Memoria Práctica 2

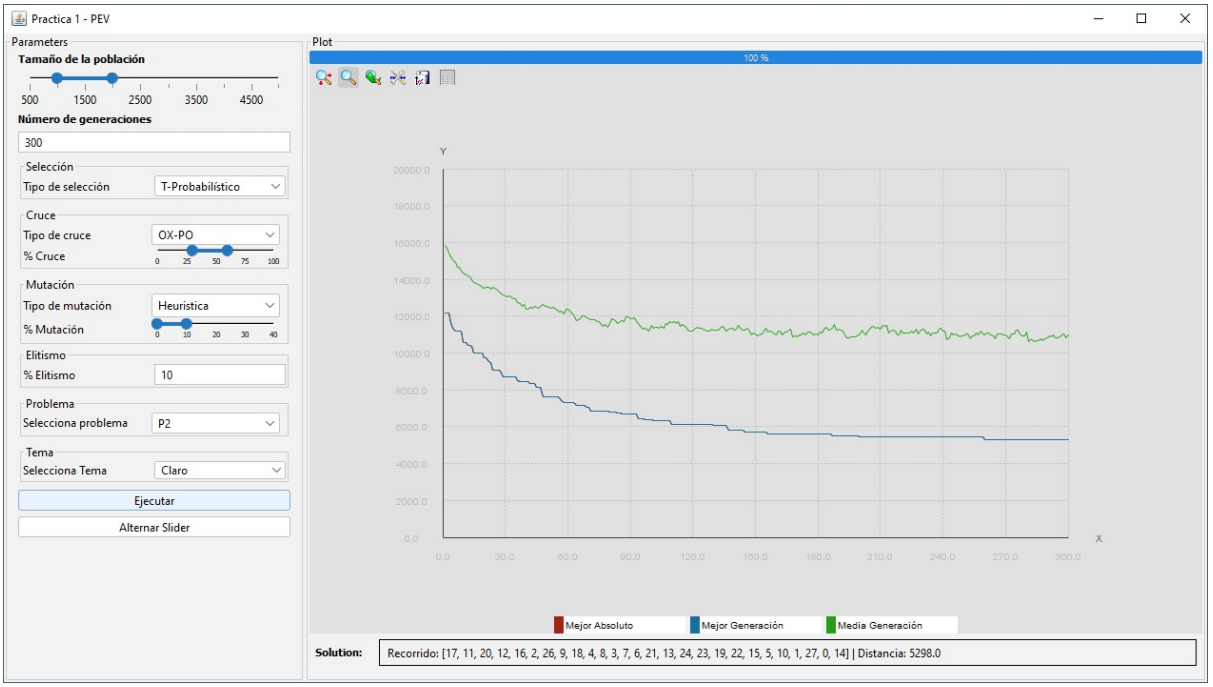


Figura 1. Interfaz del programa

Información general:

- Asignatura: Programación Evolutiva
- Grupo: 09
- Miembros: Nicolás Rosa Caballero y Javier Meitín Moreno
- Fecha de entrega: 28/03/2023

Reparto de tareas:

Nicolás Rosa Caballero:

- Algoritmos de cruce: PMX, OX, OX-PP, OX-PO ERX y CX (ciclos)
- Adaptación de fitness
- Sistema de sliders para la interfaz
- Lógica del sistema de análisis de parámetros
- Añadida barra de carga para la interfaz compatible con el sistema de análisis de parámetros
    - Ejecución normal: Muestra el progreso del algoritmo de 0 a 100
    - Análisis de parámetros: Muestra el progreso total de 0 a 100 en función del porcentaje de compleción individual de cada modelo.

Javier Meitín Moreno:

- Algoritmos de mutación: Inversión, intercambio, inserción y heurística.
- Algoritmos de cruce: Codificación ordinal.
- Algoritmos propios: Mutación y Cruce original.
- Pequeñas correcciones de la práctica 1.

Organización del proyecto:

- Las variables y miembros de campo siguen la nomenclatura Snake Case.
- Los métodos siguen la nomenclatura Camel Case.
- Los nombres de clase siguen la nomenclatura Pascal Case.

Nuestro proyecto está organizado en 12 módulos: Listamos a continuación los nuevos módulos especificos de esta práctica:

CrossAlgorithms.P2:

Para la práctica 2 hemos creado un módulo “CrossAlgorithms.P2” para separar los nuevos cruces de los de la primera (“CrossAlgorithms.P1”). La estructura de las clases es similar a la de la otra práctica. Tenemos una clase base llamada **CrossAlgorithmP2** que extiende la clase **CrossAlgorithm**. Todos los cruces nuevos extienden de esta clase y sobreescriben el método abstracto **cross(ChromosomeP2 first_child, ChromosomeP2 second_child)**.
Además, este CrossAlgorithmP2 después de hacer el cruce ejecuta un sistema de verificación para asegurarse de que los hijos generados son correctos (sin elementos repetidos) y lanza una excepción en caso contrario. Esto nos facilitó mucho el testeo de cruces. Actualmente está desactivado para no consumir recursos. Para activarlo solo hay que descomentar la línea 17 de CrossAlgorithmsP2.java

A parte de los cruces explicados en clase hemos creado un nuevo algoritmo propio inventado llamado **OriginalCross** tal como se pedía en el enunciado. Dicho cruce recuerda vagamente al cruce CO dado que usa una lista dinámica para ordenar los genes. En este caso, el algoritmo crea 2 listas (una por individuo) con los **valores** que puede tomar cada gen ordenados de menor a mayor. Luego se recorre el array de genes de los padres de manera simultánea comparando los valores de los genes[i] en ambos cromosomas. El 1er

hijo se queda con el mínimo, y el segundo con el máximo sólo si en su respectiva lista dinámica aún se encuentra dicho valor. Tras insertar el gen este se borra de la lista dinámica. Si se da el caso de que los genes[i] que se quieren insertar en el hijo ya han sido usados y por tanto no se encuentran en la lista dinámica, se insertará el primer elemento de la lista.

Ejemplo:
Datos:

- Padre1: [0, 2, 4, 5, 1]
- Padre2: [0, 5, 2, 4, 1]
- Valores1: [0, 1, 2, 4, 5]
- Valores2: [0, 1, 2, 4, 5]
- Hijo: null
- Hijo2: null

Iteración 0:

- Padre1[0] == Padre2[0] == 0
- El elemento 0 aun se encuentra en valores1 y valores2. Se inserta en Hijo1 e Hijo2 y se elimina de valores1 y valores2.
- Hijo1 == Hijo2 == 0
- Valores1: [1, 2, 4, 5]
- Valores2: [1, 2, 4, 5]

Iteración 1:

- Padre1[1] == 2
- Padre2[1] == 5
- Dado que 2 y 5 se encuentran en valores1 y valores 2 respectivamente, se pueden insertar en los hijos. (Recuerde que el mínimo va al primero)
- Hijo1: [0, 2]
- Hijo1: [0, 5]
- Valores1: [1, 4, 5]
- Valores2: [1, 2, 4]

Iteración 2:

- Padre1[2] == 4
- Padre2[2] == 2
- El mínimo en esta iteración es 2, pero este valor ya fue insertado en hijo1, asi que se pone 1 (1er elto en valores1). El 4 aun no se insertó en hijo2 así que no hay problema.
- Hijo1: [0, 2, 1]
- Hijo2: [0, 5, 4]
- Valores1: [4, 5]
- Valores2: [1, 2]

Al cabo de otras 2 iteraciones los resultados serán Hijo1 = [0, 2, 1, 4, 5] e Hijo2 = [0, 5, 4, 1, 2].

MutationAlgorithm.P2:

Al igual que con los métodos de cruce, hemos creado un nuevo módulo **MutationAlgorithms.P2.** Sin embargo, a diferencia de con los cruces, las clases de mutación extienden directamente del **MutationAlgorithm** de la práctica 1. Es decir, no hay un “MutationAlgorithmP2”.

Como se pedía en el enunciado, hemos creado un algoritmo de mutación propio llamado **OriginalMutation**. Este algoritmo tiene 2 fases. En la primera se realiza un intercambio de genes. La única diferencia con la mutación por intercambio es que en vez de calcular una nueva posición aleatoria para el gen que muta, el número aleatorio se suma a la posición actual del gen. En la segunda fase se realiza un desplazamiento a la derecha de todos los genes.

Ejemplo:

- Individuo = [0, 2, 1, 4, 5]
- Fase 1:
    - Se muta genes[2], es decir, el 1.
    - Se calcula un número aleatorio perteneciente al intervalo [0, num genes] = 3.
    - 2 + 3 = 5. Dado que 5 no es una posición válida en el array se hace 5 % num genes == 0.
    - El individuo pasa a valer [1, 2, 0, 4, 5]
- Fase 2:
    - Se desplazan los elementos a la derecha.
    - Solución = [5, 1, 2, 0, 4]

ViewController:

Cabe destacar la implementación del análisis de parámetros de ViewController que cambia bastante su funcionamiento interno. Anteriormente ejecutaba un hilo para el modelo a ejecutar y ya. En el análisis de parámetros, hemos decidido ejecutar un número configurable de hilos (en este caso 10) y un número configurable de permutaciones de variables. En este caso para cada “RangedValue” se hacen 5 pruebas. Tenemos 3 “RangedValue”: Para la población, para el cruce y para la mutación. Por lo tanto el número de permutaciones es 5 * 5 * 5 = 125
Al ejecutar, lo que hacemos es crear 125 modelos (que se guardan en array) y usando un ExecutorService, los ejecutamos todos en un hilo a parte. Mientras el ejecutor gestiona los hilos activos controlando que no se ejecuten más de 10 a la vez, se actualiza la barra de progreso. Al terminar, iteramos todos los modelos (usando programación funcional) para encontrar el que nos dio mejor resultado y mostrarlo en pantalla. En nuestras pruebas además de esto, imprimimos por consola los parámetros que se habían usado para obtener ese resultado (solo se ve si la variable debugMode de ViewController está a true).

Además, este sistema también es interrumpible, por lo que se puede usar el botón de stop para parar la ejecución en cualquier momento.

UML:

Es igual que en la Práctica 1 porque usamos la misma arquitectura

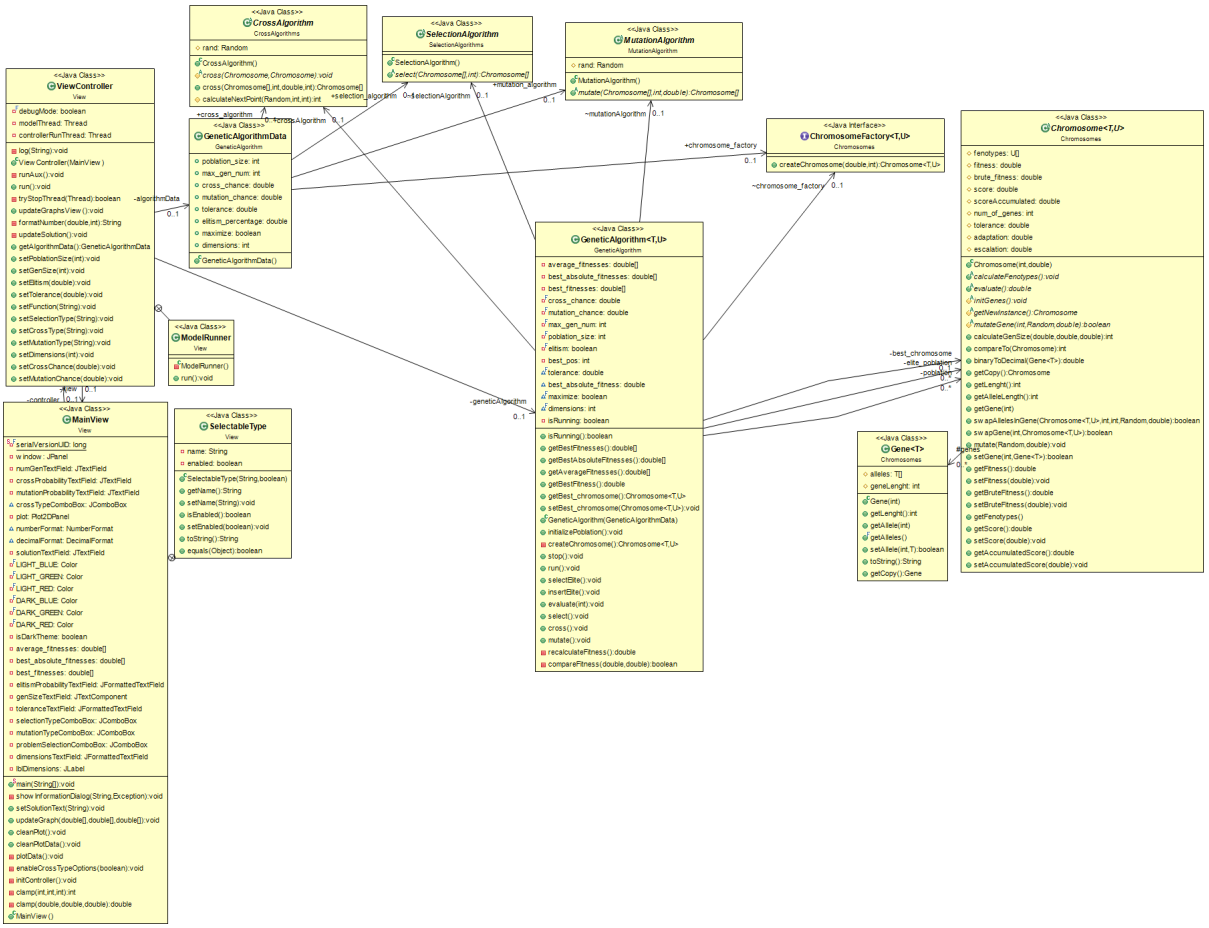


Figura 2. UML Reducido

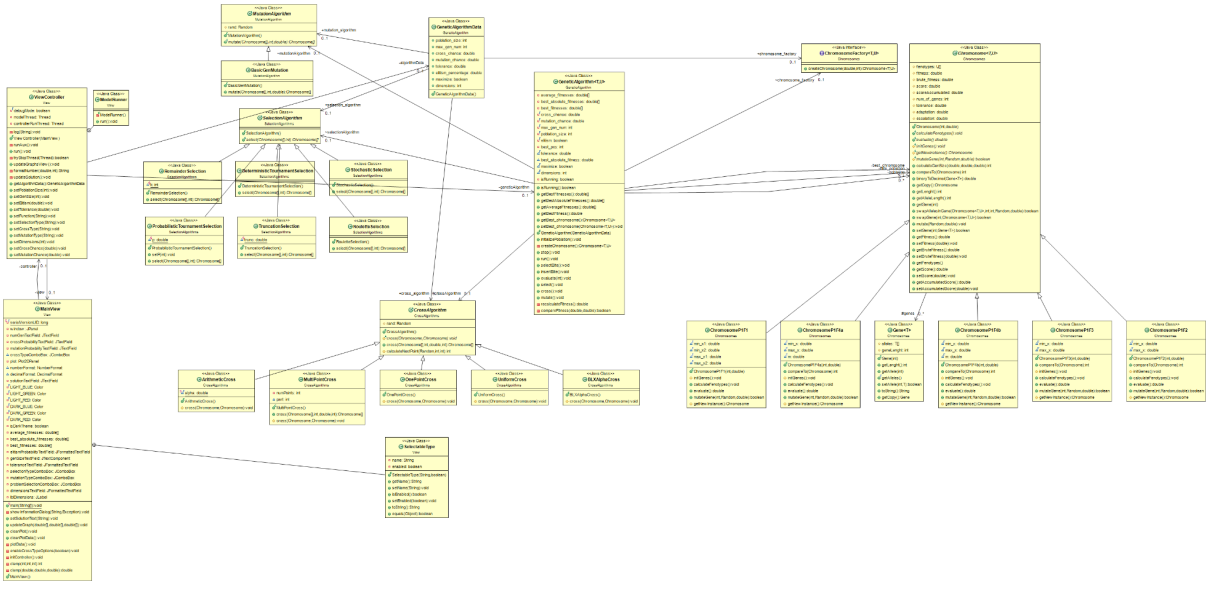


Figura 3. UML Completo

Características de la interfaz:

Parte común a la práctica 1:

El panel izquierdo de la interfaz permite configurar una serie de parámetros para el algoritmo tales como el tamaño de la población, el número de generacione, la tolerancia (intervalo mínimo al codificar números con booleanos), el tipo de selección, el tipo de cruce y su probabilidad, el tipo de mutación y su probabilidad, la probabilidad de elitismo, la función y el tema de la interfaz (soporta tema claro y oscuro). Finalmente existe un botón de ejecutar y otro de reiniciar.

El panel derecho contiene la gráfica y debajo de esta un cuadro donde se escribe la solución.

La interfaz garantiza que los datos introducidos son válidos, si el dato introducido no es válido pondrá uno por defecto. Por lo que si escribes 10sfds20 en tamaño de la población la interfaz lo cambiará a 10, que es el valor antes de que se introdujera datos no válidos. Igualmente se controla que no se introduzcan valores negativos, o en los campos de porcentajes que el valor esté entre 0 y 100. Excepcionalmente el campo tolerancia tiene un valor mínimo de 0.001.

El tamaño de la ventana está por defecto en 1280*720 píxeles. La ventana es escalable, hasta un mínimo de 512 * 256. El panel izquierdo no es scrolleable, por lo que si la ventana es muy pequeña no se podrán ver todas las opciones de configuración.

Parte común a la práctica 2:

Ahora la tolerancia no se muestra si no se usa para crear al cromosoma, es decir, no se muestra en P1-F4b ni en P2. Se ha eliminado el botón de reiniciar porque era redundante. En su lugar ahora aparece un botón “Detener” mientras se ejecuta el algoritmo, cuando acaba este se oculta.

Se ha añadido un botón “Alternar Slider” que sustituye las cajas de texto por sliders de 2 manillas que permiten configurar un intervalo para el tamaño de la población, probabilidad de cruce y probabilidad de mutación.

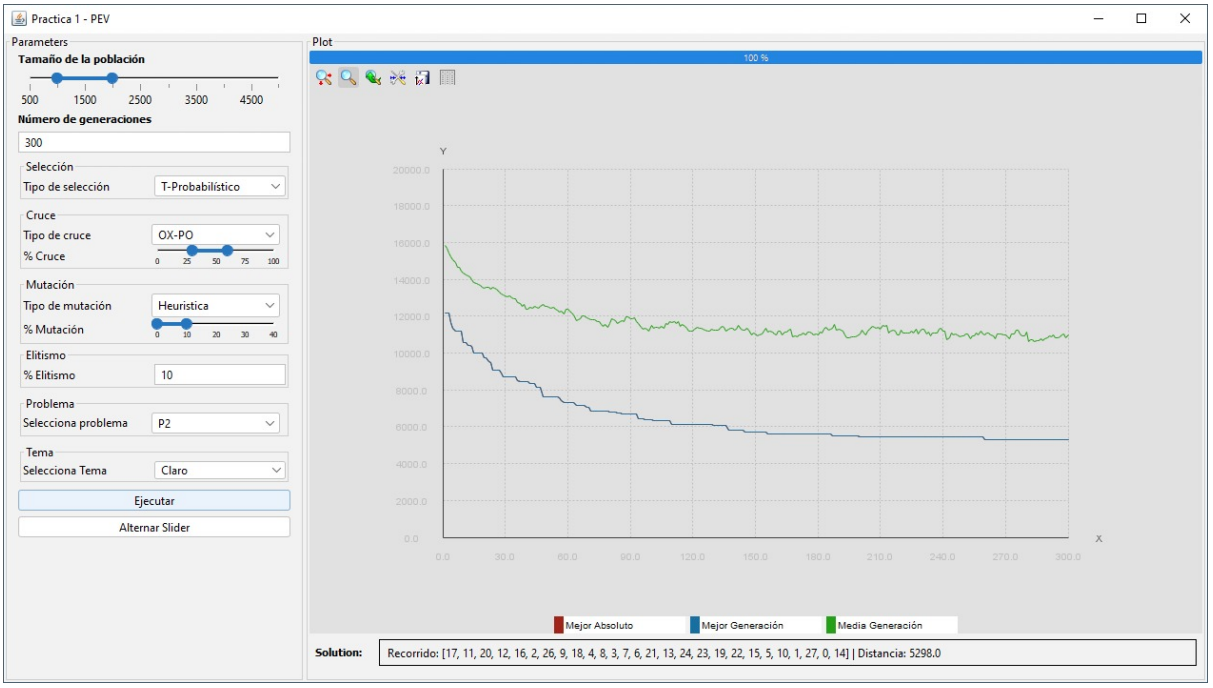
Este componente no pertenece a la librería estándar de Swing, fue descargado de un repositorio de github y modificado por nosotros para adaptarlo a la estética de nuestra interfaz. Fuente: https://github.com/ernieyu/Swing-range-slider

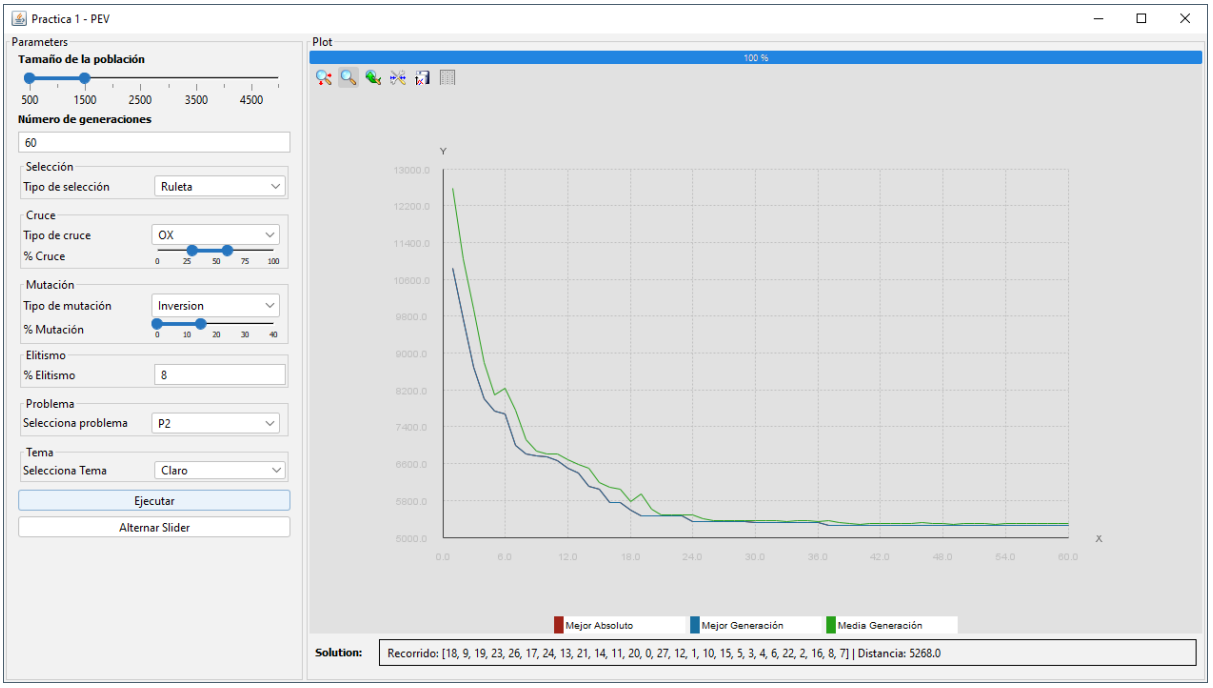
Los parámetros por defecto en la interfaz son los mismos que los indicados en el guión de la práctica 1:

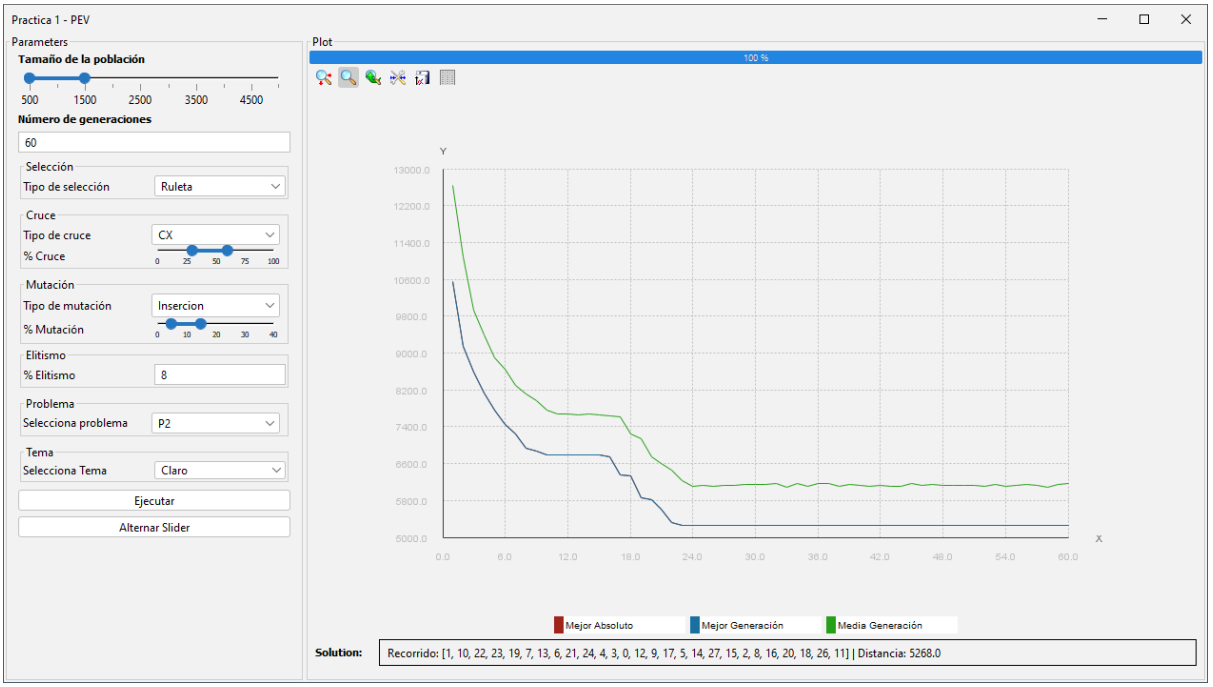
- Tamaño de la población: (100)
- Número de generaciones (100)
- Porcentaje de cruces (60%)
- Porcentaje de mutaciones (5%)

- Precisión para la discretización del intervalo (0.001)
- Método Selección
- Método Cruce
- Método Mutación
- Posibilidad de seleccionar porcentaje de elitismo
- Selección de la dimensión d en la función 4

Gráficas representativas:







Conclusiones:

- El elitismo juega un papel muy importante en este problema. Simplemente un 1% de elitismo causa que un método de cruce pase de ser inviable a ser el mejor o como mínimo funcional. Prácticamente todos los cruces provocan que los resultados empeoren en vez de que mejoren si no se usa elitismo.
- Los resultados son muy diversos en función de la combinación de parámetros elegida. En general el algoritmo OX y sus variantes dan los mejores resultados. PMX tiende a dar los peores, tanto con elitismo como sin él.
- La mayoría de métodos de selección dan malos resultados con la mayoría de cruces. El torneo probabilístico solo funciona de forma razonable con el cruce OX-PO. Con los demás los resultados llegan a ser negativos. Con el torneo determinístico esto es aún peor ya que no funciona bien ni siquiera con el cruce OX-PO.
- El mejor método de cruce es el de Ruleta, en general es el que aporta mejores resultados con la convergencia más rápida.
- Las mutaciones que mejores resultados dan en general son inversión e inserción. Intercambio no parece alterar los resultados de forma notable.
- Las mejores combinaciones que encontramos son estas dos. Ambas dan buenos resultados y convergen muy rápido (en menos de 60 generaciones ya han encontrado una solución óptima):
- ERX es el método de cruce más lento, es más, es aproximadamente 5 veces más lento que los demás.
- El método de mutación más lento es el método de inserción.
    - Primera combinación:
        - Selección: **Ruleta**
        - Cruce: **OX**
        - Porcentaje de cruce: Cualquier valor entre 40 y 60%
        - Mutación: **Inversión**

        - Porcentaje de mutación: Cualquier valor entre 5 y 10%
        - Elitismo: Cualquier valor superior a 1, se aprecian mejores resultados con valores en torno al 8%
        - Generaciones necesarias: 60 (converge en torno a 25, pero es recomendable dejar margen)
        - Tamaño población: Cualquier valor superior a 500 individuos
    - Segunda combinación:
        - Selección: **Ruleta**
        - Cruce: **CX**
        - Porcentaje de cruce: Cualquier valor entre 40 y 60%
        - Mutación: **Inserción**
        - Porcentaje de mutación: Cualquier valor entre 5 y 10%
        - Elitismo: Cualquier valor superior a 1, se aprecian mejores resultados con valores en torno al 8%
        - Generaciones necesarias: 60 (converge en torno a 25, pero es recomendable dejar margen)
        - Tamaño población: Cualquier valor superior a 500 individuos