



**Dpto. Sistemas Informáticos y Computación
UNIVERSITAT POLITÈCNICA DE VALÈNCIA**

Técnicas, Entornos y Aplicaciones de Inteligencia Artificial

Práctica Algoritmos Genéticos (Opt4J)

Contenido

1.- Introducción	1
1.1.- Instalación de Opt4J	2
1.2.- Configuración de un proyecto Java en Eclipse para utilizar Opt4J	3
1.3.- Importación de un proyecto-plantilla en Eclipse para utilizar Opt4J	4
2.- Modelado de Problemas en Java para ser resueltos en Opt4J	5
Ejemplo1. Operaciones aritméticas.....	5
Ejemplo2. Distribución de pesos sobre una estructura o viga	13
3.- Aplicación del AG y resolución de problemas con Opt4J	15
Ejemplo1. Operaciones aritméticas.....	17
Ejemplo2. Distribución de pesos sobre una estructura	19

Evaluación de la Práctica. Problema Propuesto

1.- Introducción

Esta práctica permitirá contrastar la funcionalidad de un Algoritmo Genético (AG) para la resolución de problemas de optimización. Para ello, se utilizará el entorno **Opt4J**, que proporciona los operadores básicos de un AG y permitirá la importación y resolución de nuevos problemas modelados en java.

Concretamente, **la práctica consistirá en:**

- (i) Modelar un problema de optimización mediante AG: diseñar los individuos del AG y la función fitness correspondiente (código Java a implementar usando Opt4J).
- (ii) Evaluar (usando el entorno Opt4J) la funcionalidad del AG para la resolución del problema modelado.

Opt4J (<https://sdarg.github.io/opt4j/>) es un entorno libre y abierto basado en Java para la resolución de problemas mediante métodos de optimización metaheurística (entre ellos, los algoritmos genéticos). La ventaja de trabajar con este entorno es que podemos centrarnos en el modelado del problema a resolver (en Java), sin tener que preocuparnos por los detalles de implementación del algoritmo de resolución a emplear.

Opt4J también dispone de una interfaz gráfica de usuario (IGU) que facilita la parametrización del AG y ofrece un visor gráfico que muestra el proceso de optimización, permitiendo visualizar interactivamente las soluciones encontradas. En <https://sdarg.github.io/opt4j/documentation.html> existe un tutorial, escueto pero suficiente, para instalar y comenzar a trabajar con Opt4J.

A continuación, se muestra la instalación de Opt4J (Punto 1.1) y la configuración de Eclipse para modelar problemas en java para ser resueltos en Opt4J (Puntos 1.2 y 1.3).

1.1.- Instalación de Opt4J

Opt4J se puede descargar de <https://sdarg.github.io/opt4j/download.html> y se puede ejecutar tanto desde Windows como Linux.

Una vez descargado y descomprimido el fichero (Figura 1), dispondremos del ejecutable y archivos .jar necesarios para ejecutar Opt4J y resolver nuestro problema modelado en Java.

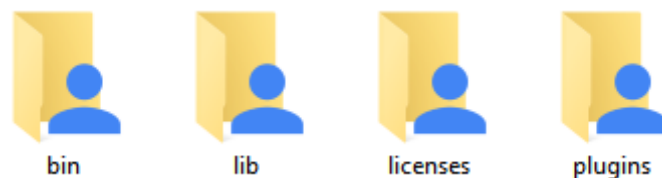


Figura 1. Estructura de carpetas de Opt4J

En el entorno Windows, ejecutaremos el archivo **opt4j.bat** que se encuentra en la carpeta bin. Si nos encontramos en Linux, ejecutaremos el archivo **opt4j**.

Una vez lanzado Opt4J, aparece la interfaz del sistema (Figura 2), donde se deben realizar los 3 pasos básicos para configurar el proceso de resolución. Basta hacer doble click para seleccionar cada elemento:

1. **Problema a resolver.** Por defecto, Opt4j viene con algunos ejemplos o benchmarks típicos como el problema de la mochila (Knapsack), n-reinas, generación de la cadena “Hello world” en base a caracteres aleatorios, etc.
2. **Modo de salida,** que puede ser un volcado a un fichero log (**Logger**) o un visor gráfico (**Viewer**). La opción del visor gráfico es más intuitiva pues, además, permite pausar/continuar el proceso de optimización y visualizar la información tanto del mejor individuo de cada iteración como del mejor individuo encontrado hasta el momento (que será resaltado en negrita).
3. **Algoritmo de optimización a utilizar.** En esta práctica utilizaremos “*EvolutionaryAlgorithm*”, que corresponde al AG, donde podremos parametrizar el número de iteraciones, tamaño de la población inicial, etc.

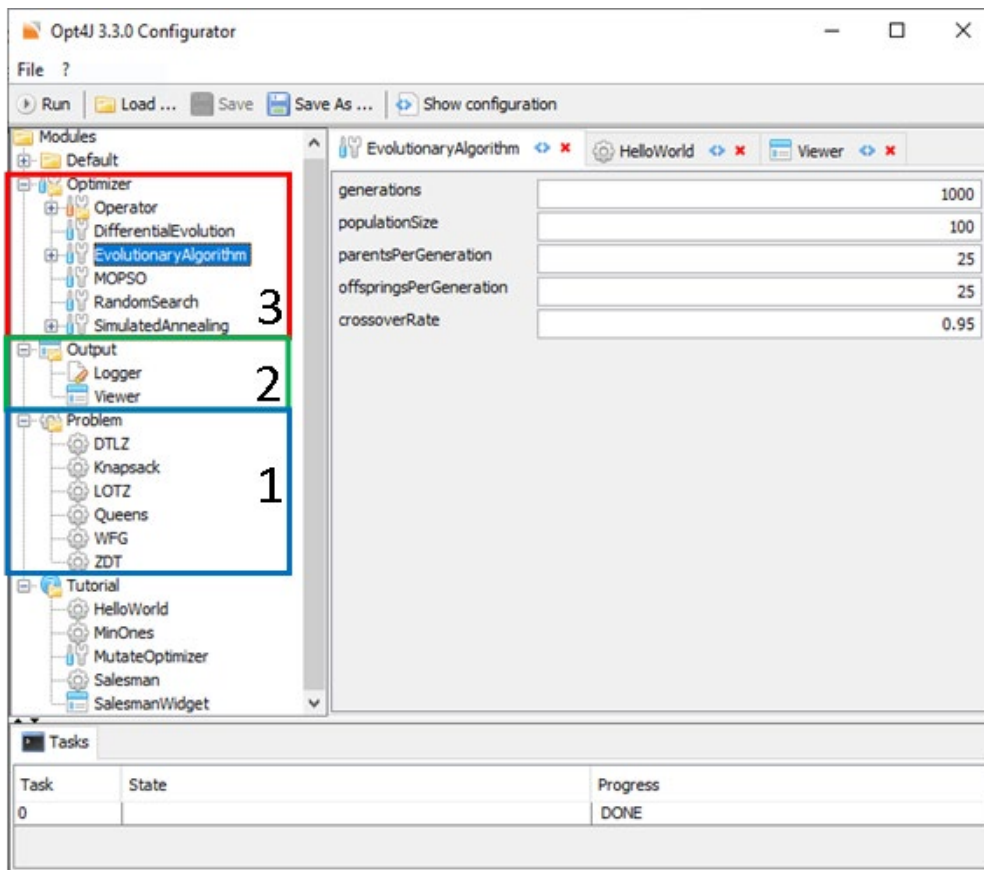


Figura 2. Opt4J en ejecución

NOTA IMPORTANTE. Para evitar problemas de ejecución, hay que elegir **solo UN elemento** en cada apartado. Es decir, SOLO UN problema, UN tipo de salida y UN algoritmo de optimización. Solo debe haber seleccionado uno de cada tipo en la parte superior de la ventana derecha. Además, para evitar problemas de inicialización se aconseja **seguir siempre** el orden marcado como 1, 2 y 3 ya que, de lo contrario, Opt4J puede generar una excepción de configuración.

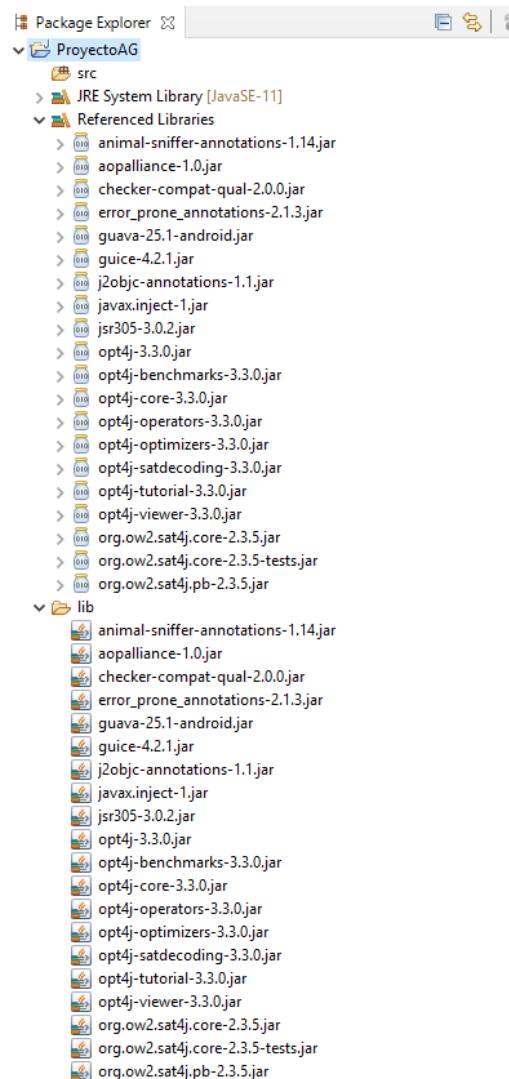
1.2.- Configuración de un proyecto Java en Eclipse para utilizar Opt4J

Nota: En el punto 1.3 se indica un método alternativo mucho más simple y **recomendable** (basado en la importación de un proyecto plantilla) a los pasos descritos en este punto.

Opt4J permite la importación y resolución de problemas previamente modelados en Java. Por simplicidad, utilizaremos el entorno Eclipse. Para modelar un problema en Eclipse y que sea resuelto en Opt4J, se debe configurar adecuadamente el proyecto Java en Eclipse.

Para ello:

1. Creamos un proyecto Java (Java Project) con dos carpetas, tal y como se muestra en la Figura 3:
 - a) Una carpeta **src** donde pondremos nuestro código fuente a medida que lo vayamos implementando.
 - b) Una carpeta **lib** donde copiamos todos los archivos .jar de la carpeta lib de Opt4J, mostrada en la Figura 1. Para actualizar contenido, botón derecho y refrescar la carpeta.
2. A continuación, configuraremos las propiedades del proyecto creado (botón derecho, 'Properties'). Para ello, en la ventana de propiedades, seleccionamos 'Java Build Path' y en la pestaña 'Libraries/Classpath', añadiremos (Add JARs...) **todos** los archivos .jar que hay en la carpeta lib creada en el paso 1.b. Tras ello, aparecerá en el proyecto 'Reference Libraries' con dichos archivos.



A partir de ahora ya tenemos nuestro proyecto Java configurado para poder trabajar con Opt4J y modelar problemas de optimización.

1.3.- Importación de un proyecto-plantilla en Eclipse para utilizar Opt4J

RECOMENDABLE. Si no deseamos configurar un proyecto Java desde cero para trabajar con Opt4J, tal y como se describe en el punto anterior, podemos importar un proyecto-plantilla **"ProyectoAG.zip"** que se proporciona en Poliformat. Los pasos son:

- 1) Descomprimir **"ProyectoAG.zip"** en un directorio donde tengamos permisos (**nunca en el escritorio de los ordenadores del laboratorio/virtualizados**). Lo ideal es hacerlo en la carpeta donde tenemos otros proyectos previos de Eclipse. La carpeta creada debe contener las carpetas bin, lib, opt4j-src, src, etc.
- 2) Desde Eclipse elegimos la opción 'File\Open Projects from File System' y en "Import source" elegiremos el directorio que se acaba de crear. **En caso de que aparezcan varios elementos** a importar, es muy importante

que **deseleccionemos todos los subdirectorios y dejemos marcados solo el proyecto principal "ProyectoAG"**, ya que solo queremos importar un único proyecto.

- 3) Ahora ya dispondremos de un proyecto base con todos los archivos, librerías configuradas y clases vacías *Creator*, *Decoder*, *Evaluator*, etc. (en la carpeta src) **que deberemos completar** para resolver nuestro problema. Evidentemente, Eclipse marcará varios errores porque las clases y parámetros necesarios todavía no están implementados.
- 4) Desde la ventana Project Explorer (accesible desde Window\Show View\Project Explorer) tendremos acceso a todas las carpetas, archivos y clases.

Tras estos pasos, ya tendremos nuestro proyecto Java configurado para poder trabajar con Opt4J y modelar problemas de optimización. La tarea a realizar consistirá en completar la implementación de las clases propuestas y ejecutar posteriormente el código correspondiente.

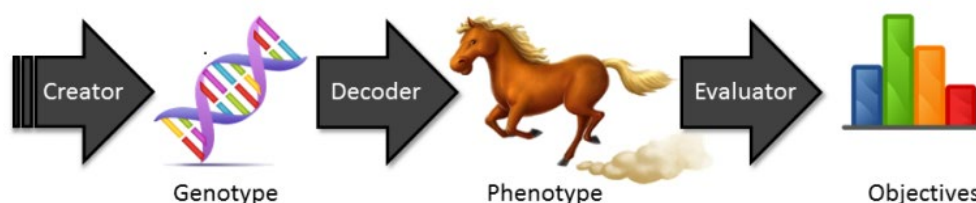
2.- Modelado de Problemas en Java para ser resueltos en Opt4J

Los pasos para modelar en Java (Eclipse) y resolver problemas en Opt4 mediante un AG son:

- 1) Definición de los elementos del problema:
 - a) Definición *externa* del individuo (**fenotipo**), que representa la solución al problema,
 - b) Codificación *interna* del individuo (**genotipo**),
 - c) Función de evaluación o **fitness** que nos permite evaluar la calidad de los individuos (soluciones).

Para ello, el problema se modela mediante la implementación en Eclipse de las tres clases Java correspondientes (Figura 4):

- 1) una clase creadora del genotipo del individuo (**Creator**),
 - 2) una clase decodificadora que, dado un genotipo, lo convierte en su fenotipo (**Decoder**), y
 - 3) una clase evaluadora que evalúa la calidad del fenotipo (**Evaluator**).
- 2) Tras la implementación de las clases (en Eclipse) ejecutaremos la aplicación como una "Aplicación Java", que lanzará automáticamente Opt4J. En la carpeta *Problem* de Opt4J aparecerán todos los problemas que hemos implementado.
- 3) A continuación, parametrizamos Opt4J (seleccionando primero el problema, segundo el modo de salida, y tercero la metaheurística elegida, que **en nuestro caso será 'EvolutionaryAlgorithm'**), y ya podremos ejecutar y resolver el problema.



Veamos en detalle el proceso de modelado en Java utilizando Eclipse con algunos ejemplos.

Ejemplo1. Operaciones aritméticas

Diseñar un algoritmo para obtener una secuencia de las operaciones aritméticas (+, -, *, /), pudiéndose repetir operaciones, tal que aplicadas sucesivamente sobre un conjunto de 6 números enteros (elegidos al azar del 1..100) se obtenga un determinado número objetivo (elegido al azar entre 101 y 999). Por ejemplo, si el conjunto de números es 75, 50, 6, 3, 9, 7 y el número objetivo es 248, una posible solución sería: $((((75 + 50) * 6) / 3) - 9) + 7$. Obviamente podría no llegar a obtenerse el número objetivo, siendo la meta aproximarse lo más posible.

Para comenzar la implementación, en nuestro proyecto Java crearemos un nuevo package para contener todo el código de este ejemplo, al que denominaremos *'operaciones.aritmeticas'*.

Paso 1. Creación de la clase Creator

La clase Creator, que implementa la creación del genotipo, se define en Opt4J de la forma:

```
public class NombreClaseCreator implements Creator<GENOTIPO>
```

La clase implementa un Creator<GENOTIPO>, donde GENOTIPO es el tipo que codifica al individuo del AG. Aunque podríamos implementar cualquier codificación como genotipo, existen varios tipos predefinidos en Opt4J que nos servirán para la mayoría de los problemas. Los tipos más comunes aparecen en la siguiente tabla.

Genotipo	Codificación de un individuo	Ejemplo de individuo
BooleanGenotype	Secuencia de valores T/F (puede contener repeticiones)	[T,F,T,T], donde el individuo es una combinación de valores a True o False; por ej. si buscamos una combinación óptima de luces encendidas/apagadas
BooleanMapGenotype<K>	Secuencia de valores que establecen un mapa entre el tipo K y un valor T/F	Asumiendo BooleanMapGenotype<String>, el individuo será del estilo [<"luces", F>, <"motor", T>, <"climatizador", F>]; por ej. si buscamos una combinación óptima de consumo de los elementos de un coche
DoubleGenotype	Secuencia de valores double (puede contener repeticiones)	[10.5, 0.78, 2.93, 4.765], donde el individuo es una combinación de valores con decimales; por ej. si buscamos una combinación óptima de valores de variables en una ecuación
DoubleMapGenotype<K>	Secuencia de valores que establecen un mapa entre el tipo K y un valor double	Asumiendo DoubleMapGenotype<Character>, el individuo será del estilo [<'A', 50.2>, <'J', 3.2>, <'f', 0.4>]; por ej. si buscamos una combinación óptima de pesos a caracteres de un alfabeto
IntegerGenotype	Secuencia de valores enteros (puede contener repeticiones)	[3, 7, 20, 4, 6, 9, 33], donde el individuo es una combinación de valores enteros; por ej. si buscamos la combinación óptima de inversión en € a distintos productos
IntegerMapGenotype<K>	Secuencia de valores que establecen un mapa entre el tipo K y un valor entero	Asumiendo IntegerMapGenotype<Integer>, el individuo será del estilo [<5, 6> <4, 3>, <10, 7>]; por ej. si buscamos la combinación óptima de inversión a un producto con un determinado identificador numérico {ej. id=5, id=4, id=10}
PermutationGenotype<E>	Secuencia ordenada y sin repeticiones de los elementos de tipo E a optimizar	Asumiendo PermutationGenotype<Integer>, el individuo será del estilo [5, 3, 2, 4, 1]; por ej. si buscamos la mejor permutación de 5 ciudades a visitar

SelectGenotype<V>	Secuencia de valores (donde sí puede haber repetición) de tipo V. Por ejemplo, <code>SelectGenotype<Character></code>	Asumiendo <code>SelectGenotype<ColorSemaforo></code> , el individuo será del estilo [rojo, rojo, verde, ámbar, verde]; por ej. si buscamos la mejor combinación de estado de los semáforos a lo largo de una calle
--------------------------------	--	--

En el Javadoc correspondiente (<https://sdarg.github.io/opt4j/documentation.html>) se puede consultar más información sobre estos y otros genotipos. Siempre que sea posible hay que tratar de utilizar los genotipos por defecto que ya vienen con su código implementado. En caso de que definamos genotipos distintos a los que proporciona Opt4J, tendríamos que implementar los métodos correspondientes para sus operadores, como por ejemplo cruce o mutación.

Retomando nuestro ejemplo, la codificación o genotipo para nuestro problema será una secuencia (con posibles repeticiones) de los símbolos: +, -, *, /. Por tanto, hay al menos dos opciones:

- **Opción 1:** Si asignamos a cada símbolo un código entero de 0..3, podríamos utilizar un `IntegerGenotype` y el individuo podría ser <1, 1, 3, 2> representando a "--/*".
- **Opción 2:** Si creamos un tipo enumerado `MathematicalSymbol` con los valores {PLUS, MINUS, MULTIPLICATION, DIVISION}, podríamos utilizar un `SelectGenotype<MathematicalSymbol>`, tal que un individuo podría ser por ejemplo <PLUS, MINUS, MINUS, MULTIPLICATION>.

Por otra parte, **también** podríamos haber utilizado un `SelectGenotype<Character>`. Sin embargo, en este ejemplo no podríamos utilizar `PermutationGenotype<MathematicalSymbol>` o `PermutationGenotype<Character>` ya que en ambos casos nos devolvería una posible permutación de los símbolos, utilizando todos ellos, pero sin repeticiones (que en nuestro caso sí son posibles y deseados).

En este ejemplo optamos por la segunda opción, utilizando el tipo enumerado, ya que resulta más elegante y fácilmente extensible en el futuro. El código está formado por tres partes:

- 1) En primer lugar, en el package 'operaciones.aritmeticas' creado, definimos el tipo Enumerado `MathematicalSymbol` (New Enum 'MathematicalSymbol') con los cuatro símbolos a usar:

```
package operaciones.aritmeticas;

public enum MathematicalSymbol
{
    PLUS, MINUS, MULTIPLICATION, DIVISION
}
```

- 2) En segundo lugar, definimos en el package una clase de Datos 'Data' (New Class 'Data') con la información inicial del problema, es decir el número de símbolos "+, -, *, /" que podemos usar (5 en nuestro caso porque hay 6 números), cuáles son los números del ejemplo y el valor objetivo a conseguir:

```
package operaciones.aritmeticas;

public class Data
{
    // estos datos se podrian calcular al azar
    public static final int numeroSimbolos = 5; // numero de simbolos a elegir
    public static final int numeros[] = {75, 50, 6, 3, 9, 7}; // en total hay "numeroSimbolos+1" numeros
    public static final int resultadoObjetivo = 248; // numero objetivo a conseguir
}
```

- 3) En último lugar, definimos la clase Creator (New Class 'OpsAritmeticasCreator') para definir el genotipo y crear la población inicial de individuos. Remarcamos los siguientes puntos:

- a) Definimos el genotipo según el tipo `SelectGenotype<V>` predefinido en Opt4J, concretamente `SelectGenotype<MathematicalSymbol>(Symbols)`.

De esta forma estamos indicando a Opt4J que el genotipo será una secuencia de *MathematicalSymbol* elegidos de la colección *Symbols* que acabamos de crear. **No utilizaremos** un *PermutationGenotype<MathematicalSymbol>* ya que podría interesarnos repetir más de un símbolo definido en *MathematicalSymbol* o, por el contrario, no usar alguno de ellos.

- b) Definimos cómo se va a inicializar el genotipo para la población inicial mediante el método *init* (ya definido en Opt4J): de forma aleatoria, eligiendo una secuencia de 5 (valor de *numeroSimbolos*) símbolos. Esto permitirá crear individuos como por ejemplo *<MULTIPLICATION, PLUS, DIVISION, PLUS, MINUS>*, *<DIVISION, MULTIPLICATION, PLUS, MINUS, PLUS>*, etc.

Por tanto, el código de la clase **Creator** queda como:

```
package operaciones.aritmeticas;

import java.util.Random;

import org.opt4j.core.genotype.SelectGenotype;
import org.opt4j.core.problem.Creator;

public class OpsAritmeticasCreator implements Creator<SelectGenotype<MathematicalSymbol>>
{
    public SelectGenotype<MathematicalSymbol> create()
    {
        MathematicalSymbol[] Symbols = {MathematicalSymbol.PLUS, MathematicalSymbol.MINUS,
            MathematicalSymbol.MULTIPLICATION, MathematicalSymbol.DIVISION };

        SelectGenotype<MathematicalSymbol> genotype = new SelectGenotype<MathematicalSymbol>(Symbols);
        // el genotipo estara formado por "numeroSimbolos" matematicos elegidos al azar
        // en nuestro caso la poblacion sera un conjunto de individuos, donde cada individuo son 5 simbolos
        genotype.init(new Random(), Data.numeroSimbolos);

        return genotype;
    }
}
```

NOTA. El número de individuos que formará la población inicial **no** lo tenemos que indicar en el código de la clase *Creator*, pues dicho valor se indicará en la parametrización del AG en Opt4J (Figura 2). Es decir, se podrán lanzar distintas parametrizaciones del AG para este mismo problema y todas ellas utilizarán el código de esta clase.

Paso 2. Creación de la clase *Decoder*

Una vez definido el genotipo, la clase *Decoder* se encarga de convertir el genotipo en su fenotipo correspondiente. En Opt4J se hace de la siguiente forma:

```
public class NombreClaseDecoder implements Decoder<GENOTIPO, FENOTIPO>
```

Esta clase implementa un *Decoder<GENOTIPO, FENOTIPO>*, donde *GENOTIPO* es el tipo definido en la clase *Creator* anterior y *FENOTIPO* es nuestro individuo.

El fenotipo puede ser cualquier clase que deseemos, pero es aconsejable que sea un tipo que se pueda imprimir directamente por pantalla para que cuando estemos ejecutando el algoritmo de resolución, el individuo se pueda imprimir (y visualizar) en el visor de Opt4J (ver Figura 2). Por tanto, algunos de los fenotipos más sencillos son: *String*, *ArrayList<X>*, o *X[]* siendo *X* un tipo imprimible (por ejemplo, *int*, *double*, tipo enumerado, etc.)

Por otra parte, podrían definirse distintas codificaciones para los genotipos y convertirlos al mismo fenotipo sin más que definir tantas clases *Decoder* como sean necesarias.

En nuestro ejemplo, optamos porque el fenotipo sea simplemente una colección de símbolos matemáticos, es decir un tipo *ArrayList<MathematicalSymbol>*. Esto nos permitirá visualizar en Opt4J la combinación de símbolos matemáticos de cada individuo.

Por tanto, definimos la clase *Decoder* (New Class 'OpsAritmeticasDecoder') cuyo código es:


```

package operaciones.aritmeticas;

import java.util.ArrayList;
import org.opt4j.core.genotype.SelectGenotype;
import org.opt4j.core.problem.Decoder;

public class OpsAritmeticasDecoder
    implements Decoder<SelectGenotype<MathematicalSymbol>, ArrayList<MathematicalSymbol>>
{
    //no es necesario devolverlo como String, pero si un tipo imprimible/visualizable en el visor Opt4J

    public ArrayList<MathematicalSymbol> decode(SelectGenotype<MathematicalSymbol> genotype)
    {
        ArrayList<MathematicalSymbol> phenotype = new ArrayList<MathematicalSymbol>();
        //aquí se podría poner código para validar que el fenotipo cumpla con ciertas restricciones
        for (int i = 0 ; i < genotype.size(); i++)
        {
            phenotype.add(genotype.getValue(i));
        }
        return phenotype;
    }
}

```

En este código se puede ver como esta clase Decoder convierte de nuestro genotipo (*SelectGenotype<MathematicalSymbol>*) a nuestro fenotipo (*ArrayList<MathematicalSymbol>>*). Para ello se define el método *decode* que, dado un genotipo, devuelve el fenotipo correspondiente. El código es muy sencillo, ya que simplemente pasa los símbolos del genotipo al fenotipo.

Es importante resaltar que, en el método *decode* se podrían realizar operaciones de validación del genotipo. Por ejemplo, supongamos que no debería haber más de dos operaciones de suma (+). En otros casos, también podríamos implementar un método para validar y corregir situaciones indeseables y hacer que el individuo satisfaga las restricciones a partir del genotipo recibido. Esto es importante, ya que estas correcciones no se pueden hacer en las operaciones internas de cruce y mutación de Opt4J (salvo que se modifique dicho código). Por tanto, deben realizarse en el proceso de decodificación.

Paso 3. Creación de la clase Evaluator

Una vez definido el fenotipo, la clase Evaluator se encarga de calcular el valor de fitness de este individuo en la función objetivo. En Opt4J se hace de la siguiente forma:

```
public class NombreClaseEvaluator implements Evaluator<FENOTIPO>
```

Esta clase implementa un *Evaluator<FENOTIPO>*, donde *FENOTIPO* es el individuo que hemos decodificado anteriormente.

La clase define un método *evaluate* que, dado un fenotipo, calcula el valor de fitness para la función objetivo, clarificando además si se quiere minimizar o maximizar. En nuestro ejemplo obtenemos el valor resultante tras aplicar los símbolos matemáticos a los números de partida (75, 50, 6, 3, 9, 7), y lo que buscamos es que el resultado sea lo más parecido al valor objetivo (en nuestro ejemplo 248).

Por tanto, el código de nuestra clase Evaluator es (New Class 'OpsAritmeticasEvaluator'):

```

package operaciones.aritmeticas;

import java.util.ArrayList;
import org.opt4j.core.Objectives;
import org.opt4j.core.Objective.Sign;
import org.opt4j.core.problem.Evaluator;

public class OpsAritmeticasEvaluator implements Evaluator<ArrayList<MathematicalSymbol>>
{
    public Objectives evaluate (ArrayList<MathematicalSymbol> phenotype)
    {
        int resultado = Data.numeros[0];

        for (int i = 0; i < phenotype.size(); ++i)
        {
            switch (phenotype.get(i))
            {
                case PLUS: resultado += Data.numeros[i+1]; break;
                case MINUS: resultado -= Data.numeros[i+1]; break;
                case MULTIPLICATION: resultado *= Data.numeros[i+1]; break;
                case DIVISION: resultado /= Data.numeros[i+1]; break;
            }
        }

        // si hay un individuo que no cumple con ciertas restricciones le ponemos un valor de fitness indeseado

        // queremos minimizar la diferencia entre el resultado objetivo y el evaluado

        Objectives objectives = new Objectives();
        objectives.add("Valor objetivo-MIN", Sign.MIN, Math.abs(Data.resultadoObjetivo - resultado));
        return objectives;
    }
}

```

Como puede observarse, lo único que hacemos es calcular el resultado de aplicar los símbolos matemáticos del fenotipo a los datos de entrada. Es importante notar que en la evaluación puede ocurrir que el individuo no cumpla ciertas restricciones que no hemos podido resolver en la clase Decoder. Por tanto, bastará con asignar a este individuo un valor de fitness indeseado (si nuestro objetivo es minimizar, podemos ponerle un valor de fitness de +infinito, y si estamos maximizando un valor de -infinito).

En las últimas líneas del método evaluate creamos un objetivo, le damos una descripción textual “Valor objetivo-MIN”, indicamos que queremos minimizarlo y le indicamos su valor.

Imaginemos una situación en la que, en lugar de estar trabajando con una función mono-objetivo, queremos también que nuestra solución tenga el mayor número de símbolos “+” posible. Es decir, ahora queremos trabajar con una **función multi-objetivo**. Esto sería tan sencillo como implementar el siguiente método evaluate que ahora tiene dos objetivos: uno que se quiere minimizar y otro que se quiere maximizar.

```

public Objectives evaluate(ArrayList<MathematicalSymbol> phenotype)
{
    int resultado = Data.numeros[0];
    int numSUM = 0;

    for (int i = 0; i < phenotype.size(); ++i)
    {
        switch (phenotype.get(i))
        {
            case PLUS: resultado += Data.numeros[i+1]; ++numSUM; break;
            case MINUS: resultado -= Data.numeros[i+1]; break;
            case MULTIPLICATION: resultado *= Data.numeros[i+1]; break;
            case DIVISION: resultado /= Data.numeros[i+1]; break;
        }
    }

    // si hay un individuo que no cumple con ciertas restricciones le ponemos un valor de fitness indeseado

    // queremos minimizar la diferencia entre el resultado objetivo y el evaluado
    Objectives objectives = new Objectives();
    objectives.add("Valor objetivo-MIN", Sign.MIN, Math.abs(Data.resultadoObjetivo - resultado));
    objectives.add("Cuenta de +", Sign.MAX, numSUM);

    return objectives;
}

```

Como puede observarse en el código, es muy sencillo añadir múltiples funciones objetivo. Simplemente basta añadir una descripción textual, indicar si se quiere MINimizar/MAXimizar y añadir el valor resultante.

Paso 4. Creación de la clase Module

Una vez definidas las clases anteriores, simplemente necesitamos definir la clase Module que configura el problema. Esta clase se define en Opt4J de la siguiente forma:

```
public class ClaseModule extends ProblemModule
```

El objetivo de esta clase es indicar qué clase Creator, Decoder y Evaluator se va a utilizar. De esta forma, en nuestro ejemplo basta con implementar la siguiente clase (New Class 'OpsAritmeticasModule') que configura el problema mediante el método bindProblem:

```
package operaciones.aritmeticas;

import org.opt4j.core.problem.ProblemModule;

public class OpsAritmeticasModule extends ProblemModule
{
    protected void config() {
        bindProblem(OpsAritmeticasCreator.class, OpsAritmeticasDecoder.class, OpsAritmeticasEvaluator.class);
    }
}
```

Paso 5. Ejecución de Opt4J con el nuevo problema diseñado

Cuando ya hemos implementado todo el código que modela nuestro problema, simplemente tenemos que ejecutar nuestra aplicación con la acción de “Run As Java Application”. Para ello haremos click sobre el botón derecho sobre nuestro proyecto Java y podremos ver el menú que aparece en la Figura 5.

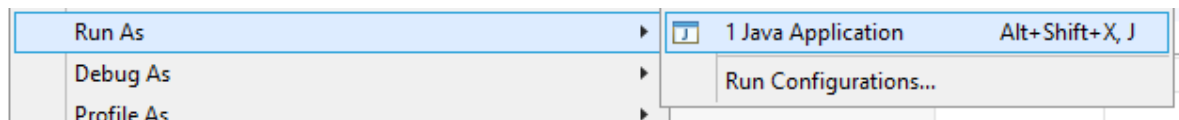


Figura 5. Menú “Run As Java Application” para ejecutar nuestro código en Opt4J

Seguidamente, lo único que tenemos que indicar es el punto de entrada de nuestra aplicación Java, que será **Opt4J – org.opt4j.core.start** (no confundir con **Opt4JStarter**), tal y como se muestra en la Figura 6.

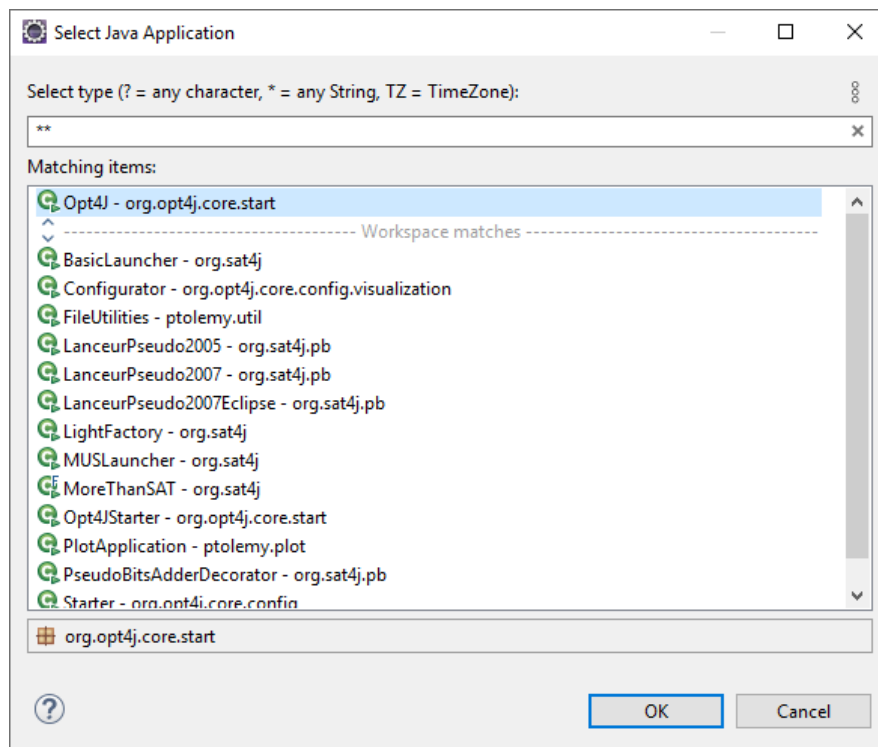


Figura 6. Selección del punto de entrada de nuestra aplicación Java

Si todo ha funcionado correctamente, en la carpeta *Problem* de Opt4J veremos nuestro problema, en este caso con nombre *OpsAritmeticas*. Si seleccionamos dicho problema, el modo de salida *Viewer* y el optimizador *EvolutionaryAlgorithm* podremos configurar el número de generaciones que deseamos, el tamaño de la población, el número de padres por generación, el número de descendientes por generación y el ratio con el que se realizará una operación de cruce entre los padres. Esto se puede observar en la Figura 7 y Figura 8, en las que se muestra la ventana de configuración y la de resolución del problema, respectivamente. En la **sección 3** de este mismo documento se explicarán estos pasos con más detalle.

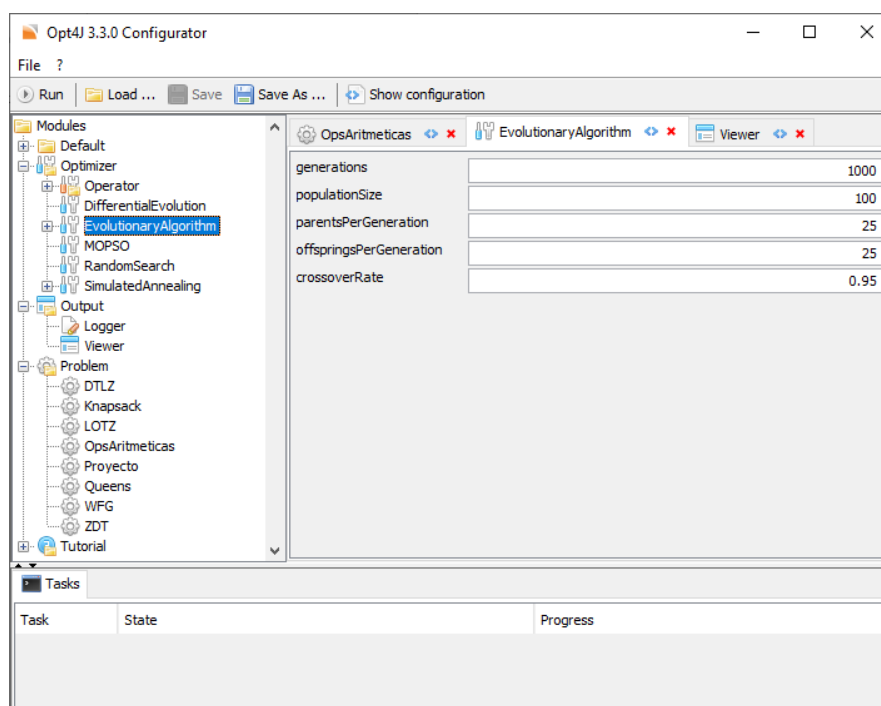


Figura 7. Configuración del problema de *OpsAritmeticas*

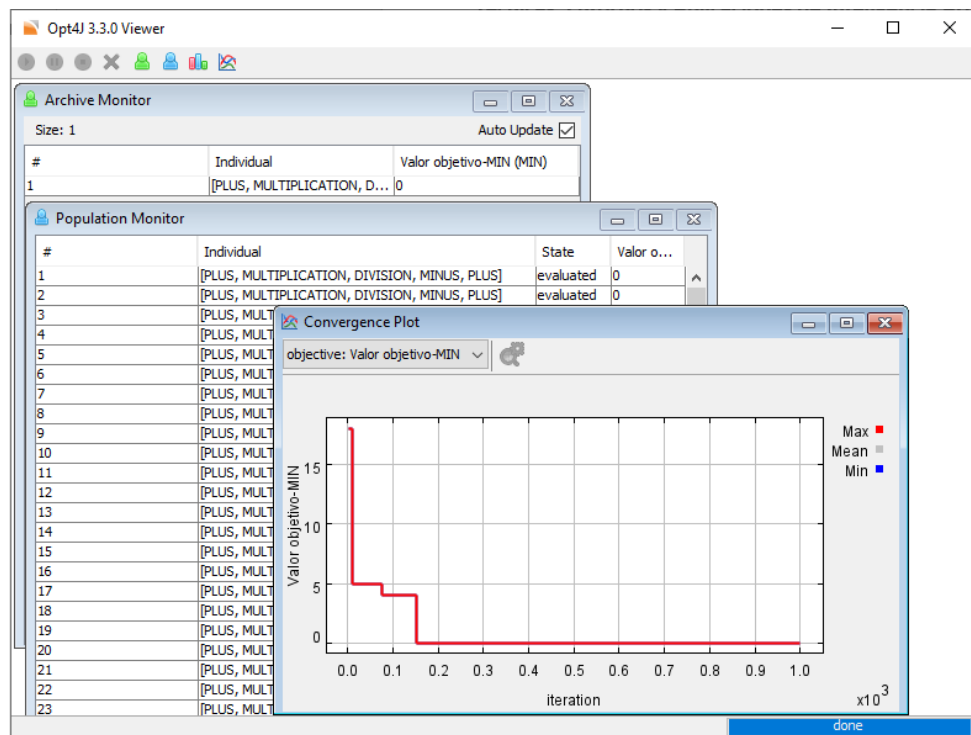


Figura 8. Resolución del problema de OpsAritmeticas

Ejemplo2. Distribución de pesos sobre una estructura o viga

Disponemos de una viga o estructura de metal que tiene que soportar un conjunto de piezas. En función de dónde esté colocada cada pieza, ejercerá una fuerza sobre la estructura u otra. Obviamente, no podemos colocar todas las piezas en un extremo o de forma aleatoria, porque la estructura no estará equilibrada. El problema a resolver consiste en encontrar la posición adecuada para que el reparto de pesos sea óptimo, y se pide diseñar un algoritmo para obtener dicha distribución, o posición exacta, de las piezas.

En nuestro problema concreto disponemos de 7 piezas, que ejercen una fuerza (o peso) de $P_1=0.5$, $P_2=1.7$, $P_3=1.1$, $P_4=0.8$, $P_5=0.15$, $P_6=1.9$ y $P_7=1.4$ unidades cada una.

Metafóricamente hablando, y para simplificar el modelado, estamos resolviendo un problema de equilibrado de pesos en una balanza. Imaginemos una balanza reglada como la de la Figura 9: el centro de la balanza es la posición 0, a la izquierda tenemos valores negativos y a la derecha valores positivos. Lo que nos interesa es que la fuerza ejercida por las piezas sobre un extremo de la balanza se contrarreste con las del otro extremo. En otras palabras, buscamos que el sumatorio de la posición de cada pieza (positiva o negativa) multiplicada por la fuerza que ejerce sea igual, o lo más cercana, al valor 0. Además, por restricciones del problema no queremos que ninguna pieza se ponga en la posición 0. Por simplicidad, asumiremos que la pieza ejerce el peso en un único punto equivalente a su centro de masa.

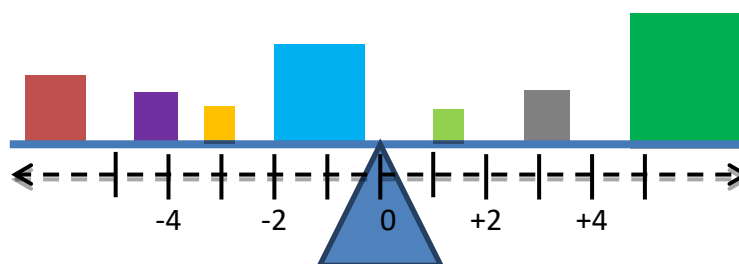


Figura 9. Balanza metafórica que usaremos para modelar el problema

Para comenzar la implementación, en nuestro proyecto Java crearemos un nuevo paquete para contener todo el código de este ejemplo al que denominaremos “distribucion.pesos”.

Paso 1. Creación de la clase Creator

En primer lugar crearemos una clase de Datos con la información inicial del problema, es decir las 7 piezas, la fuerza de cada una de ellas y un intervalo de 8, como posibles posiciones donde colocar las piezas a cada lado de la balanza.

```
package distribucion.pesos;

public class Data
{
    public static final int numeroPiezas = 7; // numero de piezas
    public static final double fuerzaPieza[] = {0.5, 1.7, 1.1, 0.8, 0.15, 1.9, 1.4}; // fuerza que ejerce cada pieza
    public static final double MAXINTERVAL = 8; // tamaño del intervalo
}
```

En la clase Creator debemos definir el genotipo. En nuestro caso, la codificación del individuo es la posición de cada pieza como un valor double en el rango [-8, +8]. Por lo tanto, utilizaremos para la codificación del genotipo el tipo predefinido en Opt4J 'DoubleGenotype' (si la posición tuviera una granularidad en valores enteros podríamos usar IntegerGenotype).

De esta forma, un individuo [0.5, -7.8, 2.9, 4.7, 6.5, -3.5, -4.8] indica que la primera pieza estaría colocada en la posición 0.5, la segunda en -7.8, la tercera en 2.9 y así sucesivamente.

El código de la clase Creator resulta:

```
public class DistPesosCreator implements Creator<DoubleGenotype>
{
    public DoubleGenotype create()
    {
        DoubleGenotype genotipo = new DoubleGenotype(-Data.MAXINTERVAL, Data.MAXINTERVAL); // rango de posiciones [-8..8]

        // el genotipo estara formado por 7 posiciones elegidas al azar en el rango [-8..8]
        // en nuestro caso la poblacion sera un conjunto de individuos, donde cada individuo son 7 doubles
        genotipo.init(new Random(), Data.numeroPiezas);

        return genotipo;
    }
}
```

Paso 2. Creación de la clase Decoder

La clase Decoder convierte el genotipo en su fenotipo correspondiente, que en nuestro caso será una simple colección de 7 valores de doubles representando la posición de las piezas. El código es simple:

```
public class DistPesosDecoder implements Decoder<DoubleGenotype, ArrayList<Double>>
{
    @Override
    public ArrayList<Double> decode(DoubleGenotype genotipo)
    {
        ArrayList<Double> fenotipo = new ArrayList<Double>();

        // copiamos los valores del DoubleGenotype en la coleccion
        for (int id = 0; id < genotipo.size(); ++id)
        {
            fenotipo.add(genotipo.get(id));
        }

        return fenotipo;
    }
}
```

Paso 3. Creación de la clase Evaluator

Esta clase tiene que calcular el valor de fitness de cada fenotipo para una función mono-objetivo que queremos que se aproxime a cero tanto como sea posible, representando el equilibrio de pesos óptimo a ambos lados de la balanza. Por restricciones del problema no deseamos piezas colocadas en la posición 0. Por tanto, si ese es el caso penalizaremos el valor del fitness haciendo que sea muy elevado, es decir el máximo valor permitido para un double.

El código de la clase es el siguiente:

```

public class DistPesosEvaluator implements Evaluator<ArrayList<Double>>
{
    @Override
    public Objectives evaluate(ArrayList<Double> fenotipo)
    {
        double resultado = 0.0;

        for (int id = 0; id < fenotipo.size(); ++id)
        {
            if (fenotipo.get(id) == 0.0)
            {
                // restriccion del problema: no queremos piezas colocadas en posicion 0.0
                resultado = Double.MAX_VALUE;
                break;
            }

            resultado += fenotipo.get(id) * Data.fuerzaPieza[id];
        }

        // queremos minimizar el resultado (lo mas cercano a 0 en valor absoluto)
        Objectives objectives = new Objectives();
        objectives.add("Distribucion pesos-MIN", Sign.MIN, Math.abs(resultado));

        return objectives;
    }
}

```

Paso 4. Creación de la clase Module

La clase que crea el módulo para el problema simplemente hará uso de las tres clases creadas anteriormente:

```

public class DistPesosModule extends ProblemModule
{
    protected void config()
    {
        bindProblem(DistPesosCreator.class, DistPesosDecoder.class, DistPesosEvaluator.class);
    }
}

```

Lo único que resta es ejecutar nuestro código (“Run As Java Application”), que lanzará Opt4J y nos permitirá configurar y resolver el problema.

3.- Aplicación del AG y resolución de problemas con Opt4J

Una vez modelado el problema, creando las clases en Java y lanzado Opt4J mediante la opción “Run As Java Application”, ya podemos resolver el problema utilizando el AG incluido en Opt4J. Esto es una clara ventaja de Opt4J y es que estamos **independizando** el modelado del problema de su algoritmo de resolución. Los pasos a realizar son los siguientes:

- a) Elección del problema. Una vez lanzado Opt4J, seleccionaremos con doble-click el problema (**Problem**) de la carpeta de problemas (Figura 10).

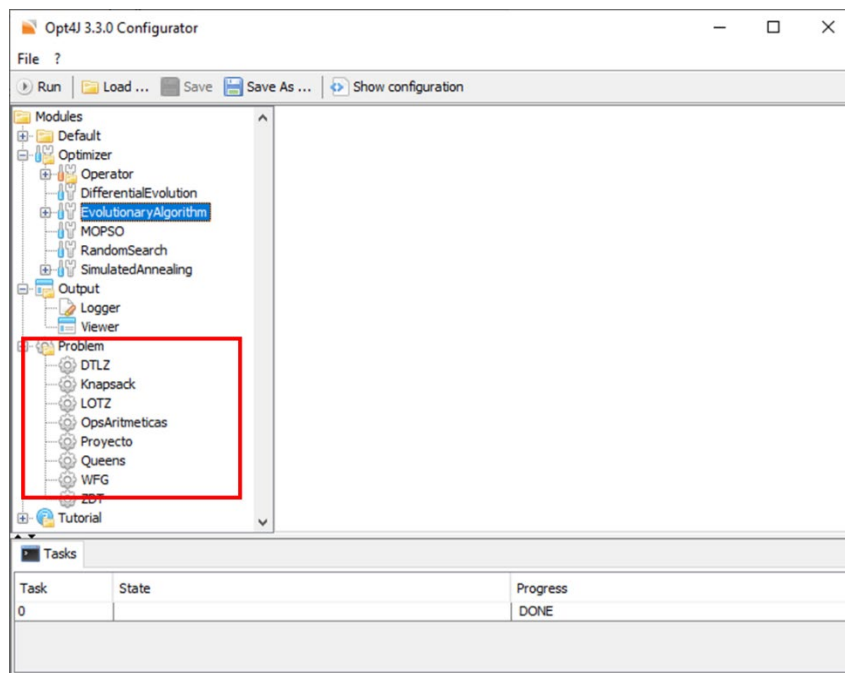


Figura 10. Opt4J con varios problemas de ejemplo

- b) Elección de la forma de visualizar los resultados, mediante doble-click desde la carpeta **Output**. Se puede seleccionar *Logger* (que volcará los resultados a un fichero texto) o *Viewer* (que ofrece una visualización gráfica del progreso del proceso de optimización). Esta última es la opción recomendada.
- c) Finalmente, elección de la metaheurística '*EvolutionaryAlgorithm*' mediante doble-click desde la carpeta **Optimizer**. Para esta metaheurística se pueden configurar sus parámetros según se muestra en la Figura 11.

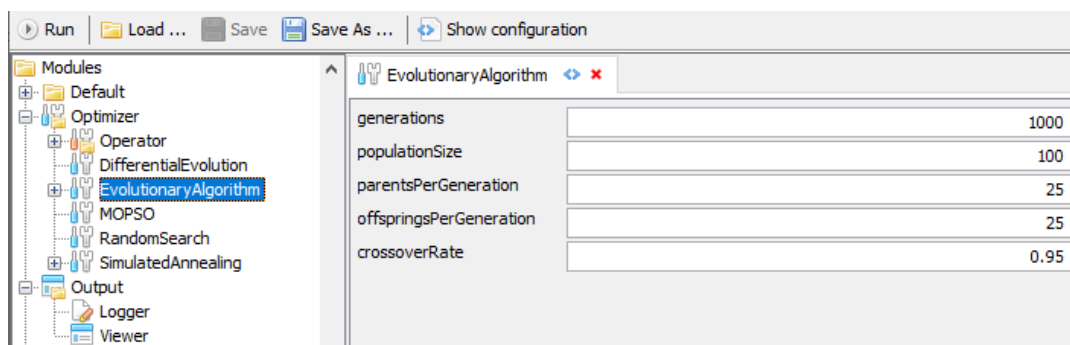


Figura 11. Selección del método AG con sus parámetros a configurar

Los parámetros configurables del AG en Opt4J son (para más información se puede consultar la documentación en <https://sdarg.github.io/opt4j/documentation.html>):

- generations: número de generaciones (iteraciones) a realizar;
- populationSize: número de individuos de la población;
- parentsPerGeneration: número de padres que se seleccionarán en cada iteración;
- offspringsPerGeneration: número de hijos que se generarán en cada iteración;
- crossoverRate: probabilidad de que dos padres se crucen.





En el método del AG implementado en Opt4J, en la siguiente iteración solo sobreviven los individuos no cruzados (de acuerdo a los valores de parentsPerGeneration y crossoverRate). Es decir, un valor crossoverRate=1 significa que todos los individuos seleccionados (parentsPerGeneration) se usan en la reproducción y no sobrevivirán.

Empíricamente, se recomienda una probabilidad de cruce entre 0.65 y 0.85, lo que implica que la probabilidad de que un individuo seleccionado sobreviva a la siguiente generación varía de 0.35 a 0.15.

En la parametrización de la Figura 11 habrá 1000 iteraciones, la población es de 100 individuos, en cada iteración se seleccionan 25 padres, de los cuales se cruzan el 95%, y se generan 25 hijos como máximo. Un 5% de los padres seleccionados (los que no se habrán cruzado) sobrevive en la siguiente iteración (en la cual estarán los 25 hijos generados, y el resto de individuos no seleccionados para cruzarse).

Ejemplo1. Operaciones aritméticas

Resolveremos el primer ejemplo, seleccionando en primer lugar el algoritmo de búsqueda aleatoria (*RandomSearch*). Cuando se ejecuta el proceso (run), se abre el visor (Viewer). En el visor hay 4 botones que nos muestran distintas ventanas de resultados (Figura 12, Figura 13 y Figura 14).

-  Mejor individuo de la población
-  Monitor de la población (en cada generación)
-  Gráfico de convergencia al objetivo
-  Frontera de Pareto (solo para problemas multi-objetivo).

En este caso podemos ver que se ha encontrado la solución óptima “+, *, /, -, *”, con valor del objetivo 0 (lo que es razonable, ya que solo tenemos 4⁵ posibles soluciones).

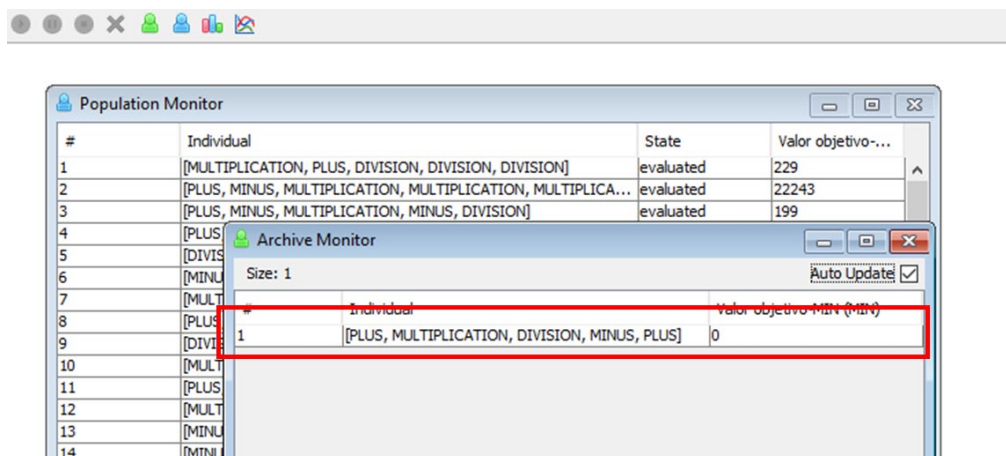


Figura 12. Solución al Ejemplo1 mediante búsqueda aleatoria

En la Figura 13 podemos observar los resultados seleccionando el algoritmo genético (*EvolutionaryAlgorithm*) que también encuentra la solución óptima. En ambas Figuras se puede ver cómo se han evaluado muchos individuos con distintos valores para la función objetivo.

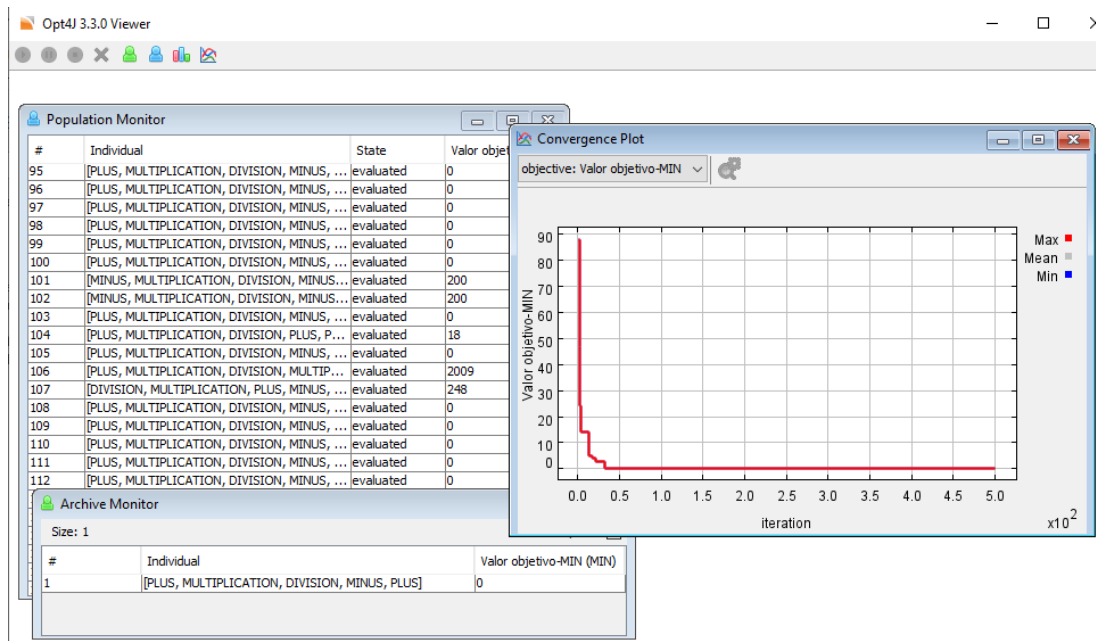


Figura 13. Solución al Ejemplo1 mediante algoritmo genético (EvolutionaryAlgorithm)

Supongamos ahora que queremos resolver la variante de este problema en su versión multi-objetivo. Es decir, buscamos aquella solución que se acerque al número objetivo y a la vez tenga el mayor número de símbolos “+” posible. Para ello, tendríamos que modificar el código de la clase Evaluator y volver a lanzar la aplicación.

El resultado de resolver esta variante con un algoritmo genético se muestra en la Figura 14. La solución ahora ya no es tan simple pues tenemos dos funciones objetivo (que habitualmente suelen ser contradictorias). El mejor individuo que optimiza la primera función objetivo puede no optimizar la segunda, como se observa en la Figura 14. En este caso se habla de optimización de Pareto, porque una solución no puede mejorar una función objetivo si no es a costa de empeorar la otra.

Por ejemplo, en la Figura 14 se puede observar como la segunda solución es óptima con respecto a encontrar el valor exacto del resultado de la operación (0), pero no con respecto a maximizar el número de símbolos “+” (2). Si queremos maximizar dicho número tendremos que empeorar el valor de la función objetivo a minimizar (como ocurre con la solución 3 y la 1). Concretamente, la solución 3 mejora el número de símbolos “+” (3) pero empeora el valor de la segunda función (18). Y la solución 1 mejora todavía más el número de símbolos “+” (5) pero empeora todavía más el valor de la segunda función (98). En este caso, es **decisión del usuario** seleccionar la solución que mejor se adapte a sus necesidades. Y esta situación ante optimización multi-objetivo ocurre aunque variemos el algoritmo de resolución.

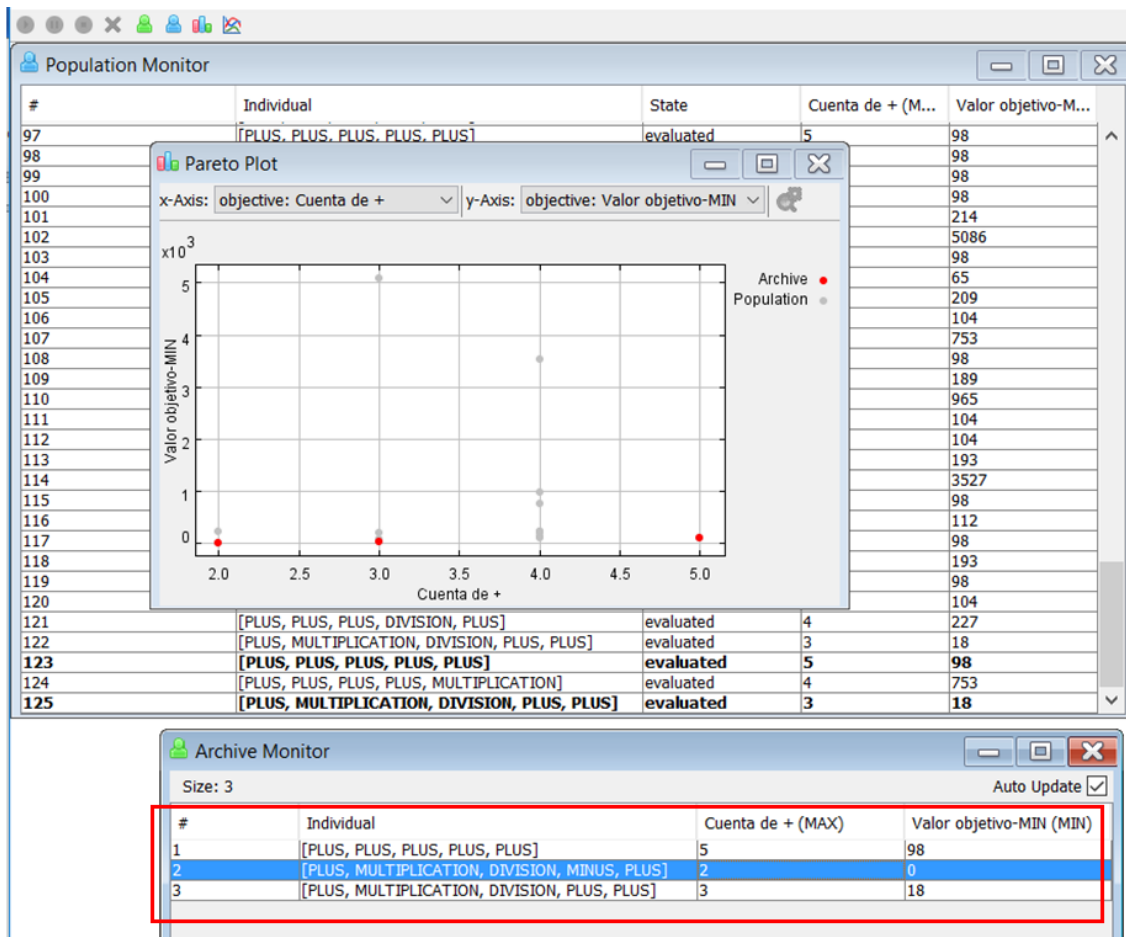


Figura 14. Solución al Ejemplo1 multi-objetivo mediante algoritmo genético (EvolutionaryAlgorithm)

Ejemplo2. Distribución de pesos sobre una estructura

A continuación, resolveremos el segundo ejemplo utilizando un algoritmo genético (ver Figura 15). Puede observarse que, aunque no se ha encontrado la solución de equilibrio óptimo de fuerzas (0.0) en la colocación de las piezas, sí se ha encontrado una solución bastante buena: 0.00005601.

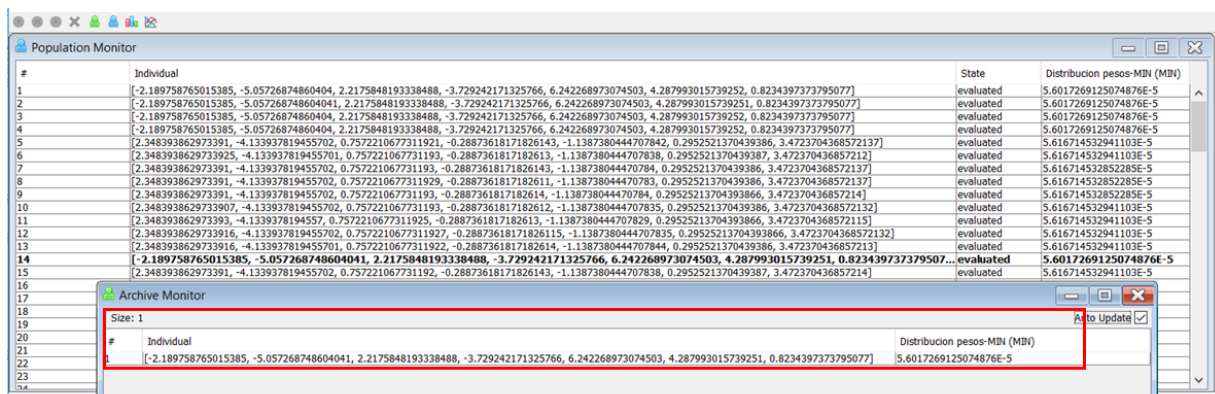


Figura 15. Solución al Ejemplo2 mediante algoritmo genético (Evolutionary Algorithm)

Evaluación de la Práctica. Problema Propuesto

Antes de realizar el desarrollo de la práctica es imprescindible leer el documento anterior para comprender el modelado de problemas y funcionamiento de Opt4J.

OBJETIVOS

El objetivo de la práctica comprende los siguientes aspectos:

- 1) Diseñar los elementos fundamentales de un problema dado (individuos y función fitness).
- 2) Modelar en Java el código necesario para representar al Genotipo, Fenotipo y función Fitness, tal y como se ha descrito en el documento anterior.
- 3) Resolver el problema mediante el algoritmo genético incluido en Opt4J,
- 4) Evaluar dicho algoritmo genético contrastando los resultados obtenidos mediante diversas parametrizaciones del AG modificando la parametrización desde Opt4J.

Se recomienda utilizar el **proyecto-plantilla disponible en Poliformat tal y como se describe en la sección 1.3**. Tras el desarrollo de la práctica, su evaluación consistirá en rediseñar y evaluar pequeñas modificaciones del problema propuesto.

Ejercicio. Distribución de tareas eléctricas

Ante la incesante subida de los precios de la electricidad y la diversidad de tramos (punta, llano y valle/reducido), una pequeña empresa de mecanizado industrial desea poder distribuir de manera eficiente (y deseablemente óptima) la fabricación de un conjunto de productos de su catálogo. Fabricar cada producto requiere de un conjunto distinto de máquinas con distintas necesidades de consumo eléctrico, repercutiendo en un beneficio para la empresa tras la venta del producto. Además, hay que tener en cuenta los precios y potencias máximas contratadas en cada uno de los tres tramos existentes.

En el problema que nos ocupa se utilizan tres tramos, aunque es posible que en el futuro se puedan añadir nuevos tramos. Para cada tramo se define una potencia máxima contratada, medida en kilo vatios (kW), que no se puede exceder, y un precio por kilovatio hora (kWh). Así, por ejemplo, si durante un tramo completo deseamos fabricar dos productos con un consumo individual de 10 kWh y tenemos contratada una potencia máxima de 20 kW, no dispondremos de más potencia para fabricar otros productos durante ese tramo. Por otro lado, el coste por energía consumida que supondrá fabricar esos dos productos será: $(10+10) \cdot n^{\circ} \text{ horas del tramo} \cdot \text{precio de kWh en ese tramo}$. **Nota importante:** la potencia máxima es un término fijo (que no depende de las horas que dura el tramo), mientras que la energía consumida (y su coste) es un término variable, pues sí depende del nº horas del tramo.

Actualmente, la empresa tiene contratado tres tramos distintos de 4 horas cada uno, correspondientes con sus turnos de trabajo. Los detalles de cada tramo se dan en la siguiente tabla

	Tramo1	Tramo2	Tramo3
Nombre	Punta	Llano	Valle/reducido
Horario	10:00-14:00	15:00-19:00	06:00-10:00
Potencia máxima contratada (kW)	15	18	25
Precio del kWh (€)	0.26	0.18	0.13

La empresa desea fabricar 20 productos (P1..20), aunque se debe realizar un diseño e implementación flexible que permita extender fácilmente el número de productos. El consumo individual y beneficio que se obtiene de la fabricación de cada producto se define en la siguiente tabla:

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	P17	P18	P19	P20
Consumo (kWh)	1.9	2.1	3.0	0.7	1.5	3.3	4.2	2.6	2.3	3.2	4.5	4.2	2.7	1.9	3.5	2.7	3.4	4.5	6.2	2.3
Beneficio (€)	3	5	6	1	3	4	9	3	4	4	8	7	4	3	4	5	4	6	9	4

Por ejemplo, el producto P1 consume 1.9 kWh (por cada hora del tramo). Ya que estamos considerando que **un producto se fabrica durante todo un tramo**, si se fabrica en el tramo 1 su coste de fabricación será: $1.9 \times 0.26 \times 4 \text{ horas} = 1.976\text{€}$. Por el contrario, si se fabrica en el tramo 2 su coste será: $1.9 \times 0.18 \times 4 \text{ horas} = 1.368\text{€}$. Por lo tanto, el beneficio de P1 si se fabrica en el tramo 1 es de $3 - 1.976 = 1.024\text{€}$ y de $3 - 1.368 = 1.632\text{€}$ si se fabrica en el tramo 2.

Como puede observarse, no es posible fabricar los 20 productos en los tres tramos existentes, ya que se requiere un consumo total para P1..P20 de 60.7 kWh y la potencia máxima contratada con los tres tramos es solo de 58 kW. Esto quiere decir que habrá que: 1) decidir qué productos se fabrican y cuáles se dejan para otra jornada para impedir que se supere la potencia máxima contratada¹, y 2) distribuir la fabricación de los productos entre los tres tramos para obtener el máximo beneficio posible.

NOTA. La mejor solución que hemos obtenido para este problema ha sido de beneficio 51.272€, que fabrica todos los productos excepto el P15. Evidentemente, esto no significa que tenga que ser necesariamente la solución óptima.

Después de analizar los resultados obtenidos, la empresa ha decidido probar una nueva política de fabricación. Concretamente, ahora interesa encontrar la distribución de fabricación de productos que **maximiza** el beneficio total y también **maximiza** el número de productos fabricados durante los tramos diurnos (primer y segundo tramo).

NOTA. Ahora no existe una única mejor solución porque la alternativa que es óptima para el beneficio no lo es para el número de productos fabricados en tramos diurnos, y viceversa. Por ejemplo, se han encontrado las siguientes soluciones de Pareto en la forma <beneficio, número de productos fabricados>: <49.04, 10>, <47.64 11>, < 47.09, 12>, etc.

Opcionalmente, pueden probarse distintas variaciones del problema propuesto.

NOTA IMPORTANTE: se recomienda una implementación flexible y trabajar con vectores para facilitar posibles ampliaciones en el número de tramos disponibles, pedidos, consumos, precios, etc.

¹ Se recomienda utilizar un *tramo simbólico* para representar que el producto no se fabrica en ninguno de los tres tramos y se deja para la próxima jornada laboral.