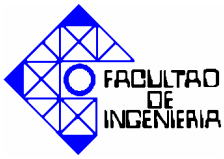
Universidad de Carabobo
Facultad de Ingeniería
Escuela de Ingeniería Industrial



Entorno de trabajo computacional para el desarrollo de algoritmos genéticos

Saúl González Uzcanga

Bárbula, mayo de 2013

Entorno de trabajo computacional para el desarrollo de algoritmos genéticos

S. González Uzcanga[*], E. Pérez Pérez[**]

Escuela de Ingeniería Industrial, Facultad de Ingeniería,
Universidad de Carabobo. Bárbula, Venezuela
mayo de 2013

Resumen

Este trabajo consiste en el diseño e implementación de un entorno de trabajo computacional (mejor conocido por su nombre en ingles, framework de software) para desarrollar algoritmos genéticos escrito en el lenguaje de programación Java. La arquitectura del framework consiste en una librería de 23 clases dividida en dos paquetes. La implementación del entorno se logra al heredar de la clase `Individuo` y definir la estructura del cromosoma, la función del aptitud, el cruce y la mutación. Se pueden escoger entre varios tipos de individuos pre configurados y varios tipos de selección y otras reglas como elitismo o selección post-cruce, aunque el usuario puede diseñar sus propios métodos de selección. Con la finalidad de describir cómo implementar la librería, se explica paso a paso el diseño (código) de una aplicación, además de cómo modificar sus distintos parámetros. El proyecto de software derivado de este Trabajo se encuentra en:

`http://saulguillermo.github.io/agapi`

Palabras claves: algoritmos genéticos, entorno java algoritmos genéticos, interfaz de programación algoritmos genéticos.

[*]Autor < `saulguillermo@gmail.com`>
[**]Tutor Académico

Índice general

Introducción

Los algoritmos genéticos son técnicas de búsqueda y optimización basadas en el principio de la genética y la selección natural inventados en los años 70 por John Holland. Actualmente y, gracias a los avances de la computación, los algoritmos genéticos se han vuelto herramientas útiles para la resolución de un amplio espectro de problemas que van desde la Biología hasta la Ingeniería.

El desarrollo de aplicaciones computacionales que utilizan algoritmos genéticos se realiza en función de las características específicas del problema a resolver, sin embargo un marco de trabajo general de programación, en muchos casos, brindaría muchas ventajas principalmente desde el punto de vista del desarrollo del código en lenguaje de programación. Este Trabajo propone un entorno de trabajo computacional escrito en lenguaje Java capaz de cumplir con ese deseo de estandarización del proceso de implementación de aplicaciones basadas en algoritmos genéticos.

En el capítulo 1 se plantea el problema descrito en el párrafo anterior y se busca responder la pregunta ¿Es posible elaborar un entorno de trabajo computacional que permita el desarrollo de Algoritmos Genéticos, escrito en un lenguaje de alto nivel y con una estructura que permita su uso por parte de estudiantes de pre-grado de Ingeniería y carreras afines? Esta interrogante conduce al plantear el objetivo general de este trabajo: la elaboración de un entorno de trabajo computacional reutilizable que permita el desarrollo de algoritmos genéticos escrito en un lenguaje de alto nivel.

El segundo capítulo describe proyectos que se han desarrollado en el área de librerías para algoritmos genéticos como JGAP y JAGA. Se hace referencia trabajos conocidos a partir de los cuales se ha creado un marco teórico que explica como funcionan los algoritmos genéticos y sus características. Además se describe el funcionamiento del lenguaje de programación Java y la estructura de un *framework* de software.

El siguiente capítulo describe la metodología utilizada, planteada como un proyecto de software con características como documentación en texto y formato HTML y suministro de código fuente.

La estructura y funcionamiento del framework se describe en el capitulo 4. Dicha estructura esta basada en 23 clases de Java, distribuidas en dos paquetes: un paquete con el núcleo donde tiene lugar la ejecución del algoritmo genético, y otro con clases opcionales disponibles con el objeto de facilitar el desarrollo de aplicaciones.

La documentación de las clases del framework es el tema del quinto capitulo. Esta documentación es una adaptación del formato electrónico HTML conocido como javadocs. Aquí se describe cada método de cada clase a modo referencia rápida para cuando se este desarrollando un nueva aplicación basada en el framework.

Por último, en el capitulo 6, se plantea un problema matemático simple que se busca resolver con una aplicación basada en el framework. Aquí se explica paso a paso cómo implementar el framework llamado *agapi* (API para algoritmos genéticos), desde su descarga web hasta la corrida exitosa del programa.

Capítulo 1.

El Problema

1.1. Planteamiento del Problema

El advenimiento de las computadoras electrónicas ha sido uno de los inventos más revolucionarios en la historia de la ciencia y la tecnología. Esta revolución ha incrementado profundamente nuestra habilidad de predicción y control de la naturaleza en formas que eran impensables hace menos de cien años. Para muchos (Mitchell, 1998) la cúspide de los logros de esta revolución es el desarrollo de inteligencia en la forma de programas de computadoras, lo que más tarde se denominó Inteligencia Artificial (Russell y Norvig, 2009).

Dentro de este nuevo campo de estudios se encuentra un área denominada Computación Evolutiva que se ha ido desarrollando desde los años cincuenta por científicos que, de forma independiente, han estudiado sistemas evolutivos (copiados de la naturaleza) con la idea que pueden ser utilizados como herramientas de optimización en problemas de ingeniería.

La optimización, entendida como el proceso de hacer algo cada vez mejor (Haupt y Haupt, 2004) tenia como finalidad buscar de forma iterativa alguna solución óptima sin importar el desempeño del método utilizado (algoritmo), ello debido a sus orígenes en el Cálculo (Goldberg,

1989). Actualmente el enfoque presta igual atención al desempeño ya que existen problemas donde es muy difícil, sino imposible hallar una solución óptima.

En los años sesenta John Holland y sus estudiantes desarrollaron una técnica conocida como Algoritmos Genéticos que, junto a las Estrategias Evolutivas y la Programación Evolutiva, se convirtieron en los pilares de la Computación Evolutiva y son actualmente herramientas claves para la resolución de un amplio espectro de problemas en campos que van desde la Biología hasta la Ingeniería pasando por infinidad de áreas tanto sociales como científicas.

Los Algoritmos Genéticos son técnicas de búsqueda y optimización basadas en el principio de la genética y la selección natural. Un Algoritmo Genético permite a una población compuesta por muchos individuos evolucionar bajo unas reglas especificas de selección a un estado que maximiza su *aptitud*. Los individuos de dicha población representan distintas soluciones del problema y son evaluadas por medio de la función de aptitud que es una adaptación de la función objetivo. Luego bajo un criterio especifico de selección (Ej. Los individuos con mayor aptitud) son seleccionados para cruzarse y mutarse, esto con el objeto de obtener nuevas soluciones que puedan mejorar en promedio la aptitud de la población anterior.

Las características de los problemas aptos para ser resueltos por medio de Algoritmos Genéticos incluyen: Problemas con espacios de soluciones muy grandes, no unimodales y poco comprendidos. Problemas con funciones objetivos con mucho ruido, donde no se requiera estrictamente de una solución óptima global, donde una búsqueda rápida sea suficiente.

Haupt y Haupt (2004) presentan algunas de las ventajas que ofrece un Algoritmo Genético sobre otros métodos exactos o heurísticos:

- Optimiza tanto variables discretas como continuas.

- Realiza una búsqueda de gran amplitud y de forma simultánea del espacio de soluciones.
- Optimiza variables con funciones objetivos muy complejas (puede liberarse de óptimos locales).
- En cada iteración provee una lista de soluciones en vez de una a la vez.
- Puede codificar las variables de modo que la optimización no se relacione con el tipo de variable.
- Funciona con datos generados numéricamente, con datos experimentales y con funciones analíticas.
- Se adapta perfectamente a la Computación Paralela.

Desarrollar un Algoritmo Genético para resolver un problema específico implica escribir algún programa de computadora en algún lenguaje de programación (Ej. C, C++, Java, Python) o seudocódigo(Ej. Matlab) y luego ejecutarlo para obtener los resultados. Esto a su vez supone que el programa debe ser especifico para el tipo de problema que se esta abordando y no puede ser aplicado a otro problema.

El mundo de la programación de computadoras ha lidiado con este problema de particularización en aplicaciones (Riehle, 2000) con una solución muy ventajosa: un entorno de trabajo computacional, (software framework en ingles).

Un entorno de trabajo computacional es una estructura de software de alto grado de abstracción que provee funcionalidad genérica para que sea modificada por el usuario con el propósito de desarrollar una aplicación de software concreta.

En la actualidad existen pocos frameworks para desarrollar Algoritmos Genéticos, algunos escasamente documentados y otros muy documentados pero con un nivel de complejidad más apto para estudios de postgrado. Otros están enfocados a toda clase de problemas incluyendo

la Programación Genética, que representa una sub-área muy especifica dentro de los Algoritmos Genéticos y es poco práctica para el resto de especialidades de la Ingeniería. La mayoría esta documentado en idioma ingles, aunque esto no debería representar un inconveniente propiamente seria deseable la existencia de un entorno de trabajo computacional escrito en español.

En la Universidad de Carabobo existen pocos trabajos en el área de los Algoritmos Genéticos y todos ellos están desarrollados de forma ad-hoc, programados de principio a fin para resolver un problema especifico lo que amerita un tiempo considerable en el diseño del algoritmo cuando ese tiempo debería ser utilizado en la obtención y análisis de resultados provenientes de dicho algoritmo.

1.1.1. Formulación del Problema

Este planteamiento lleva a la pregunta ¿Es posible elaborar un entorno de trabajo computacional que permita el desarrollo de Algoritmos Genéticos, escrito en un lenguaje de alto nivel y con una estructura que permita su uso por parte de estudiantes de pregrado de Ingeniería y carreras afines?

1.2. Objetivos

1.2.1. Objetivo General

Elaborar un entorno de trabajo computacional que permita el desarrollo de algoritmos genéticos, escrito en un lenguaje de alto nivel y con una estructura que permita su uso por parte de estudiantes de pregrado de Ingeniería y carreras afines.

1.2.2. Objetivos Específicos

- Determinar qué conceptos relacionados con los Algoritmos Genéticos deben ser incluidos dentro del entorno de trabajo computacional.
- Evaluar las características de la plataforma Java en función de las ventajas que ofrece y el nivel de complejidad que puede plantear a estudiantes de pre grado de ingeniería, todo ello con la finalidad de ser el lenguaje de programación utilizado para la escritura de la librería de clases.
- Diseñar dos conjuntos de elementos de programación: primero, diseñar las clases, interfaces, métodos y constantes que integran el núcleo del entorno, y luego, diseñar sub implementaciones que agilicen el tiempo de desarrollo de un algoritmo genético. Ambos conjuntos deben estar claramente diferenciados, es decir, separados en paquetes. Cada uno de estos elementos debe estar documentado en formato electrónico así como en texto.
- Desarrollar un aplicación de ingeniería basada en una implementación que sirva de ejemplo para demostrar el desempeño del entorno de trabajo computacional.

1.3. Justificación

Un entorno de trabajo computacional ofrece la ventaja de programar a partir de una serie de clases pre-escritas lo que resulta en una ganancia de tiempo de desarrollo. Para el caso de Algoritmos Genéticos implica dedicar más tiempo al análisis de resultados que a los detalles del algoritmo en si. Además crea un marco estándar a partir del cual se pueden comparar distintas implementaciones de distintos problemas de Algoritmos Genéticos.

El **valor practico** del presente trabajo es calculable en la medida del uso que tenga por parte de la comunidad de usuarios a los que esta orientado. La carencia de un entorno de trabajo computacional para desarrollar Algoritmos Genéticos orientado a estudiantes de pregrado escrito en español es una buena razón para llevar a cabo este Trabajo sin embargo, lo más importante es la aplicabilidad del entorno como herramienta útil en el desarrollo de Algoritmos Genéticos para resolver problemas de ingeniería.

El **valor teórico** se distingue al ofrecer a la comunidad de estudiantes de pregrado de ingeniería y carreras afines un punto de partida en el área de Algoritmos Genéticos dado que no se hace énfasis a ningún problema especifico sino que los conocimientos se resumen en un compendio básico sobre el tema. Puede ser una buena guía de referencia inicial cuando se desee abordar una investigación más especifica sobre un problema concreto de ingeniería.

1.4. Alcance y Limitaciones

1.4.1. Alcance

El presente trabajo establece utilizar un compendio de conocimientos de la Computación Evolutiva, específicamente del área de los Algoritmos Genéticos, y adaptarlos a un entorno de trabajo computacional escrito en lenguaje de programación Java en su 7ma versión.

Este compendio considera incluir la capacidad de diseñar un individuo de forma específica, definiendo la estructura de su cromosoma, cruce, mutación y generación de poblaciones aleatorias. Además permite diseñar una función objetivo positiva especifica para el individuo en cuestión, de forma que pueda ser de maximizar o minimizar.

Se considera abarcar varios tipos de criterios de selección para determinar qué individuos tendrán posibilidad de cruzarse, y varios tipos de selección post cruce para determinar qué individuos pasarán a la siguiente generación. Considera las probabilidades de cruce y mutación, así como la opción de una regla de elitismo que pasa a al individuo más apto a la siguiente generación.

Se considera documentar, tanto de forma escrita (este Trabajo) como en formato HTML (*javadocs*) todas las clases, interfaces, métodos y constantes.

Se establece realizar la construcción del framework utilizando el Entorno de Desarrollo Integrado (IDE por sus siglas en ingles) Eclipse versión 3.8 y el lenguaje Java en su versión 6 (compilación 1.6.0 6b27). Dada la capacidad de la multiplataforma Java, el entorno de trabajo computacional puede ser utilizado en sistemas operativos Windows™, Linux y Mac™.

1.4.2. Limitaciones

- No se consideran todos los criterios de selección y selección post-cruce conocidos en la literatura de Algoritmos Genéticos.
- No se considera el área específica de Algoritmos Genéticos conocida como Programación Genética.
- No se considerada el uso de cluster de computación ni supercomputadoras, esta diseñado para computadoras personales convencionales que soporten el lenguaje Java.
- Aunque no se puede determinar qué tipo de computadoras personales pueden correr con éxito un Algoritmo Genético implementado, se considera, al menos, el uso de computadoras personales con procesadores de doble núcleo con más de 2 GB de memoria RAM, fabricadas aproximadamente desde el año 2008 en adelante.

Capítulo 2.

Marco de Referencia

2.1. Antecedentes

Meffert Klaus (2012) desde el año 2002 ha mantenido un entorno de trabajo computacional para Algoritmos Genéticos llamado *JGAP - Java Genetic Algorithms and Genetic Programming Package (Paquete de Algoritmos Genéticos y Programación Genética en Java)* escrito en lenguaje Java bajo las licencias GNU General Public License (GPL) versión 2.1 y Mozilla Public License (MPL) ver 2.0 Su arquitectura esta conformada por más de 500 clases, la mayoría muy bien documentada en idioma ingles. El entorno incluye una amplia y profunda base de conocimiento en Algoritmos Genéticos incluyendo Programación Genética. Su arquitectura modular es compleja y extensa, sin embargo es flexible y adaptable. A pesar de estar bien documentada la curva de aprendizaje puede crecer con lentitud ya que considera muchos elementos para la configuración de un Algoritmo Genético. La arquitectura del entorno de trabajo computacional del presente Trabajo se sustento en una simplificación de la estructura de JGAP.

Melanie Mitchell (1998) del Massachusetts Institute of Technology es autora del libro *An Introduction to Genetic Algorithms (Una Introducción a los Algoritmos Genéticos)* que sirve como buen soporte teórico ya que se trata de un resumen de muchos trabajos importantes realizados

en materia de Algoritmos Genéticos, también explica de forma clara muchos conceptos básicos relacionados al tema.

David Goldberg (1989) alumno del John Holland, el padre de los Algoritmos Genéticos. Su libro *Genetic Algorithms in Search, Optimization, and Machine Learning (Algoritmos Genéticos en Búsqueda, Optimización y Aprendizaje de Máquinas)* es la referencia obligada en materia de Algoritmos Genéticos, en él se encuentran los principios básicos de selección, cruce y mutación así como algunos ejemplos concretos escritos en lenguaje de programación Pascal.

Greg Paperin (2004) de la University College London desarrolló un entorno de trabajo computacional escrito en Java llamado *JAGA - Java API for Genetic Algorithms (API de Java para Algoritmos Genéticos)*. Esta amparado por la licencia GNU General Public License (GPL) versión 2.1. Está conformado por unas 100 clases por lo que es significativamente menos extenso que JGAP, además el mantenimiento del proyecto desde 2004 ha sido muy escaso y su documentación esta incompleta. A pesar de lo anterior, su estructura también se utilizó como referencia para el diseño de entorno de trabajo computacional del presente Trabajo.

2.2. Marco Teórico

2.2.1. Algoritmos Genéticos

Los Algoritmos Genéticos son técnicas de búsqueda y optimización basadas en el principio de la genética y la selección natural (Goldberg, 1989; Haupt y Haupt, 2004). Son algoritmos estocásticos cuyo método de búsqueda modela algunos fenómenos naturales: la herencia genética y la batalla Darwiniana del más apto (Michalewicz, 1992).

La **invención** (Mitchell, 1998) de los Algoritmos Genéticos se debe a John Holland junto con sus alumnos y colegas de la Universidad de Michigan durante los años 60 y 70. En contraste con las Estrategias Evolutivas y la Programación Evolutiva, el objetivo original de Holland no era diseñar algoritmos para resolver problemas específicos, sino estudiar formalmente el fenómeno de la adaptación que ocurre en la naturaleza y desarrollar formas por medio de las cuales mecanismos de adaptación natural pudieran ser importados a sistemas de computadoras. En el libro de Holland de 1975, *Adaptation in Natural and Artificial Systems (Adaptación en Sistemas Naturales y Artificiales)* se presenta el Algoritmo Genético como una forma de abstracción de la evolución biológica y ofrece un marco teórico para la adaptación bajo el Algoritmo Genético.

El **método** (Holland, 1975) utilizado por Holland consiste en moverse (evolucionar) desde un población de individuos a una nueva población usando algún tipo de "selección natural" junto con operadores inspirados en la Genética como cruce y mutación. Cada individuo esta representado por un "cromosoma", por ejemplo una cadenas de unos y ceros que consiste de "genes" (bits). El valor de un gen es conocido como "alelo" (un 0 o un 1). La selección natural ocurre en la forma de un operador de selección que escoge individuos de la población para ser cruzados. Esta selección utiliza algún criterio basado en la idea Darwiniana que los individuos más aptos producirán más descendientes que los menos aptos. El cruce intercambia sub partes de dos cromosomas, imitando primitivamente la recombinación biológica que ocurre entre dos cromosomas de distintos organismos. La mutación aleatoriamente cambia el valor de un alelo ubicado en alguna posición.

Goldberg (1989) utiliza una **terminología** de Algoritmos Genéticos que hace distinción entre la naturaleza y los sistemas artificiales (Ej. cromosoma vs cadena) sin embargo en trabajos posteriores (Coello, 2010; Haupt y Haupt, 2004; Maneiro, 2001; Michalewicz, 1992; Mitchell, 1998) y el Presente se maneja el mismo lenguaje de la biología pero con significados acordes a la matemática y la computación.

Un **individuo** es un simple miembro de una población que consiste de un *cromosoma* y el valor de su respectiva *aptitud*. Un individuo representa una solución a un problema y la forma de medir que tan "apta", o sea que tan buena, es dicha solución se consigue con el cálculo de su aptitud. Un conjunto de individuos que interactúan entre si se conoce como una **población**.

Un **cromosoma** es una cadena de genes y un **gen** es la codificación de un parámetro o variable, dicha codificación pertenece a un alfabeto específico. Por ejemplo, un cromosoma de dos genes donde cada gen es una cadena binaria (ceros y unos) de 2 bits, esto significa que si se tiene el cromosoma "0110" la primera variable corresponde a la segmento "01" y la segunda variable a "10".

Un **alelo** es el valor de un gen; en el ejemplo anterior, para el primer gen, se tenía el alelo "01" de los cuatro alelos posibles que puede tener dicho gen ("00", "11", "01", "10"). Un **locus** es la posición de un gen; en el ejemplo se tenían dos locus que corresponden al primer y segundo gen respectivamente.

Se denomina **aptitud** al valor que se asigna a cada individuo y que indica qué tan bueno es este con respecto a los demás. La aptitud se calcula a través de una *función de aptitud* que generalmente es una función positiva de maximizar.

Un **operador genético** es una función crea o modifica un nuevo cromosoma. El **cruce** es un operador genético que forma un nuevo cromosoma, llamado hijo, intercambiando segmentos de dos cromosomas padres. La **mutación** es un operador genético que forma un nuevo cromosoma por medio de alteraciones (usualmente pequeñas) en la estructura de otro cromosoma.

Se llama **elitismo** al mecanismo utilizado en algunos Algoritmos Genéticos para asegurar que los individuos más aptos de una población pasen a la siguiente generación sin ser alterados por ningún operador genético. Asegura que la aptitud máxima de la población nunca se

reducirá de una generación a la siguiente. Sin embargo, no necesariamente mejora la posibilidad de localizar el óptimo global de una función.

Una **generación** es una iteración de un Algoritmo Genético. Consiste en crear una población nueva a partir de una anterior, aplicando los operadores genéticos, mecanismos y reglas.

La **selección** es el proceso de escoger individuos aptos para el cruce bajo algún criterio específico. La selección no garantiza que el individuo se cruce pero le da una oportunidad basada en una probabilidad (probabilidad de cruce). Existen muchos criterios para la selección pero generalmente se basan en el principio de Darwin: individuos con mayor aptitud tendrán más oportunidades de cruzarse.

Métodos de Selección

A continuación se mencionan algunos de los métodos de selección más utilizados en la bibliografía de Algoritmos Genéticos:

La Ruleta (Goldberg, 1989, p. 11) es un tipo de selección proporcional que consiste en asignar una "rebanada" de una rueda de ruleta de casino a un individuo, dicha "rebanada" es proporcional a la aptitud de este individuo. La ruleta se hace girar (generando un numero aleatorio) N veces. El valor de N representa el tamaño de la población. En cada giro, el individuo asignado a la rebanada seleccionada es elegido como un padre.

Muestreo Estocástico Universal (Baker, 1987) también es un tipo de selección proporcional que utiliza la misma ruleta dividida en espacios proporcionales a la aptitud como en el método de La Ruleta pero sólo realiza un giro. La selección de los N individuos se logra asignando puntos equidistantes a partir del numero aleatorio obtenido en el único giro.

Truncamiento (Haupt y Haupt, 2004, p. 250) se seleccionan los individuos cuya aptitud se encuentre por encima de un valor especifico (punto de truncamiento), esto implica realizar ajustes en el tamaño de la población para mantenerla constante.

Por Jerarquías (Goldberg, 1989, p. 124) no es una técnica de selección, sino una variación que se puede hacer a cualquier método de selección proporcional como el *de la ruleta* o el *estocástico universal*. Consiste en ordenar los individuos en base a su aptitud y asignarles jerarquías, esto es, a mayor aptitud, mayor valor de la función de jerarquía. El valor esperado puede ser tanto una función lineal como no lineal, para un caso lineal usualmente se utiliza la ecuación 2.1 planteada por James Baker (1985).

$$Valesp(i,t) = Min + (Max - Min)\frac{jerarquia(i,t) - 1}{N - 1} \qquad (2.1)$$

Donde $E(i,t)$ es el valor esperado de un individuo i en una generación t; Min y Max son valores arbitrarios con $Max > Min$; $jerarquia(i,t)$ determina la posición del individuo i en la generación t; y N es el tamaño de la población.

Por Jerarquías "Elitista" (Mitchell, 1998, p. 127) como el método anterior ordena la población, pero en lugar de utilizar un Valor Esperado, se seleccionan los primeros M individuos y se descartan los restantes.

Equiprobable (Maneiro, 2001, p. 94) consiste en asignar una probabilidad igual a cada individuo, por tanto no depende de la aptitud. La probabilidad viene dada por la ecuación 2.2.

$$p(i) = \frac{1}{N} \qquad (2.2)$$

Donde $p(i)$ es la probabilidad del individuo i y N el tamaño de la población.

Torneo (Mitchell, 1998, p. 127) toma dos individuos de la población escogidos al azar. Se genera un numero aleatorio r entre 0 y 1. Si $r < k$ (donde k es un parámetro, por ejemplo 0,75) se selecciona el individuo de mayor aptitud, de lo contrario se selecciona el de menor aptitud.

Escalamiento Sigma de Tanese (Coello, 2010, p. 120; Mitchell, 1998, p. 125) presentado por Reiko Tanese (1989), al igual que la técnica *por jerarquías* no es una técnica de selección, sino una variación que se puede hacer a cualquier método de selección proporcional como el *de la ruleta* o el *estocástico universal*. Consiste en modificar el cálculo del valor esperado, en vez de utilizar una función proporcional basada solo en la función de aptitud, emplea una ecuación (2.3) que además de la variable anterior, también tiene como parámetros la media y la desviación estándar de la población.

$$Valesp(i,t) = \begin{cases} 1 + \dfrac{f(i) - \bar{f}(t)}{2\sigma(t)} & \text{si } \sigma(t) \neq 0 \\ 1 & \text{si } \sigma(t) = 0 \end{cases} \tag{2.3}$$

Donde $Valesp(i,t)$ es el valor esperado de un individuo i en una generación t; $f(i)$ representa la función de aptitud del individuo i; y $\bar{f}(t)$ y $\sigma(t)$ son la media y la desviación estándar de las funciones de aptitudes de la población en la generación t respectivamente.

El proceso evolutivo de búsqueda presenta dos grandes inconvenientes: La diversidad de la población y la presión selectiva (Michalewicz, 1992). Estos dos factores están estrechamente relacionados: un incremento en la presión selectiva hace decrecer la diversidad de la población y viceversa. En otras palabras, una presión selectiva fuerte conduce a una convergencia prematura en el espacio de búsqueda; y una presión selectiva débil hace el proceso lento e ineficiente. Por tanto es imperativo lograr algún balance entre estos dos factores con una buena combinación de operadores genéticos (cruce, mutación, selección).

2.2.2. Programación Orientada a Objetos

Actualmente los dos paradigmas de programación de software más utilizados son *por procedimientos* (también conocido como *procedural* o *imperativo*) que es el enfoque clásico; y el *orientado a objetos*, que es más moderno. Las ventajas de este último sobre el primero están extensivamente demostradas en teoría y practica (Harrison, Samaraweera, Dobie, y Lewis, 1996) principalmente en lo concerniente a la reusabilidad del código.

La **programación orientada a objetos** (Post, 2006) es una metodología de programación donde el código se encapsula dentro de la definición de los diversos objetos (o clases). Los sistemas se desarrollan a partir de objetos (que se esperan que sean) reutilizables. El sistema se controla al manipular las propiedades de un objeto y al solicitar lo métodos del mismo.

Lo programación orientada a objetos introduce una serie de conceptos (Post, 2006) que amplían o superan conceptos ya conocidos:

Una **clase** es un descriptor para un conjunto de objetos con estructura, comportamiento y relaciones similares. Es decir, una clase es la descripción del modelo de la entidad de negocio. Un modelo de negocio puede tener una clase *Empleado*, en donde un empleado especifico es objeto de esa clase.

Un **objeto** es una instancia o ejemplo especifico de una clase. Por ejemplo, en una clase *Empleado*, un empleado individual(Sr. Pedro Pérez) seria un objeto.

Un **método** es una función u operación que puede efectuar una clase. Por ejemplo, una clase *Empleado* puede tener un método *agregarNuevo()* que se llama cuando se agrega un objeto *Empleado* nuevo.

Una **variable de instancia** define las propiedades o características de una clase y por tanto los atributos de cada objeto de esa clase. Por ejemplo, una clase *Empleado* puede tener entre sus variables de instancia *nombre*, *cedula* y *cargo*.

Una **variable local** es definida dentro de un método o subrutina y su ámbito acaba cuando acaba dicho método o subrutina.

Entre las principales **características** que describen a la programación orientada a objetos se tienen:

La abstracción es una estrategia de resolución de problemas en la cual el programador se concentra en resolver una parte del problema ignorando completamente los detalles sobre cómo se resuelven el resto de las partes. En este proceso de abstracción se considera que el resto de las partes ya han sido resueltas y por lo tanto pueden servir de apoyo para resolver la parte que recibe la atención.

El encapsulado consiste en definir atributos y métodos dentro de una clase común. Por ejemplo, se podrían juntar todas las características y opciones de la clase Empleado.

La modularidad es la capacidad de fraccionar el código en subrutinas relacionadas con un propósito común.

El polimorfismo. En una jerarquía de clases, cada clase nueva hereda los métodos de las clases anteriores. Mediante el *polimorfismo*, se invalidan esas definiciones y se asigna un método nuevo(con el mismo nombre) a la clase nueva.

La herencia es la capacidad de definir clases nuevas derivadas de clases de nivel superior. Las clases nuevas heredan todas las propiedades y métodos anteriores, de modo que el programador solo necesita definir las propiedades y los métodos nuevos.

2.2.3. Java

Java son dos cosas: un lenguaje de programación y una plataforma (Oracle Corporation, 1995, 2012). El **lenguaje Java** es el lenguaje orientado a objetos más utilizado en el mundo y el segundo más utilizado en general después del *Lenguaje C* (TIOBE Software BV, 2012), posee una comunidad de desarrolladores que supera los 9 millones de personas y su ambiente de ejecución esta instalado en más de 1.100 millones de computadoras personales y 3.000 millones de teléfonos celulares (Oracle Corporation, 2012a).

Java es un lenguaje de alto nivel caracterizado por ser:

- simple
- orientado a objetos
- distribuido
- multihilos
- dinámico
- arquitecturalmente neutro
- portátil
- de alto rendimiento
- robusto
- seguro

Cada una de estas características son detalladamente explicadas en el documento electrónico *The Java Language Environment (El Ambiente del Lenguaje Java)* de James Gosling y Henry McGilton (1996).

En el lenguaje de programación Java, todo el código fuente es escrito primero en un archivo de texto que termina con la extensión *.java*, luego los archivos de código fuente se compilan en archivos *.class* por el compilador *javac*. Un archivo *.class* no contiene código que es comprendido por el procesador, sino que en su lugar contiene *bytecodes*, que es el lenguaje de máquina de la Maquina Virtual de Java, JVM[1]. A continuación, la JVM traduce el bytecodes a lenguaje de maquina (ceros y unos) y ejecuta la aplicación (ver la figura 2.1).

La Maquina Virtual de Java es una simulación de un computador que es capaz de ejecutar el *bytecode* y esta disponible para diferentes

[1]Siglas en ingles para *java virtual machine*.

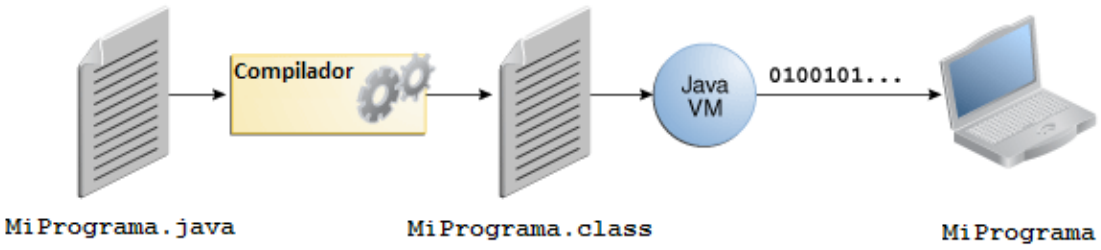


Figura 2.1.: Funcionamiento de Java (elaboración propia).

sistemas operativos. Esto significa que un mismo archivo *.class* puede ser ejecutado en Microsoft Windows™, Solaris OS™, Linux, Mac OS™ y Android™ (ver la figura 2.2).

Una plataforma es el ambiente de software en el que se ejecutan programas. Ya se han mencionado algunas de las plataformas más populares como Microsoft Windows™, Solaris OS™, Linux, Mac OS™ y Android™. La mayoría de las plataformas pueden ser descritas como una combinación de sistema operativo y hardware subyacente. La **plataforma Java** difiere de la mayoría en que es una plataforma de software exclusivamente que se ejecuta encima de otras plataformas hardware-software. Esto trae como ventaja una gran capacidad de portabilidad ya que puede desempeñarse sobre muchos tipos de hardware.

La plataforma Java esta compuesta por dos componentes:

- La máquina virtual de java (JVM)
- La interfaz de programación de aplicaciones (API[2])

Ya se ha mencionado la JVM como la base de la plataforma, la cual esta instalada en más de 4 millardos de equipos entre computadoras, celulares y electrodomésticos.

La API es una gran colección de componentes de software prefabricados que ofrecen muchas funciones útiles. Se agrupan en bibliotecas de clases e interfaces relacionadas; estas bibliotecas son conocidos como paquetes (ver la figura 2.3).

[2]Siglas en ingles para *application programming interface*.

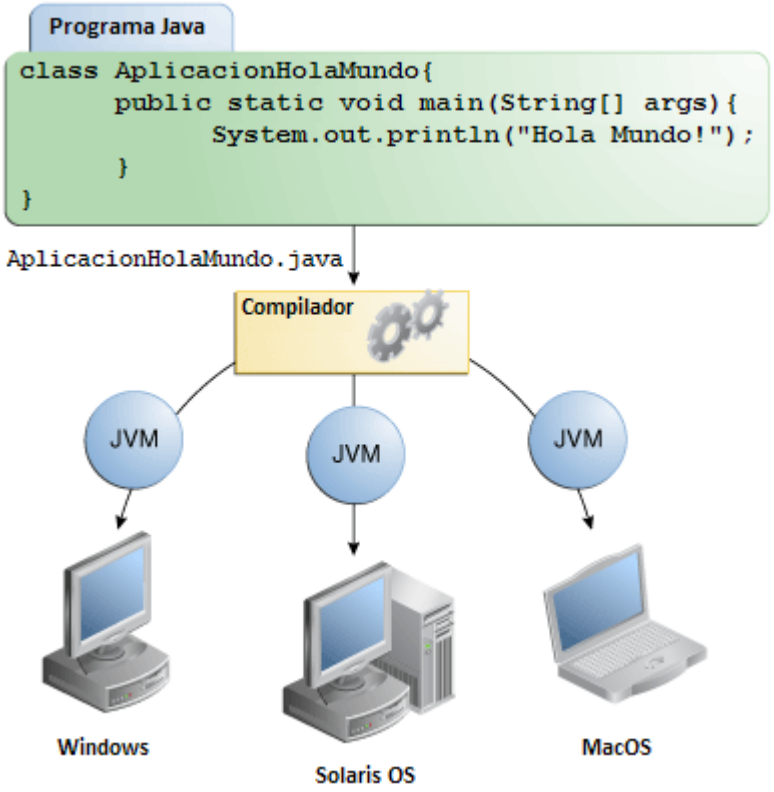


Figura 2.2.: Con la JVM, la misma aplicación se puede ejecutar en diferentes plataformas (elaboración propia).

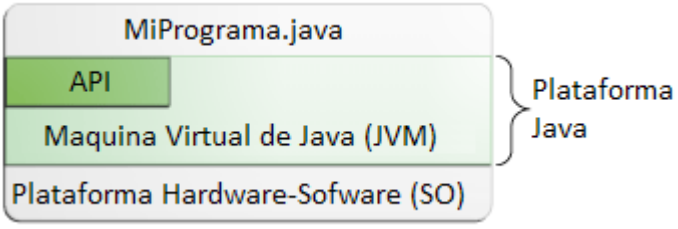


Figura 2.3.: La API y la JVM aíslan el programa del hardware subyacente (elaboración propia).

Como un entorno independiente de la plataforma, la plataforma Java puede ser un poco más lenta que el código nativo, sin embargo, los avances en las tecnologías de compilador y la máquina virtual están acercando el desempeño con el del código nativo sin representar una amenaza para la portabilidad.

2.2.4. Entorno de Trabajo Computacional

Mejor conocido por su nombre en ingles, *software framework*, o simplemente *framework*, un **entorno de trabajo computacional** (MAXXESS Systems, Inc., 2012) es una plataforma de software universal y reutilizable empleado para desarrollar aplicaciones, productos y soluciones. Pueden incluir programas de apoyo, compiladores, librerías de código, interfaces de programación de aplicaciones (API), sets de herramientas que en conjunto reúnen los componentes necesarios para desarrollar un proyecto de software.

Su diseño facilita el proceso de desarrollo permitiendo a diseñadores (de software) y programadores invertir más tiempo en el análisis que en los detalles tediosos del código. Permite a los programadores pasar menos tiempo "escribiendo código" y depurando errores y más tiempo para actividades de mayor valor agregado enfocadas al problema especifico.

Un **framework orientado a objetos** (Riehle, 2000) es un diseño reutilizable junto con una implementación. El diseño representa un modelo abstracto de la aplicación y la implementación define como este modelo debe ser ejecutado, al menos parcialmente. Un buen diseño e implementación de un framework es el resultado de un profundo entendimiento del dominio de la aplicación y debe dejar suficiente espacio para la personalización con el fin de resolver un problema particular.

Capítulo 3.

Marco Metodológico

En los Capítulos 1 y 2 se ha venido definiendo el **diseño de investigación** que sirve de base en el proceso de planeación del trabajo que se quiere abordar (desarrollo de un framework para Algoritmos Genéticos) en la perspectiva del conocimiento científico (Méndez, 2001, p. 57, 60, 65). Este diseño tiene tres componentes claramente definidas:

1. Elementos de contenido y alcance: *selección y definición del tema, planteamiento y formulación del problema, objetivos de la investigación, justificación* y *marco de referencia*. Estos elementos fueron desarrollados en el capítulo 1 y 2.

2. Elementos de apoyo metodológico: *tipo de investigación, método de investigación, fuentes y técnicas para la recolección de información,* y *tratamiento de la información*, son definidos en el presente capítulo. Los elementos *tabla de contenido* y *bibliografía* se encuentra al principio y al final del presente Trabajo respectivamente.

3. Elementos de soporte administrativo: *cronograma de trabajo* explicado en forma de diagrama de Gantt en la sección 8 del Plan de Trabajo previo a este Trabajo de Grado. Finalmente el elemento *presupuesto de la investigación* no es considerado debido a la insignificancia del impacto en el desarrollo de la investigación.

3.1. Tipo de Investigación

Los estudios exploratorios "se efectúan, normalmente, cuando el objetivo es examinar un tema o problema de investigación poco estudiado o que no ha sido abordado antes. Es decir cuando la revisión de la literatura revela que únicamente hay guías no investigadas e ideas vagamente relacionadas con el problema de estudio (. . .) Sirven para familiarizarnos con fenómenos relativamente desconocidos" (Sampieri, Collado, y Lucio, 1998).

"La investigación descriptiva tiene como objetivos describir el estado, las características, factores y procedimientos presentes en los fenómenos y hechos que ocurren en forma natural, sin explicar las relaciones que se identifiquen. Su alcance no permite la comprobación de hipótesis ni la predicción de resultados" (Lerna, 2004).

Según las definiciones descritas en los dos últimos párrafos, el presente Trabajo sigue una **investigación principalmente exploratoria con algunos elementos descriptivos**. Es exploratoria por la escasa disponibilidad de trabajos relacionados con framework para Algoritmos Genéticos, lo que conduce a observar la estructura de unos pocos entornos computacionales disponibles pero no desarrollados en el marco de un estudio formal.

Los elementos de investigación descriptiva se encuentran en la búsqueda de los extensos conocimientos sobre Algoritmos Genéticos que se han desarrollado en las ultimas décadas. Resumir y describir las formas más comunes que puede adquirir un Algoritmo Genético es algo relativamente sencillo dada la amplia bibliografía sobre el tema.

3.2. Método de Investigación

El método de investigación es "el procedimiento riguroso, formulado de una manera lógica, que el investigador debe seguir en la adquisición de conocimiento" (Méndez, 2001).

El método más conveniente a los intereses de esta investigación es la **observación**, definida como "el proceso mediante el cual se perciben deliberadamente ciertos rasgos existentes en la realidad por medio de un esquema conceptual previo y con base en ciertos propósitos definidos generalmente por una conjetura que se quiere investigar" (Ladrón, 1978).

En este Trabajo **se desea observar** la estructura y composición de frameworks sobre Algoritmos Genéticos como JAGA y JGAP, este último principalmente. Como ya se ha mencionado antes no se conocen trabajos académicos sobre este tema, lo que deja como una única vía la observación de las frameworks "informales" ya existentes.

3.3. Fuentes de Investigación

Las fuentes "son hechos o documentos a los que acude el investigador y que le permiten obtener información" (Méndez, 2001). Las técnicas para recolección de información "son los medios empleados para recolectar la información" (Méndez, 2001).

Este Trabajo se refiere exclusivamente a **fuentes secundarias de información**, las cuales suministran información básica disponible en bibliotecas e internet y se encuentran contenidas en libros, documentos electrónicos tanto personales como corporativos.

3.4. Tratamiento de la Información

El tratamiento de la información es "la determinación de los procedimientos para la tabulación y codificación de la información para el recuento, clasificación y ordenamiento de la información en tablas o cuadros" (Méndez, 2001).

Presentación de la información. El contenido de la investigación se presenta de forma escrita disponible tanto en papel impreso como en archivo electrónico PDF. Adicionalmente la documentación del entorno de trabajo computacional referente a la arquitectura del mismo se presenta en un formato electrónico (archivo HTML) estándar conocido como *javadocs* con características definidas por la herramienta de software *Javadoc Tool* (Oracle Corporation, 2012b). Como el entorno de trabajo computacional es una aplicación de software, lógicamente se presenta en formato electrónico definido como un archivo java de extensión *.class*.

Capítulo 4.

Estructura y Funcionamiento

El framework está escrito en el lenguaje de programación Java y está compuesto, en total, de 23 clases agrupadas en dos paquetes: el paquete `agapi`[1] con 9 clases, y el paquete `impl` con 14.

La palabra *agapi* es un acrónimo para *API de Algoritmos Genéticos*, y es el nombre del proyecto de software además del nombre del paquete principal. El proyecto está amparado por la licencia *GPL versión 3* de la Fundación del Software Libre. Para más información acerca del uso de la licencia, ver apéndice A.

Las clases del núcleo del framework, donde ocurre la ejecución de corridas de algoritmos genéticos, se encuentran en el paquete `agapi` mientras que en `impl` sólo hay implementaciones de las clases de `agapi` y su utilización es de carácter opcional.

El funcionamiento del framework se explica en la forma en que las clases del paquete `agapi` interactúan entre si: Todo comienza por la configuración de los parámetros de un proceso en la clase `Configuracion`, luego esta dispara el algoritmo genético con el método `iniciarProceso` que

[1] Los nombres de clases, métodos, y otros elementos del lenguaje de programación están escritos en fuente tipo `monoespaciado` similar a la de las máquinas de escribir antiguas. También se respecta la convención del lenguaje Java respecto al uso de minúsculas y mayúsculas en dichos elementos por lo que no debe ser raro ver el uso de una minúscula después de un punto y seguido o al inicio de un párrafo.

llama a `procesar`, este llama a `ejecutar` que llama a `generar` y es aquí donde se aplican los operadores genéticos y los métodos de selección. Además `generar` crea poblaciones (clase `Poblacion`) que contiene individuos (clase `Individuo`) que se cruzan (método `cruce`), mutan (método `mutacion`) y generan individuos con cromosomas aleatorios (método `aleatorio`).

4.1. Estructura

La estructura del entorno de trabajo computacional esta basada en el paradigma de la programación orientada a objetos, es decir, el código esta encapsulado en una serie de clases y objetos que se interrelacionan. La característica principal del framework es su capacidad de *abstracción*, enfocándose en los detalles básicos de la corrida de un algoritmo genético, sin prestar atención a detalles específicos de algún problema de algoritmos genéticos en particular. El framework está escrito en el lenguaje de programación Java, de uso universal tanto en el mundo académico como en el de los negocios.

El entorno consta de un total de 23 clases organizadas en dos paquetes, es decir en dos directorios:

- el paquete `agapi` con 9 clases,
- el paquete `impl` con 14 clases.

Los división de clases entre estos paquetes se basa en la función que tienen dichas clases dentro del framework. Unas clases, las ubicadas en `agapi`, son indispensables para cualquier ejecución de un problema y otras, las de `impl`, se ofrecen de forma opcional con el propósito de agilizar el proceso de desarrollo del código.

4.1.1. Paquete agapi

Dentro del paquete `agapi` se encuentran las clases del motor del framework, donde ocurre la ejecución de corridas de algoritmos genéticos, por lo tanto son indispensables para el desarrollo del cualquier problema. En este paquete se encuentran una clase abstracta e interfaces que dan el carácter *abstracto* al framework, y deben ser implementadas por clases concretas diseñadas por el usuario para la ejecución de un problema específico.

El paquete `agapi` esta conformado por 9 clases, interrelacionadas como se observa en el diagrama[2] de la figura 4.1. En el capítulo 5 se describe con detalle la función de cada clase, a continuación se hace una breve descripción de estas:

`Individuo` (abstracta) es la clase que el usuario debe heredar ya que en ella de encuentran métodos que describen un problema específico como `calcFA` donde se define la función de aptitud, y `cruce` y `mutacion` donde se determinan el *cruce* y la *mutación* respectivamente.

`Poblacion` consiste en un arreglo de objetos tipo `Individuo` que representa a una población de individuos. También aloja variables para el mejor y peor individuo, la media y desviación estándar de las funciones de aptitud de los individuos de dicha población.

`Generacion` es donde ocurren la evolución de una población a otra por medio de la acción de los operadores genéticos. Esta clase toma un objeto de tipo `Poblacion` y aplica los métodos `cruce` y `mutacion` de la clase `Individuo`. También aplica los métodos de las interfaces `Selector` y `SelectorPostCruce`. Además, `Generacion` aloja las variables `probabilidadCruce` y `probabilidadMutacion`

[2]Los diagramas de clases siguen los estándares del *Lenguaje de Modelado Unificado* (*UML*, *Unified Modeling Language*™) versión 2.0 (IBM Corporation, 2004; James Rumbaugh, 2004)

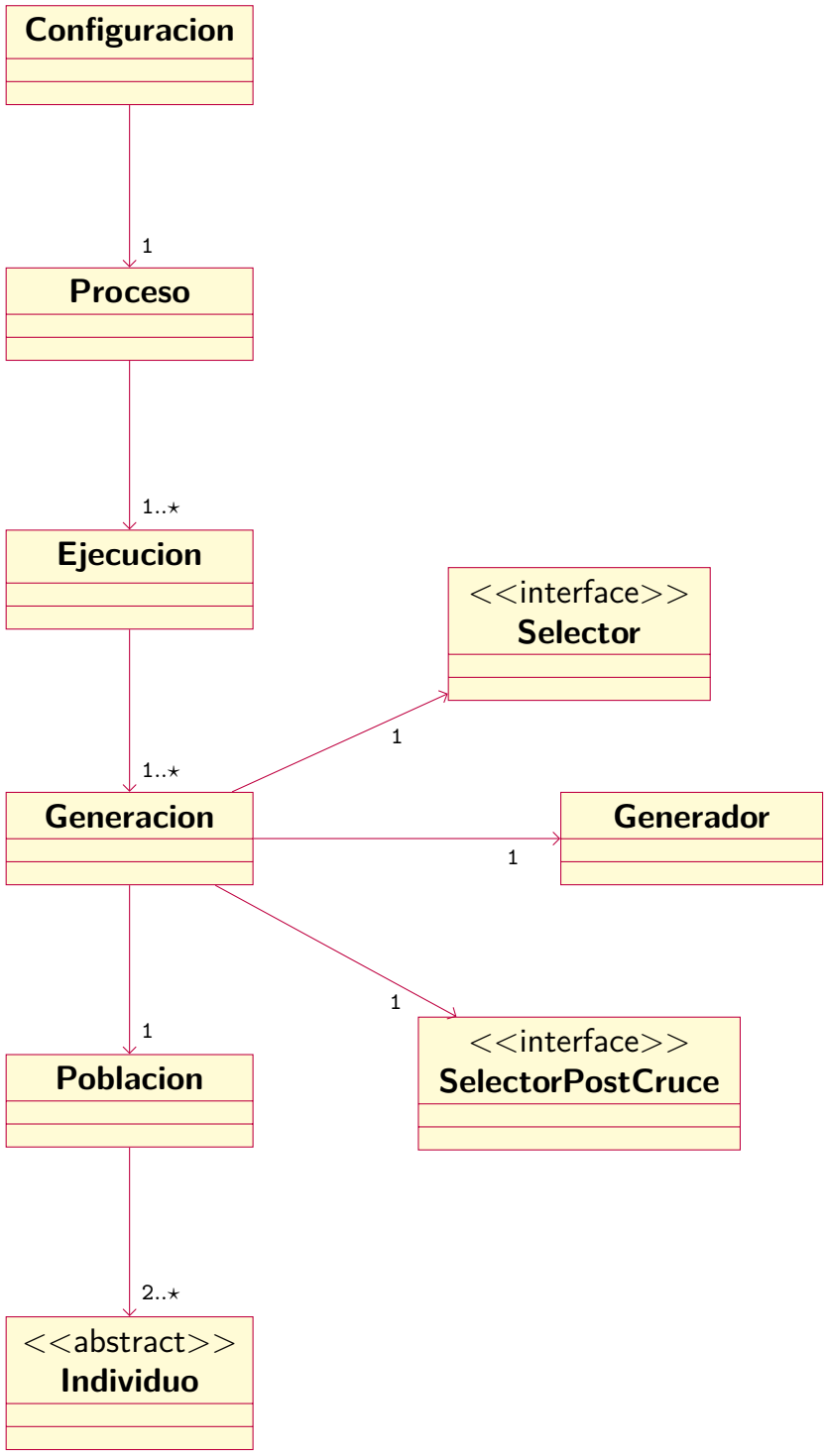


Figura 4.1.: Estructura del paquete agapi (elaboración propia).

correspondientes a los valores de las probabilidades de cruce y mutación.

`Ejecucion` es un proceso evolutivo completo, consiste en un arreglo de objetos de tipo `Generacion`. En esta clase se crea una *generación* i a partir de una *generación* $i-1$, esto se realiza tantas veces como la variable `numeroGeneraciones` lo indique.

`Proceso` es un conjunto de ejecuciones, es decir, de evoluciones independientes una de la otra. Esta formado por un arreglo de objetos tipo `Ejecucion`. El propósito de esta clase es simplemente facilitar al investigador la recolección de datos, esto se logra corriendo tantas ejecuciones como lo indique la variable `numeroEjecuciones`.

`Configuracion` es una especie de tablero de control desde el cual se puede configurar todos los parámetros de un *proceso* haciendo llamados a métodos estáticos ubicados en distintas clases del entorno. Para la corrida de cualquier aplicación es necesario crear un objeto de esta clase y hacer llamados a todos sus métodos estáticos.

`Generador` es una clase auxiliar de `Generacion` y su única función es alojar el método `generar` que contiene el código que hace evolucionar una población. `generar` es llamado desde el método homónimo ubicado en `Generacion`.

`Selector` (interfaz) se encarga de la selección de individuos que tendrán posibilidad de cruzarse. Contiene dos métodos abstractos: `seleccion` encargado de ejecutar la selección según el criterio establecido y `prepararSeleccion` que permite aplicar cálculos adicionales a toda la población que luego puedan ser utilizados por `seleccion`. Esta interfaz puede ser implementada por el usuario directamente en caso de no utilizar alguna de las implementaciones facilitadas por el framework en la clase `impl`.

`SelectorPostCruce` (interfaz) se encarga de determinar qué individuos, entre *padres* e *hijos*, pasarán a la siguiente generación después del

cruce y la mutación; esto se logra con la implementación del método `seleccionPostCruce`. Puede ser implementada por el usuario directamente en caso de no utilizar alguna de las implementaciones facilitadas por el framework en la clase `impl`.

4.1.2. Paquete impl

El paquete `impl` contiene clases concretas y abstractas que corresponden con implementaciones de las clases del paquete `agapi`. Aunque no son indispensables para la ejecución de un algoritmo, sirven como apoyo opcional al facilitar clases que describen varios tipos de *individuos* y *métodos de selección* pre configurados que pueden ahorrar tiempo de desarrollo al investigador.

El paquete `impl` está integrado por 14 clases que heredan directamente de las clases del paquete `agapi` como se observa en la figura 4.2. Esta compuesto por 2 implementaciones de la clase `Individuo`, además de una clase con el tipo de cruce PMX (`CrucePMX`) utilizada por uno de estos individuos; varias implementaciones de la interfaz `Selector` que hacen uso de otras clases que en total suman 9; y 2 implementaciones de la interfaz `SelectorPostCruce`. En el capítulo 5 se describe con detalle la función de cada clase, a continuación se hace una breve descripción de estas:

`IndividuoBinario` (clase abstracta) es una implementación de la clase `Individuo` con un cromosoma formado por una secuencia de ceros y unos. Implementa los métodos `cruce`, `mutacion` y `aleatorio` y deja abstracto `calcFA`. `cruce` consiste en una sustitución de los bits de la madre por los del padre. `mutacion` simplemente invierte el valor de un bit en una posición aleatoria. `calcFA` se deja sin cuerpo (abstracto) para que el usuario la implemente.

`IndividuoCombinatorio` (clase abstracta) al igual que la anterior, es una implementación de `Individuo` cuyo cromosoma es una se-

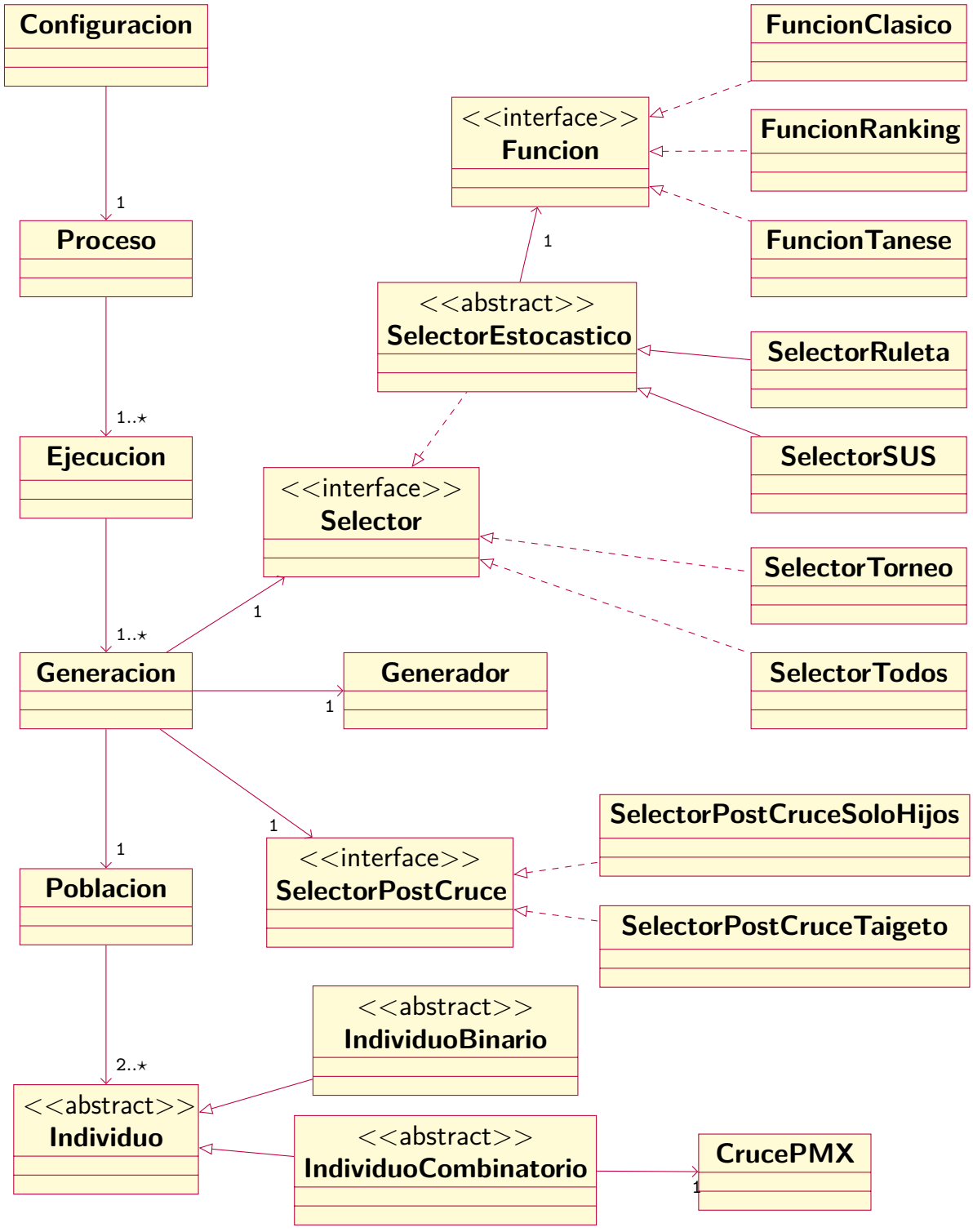


Figura 4.2.: Estructura del entorno de trabajo computacional (elaboración propia).

cuencia (arreglo) de números enteros de 1 a N (donde $N \in Z$ y $N > 0$) que luego es desordenada aleatoriamente. `cruce` utiliza el método de cruce PMX (Goldberg y Lingle, 1985) apoyado en la clase `CrucePMX` y, `mutacion` simplemente intercambia dos números en un mismo cromosoma de forma aletoria. `calcFA` se mantiene abstracto.

`CrucePMX` tiene como única función suministrar un método `cruce` de tipo PMX para la clase `IndividuoCombinatorio`. Esto se debe a la complejidad del proceso de cruce tipo PMX, lo que amerita toda una clase para su desarrollo.

`SelectorTorneo` implementa la interfaz `Selector` de forma que selecciona individuos usando una técnica *por torneo* de tipo *binaria* (compiten dos individuos) y *probabilística* (el ganador se determina por un numero aleatorio) (Goldberg y Deb, 1991).

`SelectorTodos` es una implementación de la interfaz `Selector` y su único propósito es ser utilizada en corridas de prueba ya que realmente no selecciona a ningun individuo, simplemente deja pasar a uno por uno garantizando al final que todos los individuos tenga la posibilidad de cruzarse.

`SelectorEstocastico` (clase abstracta) define un esquema de selección probabilísticos que simula una rueda de ruleta de casino con "rebanadas" en función de los valores esperados de los individuos de la población. Esta clase es heredada por `SelectorRuleta` y `SelectorSUS` que comparten dicho esquema.

`SelectorRuleta` implementa la interfaz `SelectorEstocastico` que aplica la técnica de *la ruleta* (Goldberg, 1989) que hace rodar la ruleta (generando un número aleatorio) N veces, donde N representa el número de individuos seleccionados con chance de reproducirse.

`SelectorSUS` implementa la interfaz `SelectorEstocastico` que aplica el *muestreo estocástico universal* (Baker, 1987) donde la ruleta se hace girar una vez (se genera solo un número aleatorio) utilizando

N apuntadores espaciados equidistantemente que determinan los individuos seleccionados con chance de reproducirse.

`Funcion` (clase abstracta) es utilizada por los *selectores estocásticos* (por ejemplo, `SelectorSUS`) para la definición de la función de valor esperado de los individuos.

`FuncionClasico` es una implementación de la interfaz `Funcion` que representa la función clásica de valor esperado utilizado por David Goldberg (1989). Dicha función no es más que la función de la aptitud de cada individuo de la población.

`FucionRanking` es una implementación de la interfaz `Funcion` que permite el cálculo de valores esperados en función de las *jerarquías* (*ranking*)de los individuos dentro de la población según la ecuación 2.1) propuesta por James Baker (1985).

`FuncionTanese` es una implementación de la interfaz `Funcion` que plantea la función de valor esperado con escalamiento sigma (ver ecuación 2.3) presentada por Reiko Tanese (1989).

`SelectorPostCruceSoloHijos` es una implementación de la interfaz `SelectorPostCruce` y está encargada de pasar a la siguiente generación sólo los dos hijos productos de un cruce, es decir, los padres son desechados independientemente de sus aptitudes.

`SelectorPostCruceTaigeto` como la anterior, es una implementación de la interfaz `SelectorPostCruce` y su función es pasar a la siguiente generación sólo a los mejores dos individuos entre padres e hijos.

4.2. Funcionamiento

Una corrida se inicia con la ejecución del método `iniciarProceso` de la clase `Configuracion`. Como se dijo anteriormente `Configuracion`

es un tablero de control general de los parámetros de un problema de algoritmos genéticos donde se pueden modificar desde valores como las probabilidades de cruce y mutación hasta los métodos de selección y el tipo de individuo a utilizar, todo ello por medio de llamados a métodos estáticos (exclusivamente) que al mismo tiempo llaman a métodos ubicados en distintas clases. Adicionalmente, `iniciarProceso`, hace un llamado al método `procesar` de `Proceso` lo que inicia un *proceso*.

Un *proceso* es una sucesión de *ejecuciones*, por ello la clase `Proceso` contiene un arreglo de objetos tipo `Ejecucion` desde los que se llama el método `ejecutar`. Se realizan tantas ejecuciones como lo indique la variable `numeroEjecuciones` configurable desde `Configuracion`.

Una *ejecución* es una sucesión de *generaciones* que son creadas una a partir de la otra, por ello, la clase `Ejecucion` contiene un arreglo de objetos tipo `Generacion`. El método `ejecutar` ubicado en `Ejecucion` crea una población inicial de individuos con cromosomas aleatorios utilizando el método `aleatoria` de `Poblacion`. Esta población inicial es introducida en el constructor de la primera generación del arreglo y, a partir de este momento, comienza un ciclo en el que una nueva generación (gen_i) es creada a partir de una generación anterior (gen_{i-1}) por medio del método `generar` de `Generacion` hasta alcanzar una cantidad de generaciones igual a `numeroGeneraciones`.

En el paso anterior (`ejecutar`) se logra una evolución completa de un problema específico lo que explica cómo la *ejecución* es la unidad principal de recolección de datos. Todo problema debe correr al menos una *ejecución* para llegar a una buena solución y producir todos los datos de interés como medias y desviaciones de las funciones de aptitudes, mejor y peor individuo, y tiempos de corridas.

El método `generar` de `Generacion` llamado desde el `ejecutar` de `Ejecucion` es el encargado de convertir una generación en una nueva, aplicando los operadores genéticos correspondientes. Para ello, `generar`

se apoya en la clase `Generador` pues facilita la organización del código al dedicar toda una clase al desarrollo de lo que ocurre en una generación, esto es, aplicar operadores genéticos, métodos de selección y selección post cruce, utilizar probabilidades de cruce y mutación, almacenar los valores de medias y desviaciones estándar de las funciones de aptitud, almacenar todas las características de los individuos implicados, almacenar al mejor y peor individuo de cada generación, etcétera.

En la clase `Generacion` se dan otras relaciones de composición: una `Generacion` básicamente tiene un `Selector`, un `SelectorPostCruce` y una `Poblacion`; todos elementos imprescindibles en una corrida con sus métodos llamados desde `generar`. `Selector` y `SelectorPostCruce` son interfaces utilizadas antes y después del cruce y la mutación, respectivamente. `Poblacion` contiene a los *individuos* y sus métodos correspondientes.

Cuando se produce una nueva generación también se produce una nueva población correspondiente a la primera. Este objeto de tipo `Poblacion` es construido individuo por individuo dentro del método `generar`, donde al mismo tiempo se van calculando los valores de la media y la desviación estándar y evaluando las aptitud de los individuos para determinar al mejor y al peor de ellos.

Adicionalmente en `generar` se hacen los llamados a los métodos `cruce` y `mutacion` (ubicados en `Individuo`) encargados de los procesos homónimos determinados por la implementación del usuario. Estos métodos son llamados luego de generar números aleatorios y compararlos con las probabilidades de cruce y mutación establecidas desde la clase `Configuracion`.

Cuando se realiza la primera generación se debe partir de una población inicial generada de forma aleatoria por el método `aleatoria` de `Poblacion` que realiza una construcción de individuos de forma iterativa llamando al método `fabricaIndividuo` (de `Individuo`). Luego, por

cada individuo recién construido, se generan cromosomas aleatorios con el método `aleatorio`, también de `Individuo`.

`fabricaIndividuo` es el único método que debe ser modificado en su estructura interna por el usuario ya que se encarga de crear objetos *concretos* de clases que heredan de `Individuo`.

Los métodos de `Individuo` son utilizados por otras clases, como `calcFA` que es llamado por los métodos `generar` en `Generador` y `aleatoria` en `Poblacion`, en ambos casos para el cálculo de la función objetivo de cada individuo y el cálculo de la media y la desviación estándar de una población. `cruce` y `mutacion` son utilizados en `generar` (`Generador`) y `aleatorio` en `aleatoria` de `Poblacion`.

Aunque ya se ha mencionado, es importante destacar que los métodos `calcFA`, `cruce`, `mutacion` y `aleatorio` son *abstractos* y por lo tanto no contienen ningún código dentro de ellos, su propósito es ser implementados.

Capítulo 5.

Documentación

Este capítulo es una adaptación de la documentación disponible en formato electrónico HTML conocida como *javadocs*. Dicha documentación se encuentra como comentarios de preámbulo en las clases, interfaces y métodos en el código java del framework. La documentación hace uso de macros HTML y javadocs siguiendo la convención de Oracle Corporation (1999).

Para adaptar el formato *javadocs* al presente texto se deben tomar al algunas consideraciones:

- Ligera modificación la redacción dado que no se pueden usar hipervínculos.
- Redundancia de definiciones de algunos conceptos. Esto se debe al carácter de manual de consulta rápida de esta documentación
- Traducción de algunos términos como Field (Constantes), Return (Retorna), Parameters (Parámetros), Throws (Arroja), Overrides (Sobrescribe), See Also (Ver también)

La **Javadoc Tool** (Herramienta Javadoc) es un programa que convierte los comentarios del código fuente java en formato HTML para ser presentado como página web siguiendo el diseño de la API de Java (Oracle Corporation, 1993, 2013).

Los nombres de clases, métodos, y otros elementos del lenguaje de programación están escritos en fuente tipo `monoespaciado` similar a la de las máquinas de escribir antiguas y respectando la convención del lenguaje Java respecto al uso de minúsculas y mayúsculas (Oracle Corporation, 1999).

Los figuras de diagramas de clases siguen los estándares del Lenguaje de Modelado Unificado (UML, Unified Modeling Language™) versión 2.0 (IBM Corporation, 2004; James Rumbaugh, 2004).

5.1. Paquete agapi

5.1.1. Clase Individuo (abstracta)

`public abstract class Individuo.`
`extends java.lang.Object.`
Es la clave del framework (ver figura 5.1). Esta clase se debe heredar para diseñar un individuo de forma especifica. Contiene cuatro métodos que deben ser implementados y que definen la estructura del problema a resolver:

1. `calcFA` define la función de aptitud a utilizar. Es **muy importante** que dicha función sea **positiva** y de **maximizar**, para lo cual se deben utilizar los artificios matemáticos necesarios.

2. `aleatorio` transforma el cromosoma de este individuo en un cromosoma aleatorio. Es utilizado por la clase `Poblacion` para generar una población aleatoria en el inicio de una ejecución.

3. `cruce` ejecuta un cruce entre este individuo (padre) y otro (madre). El cruce debe ser algún tipo de combinación entre el cromosoma del padre y el de la madre.

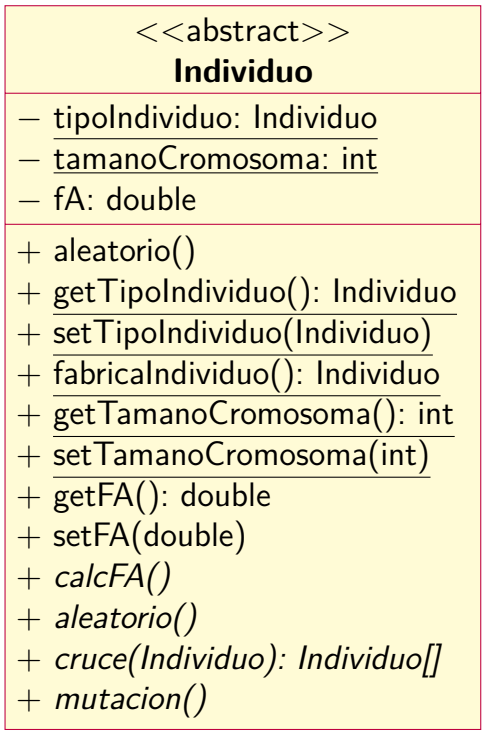


Figura 5.1.: Diagrama - Clase abstracta Individuo (elaboración propia).

4. `mutacion` modifica la estructura del cromosoma de este individuo. Esta modificación generalmente es pequeña, por ejemplo, cambiar el valor de un único gen.

Métodos

getTipoIndividuo

`public static Individuo getTipoIndividuo()`.
Devuelve el tipo de individuo del algoritmo. `tipoIndividuo` es una variable estática y es, además, un objeto que hereda de la clase `Individuo`.
Retorna: el tipo de individuo del algoritmo.

setTipoIndividuo

`public static void setTipoIndividuo(`
`Individuo tipoIndividuo).`
Establece el tipo de individuo específico a utilizar en el problema. Este método es llamado por `setTipoIndividuo` de `Configuracion` para la establecer el tipo de individuo concreto a utilizar.
Parámetros: `tipoIndividuo` - el tipo de individuo a utilizar en el problema.
Arroja: `NullPointerException` - si `tipo` es `null`.

fabricaIndividuo

`public static Individuo fabricaIndividuo().`
Devuelve un objeto del tipo correspondiente a `tipoIndividuo`. Esto permite utilizar individuos concretos dentro de un contexto abstracto, donde solo haya variables de tipo `Individuo` (clase abstracta).
Retorna: un individuo concreto (objeto) del tipo establecido en `tipoIndividuo`.

getTamanoCromosoma

`public static int getTamanoCromosoma().`
Devuelve el tamaño del cromosoma utilizado por este individuo. El parámetro `tamanoCromosoma` debe utilizarse con cuidado ya que en ciertos casos el valor del tamaño del cromosoma esta íntimamente relacionado con la función de aptitud del problema, tal es el caso de los cromosomas binarios en donde, generalmente, se debe usar un tamaño fijo de cromosoma que no debe ser modificado, de lo contrario se debe modificar también la función de aptitud.
Retorna: el tamaño del cromosoma del individuo.

setTamanoCromosoma

`public static void setTamanoCromosoma(int tamanoCromosoma)`.
Establece el tamaño del cromosoma a utilizar por este individuo. Este método es llamado por `setTamanoCromosoma` de `Configuracion` para la establecer el tamaño del cromosoma del individuo del algoritmo.

El parámetro `tamanoCromosoma` debe utilizarse con cuidado ya que en ciertos casos el valor del tamaño del cromosoma esta íntimamente relacionado con la función de aptitud del problema, tal es el caso de los cromosomas binarios en donde, generalmente, se debe usar un tamaño fijo de cromosoma que no debe ser modificado, de lo contrario se debe modificar también la función de aptitud.
Parámetros: `tamanoCromosoma` - el tamaño del cromosoma.
Arroja: `IllegalArgumentException` - si `tamanoCromosoma` es menor a 1.

getFA

`public double getFA()`.
Devuelve el valor de la función de aptitud del individuo.
Retorna: el valor de la función de aptitud del individuo.

setFA

`public void setFA(double fA)`.
Establece el valor de la función de aptitud del individuo.
Parámetros: `fA` - la función de aptitud a utilizar.

calcFA

`public abstract void calcFA()`.
Calcula el valor de la función de aptitud . Dicho cálculo es específico del problema a tratar y es una función cuya variable independiente es el cromosoma del individuo. Este cromosoma puede ser diseñado

por el usuario o proveniente de alguna clase pre configurada como `impl.IndividuoBinario` o `impl.IndividuoCombinatorio`. Es **muy importante** que dicha función sea **positiva** y de **maximizar**, para lo cual se deben utilizar los artificios matemáticos necesarios.
Retorna: el valor de la función de aptitud calculado.

aleatorio

`public abstract void aleatorio()`.
Transforma el cromosoma de un Individuo en un cromosoma aleatorio. Es utilizado por el método `aleatoria` de `Poblacion` cuando se debe generar una población aleatoria para el inicio de una *ejecución*.

cruce

`public abstract Individuo[] cruce(Individuo madre)`.
Ejecuta un cruce entre este individuo (padre) y otro (madre). El cruce debe ser algún tipo de combinación entre el cromosoma del padre y el de la madre.
Parámetros: `madre` - el individuo madre.
Retorna: Un par individuos (hijos).

mutacion

`public abstract void mutacion()`.
Modifica la estructura del cromosoma de este individuo. Esta modificación generalmente es pequeña, como cambiar el valor de un único gen.

5.1.2. Clase Poblacion

`public class Poblacion`
`extends java.lang.Object`.

Esta clase contiene variables que caracterizan a una población (ver figura 5.2). Una población es un conjunto de *individuos* organizados en un arreglo de tamaño igual a `tamanoPoblacion` más las variables media, desviación estándar, mejor y peor individuo:

`media` es la media de los valores dados por la función de aptitud de los individuos de la población.

`desviación` es la desviación estándar de los valores dados por la función de aptitud de los individuos de la población.

`mejorIndividuo` es el individuo con el mayor valor dado por la función de aptitud de la población.

`peorIndividuo` es el individuo con el menor valor dado por la función de aptitud de la población.

Constantes

ALEATORIA

`public static final int ALEATORIA`.
Especifica al constructor `Poblacion(int opcion)` crear un objeto con una población de individuos con cromosomas aleatorios.

Constructores

Poblacion

`public Poblacion()`.
Constructor por defecto.

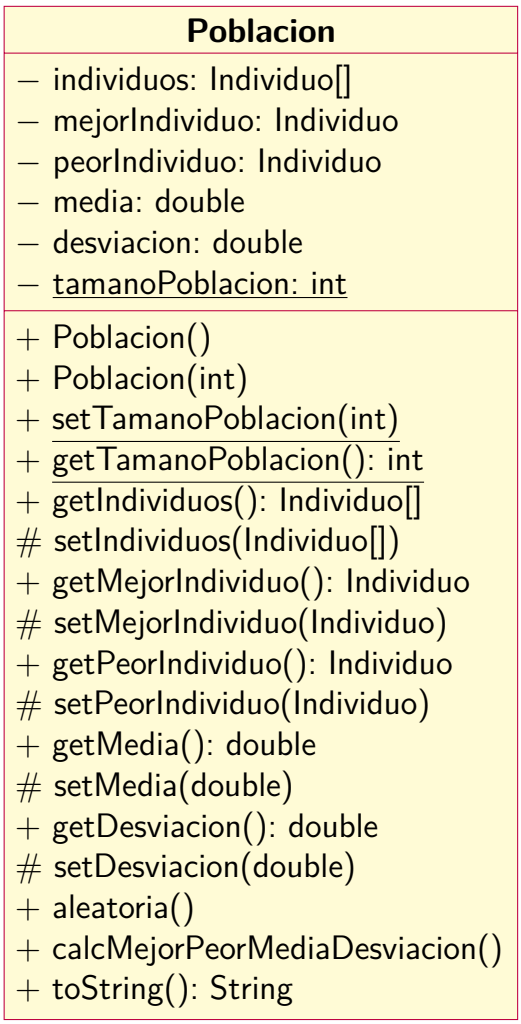


Figura 5.2.: Diagrama - Clase Poblacion (elaboración propia).

Poblacion

`public Poblacion(int opcion)`.
Crea una población de tamaño `tamanoPoblacion` (ubicado en la clase `Configuracion`) según las opciones indicadas. Las opciones válidas son:

`ALEATORIA` crea un objeto con una población de individuos con cromosomas aleatorios.

Parámetros: `opcion` - puede ser `ALEATORIA`.

Arroja: `IllegalArgumentException` - si no se selecciona alguna de las opciones válidas.

Métodos

getTamanoPoblacion

`public static int getTamanoPoblacion()`.
Retorna un valor entero que representa el tamaño de la población. Es decir, devuelve el numero de individuos que existe en una población.
Retorna: el tamaño de la población.

setTamanoPoblacion

`public static void setTamanoPoblacion(int tamanoPoblacion)`.
Establece el tamaño de la población. Es decir, establece el numero de individuos que existe en una población.
Parámetros: `tamanoPoblacion` - el tamaño de la población a utilizar.

Arroja: `IllegalArgumentException` - si `tamanoPoblacion` es menor a 2.

getIndividuos

`public Individuo [] getIndividuos()`.
Retorna un arreglo con los individuos de la población. Los individuos deben ser *individuos específicos*, es decir, objetos de una clase que ha heredado de `Individuo` y ha implementado todos sus métodos.
Retorna: un arreglo con los individuos de la población.

setIndividuos

`protected void setIndividuos(Individuo[] individuos)`.
Establece el arreglo de individuos de la población. Los individuos deben ser *individuos específicos*, es decir, objetos de una clase que ha heredado de `Individuo` y ha implementado todos sus métodos.
Parámetros: `individuos` - el arreglo de individuos a establecer.

getMejorIndividuo

`public Individuo getMejorIndividuo()`.
Devuelve el mejor individuo de la población. Es decir, el individuo con el mayor valor de la función de aptitud entre todos los miembros de la población.
Retorna: el mejor individuo de la población.

setMejorIndividuo

`protected void setMejorIndividuo(Individuo mejorIndividuo)`.
Establece el mejor individuo de la población. Este método es utilizado por el método `generar` de la clase `Generador` donde se determina el individuo con la función de aptitud más alta de los individuos de esta población.
Parámetros: `mejorIndividuo` - el mejor individuo de la población a establecer.

getPeorIndividuo

`public Individuo getPeorIndividuo()`.
Devuelve el peor individuo de la población. Es decir, el individuo con el menor valor de la función de aptitud entre todos los miembros de la población.
Retorna: el peor individuo de la población.

setPeorIndividuo

`protected void setPeorIndividuo(Individuo peorIndividuo)`.
Establece el peor individuo de la población. Este método es utilizado por el método `generar` de la clase `Generador` donde se determina el individuo con la función de aptitud más baja de los individuos de esta población.
Parámetros: `peorIndividuo` - el peor individuo de la población a establecer.

getMedia

`public double getMedia()`.
Retorna la media de las aptitudes de los individuos de la población. Esta media viene dada por la ecuación 5.1 (Montgomery y Runger, 1996, p. 18).

$$\mu = \frac{\sum_{i=1}^{N} x_i}{N} \tag{5.1}$$

Donde μ es la media de la población, x_i el valor de la función de aptitud de un individuo i y N es el tamaño de la población.

Este valor es calculado en el método `generar` de la clase `Generador` y asignado a cada individuo de cada población en cada generación.
Retorna: la media de los valores de las funciones de aptitud de los individuos de esta población.

setMedia

`protected void setMedia(double media)`.
Establece la media de las aptitudes de los individuos. Este método es utilizado por el método `generar` de la clase `Generador` donde se calcula la media de los valores de las funciones objetivos de los individuos de esta población.
Parámetros: `media` - la media a establecer.

getDesviacion

`public double getDesviacion()`.
Devuelve la desviación estándar de la *población*. Esta desviación viene por la ecuación 5.2 (Montgomery y Runger, 1996, p. 28).

$$\sigma = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \mu)^2}{N}} \tag{5.2}$$

Donde σ es la desviación estándar de la población, μ la media de la población, x_i el valor de la función de aptitud de un individuo i y N es el tamaño de la población.

Este valor es calculado en el método `generar` de la clase `Generador` y asignado a cada individuo de cada población en cada generación.
Retorna: la desviación estándar de la población.

setDesviacion

`protected void setDesviacion(double desviacion)`.
Establece la desviación estándar de las aptitudes de los individuos. Este método es utilizado por el método `generar` de la clase `Generador` donde se calcula la desviación estándar de los valores de las funciones objetivos de los individuos de esta población.
Parámetros: `desviacion` - la desviación estándar de la población a establecer.

aleatoria

`public void aleatoria()`.
Genera una población de individuos con cromosomas aleatorios. Para ello crea una arreglo de individuos utilizando el método `fabricaIndividuo` de `Individuo` encargado de producir *individuos específicos*. Luego, por cada individuo, se hace un llamado al método `aleatorio` de `Individuo` que genera cromosomas aleatorios y finalmente se cálcula la media, desviación estándar, mejor y peor individuo utilizando el método `calcMejorPeorMediaDesviacion`.

calcMejorPeorMediaDesviacion

`public void calcMejorPeorMediaDesviacion()`.
Calcula la media, la desviación estándar, el mejor y peor individuo de la población. La media viene dada por la ecuación 5.1 y la desviación estándar por la ecuación 5.2. El mejor y el peor individuo corresponden a aquellos con la función de aptitud de mayor y menor valor, respectivamente.

Todos estos cálculos se realizan en un mismo método con la finalidad de optimizar el desempeño del algoritmo. Si se realizaran en métodos separados se debería hacer un recorrido de todo el arreglo de individuos por cada método, lo cual no es conveniente.

5.1.3. Clase Generacion

`public class Generacion`
`extends java.lang.Object`.
Es una *población* más un conjunto de operadores genéticos y otras variables (ver figura 5.3). Una generación esta formada por:

1. Una *población*, a la que se aplicarán los operadores genéticos.

2. Un *selector*, encargado de determinar que individuos (de la población) tendrán oportunidad de cruzarse.

3. Un *selector post cruce*, que determina que individuos pasan a la siguiente generación.

4. La *probabilidad de cruce*, que determina de forma aleatoria que pareja de individuos se va a cruzar.

5. La *probabilidad de mutación*, que determina de forma aleatoria que descendientes van a mutar sus cromosomas.

6. Un boolean, indica si se va a utilizar *elitismo*.

7. Un *generador*, que acciona el proceso evolutivo utilizando todas las variables anteriores.

Cabe destacar que la diferencia entre una generación y otra es la *población* y el *generador* que tengan asociados, el resto de las características están dadas por variables estáticas que comparten todas las generaciones.

Constructores

Generacion

`public Generacion(Poblacion poblacion)`.
Crea una generación con una población.
Parámetros: `poblacion` - población a asociar.

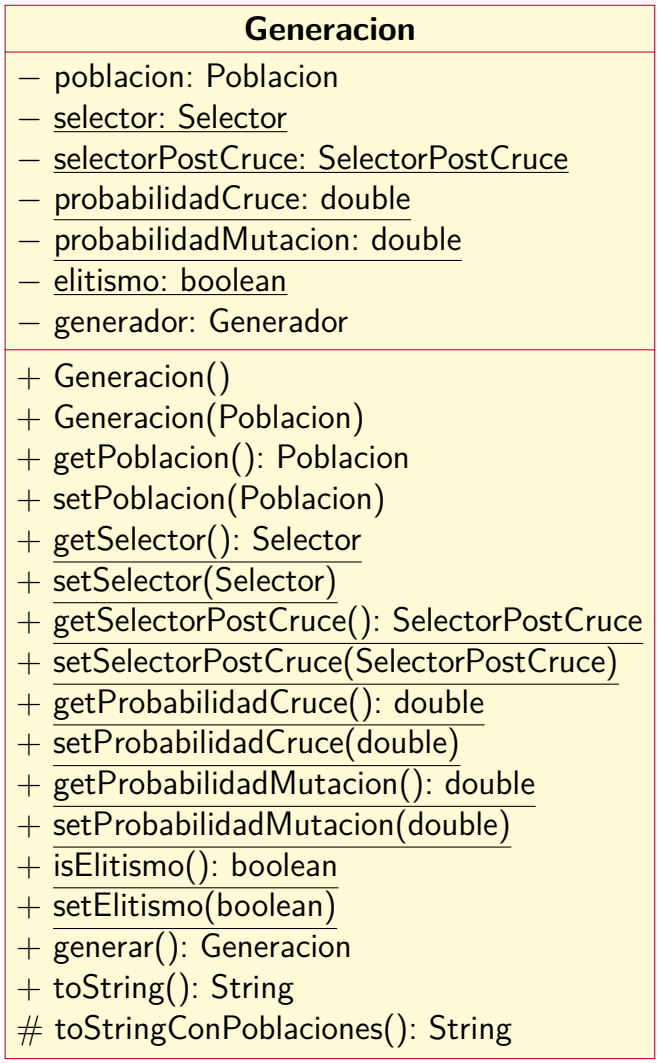


Figura 5.3.: Diagrama - Clase Generacion (elaboración propia).

Métodos

getPoblacion

`public Poblacion getPoblacion()`.
Devuelve la población asociada a esta generación. Esta población es única para esta generación.
Retorna: la población de esta generación.

setPoblacion

`public void setPoblacion(Poblacion poblacion)`.
Establece la población asociada a esta generación.
Parámetros: `poblacion` - la población con la que se va a asociar esta generación.

getSelector

`public static Selector getSelector()`.
Devuelve el *selector* utilizado por todas las generaciones.
Retorna: el selector utilizado por todas las generaciones.

setSelector

`public static void setSelector(Selector selector)`.
Establece el *selector* a utilizar por todas las generaciones.
Parámetros: `selector` - el selector que será utilizado por todas las generaciones.
Arroja: `NullPointerException` - si `selector` es `null`.

getSelectorPostCruce

`public static SelectorPostCruce getSelectorPostCruce()`.
Retorna el *selector post cruce* utilizado por todas las generaciones.

Retorna: el selector post cruce utilizado por todas las generaciones.

setSelectorPostCruce

```
public static void setSelectorPostCruce(
                        SelectorPostCruce selectorPostCruce).
```

Establece el *selector post cruce* a utilizar por todas las generaciones.

Parámetros: `selectorPostCruce` - el selector post cruce que será utilizado por todas las generaciones.

Arroja: `NullPointerException` si `selectorPostCruce` es `null`.

getProbabilidadCruce

```
public static double getProbabilidadCruce().
```

Retorna el valor de la *probabilidad de cruce* utilizado en todas las generaciones. Esta probabilidad determina de forma aleatoria que pareja de individuos se va a cruzar.

Retorna: la probabilidad de cruce utilizada por todas las generaciones.

setProbabilidadCruce

```
public static void setProbabilidadCruce(
                                    double probabilidadCruce).
```

Establece el valor de la *probabilidad de cruce* utilizado en todas las generaciones. Esta probabilidad determina de forma aleatoria que pareja de individuos se va a cruzar.

Parámetros: `probabilidadCruce` - la probabilidad de cruce a utilizar por todas las generaciones.

Arroja: `IllegalArgumentException` - si `probabilidadCruce` sale del rango (0,1).

getProbabilidadMutacion

`public static double getProbabilidadMutacion()`.
Retorna el valor de la *probabilidad de mutación* utilizado en todas las generaciones. Esta probabilidad determina de forma aleatoria que descendientes van a mutar sus cromosomas.
Retorna: la probabilidad de mutación utilizada por todas la generaciones.

setProbabilidadMutacion

```
public static void setProbabilidadMutacion(
                                    double probabilidadMutacion).
```
Establece el valor de la *probabilidad de mutación* utilizado en todas las generaciones. Esta probabilidad determina de forma aleatoria que descendientes van a mutar sus cromosomas.
Parámetros: `probabilidadMutacion` - la probabilidad de mutación a utilizar por todas las generaciones.
Arroja: `IllegalArgumentException` - si `probabilidadMutacion` sale del rango (0,1).

isElitismo

`public static boolean isElitismo()`.
Devuelve un boolean que indica si se va a utilizar *elitismo*. El elitismo es un mecanismo utilizado en algunos algoritmos genéticos para asegurar que los individuos más aptos de una población pasen a la siguiente generación sin ser alterados por ningun operador genético.

El elitismo asegura que la aptitud máxima de la población nunca se reducirá de una generación a la siguiente. Sin embargo, no necesariamente mejora la posibilidad de localizar el optimo global de una función.
Retorna: un boolean, true si se utiliza elitismo.

setElitismo

`public static void setElitismo(boolean elitismo)`.
Establece si aplica o no el *elitismo*. Se llama elitismo al mecanismo utilizado en algunos algoritmos genéticos para asegurar que los individuos más aptos de una población pasen a la siguiente generación sin ser alterados por ningun operador genético.

El elitismo asegura que la aptitud máxima de la población nunca se reducirá de una generación a la siguiente. Sin embargo, no necesariamente mejora la posibilidad de localizar el optimo global de una función.
Parámetros: `elitismo` - un boolean, true si se desea utilizar elitismo.

generar

`public Generacion generar()`.
Inicia la evolución de una generación a otra por medio de la acción de los operadores genéticos. Este método crea un objeto de tipo `Generador` y hace un llamado al método auxiliar `generar` donde son aplicados los operadores genéticos a la población haciendo uso de las probabilidades de cruce (`probabilidadCruce`) y mutación (`probabilidadMutacion`).
Retorna: una nueva generación producto de esta.

toString

`public String toString()`.
Devuelve una cadena de caracteres con datos de la generación. Estos datos consisten en variables de la población asociada a esta generación:

Media es la media de la función de aptitud de los individuos de la población.

Desviación Estándar es la desviación estándar de la función de aptitud de los individuos de la población.

Mejor Individuo es el individuo con la mayor función de aptitud de la población.

Peor Individuo es el individuo con la menor función de aptitud de la población.

Sobrescribe: `toString` en la clase `java.lang.Object`.
Retorna: una cadena de caracteres con datos de la generación.

toStringConPoblaciones

`protected String toStringConPoblaciones()`.
Devuelve una cadena de caracteres con datos de la generación incluyendo a los individuos de la población. Devuelve los mismos datos que el método `toString` y además muestra a todos los individuos de la población asociada a esta generación.
Retorna: una cadena de caracteres con datos de la generación incluyendo a los individuos de la población.

5.1.4. Clase Ejecucion

`public class Ejecucion.`
`extends java.lang.Object`.
Es una corrida independiente de un algoritmo genético (ver figura 5.4). Es un proceso evolutivo completo, consiste en un arreglo de objetos de tipo `Generacion`. En esta clase se crea una *generación i* a partir de una *generación i-1*, esto se realiza tantas veces como se indique en el método `setNumeroGeneraciones`.

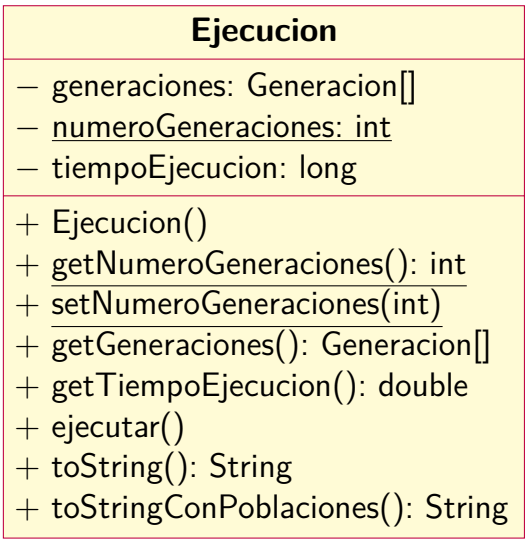


Figura 5.4.: Diagrama - Clase Ejecucion (elaboración propia).

Métodos

getNumeroGeneraciones

`public static int getNumeroGeneraciones()`.
Devuelve el numero de *generaciones* por ejecución.
Retorna: el numero de generaciones por ejecución.

setNumeroGeneraciones

`public static void setNumeroGeneraciones(int numeroGeneraciones)`.
Establece el numero de *generaciones* por ejecución. Una *generaciones* consiste en una *población* más un conjunto de operadores genéticos y otras variables que son aplicadas a dicha población para crear una nueva generación.

Una generación produce otra generación y esta a la siguiente y así sucesivamente hasta alcanzar una cantidad igual al valor establecido en este

método. Una vez se llega a la ultima generación se detiene la ejecución y se pasa a la siguiente si lo permite el parámetro `numeroEjecuciones`.
Parámetros: `numeroGeneraciones` - el numero de generaciones a establecer.
Arroja: `IllegalArgumentException` - si `numeroGeneraciones` es menor a 1.

getGeneraciones

`public Generacion[] getGeneraciones()`.
Devuelve las *generaciones* de una ejecución. A partir de estas generaciones se pueden obtener otros datos de interés como la población asociada a cada generación, los individuos, media y desviación de las funciones de aptitud.
Retorna: la generaciones de la ejecución.

getTiempoEjecucion

`public double getTiempoEjecucion()`.
Devuelve el tiempo que tarda la ejecución en completarse. Es decir, devuelve el tiempo que tardan en ejecutarse todas las *generaciones* del algoritmo genético. Esta método debe llamarse después del método `ejecutar` de esta misma clase, de lo contrario se genera un excepción `IllegalStateException`.
Retorna: el tiempo de la ejecución.
Arroja: `IllegalStateException` - si no se llama previamente al método `ejecutar`.

ejecutar

`public void ejecutar()`.
Inicia la corrida de un algoritmo genético. Este método crea una población inicial de individuos con cromosomas aleatorios utilizando el constructor `Poblacion(int)` con la opción `ALEATORIA`. Esta población

inicial es introducida en el constructor de la primera generación del arreglo y a partir de este momento, comienza un ciclo en el que una nueva generación (*i*) es creada a partir de una anterior (*i-1*) por medio del método `generar` de `Generacion` hasta alcanzar una cantidad de generaciones igual a `numeroGeneraciones`.

toString

`public String toString()`.
Devuelve una cadena de caracteres con datos de la ejecución. La cadena muestra el numero de cada generación y muestra los datos de esta. Este método **no muestra** a los individuos de las poblaciones asociadas a cada generación.
Sobrescribe: `toString` en la clase `java.lang.Object`.
Retorna: una cadena de caracteres con datos de la ejecución.

toStringConPoblaciones

`public String toStringConPoblaciones()`.
Devuelve una cadena de caracteres con datos de la ejecución incluyendo a los individuos de la población. La cadena muestra el numero de cada generación y muestra los datos de esta. Este método **muestra** a los individuos de las poblaciones asociadas a cada generación.
Retorna: una cadena de caracteres con datos de la ejecución incluyendo a los individuos de la población.

5.1.5. Clase Proceso

`public class Proceso.`
`extends java.lang.Object`.
Un proceso es un conjunto de *ejecuciones* (ver figura 5.5). Es decir, un conjunto de evoluciones independientes una de la otra. Esta formado por un arreglo de objetos tipo `Ejecucion`.

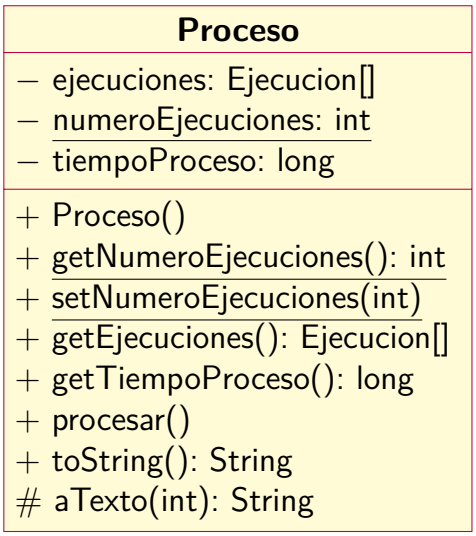


Figura 5.5.: Diagrama - Clase Proceso (elaboración propia).

La finalidad de esta clase es obtener datos de varias ejecuciones en forma automática para luego tener la posibilidad de computarlos todos estadísticamente. Esto se logra por medio del método `procesar` que inicia el proceso en una sucesión de ejecuciones.

Métodos

getNumeroEjecuciones

`public static int getNumeroEjecuciones()`.
Devuelve el numero de *ejecuciones* del proceso. Es decir, devuelve el numero de corridas independientes de algoritmos genéticos establecido.
Retorna: el numero de ejecuciones.

setNumeroEjecuciones

```
public static void setNumeroEjecuciones(
                                    int numeroEjecuciones).
```

Establece el numero de *ejecuciones* del algoritmo genético. Una *ejecución* es una corrida independiente de un algoritmo genético. Este parámetro esta disponible con el propósito de obtener datos de varias corridas en forma automática para luego tener la posibilidad de computarlos todos estadísticamente.

En las fases iniciales de desarrollo de un problema específico se recomienda mantener este valor en 1, ya que valores más elevados pudieran ocasionar colapsos de memoria en el sistema.
Parámetros: `numeroEjecuciones` - el numero de ejecuciones a utilizar.
Arroja: `IllegalArgumentException` - si `numeroEjecuciones` es menor a 1.

getEjecuciones

`public Ejecucion[] getEjecuciones()`.
Devuelve las *ejecuciones* de un proceso. A partir de estas ejecuciones se pueden obtener otros datos de interés como las generaciones, las poblaciones y los individuos.
Retorna: las ejecuciones del proceso.

getTiempoProceso

`public long getTiempoProceso()`.
Devuelve el tiempo que tarda el proceso en completarse. Es decir, devuelve el tiempo que tardan en ejecutarse todas las *ejecuciones*. Esta método debe llamarse después del método `procesar` de lo contrario se genera un excepción `IllegalStateException`.
Retorna: el tiempo que se tarda el proceso.
Arroja: `IllegalStateException` - si no se llama previamente al método `procesar`.

procesar

`public void procesar()`.
Inicia un proceso. Es decir, inicia una secuencia de *ejecuciones*. Además, `procesar`, calcula el tiempo en segundos que tarde en completarse el mismo y almacena esta valor en una variable a la que se puede acceder con el método `getTiempoProceso`.

toString

`public String toString()`.
Devuelve una cadena de caracteres con datos del proceso. Estos datos muestran la cantidad de ejecuciones más los datos del algoritmo genético, que son los mismos datos de cada ejecución y son los configurados por los *setters* a los que se llama antes del método `iniciarProceso` de la clase `Configuracion`.
Sobrescribe: `toString` en la clase `java.lang.Object`.
Retorna: una cadena de caracteres con datos del proceso.

aTexto

`protected String aTexto(int opcion)`.
Devuelve una cadena de caracteres con datos de este proceso. El tipo de resultados varia en función de las opciones que se pasen como parámetros. Las opciones retornan lo siguiente:

`Configuracion.ULTIMA_GENERACION` - Los datos básicos de la última generación de la última ejecución.

`Configuracion.GENERACIONES_SIN_POBLACIONES` - Los datos básicos de todas las generaciones de todas las ejecuciones.

`Configuracion.GENERACIONES_CON_POBLACIONES` - Los datos de todas las generaciones incluyendo sus respectivas poblaciones.

Advertencia: La cadena de caracteres depende de la representación que el usuario desee al sobrescribir el método `toString` del *individuo específico*. Se debe recordar que un *individuo específico* es una clase que hereda de la clase `Individuo`.

Parámetros: `opcion` - puede ser
`Configuracion.ULTIMA_GENERACION`,
`Configuracion.GENERACIONES_SIN_POBLACIONES` o
`Configuracion.GENERACIONES_CON_POBLACIONES`.

Retorna: una cadena de caracteres con datos del proceso.

Arroja: `IllegalArgumentException` - si no se selecciona alguna de las opciones válidas.

5.1.6. Clase Configuracion

`public class Configuracion.`
`extends java.lang.Object.`
Es un tablero de control donde se establecen a los parámetros de un proceso y se da inicio a este (ver figura 5.6). Esto se logra al hacer llamados a distintos métodos *setters* y al llamar al método `iniciarAlgoritmo` que acciona la corrida.

Al crear un objeto de tipo `Configuracion` se deben hacer llamados a la totalidad de sus 10 métodos *setters*, posteriormente se debe llamar al método `iniciarAlgoritmo` para que se inicie la corrida del algoritmo. Los pasos anteriores son de carácter **obligatorio** para cualquier corrida, y si, por ejemplo, se omite el llamado a algún *setters* y luego se llama a `iniciarAlgoritmo` se genera una excepción `IllegalStateException` indicando cual es el *setters* omitido y que debe ser llamado.

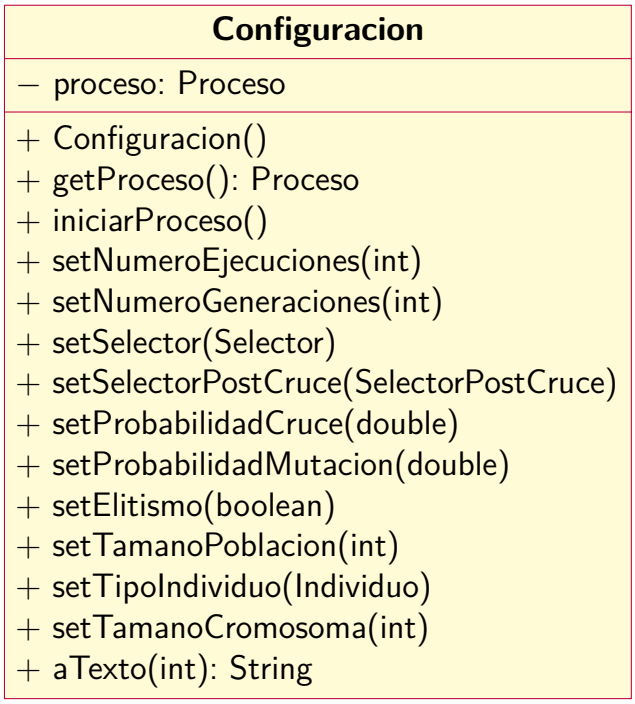


Figura 5.6.: Diagrama - Clase Configuracion (elaboración propia).

Constantes

ULTIMA_GENERACION

`public static final int ULTIMA_GENERACION`.
Especifica al método `aTexto` devolver datos básicos de la última generación de la última ejecución.

GENERACIONES_SIN_POBLACIONES

`public static final int GENERACIONES_SIN_POBLACIONES`.
Especifica al método `aTexto` devolver los datos básicos de todas las generaciones de todas las ejecuciones.

GENERACIONES_CON_POBLACIONES

`public static final int GENERACIONES_CON_POBLACIONES`.
Especifica al método `aTexto` devolver los datos de todas las generaciones incluyendo sus respectivas poblaciones.

Métodos

getProceso

`public Proceso getProceso()`.
Devuelve un objeto de tipo `Proceso`. Este objeto se crea el ejecutar el método `iniciarProceso`, por lo que devuelve el proceso asociado a esta configuración. A partir de aquí se pueden obtener los demás datos de interés como ejecuciones, generaciones, poblaciones, individuos...
Retorna: el proceso asociado a esta configuración.

iniciarProceso

`public void iniciarProceso()`.
Inicia un proceso. Es decir, inicia una secuencia de ejecuciones. Este método tiene como requisito, el previo llamado a los 10 *setters* que establecen los parámetros de un algoritmo genético, de lo contrario dispara un excepción `IllegalStateException`. Se recomienda el llamado a los *setters* en el siguiente orden:

1. `setNumeroEjecuciones`
2. `setNumeroGeneraciones`
3. `setSelector`
4. `setSelectorPostCruce`
5. `setProbabilidadCruce`
6. `setProbabilidadMutacion`
7. `setElitismo`
8. `setTamanoPoblacion`
9. `setTipoIndividuo`
10. `setTamanoCromosoma`

Arroja: `IllegalStateException` - si alguno de los 10 métodos obligatorios no ha sido llamado.

setNumeroEjecuciones

`public void setNumeroEjecuciones(int numeroEjecuciones)`.
Establece el número de ejecuciones del proceso. Una *ejecución* es una corrida independiente de un algoritmo genético. El parámetro `numeroEjecuciones` se establece con el propósito de obtener datos de varias corridas en forma automática para luego tener la posibilidad de computarlos todos estadísticamente.

En las fases iniciales de desarrollo de un problema específico se recomienda mantener este valor en 1, ya que valores más elevados pudieran ocasionar colapsos de memoria en el sistema.
Parámetros: `numeroEjecuciones` - el número de ejecuciones a utilizar.
Arroja: `IllegalArgumentException` - si `numeroEjecuciones` es menor a 1.

setNumeroGeneraciones

`public void setNumeroGeneraciones(int numeroGeneraciones)`.
Establece el número de generaciones por ejecución. Una *generación* consiste en una *población* más un conjunto de operadores genéticos y otras variables que son aplicadas a dicha población para crear una nueva generación.

Una generación produce otra generación y esta a la siguiente y así sucesivamente hasta alcanzar una cantidad igual al valor establecido en este método. Una vez se llega a la última generación se detiene la ejecución y se pasa a la siguiente si lo permite el parámetro `numeroEjecuciones`.
Parámetros: `numeroGeneraciones` - el número de generaciones a utilizar.
Arroja: `IllegalArgumentException` - si `númeroGeneraciones` es menor a 1.

setSelector

`public void setSelector(Selector selector)`.
Determina que *selector* se va a utilizar en la etapa de selección. Un *selector* es un objeto de una clase que implemente la interfaz `Selector` y que por consiguiente implemente el método `seleccion` encargado de seleccionar dos individuos de la población para que tengan la posibilidad de cruzarse, siguiendo un criterio específico.
Parámetros: `selector` - el selector a utilizar.
Arroja: `NullPointerException` - si `selector` es `null`.

setSelectorPostCruce

`public void setSelectorPostCruce(`
`SelectorPostCruce selectorPostCruce).`
Determina que *selector post cruce* se va a utilizar después del cruce. El *selector post cruce* es un objeto de una clase que implementa la interfaz `SelectorPostCruce` y que por consiguiente implemente el método `seleccionPostCruce`.

Una vez efectuado el cruce entre dos individuos se producen un par de hijos, transformando un par en dos pares de individuos, luego, `seleccionPostCruce` se encarga de determinar quienes entre los cuatro individuos pasaran a la siguiente generación.
Parámetros: `selectorPostCruce` - el selector post cruce a utilizar.
Arroja: `NullPointerException` - si `selectorPostCruce` es `null`.

setProbabilidadCruce

`public void setProbabilidadCruce(double probabilidadCruce).`
Establece el valor de la probabilidad de cruce a utilizar. Una vez que un par de individuos es seleccionado por el *selector*, sus posibilidades de cruzarse dependen del valor establecido en este método. Este valor debe estar comprendido entre cero y uno de lo contrario se produce una excepción `IllegalArgumentException`.
Parámetros: `probabilidadCruce` - el valor de la probabilidad de cruce a utilizar.
Arroja: `IllegalArgumentException` - si `probabilidadCruce` sale del rango (0,1).

setProbabilidadMutacion

`public void setProbabilidadMutacion(`
`double probabilidadMutacion).`
Establece el valor de la probabilidad de mutación a utilizar. Después que

un par de individuos se cruzan, el par de hijos resultantes tienen la posibilidad de mutar sus cromosomas, esta posibilidad la define el valor establecido en este método. Este valor debe estar comprendido entre cero y uno de lo contrario se produce una excepción `IllegalArgumentException`.

Parámetros: `probabilidadMutacion` - el valor de la probabilidad de mutación a utilizar.

Arroja: `IllegalArgumentException` - si `probabilidadMutacion` sale del rango (0,1).

setElitismo

`public void setElitismo(boolean elitismo)`.
Establece si aplica o no el *elitismo*. Se llama *elitismo* al mecanismo utilizado en algunos algoritmos genéticos para asegurar que los individuos más aptos de una población pasen a la siguiente generación sin ser alterados por ningun operador genético.

El elitismo asegura que la aptitud máxima de la población nunca se reducirá de una generación a la siguiente. Sin embargo, no necesariamente mejora la posibilidad de localizar el optimo global de una función.

Parámetros: `elitismo` - un boolean, true si se desea utilizar elitismo.

setTamanoPoblacion

`public void setTamanoPoblacion(int tamanoPoblacion)`.
Establece el tamaño de la población, es decir, la cantidad de individuos.

Parámetros: `tamanoPoblacion` - el tamaño de la población a establecer.

Arroja: `IllegalArgumentException` - si `tamanoPoblacion` es menor a 2.

setTipoIndividuo

`public void setTipoIndividuo(Individuo tipoIndividuo)`.
Establece el tipo de *individuo específico* que se va a utilizar. Un *individuo específico* es un objeto de una clase que ha heredado de `Individuo` y

ha implementado todos sus métodos, esta es una tarea que corresponde al usuario.
Parámetros: `tipoIndividuo` - el tipo de individuo a establecer.
Arroja: `NullPointerException` - si `tipoIndividuo` es `null`.

setTamanoCromosoma

`public void setTamanoCromosoma(int tamanoCromosoma)`.
Establece el tamaño del cromosoma del *individuo específico*. Este valor esta intrínsecamente relacionado con el diseño del *individuo específico* por lo que el usuario debe tener muy claro como la variación del tamaño afecta los datos del algoritmo genético. Generalmente cuando se trata de cromosomas binarios, este valor deber ser fijo.

Un *individuo específico* es un objeto de una clase que ha heredado de `Individuo` y ha implementado todos sus métodos, para lo cual, obligatoriamente, debe establecer el valor del tamaño del cromosoma como una variable determinada por el método `getTamanoCromosoma`.
Parámetros: `tamanoCromosoma` - el tamaño de cromosoma a establecer.
Arroja: `IllegalArgumentException` - si `tamanoCromosoma` es menor a 1.

aTexto

`public java.lang.String aTexto(int opcion)`.
Devuelve una cadena de caracteres con datos del proceso. El tipo de resultados varia en función de las opciones que se pasen como parámetros. Las opciones retornan lo siguiente:

`ULTIMA_GENERACION` - Los datos básicos de la última generación de la última ejecución.

`GENERACIONES_SIN_POBLACIONES` - Los datos básicos de todas las generaciones de todas las ejecuciones.

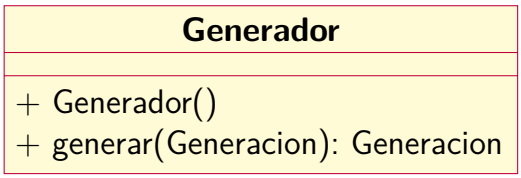


Figura 5.7.: Diagrama - Clase Generador (elaboración propia).

`GENERACIONES_CON_POBLACIONES` - Los datos de todas las generaciones incluyendo sus respectivas poblaciones.

Advertencia: La cadena de caracteres depende de la representación que el usuario desee al sobrescribir el método `toString` del *individuo específico*. Se debe recordar que un *individuo específico* es una clase que hereda de la clase `Individuo`.

Parámetros: `opcion` - puede ser `ULTIMA_GENERACION`, `GENERACIONES_SIN_POBLACIONES` o `GENERACIONES_CON_POBLACIONES`.

Retorna: una cadena de caracteres con datos del proceso.

Arroja: `IllegalArgumentException` - si no se selecciona alguna de las opciones válidas.

5.1.7. Clase Generador

`public class Generador`
`extends java.lang.Object`.
Clase auxiliar de la clase `Generacion` cuya única finalidad es alojar al método `generar` (ver figura 5.7).

Métodos

generar

`public Generacion generar(Generacion generacionActual)`.
Inicia la evolución de una generación a otra por medio de la acción de los operadores genéticos. Recibe un objeto de tipo `Generacion` y aplica a los individuos de este, los métodos `cruce` y `mutacion` de la clase `Individuo`. También aplica los métodos de las interfaces `Selector` y `SelectorPostCruce` correspondientes a las formas de selección previas y posteriores al cruce.

Además, este método hace uso de las variables `probabilidadCruce` y `probabilidadMutacion` de `Generacion` correspondientes a los valores de las probabilidades de cruce y mutación. También utiliza el valor lógico de la variable booleana `elitismo` de `Generacion` si se desea (o no) utilizar la herramienta elitista.
Parámetros: `generacionActual` - la generación a la que se aplicarán los operadores genéticos.
Retorna: una nueva generación (evolucionada).

5.1.8. Interfaz Selector

`public interface Selector`.
Un selector selecciona individuos que tendrán posibilidad de cruzarse (ver figura 5.8). Cualquier clase de desee aplicar algún criterio de selección debe implementar esta interfaz, en ella se encuentran dos métodos:

1. `seleccion` - se encarga de seleccionar una cantidad especifica de individuos que tendrán la posibilidad de cruzarse, ello según el criterio definido por el usuario al implementar esta interfaz.

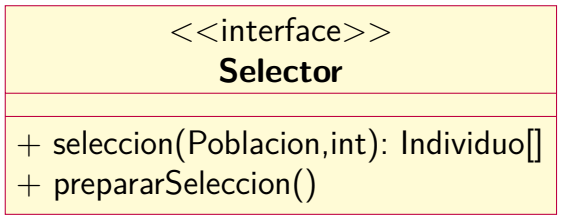


Figura 5.8.: Diagrama - Interfaz Selector (elaboración propia).

2. `preparar` - permite aplicar cálculos adicionales a toda la población que luego puedan ser utilizados por ş como, por ejemplo, calcular el valor esperado de cada individuo u ordenarlos por jerarquías. Este método se aplica a toda una población y se ejecuta antes del método `generar` de la clase `Generacion`.

Métodos (abstractos)

seleccion

```
Individuo[] seleccion(
          Poblacion poblacion, int cantidadSeleccionados).
```

Selecciona una cantidad especifica de individuos de la población que tendrán la posibilidad de cruzarse. El criterio utilizado para dicha selección es propio de la clase que implemente esta interfaz.

Este método es llamado por `generar` de `Generador` cada vez que se deseen seleccionar individuos con oportunidades de cruce. Puede utilizar el método `prepararSeleccion` como apoyo cuando se deban calcular datos adicionales o hacer algún ordenamiento.

Parámetros: `poblacion` - la población de donde serán seleccionados los individuos.
`cantidadSeleccionados` - la cantidad de individuos a seleccionar.

Retorna: un arreglo de tamaño `cantidadSeleccionados` con los individuos seleccionados.

prepararSeleccion

`void prepararSeleccion(Poblacion poblacion).`
Puede utilizarse para calcular datos adicionales a una población. Por ejemplo, se puede ordenar la población o calcular los valores esperados de cada individuo. Todo ello con la finalidad que el método `seleccion` cumpla con los requisitos del criterio de selección elegido por el usuario.

Este método es llamado por `ejecutar` de la clase `Ejecucion` una vez por generación, dado que en cada generación se debe preparar los datos que utilizara la siguiente generación.

El cuerpo de este método puede quedar vacío (sin código) si el método `seleccion` no necesita datos adicionales o algún tipo de ordenamiento.
Parámetros: `poblacion` - la población a la que se le aplicaran cálculos adicionales.

5.1.9. Interfaz SelectorPostCruce

`public interface SelectorPostCruce.`
La interfaz SelectorPostCruce se ocupa de la selección que ocurre una vez se ha realizado el cruce entre dos individuos (ver figura 5.9). Es un proceso que no tiene relación con la interfaz `Selector` ni con el método `seleccion` de la misma.

El proceso de cruce involucra a dos padres (que se cruzan) y dos hijos (que descienden de sus padres), al tener estos cuatro individuos se plantea la pregunta: ¿Quienes pasarán a la siguiente generación?. El usuario debe tomar esta decisión y estructurarla con la implementación del método `seleccionPostCruce`.

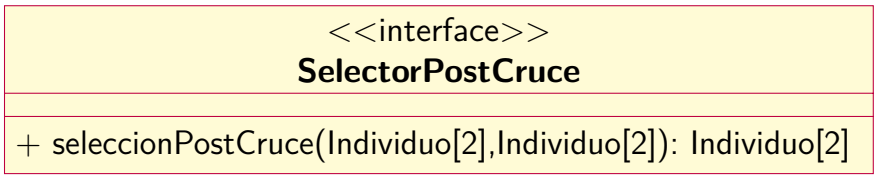


Figura 5.9.: Diagrama - Interfaz SelectorPostCruce (elaboración propia).

Generalmente se utilizan dos criterios: seleccionar solo a los hijos o seleccionar a las mejores entre padres e hijos. Ambos criterios se encuentran implementados en las clases `SelectorPostCruceSoloHijos` y `SelectorPostCruceTaigeto` ubicadas en el paquete `agapi`.

Métodos (abstractos)

seleccionPostCruce

```
Individuo[] seleccionPostCruce(
                    Individuo[] padres, Individuo[] hijos).
```

Este método selecciona dos individuos entre padres e hijos. Devuelve a un arreglo de *individuos* que **siempre** debe ser de tamaño **dos (2)**. Sus parámetros también son arreglos de individuos de tamaño dos (2) que corresponden a padres e hijos.

Parámetros: `padres` - los 2 individuos que se cruzan.
`hijos` - los 2 individuos productos del cruce.

Retorna: los 2 individuos seleccionados para la siguiente generación.

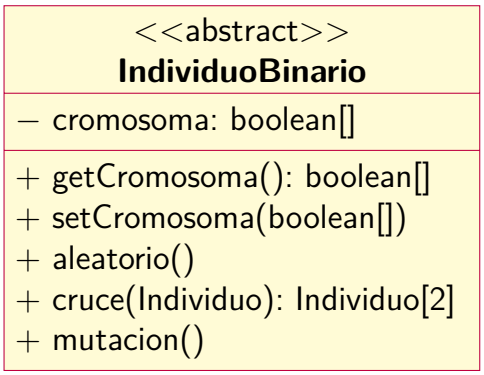


Figura 5.10.: Diagrama - Clase abstracta IndividuoBinario (elaboración propia).

5.2. Paquete impl

5.2.1. Clase IndividuoBinario (abstracta)

`public abstract class IndividuoBinario.`
`extends Individuo.`
Un individuo binario es un individuo con un cromosoma formado por una cadena de ceros y unos(ver figura 5.10). Esta clase implementa los métodos abstractos `aleatorio`, `cruce` y `mutacion` de `Individuo`. El único método que no implementa es `calcFA`, esto se debe, obviamente, a que cada función de aptitud es única para cada problema y queda de parte del usuario su implementación al heredar de esta clase.

La cadena de ceros y unos esta representada por la variable `cromosoma` que consiste en un arreglo de tipo boolean.

Métodos

getCromosoma

`public boolean[] getCromosoma().`
Devuelve la cadena de ceros y unos correspondiente al cromosoma de este individuo.
Retorna: el cromosoma de este individuo.

setCromosoma

`public void setCromosoma(boolean[] cromosoma).`
Establece una cadena de ceros y unos correspondiente al cromosoma de este individuo.
Parámetros: `cromosoma` - el cromosoma a establecer.

aleatorio

`public void aleatorio()`
Transforma el cromosoma de este individuo en una sucesión aleatoria de ceros y unos, es decir, en un arreglo de tipo boolean.

cruce

`public Individuo[] cruce(Individuo madre).`
Retorna un par individuos productos del cruce entre este individuo y el individuo madre. El cruce consiste en el intercambio de las secuencias de los cromosomas de los individuos padres. Se selecciona una posición aleatoria en la secuencia de unos y ceros, a partir de este punto se intercambian las secuencias del padre y la madre para obtener dos nuevos individuos.
Parámetros: `madre` - el individuo madre.
Retorna: un individuo producto del cruce entre este individuo y el individuo madre.

mutacion

`public void mutacion().`
Modifica el valor de un elemento en la secuencia de unos y ceros. Escoge de forma aleatoria una posición e invierte su valor: si es un uno lo cambia por un cero o viceversa.

5.2.2. Clase IndividuoCombinatorio (abstracta)

`public abstract class IndividuoCombinatorio.`
`extends Individuo.`
Un individuo combinatorio es un individuo con un cromosoma formado por una cadena de números enteros de tamaño `tamanoCromosoma` (ver figura 5.11). Los números son una secuencia no ordenada desde 1 hasta n, donde n es igual al tamaño de la población.

Esta clase implementa los métodos abstractos `aleatorio`, `cruce` y `mutacion` de `Individuo`. El único método que no implementa es `calcFA`, esto se debe, obviamente, a que cada función de aptitud es única para cada problema y queda de parte del usuario su implementación al heredar de esta clase.

La cadena de números enteros de este individuo esta representada por la variable `cromosoma` que consiste en un arreglo de tipo `int`. PMX es el tipo de cruce utilizado por este individuo, PMX son las siglas en ingles para *cruce parcialmente mapeado* (*partially matched crossover*), para la implementación de este tipo de recombinación se hace un llamado al método `cruce` de la clase `CrucePMX`. PMX se explica con mejor detalle en la descripción del método `cruce`.

La mutación consiste en el simple intercambio de dos genes (números enteros) dentro de la misma secuencia.

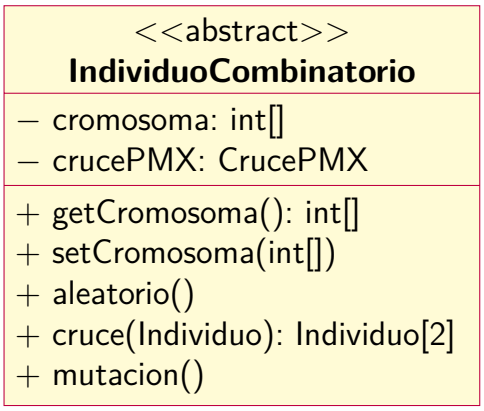


Figura 5.11.: Diagrama - Clase abstracta IndividuoCombinatorio (elaboración propia).

Métodos

getCromosoma

`public int[] getCromosoma().`
Devuelve la cadena de números enteros correspondiente al cromosoma de este individuo.
Retorna: el cromosoma de este individuo.

setCromosoma

`public void setCromosoma(int[] cromosoma).`
Establece una cadena de números enteros correspondiente al cromosoma de este individuo.
Parámetros: `cromosoma` - el cromosoma a establecer

aleatorio

`public void aleatorio().`
Transforma el cromosoma de este individuo en una sucesión aleatoria de números enteros, es decir, en un arreglo de tipo `int`.

cruce

`public Individuo[] cruce(Individuo madre).`
Retorna un par individuos combinatorios productos del cruce entre este individuo combinatorio y el individuo madre, utilizando un operador PMX. PMX son las siglas en ingles para *cruce parcialmente mapeado* (*partially matched crossover*) y es un operador de reordenamiento, es decir, que modifica el orden de una secuencia, fue presentado por Goldberg & Lingle (1985).

El cruce PMX consiste en tomar dos puntos de cruce aleatoriamente e intercambiar las secuencias de cada individuo ubicadas entre estos puntos, estos segmentos son inmodificables. Luego se debe seguir un proceso explicado por Goldberg (1989) y Haupt & Haupt (2004) donde se sustituyen los elementos repetidos de un cromosoma por los repetidos del otro.
Parámetros: `madre` - el individuo madre.
Retorna: un par de individuos combinatorios producto del cruce entre este individuo y el individuo madre.

mutacion

`public void mutacion().`
La mutación consiste en el simple intercambio de dos genes (números enteros) dentro de la misma secuencia. Para ello genera dos números aleatorios correspondientes a las posiciones y luego intercambia los elementos.

5.2.3. Clase CrucePMX

`public class CrucePMX.`
`extends java.lang.Object.`
Esta clase tiene como única función suministrar un método `cruce` de tipo PMX (Goldberg y Lingle, 1985) para la clase `IndividuoCombinatorio`

CrucePMX
+ cruce(IndividuoCombinatorio, IndividuoCombinatorio): IndividuoCombinatorio[2]

Figura 5.12.: Diagrama - Clase CrucePMX (elaboración propia).

(abstracta, ver figura 5.12). Esto se debe a la complejidad del proceso de cruce tipo PMX, lo que amerita toda una clase para su desarrollo.

Métodos

cruce

```
public IndividuoCombinatorio[] cruce(
                                IndividuoCombinatorio madre,
                                IndividuoCombinatorio padre)
```

Retorna un par de individuos combinatorios productos del cruce entre los individuos combinatorios madre y padre, utilizando un operador PMX.

Parámetros: `madre` - el individuo madre.
`padre` - el individuo padre.

Retorna: un par de individuos combinatorios producto del cruce entre madre y padre.

5.2.4. Clase SelectorTorneo

```
public class SelectorTorneo.
extends java.lang.Object.
implements Selector.
```

Es un selector que aplica la técnica de selección *por torneo, probabilístico y binario* (ver figura 5.13). En términos generales, un método de selección

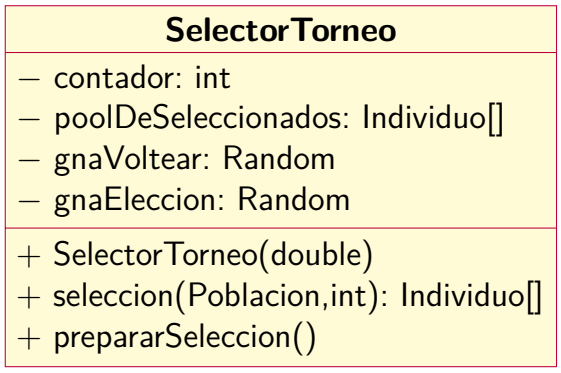


Figura 5.13.: Diagrama - Clase SelectorTorneo (elaboración propia).

por torneo consiste en escoger aleatoriamente a un grupo de individuos de la población, escoger a los mejores de este grupo y repetir el proceso hasta llenar el pool de seleccionados. En el caso concreto de este selector se escogen aleatoriamente dos individuos , luego se genera un número aleatorio r entre 0 y 1. Si $r < k$ (donde k es un parámetro entre 0.5 y 1), el individuo más apto será seleccionado, de lo contrario el menos apto será seleccionado.

Este tipo de específico de selección es llamado *binario* por realizar el torneo entre dos individuos, y *probabilístico*, por utilizar un número aleatorio r para determinar el ganador. David Golberg y Kalyanmoy Deb (1991) hacen un buen análisis de este método.
Ver también: Selector.

Constructores

SelectorTorneo

`public SelectorTorneo(double k).`
Este constructor recibe como parámetro el valor de la constante k encargada de establecer el límite entre el individuo ganador y el perdedor. Este valor debe estar comprendido entre 0.5 y 1 de lo contrario se

produce una excepción de tipo `IllegalArgumentException`.
Parámetros: `k` -valor de la constante k.
Arroja: `IllegalArgumentException` - si k está fuera del rango [0.5,1].

Métodos

seleccion

`public Individuo[] seleccion(Poblacion poblacion, int cantidadSeleccionados).`
Selecciona una cantidad especifica de individuos de la población según el la técnica *por torneo probabilístico y binario*.

Este método es llamado por `generar` de `Generador` cada vez que se deseen seleccionar individuos con oportunidades de cruce.
Parámetros: `poblacion` - la población de donde serán seleccionados los individuos.
`cantidadSeleccionados` - la cantidad de individuos a seleccionar.
Retorna: un arreglo de tamaño `cantidadSeleccionados` con los individuos seleccionados.

prepararSeleccion

`public void prepararSeleccion(Poblacion poblacion).`
Este método genera un pool de individuos seleccionados según la técnica *por torneo probabilístico y binario*.

Este método es llamado por `ejecutar` de la clase `Ejecucion` una vez por generación, dado que en cada generación se debe preparar los datos que utilizara la siguiente generación.
Parámetros: `poblacion` - la población a la que se le aplicará la técnica *por torneo*.

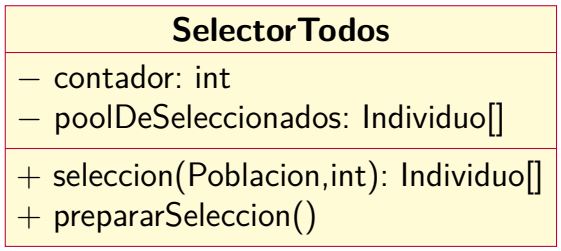


Figura 5.14.: Diagrama - Clase SelectorTodos (elaboración propia).

5.2.5. Clase SelectorTodos

```
public class SelectorTodos.
extends java.lang.Object.
implements Selector.
```

Esta clase debe ser utilizada solo para realizar pruebas ya que realmente no se ejecuta ningún tipo de *selección* (ver figura 5.14). Simplemente se toman todos los individuos de una población ordenadamente del primero al último, esto implica que todos los individuos tendrán oportunidad de cruzarse y si, por ejemplo, la *probabilidad de cruce* se configura en 1.0, absolutamente todos los individuos se cruzarán, sin excepción. También, en forma opuesta, se puede configurar la probabilidad de cruce en 0.0 y entonces ningún individuo se cruzará.
Ver también: Selector.

Métodos

seleccion

```
public Individuo[] seleccion(Poblacion poblacion,
                             int cantidadSeleccionados).
```

Selecciona a todos los individuos de la población ordenadamente desde el primero hasta el último. Este orden se reinicia cada vez que se llama al método `prepararSeleccion`.

Parámetros: `poblacion` - la población de donde serán seleccionados los individuos.
`cantidadSeleccionados` - la cantidad de individuos a seleccionar.
Retorna: un arreglo de tamaño `cantidadSeleccionados` con los individuos seleccionados.

prepararSeleccion

`public void prepararSeleccion(Poblacion poblacion).`
Su función es generar el pool con los individuos seleccionados, es decir, con TODOS los individuos de la población. También reinicia el conteo de individuos, esto con el objetivo de poder ser utilizado por el método `seleccion` para seleccionar ordenadamente un nueva población.
Parámetros: `poblacion` - la población cuyos individuos serán seleccionados (todos).

5.2.6. Clase SelectorEstocastico (abstracta)

```
public abstract class SelectorEstocastico.
extends java.lang.Object.
implements Selector.
```

Este nombre describe a un grupo de esquemas de selección probabilísticos que simulan una rueda de ruleta de casino con "rebanadas" en función de los valores esperados de los individuos de la población (ver figura 5.15). A este grupo, por ejemplo, pertenecen la técnica *de la ruleta* y el *muestreo estocástico universal*.

Un constructor de esta clase acepta como parámetro un tipo de función del valor esperado diferente a la típica probabilidad proporcional de la función de aptitud utilizado por Goldberg (1989). Se puede diseñar una función ad-hoc o utilizar alguna de las que brinda esta librería como `FuncionRanking` o `FuncionTanese`.

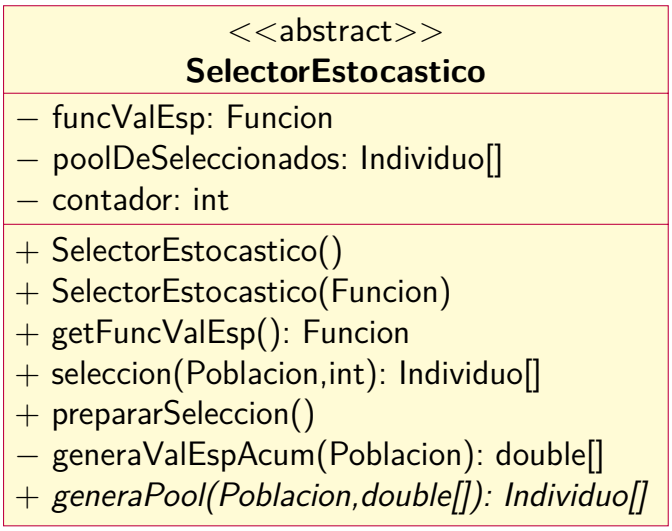


Figura 5.15.: Diagrama - Clase abstracta SelectorEstocastico (elaboración propia).

Constructores

SelectorEstocastico

`public SelectorEstocastico()`.
Constructor por defecto utiliza la función clásica (Goldberg, 1989) de valor esperado basada únicamente en la aptitud de los individuos. Es equivalente a `SelectorEstocastico(new FuncionClasico())`.

SelectorEstocastico

`public SelectorEstocastico(Funcion funcValEsp)`.
Constructor que recibe una función de valor esperado especifica.
Parámetros: `funcValEsp` - la función de valor esperado.

Métodos

getFuncValEsp

`public Funcion getFuncValEsp()`.
Devuelve la función de valor esperado utilizada por este selector.
Retorna: la función de valor esperado utilizada por este selector.

seleccion

```
public Individuo[] seleccion(Poblacion poblacion,
                             int cantidadSeleccionados).
```

Selecciona una cantidad específica de individuos de la población según la técnica estocástica que herede de esta clase al implementar el método `generaValEspAcum`. Los individuos provienen de un pool (arreglo) con todos los seleccionados de esta población. Dicho pool es generado por el método `prepararSeleción`.

Este método es llamado por `generar` de `Generador` cada vez que se deseen seleccionar individuos con oportunidades de cruce.
Parámetros: `poblacion` - la población de donde serán seleccionados los individuos.
`cantidadSeleccionados` - la cantidad de individuos a seleccionar.
Retorna: un arreglo de tamaño `cantidadSeleccionados` con los individuos seleccionados.

prepararSeleccion

`public void prepararSeleccion(Poblacion poblacion)`.
Este método genera un pool de individuos seleccionados según la técnica estocástica que herede de esta clase al implementar el método `generaValEspAcum`.

Este método es llamado por `ejecutar` de la clase `Ejecucion` una vez por generación, dado que en cada generación se debe preparar los datos que utilizará la siguiente generación.
Parámetros: `poblacion` - la población a la que se le aplicará la técnica estocástica.

generaPool

```
public abstract Individuo[] generaPool(
                Poblacion poblacion, double[] valEspAcum).
```

Devuelve un arreglo de tipo `Individuo` que representa el pool de los individuos seleccionados. La forma en que se seleccionen los individuos en este método define la técnica estocástica de la clase que herede de esta clase. Por ejemplo, la clase que implementa este método usando en criterio de selección *de la ruleta* se denomina `SelectorRuleta`.
Parámetros: `poblacion` - la población de donde se seleccionarán los individuos.
`valEspAcum` - el arreglo con los valores esperados acumulados calculados en base a la función del valor esperado pasada como parámetro en el constructor de esta clase.
Retorna: el pool de individuos seleccionados.

5.2.7. Clase SelectorRuleta

```
public class SelectorRuleta.
extends SelectorEstocastico.
```

Es un selector estocástico que aplica el método *de la ruleta* (Goldberg, 1989) para la selección de individuos (ver figura 5.16). Es una técnica de selección que consiste en asignar una "rebanada" de una rueda de ruleta de casino a un individuo. La ruleta se hace girar (generando un número aleatorio) N veces, donde N representa el número de individuos seleccionados con chance de reproducirse. En cada giro, el individuo asignado a la rebanada seleccionada es elegido como un padre.

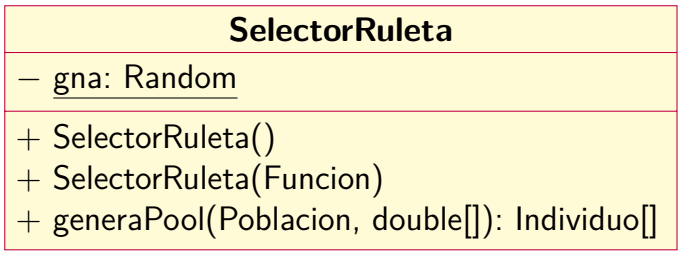


Figura 5.16.: Diagrama - Clase SelectorRuleta (elaboración propia).

La "rebanada" asignada a cada individuo está en función del valor esperado de este. En la versión original de Goldberg este valor es proporcional y depende sólo de la aptitud del individuo en analogía con la idea de individuos más aptos tienen mayores oportunidades de sobrevivir; sin embargo otros trabajos (Tanese, 1989) han utilizado métodos de *escalamiento* que toman en cuenta, no solo la función de aptitud, sino también las medias y las desviaciones estándares. También se han utilizado funciones no proporcionales (Baker, 1985) como en la selección *por jerarquías* (*ranking*) que no dependen directamente de la aptitud.
Ver también: SelectorEstocastico

Constructores

SelectorRuleta

`public SelectorRuleta()`.
Constructor por defecto utiliza la función clásica (Goldberg, 1989) de valor esperado basada únicamente en la aptitud de los individuos. Es equivalente a `SelectorRuleta(new FuncionClasico())`.

SelectorRuleta

`public SelectorRuleta(Funcion funcValEsp)`.
Constructor que recibe una función de valor esperado especifica.

Parámetros: `funcValEsp` - la función de valor esperado.

Métodos

generaPool

```
public Individuo[] generaPool(Poblacion poblacion,
                              double[] valEspAcum).
```

Devuelve un arreglo de tipo `Individuo` que representa el pool de los individuos seleccionados utilizando el método *de la ruleta*.

Parámetros: `poblacion` - la población de donde se seleccionarán los individuos.
`valEspAcum` - el arreglo con los valores esperados acumulados calculados en base a la función del valor esperado pasada como parámetro en el constructor de esta clase.

Retorna: el pool de individuos seleccionados.

5.2.8. Clase SelectorSUS

```
public class SelectorSUS.
extends SelectorEstocastico.
```

Es un selector estocástico que aplica el método de *muestreo estocástico universal* (*stochastic universal sampling*, *SUS*, por sus siglas en ingles) de James Baker (1987) para la selección de individuos (ver figura 5.17). Al igual que el método *de la ruleta*, es una técnica de selección que también asigna "rebanadas" a cada individuo en función del valor esperado, pero en vez de girar la rueda de ruleta N veces, solo se hace girar una vez (se genera solo un número aleatorio) utilizando N apuntadores espaciados equidistantemente, la distancia entre cada uno es de Sum/N, donde Sum es igual a la sumatoria de todos los valores esperados de los individuos de la población y N es el número de individuos seleccionados con chance de reproducirse.

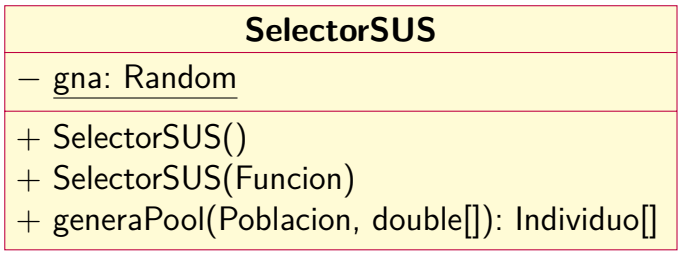


Figura 5.17.: Diagrama - Clase SelectorSUS (elaboración propia).

La "rebanada" asignada a cada individuo está en función del valor esperado de este. Este valor puede ser proporcional y depender sólo de la aptitud del individuo como en *la ruleta* clásica de Goldberg (1989) o puede ser un valor escalado como en el *escalamiento sigma* de Tanese (1989) donde toman en cuenta, no solo la función de aptitud, sino también las medias y las desviaciones estándares. También se puede utilizar funciones no proporcionales (Baker, 1985) como en la selección *por jerarquías* (*ranking*) que no dependen directamente de la aptitud.
Ver también: SelectorEstocastico.

Métodos

generaPool

```
public Individuo[] generaPool(Poblacion poblacion,
                              double[] valEspAcum).
```

Devuelve un arreglo de tipo `Individuo` que representa el pool de los individuos seleccionados utilizando la técnica de *muestreo estocástico universal*.

Parámetros: `poblacion` - la población de donde se seleccionarán los individuos.
`valEspAcum` - el arreglo con los valores esperados acumulados calculados en base a la función del valor esperado pasada como parámetro en el constructor de esta clase.

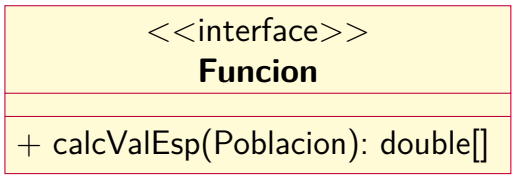


Figura 5.18.: Diagrama - Interfaz Funcion (elaboración propia).

Retorna: el pool de individuos seleccionados.

5.2.9. Interfaz Funcion

`public interface Funcion.`
Esta interfaz es utilizada por los selectores estocásticos, por ejemplo `SelectorRuleta` y `SelectorSUS`, para la definición de la función de valor esperado de sus individuos (ver figura 5.18). La clase que implemente esta interfaz debe desarrollar una función que determine todos los valores esperados de los individuos de la población pasada como parámetro.

La forma clásica de la función de valor esperado utilizado por Goldberg (1989) consiste en una función directamente proporcional a la función de aptitud y que solo depende de esta; sin embargo otros trabajos (Tanese, 1989) han utilizado métodos de *escalamiento* que toman en cuenta, no solo la función de aptitud, sino también las medias y las desviaciones estándares. También se han utilizado funciones no proporcionales (Baker, 1985) como en la selección *por jerarquías* (*ranking*) que no dependen directamente de la aptitud.

calcValEsp

`double[] calcValEsp(Poblacion poblacion).`
Devuelve un arreglo con todos los valores esperados de los individuos

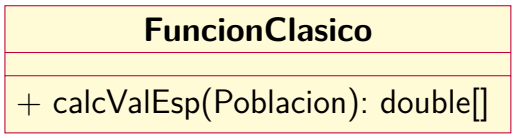


Figura 5.19.: Diagrama - Clase FuncionClasico (elaboración propia).

de la población pasada como parámetro.
Parámetros: `poblacion` - la población a cuyos individuos se les calcularán la función de valor esperado.
Retorna: un arreglo con todos los valores esperados de los individuos de la población.

5.2.10. Clase FuncionClasico

```
public class FuncionClasico.
extends java.lang.Object.
implements Funcion.
```

Esta clase representa la función clásica de valor esperado utilizado por David Goldberg (1989) (ver figura 5.19). Dicha función no es más que la función de la aptitud de cada individuo de la población.
Ver también: Funcion (interfaz).

Métodos

calcValEsp

```
public double[] calcValEsp(Poblacion poblacion).
```

Devuelve un arreglo de números reales positivos que representan los valores esperados de los individuos de la población pasada como parámetro. Dichos valores esperados corresponden con las aptitudes de cada individuo (Goldberg, 1989).

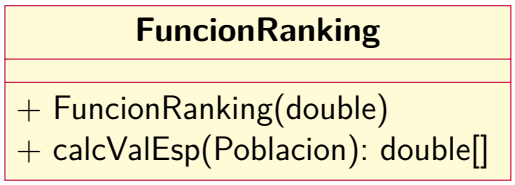


Figura 5.20.: Diagrama - Clase FuncionRanking (elaboración propia).

Parámetros: `poblacion` - la población a cuyos individuos se les calcularán la función de valor esperado.

Retorna: un arreglo con todos los valores esperados de los individuos de la población.

5.2.11. Clase FuncionRanking

```
public class FuncionRanking.
extends java.lang.Object.
implements Funcion.
```

Esta clase permite el cálculo de valores esperados en función de las *jerarquías* (*ranking*)de los individuos dentro de la población (ver figura 5.20). Dichas jerarquías están basadas en las aptitudes de los individuos. Consiste en una función lineal (ver ecuación 2.1) propuesta por James Baker (1985) que utiliza dos constantes: Max determina el valor esperado del individuo ubicado en la jerarquía más alta ($jerarquía = N$, con N = tamaño de la población); y Min el valor del individuo con la jerarquía más baja ($jerarquía = 1$). Baker recomienda usar Max en 1.1 y hacer $Min = 2{,}0 - Max$.

Ver también: Funcion (interfaz).

Constructores

FuncionRanking

`public FuncionRanking(double max).`
Constructor que acepta como parámetro el valor de la constante Max utilizado en la función de jerarquías lineal de Baker (1985).
Parámetros: `max` - constante. Valor del máximo valor para esta función, es decir, el valor del mejor individuo en la jerarquía.

Métodos

calcValEsp

`public double[] calcValEsp(Poblacion poblacion).`
Devuelve un arreglo de números reales positivos que representan los valores esperados de los individuos de la población pasada como parámetro. Dichos valores esperados corresponden con la función de valor esperado por jerarquías lineal de Baker (1985).
Parámetros: `poblacion` - la población a cuyos individuos se les calcularán la función de valor esperado.
Retorna: un arreglo con todos los valores esperados de los individuos de la población.

5.2.12. Clase FuncionTanese

```
public class FuncionTanese.
extends java.lang.Object.
implements Funcion.
```
Esta clase implementa la función de valor esperado con escalamiento sigma presentada por Reiko Tanese (1989) (ver figura 5.21). Consiste en

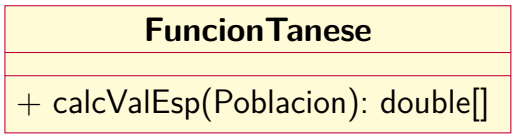


Figura 5.21.: Diagrama - Clase FuncionTanese (elaboración propia).

una función proporcional (ver ecuación 2.3) que utiliza tres variables: la función de aptitud, la media y la desviación estándar de la población.
Ver también: Funcion (interfaz).

Métodos

calcValEsp

`public double[] calcValEsp(Poblacion poblacion).`
Devuelve un arreglo de números reales que representan los valores esperados de los individuos de la población pasada como parámetro. Dichos valores están calculados con la función de escalamiento sigma utilizada por Tanese (1989).
Parámetros: `poblacion` - la población a cuyos individuos se les calcularán la función de valor esperado.
Retorna: un arreglo con todos los valores esperados de los individuos de la población.

5.2.13. Clase SelectorPostCruceSoloHijos

`public class SelectorPostCruceSoloHijos.`
`extends java.lang.Object.`
`implements SelectorPostCruce.`
Es el tipo clásico de selección post cruce (ver figura 5.22. Este tipo de selección es el clásico utilizado por David Goldberg (1989) en su

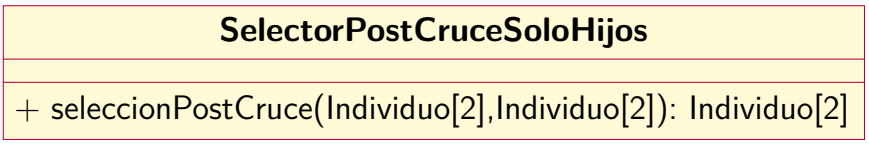


Figura 5.22.: Diagrama - Clase SelectorPostCruceSoloHijos (elaboración propia).

libro *Genetic algorithms in search, optimization, and machine learning (Algoritmos Genéticos en Búsqueda, Optimización y Aprendizaje de Máquinas)*, en el que solo los hijos pasan a la siguiente generación, los padres son desechados independientemente de su aptitud, lo que abre la posibilidad de tener descendientes menos aptos que sus padres y por tanto la convergencia hacia alguna solución pudiera ser más lenta.
Ver también: SelectorPostCruce.

Métodos

seleccionPostCruce

```
Individuo[] seleccionPostCruce(
                    Individuo[] padres, Individuo[] hijos).
```

Este método solo selecciona a los hijos productos del cruce entre los padres.
Parámetros: `padres` - los 2 individuos que se cruzan.
`hijos` - los 2 individuos productos del cruce.
Retorna: los 2 individuos seleccionados para la siguiente generación.

5.2.14. SelectorPostCruceTaigeto

```
public class SelectorPostCruceTaigeto.
extends java.lang.Object.
```

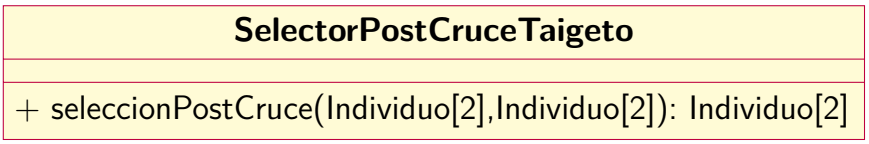


Figura 5.23.: Diagrama - Clase SelectorPostCruceTaigeto (elaboración propia).

`implements SelectorPostCruce.`
Este tipo de selección es el utilizado por Díaz & Hernández (2010) en el Trabajo de Grado *Desarrollo de un Algoritmo Genético para la secuencia de tareas en ambientes Job-Shop en el taller de mecanización de la empresa metalmecánica Moldes Mafra* (Universidad de Carabobo) (ver figura 5.23). Consiste en seleccionar a los dos mejores individuos entre los dos padres y los dos hijos basados el valor de la función de aptitud de cada uno. Esta técnica garantiza descendientes más aptos que sus padres, lo que puede traer como consecuencia una convergencia más rápida hacia alguna solución.

Métodos

seleccionPostCruce

```
Individuo[] seleccionPostCruce(
                    Individuo[] padres, Individuo[] hijos).
```
Este método selecciona a los dos mejores individuos entre dos padres y dos hijos descendientes del cruce de los primeros.
Parámetros: `padres` - los 2 individuos que se cruzan.
`hijos` - los 2 individuos productos del cruce.
Retorna: los 2 individuos seleccionados para la siguiente generación.

Capítulo 6.

Implementación

La librería *agapi* por si misma no es capaz de realizar ninguna tarea. Es simplemente una librería con clases que sirven para desarrollar una aplicación. Luego, se dice que esta aplicación *implementa* la librería *agapi*.

Este capítulo explica paso a paso la implementación de la librería *agapi* para el desarrollo de una aplicación de algoritmos genéticos capaz de resolver un problema matemático simple cuyo solución óptima es conocida. La simplicidad de tal problema tiene por objeto ilustrar el procedimiento de implementación sin enredarse en los laberintos de la matemática con funciones de aptitud complejas donde puede ser imposible determinar una solución óptima.

6.1. Problema

El problema de ejemplo utilizado consiste en determinar un conjunto (x) de números enteros $(a_1, \ldots, a_{20})$ diferentes entre si $(a_i \neq a_{i+1})$ y que estén comprendidos en el rango [1,20] que maximicen la función dada por la ecuación 6.1.

$$f(x) = \sum_{i=1}^{20} a_i \cdot i \tag{6.1}$$

donde

$$\begin{aligned} &x = \{a_1, a_2, \ldots, a_{20}\}; \\ &a_i \in \{1, \ldots, 20\}; \\ &a_i \neq a_{i+1} \end{aligned}$$

Por ejemplo, para la secuencia:

$$x_1 = \{3, 12, 6, 9, 14, 7, 2, 18, 5, 20, 11, 13, 1, 17, 19, 4, 10, 8, 16, 15\}$$

se obtiene el siguiente valor de la función objetivo:

$$\begin{aligned} f(x_1) = 3 \cdot 1 + 12 \cdot 2 + 6 \cdot 3 + 9 \cdot 4 + 14 \cdot 5 \\ + 7 \cdot 6 + 2 \cdot 7 + 18 \cdot 8 + 5 \cdot 9 + 20 \cdot 10 \\ + 11 \cdot 11 + 13 \cdot 12 + 1 \cdot 13 + 17 \cdot 14 \\ + 19 \cdot 15 + 4 \cdot 16 + 10 \cdot 17 + 8 \cdot 18 \\ + 16 \cdot 19 + 15 \cdot 20 = 2.391 \end{aligned}$$

Al observar el problema, se evidencia que el espacio de soluciones corresponde a 20! permutaciones, esto es, 2.432.902.008.176.640.000 posibles soluciones: un espacio demasiado grande para ser explorado solución por solución. Sin embargo, dada la trivialidad del problema, se hace un análisis simple (realizado por humanos) que conduce a la **solución óptima**:

$$x_{optimo} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20\}$$

y su función objetivo:

$$\begin{aligned} f(x_{optimo}) = 1 \cdot 1 + 2 \cdot 2 + 3 \cdot 3 + 4 \cdot 4 + 5 \cdot 5 \\ + 6 \cdot 6 + 7 \cdot 7 + 8 \cdot 8 + 9 \cdot 9 + 10 \cdot 10 \\ + 11 \cdot 11 + 12 \cdot 12 + 13 \cdot 13 + 14 \cdot 14 \\ + 15 \cdot 15 + 16 \cdot 16 + 17 \cdot 17 + 18 \cdot 18 \\ + 19 \cdot 19 + 20 \cdot 20 = 2.870 \end{aligned}$$

Así, se tiene un problema que cumple con características ideales para evaluar el desempeño de la librería *agapi*:

- espacio de soluciones muy grande para computarlo solución por solución,
- solución óptima conocida,
- función objetivo poco compleja

6.2. Procedimiento

6.2.1. Paso 1 - Instalación de la librería

Lo primero que se debe hacer es descargar el archivo comprimido `saulguillermo-agapi-xxxxxx.zip`[1] desde la página web del proyecto *agapi*:

`http://saulguillermo.github.io/agapi`

[1]Este archivo contiene carpetas y archivos relacionados con el proyecto de software *agapi* derivado de este Trabajo. Dichos archivos y carpetas se detallan en el apéndice B. "xxxxxx" representan una secuencia alfanumérica aleatoria determinada por la página web.

Luego, se crea una carpeta con el nombre `ejemplo` y dentro de ella se crea otra carpeta con el nombre `lib`. Ahora se descomprime el archivo `saulguillermo-agapi-xxxxxx.zip`, se abre la carpeta correspondiente `saulguillermo-agapi-xxxxxx`, se copia el archivo `agapi.tar` y se pega dentro de la carpeta `lib`.

Se abre el IDE[2], que en el caso de este Trabajo es *Eclipse*[3] y se crea un nuevo proyecto Java con el nombre `ejemplo` dentro de la carpeta `ejemplo` ya creada y se asocia con el archivo `agapi.tar` (ubicado en la carpeta `lib`) para usarlo como librería[4]

6.2.2. Paso 2 - Creación de clase IndividuoEjemplo

Una vez creado el proyecto `ejemplo` con su respectiva librería asociada (*agapi*), se crea un paquete con el nombre `ejemplo` (el mismo nombre del proyecto). Dentro de este paquete se crea la primera clase llamada `IndividuoEjemplo` y hacemos que esta herede de la clase abstracta `IndividuoCombinatorio` ubicada en la librería *agapi*, para lo cual se debe importar desde el paquete `agapi.impl`:

```
1  package ejemplo;
2
3  import agapi.impl.IndividuoCombinatorio;
4
5  public class IndividuoEjemplo extends IndividuoCombinatorio {
6
7  }
```

[2]Siglas de *integrated development environment*, esto es, *entorno de desarrollo integrado.*

[3]La decisión de qué herramienta de programación utilizar recae en el desarrollador. En la actualidad, las más comunes son los IDEs *Netbeans* y *Eclipse*.

[4]El proceso para instalar librerías a partir de archivos *tar* está bien documentado en la web, independientemente del IDE utilizado.

Luego, dentro del ámbito de la clase `IndividuoEjemplo`, debe implementarse el método `calcFA` (único método abstracto de la clase `IndividuoCombinatorio`) con el código correspondiente a la ecuación 6.1:

```
1        @Override
2        public double calcFA() {
3                double fa = 0;
4                int[] s = this.getCromosoma();
5                for (int i = 0; i < s.length; i++) {
6                        fa = (double) fa + (i + 1) * s[i];
7                }
8                return fa;
9        }
```

Ahora, se sobrescribe el método `toString` para darle una representación escrita a la clase `IndividuoEjemplo`. Esta representación es utilizada por la librería agapi para la representación en texto del proceso completo. Aquí se tiene el método `toString`:

```
1        @Override
2        public String toString() {
3                StringBuilder sb = new StringBuilder();
4                int[] s = this.getCromosoma();
5                for (int i = 0; i < s.length; i++) {
6                        if (i != s.length - 1) {
7                                sb.append(s[i] + "-");
8                        } else {
9                                sb.append(s[i]);
10                       }
11               }
12               sb.append(" FA: " + this.getFA());
13               return sb.toString();
14       }
```

Con estos dos métodos se finaliza la clase `IndividuoEjemplo`, ahora sólo falta una clase con un método *main* para ejecutar el algoritmo genético.

6.2.3. Paso 3 - Ejecución del algoritmo

Dentro del paquete `ejemplo` se crea una clase llamada `Principal` y se importan los paquetes de la librería *agapi*. Luego se crea un método `main`[5] con los parámetros del algoritmo genético:

```
1  package ejemplo;
2
3  import agapi.*;
4  import agapi.impl.*;
5
6  public class Principal {
7
8    public static void main(String[] args) {
9      Configuracion c = new Configuracion();
10
11     c.setNumeroEjecuciones(1);
12     c.setNumeroGeneraciones(10);
13     c.setSelector(new SelectorSUS(new FuncionRanking(1.1)));
14     c.setSelectorPostCruce(new SelectorPostCruceTaigeto());
15     c.setProbabilidadCruce(0.7);
16     c.setProbabilidadMutacion(0.05);
17     c.setElitismo(true);
18     c.setTamanoPoblacion(10);
19     c.setTipoIndividuo(new IndividuoEjemplo());
20     c.setTamanoCromosoma(20);
21
22     c.iniciarProceso();
```

[5]Correr un programa significa decirle a la *máquina virtual de java* que cargue la primera clase que contenga un método *main* y luego lo ejecute. El programa se mantendrá en ejecución hasta que se complete todo el código de *main*.

```
23
24      System.out.println(
25        c.aTexto(Configuracion.GENERACIONES_CON_POBLACIONES));
26        double tiempo =
27          c.getProceso().getTiempoProceso() / 1000000000.0;
28        System.out.println("Tiempo: " + tiempo + " segundos");
29    }
30  }
```

Con el código del cuadro anterior se tiene todo lo necesario para ejecutar un algoritmo genético para buscar soluciones al problema planteado utilizando la librería *agapi*. Una ejemplo de la salida de esta aplicación puede verse en el apéndice C.

Las líneas de este método `main` son una receta para el desarrollo de cualquier proceso y los parámetros de los métodos utilizados que determinan la configuración del algoritmo genético pueden ser modificados de muchas formas como se describe en la siguiente sección.

6.2.4. Paso 4 - Manipulación de los parámetros

Una vez observada la corrida exitosa del algoritmo planteada en el código del cuadro anterior se procede a modificar los parámetros del algoritmo para verificar el funcionamiento del resto de las clases del paquete `impl`. A continuación se describen las lineas de código del cuadro de la sección anterior:

La **línea 11** (`c.setNumeroEjecuciones(1);`) se modifica con el número de ejecuciones deseado, por ejemplo:

```
11  c.setNumeroEjecuciones(5);
```

La **línea 12** (`c.setNumeroGeneraciones(10);`) se modifica con otro número de generaciones, por ejemplo:

```
12 c.setNumeroGeneraciones(100);
```

La **línea 13** (`c.setSelector(new SelectorSUS(new FuncionRanking(1.1)));`) se configura con cualquiera de los tipos de *selectores* disponibles en la librería. A continuación varios ejemplos:

```
13 c.setSelector(new SelectorSUS(new FuncionClasico()));
```

```
13 c.setSelector(new SelectorSUS(new FuncionTanese()));
```

```
13 c.setSelector(new SelectorRuleta(new FuncionRanking(1.1)));
```

```
13 c.setSelector(new SelectorRuleta(new FuncionClasico()));
```

```
13 c.setSelector(new SelectorRuleta(new FuncionTanese()));
```

```
13 c.setSelector(new SelectorTorne(0.75));
```

```
13 c.setSelector(new SelectorTodos());
```

La **línea 14** (`c.setSelectorPostCruce(new SelectorPostCruceTaigeto());`) se configura con el otro tipo de *selección post cruce* disponible, por ejemplo:

```
14 c.setSelectorPostCruce(new SelectorPostCruceSoloHijos());
```

Las **líneas 15 y 16** (`c.setProbabilidadCruce(0.7);` y `c.setProbabilidadMutacion(0.05);`) se modifican con cualquier valor real comprendido entre 0 y 1, como por ejemplo:

```
15 c.setProbabilidadCruce(0.85);
16 c.setProbabilidadMutacion(0.25);
```

La **línea 17** (`c.setElitismo(true);`) se modifica con el valor complementario `false`, por ejemplo:

```
17 c.setElitismo(false);
```

La **línea 18** (`c.setTamanoPoblacion(10);`) se modifica con cualquier valor entero mayor a 1, por ejemplo:

```
18 c.setTamanoPoblacion(50);
```

La **línea 19** (`c.setTipoIndividuo(new IndividuoEjemplo());`) debe tener como parámetro un *individuo* que se haya desarrollado para el problema específico, por tanto su modificación es responsabilidad del usuario. La común sería utilizar una palabra compuesta como "IndividuoXXX" donde "XXX" lo decide el desarrollador, por ejemplo:

```
19 c.setTipoIndividuo(new IndividuoUnNombre());
```

La **línea 20** (`c.setTamanoCromosoma(20);`) en el caso específico de este problema no debe modificarse, sin embargo es muy común encontrar problemas donde este parámetro puede cambiarse con cualquier número entero mayor a cero, por ejemplo:

```
20 c.setTamanoCromosoma(10);
```

La **línea 22** (`c.iniciarProceso();`) es indispensable para dar inicio a un algoritmo genético, así como es indispensable crear un objeto de tipo `Configuracion` (**línea 9**) que permite el llamado a los métodos anteriores escritos en las líneas anteriores.

Las **líneas 24-25** presentan en pantalla el contenido facilitado por el método `atexto` con la opción `GENERACIONES_CON_POBLACIONES` pero también se utilizan dos opciones adicionales:

```
24 System.out.println(
25 c.aTexto(Configuracion.GENERACIONES_SIN_POBLACIONES));
```

```
24 System.out.println(c.aTexto(Configuracion.ULTIMA_GENERACION));
```

Las **líneas 26-28** presentan en pantalla el tiempo total que tarda el *proceso* en completarse (método `getTiempoProceso`) pero también se puede mostrar el tiempo que tarda cada ejecución insertando el siguiente bloque de código al final del método `main`:

```
Ejecucion[] ejecuciones = c.getProceso().getEjecuciones();
double tiempoEjecucion;
for (int i = 0; i < Proceso.getNumeroEjecuciones(); i++) {
        tiempoEjecucion = ejecuciones[i].getTiempoEjecucion()
                          / 1000000000.0;
        System.out.println("Tiempo de Ejecucion " + i + ": "
                         + tiempoEjecucion + " segundos");
}
```

6.3. Un mayor nivel de complejidad

Si se desea un mayor nivel de complejidad se debe hacer un uso menos intensivo de la librería *agapi*. Un caso complejo, por ejemplo, requeriría sobrescribir todos los métodos abstractos de la clase `Individuo`, dejando de lado a las clases `IndividuoBinario` e `IndividuoCombinatorio`.

Posiblemente el método `calcFA` necesitaría clases auxiliares si su nivel del complejidad lo exige. Quizás también sea necesario desarrollar un nuevo tipo de *selector*, si las técnicas de selección de *agapi* no son suficientes.

Conclusiones

El problema planteado en la sección 6.1 y su solución por medio de la aplicación desarrollada en la sección 6.2 representan un caso trivial, lejos del tipo de problemas que se pueden encontrar en el mundo real. Sin embargo y, como es lógico, se trata de un problema válido desde el punto de vista práctico, ideal para la comprobación, evaluación y afinamiento de la librería *agapi*.

Las corridas de la librería no arrojaron ningún error ni advertencias en tiempo de ejecución así como tampoco advertencias o errores de compilación, por lo que se puede concluir que la librería *agapi* funciona sin fallas en su arquitectura.

La implementación, del individuo para el caso específico del problema planteado en el capítulo 6, así como la simplicidad en la configuración de un algoritmo genético mostrada en el cuadro de código la sección 6.2.3 y la facilidad con la que cada parámetro puede modificarse (ver sección 6.2.4) permiten llegar a la conclusión que la librería *agapi* es flexible y de fácil implementación.

La disponibilidad en una página web (ver sección 6.2.1), la garantía de una licencia de software libre como la GPL (ver apéndice A) con el suministro del código fuente de la librería y el permiso de modificación y distribución permiten decir que *agapi* es una librería extensible, mejorable y adaptable a problemas más específicos de algoritmos genéticos.

Dado que esta librería esta basada en conocimientos académicamente probados y aceptados que se han vuelto hasta cierto punto “clásicos” la

posibilidad de implementar aplicaciones con algoritmos novedosas implica la modificación del núcleo de la librería, es decir el paquete `agapi` lo que sin duda requiere de un claro conocimiento de la arquitectura del software. Esto lleva a concluir que esta librería no es apta para soluciones muy específicas con algoritmos novedosos o poco documentados académicamente.

En fin, en la medida en que un problema (o un investigador) se vuelva más demandante de técnicas específicas, más insuficiente se torna la librería *agapi*, basada en lo académicamente probado y aceptado. Por el contrario si el deseo es utilizar métodos probados como por ejemplo, la técnica *de la ruleta*, *agapi* puede ser muy adaptable y eficiente en términos de desarrollo.

Recomendaciones

- Implementar el entorno en el desarrollo de aplicaciones que resuelvan problemas de ingeniería u otras áreas, especialmente en la Facultad de Ingeniería de la Universidad de Carabobo. Los estudiantes que tengan ahora que desarrollar una herramienta de optimización en algoritmos genéticos podrán programar bajo este entorno de software libre sin los problemas de derechos de autor.
- Ampliar, profundizar y renovar los elementos del entorno, es decir, no permitir que se vuelve un proyecto estático acobijado solo por este Trabajo, sino en algo dinámico con la contribución de más investigadores y desarrolladores, especialmente de las áreas de ingeniería y computación de la Universidad de Carabobo. Para ello se recomienda utilizar como mecanismo de trabajo conjunto el portal web del proyecto y el programa Git que permitan un ambiente eficiente para el desarrollo del software.
- Difundir el entorno por medios académicos como revistas u otras publicaciones, y por medios electrónicos (internet) como una alternativa de software libre con la finalidad de captar investigadores que deseen implementar el entorno y desarrolladores que lo modifiquen. Se piensa que esta librería puede ser una herramienta facilitadora desde el punto de vista académico para la mejor comprensión del uso de las metaheurísticas de optimización en la investigación de operaciones como los algoritmos genéticos.

Apéndice A.

Autorización del uso de licencia

Al inicio de cada clase de la librería del proyecto *agapi* se encuentra la autorización del uso de la General Public License (GPL) versión 3, esto es la Licencia Pública General, otorgada por la Free Software Foundation, Inc. (en español, Fundación del Software Libre). Una copia de la licencia se incluye dentro del proyecto *agapi*, también se puede obtener una copia de esta licencia en inglés en `http://www.gnu.org/licenses/`, además en internet se pueden hallar traducciones, aunque la Free Software Foundation no autoriza ninguna de ellas por motivos legales.

En resumen, en la autorización ubicada en el preámbulo de cada clase indica lo siguiente:

1. el nombre del proyecto (este item no es obligatorio),
2. el nombre del autor y el año de producción haciendo uso de la palabra “Copyright”,
3. el uso de la licencia GPL en su versión 3 o posterior,
4. la disponibilidad de una copia de la licencia dentro del proyecto,
5. si no se ofrece una copia, esta se puede obtener en la dirección web: `http://www.gnu.org/licenses/`

El preámbulo se encuentra en idioma ingles y dice **textualmente** lo siguiente:

```
/*
 * AGAPI - API para el desarrollo de Algoritmos Geneticos
 * Copyright (C) 2013 Saul Gonzalez
 *
 * This library is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program.  If not, see  <http://www.gnu.org/licenses/>.
 */
```

Una **traducción** de este preámbulo pudiera ser la siguiente:

```
/*
 * AGAPI - API para el desarrollo de Algoritmos Geneticos
 * Copyright (C) 2013 Saul Gonzalez
 *
 * Esta libreria es software libre: usted puede redistribuirlo y/o
 * modificarlo bajo los terminos de la Licencia Publica General como
 * esta publicada por la Fundacion del Software Libre, tanto en la
 * version 3 de la licencia, como (si es su decision) en cualquier
 * version posterior.
 *
 * Este libreria se distribuye con la esperanza de poder ser util, pero
 * SIN NINGUNA GARANTIA; sin ni siquiera la GARANTIA DE MERCANTIBILIDAD
 * o RENTABILIDAD PARA ALGUN PROPOSITO PARTICULAR. Vea la Licencia
 * Publica General para mas detalles.
 *
 * Usted debe recibir una copia de la Licencia Publica General junto con
 * este programa. De lo contrario, vea <http://www.gnu.org/licenses/>.
 */
```

Apéndice B.

Contenido del archivo saulguillermo-agapi-xxxxxx.zip

El archivo `saulguillermo-agapi-xxxxxx.zip` contiene carpetas y archivos relacionados con el proyecto de software *agapi* derivado de este Trabajo. Dichos archivos y carpetas se describen brevemente a continuación:

1. **`src`**: directorio que contiene el código fuente escrito en el lenguaje Java.
2. **`doc`**: directorio con la documentación en formato *javadocs*.
3. **`agapi.tar`**: archivo binario comprimido que permite utilizar la librería *agapi* directamente en un IDE.
4. **`README.md`**: archivo de texto con información del proyecto *agapi*.
5. **`COPYING.txt`**: archivo de texto con la licencia GPL version 3 (ver apéndice A)
6. **`.project`**: archivo de texto oculto generado por el IDE Eclipse.
7. **`.gitignore`**: archivo de texto oculto que indica al programa *Git* que tipos de archivos omitir en su control.
8. **`tesis.pdf`**: archivo pdf de este documento.

Apéndice C.

Salida de la aplicación implementada

A continuación se muestra un ejemplo de una salida generada por la aplicación implementada en el capítulo 6:

```
DATOS DEL PROCESO
Tipo de Individuo:     ejemplo.IndividuoEjemplo
No de Ejecuciones:     1
Metodo de Seleccion:   Muestreo Estocastico Universal con
                       Ranking (Baker 1985) con Max=1.1 y
                       Min=2-Max=0.8999999999999999
Seleccion Post Cruce:  Taigeto
Tamaño Cromosoma:      20
Tamaño de Poblacion:   10
No de Generaciones:    10
Probabilidad Cruce:    0.7
Probabilidad Mutacion: 0.05
Utilizar Elitismo?     true


______________EJECUCION 1______________

GENERACION 1
 Mejor: 8-10-4-1-9-2-14-6-11-5-17-15-12-3-20-16-13-19-7-18 FA: 2573.0
 Peor:  10-19-9-8-17-4-20-3-18-2-16-11-7-13-5-1-14-15-6-12 FA: 2096.0
 Media:  2,362.10
```

```
 Desviacion:  138.30
  1. 10-19-9-8-17-4-20-3-18-2-16-11-7-13-5-1-14-15-6-12 FA: 2096.0
  2. 11-13-14-8-4-17-15-9-6-5-10-7-16-1-12-3-19-18-20-2 FA: 2225.0
  3. 11-18-10-5-7-6-13-17-4-3-14-1-19-16-12-2-15-9-8-20 FA: 2279.0
  4. 7-6-14-15-5-2-8-20-3-19-17-1-18-12-16-9-11-4-10-13 FA: 2285.0
  5. 5-6-10-4-7-13-12-1-18-15-20-17-14-3-2-16-19-8-11-9 FA: 2370.0
  6. 10-2-11-4-14-9-13-16-7-1-15-3-20-17-12-5-6-19-18-8 FA: 2384.0
  7. 11-2-1-12-17-5-4-19-3-9-15-18-6-16-20-8-13-14-10-7 FA: 2392.0
  8. 7-2-8-12-5-11-10-6-13-1-20-4-17-15-16-3-18-14-19-9 FA: 2505.0
  9. 9-10-4-7-12-1-13-3-17-2-14-8-16-11-6-20-15-5-18-19 FA: 2512.0
 10. 8-10-4-1-9-2-14-6-11-5-17-15-12-3-20-16-13-19-7-18 FA: 2573.0

GENERACION 2
 Mejor: 8-10-4-1-9-2-14-6-11-5-17-15-12-3-20-16-13-19-7-18 FA: 2573.0
 Peor:  10-19-9-8-17-4-20-3-18-2-16-11-7-13-5-1-14-15-6-12 FA: 2096.0
 Media:  2,398.90
 Desviacion:  139.43
  1. 10-19-9-8-17-4-20-3-18-2-16-11-7-13-5-1-14-15-6-12 FA: 2096.0
  2. 11-6-14-8-4-17-15-9-13-5-10-7-16-1-12-3-19-18-20-2 FA: 2274.0
  3. 7-6-14-15-5-2-8-20-3-19-17-1-18-12-16-9-11-4-10-13 FA: 2285.0
  4. 5-6-10-4-7-13-12-1-18-15-20-17-14-3-2-16-19-8-11-9 FA: 2370.0
  5. 10-2-11-4-14-9-13-16-7-1-15-3-20-17-12-5-6-19-18-8 FA: 2384.0
  6. 11-2-8-12-5-7-10-6-13-1-20-4-17-15-16-3-18-14-19-9 FA: 2485.0
  7. 7-2-8-12-5-11-10-6-13-1-20-4-17-15-16-3-18-14-19-9 FA: 2505.0
  8. 7-2-8-12-5-11-10-6-13-1-20-4-17-15-16-3-18-14-19-9 FA: 2505.0
  9. 9-10-4-7-12-1-13-3-17-2-14-8-16-11-6-20-15-5-18-19 FA: 2512.0
 10. 8-10-4-1-9-2-14-6-11-5-17-15-12-3-20-16-13-19-7-18 FA: 2573.0

GENERACION 3
 Mejor: 8-10-4-1-9-2-14-6-11-5-17-15-12-3-20-16-13-19-7-18 FA: 2573.0
 Peor:  10-19-9-8-17-4-20-3-18-2-16-11-7-13-5-1-14-15-6-12 FA: 2096.0
 Media:  2,428.60
 Desviacion:  135.83
  1. 10-19-9-8-17-4-20-3-18-2-16-11-7-13-5-1-14-15-6-12 FA: 2096.0
  2. 7-6-14-15-5-2-8-20-3-19-17-1-18-12-16-9-11-4-10-13 FA: 2285.0
  3. 5-6-10-4-7-13-12-1-18-15-20-17-14-3-2-16-19-8-11-9 FA: 2370.0
  4. 11-2-8-16-5-7-10-6-13-1-20-4-15-17-12-3-18-14-19-9 FA: 2443.0
  5. 11-2-8-12-5-7-10-6-13-1-20-4-17-15-16-3-18-14-19-9 FA: 2485.0
  6. 7-2-8-12-5-11-10-6-13-1-20-4-17-15-16-3-18-14-19-9 FA: 2505.0
  7. 7-2-8-12-5-11-10-6-13-1-20-4-17-15-16-3-18-14-19-9 FA: 2505.0
```

```
  8. 9-10-4-7-12-1-13-3-17-2-14-8-16-11-6-20-15-5-18-19 FA: 2512.0
  9. 9-10-4-7-12-1-13-3-17-2-14-8-16-11-6-20-15-5-18-19 FA: 2512.0
 10. 8-10-4-1-9-2-14-6-11-5-17-15-12-3-20-16-13-19-7-18 FA: 2573.0

GENERACION 4
 Mejor: 8-10-4-1-9-2-14-6-11-5-17-15-12-3-20-16-13-19-7-18 FA: 2573.0
 Peor:  10-19-9-8-17-4-20-3-18-2-16-11-7-13-5-1-14-15-6-12 FA: 2096.0
 Media:  2,441.90
 Desviacion:  129.73
  1. 10-19-9-8-17-4-20-3-18-2-16-11-7-13-5-1-14-15-6-12 FA: 2096.0
  2. 5-6-10-4-7-13-12-1-18-15-20-17-14-3-2-16-19-8-11-9 FA: 2370.0
  3. 5-6-10-4-7-13-12-1-18-15-20-17-3-8-16-9-11-19-14-2 FA: 2374.0
  4. 11-2-8-12-5-7-10-6-13-1-20-4-17-15-16-3-18-14-19-9 FA: 2485.0
  5. 11-2-8-12-5-7-10-6-13-1-20-4-15-17-16-3-18-14-19-9 FA: 2487.0
  6. 7-2-8-12-5-11-10-6-13-1-20-4-17-15-16-3-18-14-19-9 FA: 2505.0
  7. 7-2-8-12-5-11-10-6-13-1-20-4-17-15-16-3-18-14-19-9 FA: 2505.0
  8. 9-10-4-7-12-1-13-3-17-2-14-8-16-11-6-20-15-5-18-19 FA: 2512.0
  9. 9-10-4-7-12-1-13-3-17-2-14-8-16-11-6-20-15-5-18-19 FA: 2512.0
 10. 8-10-4-1-9-2-14-6-11-5-17-15-12-3-20-16-13-19-7-18 FA: 2573.0

GENERACION 5
 Mejor: 7-2-4-12-5-11-10-6-13-1-14-8-17-15-16-3-18-20-19-9 FA: 2583.0
 Peor:  5-6-10-4-7-13-12-1-18-15-20-17-14-3-2-16-19-8-11-9 FA: 2370.0
 Media:  2,521.50
 Desviacion:  61.33
  1. 5-6-10-4-7-13-12-1-18-15-20-17-14-3-2-16-19-8-11-9 FA: 2370.0
  2. 11-2-8-12-5-7-10-6-13-1-20-4-17-15-16-3-18-14-19-9 FA: 2485.0
  3. 7-2-8-12-5-11-10-6-13-1-20-4-17-15-16-3-18-14-19-9 FA: 2505.0
  4. 7-2-8-12-5-11-10-6-13-1-20-4-15-17-16-3-18-14-19-9 FA: 2507.0
  5. 9-10-4-7-12-1-13-3-17-2-14-8-16-11-6-20-15-5-18-19 FA: 2512.0
  6. 5-6-10-2-7-13-12-1-11-8-20-4-17-15-16-3-18-14-19-9 FA: 2525.0
  7. 8-10-4-1-9-2-14-6-11-5-17-15-12-3-20-16-13-19-7-18 FA: 2573.0
  8. 8-10-4-1-9-2-14-6-11-5-17-15-12-3-20-16-13-19-7-18 FA: 2573.0
  9. 9-10-4-2-6-1-8-11-14-5-17-15-12-3-20-16-13-19-7-18 FA: 2582.0
 10. 7-2-4-12-5-11-10-6-13-1-14-8-17-15-16-3-18-20-19-9 FA: 2583.0

GENERACION 6
 Mejor: 7-2-4-12-5-11-10-6-13-1-14-8-17-15-16-3-18-20-19-9 FA: 2583.0
 Peor:  11-2-8-12-5-7-10-6-13-1-20-4-17-15-16-3-18-14-19-9 FA: 2485.0
 Media:  2,543.00
```

```
 Desviacion:  37.10
  1. 11-2-8-12-5-7-10-6-13-1-20-4-17-15-16-3-18-14-19-9 FA: 2485.0
  2. 7-2-8-12-5-11-10-6-13-1-20-4-15-17-16-3-18-14-19-9 FA: 2507.0
  3. 7-2-8-12-5-11-10-6-13-1-20-4-15-17-16-3-18-14-19-9 FA: 2507.0
  4. 9-10-4-7-12-1-13-3-17-2-14-8-16-11-6-20-15-5-18-19 FA: 2512.0
  5. 5-6-10-2-7-13-12-1-11-8-20-4-17-15-16-3-18-14-19-9 FA: 2525.0
  6. 8-10-4-1-9-2-14-6-11-5-17-15-12-3-20-16-13-19-7-18 FA: 2573.0
  7. 8-10-4-1-9-2-14-6-11-5-17-15-12-3-20-16-13-19-7-18 FA: 2573.0
  8. 9-10-4-2-6-1-8-11-14-5-17-15-12-3-20-16-13-19-7-18 FA: 2582.0
  9. 7-2-4-12-5-11-10-6-13-1-14-8-17-15-16-3-18-20-19-9 FA: 2583.0
 10. 7-2-4-12-5-11-10-6-13-1-14-8-17-15-16-3-18-20-19-9 FA: 2583.0

GENERACION 7
 Mejor: 7-2-4-12-5-11-10-6-13-1-14-8-17-15-16-3-18-20-19-9 FA: 2583.0
 Peor:  7-2-8-12-5-11-10-6-13-1-20-4-15-17-16-3-18-14-19-9 FA: 2507.0
 Media:  2,552.80
 Desviacion:  33.22
  1. 7-2-8-12-5-11-10-6-13-1-20-4-15-17-16-3-18-14-19-9 FA: 2507.0
  2. 7-2-8-12-5-11-10-6-13-1-20-4-15-17-16-3-18-14-19-9 FA: 2507.0
  3. 9-10-4-7-12-1-13-3-17-2-14-8-16-11-6-20-15-5-18-19 FA: 2512.0
  4. 5-6-10-2-7-13-12-1-11-8-20-4-17-15-16-3-18-14-19-9 FA: 2525.0
  5. 8-10-4-1-9-2-14-6-11-5-17-15-12-3-20-16-13-19-7-18 FA: 2573.0
  6. 8-10-4-1-9-2-14-6-11-5-17-15-12-3-20-16-13-19-7-18 FA: 2573.0
  7. 9-10-4-2-6-1-8-11-14-5-17-15-12-3-20-16-13-19-7-18 FA: 2582.0
  8. 7-2-4-12-5-11-10-6-13-1-14-8-17-15-16-3-18-20-19-9 FA: 2583.0
  9. 7-2-4-12-5-11-10-6-13-1-14-8-17-15-16-3-18-20-19-9 FA: 2583.0
 10. 7-2-4-12-5-11-10-6-13-1-14-8-17-15-16-3-18-20-19-9 FA: 2583.0

GENERACION 8
 Mejor: 7-2-4-12-5-11-10-6-13-1-14-8-17-15-16-3-18-20-19-9 FA: 2583.0
 Peor:  7-2-8-12-5-11-10-6-13-1-20-4-15-17-16-3-18-14-19-9 FA: 2507.0
 Media:  2,560.40
 Desviacion:  30.45
  1. 7-2-8-12-5-11-10-6-13-1-20-4-15-17-16-3-18-14-19-9 FA: 2507.0
  2. 9-10-4-7-12-1-13-3-17-2-14-8-16-11-6-20-15-5-18-19 FA: 2512.0
  3. 5-6-10-2-7-13-12-1-11-8-20-4-17-15-16-3-18-14-19-9 FA: 2525.0
  4. 8-10-4-1-9-2-14-6-11-5-17-15-12-3-20-16-13-19-7-18 FA: 2573.0
  5. 8-10-4-1-9-2-14-6-11-5-17-15-12-3-20-16-13-19-7-18 FA: 2573.0
  6. 9-10-4-2-6-1-8-11-14-5-17-15-12-3-20-16-13-19-7-18 FA: 2582.0
  7. 7-2-4-12-5-11-10-6-13-1-14-8-17-15-16-3-18-20-19-9 FA: 2583.0
```

```
  8. 7-2-4-12-5-11-10-6-13-1-14-8-17-15-16-3-18-20-19-9 FA: 2583.0
  9. 7-2-4-12-5-11-10-6-13-1-14-8-17-15-16-3-18-20-19-9 FA: 2583.0
 10. 7-2-4-12-5-11-10-6-13-1-14-8-17-15-16-3-18-20-19-9 FA: 2583.0

GENERACION 9
 Mejor: 7-2-4-12-5-11-10-6-13-1-14-8-17-15-16-3-18-20-19-9 FA: 2583.0
 Peor:  7-2-8-12-5-11-10-6-13-1-20-4-15-17-16-3-18-14-19-9 FA: 2507.0
 Media:  2,560.40
 Desviacion:  30.45
  1. 7-2-8-12-5-11-10-6-13-1-20-4-15-17-16-3-18-14-19-9 FA: 2507.0
  2. 9-10-4-7-12-1-13-3-17-2-14-8-16-11-6-20-15-5-18-19 FA: 2512.0
  3. 5-6-10-2-7-13-12-1-11-8-20-4-17-15-16-3-18-14-19-9 FA: 2525.0
  4. 8-10-4-1-9-2-14-6-11-5-17-15-12-3-20-16-13-19-7-18 FA: 2573.0
  5. 8-10-4-1-9-2-14-6-11-5-17-15-12-3-20-16-13-19-7-18 FA: 2573.0
  6. 9-10-4-2-6-1-8-11-14-5-17-15-12-3-20-16-13-19-7-18 FA: 2582.0
  7. 7-2-4-12-5-11-10-6-13-1-14-8-17-15-16-3-18-20-19-9 FA: 2583.0
  8. 7-2-4-12-5-11-10-6-13-1-14-8-17-15-16-3-18-20-19-9 FA: 2583.0
  9. 7-2-4-12-5-11-10-6-13-1-14-8-17-15-16-3-18-20-19-9 FA: 2583.0
 10. 7-2-4-12-5-11-10-6-13-1-14-8-17-15-16-3-18-20-19-9 FA: 2583.0

GENERACION 10
 Mejor: 7-2-4-12-5-11-10-6-13-1-14-8-17-15-16-3-18-20-19-9 FA: 2583.0
 Peor:  7-2-8-12-5-11-10-6-13-1-20-4-15-17-16-3-18-14-19-9 FA: 2507.0
 Media:  2,560.40
 Desviacion:  30.45
  1. 7-2-4-12-5-11-10-6-13-1-14-8-17-15-16-3-18-20-19-9 FA: 2583.0
  2. 7-2-8-12-5-11-10-6-13-1-20-4-15-17-16-3-18-14-19-9 FA: 2507.0
  3. 9-10-4-7-12-1-13-3-17-2-14-8-16-11-6-20-15-5-18-19 FA: 2512.0
  4. 5-6-10-2-7-13-12-1-11-8-20-4-17-15-16-3-18-14-19-9 FA: 2525.0
  5. 8-10-4-1-9-2-14-6-11-5-17-15-12-3-20-16-13-19-7-18 FA: 2573.0
  6. 8-10-4-1-9-2-14-6-11-5-17-15-12-3-20-16-13-19-7-18 FA: 2573.0
  7. 9-10-4-2-6-1-8-11-14-5-17-15-12-3-20-16-13-19-7-18 FA: 2582.0
  8. 7-2-4-12-5-11-10-6-13-1-14-8-17-15-16-3-18-20-19-9 FA: 2583.0
  9. 7-2-4-12-5-11-10-6-13-1-14-8-17-15-16-3-18-20-19-9 FA: 2583.0
 10. 7-2-4-12-5-11-10-6-13-1-14-8-17-15-16-3-18-20-19-9 FA: 2583.0

Tiempo: 1.5940468 segundos
```

Referencias

Baker, J. E. (1985). Adaptive selection methods for genetic algorithms. En *Proceedings of the 1st international conference on genetic algorithms* (pp. 101–111). Hillsdale, NJ, USA: L. Erlbaum Associates Inc.

Baker, J. E. (1987). Reducing bias and inefficiency in the selection algorithm. En *Proceedings of the second international conference on genetic algorithms on genetic algorithms and their application* (pp. 14–21). Hillsdale, NJ, USA: L. Erlbaum Associates Inc.

Coello, C. (2010). *Introducción a la computación evolutiva. notas de curso.* Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional.

Díaz, D., y Hernández, B. (2010). *Desarrollo de un algoritmo genético para la secuencia de tareas en ambientes job-shop en el taller de mecanización de la empresa metalmecanica moldes mafra.* Tesis de Master no publicada, Universidad de Carabobo.

Goldberg, D. E. (1989). *Genetic algorithms in search, optimization, and machine learning.* Addison-Wesley Publishing Company.

Goldberg, D. E., y Deb, K. (1991). A comparative analysis of selection schemes used in genetic algorithms. En *Foundations of genetic algorithms* (pp. 69–93). Morgan Kaufmann.

Goldberg, D. E., y Lingle, R., Jr. (1985). Alleles, loci and the traveling salesman problem. En *Proceedings of the 1st international confe-*

rence on genetic algorithms (pp. 154–159). Hillsdale, NJ, USA: L. Erlbaum Associates Inc.

Gosling, J., y McGilton, H. (1996, mayo). *The java language environment.* Descargado de `http://www.oracle.com/technetwork/java/langenv-140151.html`

Harrison, R., Samaraweera, L., Dobie, M., y Lewis, P. (1996). Comparing programming paradigms: an evaluation of functional and object-oriented programs. *The IEEE Software Engineering Journal*, *11*, 247 - 254.

Haupt, R. L., y Haupt, S. E. (2004). *Practical genetic algorithms.* Wiley-Interscience.

Holland, J. H. (1975). *Adaptation in natural and artificial systems, an introductory analysis with aplications to biology, control, and artificial inteligence.* Mit Press.

IBM Corporation. (2004, septiembre). *Uml basics: The class diagram, an introduction to structure diagrams in uml 2.* Descargado de `http://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/index.html`

James Rumbaugh, G. B., Ivar Jacobson. (2004). *The unified modeling language reference manual.* Addison Wesley Longman, Inc.

Ladrón, L. (1978). *Metodología de la investigación científica.* Universidad Santo Tomás.

Lerna, H. (2004). *Metodología de la investigación.* Ecoe Ediciones.

Maneiro, N. (2001). *Algoritmos genéticos aplicados a problemas de localización de facilidades.* Ph.d. thesis, Universidad de Carabobo.

MAXXESS Systems, Inc. (2012, noviembre). *What is a software framework?* Descargado de `http://www.maxxess-systems.com/whitepapers/what_is_a_software_framework.pdf`

Meffert, K. (2012, abril). *Jgap - java genetic algorithms and genetic programming package.* Descargado de `http://jgap.sf.net`

Michalewicz, Z. (1992). *Genetic algorithms + data structures = evolution programs.* Springer-Verlag.

Mitchell, M. (1998). *An introduction to genetic algorithms.* Massachusetts Institute of Technology.

Montgomery, D. C., y Runger, G. C. (1996). *Probabilidad y estadística aplicadas a la ingeniería.* McGraw-Hill / Interamericana Editores.

Méndez, C. E. (2001). *Metodología. diseño y desarrollo del proceso de investigación.* McGraw-Hill Interamericana.

Oracle Corporation. (1993, 2013, enero). *Java platform, standard edition 7. api specification.* Descargado de `http://docs.oracle.com/javase/7/docs/api/`

Oracle Corporation. (1995, 2012, noviembre). *The java tutorials.* Descargado de `http://docs.oracle.com/javase/tutorial/`

Oracle Corporation. (1999, abril). *Code conventions for the java programming language.* Descargado de `http://www.oracle.com/technetwork/java/codeconv-138413.html`

Oracle Corporation. (2012a, noviembre). *Conozca más sobre la tecnología java.* Descargado de `http://www.java.com/es/about/`

Oracle Corporation. (2012b, noviembre). *Javadoc tool (herramienta javadoc).* Descargado de `http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html`

Paperin, G. (2004). *Jaga - java api for genetic algorithms.* Descargado de `http://www.jaga.org`

Post, G. V. (2006). *Sistemas de administración de bases de datos.* McGraw Hill/Interamericana Editores.

Riehle, D. (2000). *Framework design: A role modeling approach.* Ph.d.

thesis, Eidgenössische Technische Hochschule Zürich (ETC Zürich).

Russell, S., y Norvig, P. (2009). *Artificial intelligence: A modern approach*. Prentice Hall.

Sampieri, R. H., Collado, C. F., y Lucio, P. B. (1998). *Metodología de la investigación*. McGraw-Hill Interamericana Editores.

Tanese, R. (1989). *Distributed genetic algorithms for function optimization*. Ph.d. thesis, University of Michigan, Ann Arbor, MI, USA.

TIOBE Software BV. (2012, noviembre). *Tiobe index*. Descargado de `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`