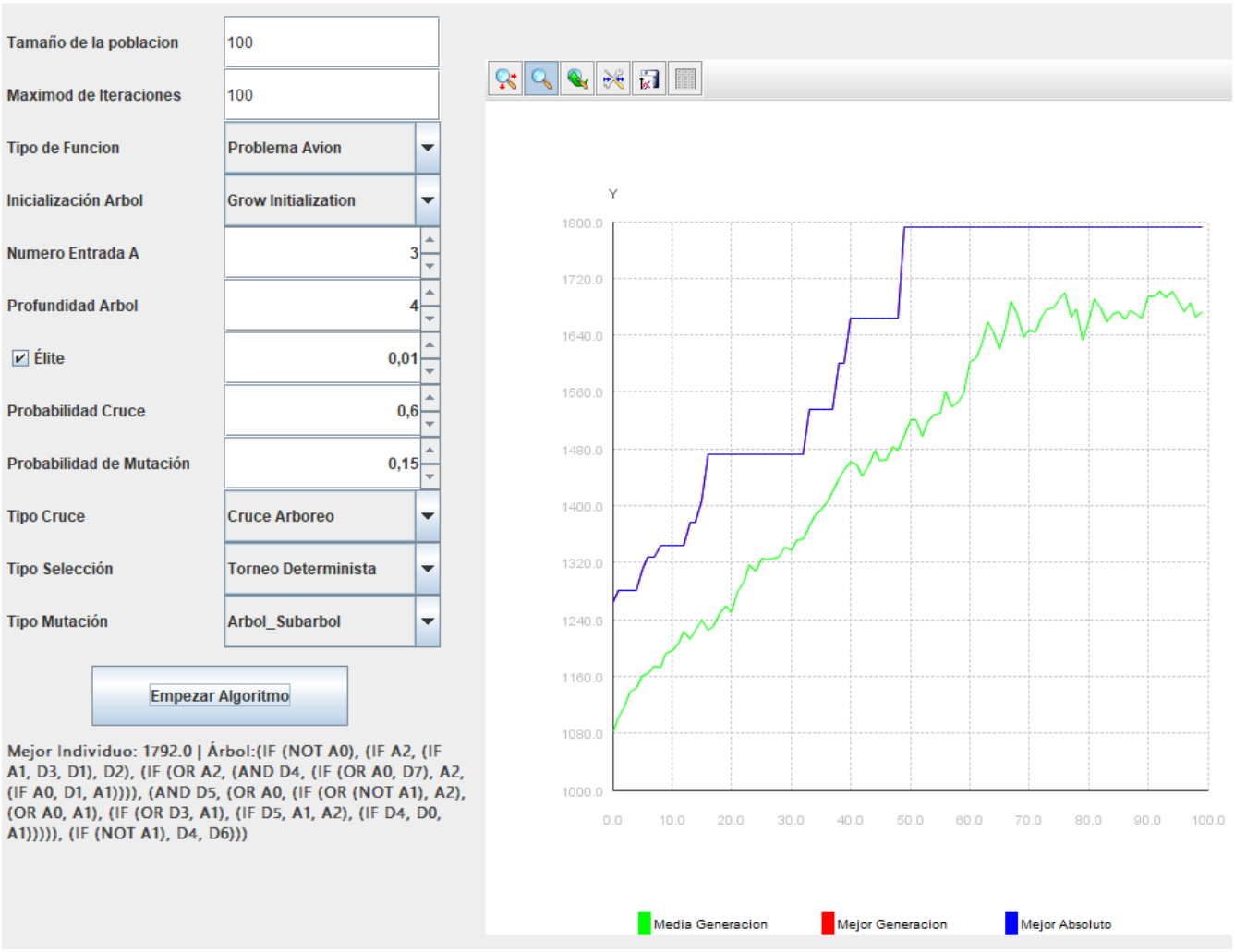
El peor individuo hallado es de 48.0 con los métodos de mutación Terminal y Hoist dado que son destructivos y terminan eliminando muchas ramas debido al bloating (el cual poda las ramas que sobrepasan cierta altura).



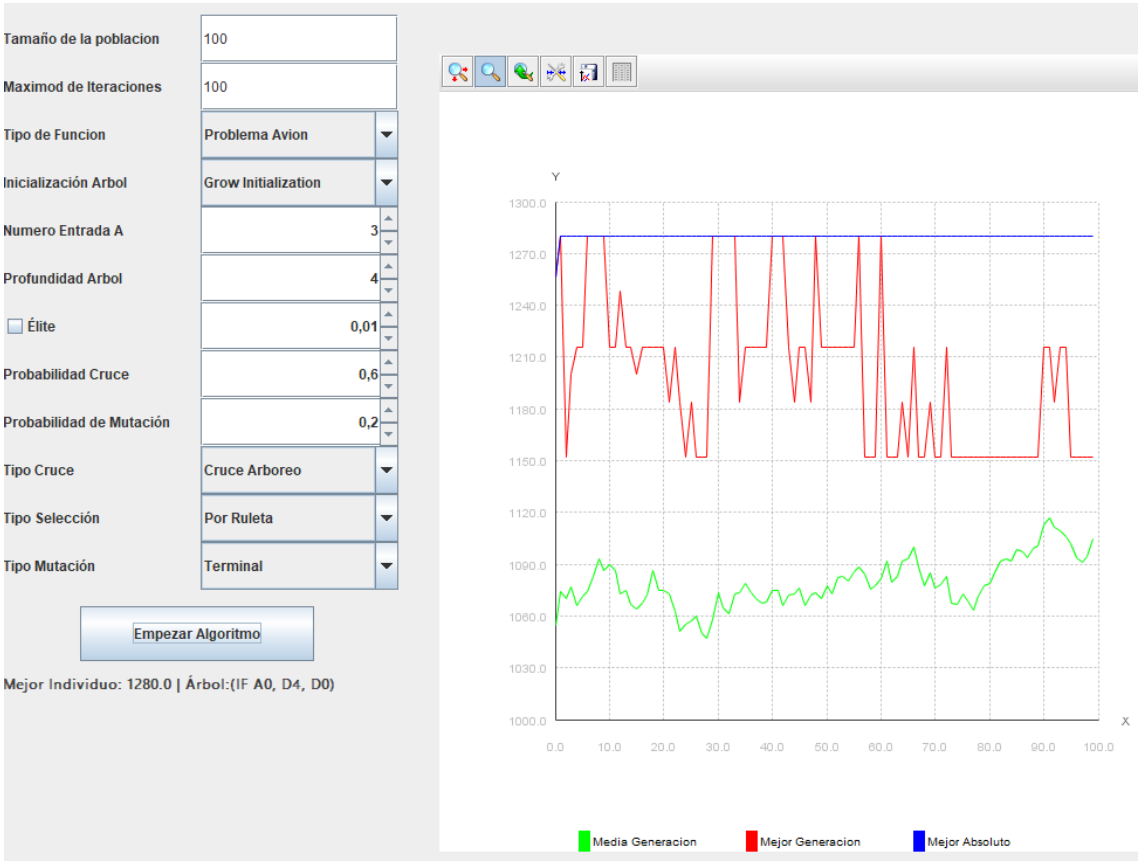
Parte 2

El mejor individuo (1792.0) encontramos que se puede hallar con el Torneo Determinista o con Truncamiento como operador de Selección y con mutación árbol-subárbol. La profundidad del árbol ha de ser mayor para este ejemplo (de 4 a 5). Usar élite mejora siempre el resultado obtenido.

La media de ejecuciones es de 1500 aproximadamente probando con distintos operadores de selección o de mutación.



El peor individuo hallado es de 1280.0 con los métodos de mutación Terminal y Hoist dado que son destructivas.



CONCLUSIONES DE LA PRÁCTICA

- Los métodos de mutación terminal y hoist no tienen buena compatibilidad con el control de bloating de poda, ya que a lo largo de las ejecuciones los individuos acaban con árboles con menos ramas y por tanto el espacio de búsqueda se reduce.
- Las mutaciones de árbol-subárbol y expansión son las que mejores resultados obtienen porque nunca cortan o destruyen ramas de los árboles del individuo.
- La mejor inicialización de árboles es ramp and half (se encuentra el óptimo en generaciones más tempranas y favorece la diversidad de individuos).
- La parte 2 del problema (2048 entradas en la tabla multiplexor) es muy costosa computacionalmente debido a que se requiere una profundidad de árbol de 4 o 5 para un resultado bueno y el proceso de evaluación consume el grueso de la ejecución dada la densidad de ramas e inputs a evaluar.
- Usar élite mejora considerablemente los resultados en ambas partes de la práctica.
- Las estructuras arbóreas usadas como cromosomas son fáciles de expandir en cuanto a diseño, ya que si se quiere añadir un nodo función o de entrada sólo hay que implementar la interfaz de Nodo y añadirlo como nodo disponible en la generación aleatoria de los árboles.

DETALLES DE IMPLEMENTACIÓN

algoritmoGenetico.cruces

Aquí encontramos todos los cruces dados. Partimos de una clase padre **Cruce** que tiene los métodos de cruzar y **buscarIndividuo** (este último sirve para buscar un individuo que no haya sido cruzado)

algoritmoGenetico.trees

Aquí encontramos las clases padre de **Node** de las cuales heredan los distintos tipos de nodos: **input** para las entradas de la tabla y **function** para las operaciones entre nodos. Entre los métodos que contienen se encuentran **evalue** (que devuelve el valor actual del nodo de forma recursiva) y distintos setters y getters. Además, la clase Node guarda la información referente a su nodo padre y una lista de los hijos que tiene..

En este paquete también encontramos la clase **Tree**, la cual genera un árbol con distintas formas de inicialización (full y grow, ramped & half se realiza fuera en algoritmoGenetico.java). Esta clase es la encargada de generar ramas aleatorias, terminales aleatorios respetando siempre la altura máxima.

algoritmoGenetico.tablaMultiplexor

Aquí encontramos la clase *singleton* **TablaMultiplexor** encargada de la creación de la tabla que contiene los casos de prueba. Esta tabla se realiza calculando las distintas permutaciones posibles de los valores.

algoritmoGenetico.individuos

Aquí encontramos el **IndividuoArboreo.** Este individuo es el contiene el árbol que define el comportamiento del individuo. En su método **getValor** se calcula el fitness evaluando el árbol y comparándolo con cada una de las soluciones de los casos de prueba. Se calcula el **número de aciertos**.

algoritmoGenetico

Aquí encontramos la clase de **AlgortimoGenético**, encargada del bucle principal del algoritmo. Contiene **métodos para la inicialización y configuración** del algoritmo además de **métodos de evaluación** que guardan la información para luego ser puesta en la **gráfica**. Aquí también se inicializa la **tabla del multiplexor.**

algoritmoGenetico.mutacion

Aquí encontramos todas las mutaciones dadas. Partimos de una clase padre **Mutacion** que tiene el método de mutar.

algoritmoGenetico.seleccion

Aquí encontramos todas las selecciones dadas. Partimos de una clase padre **Seleccion** que tiene los métodos de seleccionar y **calculaFitness** (este último calcula el fitness de los individuos y los devuelve, en caso de tener negativos, desplazados).

GUI

Aquí encontramos la clase **UIAplication** encargada de generar la ventana de **Jframe** y la interfaz del algoritmo.

REPARTO DE TAREAS

Usando pair-programming fixed de errores, clase de algoritmoGenetico, modificación de interfaz.

- **Sandra Mondragón**: todos los métodos de cruce y el paquete de algoritmoGenetico.Tree
- **Pablo Villapún**: clase TablaMultiplexor, clase IndividuoArboreo y control de Bloating