



UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA  
ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Proyecto de Fin de Grado en Ingeniería Informática

**DISEÑO, IMPLEMENTACIÓN Y EVALUACIÓN DE UN  
ALGORITMO GENÉTICO APLICADO A LA OPTIMIZACIÓN DE  
FUNCIONES REALES DE VARIAS VARIABLES**

FERMÍN SILVA

Dirigido por: SEVERINO FERNÁNDEZ GALÁN

Codirigido por:

Curso: 2014-2015





**DISEÑO, IMPLEMENTACIÓN Y EVALUACIÓN DE UN  
ALGORITMO GENÉTICO APLICADO A LA OPTIMIZACIÓN DE  
FUNCIONES REALES DE VARIAS VARIABLES**

**Proyecto de Fin de Grado en Ingeniería Informática de modalidad *específica***

Realizado por: FERMÍN SILVA

Dirigido por: SEVERINO FERNÁNDEZ GALÁN

Codirigido por:

Fecha de lectura y defensa: \_\_\_\_\_

# Resumen

Resolver problemas de optimización de funciones supone varios desafíos. Desde un punto de vista matemático, estos problemas se tornan difíciles si la función a optimizar no es continua, no es diferenciable o tiene múltiples óptimos (lo que se conoce como función multimodal). Desde un punto de vista computacional, considerando los problemas de optimización de funciones como problemas genéricos de búsqueda, su dificultad está estrechamente relacionada a la complejidad computacional de explorar el espacio de búsqueda, en términos de memoria y tiempo. Los algoritmos de búsqueda conocidos como A\*, Búsqueda en Anchura, etc. pueden ser muy efectivos en encontrar el óptimo global cuando el problema es lo suficientemente pequeño. Sin embargo, dado que su complejidad computacional es exponencial tanto en tiempo como en espacio, se hace mandatorio el uso de otros tipos de algoritmos para funciones de dimensionalidad alta, cuyos espacios de búsqueda son mucho más grandes. Se introducen entonces los Algoritmos Genéticos como una alternativa viable que escala de manera eficiente para problemas más grandes y complejos.

Este proyecto consiste en la implementación de un Algoritmo Genético completamente configurable que resuelve el problema de optimización. Al ser estocástico por naturaleza, este algoritmo necesitará de una evaluación minuciosa para asegurar que su eficiencia (complejidad en tiempo) y efectividad (calidad de la solución obtenida) son prometedoras y robustas. Para ello, se evalúa comparativamente el efecto de los distintos operadores de Recombinación, manteniendo constantes el resto de los parámetros. Esto revelará información sobre configuraciones recomendadas que prevengan uno de los problemas principales en la computación evolutiva: la convergencia prematura.

## Keywords

Optimización de Funciones, Algoritmos Estocásticos, Computación Evolutiva, Algoritmos Genéticos, Evaluación de la Recombinación

# **Summary**

## **Title**

Design, Implementation and Evaluation of a Genetic Algorithm Applied to the Optimization of Real-Valued Functions of Several Variables

## **Abstract**

Solving function optimization problems is subject to several challenges. From the mathematical perspective, such problems become difficult if the function is not continuous, not differentiable or has multiple optima (known as a multimodal function). From the computational standpoint, considering function optimization problems as generic search problems, their difficulty is closely related to the computational complexity of exploring the search space, both in terms of memory and time. Known search algorithms such as A\*, Breadth First, etc. can be very effective at finding the global optimum when the problem is small enough. However, given that their computational complexity is exponential both in time and space, the use of other types of algorithms becomes mandatory for high dimensional functions, whose search space is much larger. Genetic Algorithms are thus introduced as a viable alternative that scales efficiently for bigger and more complex problems.

This project consists of the implementation of a fully customizable Genetic Algorithm that solves the optimization problem. Being stochastic in nature, this algorithm will need thorough testing to assure that its efficiency (time complexity) and effectiveness (solution quality) are promising and robust. For that purpose, the effect of the different Recombination operators is evaluated comparatively, while keeping the remaining parameters constant. This will reveal insights on recommended configurations that prevent one of the biggest issues in evolutionary computation: premature convergence.

## **Keywords**

Function Optimization, Stochastic Algorithms, Evolutionary Computation, Genetic Algorithms, Recombination Evaluation



# Índice de Contenidos

<b>CAPÍTULO 1 : INTRODUCCIÓN .....</b>	<b>13</b>
1.1    Objetivo.....	13
1.2    Metodología .....	13
1.3    Organización de la Memoria.....	16
<b>CAPÍTULO 2 : FUNDAMENTOS DE LA COMPUTACIÓN EVOLUTIVA.....</b>	<b>19</b>
2.1    Funcionamiento General de un Algoritmo Evolutivo .....	19
2.2    Componentes Básicos.....	20
2.2.1    Representación de los Individuos .....	21
2.2.2    Selección de Padres .....	21
2.2.3    Recombinación .....	22
2.2.4    Mutación.....	22
2.2.5    Selección de Supervivientes.....	23
2.2.6    Función de Evaluación o <i>Fitness</i> .....	23
2.2.7    Criterio de Terminación .....	24
2.3    Los Algoritmos Evolutivos como Procesos de Búsqueda.....	24
<b>CAPÍTULO 3 : ALGORITMOS GENÉTICOS .....</b>	<b>27</b>
3.1    Representación.....	27
3.1.1    Representación Binaria .....	27
3.1.2    Representación con Números Reales .....	27
3.1.3    Otras Representaciones.....	28
3.2    Inicialización de la Población.....	28
3.3    Selección de Padres .....	28
3.3.1    Selección por Ranking .....	29
3.3.2    Selección por Torneo .....	29
3.4    Recombinación .....	30
3.4.1    Recombinación Discreta por Un Punto .....	31
3.4.2    Recombinación Discreta BGA.....	31
3.4.3    Recombinación Aritmética Simple.....	31
3.4.4    Recombinación Aritmética Única .....	32
3.4.5    Recombinación Aritmética Completa .....	32
3.4.6    Recombinación Aritmética K .....	32

3.4.7 Recombinación Aritmética Alfa-Beta.....	33
3.5 Mutación.....	34
3.5.1 Mutación Uniforme.....	34
3.5.2 Mutación No Uniforme.....	34
3.6 Selección de Supervivientes y Modelo Poblacional.....	35
<b>CAPÍTULO 4 : APLICACIÓN DE UN ALGORITMO GENÉTICO A LA OPTIMIZACIÓN DE FUNCIONES REALES DE VARIAS VARIABLES.....</b>	<b>37</b>
4.1 El problema de la Minimización.....	37
4.2 Algoritmo Implementado.....	38
4.3 Instalación .....	39
4.3.1 Instalación desde el Código Fuente .....	40
4.4 Estructura de Directorios .....	41
4.5 Estructura del Programa.....	42
4.5.1 Paquete Chart.....	42
4.5.2 Paquete org.jfree.chart.renderer.category .....	43
4.5.3 Paquete Función .....	44
4.5.4 Paquete Mutación.....	44
4.5.5 Paquete Selección.....	45
4.5.6 Paquete Recombinación .....	45
4.5.7 Paquete Terminación.....	46
4.5.8 Paquete Utils.....	46
4.5.9 Paquete Población.....	47
4.5.10 Paquete Params .....	47
4.5.11 Paquete Web.....	48
4.6 Librerías Auxiliares y Motivación de las Mismas .....	49
4.6.1 JCommander .....	50
4.6.2 JFreeChart.....	50
4.6.3 JUnit.....	51
4.6.4 Jetty.....	51
4.7 Ejecución y Comportamiento del Programa.....	51
4.7.1 Ejecución por Consola .....	52
4.7.2 Ejecución en Modo Benchmark.....	55
4.7.3 Ejecución en Modo Servidor .....	56
<b>CAPÍTULO 5 : EVALUACIÓN EXPERIMENTAL.....</b>	<b>69</b>
5.1 Introducción .....	69

5.2	El gráfico Box Plot .....	70
5.3	Metodología .....	72
5.4	Evaluación de la Función Griewank .....	73
5.5	Evaluación de la Función Rastrigin .....	78
5.6	Evaluación de la Función Schwefel.....	85
5.7	Evaluación del Valor de K .....	90
5.8	Discusión Final de los Resultados de Evaluación .....	93
5.8.1	Tiempos de Ejecución.....	94
5.8.2	Memoria Utilizada.....	95
5.8.3	Convergencia.....	95
5.8.4	Curva de Progreso.....	96
<b>CAPÍTULO 6 : CONCLUSIONES Y TRABAJO FUTURO .....</b>		<b>97</b>
<b>BIBLIOGRAFÍA.....</b>		<b>103</b>



# **Lista de Figuras y Tablas**

Figura 3.1 - Recombinación con promedio ponderado .....	30
Figura 3.2 - Representación de la Recombinación Alfa-Beta .....	33
Figura 4.1 - Diagrama de clases del paquete chart .....	43
Figura 4.2 - Diagrama de clases de la Recombinación .....	46
Figura 4.3 - Diagrama de clases del paquete Web.....	49
Figura 4.4 - Flujo básico de un pedido HTTP .....	58
Figura 4.5 - Diagrama de Ejecución en el Servidor Público .....	59
Figura 4.6 - Diagrama de Ejecución Local.....	61
Figura 4.7 - Diagrama de Ejecución Remota.....	62
Figura 4.8 - Pantalla de Inicio.....	63
Figura 4.9 - Pantalla de Documentos .....	64
Figura 4.10 - Pantalla de Instrucciones .....	65
Figura 4.11 - Pantalla de Ejecución .....	66
Figura 4.12 - Pantalla de Ejecución, Resultados .....	66
Figura 4.13 - Pantalla de Pruebas Empíricas.....	67
Figura 5.1 - Ejemplo de Box Plot.....	71
Figura 5.2 - Función Griewank .....	73
Figura 5.3 - Box Plot de la función Griewank .....	74
Figura 5.4 - Curva de Progreso función Griewank .....	76
Figura 5.5 – Convergencia, función Griewank .....	77
Figura 5.6 - Tiempos de ejecución, función Griewank .....	78
Figura 5.7 - Función Rastrigin.....	79
Figura 5.8 - Box Plot, función Rastrigin .....	80
Figura 5.9 - Curva de progreso, función Rastrigin .....	82
Figura 5.10 - Convergencia, función Rastrigin.....	83
Figura 5.11 - Tiempos de ejecución, función Rastrigin .....	84
Figura 5.12 - Función Schwefel .....	85
Figura 5.13 - Box Plot, función Schwefel.....	87
Figura 5.14 - Curva de progreso, función Schwefel .....	88
Figura 5.15 - Convergencia, función Schwefel.....	89
Figura 5.16 - Tiempos de ejecución, función Schwefel .....	90
Figura 5.17 - BoxPlots para los distintos valores de k .....	91

Figura 5.18 - Curvas de Progreso para los distintos valores de k .....	92
Figura 5.19 - Convergencia de los distintos valores de k.....	93
Tabla 1.1 - Cronograma de Actividades.....	14
Tabla 5.1 - Resultados de la función Rastrigin.....	81
Tabla 5.2 - Resultados de la función Schwefel.....	88
Tabla 5.3 - Resultados finales de la ejecución.....	94

# Capítulo 1: Introducción

## 1.1 Objetivo

En el presente trabajo se resuelve el problema de la optimización de funciones mediante procedimientos de búsqueda, aproximación e iteración. A diferencia de los procedimientos analíticos, los procesos de aproximación iterativa son aplicables a una mayor cantidad de funciones, por ejemplo las discontinuas o no derivables. Aun si la función a optimizar sí fuera derivable, su derivada podría no ser trivial, como en el caso de funciones periódicas, raíces cuadradas, etc., o su dimensionalidad podría ser muy elevada. Es por ello que se diseña e implementa un Algoritmo Genético como una alternativa robusta y eficiente para la optimización de funciones. En general, estos algoritmos ofrecen importantes ventajas en cuanto a eficiencia temporal, pero no garantizan la obtención del óptimo global. Esto no quiere decir que la solución sea incorrecta, sólo que podría ser un óptimo local.

Para encontrar buenas soluciones, será necesaria una cuidadosa selección de los parámetros del algoritmo, al ser este completamente configurable. Como es difícil determinar *a priori* cómo se comportará el algoritmo en base a sus parámetros de entrada, se deberá experimentar con ellos. En este proyecto se estudia el efecto de uno de los parámetros en la obtención de buenas soluciones de manera consistente y repetible: el operador de Recombinación. Se concluye el trabajo con una evaluación empírica (*a posteriori*) del algoritmo desde un punto de vista computacional y de la calidad de las soluciones obtenidas.

## 1.2 Metodología

El desarrollo del proyecto se ha dividido en fases consecutivas de Análisis y Diseño, Implementación, Evaluación y Redacción Final. Se ha dedicado aproximadamente entre 4 y 5 horas diarias de trabajo (alrededor de 25 horas semanales) durante 18 semanas, por un total de ~450 horas. Se incluye un cronograma de actividades en la Tabla 1.1.

SEM.	DESDE	HASTA	TAREA	FASE	
1	16-Feb	22-Feb	Adquisición de Base Teórica	Análisis y Diseño	
2	23-Feb	1-Mar			
3	2-Mar	8-Mar	Análisis Funcional y Diseño de Interfaces	Análisis y Diseño	
4	9-Mar	15-Mar			
5	16-Mar	22-Mar	Implementación inicial de Interfaces y Pruebas de Integración	Implementación	
6	23-Mar	29-Mar	Implementación		
7	30-Mar	5-Abr			
8	6-Abr	12-Abr	Investigación sobre Gráficos Estadísticos y Librerías para generarlos		
9	13-Abr	19-Abr	Codificación del programa para Pruebas Empíricas	Evaluación	
10	20-Abr	26-Abr	Codificación de los Generadores de Gráficos y BoxPlots		
11	27-Abr	3-May	Evaluación Experimental		
12	4-May	10-May			
13	11-May	17-May	Comienzo de la escritura de la Memoria	Redacción y Pulido final	
14	18-May	24-May	Redacción de la Memoria y finalización de pequeñas funcionalidades pendientes en el código		
15	25-May	31-May			
16	1-Jun	7-Jun			
17	8-Jun	14-Jun	Retoques Finales de la Memoria y el código		
18	15-Jun	21-Jun			

**Tabla 1.1 - Cronograma de Actividades**

En la fase de Análisis y Diseño se ha adquirido la base teórica necesaria y se ha realizado un esquema inicial del algoritmo. Para ello se han diseñado las interfaces básicas necesarias para el ensamblado del cuerpo principal del algoritmo. El diseño realizado requiere de un paradigma orientado a objetos estricto, al utilizar interfaces, clases abstractas y finalmente objetos. Se utilizan además patrones de diseño de dicho paradigma, como el patrón Estrategia y Factoría<sup>1</sup>. Por todo ello se decidió utilizar el lenguaje Java en su versión 1.6, ya que presenta un manejo de objetos avanzado. Es además un lenguaje multiplataforma, requisito esencial ya que el desarrollo se ha realizado en ordenadores con entornos y Sistemas Operativos diferentes (Windows 8.1 –aunque sólo para pruebas-, Ubuntu 14 y Mac OSX).

La fase de Implementación comenzó con la creación de implementaciones vacías que simulan la ejecución pero no realizan nada, denominadas NoOp (No Operación). De esta manera se ha podido tener un programa funcional rápidamente, aunque este difícilmente pudiera resolver el problema. Esta estrategia permite probar el software desde el comienzo y tener una integración continua entre los módulos (a diferencia de la integración en *Big*

<sup>1</sup> Véase más sobre dichos patrones en [https://es.wikipedia.org/wiki/Strategy\\_\(patrón\\_de\\_diseño\)](https://es.wikipedia.org/wiki/Strategy_(patrón_de_diseño)) y <https://msdn.microsoft.com/es-mx/library/bb972258.aspx#M11>.

*Bang*<sup>2</sup>). De ahí en adelante se siguió un modelo iterativo y ágil que sigue las fases de diseñar, implementar, probar y pulir. Este modelo de desarrollo, acompañado por la arquitectura del programa, permitió focalizarse en pequeñas partes del mismo de manera aislada, minimizando el riesgo y maximizando la productividad.

La fase de Evaluación comienza con la programación de clases destinadas específicamente a probar el algoritmo de forma exhaustiva y otras destinadas a recoger y presentar los resultados obtenidos. Finalizado esto, se realizaron de manera manual las pruebas, anotando los parámetros de entrada y la salida obtenida.

Finalmente, en la fase de Redacción, se redactó esta memoria a la vez que se pulieron los retoques finales del algoritmo, especialmente en cuanto a la presentación de la salida y la adquisición de los parámetros de entrada. Con respecto a esto, y como se comentará en la sección 4.7, la entrada y salida del algoritmo se podrán manejar por línea de comandos y vía Web, instalando la aplicación o accediendo a una dirección Web particular.

En el desarrollo de todas las fases se destaca el carácter distribuido del proyecto. Primero por utilizarse varios ordenadores con entornos diferentes para realizar la memoria. Segundo (y principal) por encontrarse el Alumno en Argentina y el Director del Proyecto en España. Todo esto ha requerido coordinación por correo electrónico y la compartición de recursos de manera Online. Esto incluye el código fuente, versiones preliminares de la Memoria, imágenes y gráficos con el comportamiento del algoritmo, etc. Para ello se han utilizado servicios gratuitos y públicos en la Nube, en especial:

- Dropbox: servicio que permite almacenar archivos en la Nube y sincronizarlos entre varios ordenadores. Si alguno de los archivos sincronizados se modifica, los cambios se suben automáticamente a la Nube y se replican en todos los demás ordenadores configurados. Permite además compartir enlaces públicos para descargar estos archivos así como poder visualizarlos de manera Online.
- Github: plataforma Web que permite el almacenamiento y control de versiones de software. A diferencia de Dropbox, este está específicamente diseñado para gestionar código fuente, los cambios realizados en cada versión, documentación online, etc. El acceso de lectura es público, por lo que no se necesita ningún permiso

---

<sup>2</sup> Existen varios tipos de pruebas de integración. Los más comunes son Big Bang, Top-Down y Bottom-Up. En Big Bang, se acoplan todos los módulos de una sola vez, reduciendo la cantidad de pruebas (pero también dificultando la detección de posibles errores).

para ver y descargar el código, fomentando una cultura de código abierto incluso más allá del marco de este proyecto. Github permite además ofrecer una página Web asociada al código. Dicha página se utiliza para la presentación del proyecto y la ejecución vía Web. Esto se ampliará en las secciones 4.3 (Instalación) y 4.7 (Modos de Ejecución).

### 1.3 Organización de la Memoria

La memoria está organizada en 6 capítulos, comenzando con una introducción general al problema a resolver y la metodología utilizada para ello.

El segundo capítulo presenta de manera general la computación evolutiva, una familia de algoritmos que siguen una estructura común. Esta se compone de una serie de fases u operadores básicos, los cuales se describen a alto nivel, ya que sus implementaciones dependerán de la rama específica de la familia de algoritmos.

En el Capítulo 3 se introducen los Algoritmos Genéticos, una de las ramas particulares de los Algoritmos Evolutivos. Se describen con más detalle los operadores básicos y se explica la implementación en código adoptada en este trabajo, incluyendo pequeñas porciones de código donde sea relevante. Se incluyen además referencias a los archivos con el código fuente para comenzar a guiar al lector sobre la estructura del programa, aunque esta se explicará con más profundidad más adelante.

A continuación en el Capítulo 4 se define el problema que se pretende resolver en el presente trabajo, así como la herramienta software construida para tal fin. Se indica cómo instalar la aplicación y se explica de manera exhaustiva la estructura del programa, sus paquetes y clases principales. Para ello se muestran porciones de código y diagramas de clase. Se concluye el capítulo con los modos de ejecución soportados por el programa, sus diagramas de flujo, ejemplos de ejecución y capturas de pantalla.

En el Capítulo 5 se realizan pruebas experimentales de cara a evaluar el rendimiento y comportamiento de la fase de Recombinación, aunque con ellas se evalúa también el rendimiento del algoritmo en su totalidad. Para las pruebas se utilizan tres funciones reales concretas con distinta complejidad y dimensionalidad, utilizadas comúnmente para evaluar algoritmos de optimización (véase [1] y [7]).

Se concluye esta memoria con el Capítulo 6, donde se establecen una serie de conclusiones sobre los algoritmos genéticos en general y sobre su aplicación a la optimización de funciones reales en particular. A lo largo de toda la memoria se enumeran

posibles extensiones y mejoras, que se recopilan en este capítulo dentro de una sección sobre trabajo futuro.



# Capítulo 2: Fundamentos de la Computación Evolutiva

La Computación Evolutiva es una familia de algoritmos inspirados en el proceso de evolución natural. Los Algoritmos Evolutivos (*AEs* de aquí en adelante), involucran casi exclusivamente mecanismos de evolución biológica como pueden ser reproducción, mutación, etc.

La idea general de los *AEs* consiste en utilizar un conjunto de soluciones candidatas (conocidas como *Individuos*) en un entorno determinado por el problema a resolver. Cada Individuo tendrá asociada una calidad o adecuación de la solución al problema (también conocida como *fitness*, del término en inglés). Al igual que en la Naturaleza, los candidatos con mejor *fitness* (los más adaptados) tendrán más posibilidades de reproducirse. Similarmente, se espera que los hijos de estos buenos candidatos sean aún mejor que los propios padres. De lo contrario, no sobrevivirán a las futuras generaciones.

De esta forma, toda la población, vista como una especie, evoluciona orgánicamente para maximizar su adaptación al entorno. Vistas como soluciones candidatas a un problema, el algoritmo maximiza la calidad de las mismas iterativamente. Una vez finalizado el proceso, se escoge la mejor solución, aunque las demás podrían llegar a ser también de interés.

## 2.1 Funcionamiento General de un Algoritmo Evolutivo

A pesar de las diferencias entre los distintos tipos básicos de *AEs*, todos siguen un esquema común, que a alto nivel se comporta como en el siguiente pseudocódigo:

```
INICIALIZAR la población con soluciones candidatas
EVALUAR cada individuo
REPETIR HASTA( CONDICIÓN TERMINACIÓN )
    SELECCIONAR padres
    RECOMBINAR los padres
    MUTAR los hijos resultantes
    EVALUAR los nuevos individuos
    SELECCIONAR supervivientes para la nueva población
END REPETIR
```

Se evidencia que se trata de un algoritmo iterativo, del tipo *generar y probar*. A cada iteración se la conoce como **generación**, por la analogía con el proceso natural de evolución.

A cada una de las partes resaltadas se las conoce como **operadores**. Existen distintas variantes de Algoritmos Evolutivos, que difieren en detalles técnicos de implementación de estos operadores, en la manera de codificar cada individuo y en la función de evaluación o *fitness*, entre otras cosas. A grandes rasgos, estas variantes son:

- **Algoritmos Genéticos:** las soluciones se codifican como cadenas de bits (1's y 0's), enteros, flotantes, etc. y buscan la solución mediante selección, recombinación y mutación de las cadenas.
- **Estrategias Evolutivas:** muy similares a los anteriores, utilizan por lo general vectores de números reales. Se diferencian en la fase de mutación, para la cual utilizan una distribución normal, que también evoluciona junto con la población. Esto les permite una autoadaptación durante la ejecución y, en ocasiones, un mejor rendimiento.
- **Programación Genética:** las soluciones son en forma de programas de computadora y su fitness se determina por su habilidad para resolver un problema computacional concreto.
- **Programación Evolutiva:** similar a la anterior, donde los programas solución son fijos y lo que evoluciona son los parámetros numéricos del mismo.

Además, para una mismo tipo de AE y dependiendo del problema en particular, se podrán intercambiar las implementaciones de cada uno de los operadores, siempre que se preserve la misma interfaz, ya que tienen poco acoplamiento entre sí. Esto favorece un diseño altamente modular y reutilizable, como se verá en el Capítulo 4 sobre la aplicación a un problema concreto.

## 2.2 Componentes Básicos

Para poder definir un AE hace falta definir la implementación concreta de cada una de sus partes. Sin profundizar en ningún algoritmo particular, se explican a continuación las partes principales y su rol dentro del algoritmo.

### 2.2.1 Representación de los Individuos

Si bien no aparece en el algoritmo de la sección 2.1, la representación es el primer paso antes de diseñar un AE, ya que condicionará el funcionamiento e implementación de las etapas posteriores. La misma consiste en definir una codificación para los genes de cada individuo (su *genotipo*), es decir, una estructura de datos que permita almacenarlos y modificarlos. Igualmente importante es definir el mapeo entre estos genes y la solución candidata que el individuo representa (su *fenotipo*).

Es importante destacar que el AE trabaja directamente modificando el genotipo de cada individuo, sin hacer muchas suposiciones sobre su fenotipo, el cual puede ser mucho más complejo.

#### Ejemplo:

Dada una lista de ciudades interconectadas, y la distancia entre cada una de ellas, se define el Problema del Viajante de Comercio (*Travelling Salesman Problem* o *TSP* por sus siglas en inglés) como encontrar la ruta más corta posible que, partiendo de una ciudad de origen, visite cada ciudad exactamente una vez y regrese al origen. Se trata de un problema clásico de optimización combinatoria.

Para resolver el TSP con un AE, el fenotipo podría ser el grafo conectado de ciudades y el orden en que deben ser recorridas, mientras que el genotipo podría ser un arreglo de números enteros. Cada número del arreglo identifica a una ciudad, y la posición o índice dentro del mismo indica el orden de visita. Por ejemplo el arreglo [1, 3, 2, 4] indica que, partiendo de la ciudad número 1, el orden de visita debe ser: en segundo lugar la número 3, luego la ciudad número 2 y luego la 4, para regresar finalmente a la primera.



### 2.2.2 Selección de Padres

La selección de padres es el proceso de elegir  $n$  individuos para formar un grupo de reproducción (conocido como *mating pool*), que será luego enviado a la etapa de Recombinación.

El propósito de esta selección es dar más importancia a los individuos más aptos con la esperanza que sus hijos (también conocidos como crías u *offspring*) sean aún mejores. La etapa de selección de padres mejora la *fitness* promedio de la población. Es por lo general un proceso probabilístico, pero no completamente aleatorio. Los mejores individuos tienen más probabilidad de ser seleccionados, aunque siempre debería ser posible que un

individuo peor sea seleccionado. Esto mantiene la diversidad genética de la población y favorece la exploración, como se verá más adelante.

La selección de padres, por lo general, no depende de la representación concreta de los individuos, ya que sólo necesita conocer su *fitness*.

### 2.2.3 Recombinación

También conocida como *crossover*, obtiene padres del *mating pool* (ya sea al azar o secuencialmente) y los cruza para obtener hijos. Por lo general, se toman dos padres y se generan dos hijos, aunque hay estudios que utilizan un número mayor [2].

La idea general es que al recombinar dos padres con buenas pero distintas características, se obtendrán dos hijos que, probablemente, heredarán lo mejor de sus padres.

La recombinación se hace en el espacio del genotipo (por lo que depende de la representación) y poco se asume sobre el fenotipo de los padres. Es, por lo tanto, un proceso estocástico con componentes aleatorias (aunque existen técnicas heurísticas para la recombinación, véase los Algoritmos Meméticos en [1]). Por eso existirán ocasiones en que los hijos no presentarán ninguna mejora o serán incluso peores.

### 2.2.4 Mutación

El rol de la mutación es generar nuevo material genético en los hijos introduciendo pequeñas variaciones aleatorias en los genotipos. La operación de mutación se aplica sobre cada hijo individualmente.

La mutación opera sobre el genotipo, por lo que es dependiente de la representación. En los algoritmos genéticos, la mutación es un operador muy importante para generar variedad genética (junto con la Recombinación). Por regla general, más allá de las modificaciones hechas en el genotipo, la variación resultante en el espacio del fenotipo debería ser pequeña y no al revés.

#### Ejemplo:

Considérese un genotipo consistente en una cadena binaria de longitud 6 y su fenotipo un número decimal. El mapeo simplemente convierte el número de base 2 a base 10. Supóngase que la mutación intercambia el valor de un gen aleatorio (de 0 a 1 y viceversa). Se muestra a continuación una serie de ejemplos de los cambios introducidos por la

mutación. A la izquierda aparece el genotipo original (cuyo fenotipo es igual a  $48_{(10)}$ ) y a la derecha el genotipo mutado con su nuevo fenotipo. En negrita se resalta el gen mutado:

$$\mathbf{110000} = 48_{(10)} \rightarrow \mathbf{110001} = 49_{(10)}$$

$$\mathbf{110000} = 48_{(10)} \rightarrow \mathbf{110100} = 52_{(10)}$$

$$\mathbf{110000} = 48_{(10)} \rightarrow \mathbf{010000} = 16_{(10)}$$

Como puede apreciarse, la diferencia entre el fenotipo original y el mutado no siempre es constante, siendo en ocasiones demasiado grande. La alternativa consiste en utilizar otro operador de mutación o utilizar una codificación que no presente este problema, como por ejemplo códigos Gray<sup>3</sup>.



### 2.2.5 Selección de Supervivientes

En la mayoría de los AEs, el tamaño de la población se mantiene constante. Es por ello que después de haber generado los hijos, es necesario determinar qué individuos formarán parte de la siguiente generación. Esta etapa es similar a la selección de padres en cuanto opera en el espacio del fenotipo, por lo que no depende de la representación concreta. Sin embargo, suele ser un proceso determinista y no estocástico, basado normalmente en la *fitness* de los individuos, aunque también suele ser común utilizar su edad (es decir, la generación), reemplazando los más antiguos.

### 2.2.6 Función de Evaluación o *Fitness*

La representación mapea el genotipo (estructura de datos interna) al fenotipo (representación de una solución candidata). En ese contexto, la función de evaluación dota al fenotipo de una medida de calidad. Cuando se hablaba de que la selección opera en el espacio del fenotipo, en realidad utiliza la función de evaluación para tomar sus decisiones, ya que lo que importa no es la solución en sí, sino su calidad.

Volviendo al ejemplo del viajante de comercio, si el fenotipo es el grafo conectado de las ciudades y el orden de visita, la función de evaluación podría devolver la diferencia entre la distancia máxima posible y la distancia total de recorrer las ciudades en ese orden. Se aprecia que es una función que devuelve un número real, que permitirá comparar soluciones entre sí sin conocer la estructura compleja utilizada internamente. Nótese que

---

<sup>3</sup> Sistema de numeración binario donde los valores sucesivos difieren solamente en uno de sus dígitos.

no puede simplemente devolverse el valor de la distancia, ya que el TSP se trata de un problema de minimización y los AE siempre maximizan la función de Evaluación (y por ende la distancia del recorrido). Se amplía este problema en la sección 4.1.

### 2.2.7 Criterio de Terminación

Al tratarse de un proceso iterativo y de aproximación, es necesario determinar en qué momento detenerse. Los criterios más comunes incluyen, pero no se limitan a:

- tras cierta cantidad de iteraciones
- cuando la *fitness* sobrepase un determinado umbral
- cuando se detecte que la población haya convergido a un punto del cual no puede escapar
- cuando la mejora en la *fitness* en las ultimas  $n$  iteraciones sea menor que un determinado umbral

## 2.3 Los Algoritmos Evolutivos como Procesos de Búsqueda

Muchos de los problemas que pueden resolverse mediante AEs son esencialmente problemas de búsqueda. Ejemplos de ello son el problema del viajante de comercio, planificación de tareas, encontrar una ruta que une dos puntos y, por supuesto, la optimización de funciones. En todos los casos se necesita encontrar una solución óptima, que será la mejor entre otras soluciones candidatas. El conjunto de todas ellas se conoce como **espacio de búsqueda**. Los algoritmos que resuelven este tipo de problemas parten de un estado inicial y, mediante **operadores de transición**, exploran el espacio de búsqueda (estados) para encontrar la mejor solución posible, aunque no serán capaces en general de distinguir de óptimos locales y el óptimo global. La excepción son aquellos algoritmos que realizan una búsqueda exhaustiva de todo el espacio de búsqueda, garantizando que la solución encontrada es efectivamente la mejor de todas (el óptimo global). La complicación radica en que el espacio de búsqueda suele ser demasiado grande como para explorarlo todo. Incluso puede ser infinito, como en el caso de la optimización de funciones reales. En esos casos es necesario recorrer sólo una parte del espacio total, quizás mediante técnicas heurísticas o mediante técnicas estocásticas. Este último es el método utilizado por los AEs.

Se puede decir que los AEs exploran el espacio de búsqueda de manera aleatoria a medida que la población evoluciona, de aquí su componente estocástica. Sin embargo, el

proceso no es completamente aleatorio, ya que los operadores básicos (Selección, Recombinación, etc.) guían la búsqueda. Esto acelera el proceso considerablemente, aunque puede introducir problemas. Si la búsqueda es demasiado guiada, no se explorará lo suficiente el espacio de búsqueda, por lo que aumenta la probabilidad de atascarse en un óptimo local. Este fenómeno se conoce como **convergencia prematura** y se comentará con más detalle en el Capítulo 5. Por tanto, debe encontrarse un compromiso entre tiempo de ejecución y exploración, que impactará en la calidad de la solución encontrada. En inglés, esto se conoce como *Exploration vs Exploitation* [4] (exploración vs explotación). La exploración en los AEs es la creación de diversidad en la población, y se logra mediante los operadores de inicialización y mutación, aportando nuevo material genético. La explotación es la reducción de la diversidad de la población, focalizándose en las mejores soluciones candidatas para mejorarlas aún más. Esta se logra con los operadores de selección y recombinación, aunque esta última puede también introducir algo de diversidad.



# Capítulo 3: Algoritmos Genéticos

Este proyecto se centra en los Algoritmos Genéticos, una variante particular dentro de la familia de los AEs: la más sencilla pero a su vez la más popular. Es por ello que se dedica este capítulo a explicar sus características, componentes principales y las técnicas e implementaciones específicas adoptadas en este trabajo. Para ello se indicará el archivo donde puede verse el código fuente (la *clase Java* en particular), así como el paquete al que pertenece. Los paquetes se explicarán en profundidad en la sección 4.5. Por el momento simplemente considerarlos como directorios ordinarios donde se agrupan clases del lenguaje Java asociadas a algún concepto.

## 3.1 Representación

La representación es una de las partes principales en los Algoritmos Genéticos (AGs de aquí en adelante), ya que condiciona varias de las siguientes etapas. Es también la primera decisión a efectuar de cara a implementar un algoritmo de este tipo.

### 3.1.1 Representación Binaria

Consiste en utilizar una cadena dígitos binarios de longitud fija. Dependiendo del problema a solucionar, se puede cambiar el mapeo al fenotipo para representar: números (basta convertir el número binario a uno real en base diez), vectores de características booleanas (cada bit representa si la característica es verdadera o falsa), conexiones en un grafo (viendo la cadena como una matriz en la que cada fila está concatenada), etc.

### 3.1.2 Representación con Números Reales

Forman el genotipo con una cadena o vector de  $n$  números reales. Especialmente útil cuando las magnitudes del AG son continuas en vez de discretas. En ocasiones, el mapeo resulta más intuitivo y directo que la versión binaria, por lo que produce mejores resultados. Es por ello que en este trabajo se ha elegido esta representación. Esto puede verse en la clase `Individuo`, donde el cromosoma se define simplemente como un arreglo de `Double`. Cada uno de estos números es lo que se conoce como un **Alelo**.

### 3.1.3 Otras Representaciones

Dependiendo del problema, pueden modificarse las representaciones anteriores o incluso crear algunas nuevas, como pueden ser:

- Conjuntos de números enteros (ya sea como variables enteras o para representar permutaciones)
- Árboles y grafos
- Cadenas de caracteres
- Imágenes o pixeles

Es claro que algunas de estas, a pesar de considerarse independientes, utilizarán internamente números enteros o reales.

## 3.2 Inicialización de la Población

Una vez elegida la representación de la población y definido su genotipo, debe elegirse el método de inicialización de la población. Este consiste en asignar valores específicos a cada uno de los alelos.

Existen en general tres variantes para la inicialización de la población:

- Aleatoria: los alelos se generan aleatoriamente, posiblemente entre ciertos valores máximos y mínimos.
- Uniforme: se eligen los alelos de forma tal que se garantice una distribución uniforme y equidistante en todo el espacio de búsqueda.
- Heurística: se utiliza conocimiento específico del problema para generar los valores del alelo .

En este proyecto se implementa la primera alternativa, primero por su simplicidad pero también por demostrar buena diversidad genética, lo que a su vez mejora el rendimiento del algoritmo. Puede verse la implementación en la clase `Individuo`, en el método `getIndividuosInicial`.

## 3.3 Selección de Padres

Dada una población de entrada, el trabajo de la selección de padres consistirá en devolver un arreglo de `Individuos`, que serán los padres para la nueva generación. Por simplicidad, el tamaño del arreglo será igual al tamaño de la población. Si el modelo

poblacional necesitará de menos individuos para la nueva generación, podrá simplemente descartarlos.

Pueden encontrarse las distintas implementaciones de este operador en el paquete `es.uned.pfg.ae.seleccion` del presente proyecto. Todas ellas deben implementar la interfaz `Selección`.

### 3.3.1 Selección por Ranking

Para que la probabilidad de selección de un Individuo sea directamente proporcional a su fitness, es común utilizar la Selección Proporcional a la Adecuación (*FPS*, *Fitness Proportionate Selection*). Sin embargo, esta presenta problemas cuando la función de adecuación retorna valores nulos o negativos, como se verá en el apartado 4.1 sobre el problema de la minimización. Una alternativa consiste en ordenar la población según su *fitness* de mayor a menor en una lista. Luego la probabilidad de selección de un Individuo será proporcional a su posición dentro de la lista, que puede ser lineal, logarítmica, exponencial, etc.

Por motivos de tiempo no se ha implementado esta variante, por lo que se deja como posible extensión futura.

### 3.3.2 Selección por Torneo

Tiene la propiedad interesante de no requerir ordenar por completo la población ni hacer ninguna normalización. Simplemente requiere una forma de comparar dos individuos para determinar cuál de ellos es mejor. El funcionamiento para obtener un *mating pool* de padres aparece en el siguiente pseudocódigo:

```

padres = { vacío }
MIENTRAS ( tamaño(padres) < CANTIDAD_PADRES )
    torneo = OBTENER k individuos aleatoriamente de la población
    padres = padres + SELECCIONAR_MEJOR( torneo )
FIN
  
```

Los parámetros de interés son  $k$  (el tamaño del torneo) y si la obtención aleatoria es o no con reemplazamiento. La cantidad de padres a seleccionar dependerá del modelo poblacional, aunque puede asumirse que es exactamente el tamaño de la población.

Puede verse la implementación en la clase `SeleccionTorneo`, donde se aprecia que la obtención es sin reemplazamiento y el tamaño del torneo es parametrizable por el usuario.

### 3.4 Recombinación

La recombinación es una de las características principales de los AGs, ya que es esencialmente lo que los separa de una búsqueda estocástica aleatoria. Es también la etapa que se estudiará con detenimiento en este trabajo (véase el Capítulo 5), por lo que se han implementado varias alternativas, todas para una representación de números reales.

Existen a grandes rasgos dos variantes de recombinación en algoritmos genéticos con representación real:

- Las discretas: cruzan los alelos (los números reales) de un padre u otro, con el inconveniente que no generan nuevos valores o nuevo material genético.
- Las aritméticas: para cada alelo realizan una operación matemática con los valores de los dos padres, para dar lugar a un nuevo número. Por lo general, son alguna variante de un promedio ponderado. Para este promedio suele utilizarse un valor de ponderación  $\alpha$  entre  $[0, 1]$ . Puede verse un ejemplo del comportamiento en la Figura 3.1, donde los padres  $x$  e  $y$  generan los hijos  $x'$  e  $y'$  en una misma línea.



**Figura 3.1 - Recombinación con promedio ponderado**

Considerando a los padres  $x$  e  $y$  como puntos en el espacio de soluciones y representados por vectores reales, estos se encontrarán a una distancia  $d$ . Esto se representa en la Figura 3.1 mostrando la distancia en un solo eje  $i$ . Las recombinaciones aritméticas utilizan entonces el parámetro  $\alpha$  para ponderar el promedio de los padres, lo que tiene el efecto de acercar a los hijos entre sí. El desplazamiento de los hijos será igual a  $ad$ , quedando la coordenada  $i$ -ésima de los hijos como:

$$\text{hijo1}[i] = \text{hijo1}[i] + \alpha d$$

$$\text{hijo2}[i] = \text{hijo1}[i] + (1-\alpha)d$$

Si  $\alpha = 0.5$  entonces ambos hijos serán idénticos. Existe también la posibilidad de utilizar un  $\alpha < 0$ , lo cual generaría hijos alejados entre sí, es decir, fuera del intervalo entre los dos padres. Aunque no se estudia dicha posibilidad, se deja como posible ampliación futura.

Pueden encontrarse las implementaciones de este operador en el paquete `es.uned.pfg.ae.recombinacion` y todas heredan de la interfaz `Recombinacion`. Se explican a continuación brevemente las distintas implementaciones de la interfaz y se remite a la sección 4.5 para una explicación y diagrama de clases completo.

### 3.4.1 Recombinación Discreta por Un Punto

También conocida como *one-point crossover*, es la recombinación discreta más sencilla. Funciona eligiendo aleatoriamente un punto de cruce  $k$ . Hasta ese punto, los genes del primer hijo provienen del primer parente. A partir de ese punto los genes provienen del segundo parente. El segundo hijo se obtiene de manera análoga con los padres intercambiados.

$$\text{hijo1} = \{x_1, \dots, x_k, y_{k+1}, \dots, y_n\}$$

$$\text{hijo2} = \{y_1, \dots, y_k, x_{k+1}, \dots, x_n\}$$

Se implementa en la clase `RecombinacionUnPunto`.

### 3.4.2 Recombinación Discreta BGA

Implementación introducida en [11], que utiliza una probabilidad de 0.5 para determinar de qué parente proviene el gen  $i$ -ésimo. Si el número generado es menor que 0.5, el gen proviene del primer parente, de lo contrario del segundo. Se sigue el mismo procedimiento para el segundo hijo invirtiendo el parente.

Se implementa en la clase `RecombinacionDiscretaBGA`.

### 3.4.3 Recombinación Aritmética Simple

Toma un punto de recombinación  $k$  aleatorio. Luego, para el primer hijo toma los primeros  $k$  números del primer parente. El resto es el promedio aritmético de los dos padres (denominados a continuación como  $x$  e  $y$ ).

$$\text{hijo1} = \{ x_1, \dots, x_k, \alpha * y_{k+1} + (1 - \alpha) * x_{k+1}, \dots, \alpha * y_n + (1 - \alpha) * x_n \}$$

$$\text{hijo2} = \{ y_1, \dots, y_k, \alpha * x_{k+1} + (1 - \alpha) * y_{k+1}, \dots, \alpha * x_n + (1 - \alpha) * y_n \}$$

En [1] figura como *Simple Arithmetic Recombination* y en este proyecto se implementa en la clase RecombinacionSimple.

### 3.4.4 Recombinación Aritmética Única

Toma un punto de recombinación  $k$  aleatorio. En ese punto se realiza el promedio aritmético de los padres y el resto de los valores se dejan sin modificaciones.

$$\text{hijo1} = \{ x_1, \dots, x_k - 1, \alpha * y_k + (1 - \alpha) * x_k, \dots, x_n \}$$

$$\text{hijo2} = \{ y_1, \dots, y_k - 1, \alpha * x_k + (1 - \alpha) * y_k, \dots, y_n \}$$

En [1] figura como *Single Arithmetic Recombination* (no confundir con *Simple*) y en este proyecto se implementa en la clase RecombinacionUnica.

Si se tienen los padres  $x$  e  $y$ , a una distancia  $d$ , el promedio ponderado genera dos hijos  $x'$  e  $y'$  en la misma línea que los padres. La Figura 3.1 muestra un ejemplo para un  $\alpha > 0$ .

### 3.4.5 Recombinación Aritmética Completa

Realiza el promedio ponderado de los padres para cada uno de los alelos. Si se elige  $\alpha = 0.5$  ambos hijos serán idénticos.

$$\text{hijo1} = \{ \alpha * x_1 + (1 - \alpha) * y_1, \dots, \alpha * x_n + (1 - \alpha) * y_n \}$$

$$\text{hijo2} = \{ \alpha * y_1 + (1 - \alpha) * x_1, \dots, \alpha * y_n + (1 - \alpha) * x_n \}$$

En [1] figura como *Whole Arithmetic Recombination* y en este proyecto se implementa en la clase RecombinacionAritmeticaCompleta.

### 3.4.6 Recombinación Aritmética K

Si se piensa en los alelos como coordenadas, puede imaginarse el genotipo como un vector de números reales que representa un punto en el espacio. Visto de esta manera, las recombinaciones aritméticas acercan a los puntos (que ahora serán los hijos) a lo largo de ciertos ejes.

La recombinación aritmética única (véase la sección 3.4.4) acerca los puntos en un único eje, mientras que la aritmética completa lo hace en los  $n$  ejes del vector. Se concibe así

a la Recombinación K como una generalización de ambas, donde el parámetro  $k$  es la cantidad de ejes a promediar. Se muestra a continuación el pseudocódigo del algoritmo:

```

ejes = k ejes aleatorios, con k entre 1 y n
PARA CADA eje EN ejes:
    crial[eje] = α * x[eje] + (1 - α) * y[eje]
    crial2[eje] = α * y[eje] + (1 - α) * x[eje]
FIN

```

Todos los demás ejes se copian de los padres directamente. De esta forma la RecombinacionUnica es una instancia particular de la RecombinacionK donde  $k = 1$ .

Esta no forma parte de la bibliografía básica y es propuesta por el alumno. Se experimentará más adelante con distintos valores para  $k$  (véase la sección 5.7).

### 3.4.7 Recombinación Aritmética Alfa-Beta

Hasta ahora ninguna de las recombinaciones hacía uso de la función *fitness* para tomar alguna decisión. En los casos anteriores, los hijos estaban igualmente espaciados de sus padres, dependiendo del valor de  $\alpha$  (véase la Figura 3.1).

En esta recombinación, propuesta por el alumno, se generan los hijos a distinta distancia de sus padres. Para ello se utilizan los parámetros  $\alpha$  (para la distancia del mejor parent) y  $\beta$  (para la distancia del peor parent).

Idealmente se utiliza  $\alpha < \beta$ , para acercar los hijos al parent con mejor *fitness*, aunque se podría experimentar con lo contrario. Se aprecia el concepto en la Figura 3.2, donde se supone que los padres se encuentran a una distancia  $d$  y que  $fitness(x) > fitness(y)$ .



**Figura 3.2 - Representación de la Recombinación Alfa-Beta**

La implementación se encuentra en la clase RecombinacionAlphaBeta.

## 3.5 Mutación

Para cada tipo de representación existen un gran número de técnicas de mutación posibles. Está fuera del alcance de esta memoria enumerarlas todas, por lo que se detallarán a continuación aquellas aplicables a las cadenas de números reales e implementadas por el alumno. Las mismas se encuentran en el paquete `es.uned.pfg.ae.mutacion` y heredan todas de la interfaz `Mutación`.

La mutación se aplica a cada uno de los hijos y opera sobre cada alelo independientemente. Para ello se asigna una probabilidad de mutación, usualmente bastante pequeña.

### 3.5.1 Mutación Uniforme

Se le asigna al valor del alelo un número aleatorio distribuido uniformemente entre todo el dominio del problema. Si el problema no tiene restricciones numéricas, el valor sería cualquier número real.

Para la implementación se ha asumido que siempre existen restricciones para los valores mínimo y máximo, por lo que el código en la clase `MutacionUniforme` se reduce a:

```
for (int j = 0; j < alelos.length; j++) {  
    if (aleatorio.isMenorQue(prob)) {  
        alelos[j] = aleatorio.getEntre(min, max);  
    }  
}
```

El método `isMenorQue` genera un número aleatorio y lo compara con el valor de `prob` (probabilidad de mutación). Si es menor que el mismo, devuelve verdadero.

### 3.5.2 Mutación No Uniforme

Otra alternativa consiste en que la distribución de los números aleatorios generados no sea uniforme, es decir que habrá números con más probabilidad de ser elegidos. Estos números se sumarán al valor del alelo incondicionalmente o con una probabilidad alta. Es por ello que se elige una distribución que devuelva por lo general, pero no siempre, un valor cercano a cero.

En este trabajo se utiliza la distribución Normal centrada en cero y con una desviación estándar proporcionada por el usuario. Esta tiene la propiedad de que dos tercios de los valores que se generan yacen a una desviación estándar de distancia y casi el 99% dentro

de 3 desviaciones. Por tanto, existe una probabilidad no nula de que se genere un número desproporcionadamente grande. Es por ello que se optó por limitar al generador de números aleatorios para que no devuelva valores a más de 3 desviaciones estándar de distancia.

La implementación puede verse en la clase `MutacionNormal` y se muestra la parte relevante a continuación:

```
for (int j = 0; j < alelos.length; j++) {
    if (aleatorio.isMenorQue(prob)) {
        double mutacion = aleatorio.getNormal(0, desviacion);
        alelos[j] = Utils.clamp(alelos[j] + mutacion, min, max);
    }
}
```

Primero obtiene un número aleatorio con distribución normal centrada en cero y con una desviación parametrizable. En la siguiente línea suma dicho número al valor actual del alelo. Como el valor resultante puede terminar fuera del dominio de la función, se utiliza la función `Clamp` (restringir en inglés) para contener el valor. Esta función simplemente restringe el número que recibe por parámetro entre unos valores mínimo y máximo. Si el número estuviese fuera del rango, simplemente se lo mueve al valor más cercano (al máximo si lo supera por encima y al mínimo si lo hace por debajo).

## 3.6 Selección de Supervivientes y Modelo Poblacional

Una vez que se han generado y mutado los hijos, hace falta saber cuáles de ellos formarán parte de la población y, como el tamaño de esta debe permanecer constante, qué individuos de la generación anterior deberán ser reemplazados.

Con respecto al modelo poblacional, existen a grandes rasgos dos variantes:

- **Modelo generacional:** los hijos reemplazan la población entera de la generación anterior.
- **Modelo *steady-state*:** sólo una pequeña parte de la población de la generación anterior es reemplazada por los nuevos hijos. Para determinar qué individuos específicos deben reemplazarse, se utiliza alguna de las técnicas de selección, por lo que se utiliza nuevamente la función *fitness*.

Otra característica transversal al modelo particular es el concepto de elitismo. De utilizarse el mismo, se preserva de generación en generación al mejor individuo. De esta

forma se impide que este se pierda (recuérdese que existe una probabilidad no nula de elegir a individuos malos), por lo que el progreso del algoritmo será monótono creciente.

Las implementaciones se encuentran en el paquete `es.uned.pfg.ae.poblacion` y se codifican como subclases de la clase base `Poblacion`. Se ha implementado solamente uno de los modelos, en la clase `PoblacionGeneracional`.

# Capítulo 4: Aplicación de un Algoritmo Genético a la Optimización de Funciones Reales de Varias Variables

Se define el problema de optimización de funciones como encontrar o seleccionar un valor para las variables de entrada de una función, de manera tal que se maximice o minimice el valor de salida de la misma.

Las técnicas de optimización no sólo son para encontrar el máximo o el mínimo de una única función  $f$ . En muchos problemas del mundo real se aplican también a conjuntos  $F$  de  $n$  funciones objetivo  $f_i$ , cada una representando un criterio a ser optimizado. En estos casos, los problemas suelen llamarse *multi-objetivo*. La optimización *multi-objetivo* suele significar balancear el compromiso entre varias metas que entran en conflicto entre sí.

En el presente trabajo se aplica un AG a la minimización de funciones de varias variables que definen un único objetivo, con la única restricción que la función sea de la forma  $\mathbb{R}^n \rightarrow \mathbb{R}$ . Cada una de las  $n$  variables de entrada deberá estar restringida a un dominio de exploración (cota mínima y máxima), que debe ser el mismo para las  $n$  variables. Por lo demás, poco se asume sobre la estructura de la función, por lo que no existen más restricciones como continuidad, derivabilidad ni una dimensión máxima específica.

Cabe destacar que el funcionamiento del algoritmo es estocástico, iterativo, aproximado y no es analítico. Esto permite ser aplicado a una mayor cantidad de funciones (por ejemplo discontinuas y no derivables), aunque implica que el resultado obtenido no será siempre el óptimo global. Como se apreciará en el capítulo sobre evaluación experimental, se requerirán varias ejecuciones e incluso cambios en los parámetros para encontrar resultados de buena calidad.

## 4.1 El problema de la Minimización

Generalmente, los problemas de optimización suelen interpretarse como de minimización. Este es el convenio que se ha seguido en este trabajo, aunque presenta una serie de complicaciones.

La literatura suele considerar a los algoritmos genéticos como aquellos que, mediante técnicas evolutivas, maximizan la *fitness* de la población. Al igual que en la Naturaleza, aquellos individuos con mayor adaptación al entorno sobreviven. Es por ello que, también

se ha adoptado este convenio. Esto presenta un problema si el mapeo de la función  $\mathbb{R}^n \rightarrow \mathbb{R}$  a la función *fitness* se hace directamente. La solución adoptada en este trabajo puede verse en la clase `Individuo`:

```
public void calcularFitness() {
    fitness = funcion.getMinimoGlobal() - funcion.calcular(valores);
}
```

Por lo general se utilizan funciones cuyo mínimo global vale cero, por lo que efectivamente se estaría maximizando el opuesto de la función a minimizar, lo cual es equivalente. Es por ello que cuando se muestren las curvas de progreso, el gráfico comenzará en valores muy bajos y se acercará progresivamente a cero, la máxima *fitness* posible.

Con esto queda resuelto el problema, aunque produce ciertos inconvenientes en otras partes del algoritmo. Por ejemplo, no es posible implementar la Selección Proporcional a la Adecuación (*Fitness Proportionate Selection, FPS*), dado que requiere que el valor absoluto de la adecuación sea mayor para los individuos más aptos. En este caso, por el mapeo utilizado, el mejor individuo posible tendrá una adecuación de cero, por lo que la probabilidad de elegirlo sería nula.

Por lo general, en la literatura al respecto se asume o requiere que la *fitness* sea mayor que cero, aunque en este trabajo no se ha puesto tal restricción.

## 4.2 Algoritmo Implementado

Para la resolución del problema se ha implementado un Algoritmo Genético en el lenguaje Java. El núcleo del algoritmo se encuentra en la clase `AlgoritmoGenetico`, y sigue la misma estructura que el pseudocódigo del algoritmo genérico mostrado en el capítulo 2.

Quitando algunos detalles de implementación (por ejemplo, chequeo de valores nulos), el algoritmo general aparece a continuación:

```
public boolean iteracion(int iteracion) {
    Individuo[] padres = seleccion.seleccionar(poblacion);
    Individuo[] crias = new Individuo[padres.length];

    for (int i = 0; i < padres.length - 1; i += 2) {
        Individuo[] c = recombinacion.getCrias(padres[i], padres[i + 1]);

        for (Individuo cria : c) {
            mutacion.mutar(cria);
        }
    }
}
```

```

        crias[i] = c[0];
        crias[i + 1] = c[1];
    }

    poblacion.setSobrevivientes(crias);

    return terminacion.isTerminado(iteracion, poblacion);
}

```

Nuevamente aparecen las etapas que se vieron anteriormente. Para cada una de ellas existe una interfaz<sup>4</sup>, por lo que el algoritmo no necesita conocer las implementaciones. Esto fomenta una alta modularidad y bajo acoplamiento, ambas propiedades deseables para permitir la expansión de las distintas etapas sin cambiar el núcleo principal.

Un paso extra, no mostrado en el código anterior, es el de inicialización de la población. Si bien existen distintas alternativas (aleatorias, heurísticas, que garanticen cierta heterogeneidad, etc.), se ha optado por la inicialización aleatoria, aunque cambiarla por una nueva implementación no supondría gran dificultad, al estar dicha función en un módulo específico.

El algoritmo se ejecuta hasta que esta función devuelva Verdadero, indicando que debe dejarse de iterar. El criterio de terminación está encapsulado en la interfaz de terminación por lo que, nuevamente, puede cambiarse con sólo implementar una nueva interfaz.

El resto del programa, como se verá más adelante, consiste en las implementaciones de las interfaces mostradas así como código auxiliar para la interfaz de usuario, parametrización, reportes y gráficos, etc.

En las siguientes secciones se examina la herramienta software en general, atendiendo a su instalación, estructura, funcionamiento, etc.

### 4.3 Instalación

Como se verá más adelante, el programa posee un modo de ejecución local por consola y otro vía Web. Para este último no se necesita por tanto instalar el programa, siempre que se acceda al servidor público como se explica en la sección 4.7.3. De querer ejecutar el programa en el ordenador local, será necesario instalar la aplicación precompilada o compilarla manualmente.

---

<sup>4</sup> En algunos casos puede tratarse de una clase abstracta y no de una interfaz, aunque en el lenguaje Java ambos conceptos son muy similares.

La instalación de la aplicación precompilada requiere exclusivamente de la máquina virtual de Java JRE (*Java Runtime Environment*) en su versión 6 o posterior. Junto con este trabajo se entrega un archivo comprimido que contiene el programa precompilado así como los directorios adicionales necesarios para la ejecución, que se comentan en la sección 4.4. Una vez descomprimido el archivo, el programa estará listo para ser ejecutado. Para ello se utilizan los archivos `ejecutar.sh` o `ejecutar.bat`, dependiendo si la plataforma es Linux/OSX o Windows respectivamente. Se explica la ejecución y los argumentos disponibles en la sección 4.7.

#### 4.3.1 Instalación desde el Código Fuente

Aunque se recomienda utilizar la versión Web pública o el archivo precompilado, es posible ejecutar el programa compilando primero los archivos fuente y empaquetándolos en un archivo `jar`<sup>5</sup>. Esto es útil si se desea realizar cambios en alguno de los archivos y probarlos, o si se quiere obtener la última versión del programa, que podrá seguir recibiendo cambios incluso luego de finalizado y entregado el trabajo.

Para descargar el código fuente debe acudirse a la plataforma Github, que ofrece repositorios gratuitos para proyectos de código abierto. Para la descarga existen dos alternativas: clonar el repositorio utilizando la herramienta Git<sup>6</sup> o descargar un archivo empaquetado con la última versión del código y archivos auxiliares.

Si se dispone de Git, sólo es necesario teclear el siguiente comando en una terminal:

```
$ git clone "https://github.com/fermin-silva/pfg.git"
```

Esto creará un directorio llamado `pfg` con el contenido del proyecto. Si se había clonado el repositorio con anterioridad y sólo quiere actualizarse el contenido a la última versión, es necesario situarse en el directorio `pfg` y ejecutar:

```
$ git pull origin master
```

Esto descargará las diferencias en el directorio desde la última vez que se actualizó.

De no disponer de Git puede descargarse la última versión del código accediendo al siguiente enlace: <https://github.com/fermin-silva/pfg/archive/master.zip>. Cada vez que se

---

<sup>5</sup> Puede verse más sobre el formato `jar` en el siguiente enlace:

<http://docs.oracle.com/javase/1.5.0/docs/guide/jar/jar.html>.

<sup>6</sup> Puede obtenerse más información de la herramienta y descargarla desde <https://git-scm.com/>

cambie algún archivo del repositorio, el archivo comprimido se regenerará, por lo que no hace falta cambiar el enlace, el cual siempre contendrá la versión más reciente. Después de haberse descargado, se debe descomprimir el archivo en un directorio llamado pfg-master.

En ambos casos, se dispondrá de la estructura completa de directorios, aunque no la versión precompilada del código. Para poder ejecutar el programa hacen falta dos pasos:

1. Compilar los archivos fuente con extensión .java a archivos objeto con extensión .class.
2. Empaquetar los archivos .class en un único archivo, comprimido o no, con extensión .jar (aunque es esencialmente un archivo .zip con ciertas estructuras específicas).

Para ambos pasos es necesario disponer de la máquina virtual de Java y las herramientas de desarrollador, denominadas JDK (*Java Development Kit*), también en su versión 6 o posterior. Nótese que de tener instalado el entorno JDK, no es necesario instalar el JRE para la ejecución, ya que el primero incluye al segundo.

Para simplificar la compilación, se ha incluido un pequeño programa auxiliar llamado `compilar.sh` (para Linux y OSX) y otro `compilar.bat` para Windows<sup>7</sup>. Estos realizan los pasos 1 y 2 conjuntamente y crean el archivo `pfg.jar`, listo para ser ejecutado. Ante cualquier inconveniente, se recomienda al lector ver el código fuente de dichos programas auxiliares.

## 4.4 Estructura de Directorios

Una vez instalado el programa, y antes de entrar en la estructura específica del código fuente, es necesario comentar sobre la estructura general de directorios.

Ya sea si se utiliza el archivo comprimido adjunto o se descarga online desde la plataforma Github, una vez descomprimido el archivo se encontrará la siguiente estructura de directorios:

- **doc:** contiene el archivo fuente del anteproyecto (escrito en *Markup*) y la memoria (escrita en Word).

---

<sup>7</sup> En Windows es necesario, dependiendo del tipo de instalación, agregar a la variable de entorno PATH la ruta del entorno de desarrollo JDK. De lo contrario, cuando el programa invoque el comando javac se obtendrá un error de comando no encontrado. Más información en

<http://docs.oracle.com/javase/7/docs/webnotes/install/windows/jdk-installation-windows.html#path>

- **img**: contiene imágenes que se utilizan en el anteproyecto y en la wiki dentro de la plataforma Github.
- **lib**: librerías externas requeridas para compilar y ejecutar el código principal.
- **src**: directorio con el código fuente escrito por el alumno
- **target**: comúnmente los programas Java utilizan un directorio con este nombre para colocar el código objeto. Se opta por seguir este convenio y, cuando se compile el programa, se utilizará este directorio para contener el código objeto intermedio.
- **test**: contiene algunos tests unitarios para probar aisladamente el correcto funcionamiento de ciertas partes del código. Requieren JUnit para ser ejecutados.
- **tmp**: directorio para archivos temporales utilizado por el programa. Puede borrarse su contenido en cualquier momento sin afectar el funcionamiento.
- **web**: contiene archivos necesarios para la visualización de la interfaz gráfica vía Web. Se verán los detalles al respecto de la interfaz Web más adelante en este capítulo.

En cuanto a los requisitos, se recomienda conservar todos los directorios en todo momento. También que tanto la aplicación como los programas de compilación tengan permisos de ejecución y de escritura en al menos los directorios tmp, target y la raíz.

## 4.5 Estructura del Programa

Las aplicaciones Java se separan en lo que se conocen como paquetes. Los mismos pueden considerarse una estructura de directorios o módulos, aunque ofrecen también un espacio de nombres (por si hubiera dos clases con el mismo nombre). El nombre de un paquete se forma concatenando los nombres de los paquetes padres con un punto, de manera similar a como se realiza con los directorios (con una barra). Así, el paquete base, común a todos los subpaquetes es es.uned.pfg.ae. En este se encuentran las clases principales para comenzar la ejecución así como otras clases que, por ser muy heterogéneas y únicas, no son agrupables entre sí.

A continuación se enumeran y explican los demás paquetes, siempre hijos del paquete base.

### 4.5.1 Paquete Chart

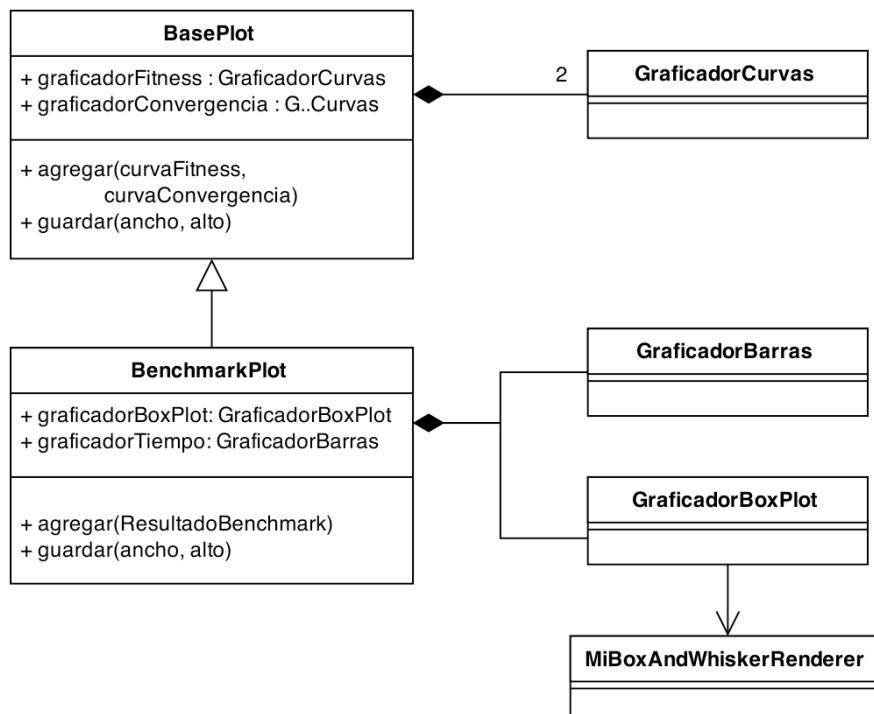
Agrupa las clases orientadas a generar gráficos para mostrar el comportamiento y rendimiento del algoritmo.

Existen distintos modos de ejecución, que se verán más adelante. Cada uno de ellos ejecuta el algoritmo genético de alguna manera y necesita distintos tipos de gráficos. Dentro del paquete hay dos tipos de clases: *graficadores* y *plots*.

Los **graficadores** se encargan de realizar un gráfico específico, utilizando la librería JFreeChart. Así se tienen las clases *GraficadorBarras*, *GraficadorBoxPlot* y *GraficadorCurvas*. Nótese que esto es sólo un convenio de nombres y no una jerarquía de clases.

Los **plots** agrupan graficadores para algún modo de ejecución en particular, parametrizando los nombres de los gráficos, datos a graficar, etc. Se tiene la clase *BasePlot* para las ejecuciones simples del algoritmo (que sólo requieren curva de progreso y de convergencia) y la clase *BenchmarkPlot*, subclase de la primera, que agrega gráficos específicos para las pruebas empíricas.

En la Figura 4.1 se muestra un diagrama de clases parcial de este paquete. La clase *MiBoxAndWhiskerRenderer* se explica en el apartado siguiente.



**Figura 4.1 - Diagrama de clases del paquete chart**

#### 4.5.2 Paquete org.jfree.chart.renderer.category

Este paquete no se encuentra dentro del paquete base y es más una necesidad que una decisión. La clase *BoxAndWhiskerRenderer* de la librería JFreeChart posee algunas partes

que no son configurables o no funcionan correctamente, por lo que ha sido necesario sobrescribir partes de su funcionamiento.

Si bien normalmente con extender la clase original como una subclase nueva sería suficiente, secciones del código no son extensibles, especialmente por los modificadores *final* o *private* de Java.

Un procedimiento alternativo, aunque poco ortodoxo, consiste en copiar el código fuente de la clase en una nueva y modificar las partes relevantes, sin utilizar herencia. Esto supone un problema cuando se utiliza el modificador *protected* de Java, que proporciona visibilidad de paquete. En este caso basta con ubicar la clase en el mismo paquete que la clase original, con lo que se obtiene una clase equivalente, que el usuario puede modificar.

Es por todo ello que se ha tenido que crear este paquete, con una única clase llamada `MiBoxAndWhiskerRenderer`, en honor a la clase original.

#### 4.5.3 Paquete Función

Este paquete contiene la interfaz genérica `Funcion`, así como implementaciones específicas para probar el algoritmo (`FuncionRastrigin`, `FuncionSchwefel`, etc.).

El método principal que las funciones deben implementar es `calcular`, que recibe el arreglo de números reales y devuelve el resultado de evaluar la función. Otros métodos importantes son `getMin` y `getMax` que se utilizan para confinar los valores de los genes, y que actúan como cota inferior y superior del dominio de la función.

En este paquete se encuentra también la clase `FuncionFactory`, que sigue el patrón *Factoría* para crear una instancia concreta de la función a minimizar, ya sea leyendo los parámetros de entrada o indicándole el nombre de la función deseada.

No se incluye un diagrama de clases al ser este extremadamente simple. No hay herencias ni composiciones, solamente una interfaz y seis implementaciones independientes.

#### 4.5.4 Paquete Mutación

Similar al paquete de funciones, contiene la interfaz base `Mutacion`, que todas las mutaciones deben implementar, así como la factoría para instanciar la implementación concreta en tiempo de ejecución.

El método principal de la interfaz es `mutar`, que recibe un individuo al que mutar. Para ello la clase necesitará acceso a la estructura interna del individuo (a sus genes, es decir el

vector de números reales). La función tiene efectos secundarios ya que modifica el estado interno del individuo, en vez de devolver un individuo nuevo mutado. Se ha optado por esta alternativa por motivos de rendimiento.

Al igual que en todo el resto de los operadores, se incluye un operador por defecto de no operación (*NoOp*), que se utiliza por defecto para deshabilitar una sección del algoritmo. Existe por tanto una `MutacionNoOp`, que no realiza ninguna modificación a los individuos, por lo cual es equivalente a saltar dicha etapa en el algoritmo.

#### 4.5.5 Paquete Selección

Su estructura es equivalente a la del paquete `mutacion`, en tanto que existe una factoría, una interfaz base y una serie de implementaciones del operador de selección de padres (incluida la `SeleccionNoOp` que no produce ningún efecto).

Nuevamente no se incluye el diagrama de clases por ser su estructura trivial.

#### 4.5.6 Paquete Recombinación

Aunque con una jerarquía de herencia un poco más compleja (véase la Figura 4.2), sigue la misma estructura que la mutación y selección. Las implementaciones se dividen en las recombinaciones discretas (BGA y Un Punto) y las aritméticas. Estas últimas heredan todas de una clase abstracta base llamada `RecombinacionAlpha`. Esta clase no realiza nada, solamente almacena y maneja el valor de `Alfa` que todas las demás clases utilizan.

Por simplicidad no se incluye en la Figura 4.2 la factoría, ya que esta tiene dependencias o relaciones con todas las demás clases, por lo que habría que trazar una flecha desde su diagrama a todos los demás, complicando la comprensión del gráfico.

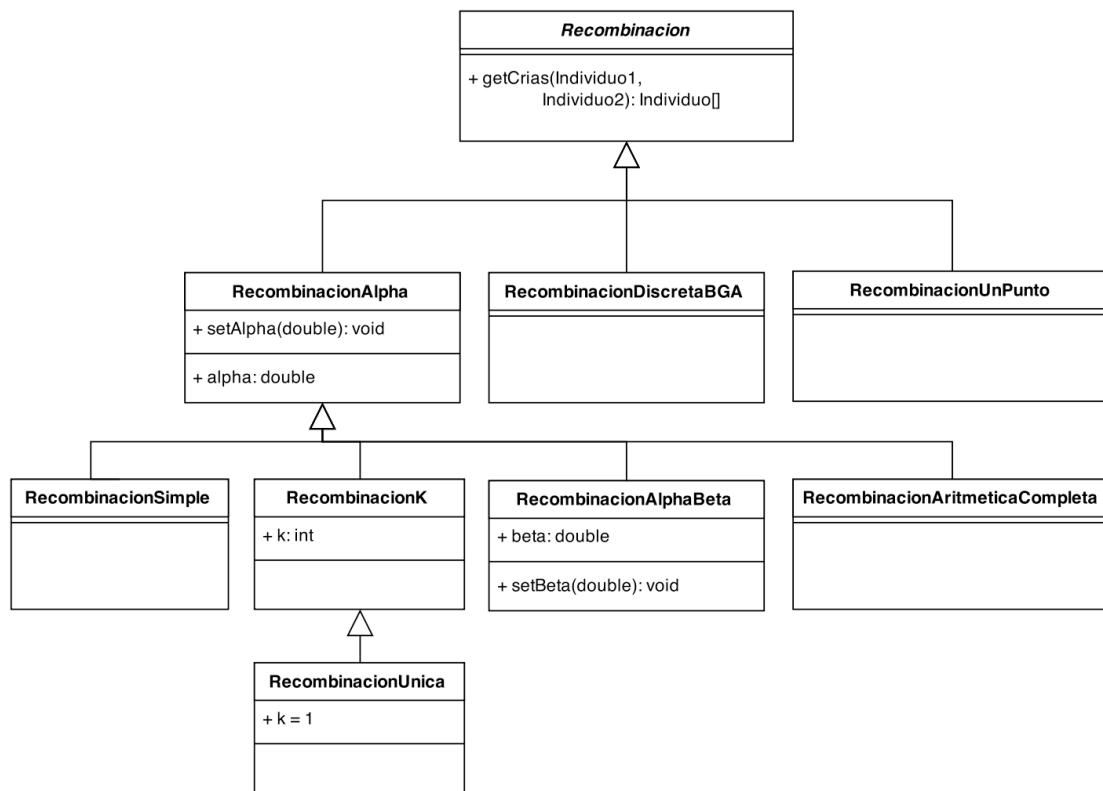


Figura 4.2 - Diagrama de clases de la Recombinación

#### 4.5.7 Paquete Terminación

No sigue exactamente el mismo patrón que los otros operadores, aunque está preparado para funcionar de la misma manera. Se dispone de la interfaz base `Terminacion` y, por el momento, sólo la implementación `TerminacionFija`, que termina cuando se hayan realizado una cantidad determinada de generaciones.

Podría extenderse de la misma forma que los otros paquetes: creando implementaciones alternativas y una factoría que permitiese elegir la instancia concreta en tiempo de ejecución.

#### 4.5.8 Paquete Utils

Paquete que contiene clases con utilidades varias, poco cohesivas entre sí, aunque tienen en común que se utilizan solamente para agrupar funciones de ayuda, auxiliares a otras secciones del código.

Se tiene así la clase `Utils` (agrupación de métodos estáticos de ayuda), `Aleatorio` (generador de números aleatorios) y `MultiMap` (clase con algunos métodos para acceder más cómodamente a la interfaz `Map` de Java).

#### 4.5.9 Paquete Población

Aunque por el momento existe sólo el modelo generacional de población (implementado en la clase `PoblacionGeneracional`), se incluye la interfaz base `Poblacion`, para permitir la extensibilidad futura. En este caso se trata en realidad de una *clase abstracta* y no una interfaz, ya que la propia clase contiene algunos métodos ya implementados. Por lo demás, actúa de manera idéntica a una interfaz.

El método de interés es `setSobrevivientes`, al que se le pasan los nuevos hijos y deberá decidir qué hacer con ellos y con los individuos de la generación anterior: reemplazar todos, reemplazar los más antiguos, etc.

Nuevamente, de querer extender la funcionalidad, podría adoptarse el mismo patrón que en los otros paquetes y crear una factoría que permita cambiar la implementación concreta, siempre y cuando herede de la clase base `Poblacion`.

#### 4.5.10 Paquete Params

Contiene las clases auxiliares para el manejo y validación de parámetros por línea de comandos con ayuda de la librería JCommander (véase la sección 4.6 sobre librerías auxiliares).

La clase `Parametros` engloba todos los parámetros posibles. Para ello sólo basta con declarar una variable con su tipo y anotarla, utilizando el operador @ de Java. Esta anotación permite elegir el nombre del parámetro, la descripción para mostrar ayuda del mismo y si requiere validaciones.

Para las validaciones deben crearse clases que implementen la interfaz `IValueValidator`. Se tienen así las clases `Validador01`, que valida que un `double` se encuentre entre [0, 1], y `MayorQueCero`, que valida que el parámetro sea numérico y mayor que cero.

##### Ejemplo:

Se muestra a continuación una porción de código para declarar un parámetro, por ejemplo el valor de *k* para la `RecombinacionK`.

```
@Parameter(names = { "-puntosCombinacion", "-k" },
           validateValueWith = MayorQueCero.class,
           description = "Insertar aquí la descripción")
public Integer puntosCombinacion;
```

Primero se utiliza la notación @Parameter para indicar que se trata de un parámetro de línea de comandos. A continuación se indican el o los nombres que el usuario deberá ingresar en la línea de comandos para identificar al parámetro. A continuación se declara el validador para el valor tecleado, seguido de una descripción en el caso de que el usuario necesite ayuda sobre el parámetro. Finalmente se incluye una variable Java que JCommander instanciará si el parámetro está presente.



#### 4.5.11 Paquete Web

En este paquete se engloban todas las clases necesarias para ejecutar un servidor Web/HTTP. Este se utiliza para poder ejecutar el AG vía Web desde un navegador de Internet. Se explicará más sobre esta modalidad en la sección 4.7.3 sobre modos de ejecución, aunque se comentan aquí las clases principales que intervienen en dicha modalidad.

La Figura 4.3 muestra un diagrama UML de clases. Recuérdese que el triángulo representa herencia, el rombo composición y la flecha simplemente uso o asociación.

La clase principal de este paquete es `Servidor`, que configura la librería Jetty para crear el servidor propiamente dicho (variable `Server`). Este escuchará la actividad HTTP en un puerto configurable y enviará el pedido a los manejadores (que deben implementar la clase `AbstractHandler`) configurados en el método `agregarHandlers`. Si un manejador no sabe cómo resolver la petición, Jetty prueba con el siguiente hasta que alguno sepa responder.

En la clase `Servidor` se configuran dos manejadores: el primero es para servir archivos estáticos desde el disco duro hacia la web (`ResourceHandler`), el segundo, hecho por el alumno, permite encaminar pedidos a distintas clases según la dirección URL configurada, titulado `BaseHandler`.

Este manejador está compuesto por un mapa de direcciones URLs a recursos Web (que deben implementar la clase `Recurso`). Por el momento sólo se mapea la dirección `/api/ag` a la clase `RecursoAG`, aunque en el futuro pueden agregarse más recursos, siempre que tengan una URL única.

Con esto, cada vez que se acceda a dicha dirección en un navegador se ejecutará el código en la clase `RecursoAG`. Esta clase es simplemente un punto de acceso vía Web, por lo que suele mantenerse sólo para ello, delegando la lógica de ejecución a una clase específica, normalmente un *controlador*. Dicha lógica se encuentra en la clase `ControladorAG`, que

realiza la ejecución del algoritmo genético y prepara la respuesta para devolver nuevamente vía Web.

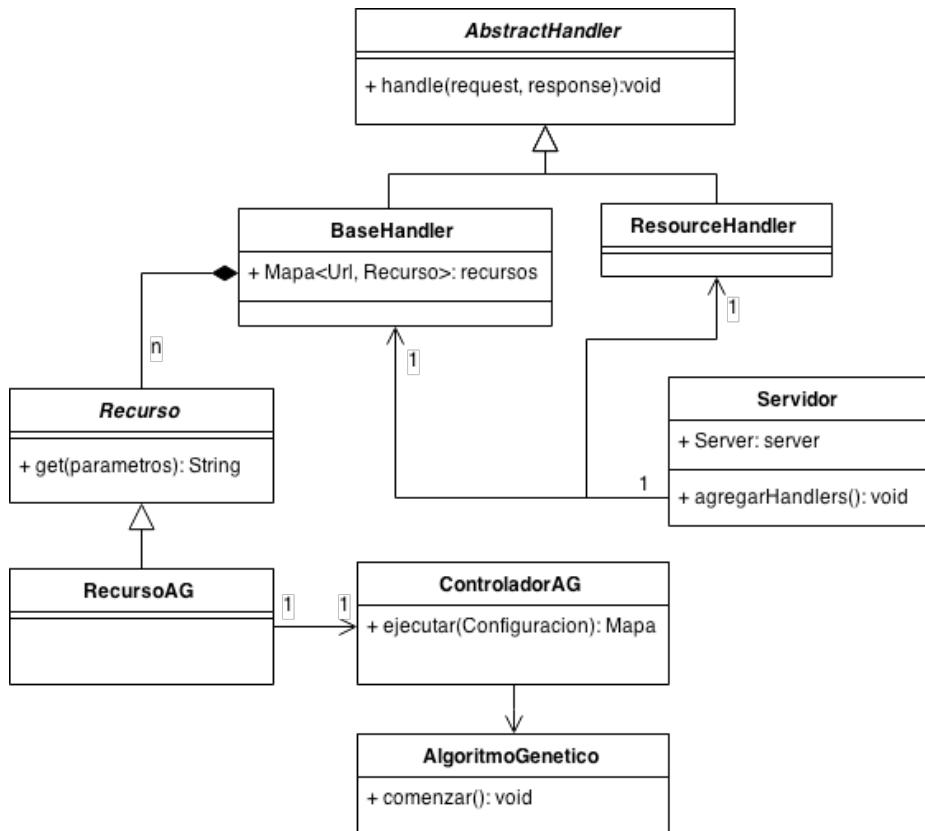


Figura 4.3 - Diagrama de clases del paquete Web

## 4.6 Librerías Auxiliares y Motivación de las Mismas

El núcleo del programa ha sido escrito enteramente por el alumno. Pero existen funcionalidades periféricas o secundarias que son necesarias para el programa en su conjunto. Para ellas se ha optado por acudir a librerías externas, siempre de código libre y abierto, todas ellas disponibles en Internet. En este apartado se discute brevemente cuáles son, así como una breve motivación de las mismas.

Más allá de las motivaciones específicas de cada librería, existen motivaciones transversales a todas ellas. Estas incluyen no reinventar la rueda, utilizar librerías maduras, fiables y probadas (gracias a que son utilizadas y mejoradas por la comunidad), acelerar el proceso de desarrollo, etc.

#### 4.6.1 JCommander

Su página principal es <http://jcommander.org/> y el código fuente puede obtenerse en <https://github.com/cbeust/jcommander>. El propósito de esta librería es el análisis y procesamiento de los parámetros enviados por línea de comandos.

Al ser la aplicación altamente parametrizable, es necesaria una manera de poder manejar dichos parámetros con comodidad. Esto incluye:

- El análisis léxico y sintáctico (*parsing*): determinar cuál es cada nombre y valor de parámetro y en qué posición se encuentran.
- La validación: que sea del tipo esperado, que sea o no requerido, que tenga determinado valor (por ejemplo menor que 1), etc.
- El procesamiento: convertir el valor al tipo esperado (número, booleano, cadena de caracteres, etc.).
- La descripción y ayuda: cada parámetro tiene una descripción asociada y, si se ejecuta la aplicación con el parámetro -h, se imprimirá una ayuda con todos los parámetros disponibles y su función.

Puede verse la utilización de esta librería en la clase `Parametros`, dentro del paquete `es.uned.pfg.ae.params`.

#### 4.6.2 JFreeChart

Su página principal es <http://www.jfree.org/jfreechart/> y el código fuente puede obtenerse en <http://sourceforge.net/projects/jfreechart/>.

La aplicación genera gráficos para mostrar las curvas de progreso, convergencia de la población, etc. Para ello puede utilizarse el paquete `java2d`, que incluye funcionalidades para dibujar primitivas básicas en 2 dimensiones. Sin embargo, es un paquete de bajo nivel, por lo que para realizar gráficos complejos requiere gran cantidad de código.

La librería JFreeChart abstrae el uso de `java2d`, para utilizar comandos de más alto nivel y crear así gráficos complejos con pocas líneas de código.

Otra alternativa hubiese sido delegar dicha tarea a alguna librería del sistema, como `GNUPlot`, aunque eso hubiese afectado la portabilidad del programa. Se utiliza por ende una solución 100% Java y autocontenido en el propio programa, por lo que no se requiere así instalar librerías externas en el propio sistema.

### 4.6.3 JUnit

Su página principal es <http://junit.org/> y el código fuente puede obtenerse en <https://github.com/junit-team/junit>.

Permite ejecutar pruebas unitarias para verificar el correcto funcionamiento de porciones de código de manera aislada.

No es requisito para que la aplicación funcione, aunque tener una alta cobertura de pruebas unitarias es buena práctica.

### 4.6.4 Jetty

Su página principal es <http://eclipse.org/jetty/>. No hay un repositorio público accesible de forma sencilla, aunque puede descargarse el código desde repositorios de terceras partes.

En <http://repo1.maven.org/maven2/org/eclipse/jetty/aggregate/jetty-all/> se pueden encontrar todas las versiones junto con su código fuente.

El propósito de esta librería es incorporar un servidor HTTP en el propio programa. El mismo se utiliza como punto de entrada alternativo a la línea de comandos, interactuando directamente con un navegador de Internet.

El estándar HTTP no es trivial y una implementación completa y segura requeriría de varios años de trabajo, por lo que se utiliza uno de los servidores HTTP más conocidos y estables del lenguaje Java.

## 4.7 Ejecución y Comportamiento del Programa

Ya se ha visto el algoritmo principal y cómo resuelve el problema de minimización. Sin embargo, aún es necesario unir todas las partes auxiliares, la instanciación de los operadores y configuración del algoritmo completo. Todo ello da lugar a distintas maneras de ejecutar el programa, que se verán a continuación.

En cualquier caso, el programa tiene un único punto de entrada, la clase `Bootstrap`, que se encarga de unir todos los módulos y ponerlos en marcha. El motivo del nombre es mero convenio y puede traducirse como “*ponerse las botas*”. Esta clase tiene el único método `main` del programa, requisito en Java para poder iniciar la ejecución.

Una vez dicha clase está en ejecución, el primer paso es instanciar la clase `Configuracion`, que no es más que un contenedor de los parámetros (ya sean por línea de comandos o en un archivo de *properties*). Luego hay tres ramas de ejecución, que se deciden

en base a la configuración en un *if* ternario: ejecución por *consola*, correr el *benchmark* o iniciar en modo *servidor*. A continuación se explican cada una de ellas en detalle.

#### 4.7.1 Ejecución por Consola

Es el modo más básico de ejecución, aunque el menos amigable para el usuario. Puede verse el código en la misma clase `Bootstrap`, en el método `correrConsola`. Allí se llaman a todas las factorías para instanciar los operadores principales, se crea la población inicial y finalmente se crea el algoritmo genético. Después de ejecutarlo hasta que se cumpla el criterio de terminación, se recolectan estadísticas para imprimir por consola, además de generar los gráficos con los resultados en el propio directorio de ejecución.

Para ejecutarlo basta con invocar el script correspondiente (dependiendo si el entorno es Windows o Linux/OSX), seleccionando los parámetros adecuados. Para ello es necesario haber instalado la aplicación correctamente y situarse en el directorio raíz del proyecto.

Téngase en cuenta que los parámetros son sensibles a minúsculas y mayúsculas, aunque no a la posición. La forma de los mismos es `-{nombre parámetro} {valor}`. Para los parámetros booleanos (por ejemplo el *elitismo*) sólo basta con que aparezcan, ya que no necesitan ningún valor.

Se muestran a continuación ejemplos ejecutados en un entorno Linux. De ejecutarse en Windows basta con cambiar el comando `sh ejecutar.sh` por `ejecutar.bat`, siendo el resto de los parámetros idénticos.

##### Ejemplo de ayuda:

Para conocer todas las opciones disponibles, sus valores permitidos y una breve descripción sobre los mismos, puede invocarse la ayuda:

```
$ sh ejecutar.sh -h
Usage: <main class> [options]
Options:
-alpha
    Para las recombinaciones que utilizan un valor
    alpha para el promedio ponderado. En archivo de
    properties es 'alpha'
-beta
    Para las recombinaciones que utilizan un valor beta
    para el promedio ponderado de la segunda crial. En
    archivo de properties es 'beta'
-desviacion
    Para la mutacion normal, el valor de la desviacion
    tipica. En archivo de properties es
    'desviacion.mutacion'
```

```
-recombinacion, -r
```

El nombre de la recombinacion a utilizar. De no utilizar ninguna se toma por defecto la RecombinacionNoOp (que no realiza nada). En archivo de properties es 'recombinacion'

...

Así se imprimirán todos los parámetros y, para los operadores, el nombre de los valores posibles.

Puede apreciarse que existen múltiples nombres o notaciones para algunos parámetros, como por ejemplo para la recombinación. Las notaciones alternativas suelen ser una larga y descriptiva (p.ej. *-recombinacion*) y otra más corta y cómoda (*-r*), útil para usuarios más avanzados o cuando la cantidad de parámetros es elevada. Ambas tienen el mismo efecto, por lo que elegir una u otra es elección del usuario.



#### Ejemplo sencillo:

```
$ sh ejecutar.sh -r Simple -f Rastrigin
```

En este caso sólo se especifica la recombinación Simple y la función Rastrigin. Los demás parámetros siempre tendrán un valor por defecto: No Operación en el caso de los operadores, 2 dimensiones para la función, etc. Pueden consultarse los valores por defecto en la clase Configuracion.



#### Un ejemplo completo:

```
$ sh ejecutar.sh -g 100 -t 2000 -s Torneo -tamañoTorneo 2 -m Normal  
-desviacion 1 -r K -k 2 -d 15 -alpha 0.1 -f Schwefel
```

Donde:

- **-g** : ejecuta el algoritmo durante 100 generaciones (iteraciones)
- **-t** : tamaño de población de 2000 individuos
- **-s** : operador selección por Torneo
- **-tamañoTorneo** : torneo binario (tamaño 2). Puede utilizarse este parámetro aunque la selección no sea por torneo, en cuyo caso no tendrá efecto.
- **-m** : el operador de mutación, en este caso mutación Normal

- `-desviacion` : sólo aplicable a la mutación normal, indica la desviación estándar para la distribución normal de números aleatorios.
- `-r` : operador de recombinación, en este caso Recombinación K.
- `-k` : para la Recombinación K, la cantidad de ejes (valor de K), valiendo 2 en este caso.
- `-d` : dimensión de la función elegida, en este caso 15 dimensiones. Recuérdese que algunas (como Shaffer2) tienen siempre dimensión 2, por lo que este parámetro se ignorará, aunque no generará ningún problema.
- `-alpha` : valor de alfa para la recombinación, en este caso 0.1.
- `-f` : la función a minimizar, en este caso la función Schwefel.



#### Ejemplo de salida:

Se muestra a continuación un ejemplo de la salida del programa después de una ejecución en particular. Primero se imprimen los parámetros de ejecución, ya sea los elegidos por el usuario o aquellos por defecto. Luego se imprimen los resultados de la ejecución y finalmente los archivos generados con las gráficas de rendimiento.

```
Funcion: F_Schwefel(10)
Seleccion: SeleccionTorneo (2)
Recombinacion: RecombinacionK (2, 0.1)
Mutacion: MutacionNormal (1.0)
Tamaño poblacional: 2000
Generaciones: 100
```

```
-----  
Progreso finalizado  
-----
```

Tiempo de Ejecucion: 393ms

Mejor Individuo

Fitness: -0.000

Coordenadas:

```
x1: +420.968
x2: +420.968
x3: +420.969
x4: +420.968
x5: +420.968
x6: +420.970
x7: +420.970
x8: +420.968
x9: +420.967
x10: +420.968
```

Centroide de la Poblacion (promedio):

```
x1: +420.969
x2: +420.970
x3: +420.969
x4: +420.968
x5: +420.969
x6: +420.968
x7: +420.969
x8: +420.968
x9: +420.968
x10: +420.968
```

Desviacion Tipica de la poblacion con respecto al centroide: +0.060

Guardando el archivo en /Users/dev/workspace/pfg./Schwefel(10)\_momento.png  
Guardando el archivo en /Users/dev/workspace/pfg./Schwefel(10)\_progreso.png



#### 4.7.2 Ejecución en Modo Benchmark

Para las pruebas empíricas de evaluación se ha optado por crear un modo de ejecución nuevo. Otra alternativa hubiese sido ejecutar el modo consola repetidas veces y anotar los resultados, aunque eso resultaría en un proceso tedioso y difícilmente repetible.

El modo benchmark o de comparación utiliza las mismas librerías de generación de gráficos y el núcleo del algoritmo, aunque las configura de distintas formas. El código puede verse en la clase Benchmark del paquete principal y el pseudocódigo puede resumirse de la siguiente manera:

```
Instanciar la SELECCIÓN s
Instanciar la MUTACIÓN m
Instanciar la TERMINACIÓN t
PARA CADA Función f implementada
    PARA CADA Recombinación r implementada
        Ejecutar n veces el AG con (s, m, t, f, r)
        Recolectar estadísticas y generar los gráficos
    FIN PARA
FIN PARA
```

Como se verá en el capítulo de evaluación experimental, en el presente trabajo se estudia el operador de Recombinación. Es por ello que se encuentra en el bucle interno. El bucle externo prueba el algoritmo entero con cada una de las funciones implementadas. Esto se repite  $n$  veces para calcular promedios, mínimos, máximos, etc.

Para ejecutar este modo sólo hace falta agregar el parámetro `-benchmark` al script de ejecución. Las demás opciones (como tamaño de población, dimensión de la función, etc.) pueden elegirse por el usuario y afectarán a todas las ejecuciones. Esto puede ser un problema si se desea probar las distintas funciones con, por ejemplo, un tamaño de población distinto. Tal caso tiene sentido considerando que las funciones tienen distinta complejidad, por lo que sería deseable probar cada una de ellas con un tamaño de población “justo”. En cualquier caso, se deja dicha funcionalidad como una posible expansión futura.

La salida de la ejecución es una serie de gráficos para cada una de las funciones, que comparan las recombinaciones entre sí.

#### 4.7.3 Ejecución en Modo Servidor

Para simplificar el uso de la aplicación es necesario incorporar alguna interfaz de usuario gráfica. Las alternativas son esencialmente dos: aplicación de escritorio o aplicación Web. La primera utiliza las librerías AWT o Swing de Java para crear una aplicación gráfica en vez de consola. Una vez que se ejecuta, se abre una ventana de manera similar a las otras aplicaciones del sistema.

La segunda alternativa consiste en ejecutar un servidor HTTP por línea de comandos. La visualización se realiza luego en una página Web, que debe ser accedida en un navegador

de Internet como Firefox o Chrome. Para ello, el navegador se comunica con el servidor mediante el protocolo HTTP estándar, quien le envía la información necesaria para visualizar la página.

En el presente trabajo se ha optado por la segunda alternativa, que presenta una serie de ventajas:

- Separación entre la ejecución del algoritmo y la visualización de sus resultados. Podrían tratarse incluso de programas distintos.
- Gracias a lo anterior, permite que la visualización se realice en un ordenador (denominado normalmente *host*) distinto a aquel que genera los datos.
- Permite compartir la aplicación con más facilidad, simplemente enviando la dirección web a la persona con la que se desea compartir, sin necesidad de instalación.
- Permite cambios y actualizaciones en un servidor central sin que los clientes o usuarios tengan que descargar nuevamente la aplicación.
- Utiliza para la visualización estándares de HTML, CSS y Javascript, ampliamente difundidos en la comunidad, de los cuales además pueden descargarse plantillas con diseños reutilizables.
- Permite disponer de varias fachadas intercambiables para la visualización, con tan solo crear nuevas páginas HTML, reutilizando el servidor central.

Para ejecutar la aplicación en modo servidor, solamente es necesario indicarle en qué puerto<sup>8</sup> debe escuchar los pedidos HTTP, mediante el parámetro `-port`.

Una vez ejecutándose, el servidor escuchará indefinidamente la actividad del puerto, por lo que el programa nunca terminará. Para finalizar la ejecución sólo hace falta cerrar la línea de comandos o enviar la señal de terminación `CTRL+C`.

Dado que el ciclo de vida del programa es algo más complicado que en una aplicación estándar (inicio, ejecución, final), se explicarán a continuación algunos flujos básicos así como las distintas topologías de ejecución: en un host remoto y en uno local. Luego se mostrarán las distintas pantallas accesibles desde el navegador.

---

<sup>8</sup> Recordar que las comunicaciones en red requieren un puerto (número entero entre 0 y 65535) y que para el caso de HTTP, se utiliza el protocolo TCP/IP.

## Flujo básico a alto nivel

Una vez que el programa se esté ejecutando en modo servidor, quedará a la espera de pedidos HTTP. Cuando llegue uno de la red (local, LAN, WAN, etc.), será la librería Jetty quien inicie la ejecución, como se muestra en la Figura 4.4.

Puede apreciarse que Jetty prueba encadenadamente los manejadores (*handlers*), hasta que alguno sea capaz de responder. Si ninguno puede, se retorna un *error 404: recurso no encontrado*. Puede encontrarse una explicación más detallada en el apartado 4.5.11, donde se explican las clases involucradas en el paquete web.

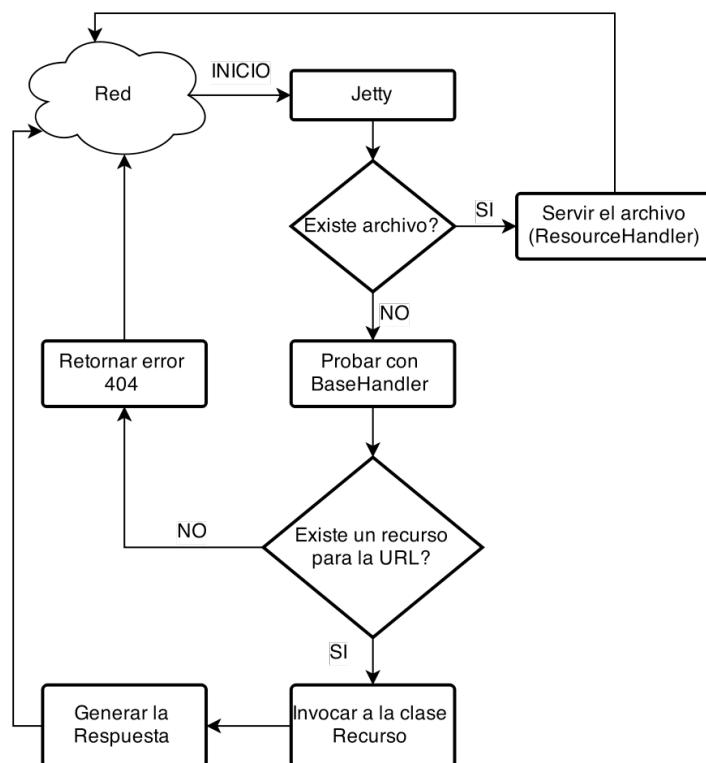


Figura 4.4 - Flujo básico de un pedido HTTP

Una vez devuelta la respuesta a la red, el servidor vuelve a estado de espera, hasta que reciba un nuevo pedido.

A su vez, Jetty dispone de una colección de hilos de ejecución o *threads* (por defecto alrededor de 200), por lo que este diagrama puede repetirse concurrentemente si varios usuarios están utilizando el servidor en el mismo instante.

## Ejecución en el Servidor Público

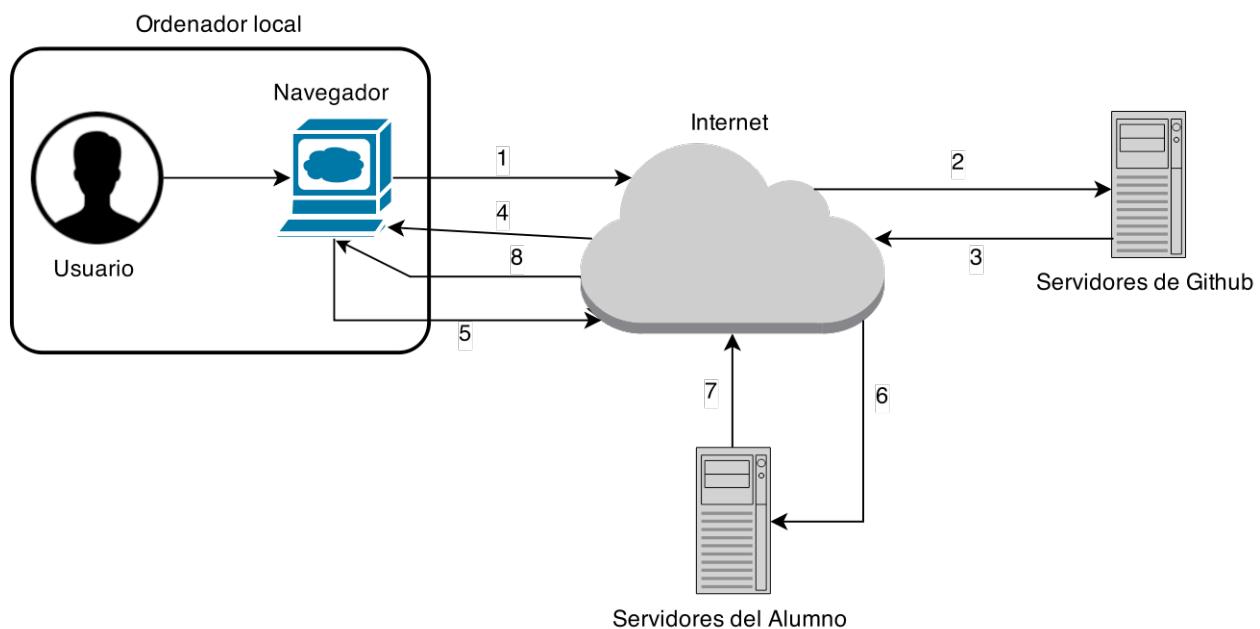
La manera más sencilla de ejecutar la aplicación consiste en utilizar el servidor público, provisto por el Alumno, accesible en <http://fermin-silva.github.io/pfg/>. Para ello, el único

requisito es disponer de, además de conexión a Internet, un navegador Web moderno como Firefox 36+, Chrome 38+ o Internet Explorer 9+. El número indica la versión mínima soportada. Aunque ciertas partes son compatibles con versiones más antiguas, puede que algún elemento se visualice incorrectamente. Por tanto, no es necesario haber instalado el programa con anterioridad.

Al acceder a dicha dirección Web se visualizarán páginas estáticas (es decir archivos HTML estáticos, no generados dinámicamente por un servidor), que son una copia de los archivos contenidos en el directorio *web* de este proyecto. La única diferencia con respecto a las incluidas en este trabajo es la dirección del servidor HTTP al que apuntan que, en vez de ser uno local, es uno provisto por el alumno.

Esta prueba de concepto es para facilitar el acceso a los evaluadores, para realizar demostraciones en vivo y, finalmente, para probar la separación entre visualización y ejecución, característica principal de las aplicaciones de tipo servidor.

Se muestra en la Figura 4.5 un diagrama de una ejecución típica de este caso.



**Figura 4.5 - Diagrama de Ejecución en el Servidor Público**

La serie de pasos involucrados en la ejecución es:

1. El usuario teclea en su navegador la dirección proporcionada.
2. Esto genera un pedido HTTP, que accede a Internet hasta llegar a los servidores de Github.

3. El servidor retorna la página estática `index.html` (idéntica a la encontrada en el directorio web). El nombre es parte del estándar de la Web, en el cual cuando se accede a la dirección raíz `/`, se accede realmente a la página `index.html`.
4. La página llega hasta el navegador del usuario, donde se interpreta y visualiza. A partir de aquí el usuario puede interactuar con la misma.
5. El usuario accede a la página de ejecución (que se verá más adelante), configura el algoritmo y realiza un pedido de ejecución, que va hacia Internet.
6. Las interacciones desde el navegador apuntan ahora al servidor del alumno (y no a Github), donde se ejecuta el algoritmo genético con el programa desarrollado en este trabajo.
7. El programa genera una respuesta en un formato de intercambio llamado JSON<sup>9</sup> (otra alternativa podría ser XML o también HTML).
8. Al llegar la respuesta al navegador, este interpreta las instrucciones contenidas en la página y, con los datos ya disponibles, visualiza la salida del algoritmo.

Se aprecia la separación que existe entre el ordenador local, que sólo ejecuta un navegador, y los hosts u ordenadores remotos, los cuales realizan la ejecución del algoritmo propiamente dicho. Nótese que en el diagrama se muestra un solo usuario, aunque nada impide que sean varios, utilizando el mismo servidor concurrentemente.

Es necesario aclarar cuál es la funcionalidad del servicio de Github, ya que en principio podría parecer prescindible. Se sabe que el propio programa puede servir archivos estáticos (entre ellos los archivos `.html` necesarios para las distintas pantallas). Si se accede a la dirección del servidor en el que se está ejecutando el programa, efectivamente se podrá acceder a los archivos estáticos. El problema radica justamente en conocer la dirección del servidor. Esta suele ser una dirección IP cambiante o un nombre *alias* proporcionado por el proveedor del servicio. En ninguno de los casos se trata de una dirección amigable, fácilmente recordable por el usuario y estática.

La plataforma Github proporciona, tras haber creado un repositorio para almacenar el código fuente, una sección para albergar páginas `.html` estáticas. También proporciona una dirección URL amigable y fija para acceder a las mismas. La estructura de dichas direcciones es:

---

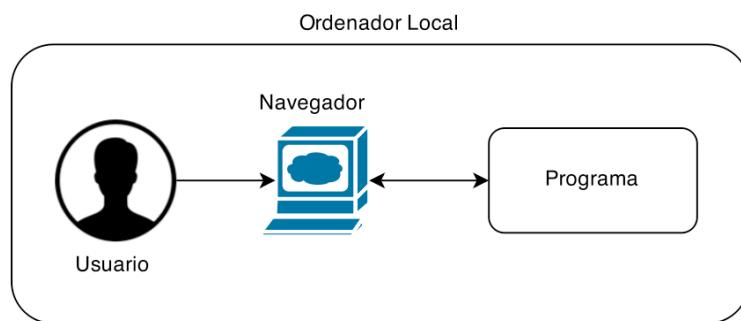
<sup>9</sup> JSON es un formato de intercambio sencillo y ligero, basado en JavaScript y utilizado ampliamente en Internet. Web oficial en <http://json.org/>

```
http://{nombre_de_usuario}.github.io/{nombre_del_proyecto}/
```

Además de la dirección amigable y la plataforma robusta y de alta disponibilidad, permite de alguna forma autocontener el proyecto con el código y las páginas Web en un mismo lugar. Por lo demás, podría prescindirse de ella y accederse al servidor directamente, con una funcionalidad idéntica. Este caso se contempla más adelante, en el apartado “Ejecución en un Host de red Remoto”.

## Ejecución en el Host Local

Otra topología consiste en ejecutar el programa y la visualización en el host local (es decir el ordenador del usuario). Para ello, además de necesitar de un navegador moderno como se indicó con anterioridad, es necesario tener instalada la máquina virtual de Java y, por supuesto, el propio programa. En este caso no es necesario disponer una conexión de red ni Internet. Se muestra un esquema de la ejecución local en la Figura 4.6.



**Figura 4.6 - Diagrama de Ejecución Local**

Aunque la cantidad de interacciones es idéntica que para el caso anterior, esta vez intervienen menos ordenadores, por lo que se simplifica el esquema. De todas formas, la cantidad de pasos es similar:

1. Se ejecuta la aplicación en modo servidor, por ejemplo con el puerto 8080 (que no debe estar en uso por ninguna otra aplicación).
2. Se ingresa en el navegador a la dirección del propio ordenador o host local con el puerto elegido anteriormente. Por lo general, se llama a esta dirección *localhost* y se le reserva la dirección IP 127.0.0.1. Suponiendo que se utilizó el puerto 8080, la dirección completa sería <http://127.0.0.1:8080> o <http://localhost:8080>.
3. El pedido no necesita acceder a Internet, por lo que se resuelve localmente.
4. El programa servidor retorna la página estática index.html.

5. La página llega hasta el navegador del usuario, donde se interpreta y visualiza. A partir de aquí el usuario puede interactuar con la misma.
6. El usuario accede a la página de ejecución (que se verá más adelante), configura el algoritmo y realiza un pedido de ejecución.
7. Las interacciones desde el navegador apuntan esta vez al ordenador local, por lo que se resuelven nuevamente sin utilizar Internet.

El resto de la ejecución es idéntico al caso del host remoto.

### Ejecución en un Host de red Remoto

Si se está ejecutando el programa en el ordenador local y se dispone de conexión en red, puede compartirse el acceso a la aplicación, siempre y cuando la otra persona se encuentre en la misma red. Para ello es necesario conocer la dirección IP pública del ordenador local (que esta vez **no** será ni *localhost* ni 127.0.0.1), y que no exista ningún cortafuegos que impida el tráfico por el puerto seleccionado. Se muestra en la Figura 4.7 cómo un ordenador remoto puede conectarse a través de la red al ordenador donde se está ejecutando la aplicación. De esta manera los usuarios, por ejemplo varios alumnos y profesores dentro de una misma universidad, pueden utilizar simultáneamente el mismo programa.

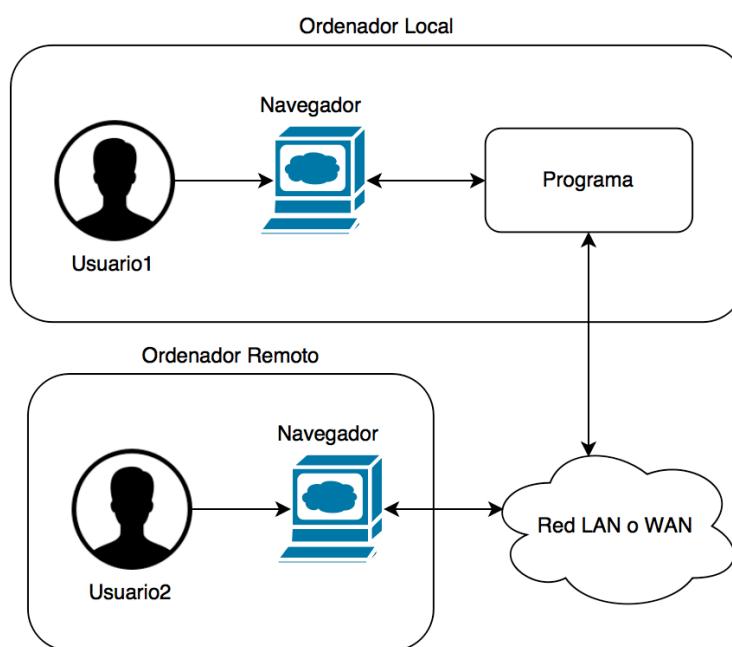


Figura 4.7 - Diagrama de Ejecución Remota

## Pantallas y Páginas Web

Se muestran a continuación las distintas pantallas y páginas principales del programa. En los ejemplos se utiliza la modalidad de ejecución en el host local, aunque todas las variantes producen el mismo resultado, variando únicamente la dirección web ingresada en el navegador. Las capturas de pantalla corresponden al navegador Web Chrome en un ordenador Mac. El diseño de la página podría variar levemente dependiendo del sistema operativo y la resolución de pantalla, ya que se utiliza un diseño adaptativo para poder visualizar las páginas en dispositivos de baja resolución como celulares y tabletas.

El servidor ofrece una serie de pantallas, cada una destinada a una funcionalidad independiente, aunque todas ellas en conjunto pueden considerarse como una única página Web. La idea principal es dotar a esta página de un carácter autocontenido, que no sólo ejecute el algoritmo, sino que presente el proyecto en su totalidad.

La Figura 4.8 muestra la pantalla principal o de bienvenida. La misma describe brevemente el proyecto y las demás pantallas (llamadas *secciones*), todas accesibles desde la barra. El código fuente de la misma se encuentra en el archivo `index.html`, dentro del directorio `web`.



**Figura 4.8 - Pantalla de Inicio**

La siguiente de las secciones se titula Documentos y contiene la última versión del Anteproyecto y la Memoria (Figura 4.9), ambas sin los datos personales. Nuevamente, esto ayuda a autocontener el proyecto, al incluir la posibilidad de visualizar la documentación en línea. Nótese que aunque el servidor y la ejecución funcionan de manera aislada sin necesitar conexión a Internet, para acceder a estos recursos sí que la necesitan, ya que el archivo *pdf* se descarga siempre desde un servidor remoto (concretamente desde Dropbox). El código fuente de estas pantallas se encuentra en los archivos *memoria.html* y *anteproyecto.html*.

La tercera pantalla aparece en la Figura 4.10, titulada Instrucciones. Allí se indican los pasos básicos para descargar, compilar y ejecutar el programa, de manera similar a como se explicó en las secciones 4.3 y 4.7, aunque de manera un poco más breve.



**Figura 4.9 - Pantalla de Documentos**



**Figura 4.10 - Pantalla de Instrucciones**

Como se comentó anteriormente, existe también una pantalla para iniciar la ejecución del algoritmo genético, que aparece en la Figura 4.11. La misma permite la elección de los parámetros de manera sencilla, así como una pequeña ayuda para cada uno de ellos. Al final de la lista de parámetros se encuentra el botón “Ejecutar”, que lanza un pedido HTTP al servidor configurado. Cuando este lo reciba, devolverá al navegador la información necesaria para visualizar los resultados, que aparecerán más abajo en la misma pantalla (véase la Figura 3.1, que muestra una pequeña sección de los resultados). Téngase en cuenta que, dependiendo de los parámetros concretos, el resultado del algoritmo puede tardar un tiempo en calcularse, por lo que se debe esperar un tiempo prudente. Si se recarga la página o se presiona varias veces el botón antes de que haya terminado la ejecución, el servidor creará varios hilos de ejecución simultáneos, lo que podría sobrecargar el servidor.

PFG Algoritmos Genéticos

Ejecutar

# Ejecución

## Elección de Parámetros

Tamaño de la Población	1000	?
Generaciones	100	?
Función	Rastrigin	?
Dimensión	3	?

## Elección de Operadores

Figura 4.11 - Pantalla de Ejecución

PFG Algoritmos Genéticos

Ejecutar

# Resultado de la Ejecución

Tiempo de Ejecucion: 112ms

Fitness del mejor individuo obtenido: -0.007

Coordenadas del mínimo:

$x_1: -0.001$	$x_2: 0.000$	$x_3: -0.000$
$x_4: 0.003$	$x_5: 0.002$	$x_6: -0.003$
$x_7: -0.001$	$x_8: -0.002$	$x_9: 0.002$
$x_{10}: 0.002$		

Coordenadas del centroide:

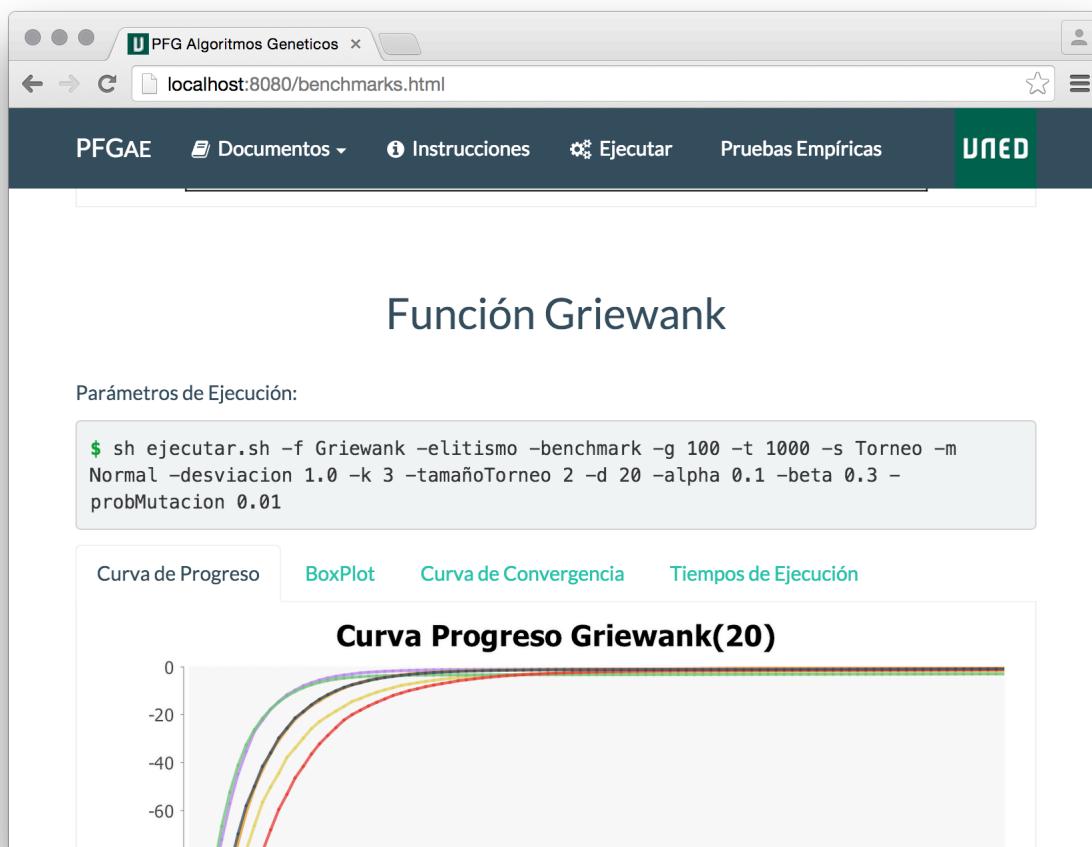
$x_1: -0.001$	$x_2: 0.001$	$x_3: -0.001$
$x_4: 0.000$	$x_5: 0.001$	$x_6: 0.001$
$x_7: -0.001$	$x_8: 0.000$	$x_9: -0.008$
$x_{10}: 0.002$		

Desviación final de la población: 0.345

Figura 4.12 - Pantalla de Ejecución, Resultados

La última pantalla de la Figura 4.13 contiene las pruebas empíricas realizadas. Es, en cierta forma, una versión resumida del Capítulo 5. Por cuestiones de espacio, se muestra una pequeña sección de la página. La pantalla muestra, para cada función evaluada, la serie de parámetros utilizados para la prueba y una serie de gráficas sobre el rendimiento. Para ver las distintas gráficas basta con pinchar sobre la pestaña deseada (Curva de Progreso, BoxPlot, Curva de Convergencia o Tiempos de Ejecución).

Cabe destacar que la única pantalla interactiva y que en consecuencia requiere interacción con el servidor es la de Ejecución. Las demás son archivos estáticos que pueden ser suministrados sin necesidad de un programa servidor<sup>10</sup>, aunque se realiza de esta forma para centralizar el acceso y permitir así una posible expansión de la funcionalidad de la página web adaptando el código del servidor.



**Figura 4.13 - Pantalla de Pruebas Empíricas**

<sup>10</sup> Recuérdese que los navegadores son capaces de abrir archivos .html directamente desde el disco duro, sin necesidad de que este sea servido por un servidor Web.



# Capítulo 5: Evaluación Experimental

En el presente trabajo se estudia la etapa de Recombinación desde un punto de vista computacional y algorítmico. Para lo primero se estudia el tiempo de ejecución del algoritmo. Para lo segundo se utilizan mediciones de la velocidad de convergencia y progreso de la población.

El estudio es esencialmente comparativo, ya que los números por sí mismos no resultan de gran interés, aunque sí la comparación de los resultados para las distintas implementaciones.

Se da en este capítulo una breve introducción sobre el problema evaluado. Luego se comenta sobre el tipo de gráfico utilizado para la visualización de los datos. Finalmente, se exponen las evaluaciones de las distintas funciones elegidas para las pruebas empíricas, concretamente: Griewank, Rastrigin y Schwefel.

## 5.1 Introducción

Al tratarse de un algoritmo estocástico con un gran factor aleatorio, difícilmente la salida del programa sea la misma en repetidas ejecuciones, aun cuando no se alteren los parámetros. La calidad de la solución encontrada por el algoritmo variará entonces entre las distintas ejecuciones. Dependiendo del tipo de problema al que se aplique el AG, esto puede tener distinta importancia. Existen problemas que requieren que el algoritmo encuentre una buena solución siempre y otros cuyos requisitos son un poco más relajados. Bajo este criterio pueden distinguirse los siguientes tipos de problemas a los que puede aplicarse un AE:

- Problemas de Diseño u *offline*.
- Problemas Repetitivos.
- Problemas *online* o de Tiempo Real.

Para los problemas de diseño lo importante es encontrar una única solución muy buena, sin requisitos especiales sobre rendimiento y, especialmente, tiempos de ejecución. En ellos puede que el algoritmo se ejecute repetidas veces (por ejemplo si una solución encontrada no fue buena), o se deje correr por varias semanas. Suelen utilizarse en proyectos de gran escala, que pueden durar varios años, por lo que es importante encontrar

la mejor solución posible. Por ejemplo, en proyectos de construcción de puentes, optimización de circuitos, etc.

Los problemas repetitivos suelen tener requisitos temporales un poco más estrictos, en el orden de minutos. Se utilizan por ejemplo para calcular la ruta más eficiente para un repartidor, planificación diaria de horarios y tareas para el personal (*scheduling*), etc. Este cálculo puede realizarse algunas veces por día o incluso una única vez. Es posible que no pueda ejecutarse el algoritmo una gran cantidad de veces, ni por muchas generaciones. Interesa entonces un rendimiento y calidad buenas (mejor que aquellas realizadas por un humano) y consistentes. Es decir, que será más importante la calidad promedio de las soluciones que la calidad pico o máxima posible. A su vez, al ejecutarse el algoritmo repetidas veces, se producirá una variación con respecto al promedio. Interesa también que esta variación sea la menor posible para evitar que se obtengan eventualmente soluciones muy malas.

Los problemas de tiempo real son conceptualmente parecidos a los repetibles, aunque con requisitos temporales mucho más estrictos. Suelen utilizarse además en sistemas críticos, por lo que también requieren una variación mínima y que el promedio de las soluciones sea lo mejor posible.

En el presente trabajo se estudia la Recombinación desde el punto de vista de los problemas repetibles, por presentar un compromiso razonable e intermedio entre calidad de la solución y consistencia. Será necesario entonces ejecutar el algoritmo repetidas veces para recoger métricas que ayuden a concluir sobre la consistencia y calidad de las soluciones.

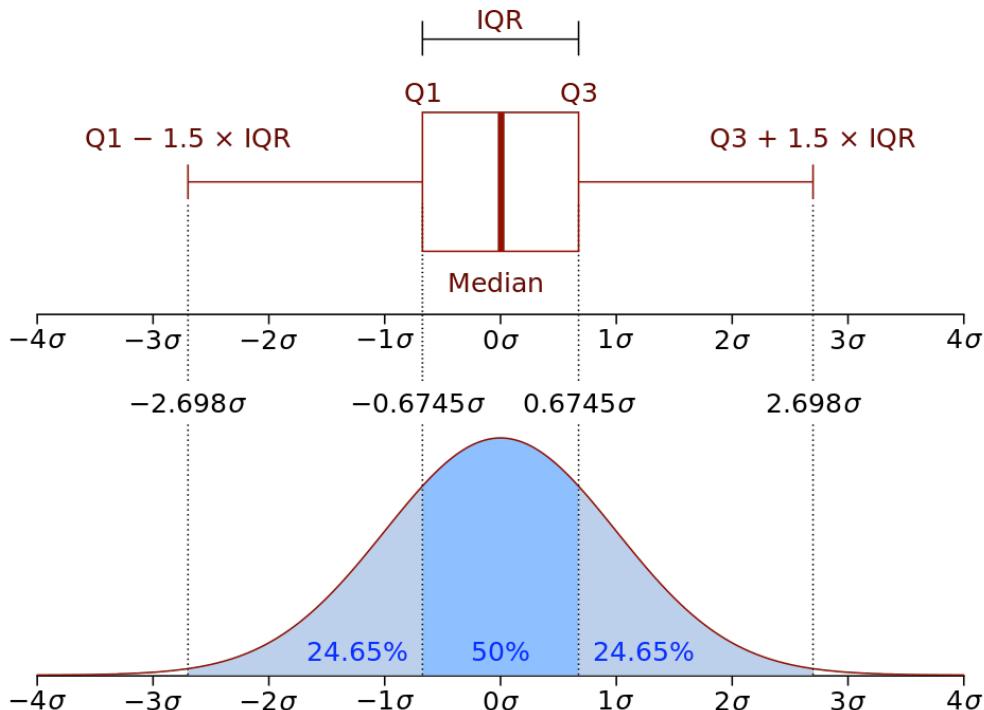
## 5.2 El gráfico Box Plot

Para sacar conclusiones sobre la calidad de las soluciones se ha acudido a medidas estándar de la estadística: mínimo, máximo, promedio, desviación típica, etc. Para ello es necesario un tipo de gráfico que incluya todas esas medidas de manera compacta y permita sacar conclusiones rápidas con un simple vistazo. Se ha elegido para este propósito el gráfico Box Plot, ejemplificado en la Figura 5.1 (imagen cortesía de Wikipedia). En ella se muestra una distribución de valores normal y, por encima, el Box Plot que resulta de dicha distribución

El gráfico condensa varias medidas en un solo objeto. El mismo está compuesto por un rectángulo (una “caja” o *box* en inglés) y unas líneas que se salen de él, que terminan en una

línea perpendicular llamada *whisker*. El rectángulo se extiende desde el primer cuartil (Q1 en la figura) hasta el tercero (Q3). El rango inter cuartil (*IQR, Inter Quartile Range* en inglés) entre Q1 y Q3 contiene por definición el 50% de los valores muestreados.

Se le agrega una línea dentro de la caja, que representa la mediana (el valor de la muestra que separa los valores a la mitad, 50% por encima y 50% por debajo). Esta no tiene por qué coincidir con el promedio (que se representará más adelante con un rombo), el cual no es generalmente un valor realmente observado en la muestra.



**Figura 5.1 - Ejemplo de Box Plot**

Los *whiskers* representan los valores mínimos y máximos obtenidos, pero sin tener en cuenta los valores extremos o atípicos (*outliers* en inglés). Existen distintas alternativas para decidir qué valores son atípicos, aunque por lo general se consideran atípicos los valores alejados más allá de 1.5 veces el rango inter cuartil. Es decir, aquellos menores a  $Q1 - 1.5 \times IQR$  o mayores a  $Q3 + 1.5 \times IQR$ . Las líneas muestran entonces el rango de valores observado y la caja donde se concentran la mayoría de los valores. Nótese que los valores extremos no se muestran en este gráfico. Esto podría ser un problema si el algoritmo retorna, ocasionalmente, una solución extremadamente mala. Se debe recordar, sin embargo, que la evaluación se realiza desde el punto de vista de los problemas repetibles y no de aquellos de tiempo real. Para estos últimos, las exigencias son mayores y deberían utilizarse otras visualizaciones adecuadas.

### 5.3 Metodología

A lo largo de este capítulo se evaluará el AG con respecto a la calidad de las soluciones obtenidas y su tiempo de ejecución, haciendo especial hincapié en el operador de Recombinación. Para ello se elegirán tres funciones a optimizar en orden creciente de complejidad, ya sea por la dimensionalidad utilizada o por la dificultad de la propia función. A menos que se indique lo contrario, los demás parámetros que no dependen de la Recombinación elegida se mantendrán constantes.

Para poder probar la consistencia del algoritmo se utiliza el modo `Benchmark` descrito en la sección 4.7.2, que ejecuta cien veces el algoritmo por cada función y recombinación disponibles. Como el algoritmo prueba todas las funciones con los mismos parámetros, y la evaluación debe realizarse de a una función a la vez, los resultados de las demás funciones simplemente se descartan.

La ejecución devuelve una tabla con las estadísticas sobre la fitness de las soluciones obtenidas y una serie de gráficos, a saber:

- BoxPlot: condensa los valores mínimo, máximo, mediana y promedio que se han observado en la fitness en las distintas ejecuciones. Realiza un BoxPlot por cada recombinación evaluada, en columnas diferentes.
- Curva de Progreso: grafica una curva por cada recombinación con el valor de la fitness del mejor individuo de la población a lo largo de las 100 generaciones (no confundir con las 100 veces que se repite la ejecución). Para suavizar las curvas y evitar anomalías en los gráficos, cada uno de los puntos es el promedio de las 100 ejecuciones, por lo que este gráfico no muestra ni las peores ni las mejores ejecuciones.
- Curva de Convergencia: similar en su implementación al de curvas de progreso, también promedia la convergencia entre las 100 ejecuciones. Está inspirado en [10] aunque, luego de algunas modificaciones en el código, dibuja la desviación típica de la población con respecto a su centroide.
- Tiempos de Ejecución: gráfico de barras con el tiempo de ejecución de las 100 generaciones del algoritmo (es decir una ejecución completa), promediado entre las 100 ejecuciones.

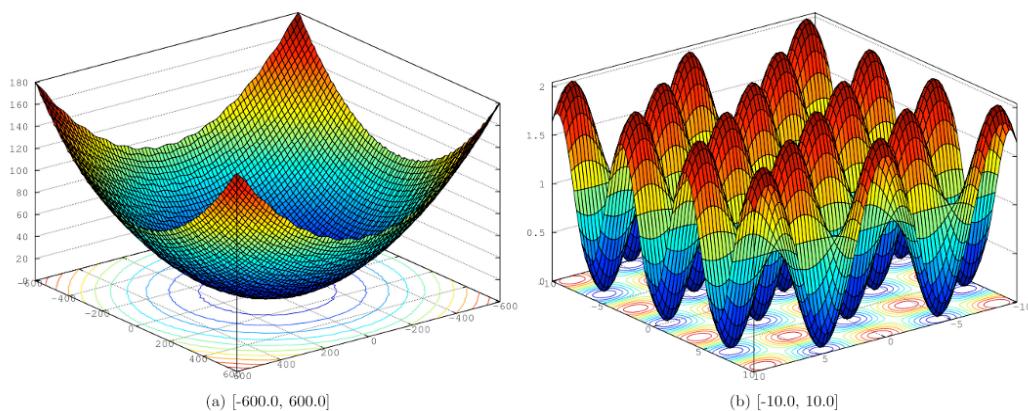
## 5.4 Evaluación de la Función Griewank

La primera función en ser evaluada, por ser la más sencilla de las tres, es la función Griewank. La misma se define como:

$$griewank(\bar{x}) = 1 + \sum_{i=1}^n \frac{x_i^2}{4000} - \prod_{i=1}^n \cos\left(\frac{x_i}{\sqrt{i}}\right)$$

Se aprecia que la función es multidimensional, en cuanto acepta vectores de cualquier dimensión, y produce un único valor de salida (es decir es de la forma  $\mathbb{R}^n \rightarrow \mathbb{R}$ ). En la Figura 5.2 se muestra una gráfica de la función para  $n = 2$ , es decir tres dimensiones (dos de entrada y una de salida, con el valor de la *fitness*). A la izquierda (a) se ve cómo la función parece ser unimodal (monótona decreciente) cuando el dominio es grande. Sin embargo, si se ve de más cerca (b), se aprecia cómo la función dispone de una gran cantidad de mínimos locales dispersos por toda su superficie.

Se ha utilizado como dominio de la función valores en el intervalo  $[-600, 600]$ , que suele ser lo habitual en el ámbito académico para esta función. El mínimo vale exactamente cero, en el vector  $(0, 0, \dots, 0)$ . Además, se ha optado por probar un caso sencillo, eligiendo  $n = 2$ , es decir dos dimensiones de entrada (y 1 de salida).



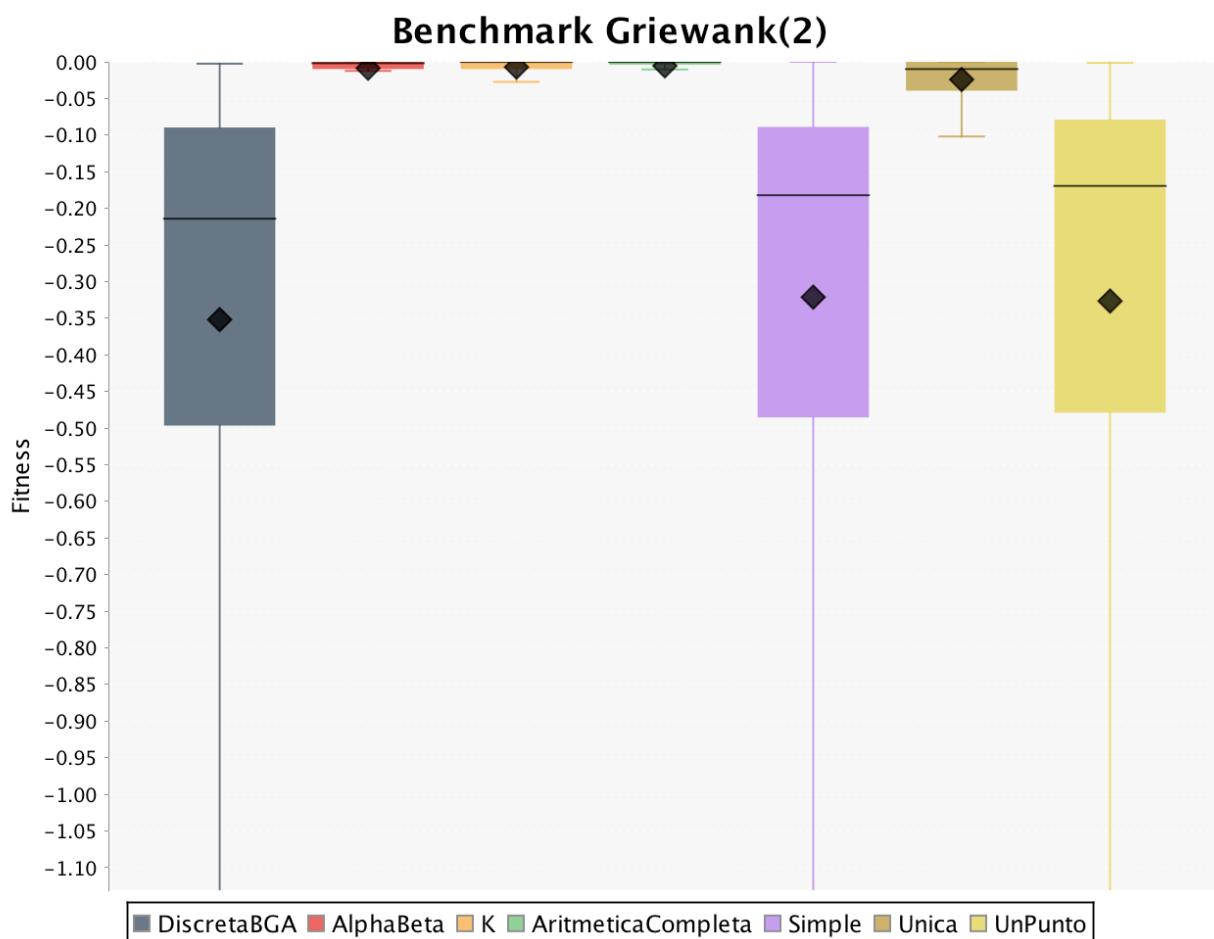
**Figura 5.2 - Función Griewank**

Se ejecuta el algoritmo con los siguientes parámetros:

```
-benchmark -g 100 -t 50 -s Torneo -m Normal -desviacion 1 -k 2 -tamañoTorneo 2
-d 2 -alpha 0.1 -beta 0.3 -elitismo
```

Si bien la función parece en principio compleja, la baja dimensionalidad (-d 2, es decir dos dimensiones) hace que el algoritmo encuentre el mínimo sin problemas, incluso para una población pequeña (-t 50, es decir cincuenta individuos).

En la Figura 5.3 se aprecia la calidad de las soluciones obtenidas por las distintas recombinaciones. Cada columna es una recombinación distinta y, de no poder apreciarse correctamente los colores, recuérdese que la leyenda está en orden con las columnas. Es decir, que el box plot de la primera columna corresponde a la primera recombinación en la leyenda, y así sucesivamente.



**Figura 5.3 - Box Plot de la función Griewank**

Vale la pena recordar que, como se explicó en la sección 4.1, la máxima fitness posible valdrá cero y las soluciones serán tanto peor cuanto menor sea su valor.

Todas las recombinaciones han alcanzado en alguna ejecución el mínimo en cero, aunque algunas de ellas presentan una dispersión elevada. Téngase en cuenta la escala del gráfico, ya que las peores ejecuciones han obtenido un valor cercano a uno. En todo el dominio de la función, los valores máximos de la función son cercanos a 180 (por lo que la

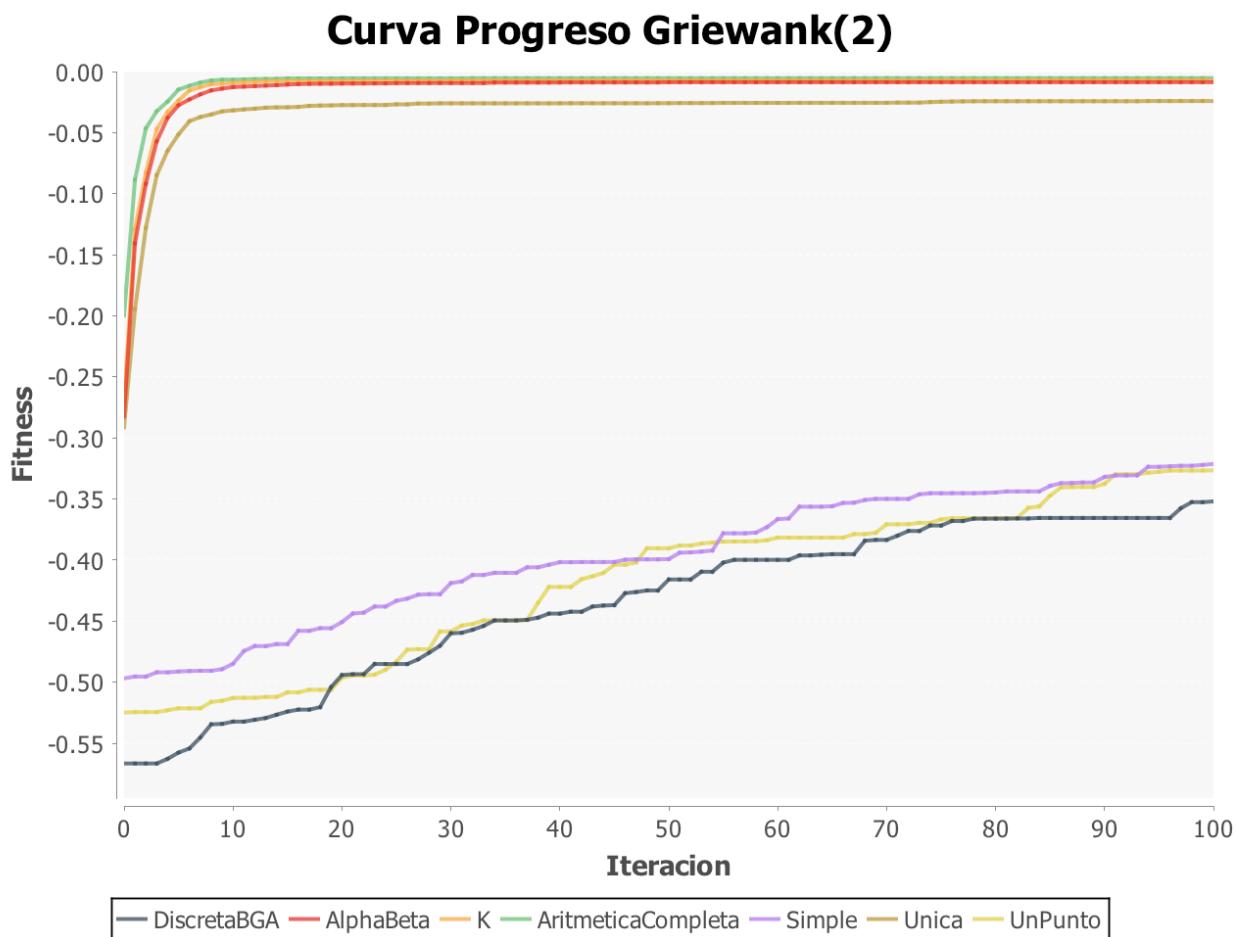
peor fitness valdría -180). Dentro del intervalo de valores [-5, 5] los peores valores de la función de adecuación son entre -1 y -2, por lo que, si bien algunas recombinaciones parecieran tener mucha dispersión, esta es pequeña si se la compara con el rango de la función.

Aunque a primera vista pareciera que las recombinaciones discretas (*DiscretaBGA* y *UnPunto*) no han tenido buenos resultados, puede verse en la Figura 5.4 la raíz del problema: la cantidad de generaciones no ha sido suficiente. Al no introducir material genético nuevo (ya que sólo intercambian genes entre los padres), la exploración es un poco más lenta, especialmente considerando que la función tiene sólo dos dimensiones de entrada. La Figura 5.5 soporta esta teoría, ya que se evidencia que la convergencia es lenta. Esto quiere decir que estas recombinaciones no han convergido completamente, por lo que de aumentarse la cantidad de generaciones, es posible que encuentren mejores soluciones.

La recombinación simple, a pesar de sí generar nuevo material genético, lo hace a partir de un punto de cruce  $i$ , mientras que el resto de los genes se conservan de los padres. Al tener los hijos sólo dos genes, los valores posibles de  $i$  serán:

- 0: se conserva el primer gen de cada parente (índice cero) y el otro se promedia (índice uno).
- 1: se conservan ambos genes de cada parente.

Entonces, el gen con índice cero nunca se promedia, por lo que se conserva en las generaciones. Existe además un 50% de probabilidades de que la recombinación no realice nada (si  $i = 1$ ), por lo que su curva de progreso será más lenta. Nótese que esto siempre puede pasar con esta recombinación, aunque la probabilidad será más grande cuanto menos dimensiones tenga la función. Resulta evidente que podría mejorarse esta recombinación evitando que los primeros  $i$  genes se copiaran **siempre** de los padres. Se podría por ejemplo, bajo cierta probabilidad, invertir la lógica y promediar los primeros  $i$  genes y copiar los últimos (de manera inversa a como funciona actualmente). Esto fomentaría una mayor exploración en las primeras dimensiones de la función.

**Figura 5.4 - Curva de Progreso función Griewank**

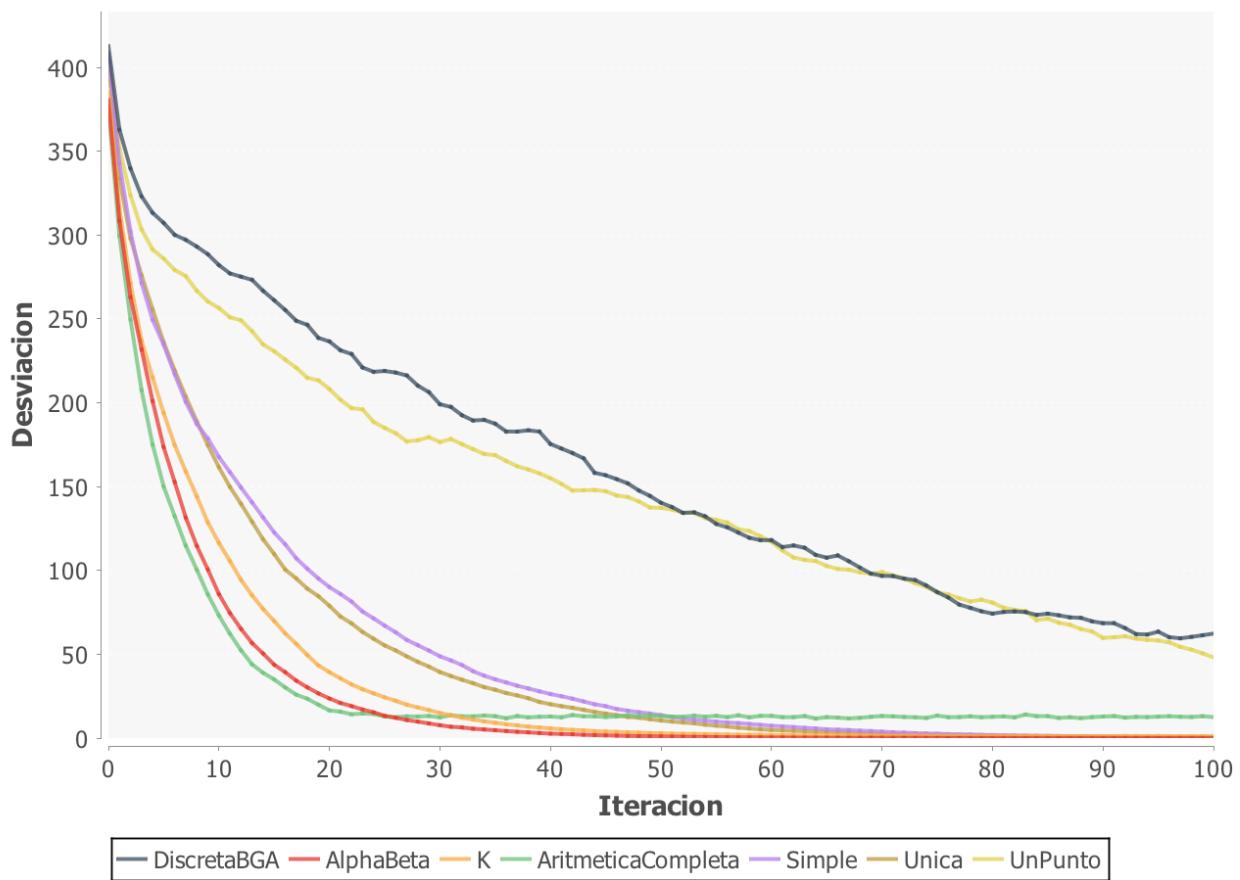
El resto de las implementaciones tienen curvas muy similares, al punto que algunas se solapan completamente entre sí. Dado que se utiliza el argumento  $k = 2$ , y la función tiene sólo dos dimensiones, cabe preguntarse por qué la recombinación K no se comporta de igual manera a la aritmética completa, ya que en principio ambas realizan el promedio aritmético de los dos ejes. El problema radica en la implementación de la recombinación K que, por simplicidad, elige  $k$  ejes para promediar **con reemplazamiento**. Esto quiere decir que pueden obtenerse los siguientes valores de  $k$  (que nunca puede valer más que la dimensión de la función):

- 0 y 0: se recomienda sólo el primer gen.
- 0 y 1 ó 1 y 0: se recombinan ambos ejes. Cualquiera de las dos opciones es equivalente, cambiando solamente el orden en el que se generan los hijos.
- 1 y 1: se recomienda sólo el segundo gen.

Se observa que existe un 50% de probabilidades de combinar un único gen, en cuyo caso se comportará de manera idéntica a la recombinación única.

Se deja como posible mejora futura la elección de ejes sin reemplazamiento, la cual podría mejorar el rendimiento del operador para funciones de baja dimensionalidad.

## Convergencia Griewank(2)

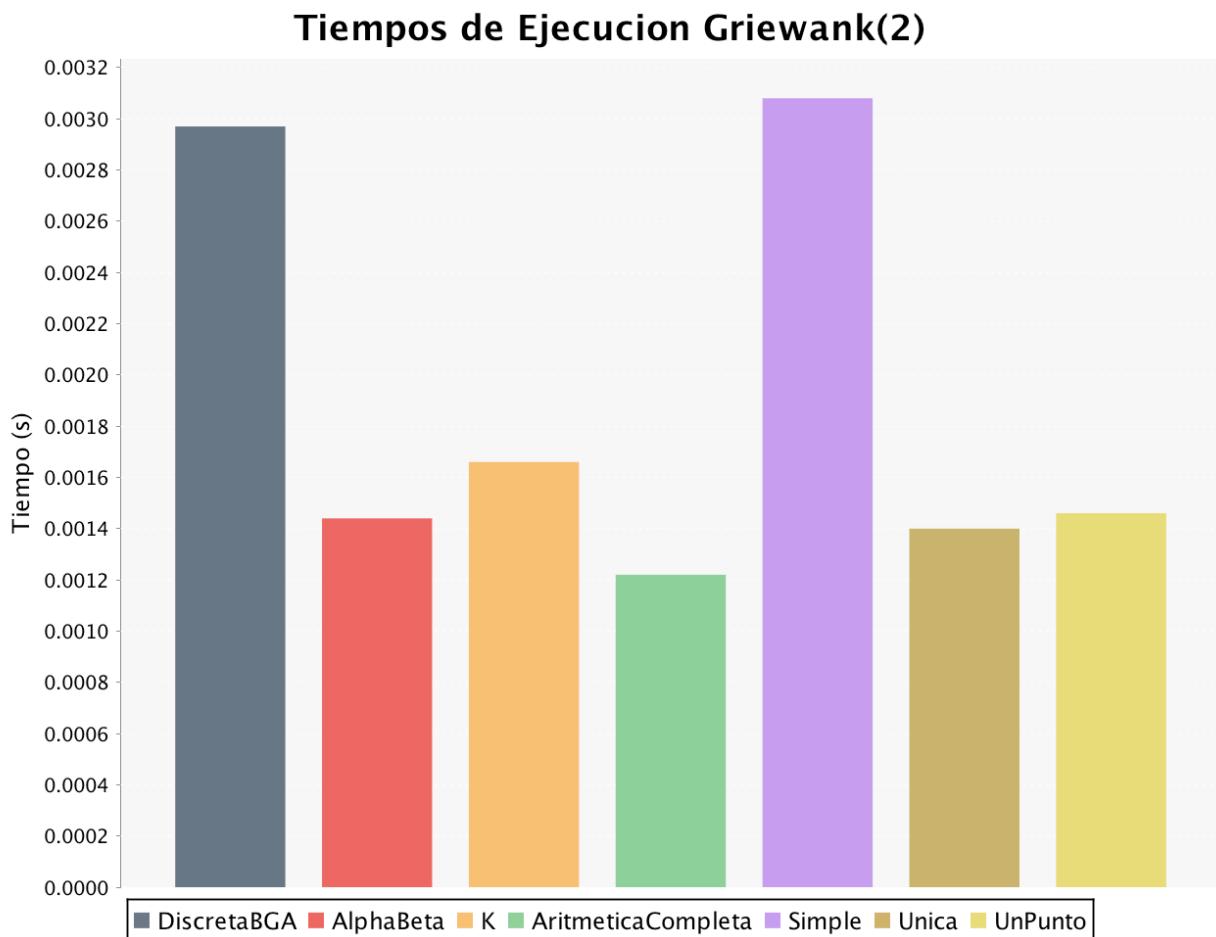


**Figura 5.5 – Convergencia, función Griewank**

La convergencia de todas las recombinaciones aritméticas (Figura 5.5) es relativamente rápida, exceptuando las discretas. Esto puede ser una propiedad deseable si se desea, por ejemplo, un algoritmo rápido que terminara una vez que la población convergiese, en vez de luego de una cantidad fija de iteraciones. Sin embargo, una convergencia muy rápida suele ser una propiedad indeseable de los AEs, dado que puede causar que el algoritmo se “atasque” en un mínimo local no muy bueno.

En cuanto a los tiempos de ejecución, puede consultarse la Figura 5.6. Aunque existen algunas recombinaciones más lentas, todas ellas son relativamente rápidas: 1.2 milisegundos la más rápida y 3.2 milisegundos la más lenta; lo suficiente para un problema de tiempo real. Se recuerda que este tiempo es la cantidad de segundos totales en realizar las 100 generaciones, es decir la ejecución completa del algoritmo, promediado entre las cien ejecuciones. Al ser números tan pequeños, es difícil medir los tiempos correctamente,

ya que cualquier actividad externa en el sistema operativo (por ejemplo alguna interrupción o cambio de contexto) podría tener una gran incidencia en este tiempo, por lo que es difícil realizar una conclusión definitiva.



**Figura 5.6 - Tiempos de ejecución, función Griewank**

La función de 2 dimensiones es un caso sencillo, que no pone demasiado a prueba al algoritmo. Además, el hecho de tener sólo 2 genes por individuo produce efectos indeseados en algunas recombinaciones. Son problemas que, además, podrían ser resueltos sencillamente de modo analítico, por lo que carecen de mucha utilidad práctica.

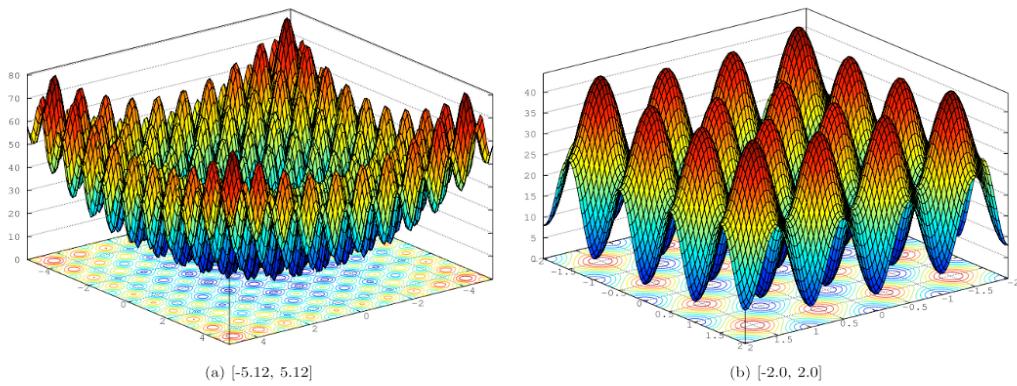
En las secciones siguientes se evaluarán funciones más complejas, tanto por su forma o superficie como por su dimensionalidad.

## 5.5 Evaluación de la Función Rastrigin

La función Rastrigin (Figura 5.7) es también multidimensional y con varios mínimos locales, al estar compuesta por una función periódica:

$$rastrigin(x) = An + \sum_{i=1}^n [x_i^2 - A \cos(2\pi x_i)]$$

donde  $A$  es normalmente 10 y  $n$  es la cantidad de dimensiones. Se ha elegido como dominio el intervalo [-5.12, 5.12], utilizado comúnmente para esta función.



**Figura 5.7 - Función Rastrigin**

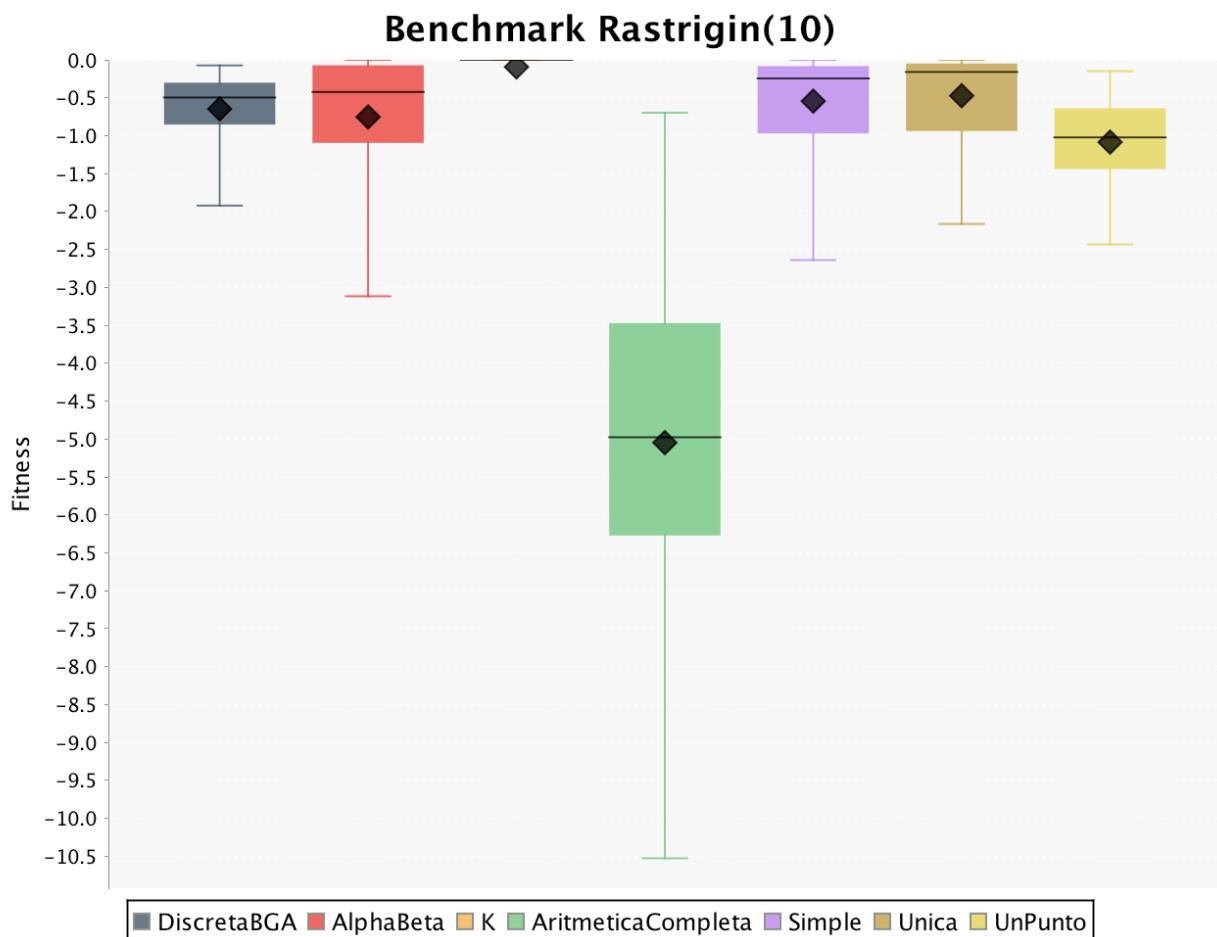
A la izquierda de la Figura 5.7 (a) se aprecia el dominio completo de la función, entre [-5.12, 5.12]. A la derecha se ve una versión ampliada para ver la superficie cerca del mínimo. El mismo vale cero y se encuentra en la coordenada (0, 0, ... , 0), independientemente de cuantas dimensiones se elijan.

Para aumentar la dificultad del problema se utiliza una dimensionalidad mayor a la utilizada anteriormente. La misma se eligió arbitrariamente en diez, lo cual aumenta considerablemente la complejidad de resolver el problema de manera analítica, justificando aún más el uso de un AG para la minimización.

Al tratarse de un problema más complejo, es necesario aumentar el tamaño de la población. Si se utiliza un tamaño muy pequeño, el algoritmo no explorará lo suficiente como para encontrar el mínimo global. Si se utiliza un tamaño muy grande, se encontrará en la mayoría de los casos una buena solución, independientemente de la recombinación elegida. Como el objetivo de la evaluación es comparar las distintas recombinaciones, es esencial que sean ellas las que influyan en la resolución del problema y no la mera probabilidad (a mayor población, mayor probabilidad de encontrar el óptimo por "casualidad"). Luego de distintas pruebas se ha tomado un tamaño de 750 individuos para la población, quedando los parámetros del algoritmo de la siguiente manera:

```
-elitismo -benchmark -g 100 -t 500 -s Torneo -m Normal -desviacion 1 -k 3
-tamañoTorneo 2 -d 10 -alpha 0.1 -beta 0.3
```

Puede verse el resultado de la ejecución en la Figura 5.8. Se destaca la calidad y consistencia de las soluciones de la recombinación K. Aunque no se ha modelado así en el código, la recombinación aritmética completa es también una implementación particular de la recombinación K, donde  $k = n$ . Cabe preguntarse entonces hasta qué punto es conveniente aumentar el valor de  $k$ , ya que de ser  $k = n$ , la calidad desciende considerablemente. La aritmética completa es, de hecho, la que peor se ha comportado para la función Rastrigin de 10 dimensiones. Por otra parte, la recombinación única ( $k = 1$ ) tampoco ha tenido tan buen comportamiento. Se estudiará más adelante en la sección 5.7 el efecto de aumentar el valor de  $k$  progresivamente.



**Figura 5.8 - Box Plot, función Rastrigin**

Para determinar el rendimiento de las recombinaciones, debe observarse el menor de los *whiskers* (las peores soluciones) y el más cercano a cero (las mejores). Es deseable que la distancia entre los mismos y el tamaño de la caja sean lo menor posible. Es por ello que la siguiente “mejor” recombinación es la DiscretaBGA, ya que su promedio y mínimo son los

mayores de todas las demás recombinaciones. Recuérdese que, aunque se trata de un problema de minimización, la fitness es tanto mejor cuanto más cercana a cero esté.

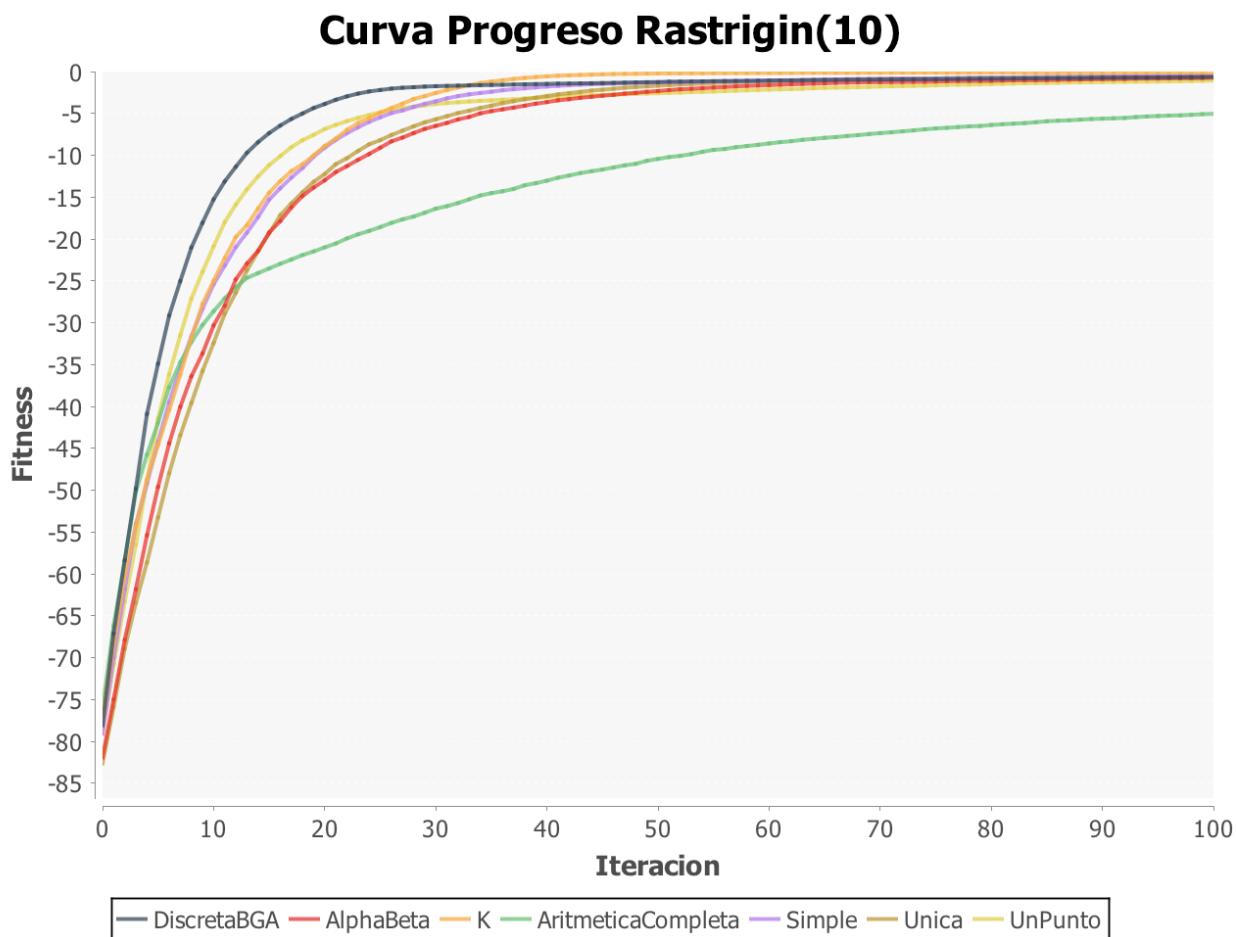
Aunque el Box Plot condensa toda la información, se incluye la Tabla 5.1 con los datos numéricos de las ejecuciones para una referencia más precisa.

Recombinación	Peor	Promedio	Mejor	Desviación
DiscretaBGA	-1.92	-0.65	-0.07	0.44
AlphaBeta (0.1, 0.3)	-3.12	-0.75	0	0.81
K (3, 0.1)	-1.01	-0.09	0	0.27
AritmeticaCompleta (0.1)	-10.53	-5.05	-0.7	2.14
Simple (0.1)	-2.64	-0.54	0	0.61
Única (0.1)	-3.26	-0.47	0	0.63
UnPunto	-3.27	-1.08	-0.15	0.58

**Tabla 5.1 - Resultados de la función Rastrigin**

Debe recordarse que el Box Plot no muestra los considerados valores extremos o atípicos. Es por ello que en la Tabla 5.1 aparece, para la recombinación K, un valor de -1.01 como la peor solución encontrada, mientras que el Box Plot se ve completamente compacto. Esto indica que alguna de las soluciones ha sido considerablemente mala, comparada con la distribución de los valores típicos. Esta solución (o soluciones) han ocurrido rara vez, lo cual se evidencia por la baja desviación, y el promedio alejado del valor mínimo o peor. Es posible que un valor atípico “arrastre” el promedio fuera del intervalo de valores más probables, y esto es exactamente lo que se aprecia para la recombinación K, donde el promedio cae fuera del intervalo inter cuartil (IQR) del Box Plot. Esto no puede pasar con la mediana (línea negra), que siempre es un valor perteneciente a la muestra y dentro del IQR, aunque no necesariamente a la mitad del mismo.

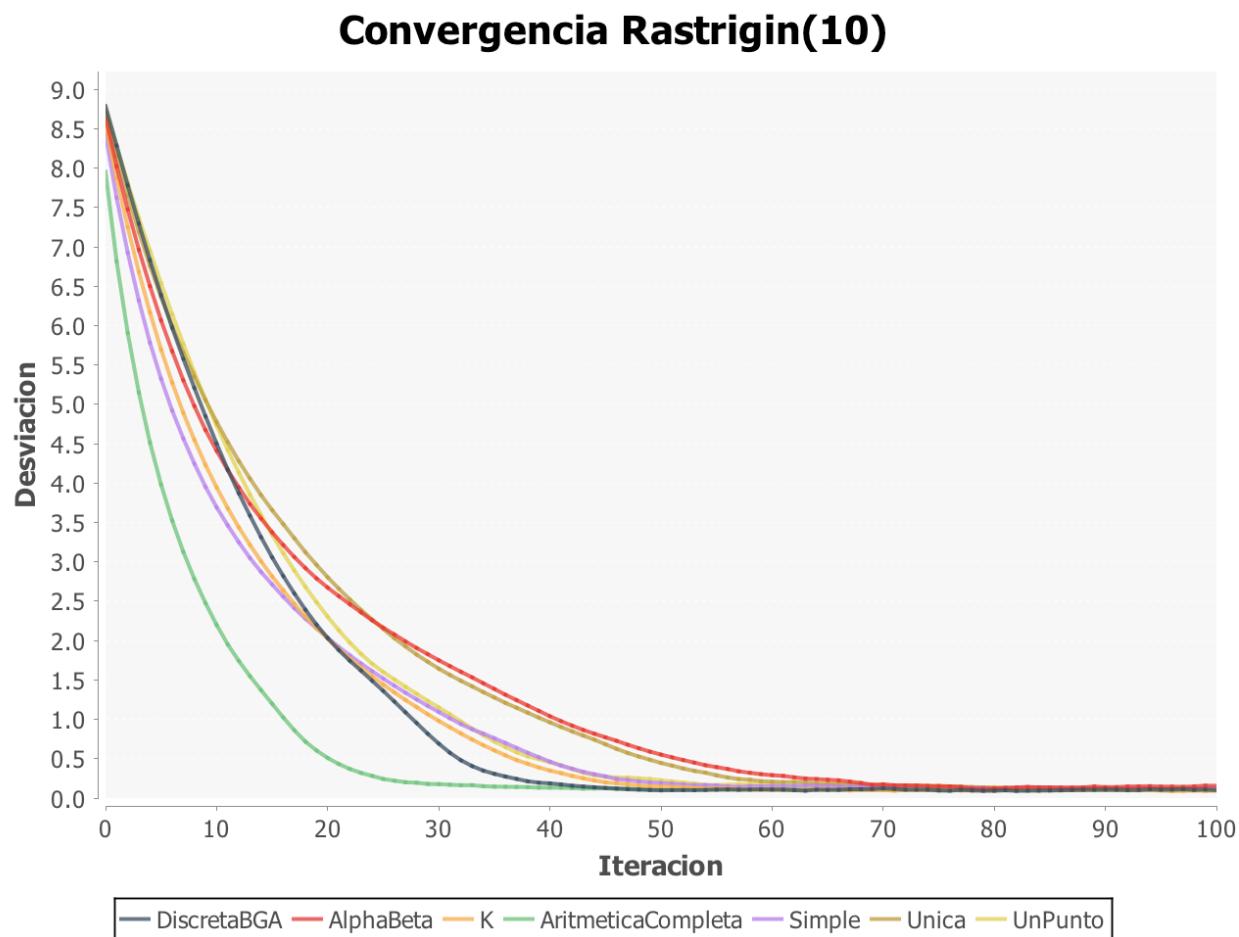
Con respecto a la curva de progreso (Figura 5.9), destaca el rápido comienzo de la recombinación aritmética completa. Sin embargo, cerca de la generación 10 esta velocidad disminuye, y todas las demás recombinaciones logran progresar más rápidamente y alcanzan mejores resultados. Un hecho similar ocurre con la recombinación DiscretaBGA, aunque esta ha alcanzado una fitness mayor.



**Figura 5.9 - Curva de progreso, función Rastrigin**

Un indicador de por qué la recombinación aritmética completa ha tenido un rendimiento tan bajo puede verse en la Figura 5.10. A pesar que la forma general de las curvas es similar, su curva de convergencia es demasiado rápida. Esto es un problema común de los AEs y que forma un tema de investigación muy activo. Se lo conoce como **convergencia prematura** y es, por lo general, una propiedad indeseable de los algoritmos basados en poblaciones.

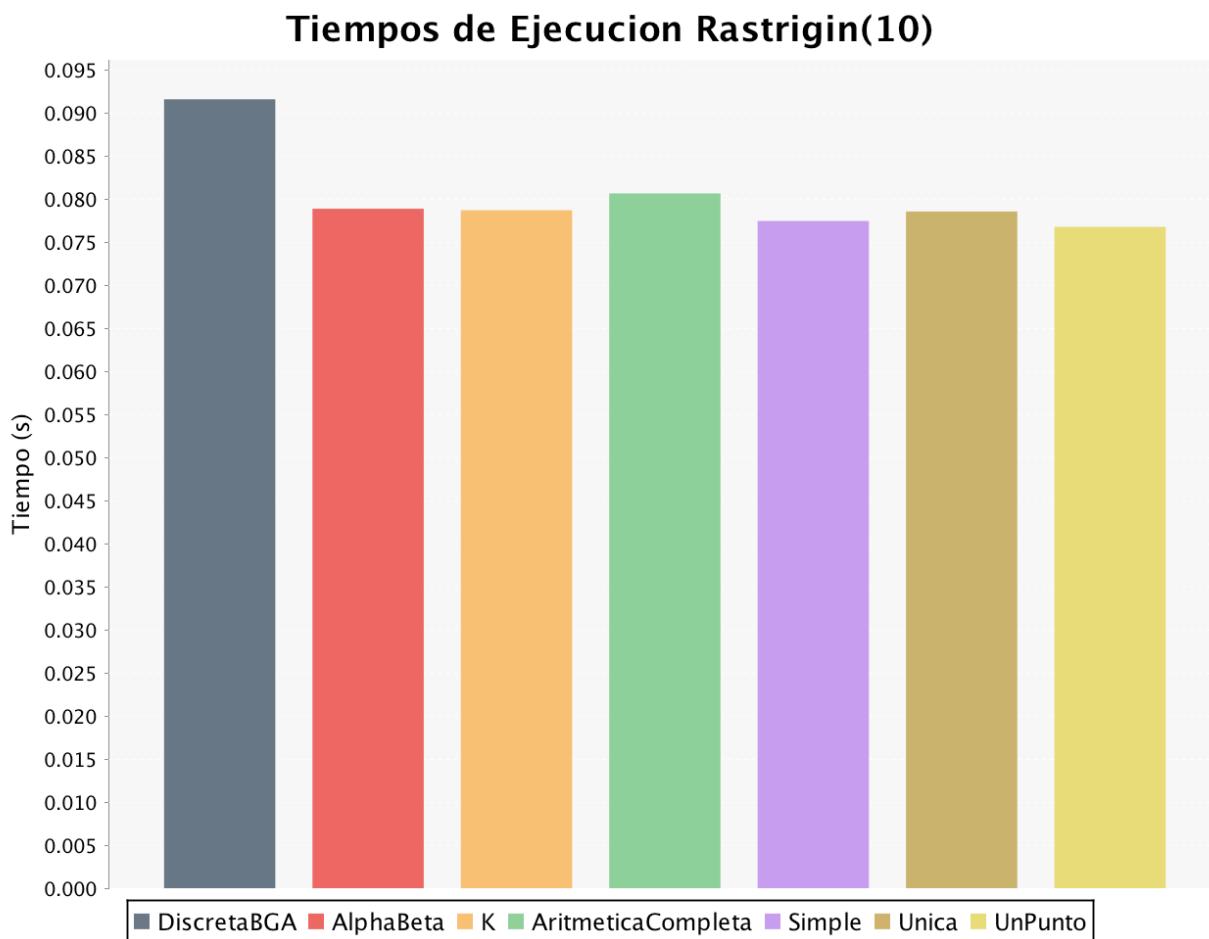
La convergencia de la recombinación Alfa-Beta es la más conveniente de cara a favorecer la exploración y evitar así la convergencia prematura. Sin embargo, aun con una curva de progreso bastante buena, tiene un promedio estándar con respecto a las demás recombinaciones. Posee además una de las desviaciones más altas, por lo que su rendimiento no puede caracterizarse como “bueno”.



**Figura 5.10 - Convergencia, función Rastrigin**

Con respecto a los tiempos de ejecución (Figura 5.11), estos son mucho más elevados que para la función Griewank. Esto por supuesto no es por la complejidad en sí de la función, sino por la cantidad de dimensiones y por el tamaño de la población. Todos los operadores son de orden de complejidad lineal en el tiempo para sus entradas (cantidad de dimensiones para las funciones, cantidad de genes para la mutación, tamaño de la población para la selección, etc.). Esto es una ventaja de los AEs con respecto a otros algoritmos de búsqueda exhaustiva, que suelen ser de orden exponencial para problemas  $NP^{11}$  (problemas difíciles). Puede aumentarse considerablemente el tamaño del problema sin aumentar demasiado el tiempo de ejecución. Se discute con más detalle sobre este tema y otras conclusiones sobre el rendimiento en la sección 5.8.

<sup>11</sup> Puede consultarse más sobre los problemas NP en <http://es.wikipedia.org/wiki/NP-completo>



**Figura 5.11 - Tiempos de ejecución, función Rastrigin**

La mayoría de las ejecuciones han tardado alrededor de 75 milisegundos, con la excepción de la DiscretaBGA, que ha tardado bastante más. Sin embargo, llama la atención tal diferencia cuando las implementaciones realizan operaciones básicas como asignaciones, sumas, multiplicaciones, etc. Para encontrar la diferencia se ha programado una prueba (`TestRendimientoAleatorio` en el directorio `test`) que repite el cuerpo de la recombinación DiscretaBGA y la Aritmética Completa en un bucle. Este se repite cien mil veces sin realizar ninguna otra tarea para minimizar interrupciones, especialmente del recolector de basura de Java. Se toman los tiempos de ambos bucles y se imprimen al final. Los tiempos han sido 32ms para la aritmética completa y 42ms para la DiscretaBGA. Independientemente de los tiempos exactos, que pueden variar entre ordenador y ordenador, interesa el rendimiento relativo, que es un 31.25% peor para la recombinación DiscretaBGA. Esto se debe al uso del generador de números aleatorios, que debe utilizarse por cada gen de cada individuo, y que realiza internamente varias operaciones matemáticas (muchas más que la aritmética completa) para garantizar la uniformidad de los valores.

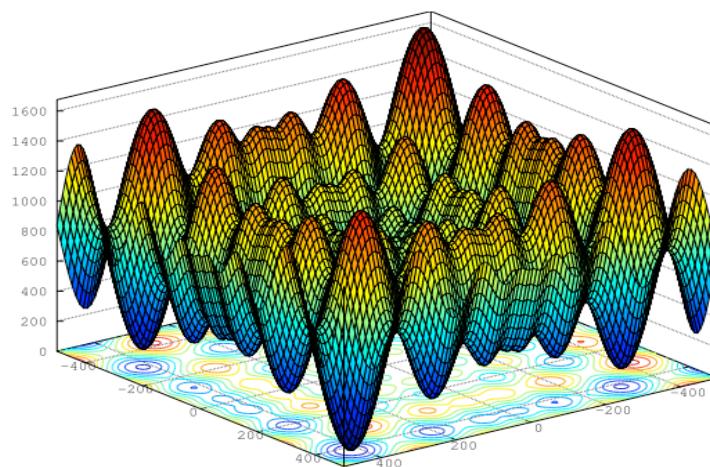
## 5.6 Evaluación de la Función Schwefel

La función Schwefel acepta también  $n$  variables, por lo que es sencillo aumentar la complejidad del problema con tan sólo aumentar la dimensionalidad de la función. La misma se define como:

$$schwefel(\bar{x}) = 418.9829d - \sum_{i=1}^n x_i \sin(\sqrt{|x_i|})$$

En la Figura 5.12 aparece una gráfica para la función en tres dimensiones. El dominio comprende los valores entre [-500, 500] para las  $n$  variables. El mínimo vale cero, en la coordenada (420.9687; ... ; 420.9687), también para todas las variables.

Se ha dejado para esta tercera evaluación esta función por haber resultado la más compleja de las tres, especialmente al aumentar la dimensionalidad. La misma presenta numerosos mínimos locales además de una forma poco regular, lo cual dificulta la exploración. Otra particularidad de esta función es que, aun cerca del mínimo, los mínimos locales tienen un valor alto, especialmente si la cantidad de dimensiones es elevada. Por ello se verá más adelante una gran variación y soluciones que en principio parecerán muy malas.



**Figura 5.12 - Función Schwefel**

Se ejecuta el programa con los siguientes parámetros:

```
-elitismo -benchmark -g 100 -t 2000 -s Torneo -m Normal -desviacion 30 -k 3
-tamañoTorneo 2 -d 20 -alpha 0.1 -beta 0.3
```

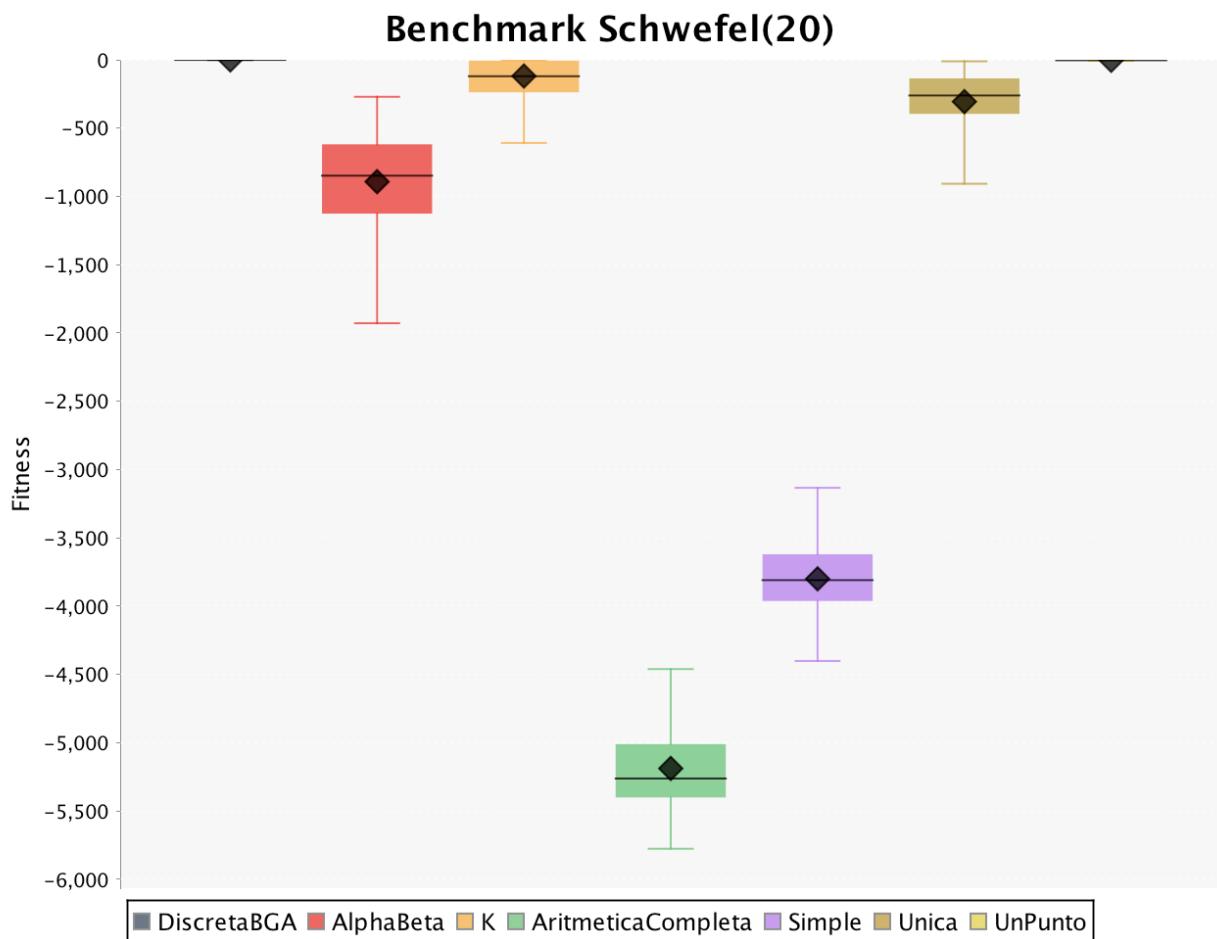
Se ha elegido 20 dimensiones para la función, lo cual complica el problema considerablemente. Se utiliza un tamaño de población de 2000 individuos para intentar

explorar lo suficiente. Destaca también el uso de una mayor desviación para la mutación Normal que en las evaluaciones anteriores. Si bien es deseable mantener los parámetros constantes entre las evaluaciones para una comparación más consistente, se ha decidido aumentar la desviación por motivos de rendimiento. La función Schwefel tiene una forma muy irregular, lo que provoca valores muy altos en la vecindad de los mínimos locales. Esto hace que las soluciones tiendan a aglomerarse en estos mínimos y les sea muy difícil escapar. Una mutación mayor ayuda a los individuos a escapar de estos valles, aunque si es muy elevada complica la convergencia más fina a un mejor óptimo. Mediante mecanismos de prueba y error se ha optado por elegir una desviación de 30.

En la Figura 5.13 se muestra el rendimiento de las distintas recombinaciones. A primera vista puede determinarse el problema de la irregularidad de la función: cuando las ejecuciones no encuentran el mínimo global o uno cercano a él, el valor del fitness resultante es muy malo. No sucedía así con otras funciones donde el rango de la función (valores posibles que puede tomar la función a optimizar) era mucho más pequeño.

Nuevamente, la recombinación aritmética completa es la que peor rendimiento ha demostrado: la mejor fitness que ha podido alcanzar es -4460.46.

Destaca de la Tabla 5.2 el rendimiento y la consistencia de las recombinaciones discretas para esta función. Por lo observado en las trazas de ejecución del algoritmo, esto pareciera deberse a la forma en la que quedan estancadas las soluciones en mínimos locales: por lo general varias de las coordenadas de las soluciones valen 420.9687 (valor correcto para el mínimo global), excepto por algunas coordenadas que tienden a acumularse en valores negativos muy alejados. Las recombinaciones discretas tienen ventaja en este caso, al intercambiar genes entre los padres directamente y no promediarlos. Por ejemplo, recombinar aritméticamente las coordenadas 420.9687 y -420.9687 tendría como resultado 0, lo cual, para esta función, es una muy mala solución. Es decir, pareciera ser sencillo para el algoritmo encontrar algunas coordenadas del mínimo, pero no todas a la vez, por lo que las recombinaciones discretas ayudan al respecto.



**Figura 5.13 - Box Plot, función Schwefel**

Las recombinaciones Única y K se han comportado relativamente bien con respecto a las demás, alcanzando en varias ocasiones el mínimo global, aunque con una desviación alta. Dependiendo del tipo de problema (diseño o repetible), esto podría no ser admisible. Las demás recombinaciones no pueden realmente recomendarse, al menos con estos parámetros.

Por otra parte, una desviación pequeña no es por sí misma una calidad buena, ya que puede darse que la recombinación tenga un rendimiento constantemente malo, y por ende una pequeña desviación.

Recombinación	Peor	Promedio	Mejor	Desviación
DiscretaBGA	-1.71	-0.84	-0.25	0.32
AlphaBeta (0.1, 0.3)	-2168.68	-892.02	-270.58	376.36
K (3, 0.1)	-608.62	-118.59	-0.42	125.07
AritmeticaCompleta (0.1)	-5776.13	-5188.1	-4460.46	300.04
Simple (0.1)	-4401.25	-3799.75	-3133.57	242.71
Unica (0.1)	-963.14	-306.11	-10.73	204.41
UnPunto	-8.62	-3.69	-1.29	1.32

Tabla 5.2 - Resultados de la función Schwefel

La curva de progreso aparece en la Figura 5.14. Destaca el rápido avance de las recombinaciones discretas, que se realiza casi por completo antes de la generación 30. La aritmética completa no ha hecho prácticamente avance alguno, por lo que se desaconseja completamente su uso para esta función. La recombinación K y Única nuevamente presentan curvas muy similares en promedio, aunque viendo las estadísticas de la Tabla 5.2 se aprecia la superioridad de la primera.

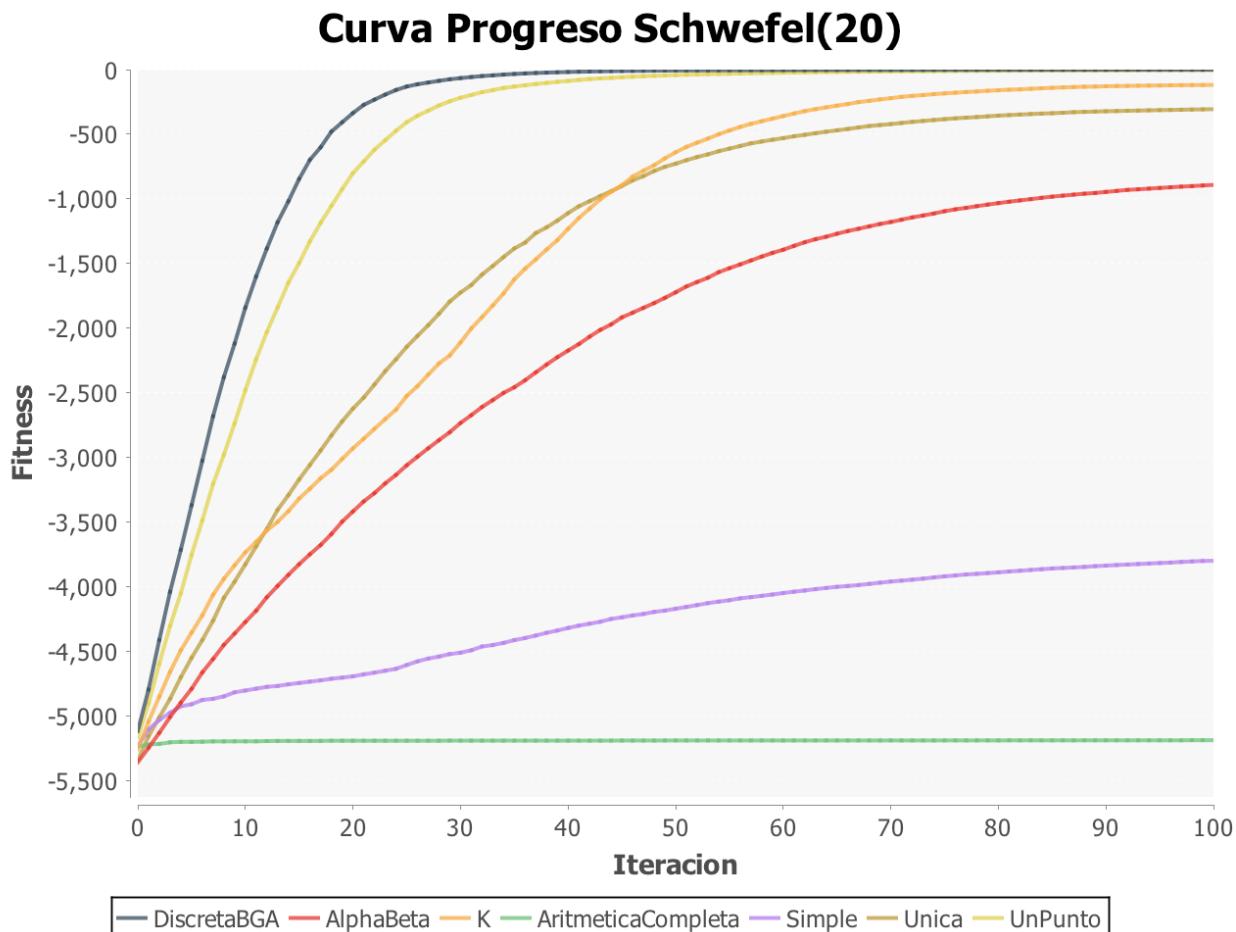


Figura 5.14 - Curva de progreso, función Schwefel

En la curva de convergencia (Figura 5.15) aparece un nuevo fenómeno no visto hasta el momento: desviación en aumento. Podría pensarse que si la curva de progreso es siempre creciente la curva de convergencia debería disminuir. Esto no es así ya que en la curva de progreso se muestra la fitness del mejor individuo, y nada se muestra del resto de la población. En este caso, las recombinaciones discretas han mejorado notablemente entre las generaciones 10 y 20, mientras exploraban la superficie de la función en vez de converger (por lo que aumenta la desviación de la población). Entre las generaciones 20 y 30 encontraron el óptimo global (o un valor cercano a él) por lo que su curva de progreso se achata. En ese instante, y al no generar nuevo material genético, su convergencia disminuye bruscamente.

También se aprecia en la Figura 5.15 la convergencia prematura de la recombinación aritmética completa, lo cual explica su mal rendimiento. La convergencia de las demás recombinaciones tiene una apariencia correcta (ni muy acelerada ni muy lenta), aunque su rendimiento no ha sido tan bueno.

### Convergencia Schwefel(20)

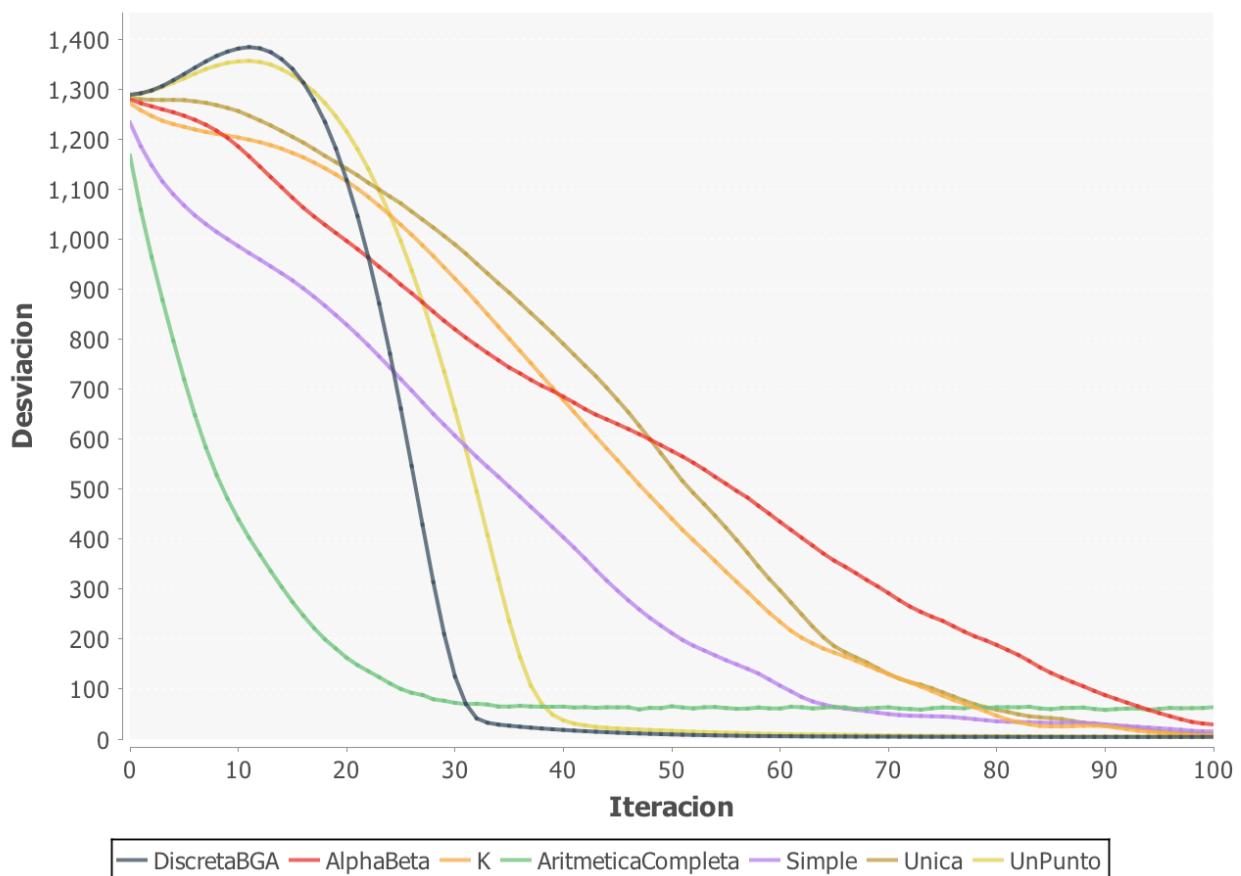
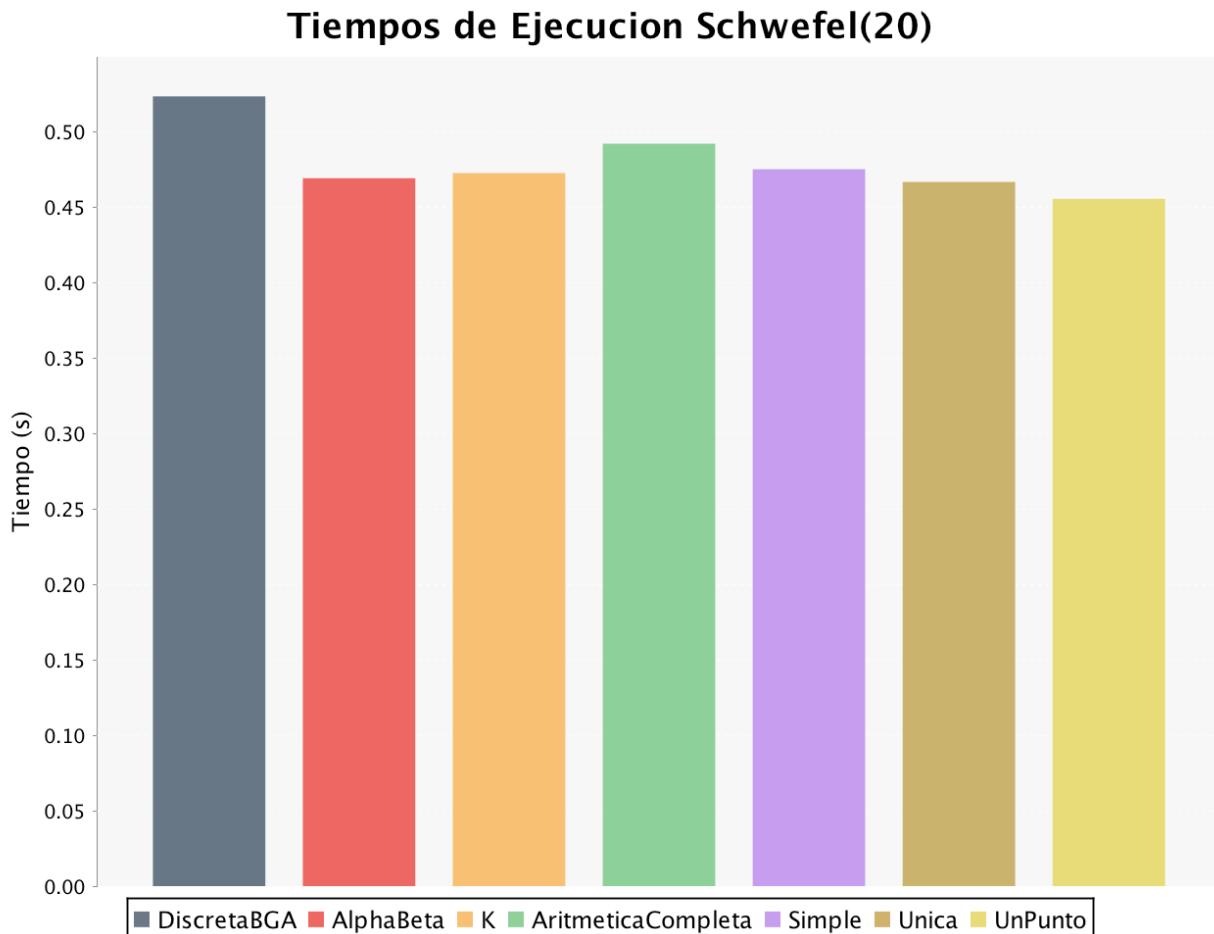


Figura 5.15 - Convergencia, función Schwefel

Con respecto a los tiempos (Figura 5.16), la recombinación DiscretaBGA es nuevamente la peor, aunque la diferencia no es tan pronunciada (alrededor del 16%). Las ejecuciones tomaron en su mayoría más de 400 milisegundos, lo cual es menos susceptible a interrupciones de sistema y otras anomalías que pudieran distorsionar las mediciones de los tiempos. Se consideran entonces las mediciones como más fiables.



**Figura 5.16 - Tiempos de ejecución, función Schwefel**

## 5.7 Evaluación del Valor de K

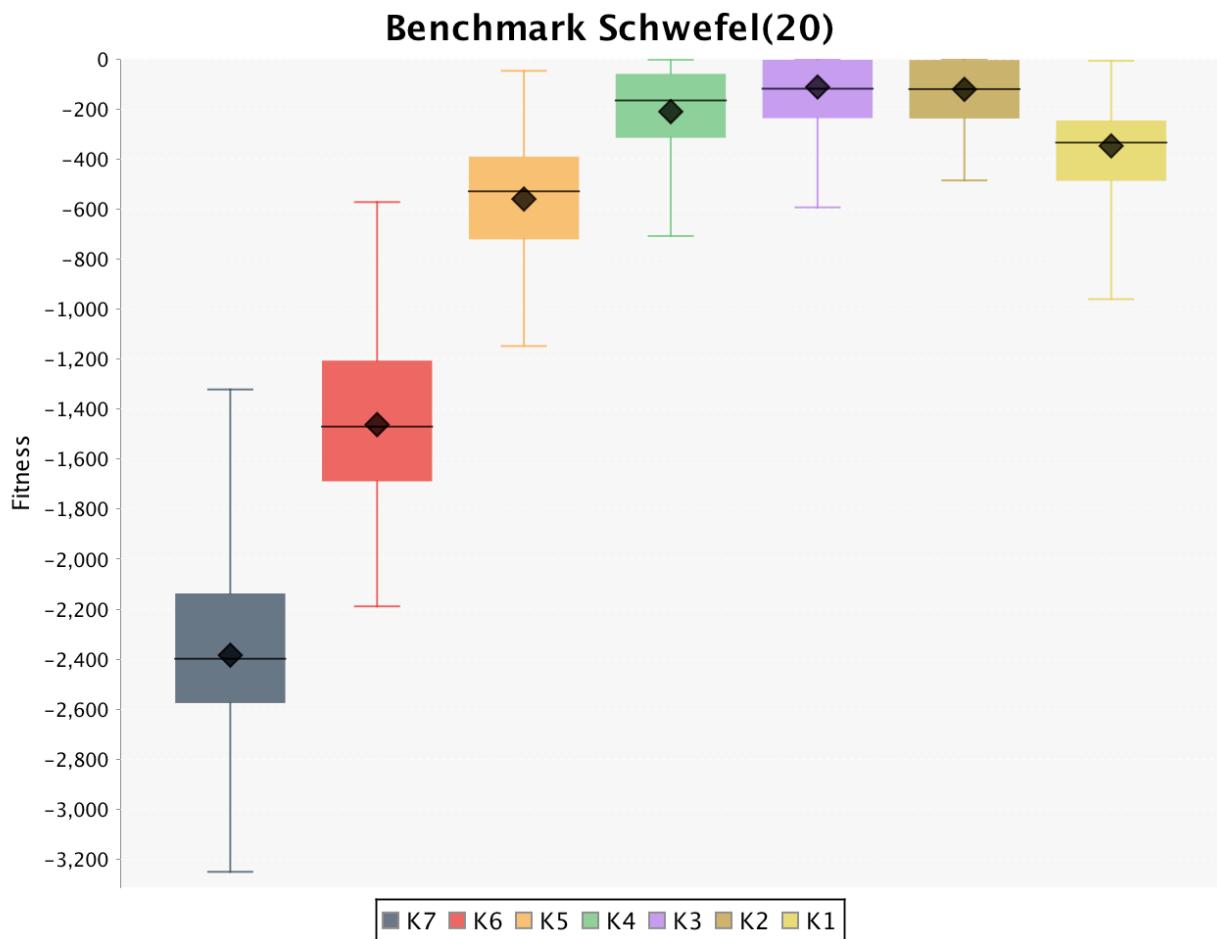
Como ya se ha comentado, la recombinación K es una generalización de otras recombinaciones, particularmente la aritmética completa (donde  $k$  es igual a la dimensión de la función) y la única (donde  $k$  es igual a 1). Esta recombinación permite probar distintos valores de  $k$  con el objetivo de encontrar aquel que otorgue un mejor rendimiento. Es por ello que en este apartado se evalúa brevemente el rendimiento del algoritmo para distintos valores de  $k$ , mientras que se mantienen fijos todos los demás parámetros.

Para esta prueba se ha realizado un modo de ejecución completamente independiente al resto del programa, llamado `TestValorK` en el directorio `test`. Se utiliza nuevamente la misma configuración que para la función Schwefel:

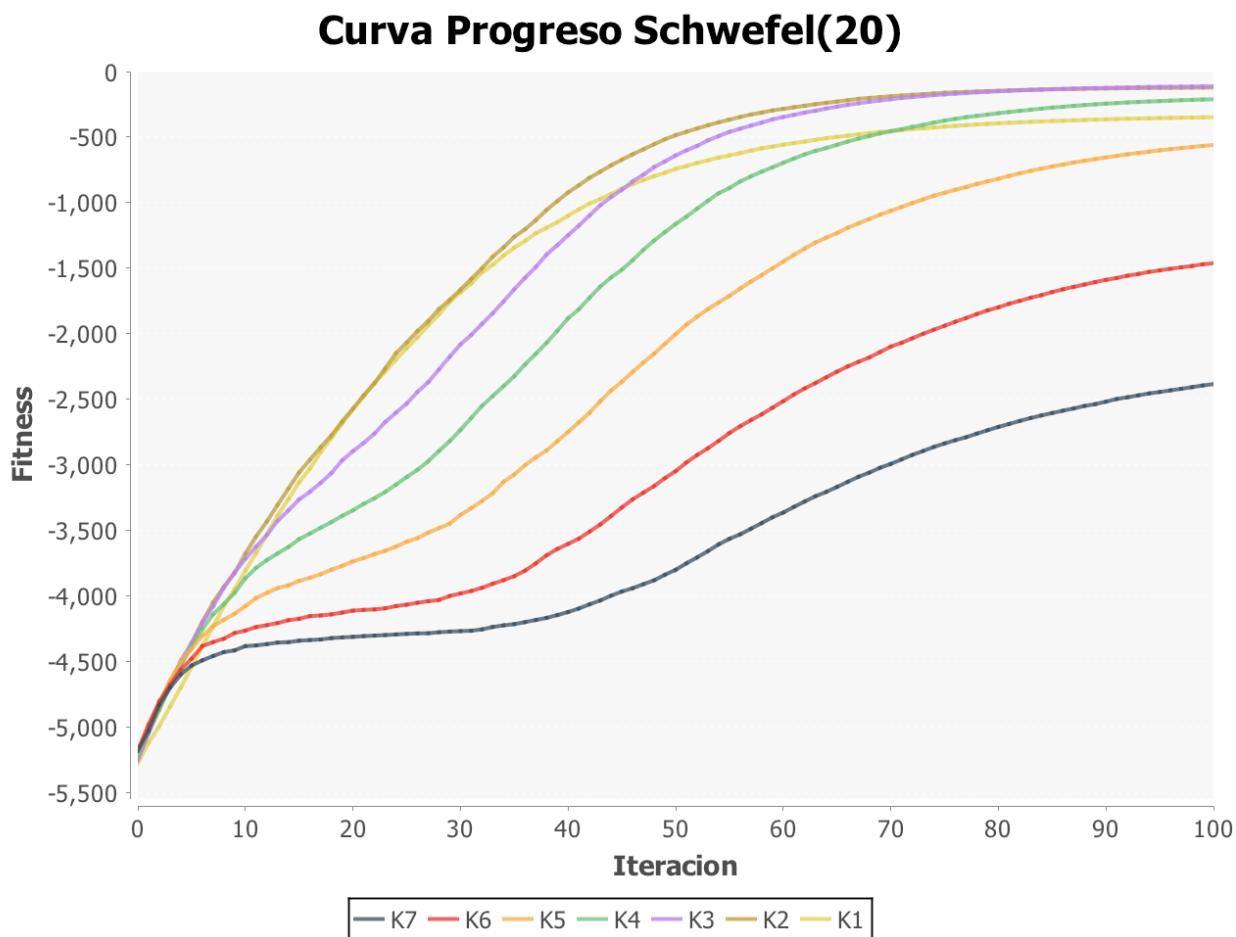
```
-elitismo -benchmark -g 100 -t 2000 -s Torneo -m Normal -desviacion 30
-k 3 -tamañoTorneo 2 -d 20 -alpha 0.1 -r K
```

sólo que esta vez no se utiliza ningún valor de beta y se especifica la recombinación K.

La metodología ha sido ejecutar el algoritmo 150 veces para valores de  $k$  entre 1 y 7. Los resultados se muestran en la Figura 5.17. Se confirman aquí las intuiciones de las evaluaciones anteriores: al aumentar el valor de  $k$  se disminuye el rendimiento. Sin embargo, el valor  $k = 1$  no pareciera respetar esta tendencia, demostrando un rendimiento mucho peor que  $k = 2$ .



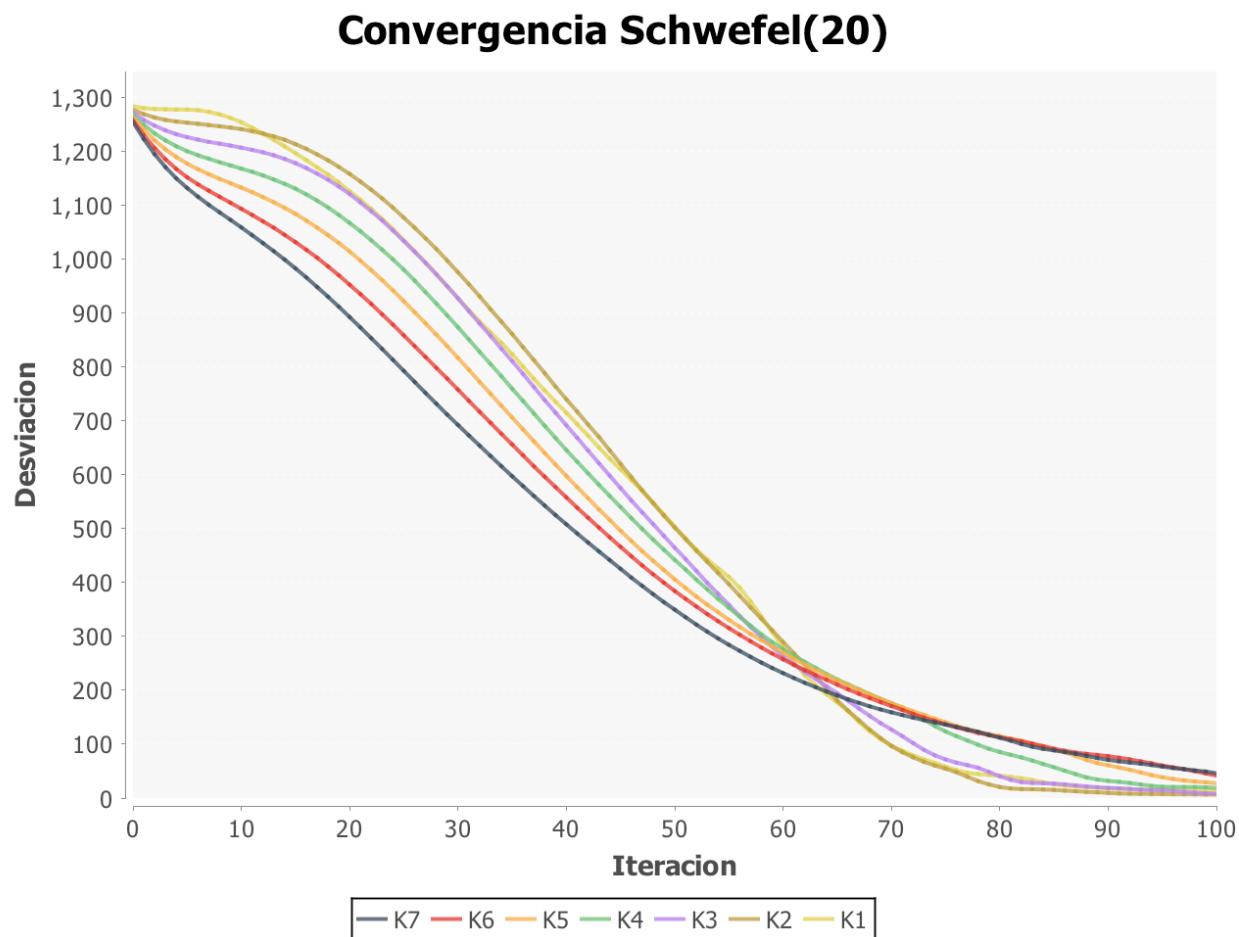
**Figura 5.17 - BoxPlots para los distintos valores de k**



**Figura 5.18 - Curvas de Progreso para los distintos valores de  $k$**

La Figura 5.18 muestra el comportamiento único de utilizar  $k = 1$ , que no sigue la forma de las demás curvas, arrancando de manera más acelerada y achatándose bruscamente. Por cuestiones de espacio no se han incluido las gráficas de valores de  $k$  superiores a 7, pero puede imaginarse cómo las curvas se achatan al aumentar el valor de  $k$ , hasta el punto de tener una curva completamente plana para  $k = n$ , como se mostró en la Figura 5.14.

En la curva de convergencia (véase la Figura 5.19) aparece un punto de inflexión para las distintas recombinaciones en la generación  $\sim 65$ . A partir de allí la recombinación con  $k = 4$  separa a aquellas que comenzaron convergiendo más rápidamente y obtuvieron malos resultados de aquellas que comenzaron más lentamente pero que llegaron más cerca del óptimo global. Si bien es una observación interesante, no debería tomarse como concluyente ni sobre el valor de  $k = 4$ , ni sobre la cantidad de generaciones (65), ya que difícilmente se traten de conclusiones generales a todas las demás funciones.



**Figura 5.19 - Convergencia de los distintos valores de k**

## 5.8 Discusión Final de los Resultados de Evaluación

En este estudio se ha intentado determinar el impacto del operador de recombinación y no necesariamente encontrar la mejor solución en todas las ejecuciones. Es por ello que se ha establecido una serie de parámetros estándar reutilizados en todas las pruebas. La elección de los mismos no es trivial y ha requerido varios ensayos de prueba y error. Ciertas combinaciones de parámetros, lo que de ahora en adelante se denominará genéricamente *configuración*, favorecían especialmente a algunas recombinaciones y perjudicaban a otras. Otras configuraciones no conseguían obtener buenos resultados en ninguna de las ejecuciones. Por último, otras encontraban siempre el óptimo global independientemente de qué recombinación se utilice, particularmente para tamaños de población elevados. Se ha optado entonces por una configuración neutra, para intentar aislar el efecto de la recombinación y estudiarlo por separado. En cualquier caso, esta será neutra para el conjunto de funciones probadas, y poco puede concluirse sobre su comportamiento frente a otros problemas o funciones, y mucho menos sobre su rendimiento en el caso general. Sin

embargo, sí pueden sacarse conclusiones sobre el rendimiento para las tres funciones evaluadas, particularmente sobre el **tiempo de ejecución, memoria utilizada, convergencia y curva de progreso** o fitness. Se incluye en la Tabla 5.3 todos los resultados obtenidos hasta ahora para una referencia más sencilla. Recuérdese que la fitness máxima posible, es decir el óptimo global, es siempre igual a cero.

Recombinación	Función	Peor	Promedio	Mejor	Desviación
AlphaBeta	Griewank	-0.16	-0.01	0	0.02
	Rastrigin	-3.12	-0.75	0	0.81
	Schwefel	-2168.68	-892.02	-270.58	376.36
AritmeticaCompleta	Griewank	-0.09	-0.01	0	0.01
	Rastrigin	-10.53	-5.05	-0.7	2.14
	Schwefel	-5776.13	-5188.1	-4460.46	300.04
DiscretaBGA	Griewank	-1.69	-0.35	0	0.35
	Rastrigin	-1.92	-0.65	-0.07	0.44
	Schwefel	-1.71	-0.84	-0.25	0.32
K	Griewank	-0.05	-0.01	0	0.01
	Rastrigin	-1.01	-0.09	0	0.27
	Schwefel	-608.62	-118.59	-0.42	125.07
Simple	Griewank	-1.44	-0.32	0	0.34
	Rastrigin	-2.64	-0.54	0	0.61
	Schwefel	-4401.25	-3799.75	-3133.57	242.71
Unica	Griewank	-0.25	-0.02	0	0.04
	Rastrigin	-3.26	-0.47	0	0.63
	Schwefel	-963.14	-306.11	-10.73	204.41
UnPunto	Griewank	-1.61	-0.33	0	0.36
	Rastrigin	-3.27	-1.08	-0.15	0.58
	Schwefel	-8.62	-3.69	-1.29	1.32

Tabla 5.3 - Resultados finales de la ejecución

### 5.8.1 Tiempos de Ejecución

En cuanto a la evaluación computacional, la RecombinacionDiscretaBGA ha sido sin dudas la que más tiempo de CPU ha consumido. A esta le sigue la RecombinacionAritmeticaCompleta. Las demás no presentan una diferencia sustancial, por lo que pueden considerarse equivalentes al estar dentro del margen de error de la medición. Con respecto a la primera, ya se comentó anteriormente que el peor rendimiento se debe al uso intensivo del generador de números aleatorios. Con respecto a la segunda, la pequeña penalización de rendimiento se debe básicamente a que debe realizar una operación matemática por cada gen, mientras que las demás la realizan solamente para un subconjunto de ellos.

Con respecto a la complejidad computacional, todas las recombinaciones son equivalentes. El tamaño de entrada del algoritmo está determinado por el tamaño de la población  $t$  y la dimensionalidad de la función  $d$ <sup>12</sup>. Todas las recombinaciones tienen un coste en tiempo del orden de  $O(d)$ , ya que realizan una cantidad de operaciones fija por cada gen del individuo, es decir, por cada dimensión de la función. A su vez, el algoritmo completo invoca a la recombinación por cada par de padres, por lo que será del orden de  $O(t/2) * O(d)$ , lo cual es equivalente a  $O(td)$ . Esto quiere decir que si bien algunas recombinaciones son más lentas que otras, en el análisis asintótico (cuando el tamaño de la entrada es lo suficientemente grande), todas son equivalentes al ser todas de coste polinómico.

### 5.8.2 Memoria Utilizada

Además del tiempo, otra medida para la complejidad computacional es el espacio o memoria consumida. Limitando el análisis al operador de recombinación, nuevamente todas las variantes son equivalentes. En todos los casos el único requisito de memoria es poder instanciar los dos hijos, a la vez que se mantienen las instancias de los padres. Son por ello del orden de  $O(4d)$ , lo cual es equivalente a  $O(d)$ . El algoritmo en su conjunto dependerá también del tamaño de la población, por lo que es del orden de  $O(2td)$ , ya que requiere almacenar por un momento a todos los padres e hijos simultáneamente, para que el modelo poblacional decida a quiénes reemplazar.

### 5.8.3 Convergencia

Aunque la curva de convergencia por si sola no otorga mucha información, es importante analizarla cuando el algoritmo no funciona correctamente o su consistencia varía mucho. No existe a priori una cantidad ideal de iteraciones necesarias o recomendadas para encontrar una solución, aunque informalmente puede considerarse entre 10 y 20 iteraciones como relativamente pocas. Como se explicó en la sección 2.3, debe encontrarse un balance entre exploración y explotación. En todas las evaluaciones, la RecombinacionAritmeticaCompleta ha explotado demasiado a los buenos individuos, no explorando lo suficiente. Las demás recombinaciones aritméticas han tenido en general

---

<sup>12</sup> Podría considerarse la cantidad de generaciones como influyente en el tamaño de entrada del algoritmo, aunque se deja fuera del análisis posterior.

buenas curvas de progreso, en el sentido que no han sido demasiado veloces y permitieron más generaciones de exploración antes de converger por completo y dejar de explorar. Llama la atención la convergencia de las recombinaciones discretas que han sido, exceptuando para la función Griewank, de las más veloces y aun así han obtenido muy buenos resultados.

#### 5.8.4 Curva de Progreso

Las recombinaciones discretas han resultado ser las más consistentes en cuanto a la mejor fitness obtenida. Son las que menor desviación han tenido y sus promedios son los más cercanos al óptimo global. Le siguen en calidad las recombinaciones K y Única, aunque sus promedios están más alejados al óptimo. Aun sin una definición formal de calidad, estas son las recombinaciones que podrían catalogarse como *buenas*. Las demás, si bien han encontrado en ocasiones el óptimo global, no son lo suficientemente consistentes, o la calidad de las soluciones empeora considerablemente al aumentar la complejidad del problema. De todas estas, las que peor se han comportado son, sin duda, la AritmeticaCompleta y Simple. Aunque su comportamiento es mejor cuando la dimensionalidad es baja (ya que se comportan de manera similar a las otras recombinaciones), su rendimiento empeora drásticamente al aumentar la dimensionalidad.

En líneas generales, no es recomendable recombinar *aritméticamente* una gran cantidad de genes (valor de  $k$ ), ya que reduce en gran medida la exploración, dando lugar a la convergencia prematura de la población. Esto no supone un problema en funciones sencillas (unimodales o de baja dimensionalidad), pero comienza a serlo de manera evidente al superar las 10 dimensiones. Utilizar un valor de  $k$  de 2 ó 3 ha dado resultados casi idénticos y ha probado ser lo más efectivo de cara a obtener buenos resultados. Esto no ha sido un problema para las recombinaciones discretas, particularmente la DiscretaBGA, que opera sobre todos los genes de cada individuo.

# Capítulo 6: Conclusiones y Trabajo Futuro

Se ha realizado en este trabajo un estudio sobre los Algoritmos Genéticos, vistos como procesos genéricos de búsqueda en un espacio intratable para los algoritmos convencionales (sección 2.3). Se ha visto cómo con una serie de operadores sencillos y fáciles de entender (sección 2.2 y Capítulo 3), se consiguen resultados sorprendentemente buenos (Capítulo 5). Presentan además un orden de complejidad computacional muy conveniente (sección 5.8), al ser este polinómico de bajo grado. Esto es de gran interés científico y práctico, ya que permite aumentar considerablemente el tamaño y complejidad del problema sin mayores inconvenientes de espacio y tiempo de CPU.

Otra característica destacable de los AGs es su carácter genérico, en tanto que no necesitan de heurísticas o conocimiento específico del problema a resolver para obtener un buen desempeño. El único punto donde se debe aplicar conocimiento específico sobre el problema se da en la elección de la representación de las soluciones (su genotipo y fenotipo). Una vez hecho esto, los demás operadores utilizarán estas estructuras de datos de manera genérica e independiente del problema a resolver. Por ejemplo, un operador de mutación podría cambiar el valor de un gen de un 0 a un 1, sin saber realmente cómo este cambio ayudaría a obtener una solución al problema y, de hecho, sin conocer siquiera cuál es el problema que está resolviendo. Dicha acción es independiente del problema específico que se esté intentando resolver, y de aquí su carácter genérico.

Este carácter genérico y la aplicabilidad a múltiples problemas es en principio una gran ventaja de los AGs. Sin embargo, debe recordarse el teorema de *No Free Lunch* (*no existe el almuerzo gratuito*) introducido en [20]. Este explica que, para ciertos tipos de problemas como la optimización [21], el rendimiento de un algoritmo, promediado entre todas las instancias de problemas de ese tipo, es el mismo que cualquier otro algoritmo. Concluye que si un algoritmo obtiene un rendimiento superior en un problema particular, este será inferior en otras instancias de dicho problema. El rendimiento podría ser tanto en aspectos computacionales como en la calidad de la solución obtenida (para el caso de los algoritmos estocásticos y/o aproximados). Por ejemplo, en la optimización de funciones, se podría tener un AG cuyo desempeño sea muy bueno para la función Rastrigin. No existe “almuerzo gratuito” en el sentido que este algoritmo particular no será igual de bueno para todas las

demás funciones. Por ejemplo, este podría tener un rendimiento muy malo para la función Schwefel y viceversa (un algoritmo muy bueno para optimizar la función Schwefel podría no serlo para la función Rastrigin). Sin embargo, los AGs tienen la ventaja de ser completamente parametrizables. Cuando se habla de un algoritmo “particular”, no se hace referencia a la implementación completa del AG, sino más bien a su configuración. Si se tiene una configuración cuyo rendimiento es muy malo para la función Schwefel, no será necesario reescribir el algoritmo, sino encontrar los parámetros adecuados (probablemente mediante prueba y error). De esto se obtienen dos importantes conclusiones: la primera es que los AGs pueden adaptarse a una multitud de problemas de manera eficiente sin reescribir el algoritmo; la segunda es que difícilmente exista una configuración cuyo rendimiento sea bueno para todos los problemas de optimización, por lo que se deberá encontrar una configuración adecuada para cada problema.

En este trabajo se ha estudiado cómo distintas configuraciones obtienen distintos rendimientos para los tres problemas estudiados (funciones Griewank, Rastrigin y Schwefel), como se estableció en la segunda conclusión anterior. No ha sido objeto de estudio obtener el mejor rendimiento en todos los casos, ya que de las distintas configuraciones posibles, se ha optado por una lo más neutra posible para aislar el efecto de la recombinación. Esto permite una comparación más consistente entre las distintas funciones estudiadas, permitiendo ver las tendencias generales de curvas de progreso y convergencia.

Aunque el trabajo se centra en estudiar el efecto de la recombinación en el rendimiento del AG, es importante destacar también el efecto de la **presión selectiva** como un factor clave para la velocidad de convergencia. Este concepto indica hasta qué punto los individuos con características desfavorables podrán sobrevivir, por lo que está relacionado con la intensidad o agresividad del proceso de selección. Medido por algunos autores [3] como **tiempo de adquisición** (*takeover time*), indica la velocidad con la cual el mejor individuo ocuparía la población completa al utilizarse repetidamente el operador de selección. Entre mayor sea el tiempo de adquisición, menor será la presión selectiva y viceversa. En un extremo, cuando no hay presión selectiva en absoluto, la búsqueda se torna completamente estocástica, por lo que se explora el espacio de búsqueda de manera aleatoria. En el otro extremo, cuando la presión selectiva es muy grande, la búsqueda será muy poco estocástica, por lo que actuará como un algoritmo de Gradiente Descendiente [15]. Por tanto, una presión selectiva muy grande equivale a una mayor explotación,

mientras que una muy pequeña favorece una mayor exploración. Por ejemplo, un parámetro que permite controlar la presión selectiva es el tamaño del torneo en la `SeleccionTorneo`. Un valor pequeño como 2 otorga grandes posibilidades a los individuos poco aptos para que estos se reproduzcan (exceptuando al peor de todos, que nunca podrá ganar un torneo). Por el contrario, un valor muy grande favorecerá a los mejores individuos. Por ejemplo, si el valor es igual al tamaño de la población y el torneo es con reemplazamiento, el mejor individuo será elegido siempre como el único que podrá reproducirse. Por supuesto, esto es un ejemplo artificial y se desaconseja completamente utilizar dicho valor.

El AG implementado en este trabajo presenta una presión selectiva moderadamente elevada. En la literatura es común encontrar AGs que se ejecutan por cientos e incluso miles de generaciones. El AG de este trabajo se ha ejecutado por no más de 100 generaciones y para algunas recombinaciones tan sólo 20 generaciones han sido suficientes. Además del operador de selección, el modelo poblacional ha influido ampliamente en la presión selectiva del algoritmo. Recordar que sólo se ha implementado el modelo generacional, que reemplaza la población entera en cada generación por los hijos recién creados. Si no se utilizara ningún operador de recombinación ni mutación, este modelo poblacional eliminaría en el mejor de los casos únicamente al peor individuo de la población, que nunca podrá ganar un torneo. Sin embargo, en el peor de los casos, siempre se elegirá al mejor individuo para participar de todos los torneos, por lo que siempre ganará él. Aunque poco probable, este hecho generaría una población de hijos todos descendientes del mismo parent. Al utilizarse el modelo generacional, todos estos hijos formarían la nueva población, por lo que el tiempo de adquisición sería igual a una generación. El modelo poblacional *steady-state* ayudaría a combatir este fenómeno, permitiendo que sólo  $m$  de los  $N$  hijos creados (con  $m < N$ ) puedan reemplazar a individuos de la generación anterior. Aun a falta de pruebas empíricas que lo corroboren (ya que sería necesario implementar para ello otros modelos poblacionales), se concluye que una presión selectiva tan elevada no ha sido beneficiosa para la obtención de buenas soluciones, especialmente para algunas recombinaciones aritméticas. Sin embargo, sí que lo ha sido para el rendimiento computacional del algoritmo, al poder encontrar una solución siempre en menos de 1 segundo.

## Trabajo Futuro y Ampliaciones

Se han señalado a lo largo de toda la memoria pequeñas secciones que podrían mejorarse o ampliarse: algunas debido a la falta de tiempo en el desarrollo, otras para simplificar el mismo y otras simplemente por estar fuera del alcance de este trabajo. Se recopilan a continuación varias líneas de trabajo futuro y ampliaciones.

En primera instancia se podría ampliar el repertorio de operadores de selección, ya que sólo se ha implementado uno (selección por torneo). Aquel más sencillo e inmediato sería la selección por Ranking, como se muestra en [1].

Con respecto a la mutación, se han implementado los dos tipos básicos más comunes: la uniforme y la normal. Aun así, podrían implementarse alternativas más sofisticadas, aunque en principio no sería tan necesario como para el caso de la selección. Otra alternativa interesante que se ha probado durante la fase de desarrollo, pero que no se incluyó en la versión final es la mutación variante. En esta se utiliza inicialmente una distribución normal con una desviación elevada y con el pasar de las generaciones esta se reduce progresivamente. Aunque se podría implementar sin grandes complicaciones, su uso comienza a entrar en el terreno de las estrategias evolutivas, que utilizan una mutación normal asociada a cada individuo y que muta junto con él.

Otro de los operadores que también podría ampliarse es el de terminación. Por el momento sólo se ha implementado la terminación fija, que termina la ejecución tras un numero prefijado de iteraciones. Existen varias implementaciones alternativas muy frecuentes que incluyen criterios dinámicos en vez de estáticos. Por ejemplo, terminar la ejecución cuando se observa que la población ha convergido por completo, cuando se cruza un determinado umbral (de convergencia o de fitness), cuando no se detectan grandes cambios durante algunas generaciones, etc.

Como se comentó al final de la sección de conclusiones, un modelo poblacional *steady-state* sería conveniente para que el algoritmo no convergiera tan rápidamente. El primer paso sería implementar dicho modelo. A continuación, dicho modelo necesita un operador de selección de supervivientes, para determinar qué individuos serán reemplazados por los nuevos  $m$  hijos generados. Para ello pueden utilizarse operadores ya implementados o crear algunos nuevos, por ejemplo que contemplen la edad de los individuos.

Como ejemplo de una funcionalidad completamente nueva, podría desarrollarse un sistema capaz de ampliar el repertorio de funciones disponibles de manera dinámica. Por

ejemplo utilizando un *parser* (analizador sintáctico) de expresiones matemáticas como ANTLR<sup>13</sup>. De esta manera, el usuario podría teclear como parámetro la expresión de la función que deseé optimizar, en vez de que esta tenga que estar precompilada e incluida en el propio programa. Se deja también como posible mejora el ampliar el repertorio de funciones de prueba (*benchmarking*), para analizar la consistencia y rendimiento de las recombinaciones implementadas.

Otra implementación pendiente de prueba es programar, para la recombinación K, la elección de ejes aleatoria sin reemplazamiento. Aunque con baja probabilidad, en la versión actual del proyecto puede suceder que se elija el mismo eje más de una vez, lo cual es una propiedad indeseable, aunque simplifica la implementación del operador.

Finalmente, se recuerda que la aplicación Web permite tener varios recursos distintos según su dirección URL. Hasta el momento el único recurso existente tiene la dirección /api/ag, que ejecuta de manera remota el algoritmo genético y devuelve los resultados. Pueden agregarse nuevos recursos con funcionalidades interesantes, como estadísticas de ejecuciones pasadas, crear funciones nuevas de manera remota (por ejemplo en alguna pantalla de administración), chequeos de estado del servidor (para monitorización remota), envío de resultados por correo electrónico (para ejecuciones largas), etc.

Por último, se deja como trabajo futuro estudiar el efecto de los distintos valores de alfa para las recombinaciones aritméticas. De particular interés es utilizar valores negativos, ya que ubicarían a los hijos por fuera del espacio entre los padres, por lo que alejaría a los hijos en vez de acercarlos. Otra posible línea de investigación sería utilizar un valor de alfa distinto en cada recombinación, en vez de uno prefijado antes de comenzar la ejecución. Este valor podría tomarse de un generador de números aleatorios siempre que se necesite recombinar dos padres. La hipótesis es que esto podría generar nuevo material genético y evitar la convergencia de todos los individuos a un mismo punto (el promedio).

Otra línea de investigación pendiente es continuar el estudio sobre los distintos valores de  $k$ . Esto ha arrojado resultados curiosos sobre determinados valores específicos de interés ( $k < 3$ ). Sería deseable estudiar el por qué utilizar un valor de  $k = 2$  tiene un rendimiento mucho mejor que  $k = 1$ . En líneas más generales, podría estudiarse el efecto de la cantidad de ejes de recombinación, ya sea para recombinaciones discretas como aritméticas.

---

<sup>13</sup> Página web oficial con documentación y descarga en <http://www.antlr.org/>



# Bibliografía

- [1] A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computing*. Springer, 2003.
- [2] A.E Eiben. *Multi-parent Recombination*. Leiden University, Netherlands.
- [3] Thomas Bäck. *Selective Pressure in Evolutionary Algorithms: a Characterisation of Selection Mechanisms*. University of Dortmund, Department of Computer Science.
- [4] Lin Hansheng and Kang Lishan. *Balance between Exploration and Exploitation in Genetic Search*. Wuhan University, Whugan 430072, China.
- [5] Johannes M. Dieterich and Bernd Hartke. *Empirical review of standard benchmark functions using evolutionary global optimization*. Kiel, Germany, 2012.
- [6] Test problems for Constrained Global Optimization,  
[http://www-optima.amp.i.kyoto-u.ac.jp/member/student/hedar/Hedar\\_files/TestGO\\_files/Page422.htm](http://www-optima.amp.i.kyoto-u.ac.jp/member/student/hedar/Hedar_files/TestGO_files/Page422.htm)
- [7] Surjanovic, S. & Bingham, D. (2013). *Virtual Library of Simulation Experiments: Test Functions and Datasets*. Retrieved June 29, 2015, from <http://www.sfu.ca/~ssurjano>.
- [8] Stjepan Picek, Domagoj Jakobovic and Marin Golub. *On the Recombination Operator in the Real-Coded Genetic Algorithms*. Zagreb, Croatia.
- [9] GEATbx: *Genetic and Evolutionary Algorithm Toolbox for use with Matlab*.  
<http://www.geatbx.com/docu/algindex-03.html>.
- [10] Ronald W. Morrison and Kenneth A. De Jong. *Measurement of Population Diversity*. George Mason University, USA.
- [11] Heinz Mühlenbein and Dirk Schlierkamp-Voosen. *Predictive models for the breeder genetic algorithm I. Continuous parameter optimization*. Sank Augustin 1, Germany.
- [12] A.E. Eiben, P-E. Raué and Zs. Ruttkay. *Genetic algorithms with multi-parent recombination*. Proceedings of the 3rd Conference on Parallel Problem Solving from Nature, LNCS 866, Springer-Verlag, 1994, pp.78-87.
- [13] Plotly. *Intro to Boxplots*. <https://plot.ly/box-plot/>.
- [14] Marek Obitko. *Introduction to Genetic Algorithms: Search Space*.  
<http://www.obitko.com/tutorials/genetic-algorithms/search-space.php>
- [15] Huayang Xie and Mengjie Zhang. *Tuning Selection Pressure in Tournament Selection*.
- [16] D. E. Goldberg and K. Deb. *A comparative analysis of selection schemes used in genetic algorithms*. Foundations of Genetic Algorithms, pp. 69–93, 1991.

- [17] Thomas Weise, Michael Zapf, Raymond Chiong, and Antonio J. Nebro. *Why Is Optimization Difficult?*
- [18] Tobias Blackly and Lothar Thiele. *A comparison of Selection Schemes used in Genetic Algorithms.* Swiss Federal Institute of Technology, Zurich.
- [19] Günter Rudolph. *On Takeover Times in Spatially Structured Populations: Array and Ring.* Dortmund, Germany
- [20] Wolpert D.H. and Macready W.G. *No Free Lunch Theorems for Search.* Technical Report SFI-TR-95-02-010, Santa Fe Institute, 1995
- [21] Wolpert, D.H., Macready, W.G. *No Free Lunch Theorems for Optimization.* IEEE Transactions on Evolutionary Computation **1**, 67, 1997  
<http://ti.arc.nasa.gov/m/profile/dhw/papers/78.pdf>