Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo Fin de Grado

# Framework web de computación evolutiva

*Web framework of evolutionary computing*

Cristian Manuel Abrante Dorta

San Cristóbal de La Laguna, 10 de junio de 2019

D. **Eduardo Manuel Segredo González**, con N.I.F. 78.564.242-Z, profesor asociado adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor,

D. **Coromoto Antonia León Hernández**, con N.I.F. 78.605.216-W, profesora Catedrática de Universidad adscrita al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como cotutora

**C E R T I F I C A N**

Que la presente memoria titulada:

"*Plataforma web de computación evolutiva.*"

ha sido realizada bajo su dirección por D. **Cristian Manuel Abrante Dorta**, con N.I.F. 45.939.508-K.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 10 de junio de 2019

# Agradecimientos

A mis tutores Eduardo y Coromoto, por aconsejarme debidamente y por la pasión que transmiten por su trabajo.

A mis padres y amigos, por ayudarme siempre y estar ahí cuando los necesito.

# Licencia

* Si quiere permitir que se compartan las adaptaciones de tu obra mientras se comparta de la misma manera y NO quieres permitir usos comerciales de tu obra indica:

**Resumen**

*La computación evolutiva es un área muy prometedora en el ámbito de la Computación y la Inteligencia Artificial. Además, el mundo de las aplicaciones web está cobrando una gran relevancia hoy en día debido al surgimiento de numerosas tecnologías que aumentan la capacidad de cómputo de los navegadores, permitiendo ejecutar algoritmos y programas con un mayor coste computacional desde el propio front-end.*

*Teniendo presente estas dos ideas, en este trabajo de fin de grado se ha desarrollado un framework completo de computación evolutiva orientado a la web, utilizando el lenguaje TypeScript. La biblioteca implementada cuenta con las operaciones más comunes referentes a los algoritmos evolutivos y además es extensible para aplicaciones concretas. A su vez, se ha desarollado una aplicación web para resolver el problema de la mochila haciendo uso del framework y así ejemplificar su uso en un problema real.*

**Palabras clave:** Computación evolutiva, framework, desarrollo de software, algoritmos evolutivos.

**Abstract**

Evolutionary computation is a very promising area in computer science and artificial intelligence. Furthermore, the world of web applications is becoming very significant nowadays because of the growth of many different technologies that enhance the computation power of web browsers, allowing them to execute very demanding algorithms and programs from just the front end.

Taking into account this two ideas, in this end of degree thesis I had developed a complete evolutionary computation framework oriented to the web, using TypeScript as programming language. The implemented library has the most common operations of evolutionary algorithms and at the same time it is extensible for specific purposes. Also, I had developed a web application for solving the knapsack problem using this framework for giving a real example for this project.

***Keywords:*** *Evolutionary computing, framework, software development, evolutionary algorithms*

# Índice general

# Índice de figuras

# Índice de tablas

# Capítulo 1

# Introducción

Este capítulo servirá para presentar una clasificación de problemas, exponiendo la complejidad que poseen algunos de ellos, lo que los hace difícilmente abordables mediante técnicas tradicionales, teniendo la necesidad de usar otras técnicas como la computación evolutiva.

## 1.1. Clasificación de problemas

Existen una gran cantidad de problemas que se abordan hoy en día desde el área de las matemáticas, la inteligencia artificial o la ingeniería. Muchos de estos problemas pueden tener diversas aplicaciones prácticas en ámbitos muy variados, y otros muchas veces sirven como formulaciones teóricas cuyo objetivo es encontrar cuales son los límites de la tecnología.

### 1.1.1. Clasificación como *caja negra*

Los sistemas computacionales encargados de resolver estos problemas, pueden entenderse como **cajas negras** (*black boxes*) [37]. Este esquema mental que usamos para describir los sistemas encargados de resolver problemas, parte de la base de que un sistema es una caja que recibe una serie de entradas desde el exterior, y que a partir de un modelo o programa que tiene almacenado, es capaz de procesar dicho conjunto de señales de entrada para devolver una salida.

El nombre **caja negra** viene dado porque normalmente, este modelo que procesa las señales no viene especificado de manera explícita, y por tanto puede tener diversas formas: por ejemplo puede ser una ecuación o conjunto de ecuaciones que procesen una entrada numérica, o también una herramienta estadística que devuelva una estimación a partir de la entrada, o incluso puede ser un modelo lógico que ejecute una serie de sentencias para procesar señales.

En cualquier caso, esta **caja negra** tiene tres partes fundamentales: las entradas, el modelo de procesamiento y la salida. Además, está claro que la parte fundamental es el modelo de computación, que de ser conocido nos permitiría calcular la salida para cualquier entrada al sistema.

Este esquema que hemos definido es muy conveniente para establecer un criterio de clasificación de problemas, en función de qué partes del sistema son conocidas y cuales no. A partir de esto podemos diferenciar en tres tipos de problemas:

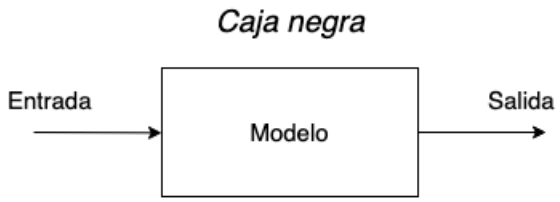


Figura 1.1: Esquema general de un modelo computacional de caja negra

**Optimización**

Los problemas de optimización son aquellos en los que se conoce el modelo, además de la salida que se espera, o al menos una descripción de la misma, y en función de ello, se debe calcular cuáles son los valores de entrada que proporcionan dicha salida.

Existen multitud de problemas que se clasifican en esta categoría, aunque quizás uno de los más clásicos es el problema de viajante de comercio (**TSP**). Este problema consiste en encontrar la secuencia de rutas de coste mínimo dado un grafo de ciudades y sus interconexiones, de tal forma que cada una de las ciudades sea visitada solo una vez.

Como vemos, en este problema está especificada cual es la salida esperada, además del modelo de computación, sin embargo, lo que desconocemos es cual será la entrada, es decir, la combinación de rutas que minimizará el coste y que satisface las restricciones.

**Modelización**

En los problemas de modelización, se conocen las entradas y sus correspondientes salidas, pero se desconoce cual es el modelo de computación que debe usarse para procesarlas.

Este es el tipo de problemas que se abordan en áreas como el **Machine Learning**, pues en ellos se suele tener un conjunto de datos en los que existe una correspondecia entre las entradas y su resultado esperado. El problema está en crear un modelo que sea capaz de *aprender* a partir de dichos datos y por tanto, sea capaz de generalizar y extraer características de datos que no haya procesado previamente.

**Simulación**

Este es el tipo de problemas más lineales, pues en ellos se conoce cuáles serán las entradas y el modelo de computación, pero se desconoce cual será la salida.

Existen multitud de ejemplos de problemas de simulación, como por ejemplo; simulación de fluidos, o simulación meteorológica. Este tipo de problemas tienen una gran utilidad, pues nos permiten predecir una realidad futura, lo cual es crucial en múltiples ámbitos.

### 1.1.2. Clasificación en función de la complejidad (Clases P y NP)

Otra clasificación posible que se puede hacer de los problemas es en función de la *complejidad* que entraña resolverlos. De esta forma una clasificación básica en este sentido pueden ser los problemas *fáciles* y *difíciles* de resolver [38].

Esta definición puede resultar ambigua, puesto que existen múltiples algoritmos que pueden

resolver un problema. Es por ello, que para medir la complejidad de un problema, elegimos el algoritmo que lo resuelve en menor tiempo. Con **tiempo de ejecución** realmente no nos referimos a una magnitud física, sino al número de pasos elementales u operaciones que lleva finalizarlo. Atendiendo a esta clasificación, nos encontramos tres clases de problemas [47]:

- **Clase** $\mathcal{P}$: Los problemas que pertenecen a la clase $\mathcal{P}$, son aquellos en los que existe un algoritmo que pueda resolverlos en tiempo polinomial. Es decir que la función que expresa la complejidad temporal de resolución del algoritmo, es un polinomio.
- **Clase** $\mathcal{NP}$: Para los problemas que están en $\mathcal{NP}$ no existe un algoritmo que pueda resolverlos en tiempo polinomial, sin embargo, se puede evaluar si una solución es válida para el problema en tiempo polinomial. Por ejemplo, el TSP es un problema que pertenece a esta clase.
- **Clase** $\mathcal{NP}-Completo$: Este es un subconjunto de problemas de $\mathcal{NP}$, que se corresponden con los problemas mas difíciles de resolver. Se considera que un problema es $\mathcal{NP}$-Completo cuando está en $\mathcal{NP}$ y además, **todos** los problemas de $\mathcal{NP}$ se pueden transformar a dicho problema mediante una **reducción polinomial**. Encontrar un problema que perteneciera a esta clase fue una tarea compleja hasta el desarrollo del **Teorema de Cook** [38].

Uno de los problemas que más trae de cabeza a la comunidad científica, considerado como una de las grandes incógnitas de la matemática moderna es el problema ¿$\mathcal{P} = \mathcal{NP}$? Este problema trata de discernir si verdaderamente existen problemas que están en $\mathcal{NP}$ [31] porque tienen una naturaleza diferente a los que están en $\mathcal{P}$, o es porque aún no hemos encontrado un algoritmo lo sufiecientemente bueno para resolverlo en un tiempo razonable.

## 1.2. Computación evolutiva

Como podemos ver, existe una gran cantidad de problemas que no poseen un algoritmo exacto que los pueda resolver en un tiempo razonable. Es por ello, que se necesita buscar alternativas que nos permitan llevar a cabo la resolución de dichos problemas, aunque muchas veces se deba hacer de manera aproximada.

La computación evolutiva es una técnica que nos permite encontrar las soluciones a dichos problemas complejos, inspirándose en el proceso evolutivo natural para lograr los resultados. El proceso natural de evolución tiene mucho potencial a la hora de construir buenas soluciones a los problemas, principalmente porque se potencia el mecanismo de "*prueba y error*"[39].

En el proceso evolutivo natural, contamos con un ambiente en el que se encuentran una serie de recursos, por los cuales tienen que competir los individuos que se encuentren en él. Además de tratar de obtener la mayor parte de los recursos disponibles, los individuos tratarán de reproducirse entre sí. De esta forma, en las sucesivas generaciones de individuos, solo los más daptados habrán conseguido sobrevivir y transmitir sus características físicas a su descendencia, a través del material genético.

Mediante un algoritmo evolutivo, se trata de simular este mismo procedimiento. Para ello, consideramos que el ambiente que contiene los recursos por los que los individuos deben competir es el problema que queremos solucionar, y las posibles soluciones del problema son los individuos. Para simular el grado de adaptación al ambiente que deben tener estos individuos se utilizará una función que medirá cual buena es la solución que se está evaluando. Al igual

que ocurre en un ambiente real, estos individuos deberán reproducirse entre sí, transmitiendo su material *genético* a la descendencia.

## 1.3. Algoritmos evolutivos

Siguiendo la idea básica que se quiere perseguir con los algoritmos evolutivos, podemos construir un esquema para determinar cuales serán las distintas fases que tendrán este tipo de algoritmos:

**Algoritmo 1** Esquema básico de un algoritmo evolutivo

```
INICIALIZAR la población con n individuos aleatorios;
EVALUACIÓN de la población mediante la función de puntuación (fitness);
while CONDICIÓN DE PARADA no sea satisfecha do
    SELECCIÓN de padres;
    RECOMBINACIÓN de pares de padres;
    MUTACIÓN de la descendencia;
    EVALUCACIÓN de la descendencia;
    SELECCIÓN de supervivientes para la siguiente generación;
end
```

En este esquema podemos ver las diferentes fases de las que se compone un algoritmo evolutivo. En primer lugar, se deberá inicializar una población con individuos generados aleatoriamente, que se corresponderán con las posibles soluciones que puede tener nuestro problema, a continuación estás soluciones deberán ser evaluadas mediante la función de puntuación (*fitness*), determinando así su grado de adaptación al medio. Seguidamente, se entrará en el bucle principal del algoritmo, el cual continuará ejecutándose mientras que una determinada condición de parada no haya sido satisfecha. Dentro de este bucle, se aplicarán los operadores principales sobre la población generada; en primer lugar, se seleccionarán los padres que se reproducirán en la siguiente generación, a los cuales se les aplicará un operador de cruce para generar una descendencia. Esta descendencia, sufrirá de manera aleatoria una mutación que afectará a su material genético, y después de ello, se evaluará de nuevo respecto a la función de fitness. Seguidamente, se deberán seleccionar que individuos son los que permanecerán en la siguiente generación, repitiendo de nuevo el bucle.

Podemos afirmar que este es un procedimiento estocástico, puesto que en él están involucrados numerosos componentes aleatorios. Esta aleatoriedad es el motor principal de las dos fuerzas que garantizan la eficacia de este tipo de algoritmos [37]:

- **Variación**: Los operadores de variación (recombinación y mutación), generan la suficiente diversidad como para explorar una gran parte del espacio de soluciones.
- **Intensificación**: Los operadores de selección tanto de padres como de descendencia, son los responsables de que se exploren las mejores soluciones garantizando que su material genético perviva en sucesivas generaciones.

Para cada una de las fases de las que se compone un algoritmo evolutivo, existen numerosos métodos que se pueden aplicar. Los cuales dependen enormemente de la codificación que se le esté aplicando a los individuos, además del problema concreto que se esté tratando de resolver.

## 1.4. Bibliotecas de algoritmos evolutivos

Tal y como se ha comentado, los algoritmos evolutivos cuentan con una serie de fases diferentes, donde cada una de estas fases se puede implementar utilizando una gran cantidad de métodos distintos.

Para construir una implementación de este tipo de algoritmos existen muchos procedimientos que se repetirían aunque la naturaleza de los problemas fuera bastante diferente. Debido sobretodo a que existen muchas técnicas que se pueden aplicar de manera independiente a la codificación de los individuos y a cual es el problema concreto que se está pretendiendo resolver.

Esta es la motivación principal para construir un *framework* de computación evolutiva, el hecho de implementar una serie de procedimientos comunes que pudieran ser reutilizados independientemente del problema concreto que se quiera realizar, y que a su vez tuviera ciertas partes extensibles y que se pudieran adaptar a las necesidades concretas del problema.

En este sentido, ya se han desarrollado una gran cantidad de frameworks que comprenden problemas de computación evolutiva y optimización. De esta forma, se puede llevar a cabo una clasificación de los mismos dependiendo del lenguaje de programación en el que se hayan desarrollado. Comenzaremos comentando algunos ejemplo desarrollados en **Java**:

- **Opt4J** [45]: Esta librería contiene muchas técnicas avanzadas en el terreno de la computación evolutiva y las meteheurísticas en general, como por ejemplo, la implementación de algoritmos de optimización por colonia de hormigas o *simulated annealing*.
- **Optimization algorithm toolkit** [32]: Este framework se centra en problemas de optimización, permitiendonos configurar instancias de los problemas clásicos en este ámbito y estableciendo sencillos procedimientos para ejecutar tests estadísticos por parte del usuario.
- **JMetal** [36]: Es uno de los frameworks más famososos y de los pocos cuyo código fuente está alojado en GitHub. Entre sus ventajas se encuentran la gran cantidad de algoritmos de optimización multiobjetivo que tiene implementados: NSGA-II, SPEA2, PAES, etc.

Por otra parte tenemos algunos otros ejemplos implementados en C++:

- **ParadisEO** [33]: El principal objetivo de este framework es llevar a cabo la implementación de problemas de optimización combinatoria, aunque también provee otras herramientas para el análisis del estado de la población, aportando métricas avanzadas.
- **METCO** [43]: Este framework ha sido desarrollado en la Universidad de La Laguna, por el grupo de lenguajes y sistemas informáticos, y ha servido como antecedente principal para el desarrollo de este proyecto.

Estos son algunos ejemplos de librerías de computación evolutiva y optimización en general desarrolladas en lenguajes con una mayor comunidad. Cabe destacar que todos ellos se encuentran en un estado bastante avanzado y tienen implementados una gran cantidad de funcionalidades diferentes.

En este proceso de búsqueda de diferentes librerías de este tipo se ha detectado la carencia de un framework completamente compatible y adaptado para la web. El auge que estan teniendo actualmente las aplicaciones web, aparejado con el aumento de la capacidad de cómputo de los

navegadores, hace que construir herramientas para ser ejecutadas en el lado del cliente sea una tarea que puede tener muchas opciones de futuro.

De esta forma, la motivación principal de este proyecto es construir un framework de computación evolutiva compatible con aplicaciones web. Intentando que sea lo más extenso posible, ofreciendo las operaciones más comunes, y que a la vez pueda ser extensible para ser utilizado como resolutor de problemas concretos.

## 1.5. Objetivos

La carencia de *frameworks* de computación evolutiva adaptados a la web es la motivación principal que se encuentra detrás de este trabajo. Pero para lograr que se lleve a cabo el desarrollo de este proyecto, se deben establecer una serie de objetivos y tareas a realizar.

En primer lugar decidiremos un nombre para el proyecto y el *framework*: **genetics.js**. La elección de este nombre está inspirada en uno de los tipos principales de algoritmos evolutivos; los algoritmos genéticos, además de añadirle el sufijo *.js*, en referencia al lenguaje **JavaScript**, en el cual se llevará a cabo el desarrollo.

De esta forma, estableceremos las características que deseamos que poseea **genetics.js**, cuya implementación serán los objetivos principales del proyecto:

- **Estar desarrollado en un lenguaje moderno que tenga una gran comunidad**: Es importante que el lenguaje en el que se desarrolle el proyecto nos permita ser ejecutado en un entorno web y a ser posible en otros entornos. Por otra parte que tenga una gran comunidad de usuarios, para así poder llevar el proyecto a una mayor cantidad de gente.
- **Poseer una buena documentación**: La documentación es una tarea fundamental, puesto que las diferentes técnicas y métodos disponibles deben estar bien explicados de tal forma que puedan ser comprensibles por los usuarios.
- **Establecer herramientas y procedimientos para garantizar la estabilidad y continuidad del proyecto**: Para garantizar la estabilidad y continuidad del proyecto se pueden utilizar herramientas como los tests, la integración continua o el control de versiones.
- **Implementación de la mayoría de métodos comunes**: Se debe poseer una implementación de los métodos más comunes en las diversas fases de un algoritmo evolutivo.
- **Capacidad de extensión**: Para que el framework pueda ser extensible para problemas concretos, se debe utilizar mecanismos de software como la herencia, o la implementación de interfaces.

En función de estos criterios, se ha establecido un mapa de desarrollo que se corresponderá con las versiones que se publicarán de la librería, atendiendo al versionado semántico (**semantic versioning** [20]):

- `0.1.0`: Implementación de la codificación de soluciones mediante **individuos**.
- `0.2.0`: Implementación de los operadores de mutación.
- `0.3.0`: Impelementación de los operadores de cruce.

- `0.4.0`: Implementación de los operadores de selección de padres.
- `0.5.0`: Implementación de los operadores de selección de supervivientes.
- `0.6.0`: Implementación de las clases gestoras de la población de individuos.

Una vez se ha introducido el proyecto a realizar y se han fijado los objetivos que debe tener el trabajo, en los siguientes capítulos de la memoria se expondrá detalladamente como se ha llevado a cabo el desarrollo del proyecto (Capítulo 2), justificando las tecnologías utilizadas y exponiendo la estructura del software implementado. A su vez, se explicará un caso de uso concreto de la librería desarrollada (Capítulo 3) para finalmente ofrecer unas conclusiones (Capítulo 4) y el valor en el que se ha presupuestado el proyecto (Capítulo 6).

# Capítulo 2

# Desarrollo y tecnologías utilizadas

En este capítulo se describirá en profundidad el framework **genetics.js**. Se expondrán tanto las tecnologías utilizadas, justificando debidamente la elección, así como la propia estructura que tiene el software que se ha desarrollado.

## 2.1. Tecnologías utilizadas

Dado que el objetivo fundamental de este proyecto es el desarrollo de un framework de computación evolutiva que sea completamente compatible con la web, las tecnologías más apropiadas para este desarrollo serán las que se utilicen en el *stack* del lenguaje JavaScript.

Por ello, en primer lugar llevaremos a cabo una introducción a dicho lenguaje de programación, para luego exponer las dependecias externas que se ha utilizado durante la fase de desarrollo y las que se han elegido para ser utilizadas en la versión de la librería en producción, es decir las que serán utilizadas por los usuarios finales.

### 2.1.1. *Stack* de desarrollo en JavaScript


Figura 2.1: Logos de JavaScript, Node y NPM

En primer lugar, es importante introducir el lenguaje de programación JavaScript y la importancia que tiene hoy en día, exponiendo las tecnologías más comunes que tiene aparejadas el desarrollo de una aplicación con este lenguaje.

JavaScript es un lenguaje de programación interpretado, multiparadigma y de tipado débil, desarrollado por Brendan Eich, durante su trabajo en Netscape, para ser utilizado por el navegador que la empresa estaba creando. Durante esa época y en sus primeros años, se consideraba un lenguaje menor, es decir que solo se utilizaba para implementar ciertos aspectos de la interacción del usuario con la página web, o para llevar a cabo operaciones sencillas en el lado del cliente.

Debido a la poca importancia que se le dio a su desarrollo desde el momento inicial, son destacables los grandes errores de diseño con los cuales cuenta [42], y los que hacen que sea bastante complejo confiar en que el software desarrollado en esta tecnología cumplirá ciertos criterios de calidad. Es por ello que diversas empresas e instituciones, han tratado de estandarizar y complementar el lenguaje para garantizar su estabilidad y escalabilidad. Ejemplo de ello, es la organización ECMA con los estándares de JavaScript [7], o Microsoft con la creación del lenguaje TypeScript.

Con los años ha ganado bastante popularidad, gracias en parte a proyectos como NodeJS [14], el cual trata de convertir a JavaScript en un lenguaje con mucho más ámbito del que tenía anteriormente, dando la posibilidad de construir un servidor completo con este lenguaje. NodeJS es una de las tecnologías más punteras para el desarrollo de servidores hoy en día, debido a su gran escalabilidad y a que soporta una gran cantidad de conexiones simultáneas, en parte grecias al uso del motor V8 de JavaScript [13], desarrollado por Google.

En este sentido, es muy destacable también NPM (Node Package Manager) [15] como gestor de dependencias de NodeJS. Este gestor de paquetes es el ejemplo perfecto de sencillez y eficacia, al permitir publicar nuestros propios módulos en un portal que los aglutina de manera centralizada, y que nos permite instalar, gestionar y utilizar dichos paquetes de manera sencilla en nuestra propia aplicación desde la línea de comandos.

La gestión de dependencias es una tarea compleja que puede acarrear ciertos problemas, sobretodo de retrocompatibilidad entre versiones. Una de las grandes ventajas de NPM es que esta tarea es bastante sencilla, centralizando todas las dependecias en un fichero *json* (*package.json*), en el cual se especifica el nombre del paquete y la versión que tenemos instalada. De esta forma se garantiza que se está utilizando en nuestra aplicación exactamente la dependencia que queremos.

Además, el versionado de los paquetes se basa en **semantic versioning** [20], contando con la posibilidad de distinguir entre versiones **minor, major y patch**, garantizando así que se pueda seguir el mapa de desarrollo previsto.

De esta forma, teniendo el potencial de una herramienta como NPM, la posibilidad de llevar esta idea a aplicaciones cliente es bastante interesante, puesto que para la web la importación de módulos externos no se gestiona de una manera tan eficaz que como se hace con NodeJS y NPM. Es ahí donde entran herramientas como Webpack [27] y Babel [5] en juego, las cuales permiten que el código que se importa mediante NPM y se utiliza en ficheros de código fuente, sea compilado para ser utilizado directamente en el *front-end*. De esta forma podemos utilizar NPM como gestor de dependencias aunque estemos trabajando en el lado del cliente.

Como vemos, la existencia de este tipo de tecnologías hace que sea muy conveniente desarrollar la librería **genetics.js** como un módulo NPM, puesto que ya no solo podría ser utillizada en el lado del cliente, sino que también hace posible que se utilice en otros ámbitos como un servidor con NodeJS o cualquier otra tecnología basada en JavaScript.

### 2.1.2. Tecnologías utilizadas para el desarrollo

Tal y como se ha comentado, la librería **genetics.js** se desarrollará como un módulo NPM para garantizar que sea compatible con tecnologías web. En este primer apartado, expondremos cuales seran las dependencias que este módulo tendrá en el desarrollo. Estas dependencias

realmente no afectarán al usuario final, puesto que no serán descargadas ni utilizadas cuando se instale el paquete, ya que solo son útiles para garantizar y facilitar el desarrollo correcto de la librería.

Las tecnologías que se han utilizado como dependencias de desarrollo han sido las siguientes:

**Control de versiones (Git y GitHub)**

Figura 2.2: Logos de Git y GitHub

Los sistemas de control de versiones sirven para que se pueda llevar un desarrollo organizado del proyecto. Tener un sistema de control de versiones es esencial porque en el repositorio que se cree estará alojado todo el código fuente que se escriba, además del historial de *commits* o confirmaciones que se lleven a cabo, de tal forma que se pueda regresar con facilidad a casi cualquier punto del desarrollo.

En este proyecto, he utilizado Git [9] como sistema de control de versiones y GitHub [19] para alojar el repositorio remoto. Además el desarrollo se ha estructurado por ramas, de tal forma que solo se encuentre en *master* la versión estable del proyecto.

Para estructurar las ramas he utilizado un convenio de nombres, de tal forma que el desarrollo del contenido de la versión concreta sea el nombre de la rama. Por ejemplo si se está desarrollando la versión 0.1.0, la rama de desarrollo llevará el nombre *v0.1.0-dev*

**TypeScript**

Figura 2.3: Logo de TypeScript

Mas que una dependencia de desarollo, TypeScript es el lenguaje de programación en el que se ha implementado el proyecto. Realmente, se considera una dependencia externa, pues aunque el desarrollo completo se llevado a cabo con este lenguaje, se necesita un compilador a JavaScript para publicar el paquete en NPM, con lo cual el usuario final tan solo utilizará código fuente JavaScript, además de las definiciones de tipos generadas.

La utilización de TypeScript como lenguaje de desarrollo viene motivada por las carencias que presenta JavaScript para realizar una librería escalable y con las garantías de calidad que se requieren para que sea fácilmente extensible.

TypeScript fue desarrollado por Microsoft y actualmente, su compilador principal es un proyecto de software libre mantenido por la propia empresa bajo la licencia Apache 2.0 [18]. Se trata de un superconjunto de JavaScript, de tal forma que cualquier fichero en JavaScript, también está en TypeScript. Esto tiene como ventaja principal la facilidad de migración de un proyecto en un lenguaje a otro.

Las ventaja principal que presenta TypeScript frente a JavaScript es que se trata de un lenguaje con tipos. Las definiciones de tipos hacen que sea más complejo programar con este lenguaje, pero a la vez garantizan mucha más seguridad en el código que se desarrolla, asegurando así que se pueda escalar de una manera mucho más efectiva.

Es destacable también, las características que presenta TypeScript que son comunes en los lenguajes orientados a objetos tradicionales; como clases o interfaces, que de manera nativa no son soportadas por JavaScript. Estas características son las idóneas para estructurar el software evitando repeticiones necesarias e imponiendo las estructura que se seguirá en un futuro.

Cabe destacar que las opciones del compilador son fácilmente configurables mediante un fichero *json* (*tsconfig.json*). En él se puede especificar multitud de opciones, sin embargo la más relevante es el estándard de JavaScript al que se quiere compilar el proyecto. En el caso de **genetics.js**, se ha elegido la versión ECMAScript5 puesto que es la que tiene un soporte más amplio por la mayoría de navegadores.

**Tests (Jest y ts-jest)**

Figura 2.4: Logo de Jest

Una parte muy importante del desarrollo del software son los tests. Mas aun cuando se trata de una librería en la que van a existir una gran cantidad de clases y estructuras de datos diferentes, en cuyo caso, es muy importante que se desarrolle una batería de tests que garantice su correcto funcionamiento.

La tecnología utilizada para este propósito ha sido Jest [11], pues nos ofrece múltiples ventajas a la hora de crear test unitarios y parametrizados a medida del componente que estemos comprobando y también porque nos permite realizar de manera sencilla *mockups*, o simulaciones, de resultados de las funciones que queramos testear.

El hecho de realizar *mockups* es de gran ventaja en una aplicación como esta, en la que tanta lógica depende de resultados aleatorios. Con esta librería de testing se puede especificar *ad-hoc* cual es el resultado “ “*aleatorio*”que nos devolverá el generador sin tocar su código fuente y de esta forma, aumentar la robostez de los test, comprobando la lógica del componente externalizándola de los valores aleatorios que le puedan llegar como entrada.

Una desventaja de este framework de desarrollo, es que no es completamente compatible con TypeScript en su versión actual, lo cual genera algunos problemas de a la hora de comprobar

los tipos de los módulos que se están testeando. A su vez, esto genera algunas incompatibilidades de tipos al generar los *mockups*, probablemente porque estos internamente se basen en el tipado débil que presenta el lenguaje JavaScript. Para solucionar estos problemas se recurre a la librería externa **ts-jest** [22].

Al igual que todas las herramientas utilizadas, Jest tiene también su fichero de configuración (*jestconfig.json*). En el se ha especificado cual será el directorio en el que se almacenarán los test además de indicar que se utilizará **ts-jest** para procesar aquellos que estén en TypeScript.

#### Integración continua (CircleCI)


Figura 2.5: Logo de CircleCI

El paso siguiente después de haber realizado una batería de tests es garantizar que estos se ejecuten de manera continua. Esto quiere decir que sean ejecutados cuando se haga un nuevo *commit* al respositorio remoto.

La batería de tests es ejecutada por un servidor externo, lo que también sirve para separar dichos test de nuestra máquina local de desarrollo y comprobar de esta forma si nuestro sistema operativo o alguna configuración local está afectando al funcionamiento del proyecto.

Llevar a cabo un procedimiento de integración continua es muy importante para garantizar que si se hace un cambio en una parte concreta de la librería, esto no tiene efectos colaterales en otros módulos. La importancia de esto radica principalmente en la seguridad que tenemos al añadir nuevas funcionalidades a lo que ya tenemos hecho, de tal forma que incorporar a nuevos colaboradores y aceptar sus cambios no sería tan arriesgado como si tan solo se pasaran los test en local.

Para llevar a cabo la integración continua se ha elegido la tecnología **CircleCI** [10], aunque hay algunas otras opciones que presentan las mismas características, como por ejemplo, **TravisCI** [21]. El fichero de configuración de esta herramienta (*.circleci/config.yml*), es básicamente calcado del estándard utilizado para aplicaciones con NodeJS. En él, tan solo es necesario especificar el comando npm que se utilizará para ejecutar los tests de integración continua.

#### Cubrimiento de código (Coveralls)


Figura 2.6: Logo de CircleCI

Otro de los aspectos fundamentales relacionados con los tests es el cubrimiento de código, el *coverage*. El cubrimiento de código es un informe que se genera para especificar el porcentaje de líneas de código que hemos cubierto con nuestros tests.

Esta no es una medida infalible del correcto funcionamiento del software desarrollado, sin embargo si que nos da una medida bastante fiable de lo extensos que hemos sido con nuestros tests, mostrando si hemos conseguido cubrir la mayoría de líneas de código existentes en el proyecto.

La herramienta encargada de recoger los informes de coverage a partir de los tests utilizados para la integración continua ha sido **Coveralls** [6]. Para ello, se ha especificado un comando de test ejecutado por la integración continua que garantiza que en el caso de que dichos tests se ejecuten sin errores, se envíe un informe al servidor de coveralls.

La configuración de este servicio es bastante simple, pues tan solo habría que especificar en el fichero de configuración (*coveralls.yml*) cual es la herramienta de integración continua que se ha utilizado, en este caso CircleCI.

### Documentación (TypeDoc)

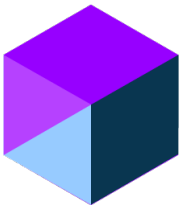

Figura 2.7: Logo de TypeDoc

La documentación es una parte esencial del desarrollo, puesto que facilita la tarea de que los usuarios puedan utilizar los diferentes módulos de la librería, teniendo información de cual será el resultado y los parámetros que se espera cada una de las funciones y cual es el objetivo de las clases desarrolladas.

Para elaborar una documentación para nuestro proyecto se ha utilizado **TypeDoc** [24]. Esta herramienta nos permite hacer comentarios en las clases, métododos, interfaces y variables para luego generar una documentación en formato HTML que puede ser desplegada en GitHub pages.

La sintaxis de los comentarios es bastante común, puesto que es muy similar a la que utilizan otras herramientas similares como Doxigen o Javadoc, para lenguajes como C++ y Java. Además, se puede formatear los comentarios, pues también admite sintaxis Markdown.

### Formateo de código e IDE (TSLint, Prettier y WebStorm)


Figura 2.8: Logos de TSLint, Prettier y WebStorm

Para finalizar, expondré las herramientas utilizadas para formatear el código que se programa, de tal forma que se garantice que sea cual sea el editor y el desarrollador que se incorpore

al proyecto, el formato y estándard que se seguirá en el código fuente va a ser el mismo.

La herramienta **Prettier** [16] se encarga de hacer este formateo del código. Se configura mediante un fichero (*.prettierrc*), en el que se indican las opciones que se desean cumplir para el código fuente generado.

De manera complementaria, la herramienta **TSLint** [23] señala en el editor de código errores tanto sintácticos como de estilo. Se puede por tanto indicar cual será el estándard que queremos que siga nuestro proyecto de software. Existen muchos, aunque uno de los más restrictivos es el diseñado por Airbnb [3].

La manera más cómoda para utilizar estas dos herramientas es de manera conjunta con el IDE o editor de código, de tal forma que cuando se almacenen los cambios en un fichero este se formatee de manera automática. En este proyecto se ha utilizado **WebStorm** [28] como IDE de desarrollo, pero hay otras buenas opciones para este propósito, como Visual Studio Code [26] o Atom [4].

### 2.1.3. Tecnologías utilizadas en producción

Las dependencias o tecnologías utilizadas en producción son aquellas que serán usadas por el usuario final cuando instale la librería, y no solo serán útiles durante la fase de desarrollo.

En este sentido, se ha tratado de construir una librería *minimalista* en cuanto a dependencias, de tal forma que el paquete incluya las menos posibles en su versión de producción. En una librería de las características de este proyecto, realmente esta tarea es bastante sencilla de llevar a cabo, pues prácticamente todo el trabajo de desarrollo se puede hacer desde cero y sin dependencias externas.

De esta forma, tan solo se ha utilizado una dependencia en producción:

#### random.js

Una de las problemáticas de trabajar con un lenguaje como JavaScript es que no existe un módulo de generación de números aletorios realmente efectivo, al estilo de la librería **random** de C++ [1]. Además, también se añade la problemática de que al estar trabajando con TypeScript, sería muy conveniente llevar a cabo la comprobación de tipos cuando utilicemos dicho paquete. Debido a estos problemas, la única librería que satisfacía las condiciones de ser un generador de números aleatorios correcto estadísticamente y que trabajara con TypeScript es **random.js** [2].

La ventaja principal que tiene trabajar con esta librería es que se puede especificar el *engine* y la semilla que se utilizará para generar el número aleatorio. Y cuenta ya con una serie de *engines* predeterminados para generar números aleatorios aprovechando característica concretas del entorno en el que se está ejecutando, como por ejemplo NodeJS o un navegador.

Aparte de las ventajas que tiene a la hora de especificar el origen del número aleatorio, es destacable también la facilidad que posee para generar datos de un cierto tipo, como por ejemplo: enteros y flotantes en un rango o valores de verdadero y falso (*boolean*).

Un ejemplo de su funcionamiento para generar un booleano aleatorio es el siguiente:

```
1 import { bool, MersenneTwister19937 } from 'random-js';
2
3 /**
4 * Genera un bool con un 0.3 de probabilidad
5 * de ser true.
6 */
7 bool(0.3)(MersenneTwister19937.autoSeed());
```

## 2.2. Estructura del software

En esta sección se describirá cual es la estructura lógica de **genetics.js**, exponiendo cada una de sus partes y justificando cuáles han sido las decisiones de diseño que se han tomado para su desarrollo.

Para construir esta librería, la idea fundamental que se ha seguido es intentar independizar lo máximo posible cada una de las fases y componentes con las que cuenta un algoritmo evolutivo.

De esta forma, para exponer el trabajo realizado, se expondrán por separado cada uno de los módulos en los que se divide esta librería, ofreciendo un diagrama que expone las funcionalidades de cada parte de manera lógica y señalando los métodos más relevantes que explican su funcionamiento de manera externa.

### 2.2.1. Individuos

Dentro de los algoritmos evolutivos, los individuos son quizás la parte fundamental. La idea principal de su diseño para esta librería es la de simplificar lo máximo posible las tareas de evaluación y modificación de su contenido.

En computación evolutiva, los individuos codifican una solución concreta de un problema. Este individuo contendrá una serie de datos, que se conocen como el **genotipo** o codificación de dicha solución. A partir de este genotipo, se puede extraer la información que verdaderamente sería la solución a nuestro problema y que se conoce como **fenotipo**.

Para llevar a cabo la conversión entre genotipo y fenotipo, se debe disponer de la información concreta del problema, y por tanto tener una función que converta el genotipo de un individuo a una solución dentro del espacio de soluciones.

Un ejemplo de esto podría ser un problema en el que se quiere buscar el máximo de una función en un dominio de números enteros. Una posible codificación puede ser establecer una correspondencia entre cada uno de los enteros presentes en el dominio y un individuo formado por una cadena de números binarios. Cuando se desee evaular un individuo, lo que se debe hacer es decodificar el número entero al que se corresponde la cadena binaria que contiene dicho individuo.

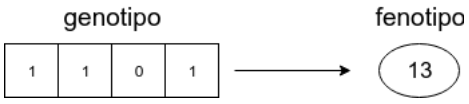


Figura 2.9: Ejemplo de conversión entre genotipo y fenotipo

Elegir una buena codificación para nuestro problema es una tarea muy importante y en ocasiones compleja, puesto que la calidad del algoritmo que se diseñe en gran medida dependerá de esta decisión. Además, es importante destacar que dependiendo de la codificación que hayamos elegido, se le podrán aplicar a los individuos involucrados en el algoritmo unas operaciones u otras.

Tal y como se ha comentado, la representación de los individuos es casi infinita, pues depende mucho del dominio del problema que se esté tratando de resolver. En **genetics.js** la decisión que se ha tomado es implementar una jerarquía de clases para modelar esta realidad.

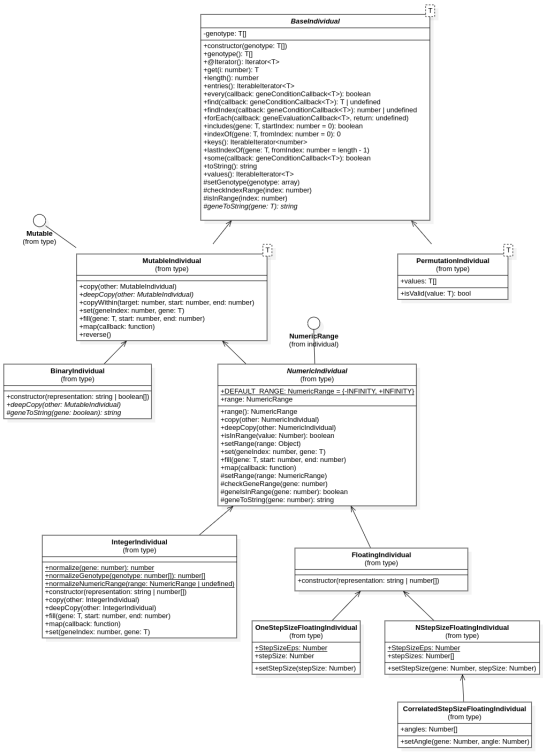


Figura 2.10: Diagrama de clases de los individuos

En este diagrama UML se puede ver la estructura de clases elegida, de las cuales será necesario explicar cada una de sus partes.

### BaseIndividual

`BaseIndividual` es una clase abstracta que representa al individuo base de la jerarquía. En ella se almacena el `array` que contiene el genotipo del individuo. Cabe destacar que se trata de una clase genérica en la que no se especifica el tipo que dato que contendrá dicho array. De esta forma se garantiza que se puede extender en clases hijas con la capacidad de reutilizar los métodos que esta tiene implementados.

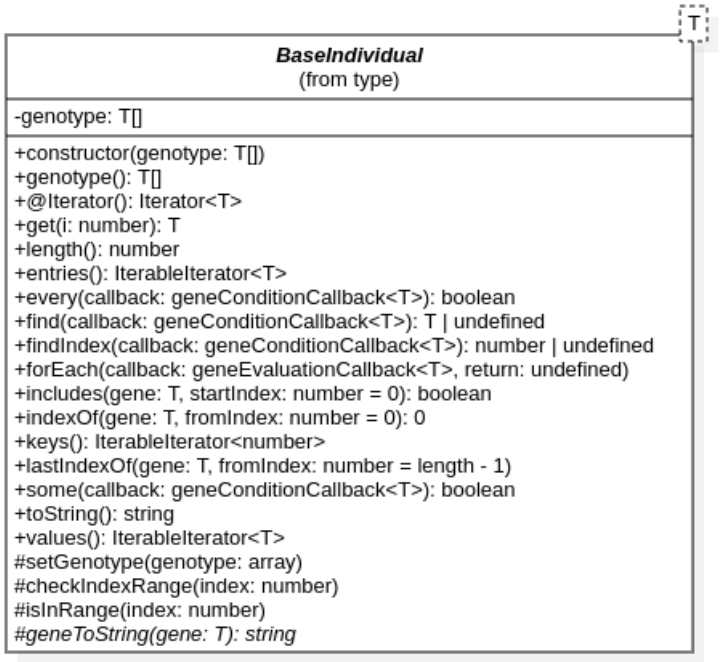


Figura 2.11: Diagrama de clase de `BaseIndividual`

Los métodos con los que cuenta esta clase son aquellos que implican una búsqueda en un array y que ya están implementados en JavaScript. Esto se hace para asegurar que los métodos presentes en esta clase son inmutables, es decir, que no cambian el contenido del genotipo.

Como método abstracto tan solo tiene `toString`, que sirve para convertir a un individuo en su representación como `string`. Este sería el único método a implementar si se quiere extender esta clase.

### MutableIndividual e interfaz Mutable

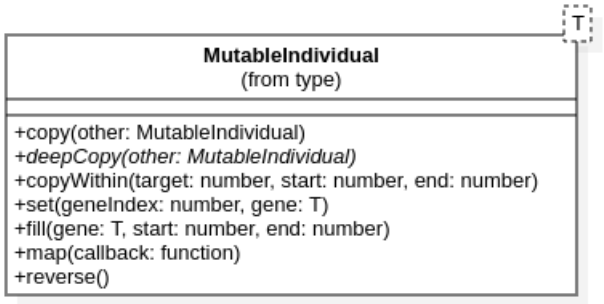


Figura 2.12: Diagrama de clase de `MutableIndividual`

`MutableIndividual` es una clase también abstracta que hereda de `BaseIndividual`, y que implementa la interfaz `Mutable`. Esta interfaz introduce los métodos más comunes para hacer cambios en el contenido del genotipo del array; como por ejemplo el método `set`, `fill` o `map`, los cuales ya están presentes en los arrays de JavaScript.

El único método abstracto que presenta esta clase es `deepCopy`, cuya utilidad es especificar como se hace una copia profunda del genotipo.

### BinaryIndividual

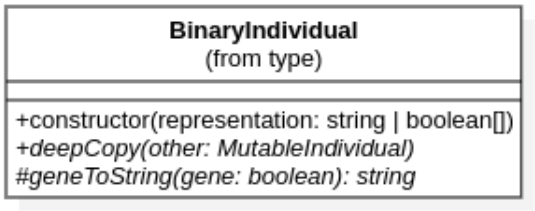


Figura 2.13: Diagrama de clase de `BinaryIndividual`

El individuo binario (`BinaryIndividual`), es uno de los más utilizados dentro de los algoritmos evolutivos. Además, fue la primera representación utilizada para constrir un algoritmo genético [40].

Para implementar este tipo de individuos, se ha extendido de la clase `MutableIndividual`, pero con el tipo de dato `boolean`, que es el que contendrá el genotipo.

Un ejemplo de su uso sería el siguiente:

```
1 import { BinaryIndividual } from "genetics-js"
2
3 /**
4 * Se genera un individuo cuyo genotipo es:
5 * [true, false, false, false, true]
6 */
7 const ind = new BinaryIndividual("10001");
```

### NumericIndividual

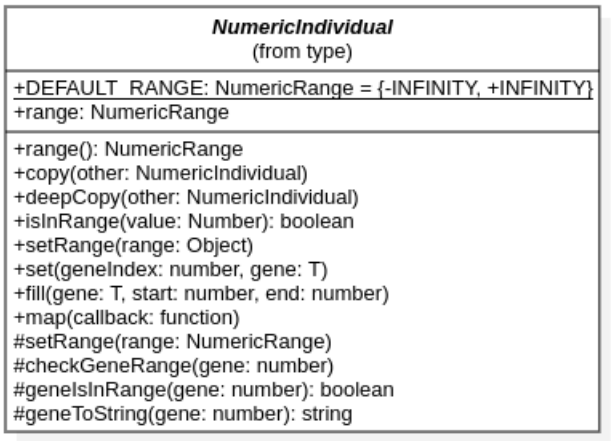


Figura 2.14: Diagrama de clase de `NumericIndividual`

El individuo numérico es una clase abstracta para aglutinar a los individuos que contienen números enteros y flotantes, los cuales tienen la mayoría de métodos comunes a excepción de

su representación.

Los individuos con codificación numérica contienen una parámetro `range` que especifica el rango en el que se encuentran los genes de dicho individuo. Para esta librería, los individuos solo podrán tener un rango, que se aplicará a todo los genes o valores del individuo, aunque puede caber la posibilidad de que se tengan varios rangos que se apliquen a los intervalos de genes especificados.

Para implementar esta clase, es necesario que se sobrescriban la mayoría de métodos presentes en `MutableIndividual` para hacer la gestión de errores, garantizando que los valores que se introducen para hacer cambios están dentro del rango permitido.

### IntegerIndividual

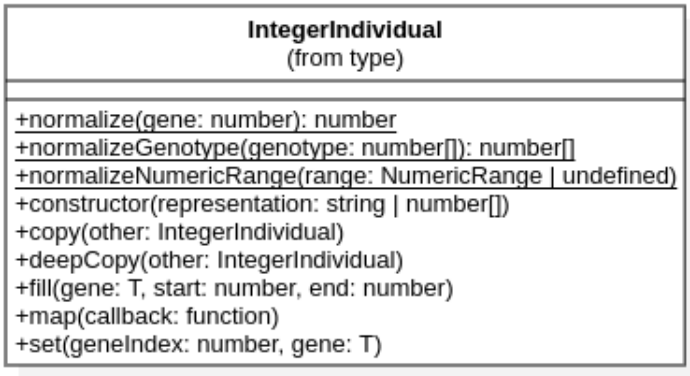


Figura 2.15: Diagrama de clase de `IntegerIndividual`

El individuo entero es un `NumericIndividual` que contiene en su genotipo tan solo números enteros, por ello, lo más importante es garantizar que cualquier número que se introduzca en el array sea un número entero.

Este criterio se puede garantizar de una manera más restrictiva si se genera un error cada vez que se intente establecer un número flotante como un valor dentro del genotipo, pero en esta implementación se ha optado por redondear siempre los valores numéricos que se intenten introducir, de tal forma se garantiza que estos siempre sean enteros.

Un ejemplo de su uso sería el siguiente:

```
1  import { IntegerIndividual } from "genetics-js"
2
3  /**
4  * Se genera un individuo cuyo genotipo es:
5  * [1, 2, -3, 4]
6  */
7  const ind1 = new IntegerIndividual("1 2 -3 4");
8  /**
9  * Se genera un individuo cuyo genotipo es:
10 * [1, 2, 3, 4]
11 */
12 const ind2 = new IntegerIndividual("1 2.3 3 4");
```

### FloatingIndividual

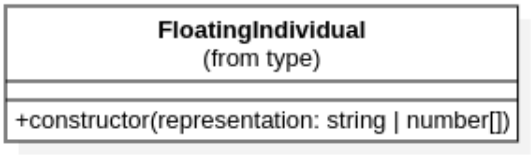


Figura 2.16: Diagrama de clase de `FloatingIndividual`

Por otra parte, tenemos el individuo numérico que contiene valores en punto flotante. Esta clase es prácticamente idéntica a la clase base de individuos numéricos (`NumericIndividual`), puesto que tan solo se debe implementar el constructor.

Un ejemplo de su uso sería el siguiente:

```
1 import { FloatingIndividual } from "genetics-js"
2
3 /**
4 * Se genera un individuo cuyo genotipo es:
5 * [0.5, -3.0, 0.04]
6 */
7 const ind1 = new FloatingIndividual("0.5 -3 4e-2");
```

## 2.2.2. Generador de individuos

Dentro del ciclo de un algoritmo evolutivo, la generación de individuos aleatorios es la fase inicial. En esta sección se expondrá como se ha implementado el generador de individuos aleatorios y las partes que este ha involucrado.

El generador de individuos aleatorios debe hacerse a medida para el individuo que se pretende generar, garantizando que sea extensible para cualquier tipo de individuo a implementar en un futuro. Para **genetics.js** se ha creado un generador aleatorio por cada tipo de individuo que se ha desarrollado, siempre tratando de reutilizar la mayor parte de código estableciendo una jerarquía de herencia.

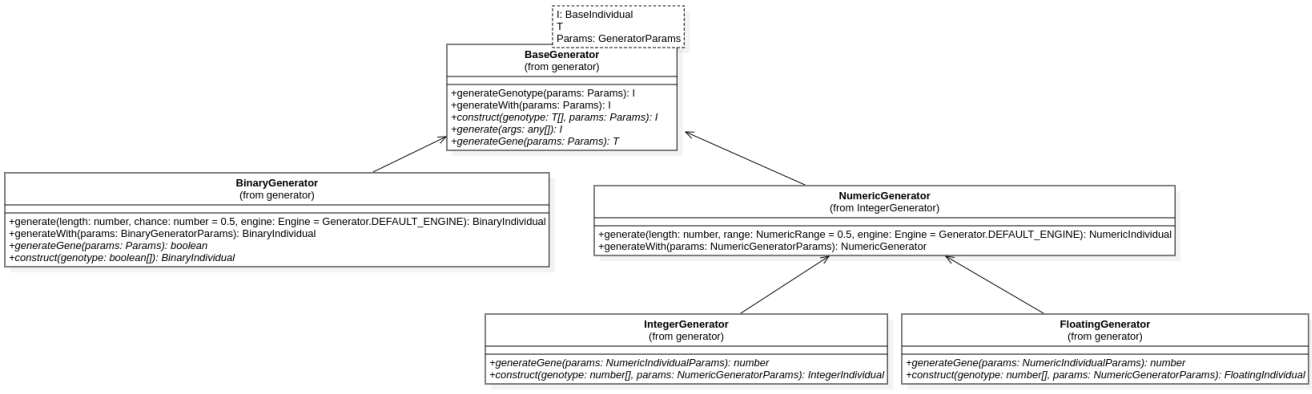


Figura 2.17: Diagrama de la jergarquía de clases del generador de individuos

### BaseGenerator e interfaz IndividualGenerator

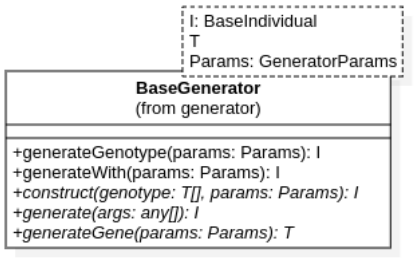


Figura 2.18: Diagrama de clase de `BaseGenerator`

`IndividualGenerator` es la interfaz básica que se debe implementar para crear un generador aleatorio de individuos. Esta interfaz es genérica, es decir, que debemos especificar cual será el tipo de individuos que pretendemos generar y además deberemos especificar los parámetros concretos del generador a implementar.

Para establecer los parámetros del generador hemos creado otra interfaz (`GeneratorParams`), la cual de manera básica tiene dos propiedades:

- `engine`: Es el engine que se utilizara como semilla del generador aleatorio.
- `length`: Es el tamaño del individuo que se va a generar.

Esta interfaz `GeneratorParams` deberá ser también extendida para indicar los parámetros concretos del generador.

Para ofrecer una implementación básica de un generador de individuos se ha creado la clase `BaseGenerator`. Esta clase tiene la mayoría de métodos de la interfaz `IndividualGenerator` implementados, estableciendo además una serie de métodos abstractos para que las clases heredadas puedan implementarlos y así simplificar la tarea de desarrollo.

Estos métodos son los siguientes:

- `construct(genotype: T[], params: Params): I`: Este método sirve para construir un individuo dado un genotipo generado, es útil porque en la clase base no se puede acceder al constructor de los individuos concretos que se generarán.
- `generate(...args: any[]): I`: Este es el método básico que podrá ser llamado desde el exterior para generar un individuo. No tiene definidos los parámetros puesto que estos se deberán especificar para cada uno de los generadores que se implementen concretamente.
- `generateGene(params: Params): T`: Este método servirá para generar un gen del genotipo. Espera recibir una lista de parámetros.

### BinaryGenerator y BinaryGeneratorParams

El generador de individuos binarios (`BinaryGenerator`) tiene bastante importancia en este framework, debido a la gran cantidad de aplicaciones existentes que utilicen individuos binarios como su codificación para las soluciones.

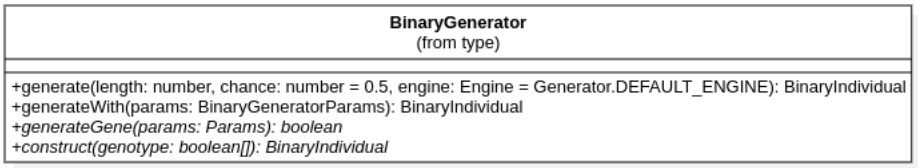


Figura 2.19: Diagrama de clase de `BinaryGenerator`

Los parámetros de este tipo de generador (`BinaryGeneratorParams`), son exactamente iguales a los del generador base, pero añadiendo a su vez un campo `chance` para establecer la probabilidad de que se genere un valor `true` en el individuo, y de esta manera sesgar el generador.

Un ejemplo de su uso sería el siguiente:

```
1  import { BinaryGenerator } from "genetics-js"
2
3  const generator = new BinaryGenerator();
4
5  /**
6  * Un posible inidivudo generado podr a tener el
7  * genotipo:
8  * [false, false, true, false]
9  */
10 console.log(generator.generateWith({ length: 4, chance: 0.3 }}))
```

### NumericGenerator y NumericGeneratorParams

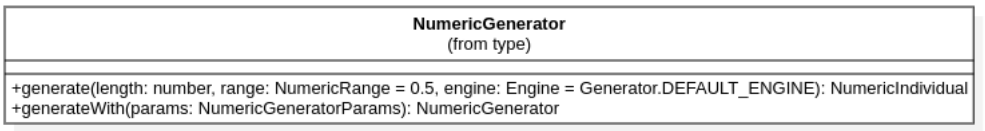


Figura 2.20: Diagrama de clase de `NumericGenerator`

El generador de individuos numéricos (`NumericGenerator`) es una clase abstracta que sirve para aglutinar los generadores de individuos numéricos, ya sean de números enteros o reales.

La utilidad principal de este generador es especificar los parámetros que tendrá un generador numérico (`NumericGeneratorParams`), el cual extiende los parámetros básicos, pero a la vez añade el campo `range` para especificar el rango que tendrán los genes de los individuos numéricos generados.

### IntegerGenerator y FloatingGenerator

Una vez se ha desarrollado `NumericGenerator` como generador numérico fundamental, es bastante sencillo extender dicha clase para crear un generador de individuos de números enteros (`IntegerGenerator`) y de números en punto flotante (`FloatingGenerator`).

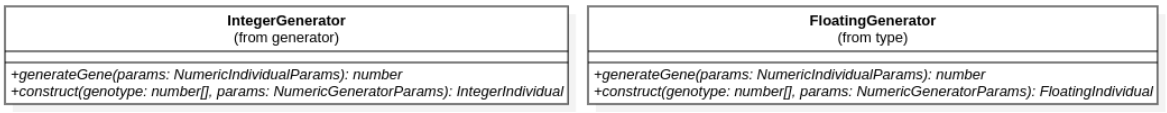


Figura 2.21: Diagrama de clase de `FloatingGenerator` e `IntegerGenerator`

Para ello, tan solo se debe implementar el método abstracto `generateGene`, creando un individuo numérico o un individuo en punto flotante respectivamente. Cabe destacar que los parámetros no varían respecto a aquellos establecidos en `NumericGeneratorParams`, pues en ambos casos tan solo es necesario especificar el rango como elemento generador.

Un ejemplo de su uso es el siguiente:

```
1  import { IntegerGenerator, FloatingGenerator } from "genetics-js"
2
3  const integerGenerator = new IntegerGenerator();
4  const floatGenerator = new FloatingGenerator();
5
6  /**
7  * Un posible inidivudo generado:
8  * [3, 3, 1, 5]
9  */
10 console.log(integerGenerator.generateWith({ length: 4, range: [1, 5] }}))
11
12 /**
13 * Un posible indivudo generado:
14 * [0.1, -0.4332, 1.0, 0.5677]
15 */
16 console.log(floatGenerator.generateWith({ length: 4, range: [-0.5, 2] }}))
```

### 2.2.3. Gestión de la población

La gestión de la población de individuos es un elemento fundamental para la ejecución de algoritmos evolutivos. Para este propósito, se ha creado una clase `Population`, que es la encargada de contener el conjunto de individuos de la población, además de proveer de métodos para acceder, modificar o evaluar a dichos individuos.

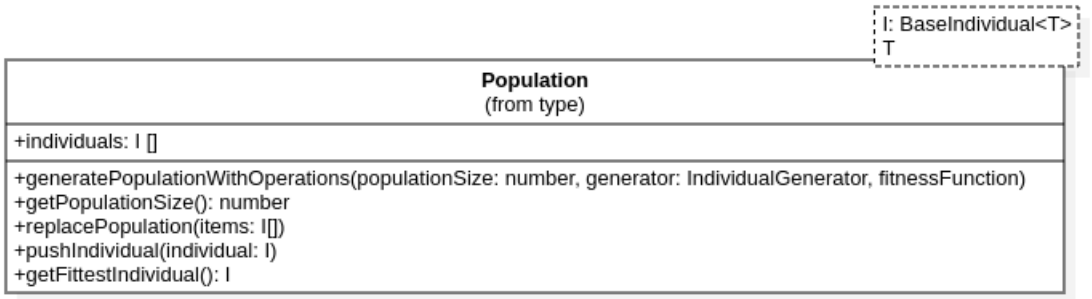


Figura 2.22: Diagrama de clase de `Population`

Los métodos que contiene esta clase son:

- `generatePopulationWithOperations`: Este método recibe un generador de individuos con sus parámetros, además de una función de fitness y crea a la población de individuos, evaluando a la misma vez su fitness.
- `getPopulationSize`: Devuelve el tamaño de la población.
- `replacePopulation`: remplaza los items de la población actual por otros.
- `pushIndividual`: Añade un individuo a la población.
- `getFittestIndividual`: devuelve el individuo que tenga una mejor evaluación según la función de fitness. Si el problema fuera de maximización se devolvería el individuo con la evaluación más alta, y si fuera de minimización sería aquel con la evaluación más baja.
- `getPopulationItems`: devuelve los items de la población en forma de `array`.

Ádemás, cabe destacar que esta clase contendrá una serie de estadísticas comunes que se pueden utilizar para visualizar cuales son las métricas que está siguiendo la población en las sucesivas generaciones del algoritmo. Las estadísticas que se pueden consultar serían las siguientes:

- `averageAge`: la edad media de los individuos de la población.
- `averageFitness`: el fitness medio de los individuos de la población.
- `fitnessSum`: la suma total de fitness de los individuos.
- `fittestIndividualIndex`: el índice del individuo una mejor evaluación según la función de fitness y el tipo de problema.

### 2.2.4. Selección de padres

La selección de padres es la fase del algoritmo que se corresponde con la elección de aquellos individuos que se reproducirán, generando una descendencia en la siguiente generación. El criterio de selección de estos individuos puede ser muy variado, pero normalmente es proporcional a cuanto de adaptados al medio estén, lo que se conoce como **fitness proportional selection**.

Para esta librería se ha desarrollado una selección proporcional al fitness cuya implementación se puede hacer mediante dos métodos diferentes.

#### RouletteWheel

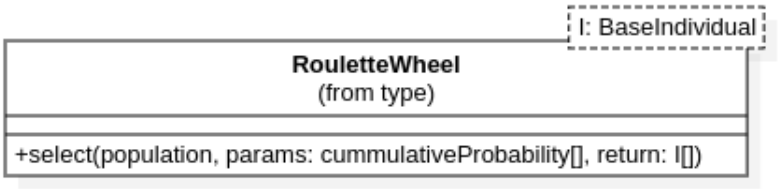


Figura 2.23: Diagrama de clase de `RouletteWheel`

La idea que se encuentra tras la selección mediante *Roulette Wheel* [44], es seleccionar a los individuos mejor adaptados, pero a su vez dejando una posibilidad a los individuos con una

evaluación de fitness peor la posibilidad de ser padres en la siguiente generación.

De manera metáforica, se utiliza una ruleta que tiene todos los individuos representados, pero con un área proporcional a su fitness. Al girar esa ruleta, es más probable que se elijan a los individuos mejor adaptados, puesto que su área de selección es mayor, sin embargo, tampoco se impide la selección de individuos mucho meños adaptados, aunque su área de seleccion sea menor. Esto nos permite garantizar la variedad en la población seleccionada para evitar que las búsquedas puedan quedar atrapadas en un óptimo local.

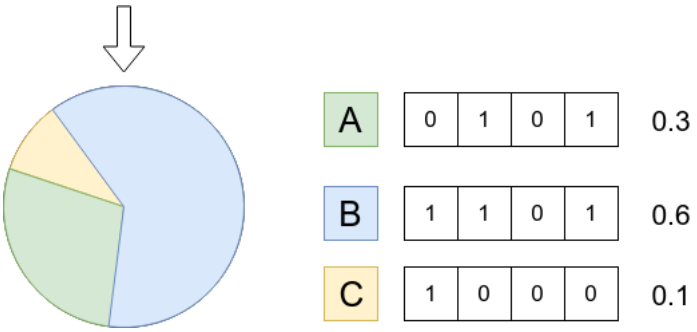


Figura 2.24: Diagrama que representa la ruleta proporcional al fitness

En la implementación de **genetics.js**, se espera como parámetros una población, además de un *array* con la probabilidad de selección acumulada normalizada. Tal y como comentamos, esta probabilidad de selección dependerá de la evaluación del fitness del individuo.

### StochasticUniversalSampling

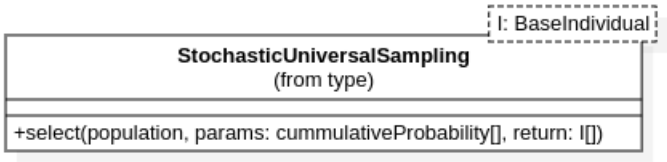


Figura 2.25: Diagrama de clase de `StochasticUniversalSampling`

La otra posibilidad de implemnentación se conoce como *Stochastic Universal Sampling* [30]. La idea es similar a *Roulette Wheel*, pues utiliza también la ruleta proporcional como idea fundamental. Sin embargo, en este método la ruleta no se hace girar tantas veces como individuos se quiera seleccionar, sino que se hace una sola tirada en la que se seleccionan todos los individuos a la vez. Sería el equivalente a una ruleta en la que existieran tantos puntos de selección como individuos, y en una sola tirada se decidieran todos.

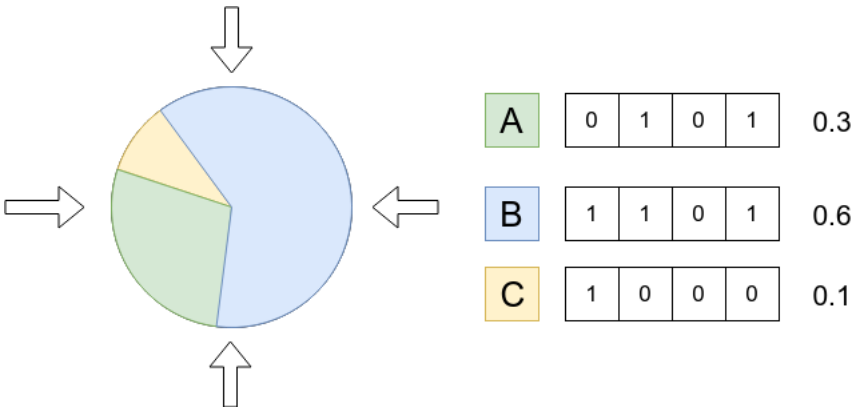


Figura 2.26: Representación de la ruleta proporcioonal al fitness con múltiples puntos de selección

Los parámetros de este tipo de selección son iguales que en *RouletteWheel*, por tanto es necesario especificar un `array` con las probabilidades normalizadas acumuladas.

### 2.2.5. Operaciones de cruce

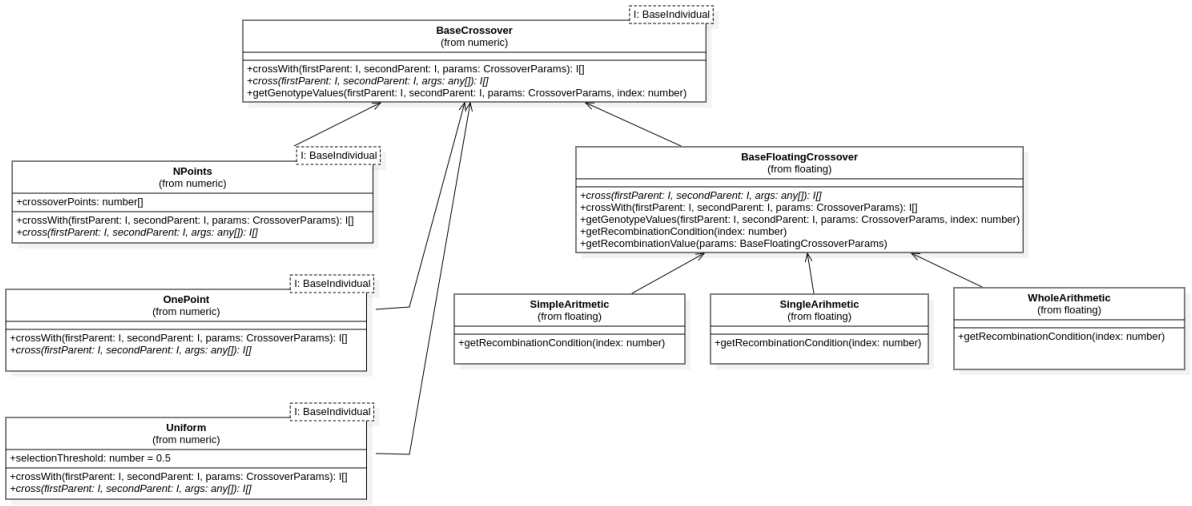


Figura 2.27: Diagrama de clase de las operaciones de cruce

Las operaciones de cruce son aquellas que se aplican entre los padres que han sido seleccionados, para así generar una descendencia. Los métodos de cruce entre individuos pueden ser muy diversos, y dependen normalmente del tipo de individuos a los que se les esté aplicando dicha operación.

Para implementar los operadores de cruce, hemos utilizado una jerarquía de herencia que se implementa a partir de una interfaz que contiene las operaciones básicas.

### BaseCrossover e interfaz Crossover

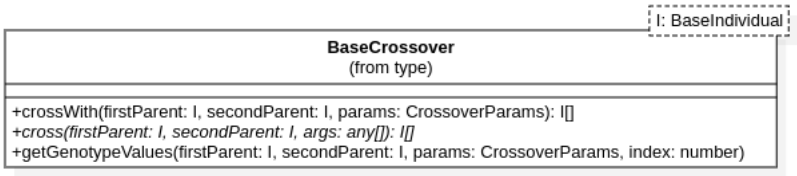


Figura 2.28: Diagrama de clase de `BaseCrossover`

En las operaciones de cruce que hemos considerado, se seleccionan dos individos como padres, y a partir de ellos, aplicando la operación correspondiente, se generan dos hijos como descendencia. Siguiendo la metodología utillizada previamente, lo que haremos será elaborar una interfaz (`Crossover`), en la que se definan los métodos que deberán tener todas las operaciones de cruce.

- `crossWith(firstParent: I, secondParent: I, params: Params): I[]`: Este método sirve para llevar a cabo las operaciones de crossover mediante los parámetros establecidos para cada uno de los métodos.
- `cross(firstParent: I, secondParent: I, ...args: any[]): I[]`: Este método sirve para llamar a la operación de cruce con dos padres y con los argumentos variables que se hayan decidido para cada método concreto. Sirve para ofrecer una llamada más amigable para la función, en lugar de pasar un objeto de parámetros.

Además de estos métodos, se ha includido una interfaz de parámetros básicos para estas operaciones (`CrossoverParams`), estos parámetros incluyen dos elementos de manera genérica:

- `engine`: Es el engine que se utilizará para el generador aleatorio.
- `individualConstructor`: Los operadores de cruce necesitan construir los individuos que conformarán la descendencia. Es por ello que un parámetro necesario para la operación de cruce es el constructor de los individos. Pasar este tipo de funciones como parámetro es una de las ventajas de usar un lenguaje como TypeScript.

A partir de la interfaz `Crossover` y los parámetros `CrossoverParams`, se ha construido una clase base abstracta (`BaseCrossover`). Esta clase contiene una implementación básica de una operación de cruce, de tal forma que facilita la tarea a la hora de llevar a cabo la extensión e implementación de nuestras propias operaciones.

El único método abstracto que posee esta clase y que se puede implementar es el siguiente:

- `getGenotypeValues(firstParent: I, secondParent: I, index: number)`: Este método se utiliza para obtener el valor del genotipo de la descendencia dado un índice concreto.

### NPointsCrossover

El crossover por n puntos (`NPointsCrossover`), es uno de los tipos de crossover más famosos [48]. Su procedimiento consiste en elegir dos padres, y elegir n puntos de cruce entre ellos aleatoriamente. Cada uno de los dos individuos de la descendencia se formará escogiendo de manera correlativa los genes de los padres.

Para nuestra implementación, se ha creado la clase `NPointsCrossover` que extiende de `BaseCrossover` y que tiene implementado el método `getGenotypeValues`, cuyo objetivo es

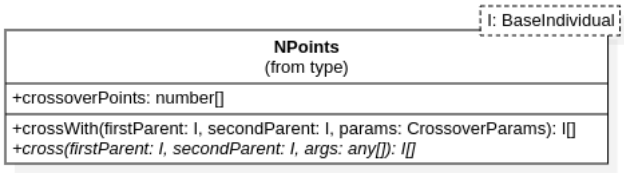


Figura 2.29: Diagrama de clase de `NPointsCrossover`

decidir de cual de cada uno de los padres elige el material genético, teniendo en cuenta que se debe hacer de manera alternativa.

A su vez, también se ha creado la interfaz `NPointsCrossoverParams`, esta interfaz sirve para especificar los parámetros de esta operación, extendiendo de `CrossoverParams`. El único parámetro adicional que se ha añadido ha sido:

- `numberOfCrossoverPoints`: es el número de puntos de cruce elegidos para llevar a cabo la operación.

Un ejemplo de su uso sería el siguiente:

```
1  import { NPointsCrossover, BinaryIndividual } from "genetics-js"
2
3  const cross = new NPointsCrossover();
4  const ind1 = new BinaryIndividual("0110010");
5  const ind2 = new BinaryIndividual("0100100");
6
7  /**
8  * Una posible descendencia que se generara con n = 3:
9  * 0 | 11 | 00 | 10 -> 0100000
10 * 0 | 10 | 01 | 00 -> 0110110
11 */
12 cross.crossWith(ind1, ind2, { numberOfCrossoverPoints: 3 }}))
```

### OnePointCrossover

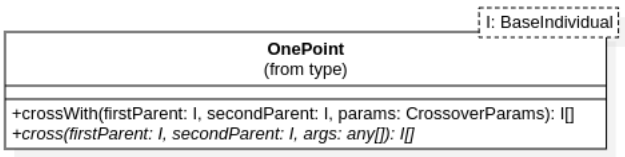


Figura 2.30: Diagrama de clase de `OnePointCrossover`

El crossover de un solo punto [35] es un caso concreto de el crossover de n puntos para $n = 1$. La implementación de este método es bastante sencilla una vez hecho el crossover de n puntos, puesto que tan solo habría que llamar a este método con `numberOfCrossoverPoints: 1`. Además, cabe destacar que este tipo de operación no tiene ningún parámetro adicional.

Un ejemplo de su uso sería:

```
import { OnePointCrossover, BinaryIndividual } from "genetics-js"

const cross = new OnePointCrossover();
const ind1 = new BinaryIndividual("0110010");
const ind2 = new BinaryIndividual("0100100");

/**
* Una posible descendencia que se generar a:
* 011 | 0010 -> 0110100
* 010 | 0100 -> 0100010
*/
cross.crossWith(ind1, ind2)
```

### UniformCrossover

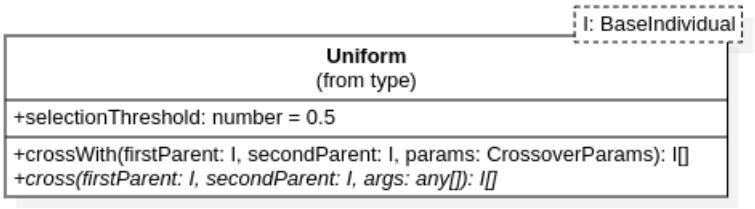


Figura 2.31: Diagrama de clase de `UniformCrossover`

El crossover uniforme [49] consiste en elegir de cuales de los dos padres se escoge el material genético en función de un valor aleatorio de probabilidad que se genera por cada gen, comparándolo con un valor umbral dado como parámetro. Este método de recombinación permite hacer variaciones más importantes en los cromosomas de la descendencia.

Para la implementación de este método se ha creado la clase `UniformCrossover` que extiende de `BaseCrossover`, además de la interfaz `UniformCrossoverParams` que extiende de `CrossoverParams` y que tiene como parámetro el umbral que se utilizará para determinar de cual de los padres se obtiene el material genético para la descendencia.

Un ejemplo de su uso podría ser:

```
import { UniformCrossover, BinaryIndividual } from "genetics-js"

const cross = new UniformCrossover();
const ind1 = new BinaryIndividual("0110010");
const ind2 = new BinaryIndividual("0100100");

/**
* Una posible descendencia que se generar a con umbral 0.5,
* y con los valores:
* [ 0.3, 0.5, 0.6, 0.1, 0.2, 0.8, 0.9 ]
* 0110010 -> 0100000
* 0100100 -> 0110110
*/
cross.crossWith(ind1, ind2, { selectionThreshold: 0.5 })
```

### BaseFloatingCrossover

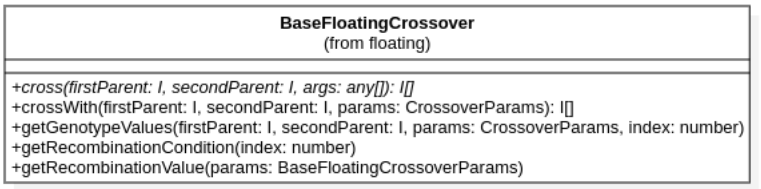


Figura 2.32: Diagrama de clase de `BaseFloatingCrossover`

Mientras que los tres tipos de operaciones de cruce descritos anteriormente pueden aplicarse a cualquier tipo de individuos, existen métodos que solo pueden aplicarse a individuos concretos. Es el caso de los métodos siguientes, que solo serán aplicados a individuos de punto flotante.

Para aglutinar los métodos de crossover de individuos flotantes [46] se ha creado la clase abstracta `BaseFloatingCrossover`. Esta clase, tiene implementado el método `getGenotypesValue`, de tal forma que en función de una condición que se implementará de manera concreta en las clases heredadas, se devolverá el valor del genotipo como:

$$z_i^x = \alpha x_i + (1 - \alpha) y_i \tag{2.1}$$

$$z_i^y = \alpha y_i + (1 - \alpha) x_i \tag{2.2}$$

Donde $x$ e $y$ son los padres elegidos en la operación de recombinación y $z^x$ y $z^y$ son los individuos $x$ e $y$ de la descendencia respectivamente. Todos ellos están evaluados en su gen $i$, es decir en el $i$-esimo valor de su genotipo.

Tal y como podemos comprobar, esta operación depende de un parámetro $\alpha$, que debe ser pasado como parámetro en la interfaz `BaseFloatingCrossoverParams` que exiende de `CrossoverParams`.

### SimpleArithmeticRecombination

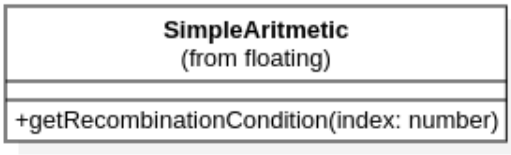


Figura 2.33: Diagrama de clase de `SimpleArithmeticRecombination`

Este es uno de los tres tipos de recombinación que se han implementado para individuos de punto flotante. Esta recombinación consiste en hacer la mezcla entre los valores de los genotipos de los padres seleccionados en función de si el índice del elemento del genotipo que se quiere obtener es menor que un cierto índice $k$ generado aleatoriamente.

La expresión que se utilizaría para definir a la descendencia sería:

$$< x_1, \ldots, x_k, \alpha \cdot y_{k+1} + (1 - \alpha) \cdot x_{k+1}, \ldots, \alpha \cdot y_n + (1 - \alpha) \cdot x_n > \tag{2.3}$$

$$< y_1, \ldots, y_k, \alpha \cdot x_{k+1} + (1-\alpha) \cdot y_{k+1}, \ldots, \alpha \cdot x_n + (1-\alpha) \cdot y_n > \quad (2.4)$$

Como vemos, este método de recombinación depende también de un parámetro $\alpha$.

La implementación de este método se ha hecho mediante la clase `SimpleArithmeticRecombination`, con la interfaz de parámetros `BaseFloatingCrossoverParams`, en la que ya viene especificado el parametro $\alpha$.

Un ejemplo de su uso sería el siguiente:

```
1  import {
2    SimpleArithmeticRecombination,
3    FloatingIndividual
4  } from "genetics-js"
5
6  const cross = new SimpleArithmeticRecombination();
7  const ind1 = new FloatingIndividual ("0.4 0.5 0.6 0.7 0.8 0.9");
8  const ind2 = new FloatingIndividual ("0.2 0.3 0.2 0.3 0.2 0.3");
9
10 /**
11 * Una posible descendencia que se generar a con alpha = 0.5,
12 * y k = 4
13 * 0.4 0.5 0.6 | 0.7 0.8 0.9 -> 0.4 0.5 0.6 | 0.5 0.5 0.6
14 * 0.2 0.3 0.2 | 0.3 0.2 0.3 -> 0.2 0.3 0.2 | 0.5 0.5 0.6
15 */
16 cross.crossWith(ind1, ind2, { alpha: 0.5 })
```

### SingleArithmeticRecombination

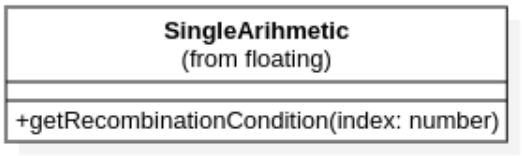


Figura 2.34: Diagrama de clase de `SingleArithmeticRecombination`

Esta operación destaca por producir pocos cambios en la descendencia respecto a los padres, pues tan solo se aplicará la expresión de mezcla de genotipos en una sola posición $k$ elegida de manera aleatoria.

La expresión que repesenta este método es la siguiente:

$$< x_1, \cdots, x_{k1}, \alpha \cdot y_k + (1-\alpha) \cdot x_k, x_{k+1}, \cdots, x_n > \quad (2.5)$$

$$< y_1, \cdots, y_{k1}, \alpha \cdot x_k + (1-\alpha) \cdot y_k, y_{k+1}, \cdots, y_n > \quad (2.6)$$

Para su implementación se ha creado la clase `SingleArtihmeticRecombination`, con la interfaz de parámetros `BaseFloatingCrossoverParams`.

Un ejemplo de uso sería:

```
import {
  SingleArithmeticRecombination,
  FloatingIndividual
} from "genetics-js"

const cross = new SingleArithmeticRecombination();
const ind1 = new FloatingIndividual ("0.4 0.5 0.6 0.7 0.8 0.9");
const ind2 = new FloatingIndividual ("0.2 0.3 0.2 0.3 0.2 0.3");

/**
* Una posible descendencia que se generar a con alpha = 0.5,
* y k = 4
* 0.4 0.5 0.6 0.7 0.8 0.9 -> 0.4 0.5 0.6 0.7 0.5 0.9
* 0.2 0.3 0.2 0.3 0.2 0.3 -> 0.2 0.3 0.2 0.3 0.5 0.3
*/
cross.crossWith(ind1, ind2, { alpha: 0.5 })
```

### WholeArithmeticRecombination

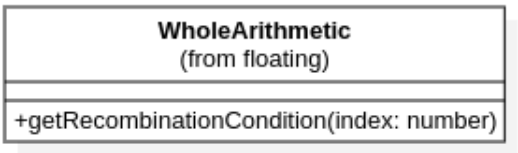


Figura 2.35: Diagrama de clase de `WholeArithmeticRecombination`

Este es el último tipo de recombinación para individuos de números en punto flotante. Se caracteriza por provocar un cambio bastante drástico en el genotipo de la descendencia respecto al genotipo de los padres, pues aplica la expresión de mezcla de genotipos en todos los genes de los individuos generados.

La expresión que lo caracteriza es la siguiente:

$$z^x = \overline{x} + (1 - \alpha) \cdot \overline{y} \tag{2.7}$$

$$z^y = \overline{y} + (1 - \alpha) \cdot \overline{x} \tag{2.8}$$

La implementación de este método se ha hecho mediante la clase `WholeArithmeticRecombination`.

Un ejemplo de uso sería:

```
import {
  WholeArithmeticRecombination,
  FloatingIndividual
} from "genetics-js"

const cross = new WholeArithmeticRecombination();
const ind1 = new FloatingIndividual ("0.4 0.5 0.6 0.7 0.8 0.9");
const ind2 = new FloatingIndividual ("0.2 0.3 0.2 0.3 0.2 0.3");

/**
* La descendencia que se generar a con alpha = 0.5:
* 0.4 0.5 0.6 0.7 0.8 0.9 -> 0.3 0.4 0.4 0.5 0.5 0.6
* 0.2 0.3 0.2 0.3 0.2 0.3 -> 0.3 0.4 0.4 0.5 0.5 0.6
*/
cross.crossWith(ind1, ind2, { alpha: 0.5 })
```

### 2.2.6. Mutaciones

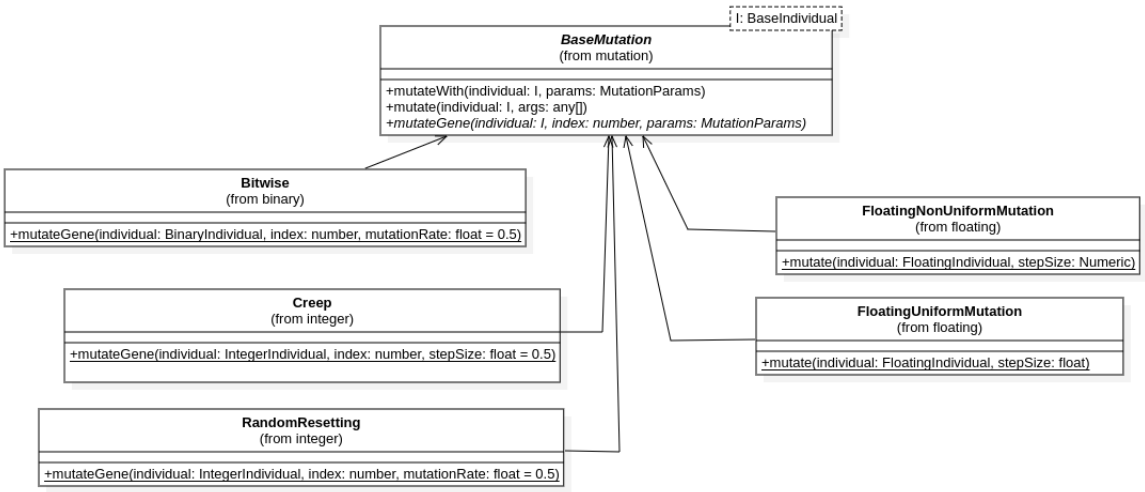


Figura 2.36: Diagrama de clase para los operadores de mutación

Las operaciones de mutación sirven para aplicar un cierto grado de aleatoriedad a los individuos generados como descendencia después de aplicar una operación de recombinación.

Dentro de los algoritmos evolutivos, este tipo de operaciones son las que incrementan el grado de diversidad existente en las soluciones obtenidas a partir de la recombinación, por ello se aplica a la descendencia obtenida.

Para su implementación se ha seguido el mismo patrón que con la recombinación, es decir, crear una jerarquía de clases que se implemente a partir de una interfaz base.

#### MutationBase e interfaz Mutation

Para definir las operaciones de mutación que hemos implementado, se ha creado la interfaz `Mutation`, la cual cuenta con dos métodos, que deben ser implementados para crear la operación deseada.

- `mutateWith(individual: I, params: Params): void`: Este método sirve para aplicar una mutación sobre el individuo que se pasa como parámetro con los parámetros que se indican.
- `mutate(individual: I, ...args: any[]): void`: Este método sirve también para aplicar la mutación sobre el individuo que se pasa como parámetro, sin embargo, sirve para especificar los parámetros de la mutación de manera más cómoda.

Al igual que ocurre con las operaciones de cruce, en la mutación también debemos definir los parámetros con los cuales se aplicará. Para ello, hemos definido la clase genérica `MutationParams`, la cual tiene un solo componente:

- `engine`: El engine que se utilizará para el generador aleatorio.

A partir de la interfaz `Mutation` y con los parámetros `MutationParams`, se ha creado una clase base abstracta (`MutationBase`) que implementa la mayoría de métodos utilizados para llevar a cabo operaciones de mutación.

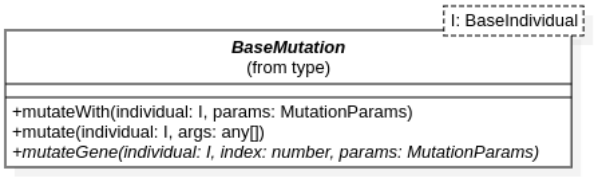


Figura 2.37: Diagrama de clase de `MutationBase`

Esta clase posee una serie de métodos abstractos que se deben implementar para extenderla y facilitar así la tarea de desarrollar nuestros propios procedimientos de mutación.

- `mutate(individual: I, ...args: any[]): void`: Este método heredado de la interfaz `Mutable` no se ha implementado, pues los parámetros concretos dependen del método de mutación.
- `mutateGene(individual: I, index: number, params: Params): void`: Este método abstracto será utilizado para mutar el gen especificado por el índice.

### BitwiseMutation

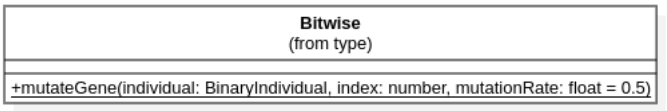


Figura 2.38: Diagrama de clase de `BitwiseMutation`

Esta mutación se aplica a individuos binarios (`BinaryIndividual`) y es muy común en muchos algoritmos evolutivos [40]. Consiste en intercambiar los bits del individuo, verdadero por falso y viceversa, en función de un parámetro que se conoce como la tasa de mutación (*mutation rate*).

Para implementar este método de mutación se ha creado la clase `BitwiseMutation`. Además se ha utilizado la interfaz `BitwiseMutationParams`, que extiende `MutationParams` y que incluye un miembro `mutationRate`, que especifica la probabilidad que tiene cada gen de ser mutado.

Un ejemplo de uso sería el siguiente:

```
1  import { BitwiseMutation, BinaryIndividual } from "genetics-js"
2
3  const mutation = new BitwiseMutation();
4  const ind = new BinaryIndividual("0010010");
5
6  /**
7  * Posible resultado tras elegir los bits 1 y 5 para ser mutados:
8  * 0010010 -> 0110000
9  */
10 mutation.mutateWith(ind, { mutationRate: 0.1 })
```

### CreepMutation

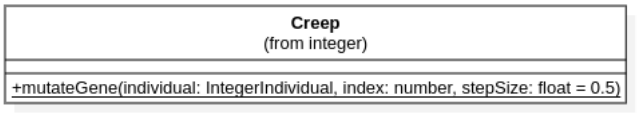


Figura 2.39: Diagrama de clase de `CreepMutation`

El siguiente tipo de mutación se aplica a individuos de tipo entero (`IntegerIndividual`), y consiste en mutar cada uno de los genes del individuo añadiendo una cantidad $\delta$ al valor del gen [34], obtenida a partir de una distribución normal de probabilidad y redondeado para devolver un número entero.

Esta distribución de probabilidad tiene como parámetros $\mu = 0$ y $\sigma$, donde el último se corresponde al *step size* o tamaño en el que se quiere variar el gen:

$$x_i = x_i + |\mathcal{N}(0, \sigma)| \tag{2.9}$$

Para implementar este método de mutación se ha utilizado la clase `CreepMutation`, y la interfaz `CreepMutationParams`, la cual tiene especificado el `stepSize` que se desea como parámetro.

Un ejemplo de uso sería el siguiente:

```
1  import { CreepMutation, IntegerIndividual } from "genetics-js"
2
3  const mutation = new CreepMutation();
4  const ind = new IntegerIndividual("1 3 4 5");
5
6  /**
7  * Un posible resultado con step size 2 es:
8  * 1 3 4 5 -> 0 4 2 7
9  */
10 mutation.mutateWith(ind, { stepSize: 2 })
```

### RandomResetting

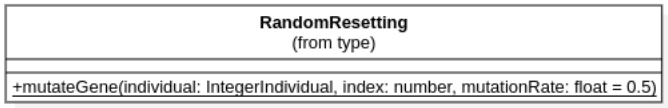


Figura 2.40: Diagrama de clase de `RandomResetting`

La operación de *random resetting* sería el equivalente al *bitwise mutation* para individuos enteros (`IntegerIndividual`). En este tipo de mutación, se elige de manera uniforme un número al azar entre el rango de valores permitidos, el cual se establecerá como el nuevo valor del gen mutado.

Para implementar este método se ha utilizado la clase `RandomResetting` la cual no tiene parámetros, puesto que el rango permitido de valores es elegido desde el propio individuo que se quiere mutar.

Un ejemplo de uso sería el siguiente:

```
1  import { RandomResetting, IntegerIndividual } from "genetics-js"
2
3  const mutation = new CreepMutation();
4  const ind = new IntegerIndividual("1 3 4 5", { range: [0, 7] });
5
6  /**
7  * Un posible resultado es:
8  * 1 3 4 5 -> 0 4 5 5
9  */
10 mutation.mutateWith(ind)
```

### FloatingNonUniformMutation

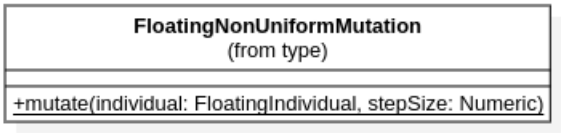


Figura 2.41: Diagrama de clase de `FloatingNonUniform`

Esta mutación es similar a *Creep mutation* pero para individuos en punto flotante. A partir de una distribución normal de probabilidad se obtiene un número flotante que se añade al valor del gen, haciendo que se produzca una variación del mismo.

$$x_i = x_i + \mathcal{N}(0, \sigma) \tag{2.10}$$

La distribución normal de probabilidad tiene dos parámetros ($\mu$ y $\sigma$), pero al igual que en *Creep mutation*, se establece el valor $\mu = 0$ y a $\sigma$ se le denomina *step size*, siendo la variable que controla la cantidad de variación que se aplicará a los genes.

Para la impelemntación de este método se ha creado la clase `FloatingNonUniformMutation`, y la interfaz de parámetros `FloatingNonUniformMutationParams` que contiene un único parámetro y es el `stepSize` que se le aplicará a la operación.

Un ejemplo de uso sería el siguiente:

```
1  import {
2    FloatingNonUniformMutation,
3    FloatingIndividual
4  } from "genetics-js";
5
6  const mutation = new FloatingNonUniformMutation();
7  const ind = new FloatingIndividual("1 3 4 5");
8
9  /**
10 * Un posible resultado con step size 1.0 es:
11 * 1 3 4 5 -> 1.2 2.45 4.75 6
12 */
13 mutation.mutateWith(ind, { stepSize: 1.0 })
```

### FloatingUniformMutation

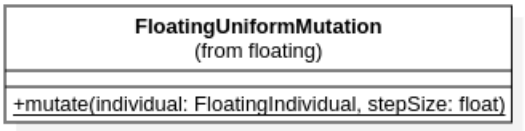


Figura 2.42: Diagrama de clase de `FloatingUniform`

De manera equivalente a como ocurre en el caso de individuos de números enteros, también existe un equivalente al `BitwiseMutation` para números en punto flotante y es la mutación uniforme. Esta mutación consiste en sustituir el gen mutado por un valor seleccionado de manera uniforme en el rango de valores permitidos.

La implementación de este método se lleva a cabo mediante la clase `FloatingUniformMutation`, que al igual que en `RandomResetting` no tiene parámetros adicionales puesto que el rango de valores es un atributo del individuo que se desea mutar.

Un ejemplo de uso sería el siguiente:

```
1  import { FloatingUniformMutation, FloatingIndividual } from "genetics-js";
2
3  const mutation = new FloatingUniformMutation();
4  const ind = new FloatingIndividual("1.2 3.4 4 5.0", { range: [0, 7] });
5
6  /**
7  * Un posible resultado es:
8  * 1.2 3.4 4 5.0 -> 0.0 1.22 4.5 7.0
9  */
10 mutation.mutateWith(ind)
```

### 2.2.7. Reemplazo

El reemplazo de los individuos es la fase que sigue una vez se ha generado la descendencia a partir de la población que se encontraba inicialmente en la generación.

Una vez que se ha generado esa descendencia tendremos una cantidad de individuos perteneciente a la población inicial y otra cantidad de individuos perteneciente a la descendencia, el proceso de selección se encargará de discernir cuales dentro de todos estos individuos serán los que pasen a la siguiente generación.

En la implementación de esta librería, hemos considerado que el tamaño de la población y descendencia es igual. Por tanto si definimos $\lambda$ como el tamaño de la población, el tamaño de la descendencia y de la población sería $\lambda + \lambda$, y de este tamaño debemos elegir tan solo $\lambda$ siguiendo algún criterio.

$$\lambda + \lambda \rightarrow \lambda \tag{2.11}$$

Los métodos de reemplazo que se han implementado han sido los siguientes:

#### AgeBasedReplacement

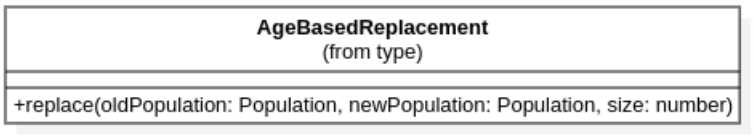


Figura 2.43: Diagrama de clase de `AgeBasedReplacement`

Este método de remplazo consiste en seleccionar a los individuos mas jóvenes de entre la población y la descendencia, también se conoce como algoritmo generacional. En nuestro caso, en el que los tamaños de la población y la descendencia coinciden, esta operación se corresponderá a seleccionar tan solo a los individuos de la descendencia.

La principal ventaja de este método es la eficiencia, pues a la hora de tener poblaciones de una gran cantidad de individuos tan solo deberían seleccionarse los individuos de la nueva población.

Para implementarlo se ha utilizado la clase `AgeBasedReplacement`, la cual tiene un único método `replace` que tiene como parámetros la población inicial y el offspring, a partir de los cuales se quiere generar la población de remplazo.

#### FitnessBasedReplacement

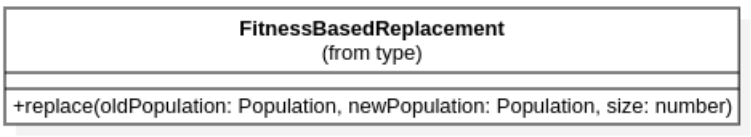


Figura 2.44: Diagrama de clase de `FitnessBasedReplacement`

El criterio que se sigue para llevar a cabo el remplazo mediante este método es el fitness que posee el individuo. De esta forma, cuando se tiene la población inicial y la descendencia,

los individuos de ambas se deben ordenar de acuerdo a su fitness, seleccionando solo los mejores.

Este método es bastante elitista, es decir que tan solo elige a los mejores de la población y la descendencia. Esto hace que se conserven los individuos mejores adaptados durante más generaciones. Sin embargo, también conlleva una pérdida de diversidad en los individuos y un aumento en la complejidad de la evaluación en poblaciones de un gran tamaño.

La implementación se ha hecho mediante la clase `FitnessBasedReplacement`, la cual tiene un método replace, que recibe la población inicial y la descendencia, al igual que en el caso anterior.

### 2.2.8. Fitness

La función de fitness se utiliza para evaluar cuanto de adaptado se encuentra un individuo al entorno. Si estamos hablando de un problema de optimización combinatoria, esta función se corresponde con la que queremos optimizar. En esta librería, la función de fitness se trata de maximizar, de tal forma que se consideran mejores las funciones de fitness con valores más altos.

En el caso en el que se quisiera minimizar la función se debería especificar a la hora de establecer los parámetros del algoritmo, y el procedimiento que se aplicaría internamente consistiría en invertir la función de fitness.

La implementación de la función de fitness ha sido bastante simple, de tal foma que solo se ha establecido el esquema de la función como un tipo.

```
(individual: I, ...args: any[]) =>number
```

Como podemos ver es una función que recibe como parámetros el individuo que se quiere evaluar además de cualquier parámetro adicional. De esta forma dependiendo de la aplicación concreta para la cual queramos utilizar esta librería, se debera implementar esta función de manera concreta.

Se realizará un ejemplo en el que se implementará una función de fitness que devuelva el número de valores `true` en un individuo binario:

```
1 import { FitnessFunction, BinaryIndividual } from "genetics-js";
2
3 const fitness: FitnessFunction<BinaryIndividual, boolean> =
4     (individual) => {
5         const count = 0;
6         individual.forEach(gene => count += gene ? 1 : 0);
7         return count;
8     }
```

### 2.2.9. Condición de finalización

La condición de finalización es la que indica cuando la ejecución del algoritmo evolutivo ha terminado. Este criterio de finalización se suele establecer en función del estado de la población en el momento concreto de la ejecución.

En el caso de **genetics.js**, la condición de parada que se ha logrado implementar es la de llegar a un número concreto de generaciones, es decir que el algoritmo se pare cuando se hayan ejecutado una serie de generaciones que se establecen como parámetro.

Esto se hace mediante la clase `MaxGenerations`, que recibe como parámetro el número máximo de generaciones que queremos que se ejecuten.

Un ejemplo de uso sería:

```
1 import { MaxGenerations } from "genetics-js";
2
3 const condition = new MaxGenerations(10);
4
5 /*
6 * Se puede utilizar este metodo para comprobar
7 * si la condici n de parada se ha establecido.
8 */
9 condition.isSatisfied(population);
```

# Capítulo 3

# Modo de uso de genetics.js

En este capítulo se expondrá una aplicación concreta implementada con **genetics.js**, la librería desarrollada. En esta implementación se ha construido una aplicación web usando React como framework de creación de interfaces de usuario, además de usar la librería para ejecutar un algoritmo evolutivo en el lado del cliente.

La intención que tiene este desarrollo es mostrar la capacidad de la que dispone la librería para ser ejecutada en el *front-end*, junto a tecnologías modernas para la creación de páginas web.

## 3.1. Descripción del problema a implementar

La aplicación que se ha implementado es un simulador del problema de la mochila (*Knapsack problem*), resuelto mediante un algoritmo evolutivo. La intención principal es que se puedan seleccionar la mayoría de parámetros posibles del algoritmo, para así poder ver como cambia el rendimiento con diferentes configuraciones de entrada.

Pero antes de describir el procedimiento concreto para desarrollar la aplicación, es importante realizar una definición del problema que se pretende abordar. Cabe destacar que el problema de la mochila es clásico en el ámbito de la optimización combinatoria [41], y ha sido objeto de estudio desde el nacimiento del campo.

En la formulación del problema disponemos de $n$ items que pueden ser introducidos en la mochila. Cada item tiene dos atributos, su peso ($w_i$) y su valor o beneficio ($v_i$). De esta forma, la solución al problema trata de descubrir cual es el conjunto de items que debe introducirse en la mochila de tal forma que se maximize el beneficio total pero sin sobrepasar la capacidad ($W$) de la mochila.

De manera formal se puede describir el problema de esta forma:

$$\begin{aligned} \max_{X} \quad & \sum_{i=1}^{n} v_i x_i \\ \text{sujeto a} \quad & \sum_{i=1}^{n} w_i x_i \leq W \\ & x_i \in \{0, 1\} \end{aligned}$$

Donde $X = \{x_1, \ldots, x_n\}$ es el conjunto de variables de decisión que establecen si el item

$i$-esimo está o no presente en la mochila.

Existen múltiples técnicas para abordar este problema, incluso pudiendo ser resuelto mediante una técnicas exactas. Pero con motivo de probar la librería desarrollada, el problema se resolverá con un algoritmo genético.

La codificación que se utilizará para los individuos es la **codificación binaria**, donde cada uno de los genes del individuo representará si el ítem en la posición establecida se encuentra o no en la mochila.

Los parámetros del algoritmo que podrán seleccionarse serán:

- **Tamaño de la población**: El tamaño de la población deberá estar comprendido entre 0 y 25. Donde el límite superior se ha establecido por criterios de claridad en la visualización.
- **Tipo de operador de cruce**: El operador de cruce que se puede seleccionar puede ser el crossover de un punto (`OnePointCrossover`), el crossover de n puntos (`NPointsCrossover`) o el crossover uniforme (`UniformCrossover`). Cabe destacar que dependiendo del operador elegido, se deberán establecer sus parámetros.
- **Método de selección de padres**: Se puede elegir entre *roulette wheel* y *stochastic universal sampling*.
- **Método de reemplazo**: Se puede seleccionar entre el remplazo basado en la edad (`AgeBasedReplacement`) y el basado en el fitness (`FitnessBasedReplacement`).
- **La tasa de mutación**: La tasa de mutación (*mutation rate*), puede establecerse, teniendo 0.5 como valor por defecto.
- **Numero máximo de generaciones**: El número máximo de generaciones será también un parámetro a establecer, aunque por defecto son 50.

## 3.2. Desarrollo de la aplicación

Para implementar la aplicación web, hemos utilizado **React** [17] como librería para la creación de la interfaz de usuario, y **TypeScript** como lenguaje de desarrollo. Esta elección se ha tomado porque es mucho más cómodo y flexible trabajar con una librería como React para crear la interfaz de usuario en lugar de trabajar con HTML y CSS directamente.

Además, la metodología de trabajo de React nos permite desarrollar a partir de componentes, lo que añade una gran modularidad al diseño, simplificando la tarea de mostrar la información referente al algoritmo que pretendemos ejecutar.

Para simplificar la tarea de gestionar diferentes elementos visuales se ha utilizado la librería **Material UI** [12], que provee una serie de componentes inspirados en el estandard Material Design, desarrollado por Google. Como la aplicación cuenta con bastantes formularios diferentes, se ha utilizado la librería **Formik** [8] para la gestión, además de usar **Yup** [29] para garantizar la validez de los datos introducidos.

En cuanto al diseño visual de la aplicación, cabe destacar que contamos con dos pantallas, la primera utilizada para establecer los parámetros del algoritmo y la segunda para mostrar la

ejecución del mismo. Explicaremos ambas pantallas por separado.

En primer lugar, expondremos la pantalla principal. En ella podremos seleccionar todos los parámetros referentes al algoritmo además de establecer los items que se encontrarán en la mochila, así como su capacidad.

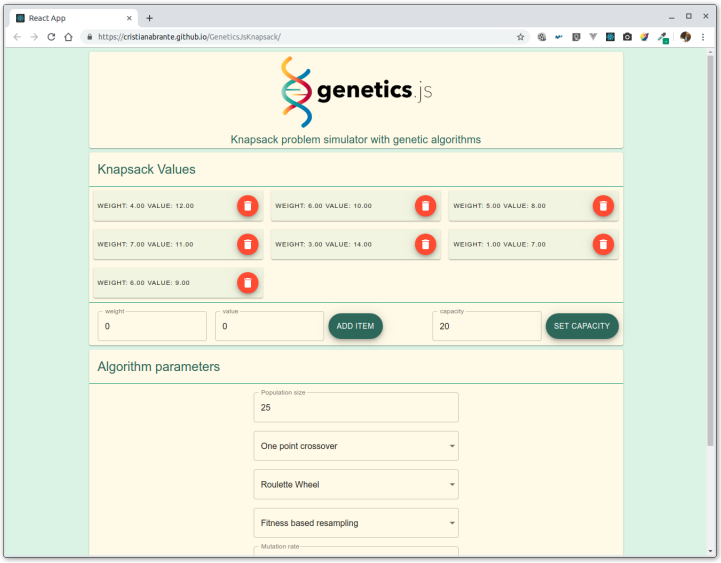


Figura 3.1: Pantalla principal de la aplicación web desarrollada

En la parte central de la pantalla tendremos el formulario en el que se muestran los items que formarán parte de la mochila. Podremos introducir items y también eliminarlos, pudiendo a su vez establecer el peso y cantidad del item en cuestión. Además, en la parte derecha contamos con un campo de texto en el que podremos introducir la capacidad de la mochila. Cabe señalar que ambos formularios están verificados para impedir que se introduzcan valores erróneos.

El segundo formulario con el que nos encontramos es el de parámteros del algoritmo. En el podremos elegir cuales serán las operaciones que se aplicarán a la hora de ejecutar el algoritmo genético. Al igual que ocurría en el caso anterior, en este formulario también se ha llevado a cabo una verificación para evitar los valores erróneos.

Una vez pulsamos el boton **execute algorithm** de la parte inferior, la aplicación nos llevaría a una segunda pantalla en la que se mostraría la ejecución del algoritmo.

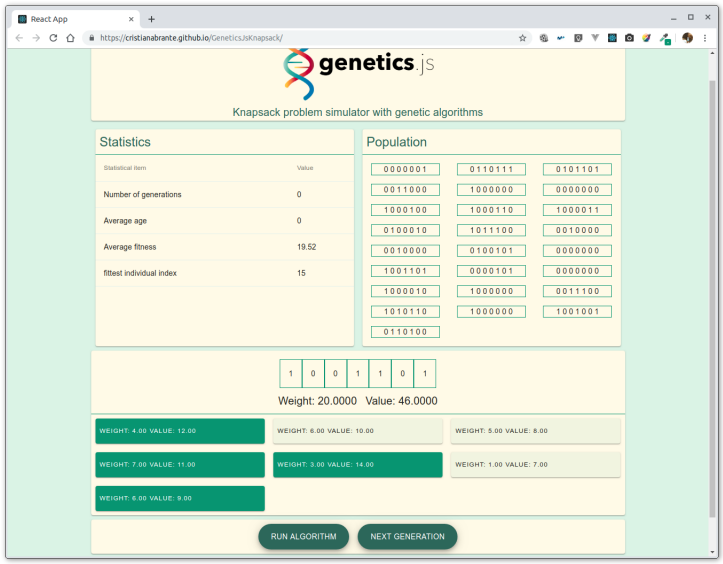


Figura 3.2: Pantalla de ejecución del algoritmo

Esta pantalla está dividida en una serie de secciones diferentes. En la parte izquierda, contamos con una tabla en la que se muestran las estadísticas del algoritmo evolutivo, actualizadas durante la ejecución. Entre las estadísticas que se muestran encontramos: el número de generaciones, la edad media de los individuos, o el fitness medio de la población.

A la izquierda de este recuadro se puede ver una representación visual de la población, la cual se irá actualizando a medida que el algoritmo se ejecute. La visualización de este apartado es la que impide que se tengan tamaños de población muy grandes.

Finalmente y en la parte inferior, nos encontramos una representación del individuo con un mayor valor de fitness, indicando cuales son los items que ha escogido para introducir en la mochila, así como el peso y valor que tendrá dicha selección.

La ejecución del algoritmo se puede hacer de manera contínua, ejecutando de una sola vez el algoritmo hasta que se llegue a un máximo de generaciones, o también se puede hacer paso a paso, por si se quiere depurar el resultado.

El despliegue de esta aplicación se ha hecho en **GitHub pages**, y por tanto está disponible en este enlace: https://cristianabrante.github.io/GeneticsJsKnapsack/.

# Capítulo 4

# Conclusiones y líneas futuras

Tras haber completado el desarrollo de este proyecto, se debe hacer un balance para determinar cual ha sido el grado de satisfacción con el resultado, y de esta forma señalar si la se ha logrado la consecución de los objetivos y establecer cuales serán las líneas futuras de desarrollo.

En primer lugar, cabe destacar la dificultad que entraña el diseño e implementación de un framework de computación evolutiva. Esta dificultad viene determinada sobretodo por la importancia actual que tiene este campo dentro de la computación y la inteligencia artificial, por lo cual existen una gran cantidad de aplicaciones diferentes que desarrollan nuevos métodos y técnicas cada vez más precisas y avanzadas. De esta forma, crear una librería que sea lo suficientemente flexible como para adaptarse correctamente a estos cambios, pero que a la vez disponga de una implementación concreta para las funciones más comunes entraña dificultades muy importantes.

De esta forma, considero que los métodos y técnicas implementados cubren la mayoría de aplicaciones básicas, aunque para abarcar muchas más técnicas y métodos, se debería intentar utilizar esta librería para problemas más avanzados. De esta forma se determinarían las carencias existentes y podrían ser solucionadas de manera paulatina.

En cuanto a las tecnologías de desarrollo, creo que la mayoría de decisiones han sido acertadas. Por una parte, porque se han elegido herramientas actuales, que cuentan con un gran número de usuarios, lo cual garantiza en mayor medida su estabilidad. Y por otra parte, porque al realizar la implementación de la libería adaptada a la web se puede garantizar que se pueda utilizar esta librería con las aplicaciones que se desarrollen en un futuro. Además, el desarrollo como librería web, es el más flexible pues ya no solo podrá ejecutarse en el *front-end*, sino que también nos permite ser ejecutada en un servidor.

A continuación, pasaré a describir las lineas futuras que tiene este trabajo, las cuales he decidido dividir en varias categorías, para así distinguir que trabajo se llevará a cabo en cada punto concreto.

En primer lugar expondrá las lineas futuras en cuanto a **tecnologías utilizadas**:

- **Utilizar un monorepo para estructurar el repositorio**: La necesidad de utilizar un monorepo es gestionar mejor las diferentes partes de las que se compone este proyecto: por una parte tendríamos la propia librería, luego la web con la documentación y finalmente un paquete con ejemplos de implementaciones concretas para problemas comunes como el TSP o el problema de la mochila. Para manejar este *monorepo* se puede utilizar la tecnología **Lerna** y **Yarn** en lugar de NPM para manejar la instalación de dependencias

y la ejecución de scripts.

- **Cambio de tecnología de documentación**: La elección de **TypeDoc** como instrumento de documentación no creo que haya sido una mala idea. Sin embargo, creo que este estilo de documentación no puede ser escalable al futuro, sino que se necesita hacer una página web a medida para la documentación, similar a las que existen en otros proyectos de grandes dimensiones. La tecnología a la que propongo realizar la migración es **Docusaurus**, principalmente por su gran integración con **React** y **Gatsby**.

En segundo lugar, estas serán las líneas futuras que se seguirán en cuanto al **desarrollo**.

- **Reestructurar la clases desarrolladas más recientemente**: Por motivo de escasez de tiempo, los métodos implementados en último lugar (selección y criterio de finalización), no se han hecho con el mismo trabajo de diseño y desarrollo que las otras, con lo cual necesitarían una reestructuración.
- **Reestructurar la organización de ficheros**: En muchos casos existe una mala organización de ficheros, lo cual debe corregirse.
- **Mejora de los tests**: Actualmente, para testear la librería se utiliza una batería de tests, pero aun no existen suficientes ejemplos como para garantizar el correcto funcionamiento en todos los módulos.
- **Implementación de un *Command Line Interface* (CLI)**: Creo que una buena idea es que se permitiera ejecutar algunos algoritmos mediante la linea de comandos, pudiendo elegir los parámetros de manera sencilla y garantizando que existe una correcta visualización en la terminal. Para este propósito, propongo la herramienta **enquirer**
- **Soporte de asincronía**: Los algoritmos evolutivos son una tarea que requiere de una gran carga computacional, así que sería un gran avance que estas tareas se pudieran ejecutar de manera asíncrona.

Finalmente, ofreceré unas líneas futuras en cuanto a la **algoritmia y nuevas funcionalidades**:

- **Permitir tests estadísticos por parte del usuario**: Una tarea fundamental para evaluar un algoritmo evolutivo es ejecutar tests estadísticos sobre el resultado, es una tarea que se le debe facilitar al usuario.
- **Implementar individuos de permutación**: Este tipo de individuos son ampliamente utilizados, creo que es importante ofrecer una implementación a medida para ellos.
- **Considerar individuos numéricos multirango**: Por ahora, los individuos numéricos solo pueden tener un rango, sería interesante poder extender ese comportamiento.
- **Considerar individuos no factibles y permitir operadores de reparación**: Actualmente no se consideran los individuos no factibles. Creo que se deberían añadir en un futuro además de diseñar operadores específicos que reparen soluciones no factibles.
- **Implementar otras categorías de algoritmos**: Implementar otras categorías de algoritmos evolutivos como los algoritmos meméticos sería muy útil para los usuarios.

# Capítulo 5

# Summary and Future lines

After having completed the development of the project, it is necessary to make a balance to determine which is the satisfaction degree with the result, thus knowing if the objectives has been reached and which future lines of development we can follow.

Should be noted the difficulty of designing and implementing an evolutionary computing framework. This difficulty is determined mostly for the actual relevance of the field, which aims to create a big amount of different applications which develop new advanced and precise methods and techniques. In this way, creating a library which is flexible enough for adapting correctly to those changes and at the same time which provides the most common features has enormous difficulties.

Hence, I consider that the methods and techniques implemented cover the majority of the basic applications. Nevertheless it should be tested in more demanding scenarios for pointing out the existing weaknesses to solve them gradually.

I think that the decisions about the development technologies were right. On the one hand, because modern technologies, which are frequently applied by a significant number of users, where considered. The above ensures the stability of the library designed. On the other hand, because the development of a web application have a lot of future perspective, due to its ability for being executed on the *front-end* and on the server.

Next, I am going to describe the future lines of the project, divided into different categories. First, I am going to explain the future lines concerning the **used technologies**:

- **Using a monorepo for structuring the repository**: The need for using a monorepo is aimed to manage in a better way the different parts of the project. We can have a package in the monorepo which contains the library itself, other for the documentation, and finally one with examples with common test problems. I recommend using **Lerna** and **Yarn** instead of NPM for managing the dependencies.
- **Changing the documentation technology**: Although **TypeDoc** was a good election as a documentation system, I think that its documentation style is not scalable for the future. This is why I propose the migration to a better option like **Docusaurus**, mostly because of its great integration with **React** and **Gatsby**.

Secondly, the future lines about **development** are going to be explained:

- **Restructuring the most recently developed classes**: Due to time restrictions, the last methods implemented (selection and stopping criteria), were not so developed as the remaining ones, so they would need to be restructured.

- **Restructuring the files**: In many cases there exists a bad organization in files, which should be fixed.
- **Improving the tests**: Currently, for testing the library, a benchmark is used, but is is not robust enough, and consequently, it does not ensure the correct working of all the modules.
- **Implementation of a Command Line Interface (CLI)**: I think that a good idea will be allowing to execute some algorithms via command line. For this purpose I recommend the usage of **enquirer**.
- **Asynchrony support**: Some evolutionary algorithms take a lot of computing resources, so executing those tasks in an asynchronous way would be a great advance.

Finally I would offer some future lines about **algorithms and new features**:

- **Allowing the user to execute statistical tests**: A fundamental task for evaluating an evolutionary algorithm is to execute statistical tests over the results, so this possibility should be provided to the user.
- **Implementing permutation individuals**: This type of individuals are widely used, so it would be interesting to have a native implementation for them.
- **Considering multi-range numerical individuals**: Nowadays, only a general range is allowed in numeric individuals. As a result, this behaviour should be extended.
- **Considering not feasible individuals**: Currently, not feasible individuals are not allowed. I think that in the future they should be considered.
- **Implementing new algorithms categories**: New algorithm categories, like memetic algorithms, should be added to the library.

# Capítulo 6

# Presupuesto

El presupuesto elaborado para este proyecto contiene dos partes fundamentales. Por una parte tenemos los costes tecnológicos aparejados al desarrollo y por otra debemos tener en cuenta los recursos humanos.

En primer lugar se expondrán los costes tecnológicos:

| Tipo | Descripción | Coste |
|---|---|---|
| NPM | Hosting privado en NPM con múltiples paquetes | 21€ (7€/mes) |
| CircleCI | Serivicio premium de CircleCI para múltiples tests en paralelo | 150€ |
| dominio | Registrar el dominio genetics.js para su utilización en la web de documentación | 50€ |
| Hosting | Hosting privado para alojar la documentación | 100€ |
| Plantilla web | Adquisición de una plantilla web para la documentación | 50€ |
| | | 371€ |

Tabla 6.1: Costes tecnológicos

Para calcular los costes de recursos humanos, se debe aclarar que la duración de este proyecto es de **300** horas puesto que la asignatura **Trabajo fin de grado** cuenta con 12 créditos ECTS y cada crédito equivale a 25 horas.

| Tipo | Descripción | Coste |
|---|---|---|
| Trabajo de desarrollo | Este trabajo tiene un coste de 15€ por hora. | 4500€ |

Tabla 6.2: Costes de recursos humanos

Finalmente, calcularemos los costes totales:

| Tipo | Costes |
|---|---|
| Coste tecnológico | 371€ |
| Recursos humanos | 4500€ |
| | 4871€ |

Tabla 6.3: Costes totales

# Bibliografía


[1] Random de C++. http://www.cplusplus.com/reference/cstdlib/rand/.

[2] random.js. https://github.com/ckknight/random-js.

[3] Airbnb JavaScript standard. https://github.com/airbnb/javascript, 2019. [Online; accedido en junio de 2019].

[4] Atom. https://atom.io/, 2019. [Online; accedido en junio de 2019].

[5] Babel. https://babeljs.io/, 2019. [Online; accedido en junio de 2019].

[6] Coveralls. https://coveralls.io/github/CristianAbrante/GeneticsJS, 2019. [Online; accedido en junio de 2019].

[7] Estándard de JavaScript (ECMAScript). https://www.ecma-international.org/ecma-262/9.0/index.htmlTitle, 2019. [Online; accedido en junio de 2019].

[8] Formik. https://jaredpalmer.com/formik/, 2019. [Online; accessed may-2019].

[9] Git. https://git-scm.com/, 2019. [Online; accedido en junio de 2019].

[10] Informe de CircleCI. https://circleci.com/gh/CristianAbrante/workflows/GeneticsJS, 2019. [Online; accedido en junio de 2019].

[11] Jest. https://jestjs.io/, 2019. [Online; accedido en junio de 2019].

[12] Material UI. https://material-ui.com/, 2019. [Online; accessed may-2019].

[13] Motor V8. https://v8.dev/, 2019. [Online; accedido en junio de 2019].

[14] NodeJS. https://nodejs.org/es/, 2019. [Online; accedido en junio de 2019].

[15] NPM: Node Package Manager. https://www.npmjs.com/, 2019. [Online; accedido en junio de 2019].

[16] Prettier. https://prettier.io/, 2019. [Online; accedido en junio de 2019].

[17] ReactJS. https://reactjs.org/, 2019. [Online; accessed may-2019].

[18] Repositorio de TypeScript. https://github.com/Microsoft/TypeScript, 2019. [Online; accedido en junio de 2019].

[19] Repositorio en GitHub. https://github.com/CristianAbrante/GeneticsJS/, 2019. [Online; accedido en junio de 2019].

[20] Semantic versioning. https://semver.org/, 2019. [Online; accedido en junio de 2019].

[21] TravisCI. https://travis-ci.org/, 2019. [Online; accedido en junio de 2019].

[22] ts-jest. https://github.com/kulshekhar/ts-jest, 2019. [Online; accedido en junio de 2019].

[23] TS Lint. https://palantir.github.io/tslint/, 2019. [Online; accedido en junio de 2019].

[24] TypeDoc. https://cristianabrante.github.io/GeneticsJS/, 2019. [Online; accedido en junio de 2019].

[25] TypeScript. https://www.typescriptlang.org/, 2019. [Online; accedido en junio de 2019].

[26] Visual Studio Code. https://code.visualstudio.com/, 2019. [Online; accedido en junio de 2019].

[27] WebPack. https://webpack.js.org/, 2019. [Online; accedido en junio de 2019].

[28] WebStorm IDE. https://www.jetbrains.com/webstorm/, 2019. [Online; accedido en junio de 2019].

[29] Yup. https://github.com/jquense/yup, 2019. [Online; accessed may-2019].

[30] James E Baker. Reducing bias and inefficiency in the selection algorithm. In *Proceedings of the second international conference on genetic algorithms*, volume 206, pages 14–21, 1987.

[31] Theodore Baker, John Gill, and Robert Solovay. Relativizations of the p=?np question. *SIAM Journal on computing*, 4(4):431–442, 1975.

[32] Jason Brownlee et al. Oat: The optimization algorithm toolkit. *Complex Intelligent Systems Laboratory (CIS), Centre for Information Technology Research (CITR), Faculty of Information and Communication Technologies (ICT), Swinburne University of Technology, Victoria, Australia, Technical Report A*, 20071220, 2007.

[33] Sébastien Cahon, Nordine Melab, and E-G Talbi. Paradiseo: A framework for the reusable design of parallel and distributed metaheuristics. *Journal of heuristics*, 10(3):357–380, 2004.

[34] Lawrence Davis. Handbook of genetic algorithms. 1991.

[35] Kenneth Alan De Jong. Analysis of the behavior of a class of genetic adaptive systems. 1975.

[36] Juan J Durillo and Antonio J Nebro. jmetal: A java framework for multi-objective optimization. *Advances in Engineering Software*, 42(10):760–771, 2011.

[37] Agoston E Eiben, James E Smith, et al. *Introduction to evolutionary computing*, volume 53. Springer, 2003.

[38] Michael R Garey and David S Johnson. *Computers and intractability*, volume 29. wh freeman New York, 2002.

[39] John H Holland. Genetic algorithms and the optimal allocation of trials. *SIAM Journal on Computing*, 2(2):88–105, 1973.

[40] John Henry Holland et al. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[41] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[42] Richard Kenneth Eng. JavaScript is a Dysfunctional Programming Language. https://medium.com/javascript-non-grata/javascript-is-a-dysfunctional-programming-language-a1f4866e186f, 2019. [Online; accedido en junio de 2019].

[43] Coromoto León, Gara Miranda, and Carlos Segura. Metco: a parallel plugin-based framework for multi-objective optimization. *International Journal on Artificial Intelligence Tools*, 18(04):569–588, 2009.

[44] Adam Lipowski and Dorota Lipowska. Roulette-wheel selection via stochastic acceptance. *CoRR*, abs/1109.3627, 2011.

[45] Martin Lukasiewycz, M Glaß, F Reimann, and S Helwig. Opt4j-the optimization framework for java, 2009.

[46] Zbigniew Michalewicz. *Genetic algorithms+ data structures= evolution programs*. Springer Science & Business Media, 2013.

[47] Christos H Papadimitriou. *Computational complexity*. John Wiley and Sons Ltd., 2003.

[48] William M Spears and Kenneth A De Jong. An analysis of multi-point crossover. In *Foundations of genetic algorithms*, volume 1, pages 301–315. Elsevier, 1991.

[49] Gilbert Syswerda. Uniform crossover in genetic algorithms. In *Proceedings of the third international conference on Genetic algorithms*, pages 2–9. Morgan Kaufmann Publishers, 1989.