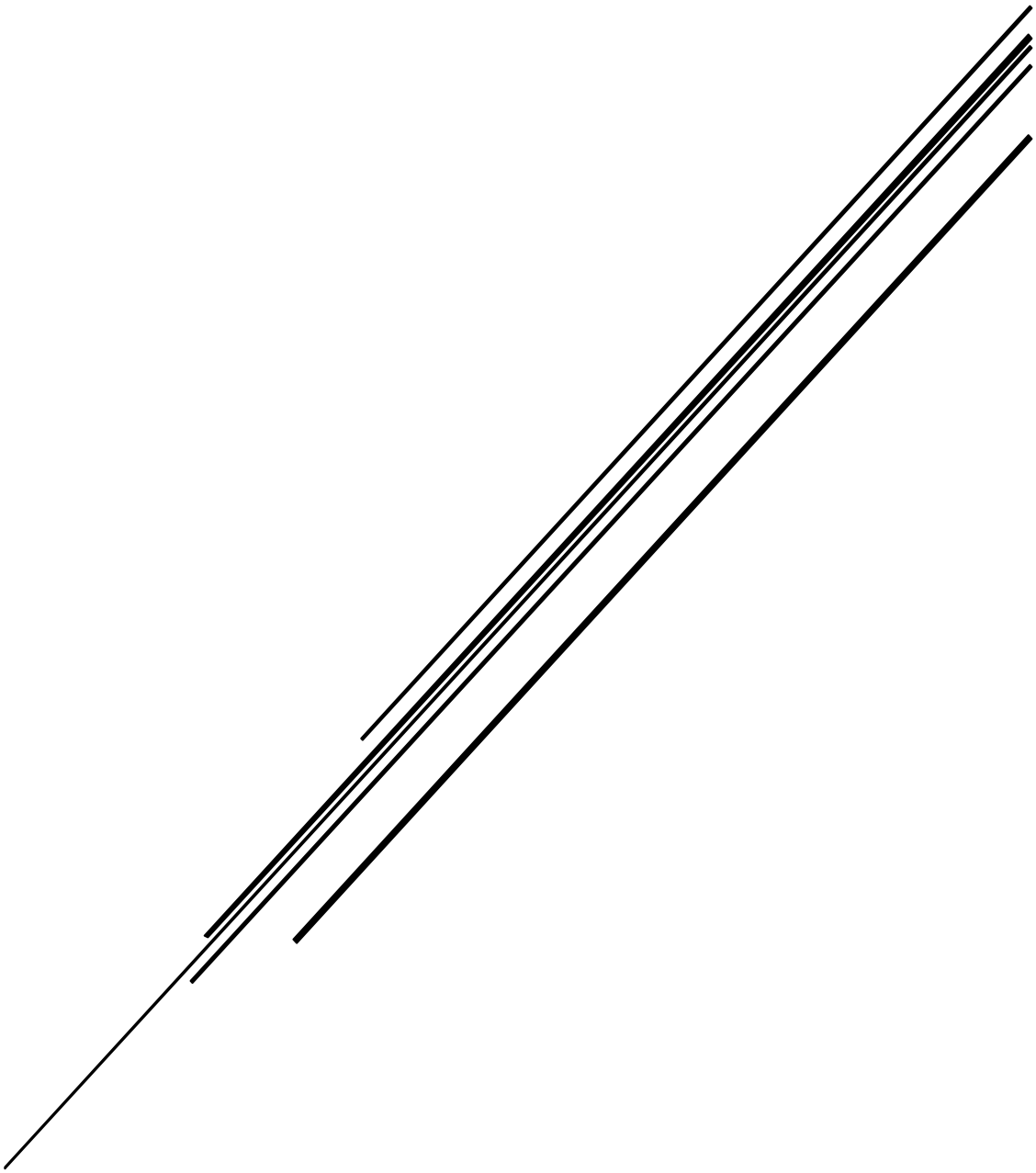
MEMORIA PRÁCTICA I:
“Optimización de funciones”

PROGRAMACIÓN EVOLUTIVA



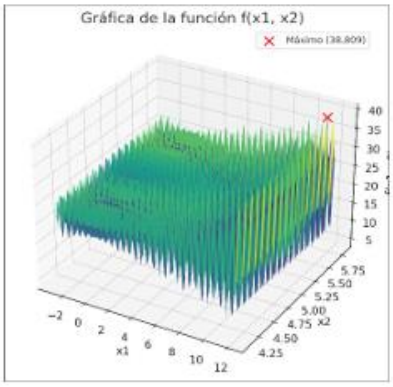
Grado en Ingeniería de Datos e Inteligencia Artificial
Artem Vartanov y Mario López Díaz

ÍNDICE

1. Capturas de resultados

Función 1: calibración y prueba

$$f(x_1, x_2) = 21.5 + x_1.sin(4\pi x_1) + x_2.sin(20\pi x_2)$$



que presenta un **máximo** de **38.809** en $x_1 = 11.625$ y $x_2 = 5.726$

$x_1 \in [-3.0, 12.1]$ $x_2 \in [4.1, 5.8]$

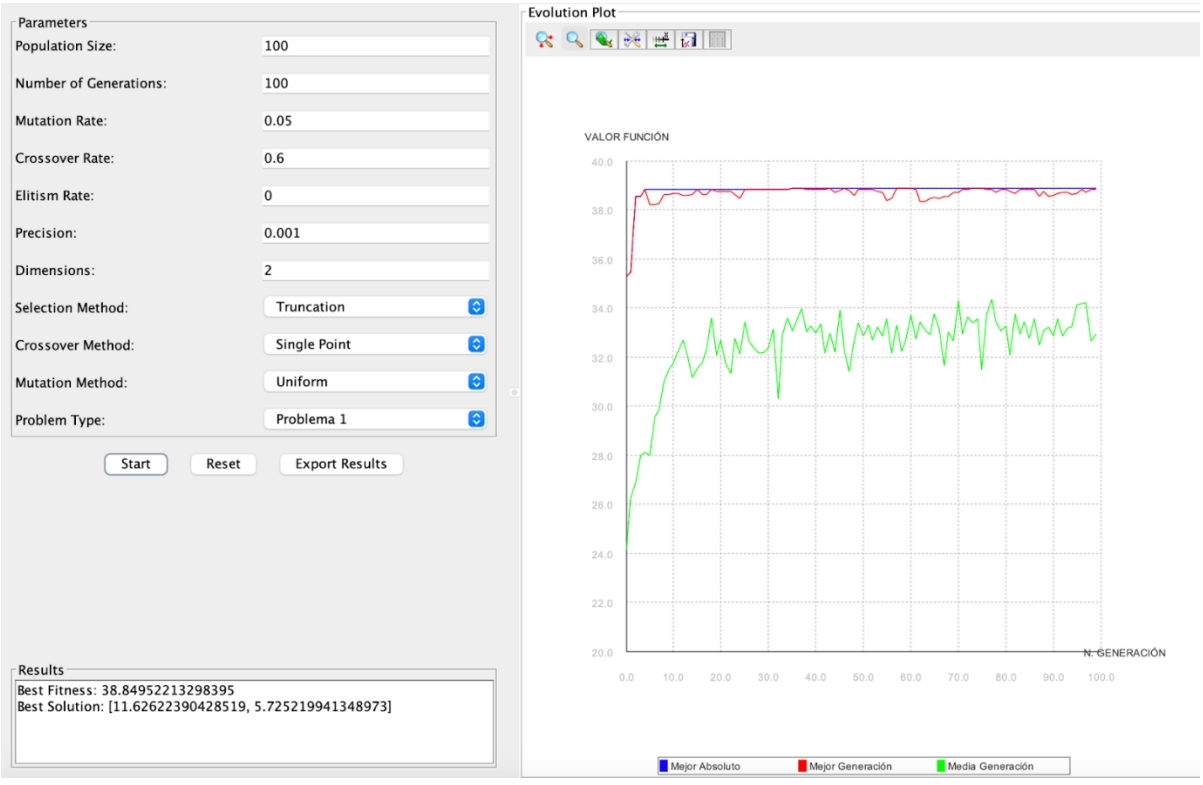


Figura 1.1 Ejecución SIN elitismo y selección por truncamiento y cruce mono punto

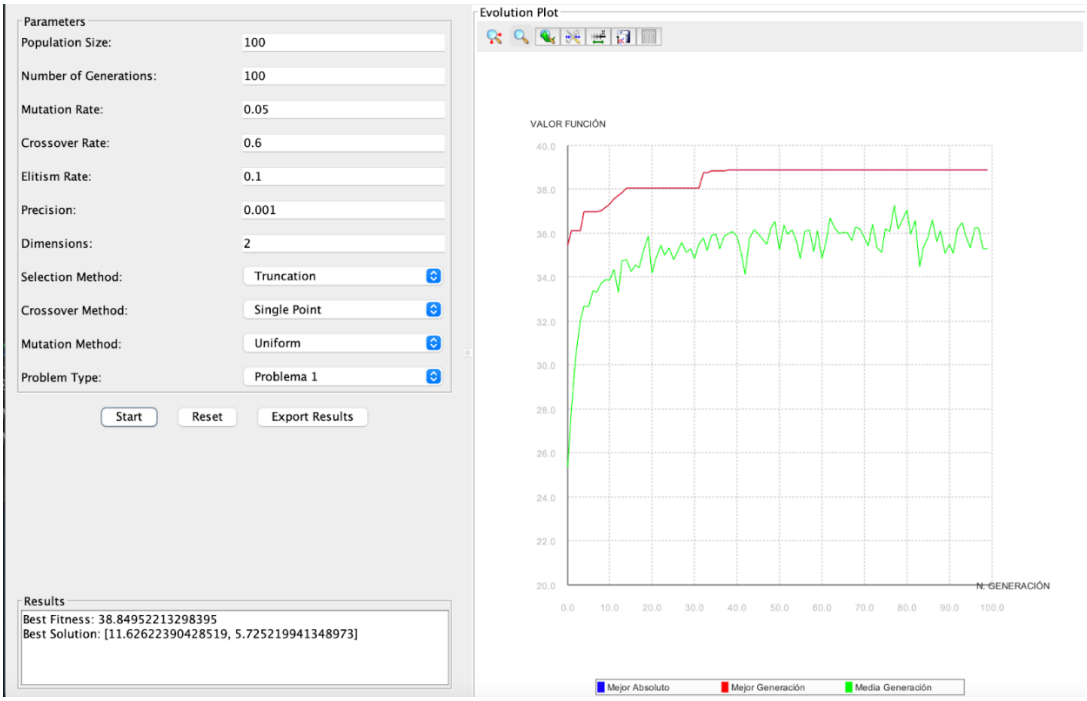
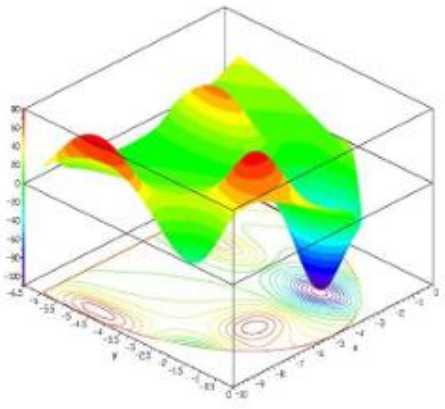


Figura 1.2 Ejecución CON elitismo (10%), selección truncamiento y cruce mono punto

Función 2: Mishra Bird

$$f(x_1,x_2) = \sin(x_2)\exp(1-\cos(x_1))^2 + \cos(x_1)\exp(1-\sin(x_2))^2 + (x_1-x_2)^2$$



$x_1 \in [-10, 0]$

$x_2 \in [-6.5, 0]$

que presenta un mínimo global de **−106.7645367**

en $x^* =$ **−3.1302468, −1.5821422**

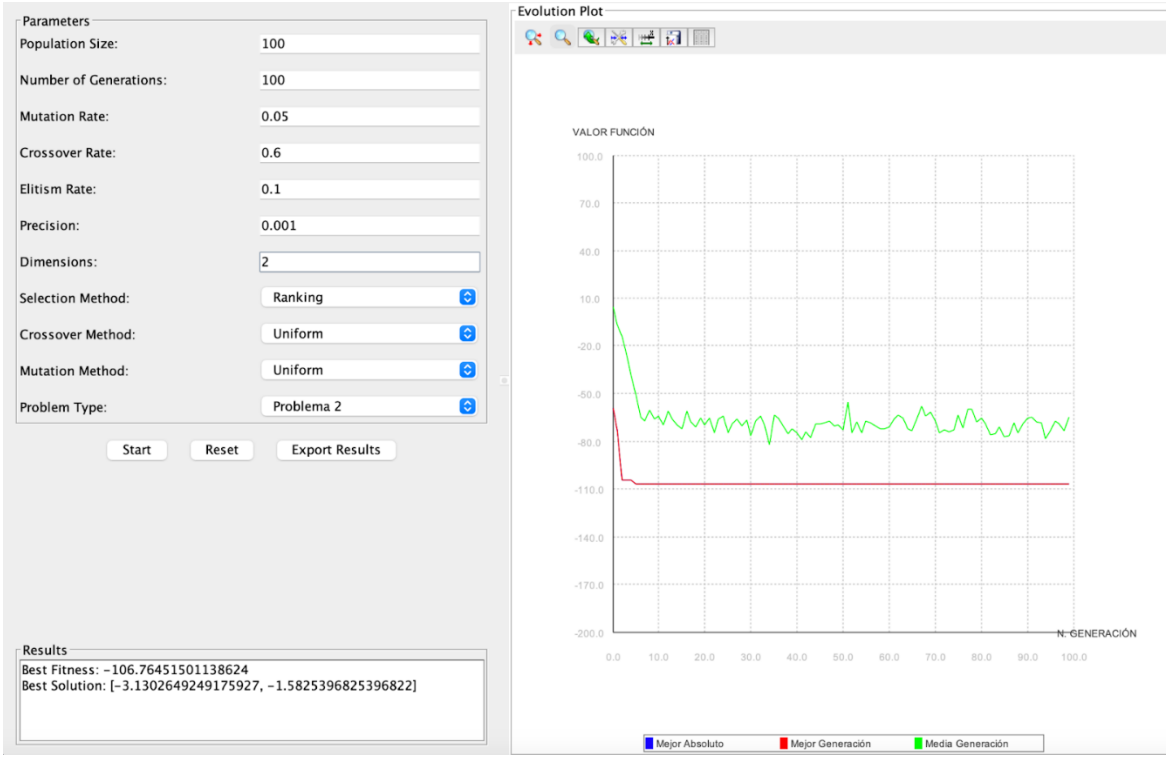
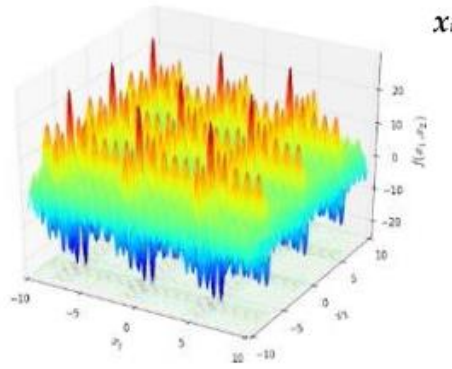


Figura 1.3 Ejecución CON elitismo (10%), selección por ranking y cruce uniforme

Función 3: Schubert

$$f(x_1,x_2) = \left(\sum_{i=1}^{5} i\cos((i+1)x_1 + i)\right)\left(\sum_{i=1}^{5} i\cos((i+1)x_2 + i)\right)$$

$x_i \in [-10,10]$ que presenta 18 mínimos de **-186.7309**



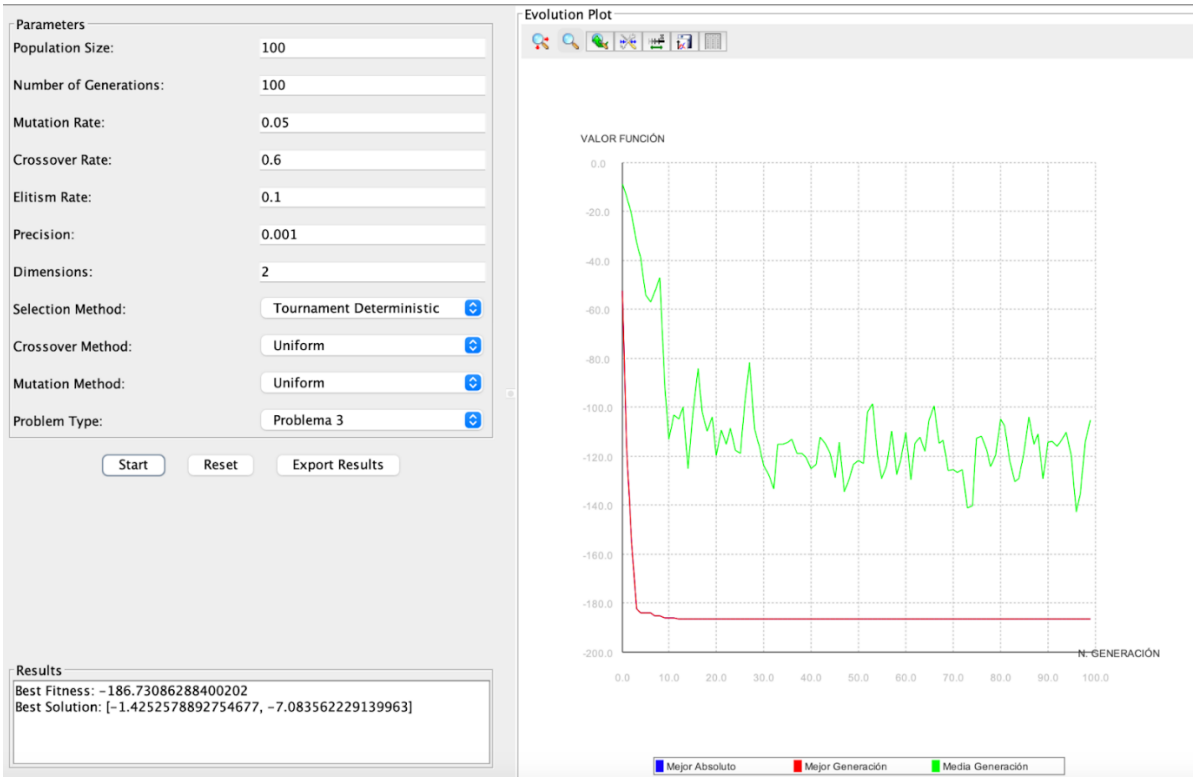


Figura 1.4 Ejecución CON elitismo (10%), torneo determinista y cruce uniforme

Función 4: Michalewicz con codificación binaria

$$f(\mathbf{x}) = -\sum_{i=1}^{d} \sin(x_i)\sin^{2m}\left(\frac{ix_i^2}{\pi}\right)$$

$$x_i \in [0, \pi] \quad m = 10$$

que presenta los siguientes mínimos en función de d:

d= 2 f(x*) = - 1.8013 en x* = (2.20, 1.57)

d = 5 f(x*) = - 4.6876

d = 10 f(x*) = - 9.6601

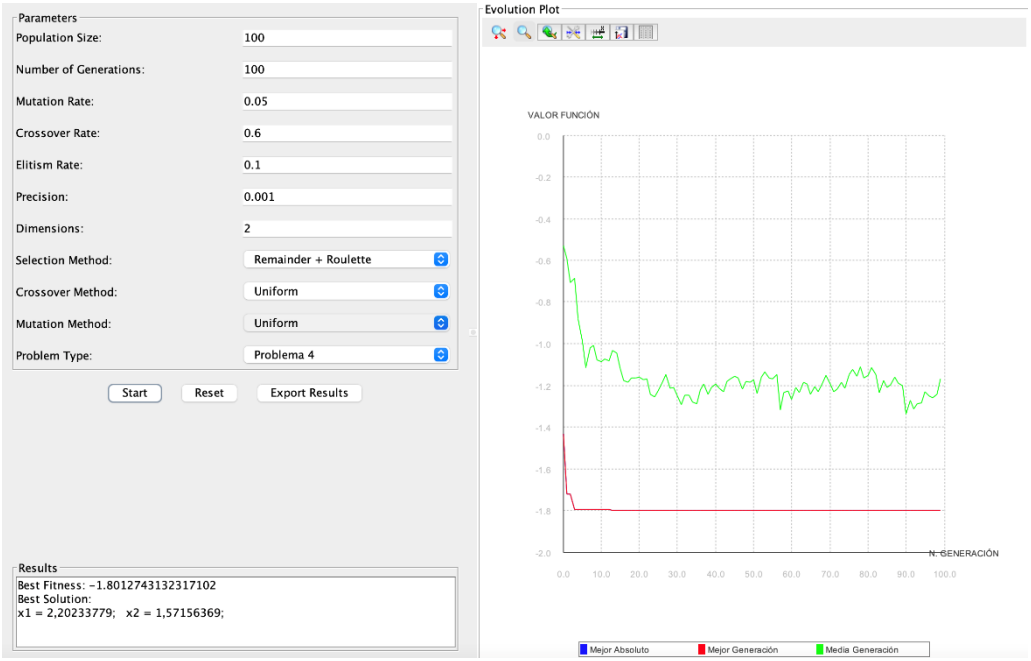


Figura 1.5 Ejecución (d = 2) CON elitismo (10%), selección por restos, cruce uniforme

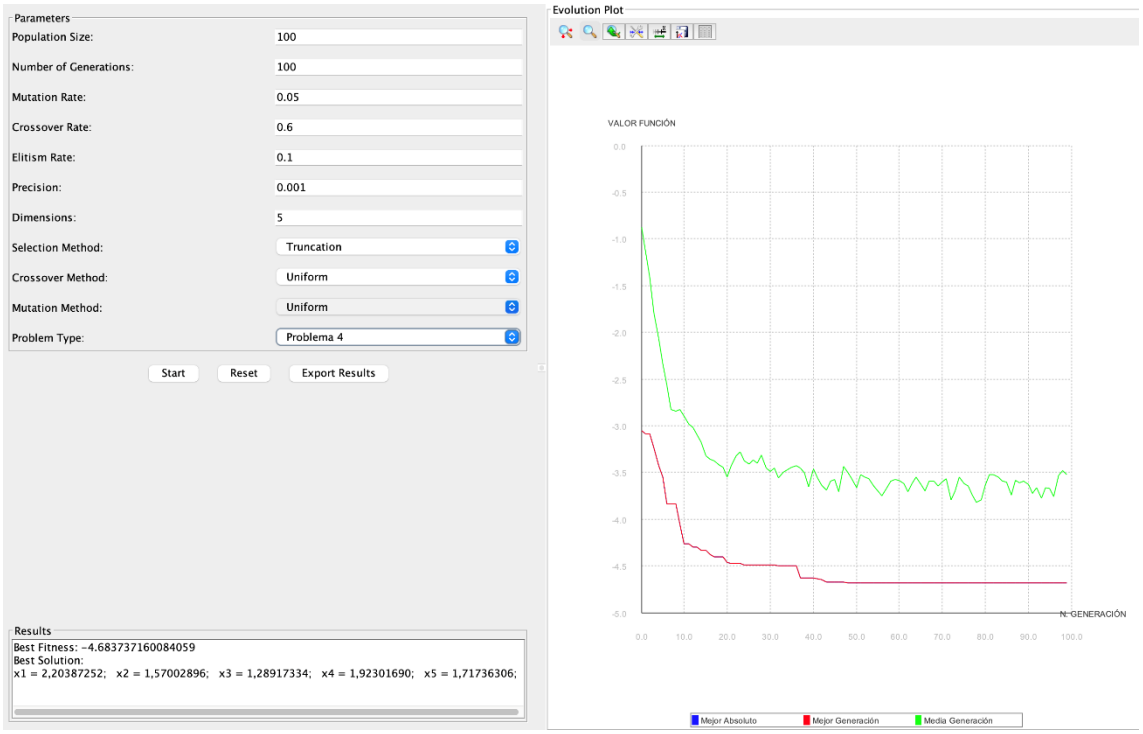


Figura 1.6 Ejecución (d = 5) CON elitismo (10%), truncamiento, cruce uniforme

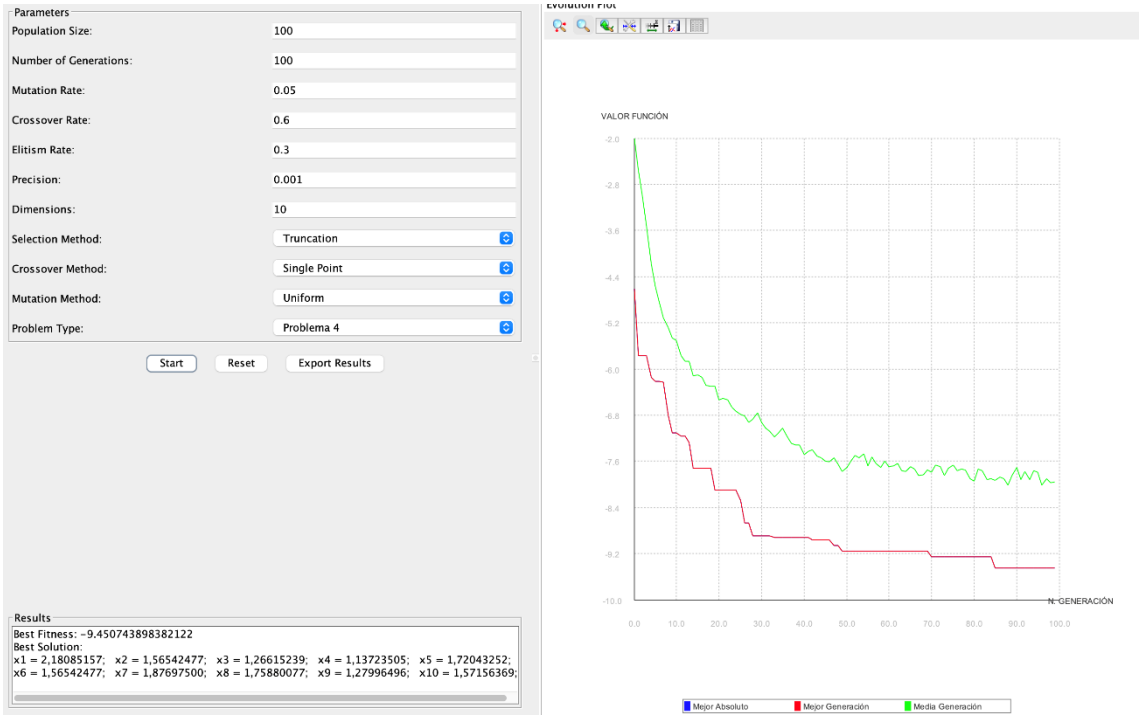


Figura 1.7 Ejecución (d = 10) CON elitismo (30%), truncamiento, cruce mono punto

Función 5: Michalewicz con codificación reaL

Igual que la anterior, pero en este caso utilizaremos individuos con codificación real. Ahora el cromosoma está formado por genes con números reales. La dimensión o número de variables a utilizar ***d*** será un valor seleccionable en la interfaz. Por ejemplo, para d = **5** variables, el cromosoma podría ser:

3.1241	2.7112	2.3454	0.3425	1.6832
X1	X2	X3	X4	X5

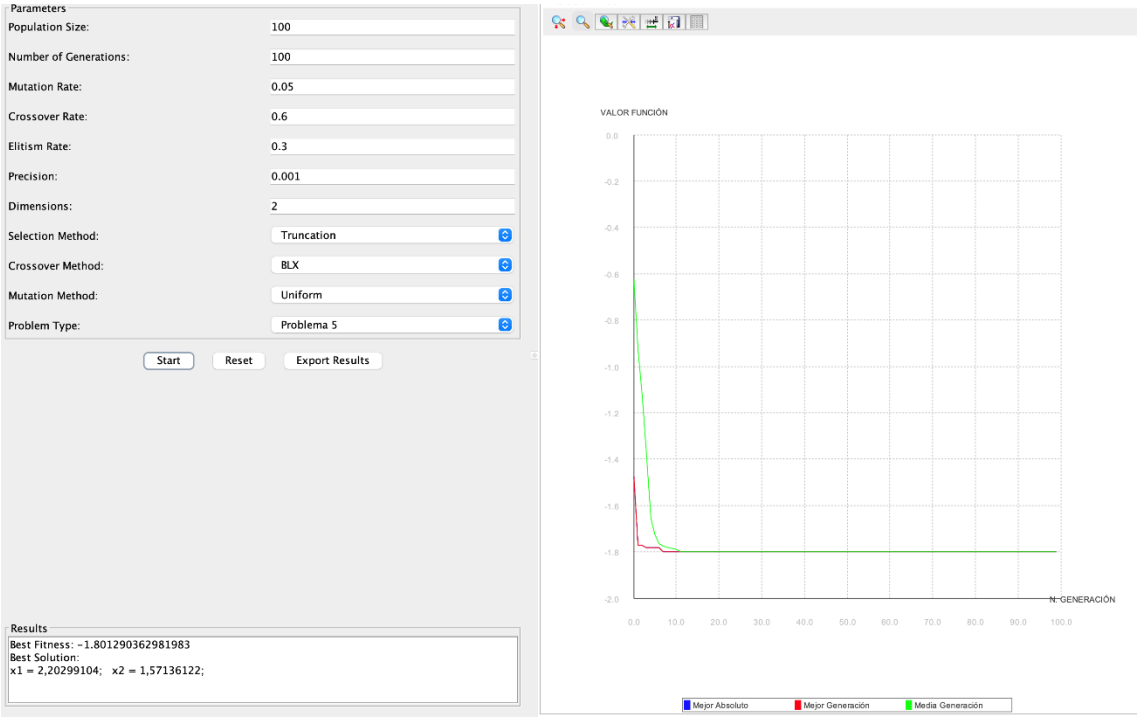


Figura 1.8 Ejecución (d = 2) CON elitismo (30%), truncamiento, cruce BLX

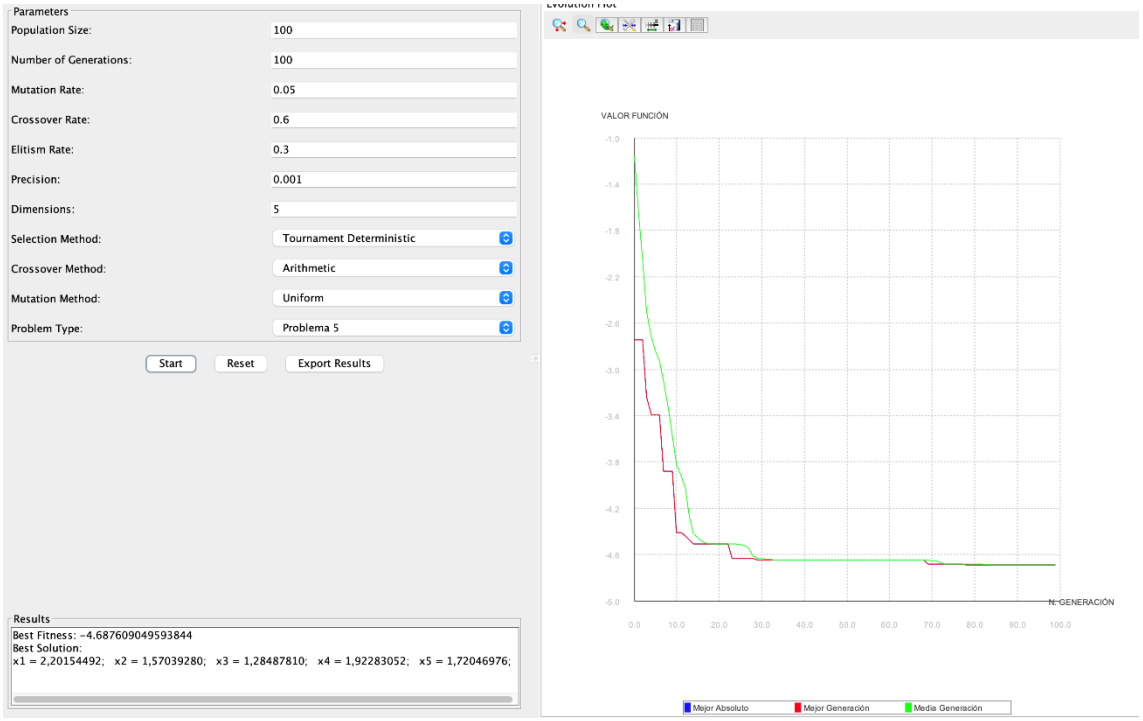


Figura 1.9 Ejecución (d = 5) CON elitismo (30%), torneo determinista, cruce aritmético

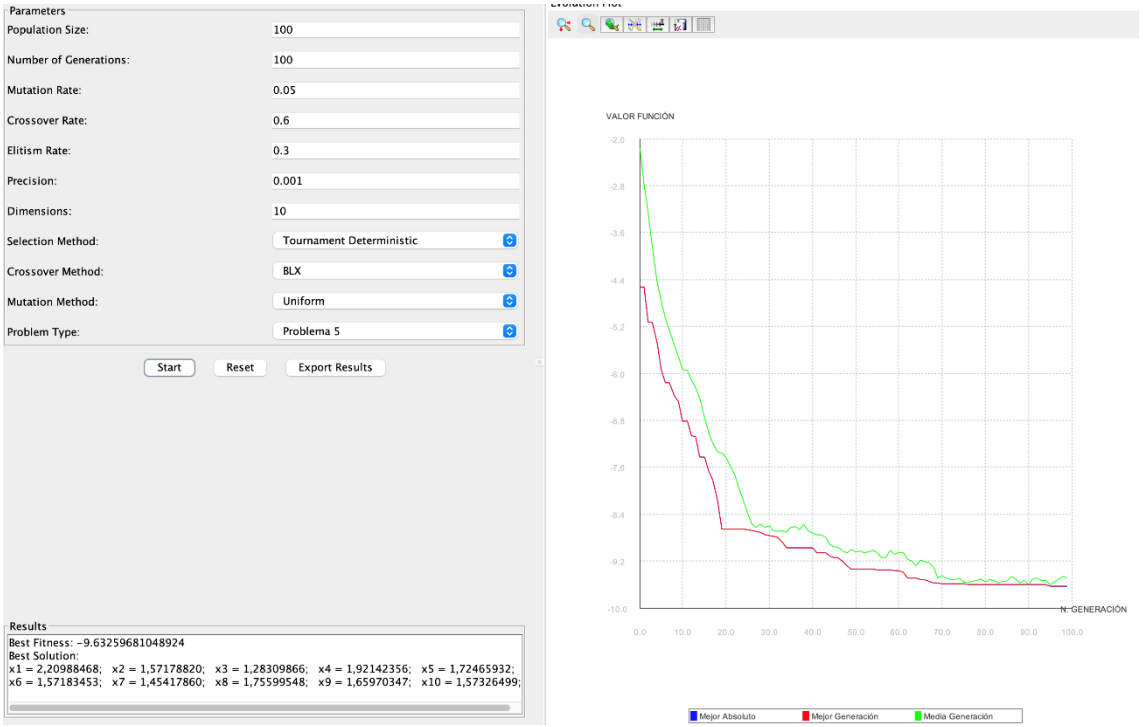


Figura 1.10 Ejecución (d = 10) CON elitismo (30%), torneo determinista, cruce BLX

2. Observaciones de la práctica

Al analizar las gráficas se puede ver que nos hemos acercado bastante a los óptimos globales, aconsejando un correcto funcionamiento del algoritmo. Hemos utilizado el mismo número de generaciones y tamaño de población para que la comparación sea más justa. En los últimos 2 apartados para la dimensión 10 la línea no parece converger, mientras que en los otros apartados converge bastante más rápido, como en el apartado 2.

El elitismo guarda los mejores individuos, y luego los reinserta en la población transformada, reemplazando los individuos con el peor fitness. En las gráficas en casi todas usamos elitismo al 10%, excepto en los últimos 2 apartados (misma función), donde el porcentaje lo hemos subido al 30% para conseguir mejores resultados. Es por ello porque en las gráficas con menos elitismo las líneas rojas y verdes guardan mayor distancia, que en gráficas con más elitismo.

3. Arquitectura del código

1. Paquete es.ucm

Este paquete contiene las clases principales que gestionan la ejecución del algoritmo genético y la interfaz gráfica.

Clase AlgoritmoGenetico: Gestiona la ejecución del algoritmo genético.

Clase Main: Implementa la interfaz gráfica para configurar y ejecutar el algoritmo genético pasándole los parámetros y sacando de él los resultados para mostrarlos en la gráfica.

2. Paquete es.ucm.factories

Este paquete contiene las fábricas de individuos, que se utilizan para crear individuos de diferentes tipos (según la función a optimizar).

Clase IndividuoFactory: Define una interfaz para crear individuos.

Clases Individuo1Factory, Individuo2Factory, etc.: Implementan fábricas específicas para cada tipo de individuo (función a optimizar).

3. Paquete es.ucm.individuos

Este paquete contiene las clases que representan a los individuos de la población. (cada problema tiene asignado un tipo de individuo, ej. Individuo1 para problema 1)

Clase Individuo: Representa a un individuo de la población.

4. Paquete es.ucm.genes

Este paquete contiene las clases que representan los genes de los individuos.

Clase Gen: Define una interfaz para los genes.

Clase BooleanGen: Representa un gen cuyo genotipo es binario.

Clase RealGen: Representa un gen cuyo genotipo es real

5. Paquete es.ucm.selection

Este paquete contiene las clases que implementan los métodos de selección de individuos.

Clase AbstractSelection: Define una interfaz para los métodos de selección.

Clases RouletteSelection, TorneoSelection, etc.: Implementan métodos de selección específicos (ruleta, torneo, truncamiento, etc.)

6. Paquete es.ucm.cross

Este paquete contiene las clases que implementan los métodos de cruce.

Clase AbstractCross: Define una interfaz para los métodos de cruce.

Clases SinglePointCross, UniformCross, etc.: Implementan métodos de cruce específicos (un punto, uniforme, aritmético, etc.)

7. Paquete es.ucm.mutation

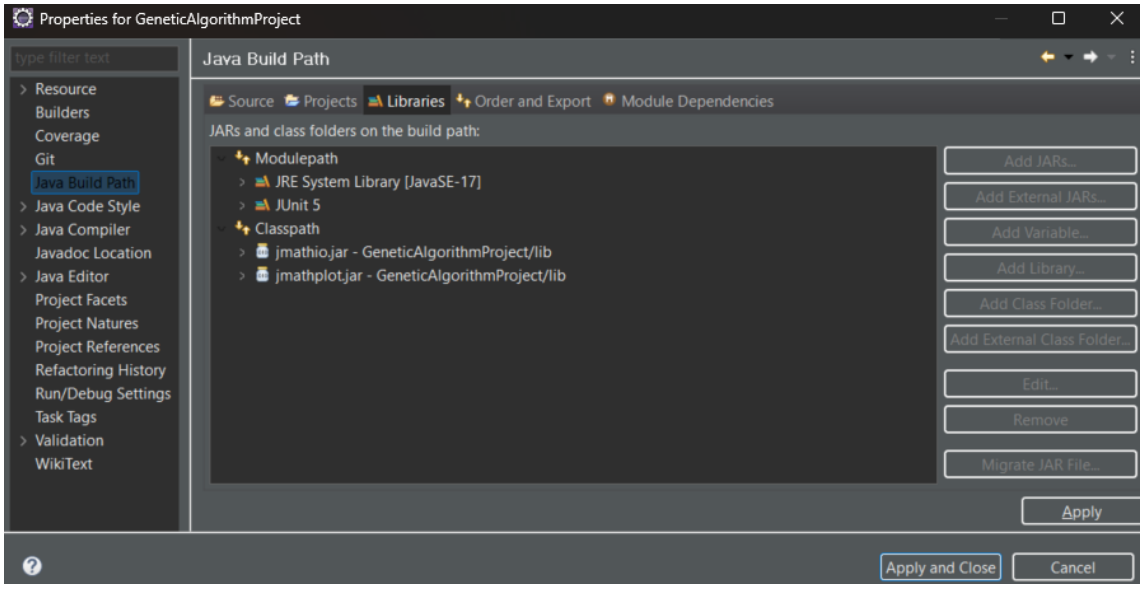
Este paquete contiene las clases que implementan los métodos de mutación.

Clase AbstractMutate: Define una interfaz para los métodos de mutación.

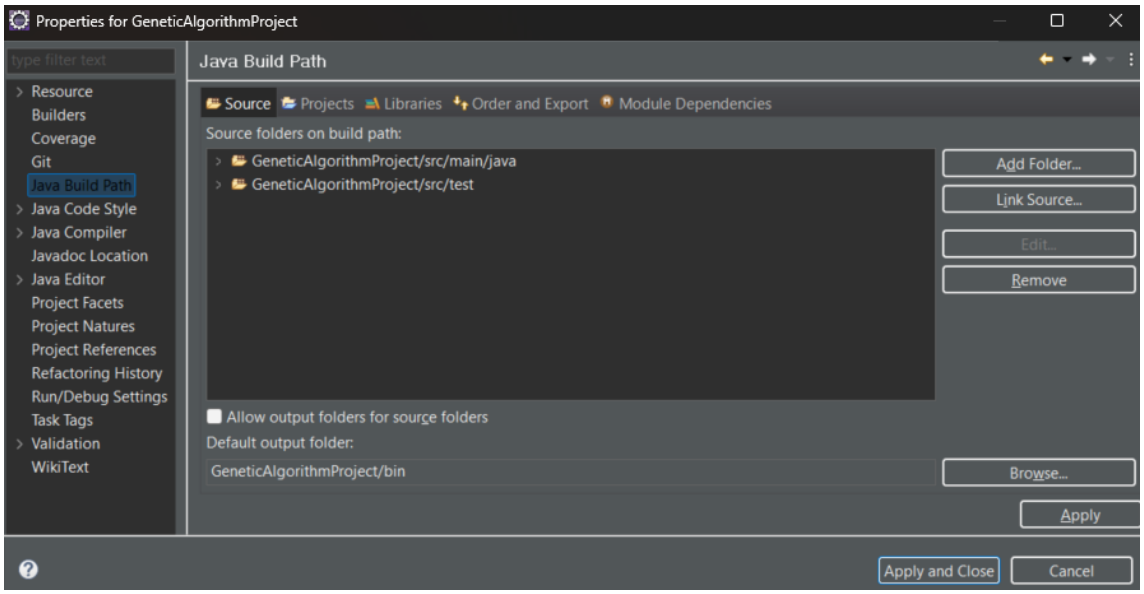
Clase UniformMutate: Implementa la mutación uniforme.

4. Ejecución del proyecto

Para ejecutar el código es necesario importarlo como proyecto de Java en Eclipse. Posteriormente, antes de poder darle al *run,* es necesario asegurarse de que las propiedades del proyecto tienen la siguiente configuración aplicada:



Properties -> Java Build Path -> Libraries -> Archivos .jar



Properties -> Java Build Path -> Source -> src/main/java, src/test

Tras asegurarse de que el proyecto está bien configurado, sólo quedará elegir la configuración del run para poder ejecutarlo, el cual será el archivo Main.java.

5. Distribución del trabajo

Artem:

Ha implementado las estructuras que guardan los valores de las variables (genes reales y genes booleanos), añadiendo métodos para facilitar la mutación y el cruce. Además, ha implementado los métodos de selección y he añadido el elitismo al algoritmo.

Mario:

Ha implementado la interfaz gráfica. Por otro lado, sobre la representación de los genes, ha implementado los métodos de cruce y mutación, y ha utilizado todo ello para implementar el algoritmo genético.

6. Conclusiones

Los algoritmos genéticos han rendido bastante bien en los problemas propuestos. Sin embargo, el problema que tienen es que no garantizan convergencia. Era interesante ver las diferencias entre diferentes métodos de selección y cruce y la aplicación del elitismo, que generalmente superaba los resultados sin elitismo.

Además, otra cosa interesante era la optimización de la función de los ejercicios 4 y 5. Hemos notado que al utilizar la representación real se conseguía mejores resultados, aspecto por tanto importante que podríamos tener en cuenta para futuros problemas de optimización.

Aunque los algoritmos genéticos requieren ajustar algunos parámetros y pueden ser lentos, su capacidad para explorar muchas soluciones y evitar quedarse atascado en soluciones no óptimas los hace muy aplicados.

En esta práctica, hemos visto cómo funcionan y cómo se aplican a problemas reales, confirmando que son una opción efectiva para encontrar buenas soluciones.