Práctica 4

Herencia, interfaces y excepciones

Inicio: Semana del 19 de marzo.
Duración: 3 semanas.
Entrega: Semana del 16 de abril
Peso de la práctica: 30%

El objetivo de esta práctica es el **diseño e implementación de un algoritmo conocido como programación genética**, utilizando técnicas de programación orientada a objetos más avanzadas que en las prácticas precedentes, como son la herencia, la ligadura dinámica, las excepciones y las *interfaces*. Se trata de servirse de estas técnicas para reducir código redundante y desarrollar un programa bien estructurado que refleje de una forma directa los conceptos del dominio del problema.

En esta práctica se utilizarán principalmente los siguientes conceptos de Java y de programación orientada a objetos:

- *Herencia y ligadura dinámica*
- *Manejo de excepciones*
- *Uso de interfaces Java*
- *Estructuración en paquetes*
- *Entrada/Salida*

Introducción

La programación genética [1] es una rama de la computación evolutiva que trata de aprovechar conceptos propios de la biología para desarrollar algoritmos cuyo objetivo es resolver algunos problemas computacionales. Sin extendernos demasiado en los detalles, estos algoritmos crean aleatoriamente una población de "**individuos**", que representan de alguna forma el dominio del problema, y realizan con ellos una serie de *operaciones* y *evaluaciones* tratando de buscar una solución al problema. Por tanto, y a efectos prácticos, los principales aspectos a tratar cuando se diseña un algoritmo de programación genética son tres:

1. Representar el dominio del problema de forma adecuada.
2. Seleccionar el conjunto de *operadores genéticos* a aplicar.
3. Determinar la *forma en que se evaluarán los individuos*.
4. Determinar la *condición que permitirá decidir si se continúa o no la búsqueda* de la solución al problema.

La representación del problema es fundamental a la hora de escribir el algoritmo. En la programación genética el problema se describe en términos de los conjuntos de ***terminales*** y de ***funciones***. Cada uno de estos conjuntos debe incluir los elementos adecuados. Un individuo se describe a partir de estos conjuntos y se representa generalmente en forma de árbol.

Los **operadores genéticos** más habituales son el *cruce* y la *mutación*. El primero parte de dos individuos y genera dos descendientes que contienen material mezclado de ambos padres. El segundo altera la representación del individuo de forma aleatoria.

El algoritmo termina cuando se llega a un número especificado de iteraciones o se encuentra una solución al problema. La forma de evaluar a los individuos determina si dicha solución podrá o no resolverse empleando este tipo de algoritmo.

El problema a resolver en esta práctica mediante programación genética es una regresión simbólica, que consiste en averiguar la forma de la ecuación $X^4+X^3+X^2+X$ a partir de un conjunto de datos de la función real en el rango [-10,10]. Aunque se trate de un problema establecido de antemano, debemos intentar realizar un diseño que permita al algoritmo resolver problemas en dominios similares sin realizar excesivos cambios.

Apartado 1: Terminales, Funciones, Nodos e Individuos (3,5 puntos)

En este apartado se implementan la representación para los individuos que componen la población a evolucionar. Como se ha indicado en la introducción, la representación del problema es una cuestión fundamental en el desarrollo del algoritmo genético. En el caso de la programación genética los individuos se describen a partir de un conjunto de Terminales y de Funciones.

Los **terminales** son símbolos finales que no pueden expandirse, mientras que las **funciones** pueden tener argumentos, que serán a su vez, funciones o terminales.

Un *Individuo* estará compuesto de *Nodos*, los cuales podrán ser *Terminales* o *Funciones*. La inclusión del concepto de Nodo nos permitirá recoger aspectos comunes de los Terminales y las Funciones, como puede ser el nombre del símbolo, y también permitirá definir una serie de operaciones comunes a ambos conceptos. La interfaz *INodo* describe algunos métodos que puedes incluir en la clase que la implemente.

```
public interface INodo {
     public String getRaiz();
     public List<INodo> getDescendientes();
     public void incluirDescendiente(INodo nodo);
     public double calcular();
     public INodo copy();
}
```

El método *getRaiz* devuelve el símbolo que representa el nodo, es decir, el símbolo del Terminal o la Función.

El método *getDescendientes* devuelve los argumentos, en el caso de las funciones. En el caso de los terminales, la lista de descendientes estará vacía.

El método *incluirDescendiente* permite añadir un nuevo nodo (Terminal o Función) a un nodo existente-. Su argumento es el nodo a incluir en la lista de descendientes.

El método *calcular()* es muy importante ya que permite devolver un valor asociado al nodo. En el caso de un terminal será simplemente un valor determinado, pero en el caso de las funciones debería resolver alguna operación.

Finalmente, el método *copy* permite hacer una copia del nodo. Es muy importante hacer copias para evitar sorpresas desagradables a la hora de realizar ciertas operaciones con los nodos.

Como se ha explicado, los individuos se componen de nodos. La interfaz *IIndividuo* define algunos métodos relevantes en este concepto.

```
public interface IIndividuo {
     public INodo getExpresion();
     public void setExpresion(INodo expresion);
     public double getFitness();
     public void setFitness(double fitness);
     public void crearIndividuoAleatorio(int profundidad, List<Terminal> terminales, List<Funcion> funciones);
     public double calcularExpresion();
     public int getNumeroNodos();
     public void writeIndividuo();
}
```

El método *getExpresion* devuelve un Nodo que representa la raíz, es decir, el Nodo inicial del individuo. Denominaremos expresión a la representación del individuo mediante nodos. El método *setExpresion* es autoexplicativo.

El método *getFitness* devuelve el "**fitness**" del individuo. Este valor representa la aptitud del individuo para resolver el problema. **En esta práctica, el fitness se obtiene calculando la expresión aritmética que representa el individuo sobre un conjunto de datos de prueba y comparar cada resultado estimado con el resultado real**.

El método *crearIndividuoAleatorio* construye un individuo aleatorio y se utiliza cuando se crea la población inicial, como veremos cuando se describa el algoritmo. Existen varios métodos para crear un individuo pero aquí seguiremos uno relativamente sencillo que consiste en construir individuos con un tamaño determinado, especificado por el argumento profundidad (niveles del árbol que representa el individuo). El conjunto de terminales y funciones con los que construir el individuo se pasan como argumentos.

El método *calcularExpresion* realiza la operación representada por la expresión.

El método *getNumeroNodos* devuelve el número de nodos totales del individuo.

El método *writeIndividuo* muestra la expresión que corresponde al individuo.

Los conceptos de Individuo y Nodo quedan más claros a partir de los Terminales y Funciones que los componen. Se recomienda que los terminales y las funciones se estructuren en diferentes niveles, de manera que el diseño sea lo más flexible posible. Por ejemplo, ciertas operaciones son comunes a ambos tipos, con independencia del problema y su representación, de manera que se podrían crear clases abstractas para estos conceptos y especializar comportamientos en función del problema. En el caso de la regresión simbólica, los terminales y las funciones deben ser operadores y operandos aritméticos. En concreto, como terminal solo se usará el símbolo "x", y como funciones usaremos la suma ("+"), la resta ("–") y el producto ("*"). Las tres funciones tienen dos argumentos, que podrán ser un terminal u otra función aritmética. A modo de ejemplo, el programa siguiente debería producir la salida de más abajo.

```
public class TesterIndividuos {

    public static void main(String[] args) {
        Terminal x = new TerminalAritmetico("x");
        Funcion suma = new FuncionSuma("+", 2);
        Funcion resta = new FuncionResta("-", 2);
        Funcion multi = new FuncionMultiplicacion("*", 2);

        multi.incluirDescendiente(x);
        multi.incluirDescendiente(x);
        suma.incluirDescendiente(multi);
        suma.incluirDescendiente(x);
        resta.incluirDescendiente(suma);
        resta.incluirDescendiente(multi);

        System.out.println("Función multiplicación: " + multi);
        System.out.println();
        System.out.println("Función suma: " + suma);
        System.out.println();
        System.out.println("Función resta: " + resta);

        IIndividuo indiv = new Individuo();
        indiv.setExpresion(resta);
        System.out.println();
        System.out.println("INDIVIDUO");
        indiv.writeIndividuo();
    }
}
```

Salida esperada:

```
Función multiplicación: ( * x x )

Función suma: ( + ( * x x ) x )

Función resta: ( - ( + ( * x x ) x ) ( * x x ) )

INDIVIDUO
Expresión: ( - ( + ( * x x ) x ) ( * x x ) )
```

En la salida se ha utilizado la **notación prefija**, en la que el operador aparece antes de los operandos. Simplemente es una cuestión de formato que puede o no seguirse en la representación interna. Como puedes ver en el tester, la clase *TerminalAritmético* representa un Terminal para el caso concreto de la regresión simbólica. De la misma forma, las clases *FuncionSuma*, *FuncionResta* y *FuncionMultiplicacion* corresponden a una representación adecuada para este problema. Fíjate que en el primer caso se pasa como argumento el símbolo del terminal y en el segundo se pasan tanto el símbolo, como los argumentos. Otro aspecto a destacar es que el método *incluirDescendiente* que se propone para los Nodos tiene sentido cuando se trabaja con Funciones. A partir de estas estructuras sería recomendable que el algoritmo de programación genética pudiera definir su conjunto de Terminales y Funciones de manera lo más genérica posible, abstrayendo el dominio de representación del propio algoritmo.

Apartado 2: Dominios, Cálculo del Fitness y Lectura de Valores de Prueba (2 puntos)

De la misma forma que los terminales y funciones son específicos de cada problema, la forma de evaluar los individuos también depende del mismo problema. En el caso de la regresión simbólica, un individuo no es más que una expresión aritmética que debe evaluarse frente a un conjunto de datos de prueba, que se utilizarán para que el terminal "x" tenga un valor concreto. Otros

problemas tendrán representaciones y evaluaciones distintas. Intenta abstraer el detalle de la evaluación en la medida de lo posible. Una sugerencia para lograrlo sería implementar una interfaz *IDominio*, que contiene métodos aplicables a diferentes casos.

```
public interface IDominio {
     public List<Terminal> definirConjuntoTerminales(String... terminales);
     public List<Funcion> definirConjuntoFunciones(int[] argumentos, String... funciones) throws
                                                                                    ArgsDistintosFuncionesException;
     public void definirValoresPrueba(String ficheroDatos) throws FileNotFoundException, IOException;
     public double calcularFitness(IIndividuo individuo);
}
```

Los dos primeros métodos crean un conjunto de terminales y de funciones apropiado para el problema y serían llamados por el algoritmo genético para definir ambos conjuntos. Fíjate en que para construir el conjunto de funciones se pasan los argumentos de cada función. Si no coinciden en número ambos argumentos se lanzará una excepción *ArgsDistintosFuncionesException*.

El método *definirValoresPrueba* establece el conjunto de datos de prueba para los cálculos. En esta práctica, estamos tratando de descubrir la ecuación $X^4+X^3+X^2+X$ a partir de un conjunto de datos de la función real en el rango [-10,10]. Esta información real se encuentra almacenada en un fichero **valores.txt** que contiene dos columnas, la primera con el valor del símbolo "x" y la segunda con el resultado real de la ecuación. Las excepciones que puede devuelve este método son las habituales al tratar con E/S. Las primeras tres filas de este fichero serían:

```
-10        9090
-9         5904
-8         3640
…..
```

Para evaluar un individuo como `( + ( * x x ) x )`, equivalente a X^2+X usaríamos los valores -10, -9, -8… y obtendríamos 90, 72 y 56 que están bastante alejados de los valores reales. El fitness del individuo sería la suma de los aciertos en los 20 valores de prueba, considerando un margen de error entre el valor estimado y el real. De la realización de estos cálculos se encarga precisamente el método *calcularFitness*, que recibe el individuo para el que se tiene calcular el fitness. A modo de ejemplo, el tester que se muestra a continuación, prueba la lectura y el cálculo del fitness para un individuo en el problema de la regresión simbólica.

```
public class TesterLecturaYFitness {

     public static void main(String[] args) throws IOException, IOException {
          IDominio    domAritm;
          double      fitness;

          domAritm = new DominioAritmetico();
          domAritm.definirValoresPrueba("valoresReducido.txt");
          Terminal x = new TerminalAritmetico("x");
          Funcion suma = new FuncionSuma("+", 2);
          Funcion resta = new FuncionResta("-", 2);
          Funcion multi = new FuncionMultiplicacion("*", 2);

          multi.incluirDescendiente(x);
          multi.incluirDescendiente(x);
          suma.incluirDescendiente(multi);
          suma.incluirDescendiente(x);
          resta.incluirDescendiente(suma);
          resta.incluirDescendiente(multi);

          IIndividuo indiv = new Individuo();
          indiv.setExpresion(resta);
          System.out.println();
          System.out.println("INDIVIDUO");
          indiv.writeIndividuo();
          System.out.println();
          fitness = domAritm.calcularFitness(indiv);
          System.out.println("\nFITNESS= "+fitness);
     }

}
```

La salida esperada sería la siguiente:

```
INDIVIDUO
Expresión: ( - ( + ( * x x ) x ) ( * x x ) )

Valor -3.0 <-> Rdo estimado: -3.0 <-> Rdo real: 60.0
Valor -2.0 <-> Rdo estimado: -2.0 <-> Rdo real: 10.0
Valor -1.0 <-> Rdo estimado: -1.0 <-> Rdo real: 0.0
Valor 0.0 <-> Rdo estimado: 0.0 <-> Rdo real: 0.0
Valor 1.0 <-> Rdo estimado: 1.0 <-> Rdo real: 4.0
Valor 2.0 <-> Rdo estimado: 2.0 <-> Rdo real: 30.0
Valor 3.0 <-> Rdo estimado: 3.0 <-> Rdo real: 120.0

FITNESS= 2.0
```

El fichero valoresReducido.txt es una versión del fichero valores.txt considerando solo el rango [-3,3]. Se ha considerado como margen de error una diferencia de ±1.

Apartado 3: Operadores Genéticos: Cruce (2 puntos)

Los operadores más habituales en los algoritmos evolutivos son el ***cruce*** y la ***mutación***. En esta práctica solo vamos a implementar el cruce, esperando que sea suficiente para resolver el problema. La idea de este operador es sencilla. Se toman dos progenitores, es decir, dos individuos, y se crean dos descendientes a partir de ellos.

Para implementar el cruce se define un punto de cruce en cada uno de los progenitores y posteriormente se intercambian partes de cada uno de ellos. De esta forma se obtienen los dos descendientes que serán una mezcla de los progenitores. De forma gráfica este operador se aplicaría como se muestra en la Figura 1.

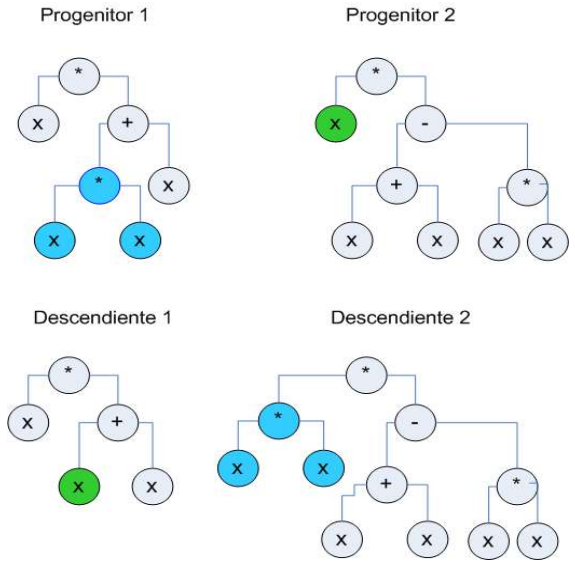

Figura 1: Aplicación del operador de cruce

Para implementar el operador de cruce puede resultarte útil crear un método que permita etiquetar los nodos siguiendo un orden determinado, por ejemplo, recorriendo el árbol siguiendo la estrategia primero en profundidad de izquierda a derecha. El etiquetado del individuo, más concretamente de sus nodos, puede facilitar la búsqueda del punto de cruce y el intercambio de ramas en los árboles. Ten en cuenta que esta operación depende de tu diseño de la clase Nodo y deberías tener cuidado a la hora de borrar y añadir ramas si lo haces a través de la lista de descendientes del nodo. El objetivo del operador de cruce es el de crear nuevos individuos a partir de la población inicial. Gracias a su aplicación el algoritmo evoluciona las estructuras iniciales en la búsqueda de una solución al problema. Sin embargo, la aplicación del cruce no basta para lograr este objetivo. El efecto combinado del cruce y la evaluación del fitness son los elementos a través de los que el algoritmo trata de encontrar la solución.

En este apartado basta con crear un método de cruce que reciba dos *Individuos* (progenitores) y devuelva una lista con dos *Individuos* (descendientes). El punto de cruce de cada progenitor se elige de forma aleatoria, con un valor entre 1 y el número máximo de nodos del progenitor. Si el punto de cruce aleatorio en los dos casos es 1, se emitirá una excepción de *CruceNuloException* y se devolverán los progenitores iniciales. Para probar este método podemos crear utilizar este tester:

```
public class TesterCruce {

    public static void main(String[] args) {
        PruebaCruce prueba = new PruebaCruce();
        List<IIndividuo> descendientes = new ArrayList<IIndividuo>();
        Terminal x = new TerminalAritmetico("x");
        Funcion suma = new FuncionSuma("+", 2);
        Funcion resta = new FuncionResta("-", 2);
        Funcion multi = new FuncionMultiplicacion("*", 2);

        multi.incluirDescendiente(x);
        multi.incluirDescendiente(x);
        suma.incluirDescendiente(multi);
        suma.incluirDescendiente(x);
        resta.incluirDescendiente(suma);
        resta.incluirDescendiente(multi);

        IIndividuo prog1 = new Individuo();
        prog1.setExpresion(resta);
        prog1.etiquetaNodos();
        IIndividuo prog2 = new Individuo();
        prog2.setExpresion(suma);
        prog2.etiquetaNodos();
        System.out.println();
        System.out.println("PROGENITOR 1");
        prog1.writeIndividuo();
        System.out.println("PROGENITOR 2");
        prog2.writeIndividuo();
        try {
            descendientes = prueba.cruce(prog1, prog2);
            System.out.println();
            System.out.println("DESCENDIENTE 1");
            descendientes.get(0).writeIndividuo();
            System.out.println("DESCENDIENTE 2");
            descendientes.get(1).writeIndividuo();
        } catch (CruceNuloException e) {
            e.printStackTrace();
        }
    }
}
```

Y una posible salida sería la siguiente. Ten en cuenta que el punto de cruce se elige de forma aleatoria, por lo que la salida puede variar entre ejecuciones.

```
PROGENITOR 1
Expresión: ( - ( + ( * x x ) x ) ( * x x ) )
PROGENITOR 1
Expresión: ( + ( * x x ) x )

Punto de cruce del progenitor 1: 7
Punto de cruce del progenitor 2: 3

PROGENITOR 1
Expresión: ( - ( + ( * x x ) x ) x )
PROGENITOR 1
Expresión: ( + ( * ( * x x ) x ) x )
```

Algunos comentarios sobre el tester y su salida: La clase *PruebaCruce* sólo se contiene el método de cruce y se usa simplemente para el tester. No se pide como entrega. El cruce se llamará realmente desde el algoritmo de programación genética que se describe en el último apartado. Los mensajes "*Punto de cruce del progenitor ....*" se han incluido en el método de cruce para mostrar qué puntos han salido aleatoriamente. A partir de ellos se han cruzado los

progenitores. Comprueba los progenitores y sus descendientes tras aplicar el cruce en esos puntos. En el ejemplo, los nodos se han etiquetado siguiendo la estrategia primero en profundidad de izquierda a derecha. Si usas otra estrategia los resultados serían distintos. El cruce no limita el tamaño de los descendientes.

Apartado 4: Algoritmo de Programación Genética (2 puntos)

En este apartado, implementarás el algoritmo que simula la Programación Genética. El pseudocódigo de este algoritmo es muy sencillo y se muestra a continuación:

```
Crear población inicial de individuos de forma aleatoria con profundidad fija
Mientras no se alcance el máximo de generaciones o no se encuentre una solución aceptable:
        Si la generación actual no es la inicial entonces crear una nueva población aplicando operadores
        Para cada individuo de la población se calcula el fitness
```

Para la creación de la población inicial utilizaremos los conceptos explicados en el apartado 1. Una población no es más que una colección de individuos, compuestos de nodos.

En todas las generaciones se utiliza una población de individuos. La primera generación emplea una iniciada aleatoriamente, pero las siguientes crean una nueva a partir de la aplicación de operadores genéticos a la población previa. El cruce se aplica solo a parte de la población, en función de una probabilidad que podemos establecer en un 90%. Esto quiere decir que el 10% restante de la población se pasará a la siguiente generación directamente, sin cambiar su forma ya que no se le aplica el cruce. Para elegir qué individuos se seleccionan, dentro del porcentaje de cruce, se pueden emplear diferentes métodos.

En esta práctica, vamos a usar el conocido como “torneo”. En el torneo se seleccionan aleatoriamente K individuos de la población. De estos K individuos se eligen los dos con mejor fitness como progenitores y son éstos los que se enviarían al cruce. Valores típicos de K podrían ser 4, 6 u 8.

Puede resultar útil e interesante reservar el mejor individuo de cada generación y pasarlo a la siguiente, dentro del 10% de individuos que no cruzan. Esta técnica se conoce como *elitismo.*

La interfaz IAlgoritmo muestra algunos métodos que pueden ser útiles en una clase que implemente dicha interfaz.

```
public interface IAlgoritmo {
     public void defineConjuntoTerminales(List<Terminal> terminales);
     public void defineConjuntoFunciones(List<Funcion> funciones) throws ArgsDistintosFuncionesException;
     public void crearPoblacion();
     public List<IIndividuo> cruce(IIndividuo prog1, IIndividuo prog2) throws CruceNuloException;
     public void crearNuevaPoblacion();
     public void ejecutar(IDominio dominio);
}
```

Los métodos *defineConjuntoTerminales* y *defineConjuntoFunciones* simplemente guardan ambos conjuntos una vez creados según el dominio específico. El método *crearPoblación* crea la población aleatoria inicial.

El método *cruce* implementa el cruce entre dos individuos.

El método *crearNuevaPoblacion* crea una nueva población a partir de una población anterior, aplicando el operador de cruce y el paso directo de individuos no cruzados o elitistas.

Finalmente, el método *ejecutar* equivale a un main para el algoritmo de programación genética e implementa el pseudocódigo comentado anteriormente. Cuando se crea un Algoritmo de Programación Genética se deben especificar algunos parámetros, como la profundidad máxima de los individuos en la población inicial, el número de individuos de la población, la probabilidad de cruce, el número máximo de generaciones y el valor de K par los torneos.

La ejecución de los algoritmos de programación genética suele ser costosa en tiempo y recursos. Como información durante la ejecución basta con que muestres el número de generación, el mejor individuo encontrado en esa generación y el fitness de dicho individuo.

Apartado 5: Diagrama de Clases y Explicación del Diseño (0,5 puntos)

a) Debes entregar un diagrama de clases que muestre el diseño efectuado obviando constructores, getters y setters, y debes representar las colecciones, arrays y referencias usadas como asociaciones del tipo más adecuado. No olvides incluir las interfaces que has usado, y las relaciones de implementación entre clases e interfaces. Incluye una pequeña explicación del diseño, así como de las decisiones que has tomado. Las interfaces propuestas aquí son una sugerencia y si utilizas otras debes justificarlo.

b) ¿Es extensible tu diseño? Indica qué pasos habría que dar para resolver un problema diferente.

Apartado 6 (Opcional): Problema de Funciones Booleanas (1 punto)

Otro problema muy común que suele resolverse con Programación Genética son las funciones booleanas de paridad par o impar. Si el número de la paridad es muy alto se ha demostrado que es un problema muy complejo, pero para una paridad de 3 se puede encontrar solución con esta técnica. Básicamente este problema utiliza funciones lógicas como *and*, *or*, *nand* o *nor* y terminales como *d0*, *d1* o *d2* para representar los operadores y los operandos de las expresiones a construir. Por tanto, el conjunto de terminales y de funciones queda definido con estos nuevos símbolos.

El objetivo es similar al del apartado principal, aquí buscamos una expresión lógica que resuelva un problema de paridad 3 par, es decir, una expresión que devuelva cierto si hay un número par de bits puestos a cierto (1) y falso (0) cuando no los haya. El conjunto de datos de prueba sería el siguiente:

```
false   false   false   false
false   false   true    false
false   true    false   false
false   true    true    true
true    false   false   false
true    false   true    true
true    true    false   true
true    true    true    false
```

El fitness sería el número de aciertos en los 8 casos de prueba. Debes crear un conjunto de Terminales y de Funciones que te permita representar este nuevo problema. También debes modificar la lectura de los datos de prueba y el cálculo del fitness, para acomodarlos al nuevo formato del fichero de entrada y al trabajo con funciones booleanas. Si tu diseño permite la flexibilidad necesaria no deberían ser necesarios muchos cambios más.

Normas de Entrega:

Se deberá entregar

- un directorio **src** con todo el código Java de todos los apartados, incluidos los datos de prueba y testers adicionales que hayas desarrollado en los apartados que lo requieren. MUY IMPORTANTE: Estructura correctamente en paquetes el código Java que entregues.
- un directorio **doc** con la documentación generada
- un archivo PDF con el apartado 5.
- Si has hecho el apartado optativo, entrégalo en otro proyecto o directorio.

Se debe entregar un único fichero ZIP con todo lo solicitado, que deberá llamarse de la siguiente manera: GR<numero_grupo>_<nombre_estudiantes>.zip. Por ejemplo Marisa y Pedro, del grupo 2213, entregarían el fichero: GR2213_MarisaPedro.zip.

Referencias

[1]. Koza, J. R. *Genetic Programming*. http://geneticprogramming.com/tutorial/