Títol: **Estudio de bloques constructivos en algoritmos genéticos**

Volum: **1/1**

Alumne: **Angel Murias Rodríguez**

Director/Ponent: **Lluís Antoni Belanche Muñoz**

Departament: **LSI**

Data: **10.12.2007**

DADES DEL PROJECTE

Títol del Projecte: **Estudio de bloques constructivos en algoritmos genéticos**

Nom de l'estudiant: **Angel Murias Rodríguez**
Titulació: **Enginyería en Informàtica**
Crèdits: **30**
Director/Ponent: **Lluís Antoni Belanche Muñoz**
Departament: **LSI**

MEMBRES DEL TRIBUNAL *(nom i signatura)*

President: **Enrique Romero Merino**

Vocal: **Josep Grané Manlleu**

Secretari: **Lluís Antoni Belanche Muñoz**

QUALIFICACIÓ

Qualificació numèrica:
Qualificació descriptiva:

Data:

Estudio de bloques constructivos en algoritmos genéticos

Angel Murias Rodríguez

Barcelona

10 de diciembre de 2007

Índice general

1 Introducción **1**

1.1 Motivación 1
1.2 Objetivos del proyecto 3
1.3 Organización de la memoria 3

2 Conocimientos básicos y conceptos fundamentales **4**

2.1 Funcionamiento de un algoritmo evolutivo 5
2.2 Población de individuos 7
2.3 Inicialización 8
2.4 Evaluación 8
2.4.1 Mapeo 8
2.4.2 Restricciones 9
2.5 Selección 9
2.5.1 Selección uniforme 10
2.5.2 Selección proporcional al *fitness* 10
2.5.3 Selección por torneo 14
2.5.4 *Remainder stochastic sampling without replacement* 14
2.5.5 Selección por corte 14
2.5.6 Selección plus 14
2.5.7 Selección coma 14
2.5.8 Elitismo 15
2.5.9 *Generation gap* 15
2.6 Recombinación 15
2.7 Mutación 16
2.8 Criterio de terminación 16
2.9 Ajuste de parámetros 17
2.10 Medidas de rendimiento 17
2.11 Familias de algoritmos evolutivos 18
2.11.1 Estrategias de evolución (ES) 18
2.11.2 Programación evolutiva (EP) 19
2.11.3 Programación genética (GP) 19
2.11.4 Algoritmos genéticos (GA) 19
2.12 Teoría de esquemas 23
2.12.1 Teorema de esquemas 24

2.12.2 Hipótesis de bloques constructivos 26

3 Modificaciones de un algoritmo genético 27
3.1 Algoritmo genético con inicialización canónica 27
3.1.1 Inicialización 27
3.2 Algoritmo genético con auto-adaptación 28
3.2.1 Componentes del individuo 28
3.2.2 Mutación 28
3.2.3 Recombinación 28
3.3 Algoritmo genético con detección de bloques constructivos 30
3.3.1 Componentes del individuo 30
3.3.2 Funcionamiento del algoritmo 30
3.3.3 Recombinación 31

4 Desarrollo 34
4.1 Análisis 34
4.1.1 Casos de uso 34
4.2 Diseño 36
4.2.1 Diagramas de clases 36
4.3 Implementación 43

5 Estudio experimental 47
5.1 Metodología 47
5.1.1 Definición de objetivos 47
5.1.2 Funciones de evaluación 48
5.1.3 Selección de parámetros 55
5.1.4 Modelo de validación 57
5.2 Resultados experimentales 59
5.2.1 Resultados de la bondad de las soluciones 59
5.2.2 Resultados del comportamiento de convergencia 71
5.3 Discusión de resultados 79
5.3.1 Observaciones generales 79
5.3.2 Observaciones de la bondad 79
5.3.3 Observaciones del comportamiento de convergencia 81

6 Conclusiones 82

A Manual de usuario 84
A.1 GUI 84
A.2 Línea de comandos 85

B Análisis económico global y temporal del proyecto 90

Bibliografía **92**

1 Introducción

La *optimización* se suele interpretar como una mejora de algún proceso o estado con respecto a uno o varios aspectos como, p. ej. , eficiencia, coste, efectividad, etc. Puede servir como método de búsqueda de alternativas para tomar una decisión mejor posible. Estas alternativas son soluciones factibles y pueden estar sujetas a ciertas restricciones. En el proceso de optimización se trata de determinar una solución satisfactoriamente buena con respecto a una *función de objetivo*, p. ej. , una función que depende de unas variables que se han de determinar para maximizarla. En general, un método de optimización es un algoritmo que es capaz de encontrar el óptimo de un sistema a partir de una configuración inicial. Para resolver problemas de optimización existen varios métodos que en el jerga de la informática se llaman *algoritmos de optimización*, entre los cuales aparecen los *algoritmos evolutivos*. Los *algoritmos genéticos* son una variante de los algoritmos evolutivos y se basan en la analogía biológica de selección natural, es decir, en el principio de Darwin de que sobreviven los individuos que están mejor adaptados dentro de una población. En los años 60 John Holland tuvo la idea de pasar del contexto natural de evolución al contexto de algoritmos genéticos. Interpretó los individuos representados por cromosomas como las soluciones representadas por una cadena binaria, es decir, una secuencia de valores binarios ($\{0, 1\}$) que codifica los elementos del problema. El entorno que determina la aptitud de supervivencia de individuos es una función heurística que indica la bondad de las soluciones - las mejores pasan a la siguiente generación. También se aplican un cruce y una mutación a las soluciones codificadas. A partir de esta idea han aparecido algoritmos genéticos construidos con funcionalidades y parámetros muy diversos.

1.1 Motivación

Según el teorema de *no-free-lunch* [WM95], no existe ningún algoritmo que resuelve todos los problemas de optimización siendo siempre superior a cualquier otro. Más bien tiene sentido averiguar bajo qué circunstancias un algoritmo evolutivo se comporta mejor que otros métodos al solucionar una clase determinada de problemas. Así métodos tradicionales son apropiados por su eficiencia para, p. ej. , problemas lineales, cuadráticos, unimodales, etc., pero no son satisfactorios para la gran mayoría de problemas, que pueden ser discontinuos, no diferenciables, multimodales, ruidosos. . . Estos se resuelven mejor con los robustos algoritmos evolutivos. Aplicaciones típicas son los problemas clásicos de optimización (p. ej. *maximum independent set*), planificación (p. ej. rutas, *schedu-*

ling), control (p. ej. diseño de sistemas de control, controlador de vehículos autónomos), *packing* (p. ej. diseño de circuitos), reconocimiento de patrones, modelos de simulación, etc.

Los algoritmos evolutivos también son aplicables a la optimización temporal, donde el entorno del problema no es estático. Dado que se basan en el "*feedback*" del entorno, la adaptación es muy robusta a cambios.

Muchos aspectos de la evolución natural todavía no se conocen o no se simulan. Existe un gran potencial para descubrir nuevas funcionalidades y fenómenos. Aparte de eso no sólo se trata de aplicar conocimientos biológicos, sino que también se puede intentar descubrir variantes útiles a partir de ideas prometedoras, como es el caso en este proyecto, donde se presentan diferentes modificaciones de un algoritmo genético básico debido a ideas que se basan en la intención de mejorar su rendimiento. Un *esquema* es un patrón con el cual se representa un subconjunto de una parte de la codificación de soluciones. Así en una cadena binaria se fijan algunas posiciones y otras se dejan libres. Cuando la evolución está en plena actividad, en los individuos de la población aparecen y perpetúan los esquemas que contienen soluciones buenas. Estos esquemas se combinan para formar cadenas cada vez mejores y se llaman *bloques constructivos*. Según Holland, esto ocurre de manera implícita. La idea fundamental es modificar el algoritmo para aprovechar estos bloques explícitamente. Para ello se incluirán las tres modificaciones siguientes:

- En un algoritmo genético básico se suelen inicializar las cadenas con un valor aleatorio. La primera modificación consiste en incorporar una inicialización especial llamada *canónica* de las cadenas binarias de los individuos de la población.
- Los cruces entre individuos, los puntos de cruces y las mutaciones en sus cadenas se suelen hacer con cierta probabilidad según unos parámetros previamente fijados. Muchas veces resulta difícil encontrar parámetros adecuados, e incluso puede ser muy razonable cambiar sus valores durante el proceso de optimización. La segunda modificación consiste en incorporar unos parámetros en la codificación de los individuos para dejarlos participar en la evolución, es decir, posibilitar la *auto-adaptación* de ciertos parámetros.
- La tercera modificación consiste en la detección y el intercambio de posibles bloques constructivos. Cada cierto número de generaciones se aplica una estrategia para detectar en las cadenas de los individuos unos puntos de cruce que posiblemente delimitan bloques constructivos. De esta manera los individuos podrán centrarse en intercambiar explícitamente bloques enteros prometedores con los demás sin desbaratar tanto los bloques en cruces poco razonables.

1.2 Objetivos del proyecto

El proyecto tiene el objetivo de estudiar experimentalmente el efecto de las modificaciones mencionadas. Para conseguirlo, primero se diseña un algoritmo genético genérico con posibilidad de adaptarlo a un problema específico (función de objetivo a optimizar) e incorporar fácilmente diferentes características que pueden tener estos algoritmos. Como referencia de comparación se diseña y se implementa un *algoritmo genético estándar* (véase 2.11.4). Luego se diseñan y se implementan los algoritmos con las modificaciones. Se prueban los algoritmos con unas funciones de evaluación típicas (*test problems*) para comprobar su funcionamiento y examinar los efectos que surgen a causa de las modificaciones. Interesa evaluar empíricamente el rendimiento de los algoritmos modificados comparándolos con un algoritmo genético estándar. La hipótesis del proyecto es que las modificaciones pueden favorecer el rendimiento en algunos casos a determinar, ya que un algoritmo con rendimiento superior general no existe según el teorema de *no-free-lunch*.

1.3 Organización de la memoria

Esta memoria está estructurada de la manera siguiente:

El capítulo 2 da una descripción general de algoritmos evolutivos. Luego se presenta un algoritmo genético estándar: su funcionamiento, sus parámetros y componentes. También se da una explicación teórica sobre algoritmos genéticos y bloques constructivos.

En capítulo 3 se explican con más detalle las modificaciones y diferencias que se estudian y realizan a partir de unas ideas para mejorar el algoritmo básico.

El capítulo 4 presenta la especificación, el análisis, el diseño e información importante sobre la implementación de los algoritmos que se utilizan en el estudio.

El capítulo 5 explica en qué consiste el estudio experimental para comparar cada algoritmo modificado con el algoritmo genético estándar y muestra los resultados de los experimentos. Consecuentemente se analizan los resultados.

En capítulo 6 se mencionan las conclusiones más importantes de este proyecto y posibilidades para un trabajo futuro.

2 Conocimientos básicos y conceptos fundamentales

Definición 2.1 (Algoritmo evolutivo)
Un *algoritmo evolutivo* (EA) es un procedimiento probabilista y adaptativo de búsqueda (parcialmente paralela[1]) el comportamiento del cual imita algunos aspectos de la evolución biológica:

- Variación de individuos
- Selección
- Almacenamiento de información hereditaria

Durante muchas generaciones las poblaciones de la naturaleza evolucionan según el principio de la evolución natural y la supervivencia del mejor adaptado (según Charles Darwin). Imitando estos principios naturales, los algoritmos evolutivos pueden generar poblaciones artificiales y someterlas a una evolución que tiene como objetivo solucionar un problema dado.

Los algoritmos evolutivos operan sobre un *espacio de búsqueda* S que puede ser cualquier conjunto. Esto hace posible aplicar estos algoritmos a casi cualquier estructura de espacio que no está limitado por suposiciones restrictivas y que puede albergar varios óptimos locales. En la investigación se utilizan con mayor frecuencia los conjuntos $S = \mathbb{B}^n = \{0,1\}^n$, $S = \mathbb{R}^n$ y $S = S_n = \{\pi \mid \pi \text{ es permutación sobre } \{1,\ldots,n\}\}$. En estos tres conjuntos típicos los elementos tienen un número fijo de n componentes. Pero también puede tener mucho sentido definir un espacio de búsqueda con elementos de "longitud" variable, como p. ej. árboles para describir programas (2.11.3).

Según [Bäc96, p. 33], un algoritmo evolutivo se clasifica como una técnica de *aprendizaje no supervisado*, o sea de aprendizaje inductivo por observación y descubrimiento, ya que no existe algo que supervise el algoritmo con ejemplos que proporcionen conocimientos, sino que el algoritmo mismo elige los ejemplos, y la búsqueda procede de inducciones basadas en conocimiento adquirido propiamente.

[1] véase "paralelismo implícito" en sección 2.12.2

2.1 Funcionamiento de un algoritmo evolutivo

En un algoritmo evolutivo existe una *función de objetivo* g, que evalúa los elementos de un *espacio de búsqueda* S. El objetivo del algoritmo es encontrar un punto en S para minimizar o maximizar g según sea el caso. A partir del objetivo se define una *función de fitness* f que valora la bondad de los puntos, dando valores más altos a aquellos que estén relativamente más cerca al objetivo al aplicar g. El algoritmo administra un conjunto de *individuos* que se denomina *población*. Cada individuo representa un elemento en S. En la *inicialización* se suele asignar cualquier población que, p. ej. , contiene individuos creados con valores aleatorios. A continuación el algoritmo se desarrolla en ciclos iguales denominados *generaciones*, hasta que se cumpla un *criterio de terminación*.

Al principio de un ciclo se evalúan los individuos de la población, es decir, se aplica la función de objetivo a los puntos del espacio de búsqueda que están representados por los individuos; a estos valores luego se le aplica la función de *fitness* para finalmente obtener los valores de *fitness*, que se pueden interpretar como una medida para la bondad de una solución. Luego se seleccionan individuos progenitores que sirven como base para crear nuevos individuos descendientes. El mecanismo de *selección* puede incluir cierta aleatoriedad, no obstante se prefieren los individuos prometedores, es decir, la probabilidad de seleccionar un individuo como progenitor aumenta con su valor de *fitness*. Este mecanismo se denomina *selección para reproducción*. Después se crean los descendientes a partir de los progenitores. Estos se modifican ligeramente en un proceso de *variación* que consiste en unos operadores clasificados según la cantidad de progenitores que intervienen. El operador unario se llama operador de *mutación*: se crea un descendiente a partir de un progenitor. El operador n-ario se llama operador de *recombinación*: se crea uno o varios descendientes a partir de varios progenitores. Se suele combinar estos operadores de manera que se aplica la recombinación y luego a sus descendientes la mutación. Después de la variación se evalúan los individuos descendientes. Normalmente el tamaño de la población se deja fijo para la próxima generación, así que ahora se debe seleccionar los individuos que sobreviven (resp. los que se rechazan). Esta selección se denomina *selección para reemplazamiento*. Finalmente se verifica si se cumple un *criterio de terminación*. De ser así el algoritmo termina, de lo contrario se vuelve a empezar con la nueva generación por la selección para reproducción. En figura 2.1 se representa este procedimiento.

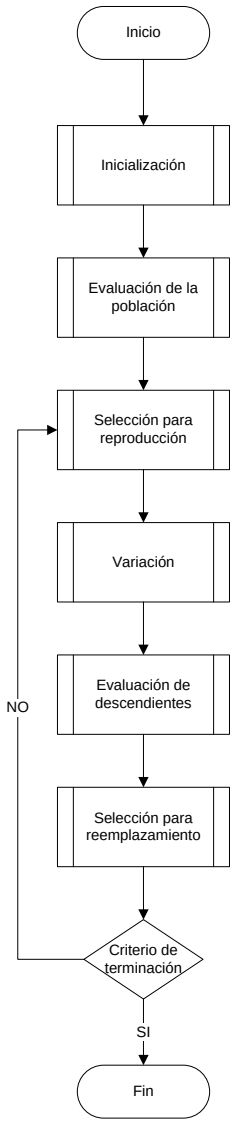

Figura 2.1: Funcionamiento general de un algoritmo evolutivo

Un algoritmo evolutivo está compuesto por los componentes siguientes:

- Población de individuos
- Inicialización
- Evaluación
- Selección
- Operadores de variación
  - Recombinación
  - Mutación
- Criterio de terminación

2.2 Población de individuos

Definición 2.2 (Individuo y población)
Sean A_x y A_s cualquier conjunto, e $I = A_x \times A_s$. Un *individuo* $a_k \in I$ contiene un conjunto de *parámetros de objeto* $x_k \in A_x$, que también se suele denominar *cromosoma*, y eventualmente *parámetros endógenos de estrategia* $y_k \in A_s$. El conjunto I es el *espacio de individuos* $a_k = (x_k, y_k) \in I$. Una *población* es cualquier subconjunto finito $P \subseteq I$.

Los parámetros de objeto representan la posición del individuo en el espacio de búsqueda S. En primer lugar a estos parámetros se aplica la función de objetivo. Luego se calcula su *fitness* con la función de *fitness*. Es posible que los parámetros de objeto de los individuos estén codificados de manera que A_x pueda representar "demasiados" elementos que estén fuera del espacio de búsqueda S. Esto ocurre típicamente cuando la función de objetivo incluye restricciones que no se tienen en cuenta o no se pueden considerar en la codificación y significa que $S \subseteq A_x$. Estos elementos no presentan ningún problema grave, como se verá en 2.4, mientras que la gran mayoría de los parámetros de objeto pertenezcan a S. En estos casos se suelen penalizar los individuos por la función de *fitness*.

Los parámetros endógenos de estrategia determinan el desarrollo y la modificación del individuo. Desempeñan su papel cuando se muten y/o reproduzcan los individuos.

Además existen *parámetros exógenos de estrategia*. Estos suelen ser valores globales del algoritmo que permanecen constantes.

2.3 Inicialización

En la *inicialización* se suele dar valores aleatorios a los componentes de los individuos. A veces puede tener sentido inicializar una población con individuos prometedores, pero si sufre la diversidad de los individuos, el algoritmo puede funcionar mal. Una inicialización determinada también puede ser útil para observar unos efectos intencionados. En el capítulo 3 se presenta una inicialización determinada.

2.4 Evaluación

Se distingue entre la función de *objetivo* y la función de *fitness*. La función de objetivo es una función $g : A_x \to \mathbb{R}$ que devuelve un valor a partir de los parámetros de objeto. En el problema de optimización se trata de encontrar los parámetros de objeto que minimicen o maximicen esta función. Realmente sólo se necesita como una caja negra que nos devuelve un valor real, no interesa su aspecto interior. A pesar de esto hace falta interpretar la *bondad* de estos parámetros de objeto con respeto al objetivo, ya que depende de si el objetivo es maximizar o minimizar la función de objetivo o si hay restricciones, penalizaciones, etc. Incluso puede interesar definir la bondad con relación a los demás individuos de la población. Consecuentemente se debería mapear el resultado de la función de objetivo a un valor de bondad. La función de *fitness* es una función $f : A_x \to \mathbb{R}^+$ que tiene en cuenta este mapeo: Mejores individuos obtienen valores más altos de *fitness*.

2.4.1 Mapeo

El mapeo es necesario para convertir el valor de la función de evaluación al *fitness*, un valor positivo.

Mapeo para minimización

Si se quiere minimizar una función, entonces el *fitness* debe aumentar cuando el valor de la evaluación disminuye y vice versa. Sea g la función de evaluación y f la función de *fitness*, entonces esto se consigue con

$$f(x) = \begin{cases} c_{max} - g(x) & \text{para } g(x) < c_{max} \\ 0 & \text{para } g(x) \geq c_{max} \end{cases} \tag{2.4.1}$$

c_{max} puede ser el valor de evaluación máximo de los individuos de la generación actual o de las últimas k generaciones.

Mapeo para *fitness* no negativo

Para que el *fitness* obtenga sólo valores positivos, hay que desplazar la función de evaluación hacia valores positivos:

$$f(x) = \begin{cases} g(x) + c_{min} & \text{para } g(x) + c_{min} > 0 \\ 0 & \text{para } g(x) + c_{min} \leq 0 \end{cases} \tag{2.4.2}$$

c_{min} es el desplazamiento que se puede calcular análogamente a c_{max}.

2.4.2 Restricciones

Cuando un individuo no cumpla todas las restricciones de la función de objetivo, tendríamos una solución no factible que no pertenece al espacio de búsqueda S. De una manera indirecta se puede rechazar a este individuo "castigándolo" con un valor de *fitness* muy bajo. También hay que tener en cuenta que entre las soluciones no factibles hay diferencias que pueden ser importantes, p. ej. , el número de restricciones no cumplidas. Un problema de optimización normalizado con restricciones $h_i(x)$

$$\begin{cases} \text{mín } g(x) \\ \text{sujeto a } h_i(x) \geq 0 \end{cases}$$

podemos convertirlo a un método de castigo:

$$\text{mín } g(x) + r \cdot \sum_i \Phi[h_i(x)], \tag{2.4.3}$$

donde r es el *coeficiente de castigo* y Φ es la *función de castigo*, p. ej. $\Phi[h] = h^2$.

2.5 Selección

La *selección* mapea un conjunto de n individuos a un subconjunto suyo (de tamaño $m \leq n$)

$$f_{sel} : I^n \to I^m,$$

$$m \leq n.$$

Se distingue dos usos de selección:

- Selección para reproducción
- Selección para reemplazamiento

En la *selección para reproducción* se seleccionan los individuos (buenos) como progenitores para reproducir descendientes.

En la *selección para reemplazamiento* se seleccionan los individuos (buenos) que sobreviven. Cuando se reproduzcan nuevos individuos, la población crece. Para que el tamaño de la población vuelva a sus límites, lo reducimos seleccionando los individuos que sobreviven.

En cualquier selección se prefieren los individuos con un *fitness* alto, es decir, se restringen pero se conducen los individuos hacia características más prometedoras en el espacio de búsqueda. Existen métodos de selección que dependen también del mismo individuo y no solamente de su *fitness*. P. ej. , puede interesar que en una población no se deben acumular individuos que se parezcan mucho entre sí, sino que se mantenga cierta diversidad entre ellos. Entonces en la selección se consideraría la similitud entre individuos mediante una *función de distancia*. Al realizar la selección se podría utilizar esta función para decidir la supervivencia o posibilidad de reproducción de individuos, consiguiendo así que individuos con mayor diversidad entre sí se reproduzcan más a menudo para contribuir a dar pasos más diversos en el espacio de búsqueda.

Algunas de las operaciones más importantes y corrientes de selección de individuos se presentan a continuación; muchas de ellas se pueden utilizar tanto como selección para reproducción como para reemplazamiento.

2.5.1 Selección uniforme

Se selecciona un individuo con distribución uniforme.

2.5.2 Selección proporcional al *fitness*

Se asume que los valores de la función de *fitness* f den resultados positivos. Entonces la probabilidad de seleccionar a un individuo a_k de la población P es

$$p(a_k) = \frac{f(a_k)}{\sum_{a_x \in P} f(a_x)}. \tag{2.5.1}$$

La selección se realiza mediante una "rueda de ruleta". Se le asigna a cada individuo una parte de la rueda de ruleta que tiene el tamaño proporcional al *fitness* en la población. Se gira la rueda y se selecciona el individuo indicado.

Este método puede tener algunas desventajas a causa de depender de manera muy directa de los valores de *fitness*. Si hiciéramos un cambio aditivo, es decir, si sumáramos un valor fijo a todos los valores de *fitness*, la selección ya tendría otro comportamiento aunque las diferencias de los *fitness* serían las mismas, cosa que no parece tener mucho sentido. Por un lado, grandes diferencias entre los valores *fitness* pueden provocar que

sólo se seleccione con probabilidad de casi 1 unos cuantos pocos individuos con valores altos de *fitness* que dominan la selección. Por otro lado, diferencias mínimas entre los valores *fitness* pueden resultar en una selección casi uniforme donde haría falta acentuar más estas diferencias. Por añadidura, estos dos casos suelen aparecer durante el proceso de evolución: Al principio de la evolución las diferencias de *fitness* serán muy grandes mientras que más adelante, cuando las soluciones hayan mejorado, el algoritmo puede converger ya que se desprecian demasiado las diferencias mínimas — pero todavía importantes — de *fitness*.

Para obtener valores más razonables de *fitness* aparecieron algunas variantes de la selección proporcional al *fitness*. En la *selección proporcional al* fitness *escalado* se pretende conseguir que las probabilidades de selección estén distribuidas razonablemente, escalando los valores de *fitness*. Existen varios mecanismos de escalamiento, la mayoría de ellos son probabilistas. Estos son los más corrientes:

Escalamiento lineal

En este mecanismo se escalan linealmente los valores de *fitness*. En general, la función de escalamiento de este mecanismo tiene la forma

$$f_{scl} = a \cdot f + b. \tag{2.5.2}$$

Los parámetros a y b se suelen elegir tal que se cumplan las siguientes propiedades:

- La probabilidad de seleccionar un individuo con valor medio de *fitness* es $\frac{1}{n}$ (n es el tamaño poblacional).
- La probabilidad de seleccionar el individuo con valor máximo de *fitness* es $\frac{c_{mult}}{n}$ (c_{mult} veces más alto que un individuo con *fitness* medio).

c_{mult} se denomina *factor de escalamiento* y suele ser un valor entre 1.2 y 2.
Consecuentemente podemos deducir las siguientes igualdades para la función de *fitness* escalado:

$$avg(f_{scl}) = avg(f)$$
$$max(f_{scl}) = c_{mult} \cdot avg(f),$$

donde avg da el valor medio de *fitness*, max el valor máximo de *fitness*. En figura 2.2 se muestra el escalamiento de los valores de la función de *fitness* en el eje horizontal a los valores escalados en el eje vertical.

Entonces las constantes a y b se obtienen fácilmente:

$$a = \frac{(c_{mult} - 1) \cdot avg(f)}{max(f) - avg(f)} \tag{2.5.3}$$

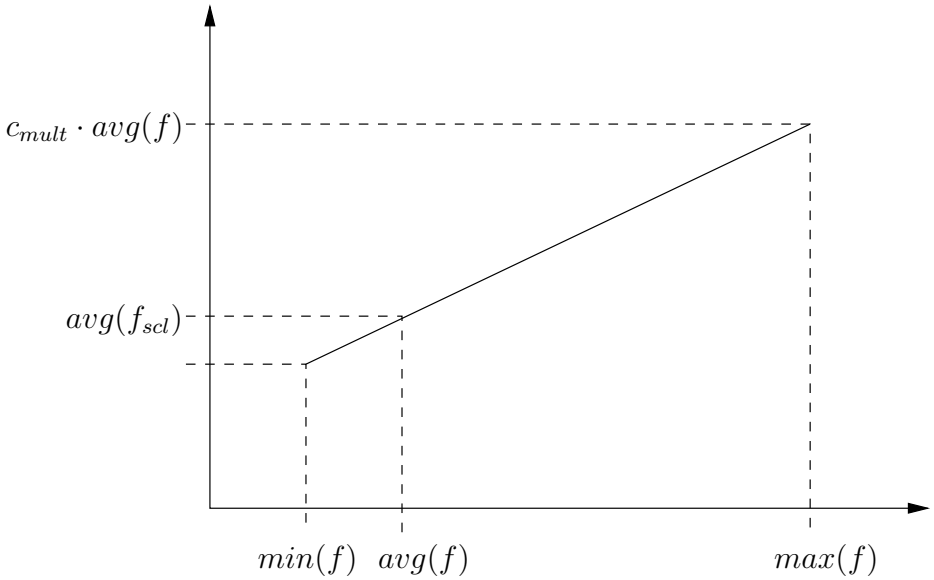

Figura 2.2: Escalamiento lineal

$$b = avg(f) \cdot \frac{max(f) - c_{mult} \cdot avg(f)}{max(f) - avg(f)} \tag{2.5.4}$$

Si se aplica el escalamiento de esta manera, puede ocurrir que $min(f_{scl}) = f_{scl}(min(f))$ tome valores negativos, que para una función de *fitness* no están admitidos. En estos casos se corrige la recta que ahora tendría un aspecto como en figura 2.3, tal como se muestra, para que $f_{scl}(min(f)) = 0$.

Ahora a y b obtienen los valores

$$a = \frac{avg(f)}{avg(f) - min(f)} \tag{2.5.5}$$

$$b = -\frac{min(f) \cdot avg(f)}{avg(f) - min(f)}. \tag{2.5.6}$$

Este método no resuelve todos los problemas mencionados de selección proporcional al *fitness*.

Sigma truncation

La función de escalamiento de este mecanismo es

$$f_{scl} = f - (avg(f) - c \cdot \sigma), \tag{2.5.7}$$

donde $c \in [1, 3]$ y σ es la desviación estándar de los valores de f aplicada a la población actual.

En los casos donde f_{scl} tome valores negativos, se asigna 0.

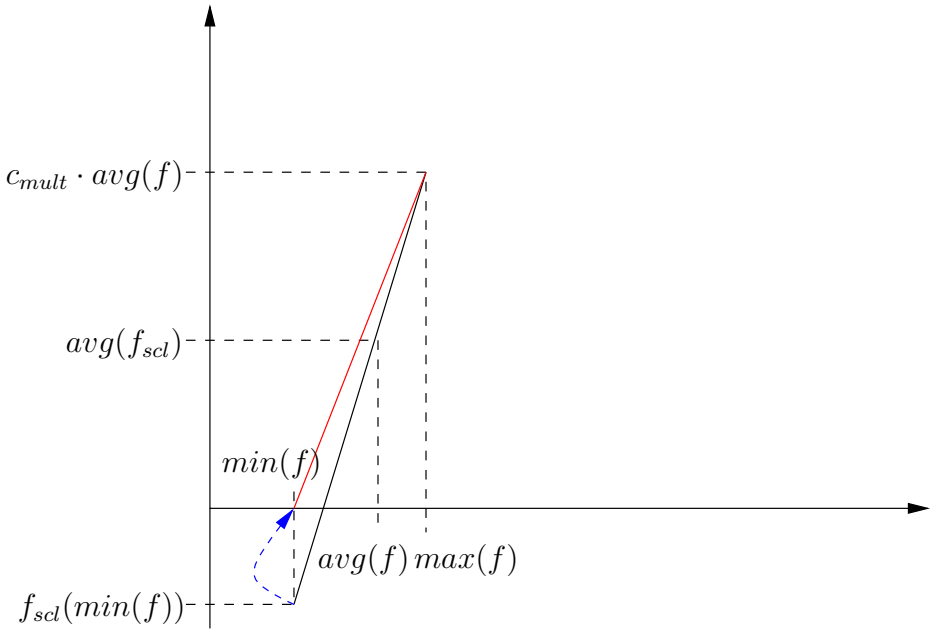

Figura 2.3: Escalamiento lineal corregido

Power law scaling

La función de escalamiento de este mecanismo es

$$f_{scl} = f^k, \tag{2.5.8}$$

donde k suele ser 1.005. Sin embargo este valor depende del problema de optimización.

Selección de Boltzmann

La función de escalamiento de este mecanismo es

$$f_{scl} = e^{\frac{f(x)}{T}}. \tag{2.5.9}$$

El parámetro T, que se denomina *temperatura*, permite regular la influencia de los valores de *fitness*. Este valor suele depender solamente del tiempo. Se empieza con una temperatura alta y se va disminuyendo (enfriando) poco a poco.

Este método tiene la ventaja de que los cambios aditivos de *fitness* no afectan la probabilidad de selección.

Selección por rango

En este método se ordena la población según el *fitness*. La posición del individuo es determinante para la probabilidad de selección. Puede ser una desventaja desasociar los valores de *fitness*, el tamaño de las diferencias entre los valores de *fitness* ya no influye.

2.5.3 Selección por torneo

El método consiste en seleccionar uniformemente k individuos de los cuales se selecciona aquel con *fitness* más alto. El valor de k es mayor que 1 y se denomina *tamaño de torneo*. Este operador de selección pertenece a los operadores *locales* de selección ya que sólo hace falta conocer los valores de *fitness* de un conjunto local de individuos. Cuanto más alto sea el valor de k, más importancia tendrán los valores de *fitness*. Jamás se seleccionará el individuo peor.

2.5.4 *Remainder stochastic sampling without replacement*

El número esperado de selecciones de un individuo particular es

$$\text{número total de selecciones} \cdot \frac{\text{fitness del individuo}}{\text{suma de fitness de población}}.$$

Primero se seleccionan tantos individuos según la parte entera del valor esperado. El resto (hasta el número total de selecciones) se selecciona según las partes fraccionales por el método de la rueda de ruleta (véase 2.5.2).

Los siguientes métodos se suelen utilizar en la selección para reemplazamiento. Se seleccionan varios individuos a la vez.

2.5.5 Selección por corte

Con este método no se seleccionan los individuos uno por uno, tal como se hizo en los métodos anteriores, sino que se seleccionan los k individuos con mayor *fitness*, donde k es un número entre 1 y el tamaño poblacional.

2.5.6 Selección plus

De los μ individuos, que formaban la población vieja, y los λ individuos descendientes se seleccionan (a la vez) los μ individuos con mayor *fitness*; los λ peores individuos se reemplazan. Se suele usar la notación selección $(\mu+\lambda)$. A veces también se incluye una ρ a la notación, indicando el número de progenitores que participan en cada recombinación.

2.5.7 Selección coma

De los λ individuos descendientes se seleccionan los μ individuos con mayor *fitness*; los μ individuos de la población vieja y los $\lambda - \mu$ individuos descendientes con menor *fitness* se reemplazan. Se suele usar la notación selección (μ, λ).

Los siguientes métodos de selección para reemplazamiento se combinan con otros. Tratan de incorporar una característica adicional al funcionamiento de cualquier otra selección.

2.5.8 Elitismo

Definición 2.3 (Elitismo)
En una selección de reemplazamiento con *elitismo* k sobreviven los mejores k individuos.

Una manera muy típica de realizar el método de reemplazamiento elitista consiste en substituir los k individuos peores de la población actual por los k mejores de la población anterior.

2.5.9 *Generation gap*

Definición 2.4 (*Generation gap*)
El *generation gap* G es el porcentaje de la población a reemplazar. Si $G < 100\,\%$, se habla de una *población solapada*.

Para hacer el reemplazamiento se selecciona aleatoriamente un $G\,\%$ de la población para permitir solamente a estos individuos participar en la selección para reproducción y luego ser reemplazados en la selección para reemplazamiento. El resto de los individuos sobrevive.

2.6 Recombinación

La *recombinación* o *cruce* mapea un conjunto de individuos a un conjunto de individuos

$$f_{rec} : I^{\rho} \to I^{m}.$$

El cromosoma de un descendiente se compone a partir de los cromosomas de los ρ progenitores. En general los ρ progenitores pueden crear más de un descendiente. Tradicionalmente se distingue entre

- *Recombinación discreta*, donde se decide por cada componente de quién de los progenitores se copia la componente
- *Recombinación intermedia o aritmética*, donde cada componente se calcula como valor medio a partir de la componente correspondiente de todos los progenitores.

Existen multitud de variantes de recombinación para diferentes espacios de búsqueda. En la sección 2.11.4 se presentarán los típicos para algoritmos genéticos que operan en espacios $S = \mathbb{B}^n$.

Para recombinaciones de permutaciones hay que tener en cuenta lo que provoca una recombinación en el problema particular al generar un descendiente. Por eso han aparecido muchos mecanismos para estos casos. Los más corrientes son el *partially matched crossover*, el *order crossover* y el *cycle crossover*. Se explican en detalle en [Gol89], pero para este proyecto no se consideran.

En $\mathbb{R}^n$ los operadores corrientes son el *multi-point crossover*, el *uniform crossover* y la *recombinación aritmética*. Los dos primeros se explican en sección 2.11.4 para el espacio $\mathbb{B}^n$, funcionan de manera análoga para $\mathbb{R}^n$. La recombinación aritmética se explica en sección 2.11.1.

2.7 Mutación

La *mutación* es un mapeo en el espacio de individuos

$$f_{mut} : I \to I.$$

El individuo descendiente de esta operación suele distinguirse muy poco del progenitor.

Para problemas en S_n, donde se buscan permutaciones óptimas, el operador depende mucho del problema en sí. Es decir, a veces hay que incluir un poco de conocimiento en los operadores de variación, en el sentido de que se busca una estrategia de variación adecuada (con cambios razonables para el problema). Mutaciones típicas para S_n son intercambios de componentes del cromosoma o saltos donde se desplazan partes del cromosoma, o una combinación de las dos.

Los operadores típicos para el espacio $S = \mathbb{B}^n$ y $S = \mathbb{R}^n$ se presentarán en sección 2.11.4 y sección 2.11.1, respectivamente.

2.8 Criterio de terminación

La terminación es una función booleana

$$f_{ter} : I^\rho \to \{verdadero, falso\}.$$

Es muy corriente especificar un número máximo de generaciones para terminar el algoritmo (véase [Bäc96, p. 100]).

A veces es conveniente terminar el algoritmo cuando ya no haya diversidad de cromosomas o cuando casi no hay mejora y pasen muchas generaciones.

2.9 Ajuste de parámetros

Los algoritmos evolutivos pueden tener muchísimos parámetros que hay que ajustar al problema dado. Hay diferentes maneras de cómo ajustarlos. A veces el ajuste puede significar una optimización de por sí.

La manera más simple y habitual es el *ajuste estático* de parámetros, donde éstos se fijan al principio del algoritmo y ya no se cambian. Esto puede ser bastante perjudicial, ya que en el transcurso de la optimización pueden no ser adecuados.

Con el *ajuste dinámico* se trata de mantenerlos ajustados durante el transcurso, asignándoles los valores en función del tiempo, como p. ej. el número de generación.

En el *ajuste adaptativo* se va aún más allá: Aquí los valores de parámetros podrían resultar de una función que depende del proceso de búsqueda mismo, como una especie de "*feedback*". Todos los individuos hasta la fecha, las evaluaciones y cualquier experimento realizado previamente pueden influir en el valor.

La *auto-adaptación* es un caso especial de ajuste adaptativo. Los parámetros forman parte del cromosoma codificado de los individuos, es decir, con este método se amplía el espacio de búsqueda. Al mismo tiempo que los parámetros de objeto, evolucionan los parámetros endógenos de estrategia (véase 2.2) que también estarán sometidos a los operadores de variación.

2.10 Medidas de rendimiento

Las siguientes funciones son muy corrientes para medir el rendimiento de un algoritmo evolutivo.

Definición 2.5 (*Offline performance*)

$$x^*(T) = \frac{1}{T}\sum_{t=1}^{T} \text{mejor } \{\text{mejor } \{g(x_k)\}_{a_k=(x_k,y_k)\in P(i)}\}_{i=1..t},$$

T es la generación actual, $P(i)$ es la población en la i-ésima generación y la función "mejor" es el mínimo o el máximo según el objetivo.

Es decir, se toma la media de las T mejores evaluaciones que aparecieron hasta cada generación t.

Definición 2.6 (*Online performance*)

$$\bar{x}\ (T) = \frac{1}{T}\sum_{t=1}^{T} \text{mejor}\ \{g(x_k)\}_{a_k=(x_k,y_k)\in P(t)},$$

es decir, la media de las T mejores evaluaciones que aparecieron en cada generación t.

2.11 Familias de algoritmos evolutivos

Todas las ramas de algoritmos evolutivos tienen en común que imitan el proceso de la evolución natural. Antes de haber sido reunidas en 1990 bajo el concepto de **algoritmos evolutivos** había diferentes principios y direcciones en investigaciones independientes entre sí. Se da una breve explicación de las direcciones principales.

2.11.1 Estrategias de evolución (ES)

(Hans-Paul Schwefel [Sch77], Ingo Rechenberg [Rec73])
El espacio de búsqueda suele ser $\mathbb{R}^n$. Consecuentemente no hace falta codificar un individuo si el espacio es continuo, pero tampoco se pueden representar individuos de problemas que necesitan otros espacios de búsqueda que $\mathbb{R}^n$. Se utiliza selección plus o selección coma (véase 2.5.6 y 2.5.7), que se basan en el rango del *fitness*.

Recombinación en un ES

La recombinación no se utiliza o solamente se aplica recombinación intermedia, que funciona de la manera siguiente:

Si $x_1, \ldots, x_\mu$ son los cromosomas progenitores seleccionados para el cruce, entonces el descendiente es

$$y = a_1 \cdot x_1 + \cdots + a_\mu \cdot x_\mu,$$

con $a_1 + \cdots + a_\mu = 1$. En la recombinación intermedia los pesos a_i suelen tener el valor $\frac{1}{\mu}$.

Mutación en un ES

Este es el operador esencial. El cromosoma del descendiente típicamente resulta sumando un vector $v_{mut} \in \mathbb{R}^n$ al cromosoma del progenitor. Los componentes de v_{mut} pueden ser diferentes, aunque es más habitual que todos tengan un mismo valor $m \in \mathbb{R}$. Una idea simple de dar valor a m sería asignarle un valor aleatorio uniformemente de un intervalo limitado $[-c; c]$, $c \in \mathbb{R}^+$. Más razonable y corriente es darle un valor aleatorio con esperanza 0, procurando al mismo tiempo que la probabilidad para valores lejanos de

0 disminuya. Mediante una distribución normal se consigue que cambios grandes sean menos probables que cambios pequeños:

$$p(m = r) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-r^2}{2\sigma^2}}$$

σ es la desviación estándar la cual permite manipular la amplitud de m. Esta propiedad se aprovecha durante el procedimiento para asignar a σ dinámicamente un valor adaptado, es decir, cuanto más lejos esté la posición del óptimo, más grande será el valor de σ para acercarse con pasos más grandes al óptimo, mientras que si ya se encuentra cerca del óptimo, σ sólo permitirá dar pasos pequeños para buscar el óptimo exacto. El valor de σ se denomina *distancia de paso*. Alternativamente, se puede dejar evolucionar la distancia de paso como parámetro de estrategia (véase 2.2). Así el proceso evolutivo se gobierna mediante la auto-adaptación (véase 3.2). A veces también se añade una componente de correlaciones entre los individuos.

2.11.2 Programación evolutiva (EP)

(Larry Fogel [FOW66])
Aunque no existe una definición fija para la mutación y la representación, la idea inicial fue la optimización de algoritmos con individuos que corresponden a estados de un autómata finito. Están compuestos por un valor del estado actual, un símbolo del alfabeto y un nuevo estado al que se nos lleva en caso de coincidir con el valor del símbolo actual. No se utiliza la recombinación ya que los individuos se consideran especies que no se pueden cruzar entre ellos. Se utiliza selección plus $(\mu + \mu)$ (véase 2.5.6) o por torneo (véase 2.5.3).

2.11.3 Programación genética (GP)

(John Koza [Koz92])
Como base se utiliza un algoritmo genético (véase 2.11.4). A diferencia de éste, el espacio de búsqueda es un conjunto de árboles que corresponde a expresiones lógicas o programas.

2.11.4 Algoritmos genéticos (GA)

(John Holland [Hol75])
Los algoritmos genéticos operan normalmente sobre el espacio de búsqueda $\mathbb{B}^n$. Los componentes son bits que imitan los genes naturales, aunque estos suelen utilizar un alfabeto más amplio. El cromosoma también se denomina *bitstring*. La configuración típica de un algoritmo genético es la siguiente:

- *Inicialización*: Valor aleatorio de los bits

- *Selección para reproducción*: Selección proporcional al *fitness* (véase 2.5.2)
- *Selección para reemplazamiento*: Selección (μ, μ) (véase 2.5.7)
- *Recombinación*: *Multi-point crossover* con 1 o 2 puntos (véase 2.11.4)
- *Mutación*: *Standard bit mutation* (véase 2.11.4)
- *Criterio de terminación*: Número de generaciones

A éste se le llamará *algoritmo genético estándar* (SGA).

Recombinación en un GA

La recombinación es el operador de búsqueda esencial. Suelen participar 2 progenitores en el cruce. Las recombinaciones corrientes para GAs son

- *Multi-point crossover*
  Un parámetro determina la probabilidad de que se haga la recombinación (probabilidad de recombinación p_c). En caso de no hacerlo, simplemente se copian los progenitores. Se eligen k posiciones diferentes entre 1 y $n-1$ del *bitstring* (de longitud n) con aleatoriedad uniforme. Se utilizan las k posiciones de corte para componer el descendiente alternando de trozos de los progenitores. Normalmente el valor de k es 1 o 2. Sean $x^{(1)}, x^{(2)} \in \mathbb{B}^n$ los progenitores del cruce y p_i $(i = 1, \ldots, k)$ las posiciones ordenadas $p_1 < \ldots < p_k$. Entonces el descendiente es $x_1^{(1)} x_2^{(1)} \cdots x_{p_1}^{(1)} x_{p_1+1}^{(2)} \cdots x_{p_2}^{(2)} x_{p_2+1}^{(1)} \cdots$. Figura 2.4 muestra un ejemplo del cruce con $k = 2$.

- Uniform crossover
  Un parámetro determina la probabilidad de que se haga el cruce. En caso de no hacerlo, simplemente se copian los progenitores. Por cada bit del descendiente se elige aleatoriamente uno de los dos progenitores para copiar el bit correspondiente. Este método aumenta la diversidad de los descendientes.

Mutación en un GA

La mutación más bien sirve de operador en "*background*" para aumentar la diversidad de los cromosomas, consiguiendo así que el GA no converja hacia un óptimo local cuando los cromosomas se parezcan demasiado entre sí. Así se apreciaría otras soluciones posiblemente mejores al mutar bits.

Las mutaciones corrientes para GAs son

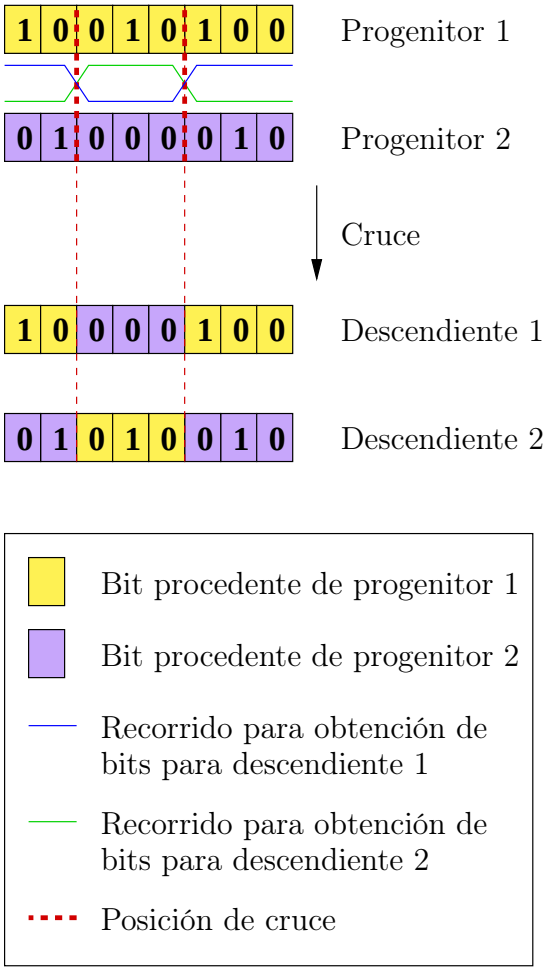

Figura 2.4: *Multi-point crossover* con $k = 2$

- *Standard bit mutation*
  Se genera un descendiente como copia del progenitor. Luego se recorre el *bitstring* y se invierte cada bit con una probabilidad $p_m \in (0; \frac{1}{2}]$. Valores típicos para p_m son 0.01, 0.001, $\frac{1}{n}$ ($n =$ "longitud" del *bitstring*).

- *k-bit-mutation*
  Se genera un descendiente como copia del progenitor. Luego se eligen k de sus bits con aleatoriedad uniforme para invertirlos. Normalmente el valor de k es 1. Con este método se consigue que el descendiente sea uno de los vecinos de la búsqueda con distancia de Hamming k (número de bits diferentes).

Codificación

Para codificar los valores de objetivo del problema de optimización se debe utilizar el alfabeto más corto que permite una expresión natural del problema ya que una cardinalidad baja del alfabeto maximiza la semejanza entre los cromosomas. Además con un alfabeto como el binario se maximiza el número de esquemas por cada bit (esquemas se explican en 2.12). Esta orientación se denomina *principio de alfabetos mínimos.*

Muchas veces es necesario decodificar números reales dentro del *bitstring* del GA. Si el rango del valor real es $[u, v]$ y disponemos de l bits para el valor, entonces la función de mapeo lineal f_{map} aplicado a los bits $(a_1, \ldots, a_l)$ da

$$f_{map}(a_1, \ldots, a_l) = u + \frac{v-u}{2^l - 1} \cdot \left(\sum_{j=0}^{l-1} a_{l-j} \cdot 2^j \right) \tag{2.11.1}$$

(véase [Bäc96, p. 109]).

El cociente $\frac{v-u}{2^l-1}$ se denomina *precisión.* Esto significa que realmente la búsqueda no se refiere a un valor continuo sino que se busca en una rejilla, es decir, en un espacio de búsqueda discreto. El mapeo es inyectivo pero no exhaustivo: No todos los puntos del espacio de búsqueda real son representables, pero se puede aumentar la exactitud de los resultados aumentando l. Siempre existe el peligro de perder una solución óptima a causa de la precisión, aunque las soluciones suelen ser aceptables.

Para codificar números enteros el método más conocido es la codificación binaria estándar. Sin embargo es preferible la codificación *Gray* ya que ésta tiene la ventaja de que números enteros adyacentes tienen una *distancia de Hamming* 1 (véase también [Bäc96, p. 110]).

Una distancia de Hamming 1 en cualquier posición no necesariamente significa que los enteros sean adyacentes, sino que también puede causar una diferencia grande entre los valores correspondientes decodificados.

Para decodificar en código Gray números reales, se calcula

$$f_{gray}(a_1, \ldots, a_l) = u + \frac{v-u}{2^l - 1} \cdot \left(\sum_{j=0}^{l-1} \left(\bigoplus_{k=1}^{l-j} a_k \right) \cdot 2^j \right) \tag{2.11.2}$$

donde $\oplus$ es la suma módulo 2 en $\mathbb{B}$.

2.12 Teoría de esquemas

Definición 2.7 (Esquema *H*)
Si A es el alfabeto de los símbolos de la codificación de los individuos, entonces $A \cup \{*\}$ es el alfabeto de los esquemas. Los símbolos de '$*$' pueden representar cualquier elemento de A. Consecuentemente, un determinado *esquema* H representa un subconjunto del espacio de todos los individuos.

Ejemplo: (para alfabeto binario)
Sea A el alfabeto binario $\{0, 1\}$, entonces el alfabeto de esquemas es $\{0, 1, *\}$. El esquema $H = [1 * 0*]$ contiene los cromosomas 1000, 1001, 1100 y 1101.

Definición 2.8 (Símbolos definidos)
Si $A \cup \{*\}$ es el alfabeto de un esquema H, entonces los *símbolos definidos* de H son los que pertenecen a A.

Definición 2.9 (Orden de esquema)
El *orden de esquema* $o(H)$ es el número de símbolos definidos en H.

Ejemplo:
$o([1 * 01 * *1*]) = 4$.

Definición 2.10 (Longitud definida de un esquema)
La *longitud definida de un esquema* $\delta(H)$ es la distancia entre los símbolos definidos más lejanos en H.

Ejemplo:
$\delta([1 * 01 * 10 * *0*]) = 9$.

Propiedades:
Para un cromosoma de longitud l con elementos de un alfabeto de cardinalidad k, existen $(k+1)^l$ esquemas de longitud l. Cada cromosoma de longitud l puede pertenecer a 2^l esquemas (cada posición de un esquema puede tener el valor del cromosoma o '$*$'). Una población de tamaño n tiene entre 2^l y $n \cdot 2^l$ esquemas.

2.12.1 Teorema de esquemas

El objetivo del *teorema de esquemas*, desarrollado por John Holland en 1975, es proporcionar un modelo formal para la eficacia de un SGA (con *1-point crossover*) observando lo que ocurre con un esquema.

Lema 2.1
Bajo selección proporcional al *fitness*, el número esperado de individuos que pertenecen al esquema H en la generación $t+1$ es

$$E[m(H,t+1)] = n \times p(a_k \in H) = \frac{m(H,t)f(H,t)}{\bar{f}(t)},$$

donde
$n \equiv$ tamaño poblacional,
$m(H,t) \equiv$ el número de instancias de H en la generación t,
$f(H,t) \equiv$ el *fitness* medio de los individuos que pertenecen a H en la población actual,
$\bar{f}(t) \equiv$ el *fitness* medio de la población en la generación t.

Demostración
Buscamos la probabilidad de que un individuo a_k seleccionado pertenezca al esquema H

$$p(a_k \in H).$$

Por un lado sabemos que la probabilidad de que un individuo cualquiera que pertenezca a H sea seleccionado, depende proporcionalmente del número de instancias que pertenecen a H en la población.

Por otro lado sabemos que la probabilidad de que un individuo cualquiera que pertenezca a H sea seleccionado, depende proporcionalmente de la relación entre el *fitness* medio de H y el *fitness* medio de la población.

Por lo tanto,

$$p(h \in H) = \frac{m(H,t)f(H,t)}{n\bar{f}(t)}$$

Para obtener el número esperado de nuevas instancias de H se multiplica por n. ■

Lema 2.2
Bajo *1-point-crossover* la probabilidad de supervivencia del esquema H en una generación t es

$$1 - p_c \frac{\delta(H)}{l-1} p_{dif}(H,t),$$

donde
$p_{dif}(H,t) \equiv$ probabilidad de que el segundo progenitor no pertenezca a H.

Demostración
Un esquema no sobrevive cuando el cruce provoque un desbarate en la zona relevante para la longitud definida del esquema. La probabilidad de que el cruce caiga dentro de esta zona es

$$p_c \frac{\delta(H)}{l-1}$$

Esto no es del todo correcto ya que puede ocurrir que el segundo progenitor "salve" el esquema. Por eso, para que no sobreviva el esquema, también hay que tener en cuenta la probabilidad de que el segundo progenitor no pertenece al esquema:

$$p_c \frac{\delta(H)}{l-1} p_{dif}(H,t)$$

La probabilidad de que H sobreviva es 1 – la probabilidad de que H no sobrevive.

■

Lema 2.3
Bajo *standard bit mutation* la probabilidad de que un esquema H de orden $o(H)$ sobreviva una generación t es

$$1 - o(H)p_m$$

Demostración
Para que sobreviva un esquema, todos sus símbolos fijos (los que no son '$*$') no se pueden mutar (invertir). El número de símbolos fijos es $o(H)$. La probabilidad de no mutar un símbolo de una posición del cromosoma es $1 - p_m$. Consecuentemente, la probabilidad de no mutar a ningúno de los símbolos fijos es

$$(1 - p_m)^{o(H)}.$$

En GAs la probabilidad de mutación suele ser muy pequeña, $p_m \ll 1$.
Por lo tanto,

$$(1 - p_m)^{o(H)} \approx 1 - o(H)p_m.$$

■

Si aplicamos los tres lemas, obtenemos una cota inferior para el número esperado de individuos de H en generación $t+1$ a partir de t:

Lema 2.4
Teorema de esquemas:

$$E[m(H,t+1)] \geq \frac{m(H,t)f(H,t)}{\bar{f}(t)} \left(1 - p_c \frac{\delta(H)}{l-1} p_{dif}(H,t) - o(H)p_m\right) \qquad (2.12.1)$$

Para simplificar se parte de $p_{dif} = 1$ (en el peor de los casos se causa mayor desbarate), por eso el teorema se convierte en una desigualdad.

Existen varias críticas sobre este teorema entre las cuales aparecen las siguientes:
A parte de "sólo" ser una desigualdad, el *fitness* de un hiperplano (o esquema) del espacio de búsqueda solamente se puede obtener por valores ya observados que no necesariamente se parecen al *fitness* real del esquema. Y el hecho de que durante el procedimiento de evolución los esquemas también están cada vez más sesgados hace que todavía aumente la inexactitud. Además en el teorema sólo se observa un esquema que no depende de lo que hacen los demás esquemas, quizás habrá algunos que se favorecen más. El teorema sólo considera las influencias negativas de mutación y cruce, pero no "cuenta" nada sobre los efectos positivos de la recombinación.
A pesar de las críticas el teorema en el mundo científico frecuentemente sirve de base teórica para el análisis a nivel de esquemas en vez de contemplar individuos sueltos.

2.12.2 Hipótesis de bloques constructivos

Definición 2.11 (Bloque constructivo)
Un *bloque constructivo* es un esquema corto (de longitud definida corta), de orden pequeño y con *fitness* alto, es decir, con un *fitness* al menos superior al *fitness* medio.

A partir del teorema de esquemas se observa que los bloques constructivos son los favorecidos por un SGA, lo que provoca que se combinen para formar cadenas buenas. Esta declaración se denomina *hipótesis de bloques constructivos*. Un bloque constructivo se considera como una solución a un subproblema del problema global de optimización. Este primero se subdivide en unos subproblemas que son más simples de tratar, y que luego se combinan siendo cada vez más complicados hasta encontrar una solución.
También existen críticas sobre esta hipótesis, véase [Gol89, "deceptive problems"].

Cuando evaluemos un individuo, estamos evaluando simultáneamente varios esquemas, o sea estamos paralelizando la búsqueda en el espacio. Este proceso se denomina *paralelismo implícito*. Aunque una población tenga μ individuos el número de esquemas procesados en cada generación es de orden $O(\mu^3)$ ([Gol89, p. 40-41]).

3 Modificaciones de un algoritmo genético

Se presentan tres variantes de un algoritmo genético a base de unas ideas que tienen en común aprovechar bloques constructivos con el objetivo de mejorar el rendimiento, aunque sea sólo en casos determinados, según el teorema de *no-free-lunch*. Las estrategias de las modificaciones tienen como intención facilitar la extensión y combinación de los bloques constructivos. Como base para hacer las modificaciones sirve el algoritmo genético estándar. En los próximos apartados se explican las ideas y los cambios realizados con respecto al SGA.

3.1 Algoritmo genético con inicialización canónica

La estrategia de esta modificación es la de mejorar la inicialización como punto de partida para el desarrollo de los individuos. Al inicializar la población con ciertos valores llamados "canónicos" en muy pocos pasos de recombinación es posible que se formen esquemas con cualquier combinación buena de bits fácilmente. Lo importante aquí es que todos los esquemas tienen una oportunidad más directa de formarse. Se espera que así puedan propagarse más temprano como bloque constructivo, ya poco después del inicio del algoritmo.

3.1.1 Inicialización

Mientras que en un SGA la inicialización de la población ocurre al azar, la población del algoritmo genético con inicialización canónica (GACI) inicialmente está compuesta por $2n$ individuos con *bitstrings* de longitud n con valores canónicos, tal que

$$b_{i,j} = \begin{cases} 1 & \text{si } j = i \\ 0 & \text{si } j \neq i \end{cases} \tag{3.1.1}$$

$$b_{n+i,j} = \begin{cases} 0 & \text{si } j = i \\ 1 & \text{si } j \neq i \end{cases} \tag{3.1.2}$$

$\forall i, j \in \{1, \ldots, n\}$, siendo $b_{i,j}$ el valor del j-ésimo bit del i-ésimo individuo.

3.2 Algoritmo genético con auto-adaptación

Normalmente la auto-adaptación (véase 2.9) se utiliza en estrategias de evolución (véase 2.11.1). En este caso la modificación consiste en incluirla en un algoritmo genético (GASA). Bäck [Bäc96] ya propuso un algoritmo genético donde incluyó unos bits en el cromosoma representando la probabilidad de mutación como parámetro endógeno. En cambio en esta modificación el parámetro más importante que se añade al cromosoma son los puntos de corte. Se espera que éstos - al evolucionar - deberían indicar los bloques constructivos. Así al cruzar sería posible hacer los cortes por bloques y combinarlos más temprano de una manera más explícita. También ya se experimentó con este modelo [SM87], pero con un operador de recombinación diferente ("*punctuated crossover*").

3.2.1 Componentes del individuo

Los parámetros que se desean auto-adaptar son la probabilidad de mutación, la probabilidad de recombinación y los puntos de corte para realizar la recombinación. Se alarga el *bitstring* para representar los parámetros (véase figura 3.1).

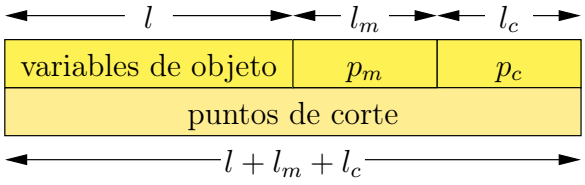

Figura 3.1: Componentes del *bitstring* auto-adaptativo

Sean l la longitud del *bitstring* de las variables de objeto, l_m la longitud del *bitstring* para representar la probabilidad de mutación p_m, y l_c la longitud de la de recombinación p_c. Para posibilitar la auto-adaptación de los puntos de corte en cualquiera de estos tres *bitstrings*, se necesita un *bitstring* adicional que consecuentemente tiene la longitud de la suma de estos *bitstrings*. En total, la longitud necesaria para incluir la auto-adaptación es $(l + l_m + l_c) \cdot 2$. Para codificar las probabilidades se utiliza código Gray (véase 2.11.4).

3.2.2 Mutación

El método de mutación en primer lugar extrae la probabilidad de mutación p_m y luego aplica *standard bit mutation* (véase 2.11.4) con p_m al *bitstring* entero.

3.2.3 Recombinación

El método de recombinación en primer lugar extrae de los dos progenitores las probabilidades de recombinación. Como probabilidad de recombinación p_c resultante sirve la media de los dos valores. A partir de ese valor se decide si se hace el cruce. En caso de

no hacerlo, simplemente se copian los progenitores.

Un punto de corte se identifica por la "máscara de cortes" (la segunda mitad de su cromosoma). Cuando aparezca un bit de valor 1, hay un punto de corte antes de ese bit. Para hacer el cruce se parte de un progenitor y se corta en su siguiente punto de corte según su "máscara de cortes". Luego se sigue en el otro progenitor hasta el siguiente punto de corte de este progenitor etc. Al mismo tiempo que se heredan los bits en este recorrido, también se heredan los puntos de corte (véase figura 3.2). Lo mismo se hace partiendo del otro progenitor (para obtener el otro descendiente).

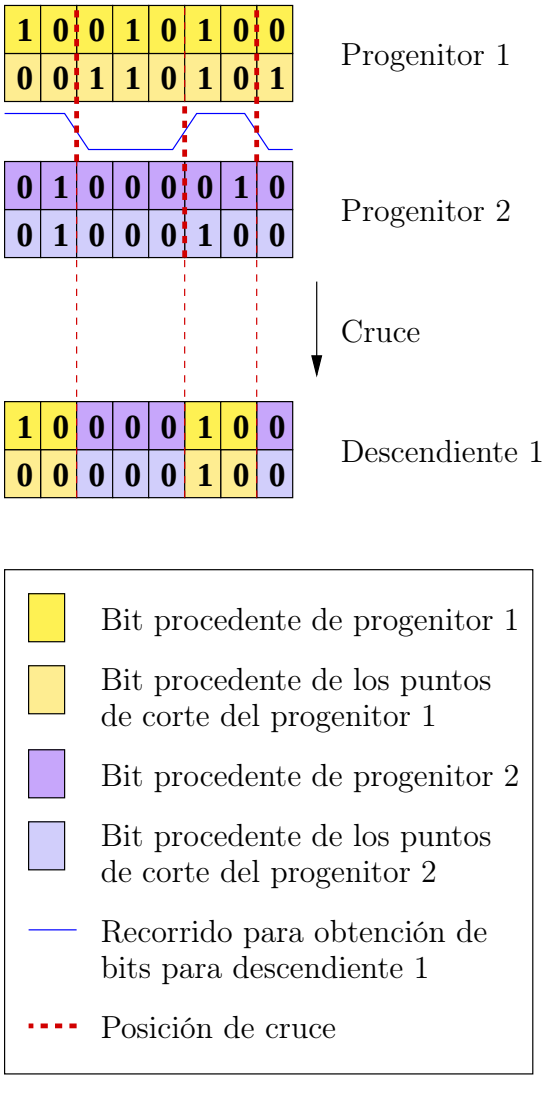

Figura 3.2: Recombinación auto-adaptativa

Ejemplo:
En el ejemplo que representa la figura 3.2 partimos del progenitor 1. Las variables de objeto, p_m y p_c corresponden a la parte superior (amarillo oscuro) del *bitstring* del progenitor 1, y su máscara de cortes a la parte inferior (amarillo claro). De manera análoga se compone el progenitor 2 (en azul). Siguiendo la máscara de cortes del progenitor 1, el primer bit de valor 1 indica que haremos un corte justo antes de ese bit. Los bits - tanto de la parte superior ('10') como de la inferior ('00') del progenitor 1 - hasta este corte se copian al descendiente. El corte significa que se sigue con el otro progenitor, es decir el 2, con los bits que suceden al corte. Aquí se hace lo mismo: Se sigue la máscara de cortes en la parte inferior del progenitor 2 hasta encontrar un bit de valor 1 que indica el siguiente corte. Se copian los bits de la parte superior ('000') y de la inferior ('000') recorridos hasta este corte al descendiente. Luego se sigue con el progenitor 1, y así sucesivamente.

3.3 Algoritmo genético con detección de bloques constructivos

Tal como se dice en sección 2.12.2, en el transcurso de un GA se favorecen bloques constructivos que se combinan para formar cadenas buenas. La intención de la siguiente modificación (GABB) es detectar *explícitamente* unos candidatos a bloques constructivos a tiempo para aprovecharlos antes y así posiblemente aumentar el rendimiento.

3.3.1 Componentes del individuo

Se alarga el *bitstring* de cada individuo para representar en la segunda mitad los puntos de corte que delimitan los bloques del cromosoma original, es decir, se duplica la longitud del *bitstring* (véase figura 3.3).

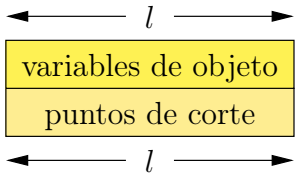

Figura 3.3: Componentes del *bitstring* con delimitaciones de bloques constructivos

3.3.2 Funcionamiento del algoritmo

La detección de bloques ocurre cada cierto número de generaciones, que se debe fijar. Después de la detección se reinicializa la población aprovechando los bloques detectados.

Estos estarán incluidos en los individuos. Además los individuos estarán preparados para intercambiar los bloques entre sí en las próximas generaciones. Esto se consigue permitiendo los cruces solamente en posiciones que delimitan los bloques detectados en cada uno de los individuos.

Detección de bloques constructivos

Se calcula $\forall k \in \{1, \ldots, n\}, \forall j \in \{1, \ldots, l\}$

$$g'(k, j) \equiv g(a_k) - g(\bar{a}_k(j)), \tag{3.3.1}$$

donde a_k es el k-ésimo individuo y $\bar{a}_k(j)$ es un individuo que contiene los mismos bits que a_k menos en la posición j, donde el bit está invertido.

Los bloques de bits consecutivos con $g'(k, j) < 0$ son los candidatos a bloques constructivos[1].

Reinicialización de la población

A cada individuo se le asignan los puntos de corte en los límites de sus bloques detectados. Los bits que pertenecen a los bloques no se alteran. Los bits que están fuera de los bloques se pueden dejar como están o – opcionalmente – darles un valor aleatorio.

3.3.3 Recombinación

Para hacer el cruce se parte de un progenitor y se copian sus bits hasta que aparezca un punto de corte en uno de los dos progenitores. Ahora se decide aleatoriamente si se sigue con el mismo progenitor o con el otro progenitor. Los puntos de corte sin embargo siempre serán heredados de un progenitor solamente (véase figura 3.4). El segundo descendiente tendrá cada bit del progenitor opuesto.

Ejemplo:
En el ejemplo que representa la figura 3.4 partimos del progenitor 1. Las variables de objeto corresponden a la parte superior (amarillo oscuro) del *bitstring* del progenitor 1, y su máscara de cortes a la parte inferior (amarillo claro). De manera análoga se compone el progenitor 2 (en azul). Siguiendo las máscaras de cortes de ambos progenitores, el primer bit de valor 1 en cualquiera de ambas máscaras indica que se decide al azar si haremos un corte justo antes de ese bit. Aquí aparece en la máscara de cortes del progenitor 2 tras el primer bit. Los bits de la parte superior ('1') del progenitor 1 hasta esta posición se copian al descendiente 1. Ahora se decide al azar si realmente hay un corte o no. De no haber un corte se seguiría con el mismo progenitor, pero en el ejemplo en esta

[1] Se refiere al objetivo de maximizar g. Para minimizar g tiene que cumplirse $g'(k, j) > 0$.

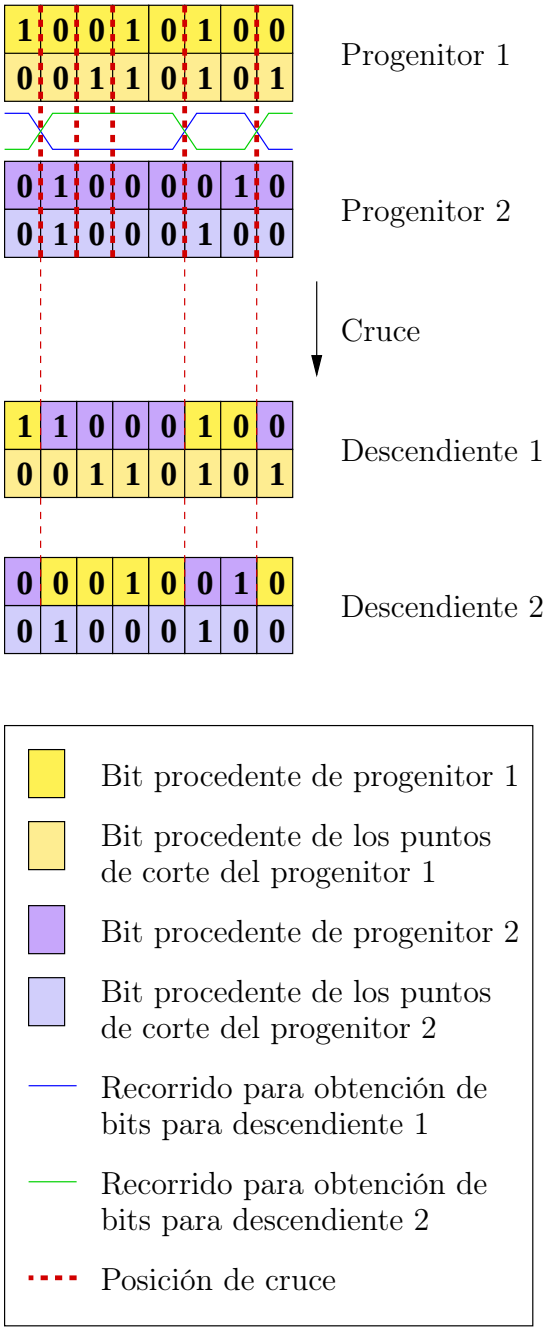

Figura 3.4: Recombinación con delimitaciones de bloques constructivos

posición se decide a favor de un corte. Esto significa que se sigue con el otro progenitor, es decir el 2, con los bits que suceden al corte. Aquí se hace lo mismo: Se siguen las máscaras de cortes en la parte inferior de ambos progenitores hasta encontrar un bit de valor 1. Esta vez lo encontramos en la máscara del progenitor 1. Los bits recorridos de la parte superior ('1') del progenitor 2 hasta esta posición se copian al descendiente 1. Se decide al azar si haremos un corte. Esta vez, en el ejemplo se decide en contra de un corte. Consecuentemente seguiremos con nuestro recorrido sin cambiar de progenitor (nos quedamos en el 2). El próximo posible corte aparece inmediatamente después en la máscara del progenitor 1. Otra vez copiamos los bits superiores ('0') del progenitor 2 (el actual) al descendiente 1. Decidimos de nuevo no hacer un corte y seguimos con el mismo progenitor 2. Tal como indica figura 3.4 aparecen dos posiciones más de posibles cortes. En ambos casos se decide a favor de un corte, lo cual significa que nuestro recorrido cambia todavía dos veces más de progenitor.

Después de finalizar este recorrido se copia la máscara de cortes del progenitor 1 al descendiente 1. Tal como se describe, el descendiente 2 tendrá cada bit del progenitor opuesto.

4 Desarrollo

Para poder realizar el estudio experimental y evaluar el rendimiento del efecto de las modificaciones, se desarrolla un entorno software para algoritmos evolutivos. Para describir el desarrollo del software se utiliza el *Unified Modeling Language (UML)*, un lenguaje muy utilizado en ingeniería de software que permite especificar, visualizar y construir un sistema.

En este capítulo se asume que el lector tiene unos conocimientos básicos sobre la ingeniería de software y la notación del lenguaje UML para desarrollar un sistema orientado a objetos, ya que el volumen de esta memoria se excedería al profundizar explicaciones de estos conceptos. Más bien la siguiente bibliografía se recomienda para utilizar como referencia: [Lar98], [BJR99], [RJB98] y [GHJV00].

En el desarrollo se trata de destacar los detalles relevantes para facilitar la comprensión del sistema. De lo contrario se "sobrecarga" al lector con aspectos que no son esenciales. Además, para no exceder el volumen de esta memoria, se prescinde de modelar en UML algunos detalles que no se consideran importantes en este proyecto.

Respecto a los factores de calidad de este software, las propiedades más consideradas serán la de ser modificable y portable, es decir facilitar la extensión del software con nuevas funcionalidades o mejoras y p. ej. cambios de plataformas de sistemas operativos. La eficiencia de un sistema que implementa un algoritmo evolutivo depende en la práctica sobre todo de la función de evaluación, por lo que no será considerada como factor predominante.

Los procesos de desarrollo que se presentarán a continuación son el análisis, el diseño y la implementación.

4.1 Análisis

4.1.1 Casos de uso

Los casos de uso describen quiénes son los actores que utilizan el sistema y con qué fin lo utilizan. El fin de utilizarlo tiene que tener algún valor para el actor. Estos son los casos de uso del sistema:

- El usuario, como actor, puede realizar un experimento con un algoritmo evolutivo. Para ello puede especificar los parámetros del experimento introduciéndolos por la línea de comandos. Alternativamente puede indicar por la línea de comandos un fichero que facilita la especificación del experimento deseado. Como resultado obtiene el fichero indicado que contiene la mejor solución encontrada durante la optimización y - opcionalmente - datos del transcurso de la evolución. Este le podrá servir para hacer un estudio experimental.
- El usuario puede realizar un experimento con un algoritmo evolutivo como en el caso anterior, pero esta vez accede al sistema mediante un interfaz gráfico (GUI) donde especifica los parámetros del experimento. El GUI también puede mostrar un gráfico que indica la media de los mejores valores de evaluación por generación y el valor de evaluación de la mejor solución encontrada durante el transcurso del experimento.
- El usuario puede transformar los datos del fichero, que obtuvo como resultado de un experimento y que contiene el recorrido de la evolución, a datos estadísticos más resumidos que obtendrá en otro fichero indicado. Este le podrá servir para hacer un análisis estadístico y visualizar el recorrido del experimento.

En figura 4.1 se ilustran estos casos, los actores y las relaciones entre actores y los casos de uso.

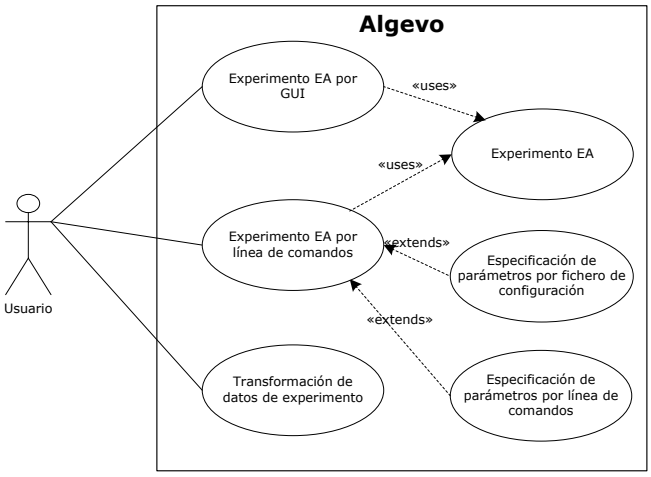

Figura 4.1: Diagrama de casos de uso

4.2 Diseño

4.2.1 Diagramas de clases

A continuación se presentan los diagramas de clase para explicar el diseño del sistema. Se subdivide el diagrama completo en partes coherentes del sistema. Se explican estas partes para luego juntarlas en el diagrama completo. En las clases del diagrama no se muestran las operaciones estándar como las de obtención y asignación de atributos de una clase (*set, get*), o de la copia de un objeto (*copy, clone*), etc. Solamente se muestra información relevante para la comprensión del diseño.

El diagrama se subdivide básicamente a partir de las siguientes clases:

- EvolutionaryAlgorithm
- Population
- Selection
- Individual
- EvolutionaryAlgorithmFactory

La interfaz gráfica del usuario (GUI) no está incluido en el diagrama porque solamente interesa el entorno para realizar algoritmos evolutivos.

Diagrama de *EvolutionaryAlgorithm*

La clase *Experiment* representa el controlador fachada del sistema para tratar los eventos externos. Se encarga de cargar, guardar, generar, ejecutar y administrar experimentos con algoritmos evolutivos.

La clase *EvolutionaryAlgorithm* proporciona los métodos para el funcionamiento principal de un algoritmo evolutivo. El SGA y los algoritmos modificados heredan esta clase y pueden determinar su funcionamiento y su especificación individual. Cualquier algoritmo evolutivo requiere al menos una población de individuos. Además se le asocia un criterio de terminación. Hay muchas diferentes maneras - permitidas y no permitidas - de componer un determinado algoritmo evolutivo. Para fijar cómo componer cada uno de ellos se le asigna una fábrica que se encarga de crear sus componentes adecuados (véase 4.2.1).

La clase *Termination* es la interfaz para cualquier criterio de terminación de un EA. Se representa en una propia clase al aplicar el patrón de diseño *estrategia*. Este patrón se explica en 4.2.1. La estrategia concreta que se utiliza para los algoritmos de este proyecto es la terminación por número de generaciones (*GenerationalTermination*).

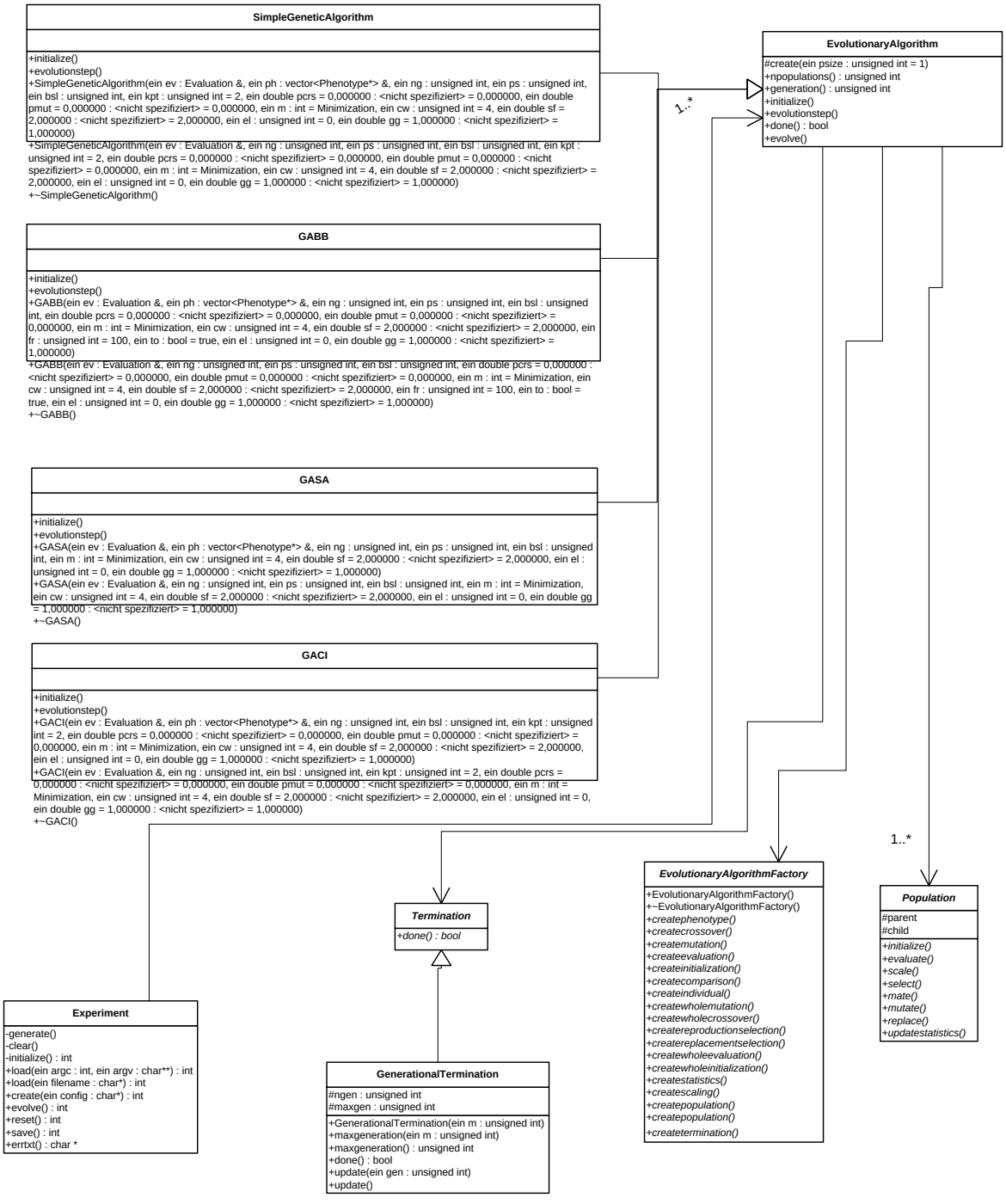

Figura 4.2: Diagrama de EvolutionaryAlgorithm

Diagrama de *Population*

La clase *Population* es la interfaz para una población. Todas las poblaciones contienen descendientes (*child*), progenitores (*parent*) y métodos para su inicialización, evaluación de sus individuos, selección para reproducción y reemplazamiento, escalamiento para determinar el *fitness*, recombinación y mutación de sus individuos. Además se administran unos valores estadísticos en *Statistics*. De nuevo se utiliza el patrón de diseño *estrategia*. La clase *StandardPopulation* hereda e implementa *Population* y también especifica un número de progenitores para la recombinación (*mixing number*). La clase *BuildingBlockPopulation* sirve para el GABB. Hereda de *StandardPopulation* y le añade una frecuencia que determina el número de generaciones después del cual se detectan posibles bloques constructivos a partir de los cuales se genera la población. La manera de detectar y generar los bloques se especifica por una asociación a un *BuildingBlockStrategy*, utilizado por el método *generatebb*.

Las próximas cuatro clases explicadas también se utilizan en el sentido del patrón *estrategia*, es decir, se pueden heredar y reemplazar por un funcionamiento distinto al que tienen por defecto.

La clase *WholeInitialization* realiza la inicialización de la población como totalidad, es decir, llama a la inicialización de cada individuo pero no es responsable de cómo hacer la inicialización individual. En cambio la clase *CanonicInitialization*, que hereda *WholeInitialization*, realiza la inicialización para el GACI asignando de manera correspondiente directamente los bits a todos los individuos, ya que al depender del tamaño poblacional no puede pasar la responsabilidad de inicializar a los individuos.

La clase *WholeCrossover* realiza la recombinación de los individuos de la población como totalidad. Llama al método de recombinación de los individuos progenitores (tantos como se especifica en *mixing number*) y los añade a la población como descendientes. La responsabilidad de efectuar cada recombinación se le pasa al individuo.

La clase *WholeMutation* funciona de manera análoga. Llama al método de mutación de cada individuo descendiente previamente recombinado.

La clase *WholeEvaluation* llama a la función de evaluación de cada individuo. La evaluación concreta se realiza a partir de la función de evaluación que tiene asignado cada individuo.

El *Scaling* aplica el escalamiento a un individuo determinado para obtener el *fitness* a partir de su valor de evaluación. Obtiene como parámetro un vector de valores de evaluación para calcular los *fitness* y devolverlos. Así - si el *fitness* depende de los valores de los demás individuos (como es el caso en el escalamiento lineal) - se puede pasar y

calcular todo un vector de *fitness*. El *LinearScaling* realiza el escalamiento lineal y tiene el atributo c_{mult} (véase 2.5.2). El *MinimumMapping* y el *MaximumMapping* realizan un mapeo para minimizar o maximizar respectivamente (véase 5.1.3). Se aplica el patrón de diseño *composición* que se explica en la parte del diagrama de selección. Este patrón trata el mapeo como una composición de escalamiento. Permite componer un escalamiento que primero aplica el mapeo y luego cualquier otro escalamiento, como el escalamiento lineal, a partir de los valores mapeados.

La clase *Statistics* proporciona los datos estadísticos y un método que los actualiza después de cada generación. Los datos son *online performance*, *offline performance* (véase 2.10), el mejor valor de evaluación jamás obtenido, la diversidad entre los individuos [1], la convergencia [2], el mejor individuo y otros valores que se refieren a los valores de evaluación y al *fitness*; estos son el mejor valor [3], la media, la suma, la varianza y la desviación estándar.

La clase *BuildingBlockStrategy* es la interfaz para las clases funcionales que realizan la detección de bloques constructivos y la reinicialización de la población a partir de los bloques detectados. Las clases *BBFlipBitwiseTakeover* y *BBFlipBitwiseRandom* implementan las dos opciones que se describen en 3.3.2.

Diagrama de *Selection*

La clase *Selection* es la interfaz para la selección para reproducción y selección para reemplazamiento, que se implementan en clases que la heredan. Su método obtiene un vector de individuos como entrada, a partir de los cuales aplica la selección. Los individuos seleccionados se devuelven en otro vector de salida. La clase *CutSelection* realiza la selección por corte (2.5.5) y la clase *TournamentSelection* la selección por torneo (2.5.3). En ellas hay que especificar el número de individuos a seleccionar, en *TournamentSelection* además el número de individuos que participan en cada torneo. El *FitnessProportionalSelection* es la interfaz para la selección proporcional al *fitness*. Tanto esta clase como la clase *CutSelection* tienen un escalamiento asignado para poder calcular el *fitness*. Al *FitnessProportionalSelection* pertenecen el *RouletteWheelSelection* y el *RemainderStochasticSamplingWithoutReplacement* (véase 2.5.2 y 2.5.4). El *AgedSelection* realiza una selección de reemplazamiento de individuos que tienen una edad máxima indicada. El *CommaSelection* realiza la selección coma (2.5.7) y es un caso especial del *AgedSelection* ya que selecciona los individuos con la edad máxima de una generación. El *Elitism* realiza la selección de reemplazamiento de elitismo (2.5.8). Tanto el *AgedSelection* como

[1] La diversidad es la media de los valores que resultan de comparar todas las parejas de individuos de la población (aplicando el método *compare*).

[2] Para la convergencia se divide el mejor valor de evaluación actual entre el mejor valor de evaluación de n generaciones atrás, donde n es la ventana de convergencia (*convergence window*).

[3] Según el objetivo es el máximo o el mínimo.

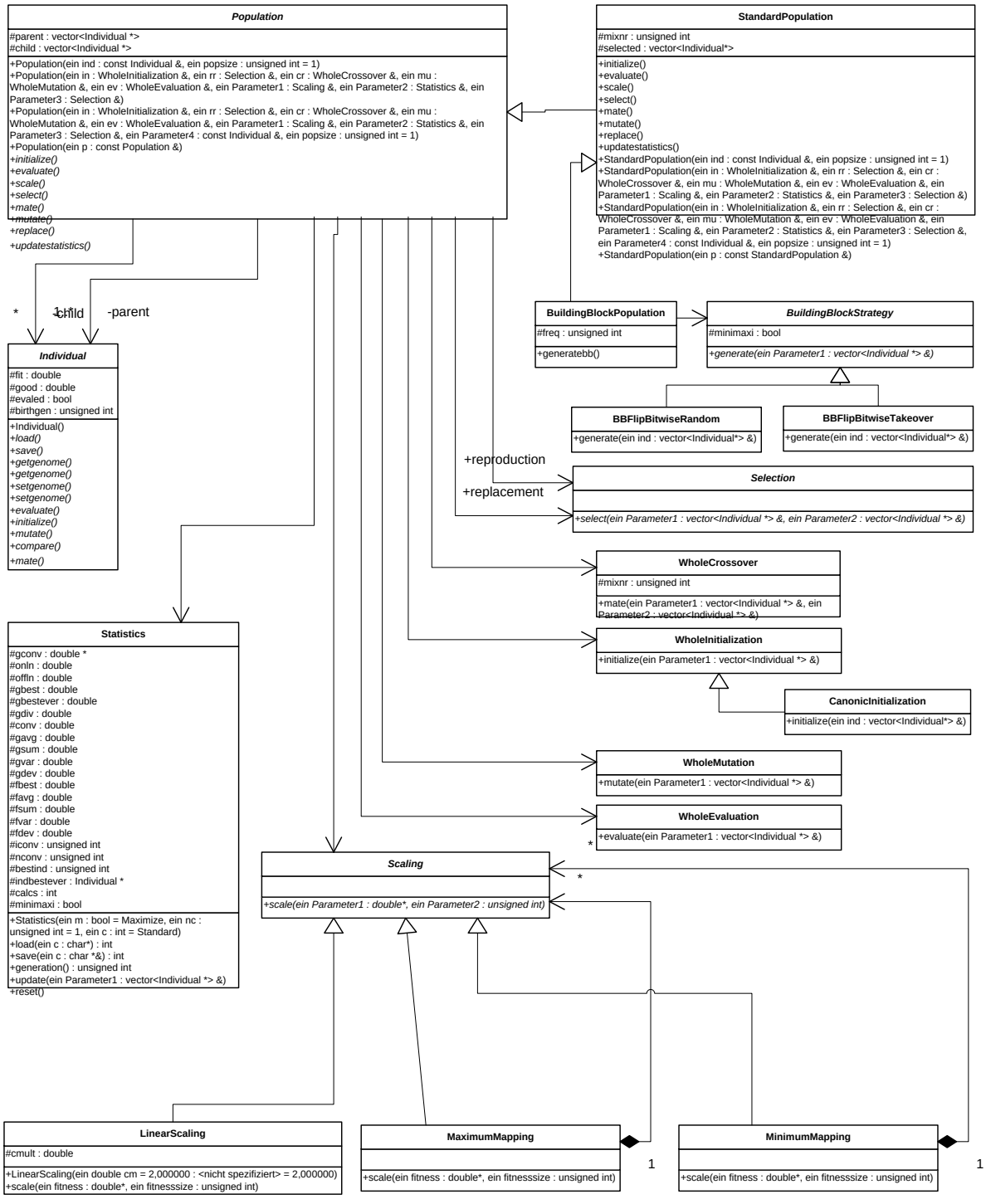

Figura 4.3: Diagrama de Population

el *Elitism* aplican el patrón de diseño *composición*. Este patrón compone familias de clases funcionales como la de selección. En este caso varias selecciones resultan en una selección. Así, p. ej. , se aplica el elitismo después de haber aplicado la selección coma. De este modo una selección está compuesta de otras y el cliente (aquí un *Population*) puede tratar cualquier selección de la misma manera.

Diagrama de *Individual*

La clase *Individual* es la interfaz para representar un individuo. Contiene el valor de evaluación, el *fitness* y la generación de su creación. Esta clase clase permite definir un individuo concreto con un cromosoma de cualquier estructura. El *GAIndividual*, p. ej. , contiene un *bitstring* como cromosoma. Los métodos que debe implementar un individuo permiten compararlo con otros, inicializarlo, mutarlo, recombinarlo, evaluarlo e interpretar su cromosoma especificando fenotipos. Para estos métodos se aplica el patrón *estrategia*, es decir, la funcionalidad se le pasa a una clase funcional que tiene la responsabilidad de realizarla. El objetivo del patrón *estrategia* es el de hacer cambiable un algoritmo permitiendo que varíe independientemente del cliente (aquí un individuo). La interfaz solo declara los métodos y atributos que debe conocer el cliente. Los procedimientos de mutación, recombinación, etc. ya no están acoplados al individuo. La responsabilidad se le pasa a la estrategia, la clase funcional. Aquí, a sus métodos solamente hace falta pasarles como parámetro la estructura de datos necesaria - y no el individuo completo - para realizar la operación. Las estrategias no tienen que tener conocimientos completos sobre un individuo. El patrón *estrategia* suele ser útil cuando algunas clases solo se distinguen en su comportamiento pero no en su interfaz. En el diseño de un entorno software sobre algoritmos evolutivos frecuentemente aparecen variantes de determinadas funcionalidades; p. ej. existen muchas maneras de mutación, recombinación, etc. Con este patrón se puede construir jerarquías para estas funcionalidades. Además se puede decidir en tiempo de ejecución cuál de las estrategias se aplica.

La clase *Initialization* es la interfaz para inicializar un individuo. El *BitStringRandomInit* da valores aleatorios a los bits de un *GAIndividual*.

La clase *Evaluation* es la interfaz para evaluar un individuo. Todas las funciones de evaluación la heredan e implementan su evaluación. El método devuelve el valor de evaluación. En el diagrama se muestra como ejemplo la función *F1*.

La clase *Comparison* es la interfaz para comparar un individuo con otro. Su método devuelve un valor que representa la similitud de individuos. El *Hamming* implementa la distancia de Hamming.

La clase *Crossover* es la interfaz para la recombinación de individuos. Su método obtiene como entrada los cromosomas de los progenitores y devuelve los cromosomas

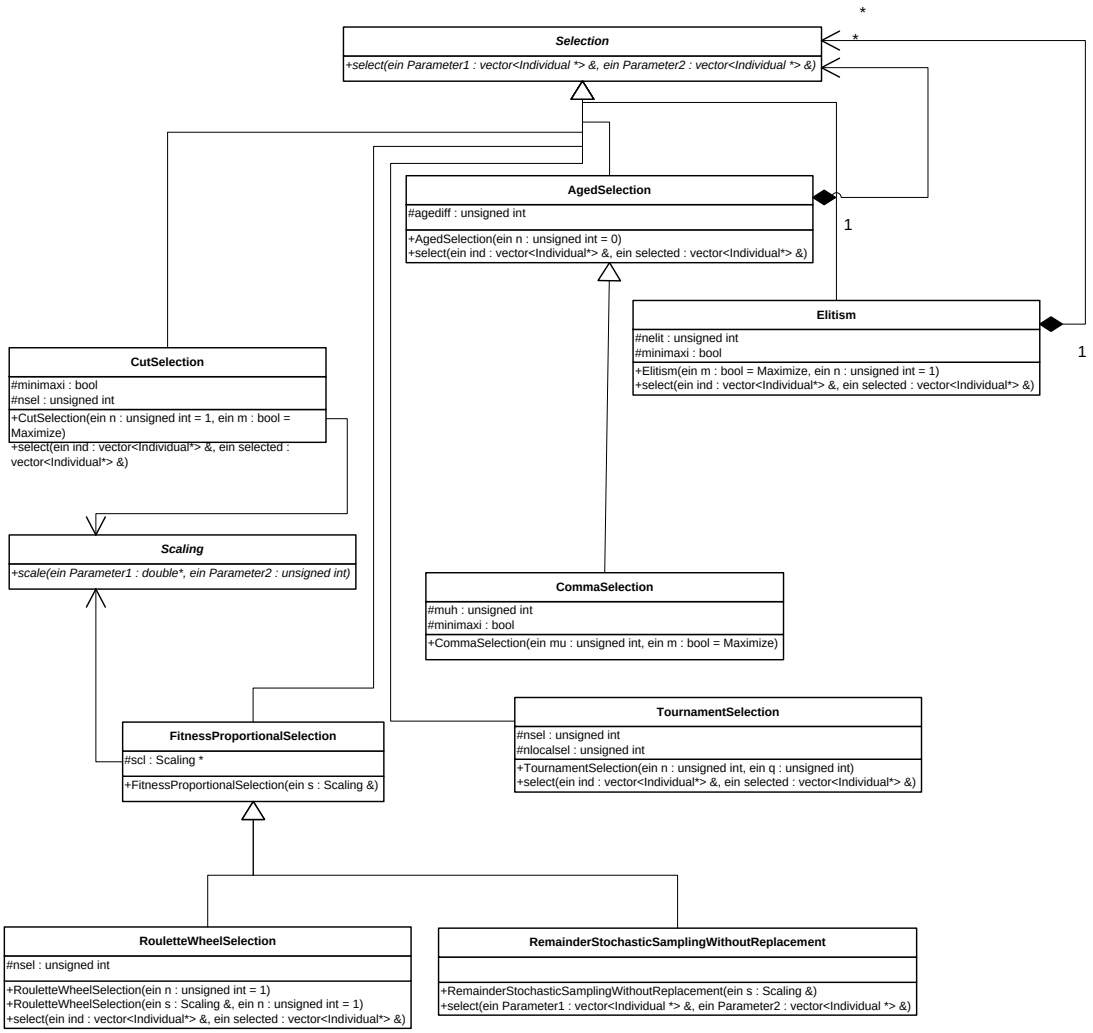

Figura 4.4: Diagrama de Selection

creados para los descendientes.

La clase *Mutation* es la interfaz para la mutación de individuos. Su método obtiene como entrada un cromosoma y lo muta.

La clase *Phenotype* es la interfaz para codificar y decodificar cromosomas.

Diagrama de *EvolutionaryAlgorithmFactory*

El diseño de este entorno es muy flexible y permite representar diversas estrategias para las diferentes partes de un algoritmo evolutivo. No todas las combinaciones de estas estrategias tienen sentido. P. ej. , se podría tener un individuo con un cromosoma de números reales. Entonces la operación de mutación *Standard bit mutation* no se le puede aplicar. Consecuentemente hace falta un mecanismo para restringir las estrategias en cada tipo de algoritmo evolutivo. Esto se consigue con el patrón de diseño *fábrica* que controla la creación de las partes de cada tipo de algoritmo con un interfaz común.

Diagrama completo

En el diagrama completo se muestran todas las clases (excepto las del GUI). Para facilitar al GUI los comandos para controlar el sistema, se ha utilizado el patrón de diseño *Command*.

4.3 Implementación

El código fuente de la implementación está en el CD del proyecto. Como lenguaje de programación se utiliza el C++. Para hacer el *parsing* de la línea de comandos o de un fichero que especifica un experimento, se genera un *parser* mediante las utilidades *Flex* y *Bison*. Esto permite modificar y extender fácilmente la gramática para especificar experimentos. La parte del GUI se realiza con la librería Qt y Qwt. Para la generación de números aleatorios se ha adaptado un código C de un generador de la Universidad de California a código C++.

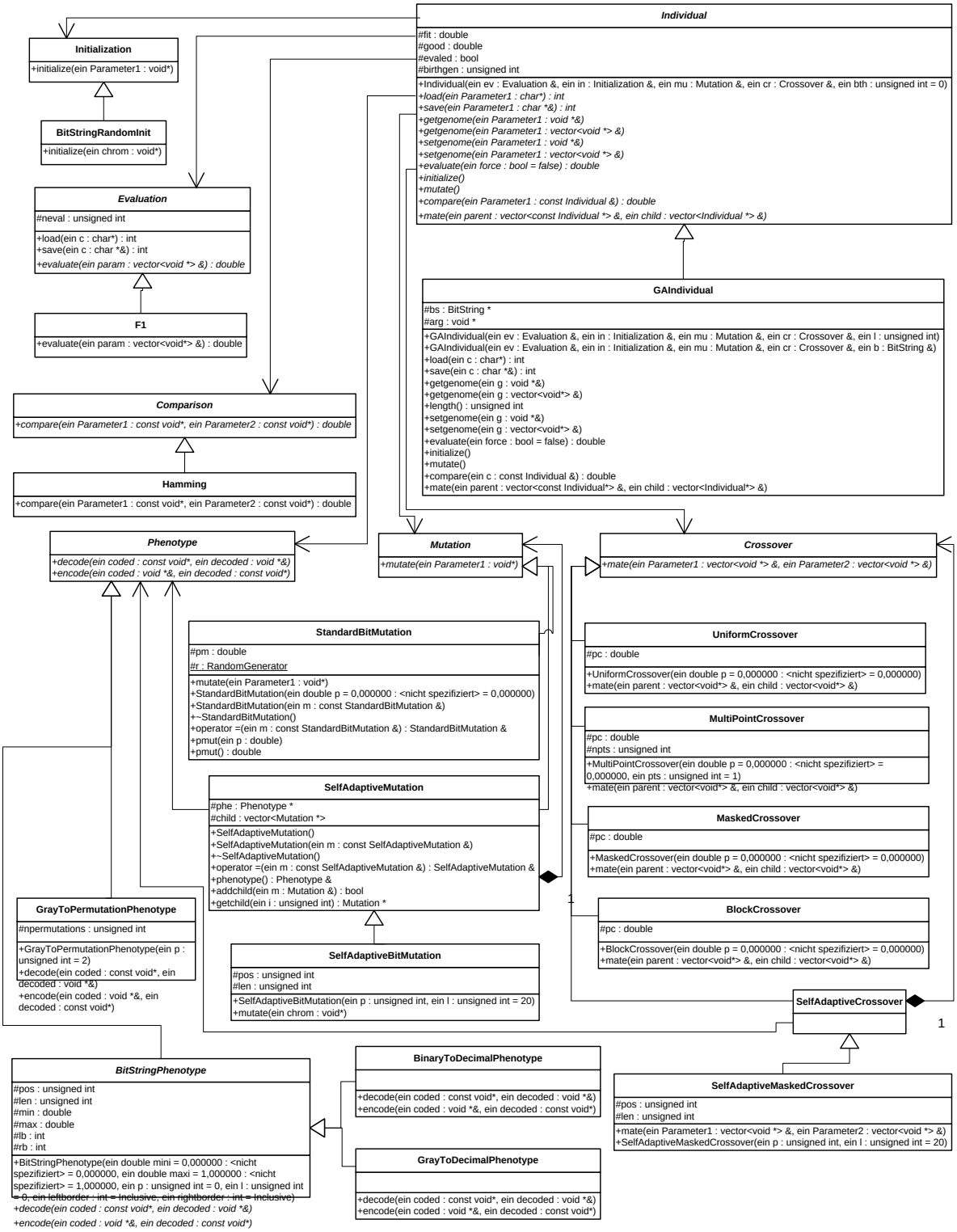

Figura 4.5: Diagrama de Individual

Figura 4.6: Diagrama de EvolutionaryAlgorithmFactory

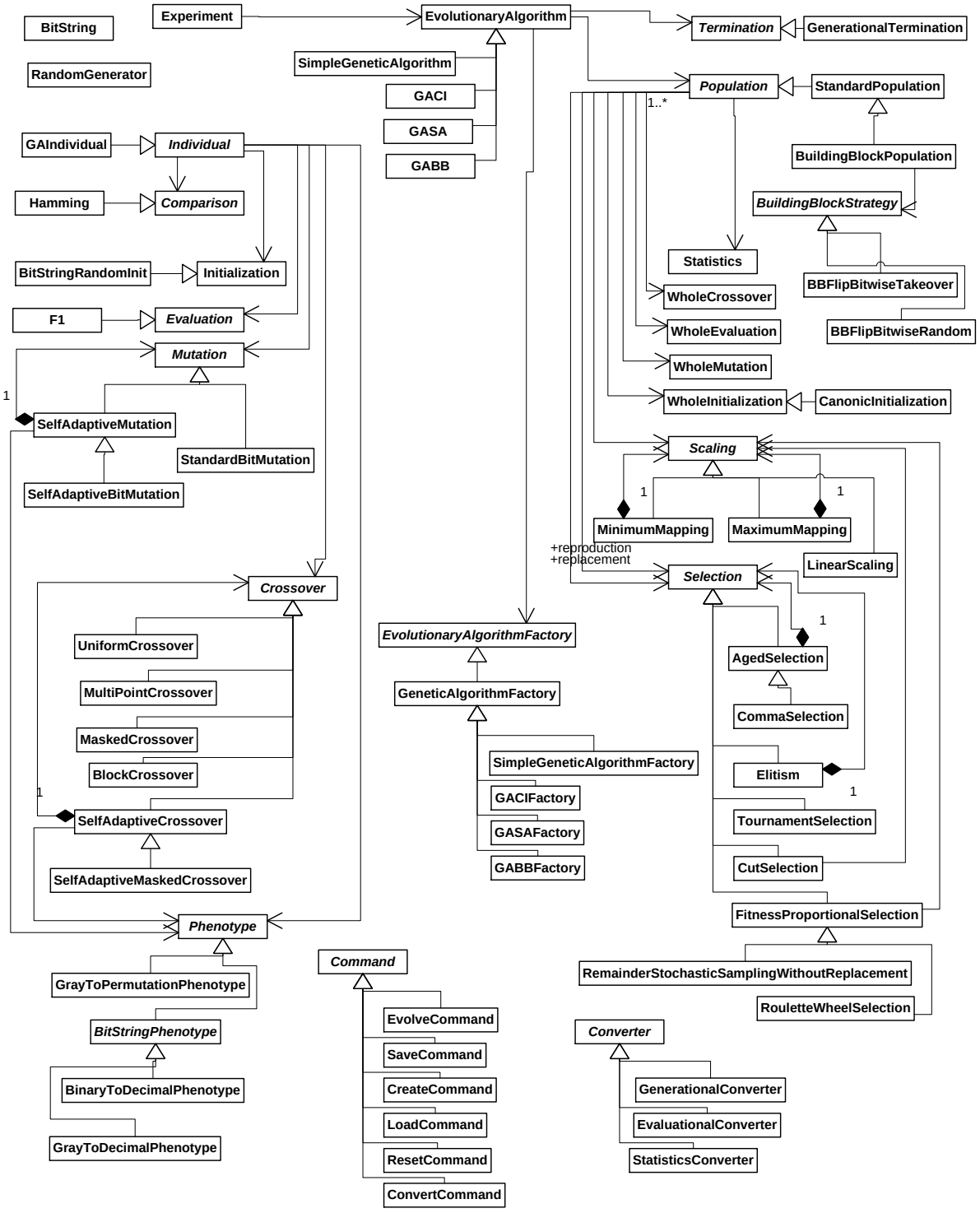

Figura 4.7: Diagrama completo

5 Estudio experimental

El objetivo de estos experimentos es el de analizar y comparar el rendimiento de los algoritmos modificados y el algoritmo genético estándar. Para ello se ejecutará el sistema desarrollado.

5.1 Metodología

Para conseguir una comparación razonable hay que considerar varios aspectos en la realización del estudio experimental. A continuación se explican detalladamente los métodos que se siguen en cada fase del estudio para permitir una interpretación real de la validación y posibilitar la reproducción del estudio. Las fases de la experimentación son las siguientes:

- Definición de los objetivos del experimento
- Selección de funciones de evaluación (*test problems*)
- Selección de parámetros adecuados para los algoritmos estudiados
- Selección del modelo de validación
- Ejecución de los algoritmos del estudio

5.1.1 Definición de objetivos

En general, el objetivo del estudio es la comparación del rendimiento de los algoritmos SGA, GACI, GASA y dos variantes de GABB llamados GABBTO y GABBRA para unas clases de problemas caracterizados. En la reinicialización de la población, en el GABBTO no se modifican los bits que están fuera de los bloques mientras que en el GABBRA a estos se les da un valor aleatorio (véase 3.3.2).

Se mide el rendimiento a partir de dos criterios:

- Bondad de la solución conseguida
- Rapidez de conseguir soluciones buenas

Bondad de la solución conseguida

La bondad de la solución será el criterio más valorado y representa la efectividad del algoritmo. Es representado por el mejor valor de evaluación de cada algoritmo. Estos valores se comparan cuando se haya cumplido un criterio de terminación para determinar la solución conseguida. En la práctica el factor decisivo que retarda un algoritmo evolutivo es el número de evaluaciones ejecutadas. La evaluación - a parte de ser llamada muy a menudo - puede y suele ser una función bastante compleja (p. ej. una simulación) que necesita mucho más tiempo que el resto de las operaciones del algoritmo. Consecuentemente lo más justo es comparar los valores de evaluación después de un determinado número de evaluaciones. Así los diferentes algoritmos obtienen los mismos recursos para resolver un mismo problema de optimización. Para fijar el número de evaluaciones como criterio de terminación, se averigua el instante en el cual el mejor valor de evaluación de cada algoritmo empieza a estancarse. Esto se hará "a ojo" aunque también se puede determinar el instante contando las evaluaciones que lleva el algoritmo sin encontrar una solución mejor. De todas formas no hace falta determinar el instante muy exacto porque al estancarse el valor apenas cambia, por lo que la exactitud del instante no tendrá consecuencias a la hora de validar las bondades.

Hay casos en los cuales alguno de los algoritmos resulta inadecuado y necesita un número de evaluaciones muy alto para estancarse en comparación con los demás. En estos casos se determina el número de evaluaciones a partir de los algoritmos adecuados.

Rapidez de conseguir soluciones buenas

En caso de que las bondades de las soluciones no permitan determinar el mejor algoritmo, se puede tomar en consideración el comportamiento de cada algoritmo para encontrar soluciones buenas. Durante el transcurso del algoritmo se observan las mejores soluciones de cada algoritmo. Puede haber alguno que encuentre soluciones buenas más rápido que otro, es decir, que necesite menos evaluaciones para estancarse porque tiene una mayor eficiencia.

5.1.2 Funciones de evaluación

Para evaluar el rendimiento de los algoritmos se utilizan problemas de optimización con diferentes características. Esto es necesario ya que no se quiere obtener resultados para solo una clase determinada de problemas sino que se quiere estudiar los algoritmos con problemas de diferente complejidad. A continuación se presentan los 26 problemas usados y categorizados según sus características más importantes. Todos ellos son problemas muy típicos de optimización utilizados y reconocidos como *benchmark functions* por muchos científicos, como muestran p. ej. las publicaciones [YL97], [Bal94], [Kea95] y [Tai93], de donde se extrajeron en este proyecto.

Funciones unimodales

Las siguientes funciones tienen una dimensión media y contienen solamente un mínimo. La función f_6 es la *step function*. Es discontinua y su topología contiene superficies planas. Esto es un obstáculo para el algoritmo de optimización ya que no se facilita información sobre la dirección favorable a seguir. La función f_7 es ruidosa porque contiene valores aleatorios. Esto provoca que en el mismo punto cada vez se obtiene un valor diferente. Muchos problemas reales contienen datos ruidosos. Por eso puede ser importante conocer el rendimiento para este tipo de problema.

- ***Sphere Model***

$$f_1(x) = \sum_{i=1}^{n} x_i^2$$

$$n = 30, \quad -100 \leq x_i \leq 100, \quad \text{mín}(f_1) = f_1(0, \ldots, 0) = 0$$

- ***Schwefel's Problem 2.22***

$$f_2(x) = \sum_{i=1}^{n} |x_i| + \prod_{i=1}^{n} |x_i|$$

$$n = 30, \quad -10 \leq x_i \leq 10, \quad \text{mín}(f_2) = f_2(0, \ldots, 0) = 0$$

- ***Schwefel's Problem 1.2***

$$f_3(x) = \sum_{i=1}^{n} \left(\sum_{j=1}^{i} x_j \right)^2$$

$$n = 30, \quad -100 \leq x_i \leq 100, \quad \text{mín}(f_3) = f_3(0, \ldots, 0) = 0$$

- ***Schwefel's Problem 2.21***

$$f_4(x) = \underset{i}{\text{máx}} \{|x_i|, 1 \leq i \leq n\}$$

$$n = 30, \quad -100 \leq x_i \leq 100, \quad \text{mín}(f_4) = f_4(0, \ldots, 0) = 0$$

- ***Generalised Rosenbrock's Function***

$$f_5(x) = \sum_{i=1}^{n-1} [100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2]$$

$$n = 30, \quad -30 \leq x_i \leq 30, \quad \text{mín}(f_5) = f_5(1, \ldots, 1) = 0$$

- ***Step Function***

$$f_6(x) = \sum_{i=1}^{n} (\lfloor x_i + 0.5 \rfloor)^2$$

$$n = 30, \quad -100 \leq x_i \leq 100, \quad \text{mín}(f_6) = f_6(0, \ldots, 0) = 0$$

- ***Quartic Function with Noise***

$$f_7(x) = \sum_{i=1}^{n} i x_i^4 + random[0, 1)$$

$$n = 30, \quad -1.28 \leq x_i \leq 1.28, \quad \text{mín}(f_7) = f_7(0, \ldots, 0) = 0$$

Funciones multimodales con muchos mínimos locales

Estas funciones tienen una dimensión media y contienen muchos mínimos. El número de mínimos crece exponencialmente con la dimensión de la función. Pueden provocar que el algoritmo de optimización se centre demasiado en un mínimo local descartando a otros mínimos mejores.

- ***Generalised Schwefel's Problem 2.26***

$$f_8(x) = -\sum_{i=1}^{n} \left(x_i \sin\left(\sqrt{|x_i|} \right) \right) + 12569.5$$

$$n = 30, \quad -500 \leq x_i \leq 500, \quad \text{mín}(f_8) = f_8(420.9687, \ldots, 420.9687) = 0$$

- ***Generalised Rastrigin's Function***

$$f_9(x) = \sum_{i=1}^{n} [x_i^2 - 10\cos(2\pi x_i) + 10]$$

$$n = 30, \quad -5.12 \leq x_i \leq 5.12, \quad \text{mín}(f_9) = f_9(0, \ldots, 0) = 0$$

- ***Ackley's Function***

$$f_{10}(x) = -20\exp\left(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2} \right) - \exp\left(\frac{1}{n}\sum_{i=1}^{n} \cos 2\pi x_i \right) + 20 + e$$

$$n = 30, \quad -32 \leq x_i \leq 32, \quad \text{mín}(f_{10}) = f_{10}(0, \ldots, 0) = 0$$

- ***Generalised Griewank Function***

$$f_{11}(x) = \frac{1}{4000}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$$

$$n = 30, \quad -600 \leq x_i \leq 600, \quad \text{mín}(f_{11}) = f_{11}(0, \ldots, 0) = 0$$

- ***Generalised Penalised Functions***

$$\begin{aligned} f_{12}(x) &= \frac{\pi}{n}\left\{10\sin^2(\pi y_1) + \sum_{i=1}^{n-1}(y_i - 1)^2[1 + 10\sin^2(\pi y_{i+1})] + (y_n - 1)^2\right\} \\ &+ \sum_{i=1}^{n} u(x_i, 10, 100, 4) \end{aligned}$$

$$n = 30, \quad -50 \leq x_i \leq 50, \quad \text{mín}(f_{12}) = f_{12}(-1, \ldots, -1) = 0$$

$$\begin{aligned} f_{13}(x) &= 0.1\Big\{\sin^2(\pi 3x_1) + \sum_{i=1}^{n-1}(x_i - 1)^2[1 + \sin^2(3\pi x_{i+1})] \\ &+ (x_n - 1)[1 + \sin^2(2\pi x_n)]\Big\} + \sum_{i=1}^{n} u(x_i, 5, 100, 4) \end{aligned}$$

$$n = 30, \quad -50 \leq x_i \leq 50, \quad \text{mín}(f_{13}) = f_{13}(1, \ldots, 1) = 0$$

donde

$$u(x_i, a, k, m) = \begin{cases} k(x_i - a)^m, & x_i > a \\ 0 & -a \leq x_i \leq a, \\ k(-x_i - a)^m, & x_i < -a. \end{cases}$$

$$y_i = 1 + \frac{1}{4}(x_i + 1)$$

Funciones multimodales con pocos mínimos locales

Estas funciones tienen una dimensión baja y contienen pocos mínimos.

- ***Shekel's Foxholes Function***

$$f_{14}(x) = \left[\frac{1}{500} + \sum_{j=1}^{25} \frac{1}{j + \sum_{i=1}^{2}(x_i - a_{ij})^6}\right]^{-1}$$

$$-65.536 \leq x_i \leq 65.536, \quad \text{mín}(f_{14}) = f_{14}(-32, -32) \approx 1$$

donde

$$(a_{ij}) = \begin{pmatrix} -32 & -16 & 0 & 16 & 32 & -32 & \cdots & 0 & 16 & 32 \\ -32 & -32 & -32 & -32 & -32 & -16 & \cdots & 32 & 32 & 32 \end{pmatrix}$$

- ***Kowalik's Function***

$$f_{15}(x) = \sum_{i=1}^{11} \left[a_i - \frac{x_1(b_i^2 + b_i + x_2)}{b_i^2 + b_i x_3 + x_4} \right]^2$$

$$-5 \leq x_i \leq 5, \quad \text{mín}(f_{15}) \approx f_{15}(0.1928, 0.1908, 0.1231, 0.1358) \approx 0.0003075$$

i	a_i	b_i^{-1}
1	0.1957	0.25
2	0.1947	0.5
3	0.1735	1
4	0.1600	2
5	0.0844	4
6	0.0627	6
7	0.0456	8
8	0.0342	10
9	0.0323	12
10	0.0235	14
11	0.0246	16

- ***Six-hump Camel-Back Function***

$$f_{16}(x) = 4x_1^2 - 2.1x_1^4 + \frac{1}{3}x_1^6 + x_1x_2 - 4x_2^2 + 4x_2^4 + 1.0316285$$

$$-5 \leq x_i \leq 5$$

$$x_{min} = (0.08983, -0.7126), \ (-0.08983, 0.7126)$$

$$\text{mín}(f_{16}) = 0$$

- ***Branin Function***

$$f_{17}(x) = \left(x_2 - \frac{5.1}{4\pi^2}x_1^2 + \frac{5}{\pi}x_1 - 6 \right)^2 + 10\left(1 - \frac{1}{8\pi}\right)\cos x_1 + 10$$

$$-5 \leq x_1 \leq 10, \ 0 \leq x_2 \leq 15$$

$$x_{min} = (-3.142, 12.275), \ (3.142, 2.275), \ (9.425, 2.425)$$

$$\text{mín}(f_{17}) = 0.398$$

- ***Goldstein-Price Function***

$$\begin{aligned} f_{18}(x) \ = \ & [1 + (x_1 + x_2 + 1)^2(19 - 14x_1 + 3x_1^2 - 14x_2 + 6x_1x_2 + 3x_2^2)] \\ & \times [30 + (2x_1 - 3x_2)^2(18 - 32x_1 + 12x_1^2 + 48x_2 - 36x_1x_2 + 27x_2^2)] \end{aligned}$$

$$-2 \leq x_i \leq 2, \quad \text{mín}(f_{18}) = f_{18}(0, -1) = 3$$

Cuadro 5.1: Hartman Function f_{19}

i	$a_{ij}, j = 1, 2, 3$			c_i	$p_{ij}, j = 1, 2, 3$		
1	3	10	30	1	0.3689	0.1170	0.2673
2	0.1	10	35	1.2	0.4699	0.4387	0.7470
3	3	10	30	3	0.1091	0.8732	0.5547
4	0.1	10	35	3.2	0.038150	0.5743	0.8828

Cuadro 5.2: Hartman Function f_{20}

i	$a_{ij}, j = 1, \ldots, 6$						c_i	$p_{ij}, j = 1, \ldots, 6$					
1	10	3	17	3.5	1.7	8	1	0.1312	0.1696	0.5569	0.0124	0.8283	0.5886
2	0.05	10	17	0.1	8	14	1.2	0.2329	0.4135	0.8307	0.3736	0.1004	0.9991
3	3	3.5	1.7	10	17	8	3	0.2348	0.1415	0.3522	0.2883	0.3047	0.6650
4	17	8	0.05	10	0.1	14	3.2	0.4047	0.8828	0.8732	0.5743	0.1091	0.0381

- ***Hartman's Family***

$$f_{hartman}(x) = -\sum_{i=1}^{4} c_i \exp\left[-\sum_{j=1}^{n} a_{ij}(x_j - p_{ij})^2\right]$$

donde $n = 3,\ 6$ para $f_{19}(x)$ y $f_{20}(x)$, respectivamente, $0 \leq x_j \leq 1$. Los coeficientes están definidos en los cuadros 5.1 y 5.2, respectivamente.

Para $f_{19}(x)$ el mínimo global es -3.86 en el punto (0.114, 0.556, 0.852, 0.740). Para $f_{20}(x)$ el mínimo global es -3.32 en el punto (0.201, 0.150, 0.477, 0.275, 0.311, 0.657).

- ***Shekel's Family***

$$f_{shekel}(x) = -\sum_{i=1}^{m} [(x - a_i)(x - a_i)^T + c_i]^{-1}$$

donde $m = 5,\ 7,\ 10$ para f_{21}, f_{22} y f_{23}, respectivamente, $0 \leq x_j \leq 10$.

Estas funciones tienen 5, 7 y 10 mínimos locales para f_{21}, f_{22} y f_{23}, respectivamente, $x_{optimolocal} \approx a_i, f(x_{optimolocal}) \approx 1/c_i$ para $1 \leq i \leq m$. Los coeficientes están definidos en el cuadro 5.3.

Nota: Para realizar los experimentos a las funciones $f_{19} - f_{23}$ se les ha sumado un valor para que el óptimo sea positivo.

Cuadro 5.3: Shekel Functions f_{21}, f_{22}, f_{23}

i	$a_{ij}, j = 1, \ldots, 4$				c_i
1	4	4	4	4	0.1
2	1	1	1	1	0.2
3	8	8	8	8	0.2
4	6	6	6	6	0.4
5	3	7	3	7	0.4
6	2	9	2	9	0.6
7	5	5	3	3	0.3
8	8	1	8	1	0.7
9	6	2	6	2	0.5
10	7	3.6	7	3.6	0.5

Problemas de optimización combinatoria

Estos problemas pertenecen a los más difíciles que existen en la optimización.

- ***John Holland Royal Road*** **(JHRR)**
  Este problema obtiene un *bitstring* como entrada y devuelve un valor real que se ha de maximizar. El *bitstring* está compuesto de 2^k regiones continuas no solapadas, cada una de longitud $b + g$. Con los valores por defecto de Holland, $k = 4$, $b = 8$, $g = 7$, hay 16 regiones de longitud 15, resultando en una longitud total del *bitstring* de 240. Cada región está dividida en dos partes no solapadas. La primera parte de longitud b se llama *bloque*, y la segunda de longitud g *hueco*. En el cálculo del valor de evaluación solamente se considera la parte del bloque de cada región. El cálculo consiste en dos pasos. El primer cálculo suma en cada bloque un valor v por cada bit de valor 1 que contiene el bloque hasta un límite de m^*. Si un bloque contiene más de m^* bits de valor 1 pero menos que b bits de valor 1, se le resta v por cada bit de valor 1 que supera el límite. Los valores por defecto son $v = 0.02$ y $m^* = 4$, así a un bloque con 6 bits de valor de 1 se le asignará el valor $(6 - 4) \times -0.02 = -0.04$. Finalmente, si un bloque contiene solo bits de valor 1, no obtendrá ningún valor en este primer cálculo, sino que se considerará completo. Esto lleva al segundo cálculo que recompensa bloques completos. Al primer bloque completo se le suma un valor u^*, por defecto 1.0, y cada bloque siguiente un valor u, por defecto 0.3. Luego se recompensa parejas de bloques completos adyacentes de la misma manera, y luego 4 bloques completos en una fila y así siguiendo hasta todos los 16 bloques estén completos. Esto lleva al valor de evaluación máximo que es $1.0 + (1.0 + 0.3) + (1.0 + 3 \times 0.3) + (1.0 + 7 \times 0.3) + (1.0 + 15 \times 0.3) = 12.8$

  Los huecos de las regiones, que no afectan el cálculo, garantizan que exista un

número considerable de *bitstrings* que obtienen el valor máximo.

- ***Permutation Flowshop Problem* (PFSP)**
  En este problema existen n tareas que tienen que ser procesadas en m máquinas; cada tarea consiste en m operaciones. Cada operación de una tarea usa una máquina diferente durante un tiempo dado y puede esperar antes de ser procesada. Las operaciones de cada tarea tienen que ser procesadas en las máquinas $1, \ldots, m$ en este orden. Además, el orden de procesar las tareas en las máquinas es el mismo en cada máquina. El problema consiste en encontrar una permutación de las n tareas que minimice el tiempo total para terminar las tareas. En [Tai93] se presentan muchas instancias de este problema con diferente complejidad. Se utilizará el problema 51 de este documento con 50 tareas y 20 máquinas. Para esta instancia se desconoce el óptimo real pero hasta la fecha se pudo conseguir un valor de 3847 como mínimo tiempo total.

  La codificación binaria de un PFSP con n tareas se realiza con un *bitstring* de longitud $n \log_2 n$ bits. A una tarea se le asigna una parte continua del *bitstring* de longitud $\log_2 n$. Cada parte se interpreta como número entero del rango $[0, n-1]$. La tarea con el menor número entero es la primera, la tarea con el segundo menor entero es la segunda, etc. En caso de empate, la tarea de la cual la parte aparece antes será la primera de ellas.

- ***Checkerboard Problem* (CKBD)**
  Este problema requiere un *bitstring* de longitud 400 que se interpreta como una rejilla de 20×20. El objetivo del problema es crear un patrón de un tablero de ajedrez con bits en una rejilla de 20×20. Cada posición con valor 1 debe estar rodeada en las cuatro direcciones por un valor 0, y vice-versa. Solamente se consideran las cuatro direcciones principales en la evaluación. A partir del valor actual de cada bit en la rejilla de 18×18 (centrada en la de 20×20) se calcula el valor de evaluación contando el número correcto de los bits que lo rodean. De esta manera, las esquinas no cuentan en la evaluación. El valor máximo de evaluación es 1296 ($18 \times 18 \times 4$).

Nota: No se prueba el algoritmo GACI con el problema JHRR, ya que por casualidad la inicialización del GACI provoca que contiene la solución óptima directamente.

5.1.3 Selección de parámetros

Como se pudo ver en los capítulos anteriores, para los algoritmos genéticos necesitan especificarse varios parámetros. Si no se utilizan parámetros razonables en cada caso, los algoritmos pueden funcionar mal porque estos son factores que influyen el comportamiento del algoritmo. No es justo comparar un algoritmo con parámetros mal asignados con otro que los tiene bien. Hace falta una fase de *pretesting* para determinar los parámetros óptimos de cada algoritmo estudiado en cada problema de optimización aplicado.

Por un lado pueden haber infinitas asignaciones diferentes para los valores de los parámetros. Por otro lado se quieren determinar con pocas ejecuciones.

La manera de determinarlos en este proyecto es con el método *one-factor-at-a-time*. En este método unos determinados parámetros se varían mientras que los demás se dejan fijos, para así estimar la influencia que tiene un único parámetro en unas ciertas condiciones fijas previamente seleccionadas. Este método es muy corriente pero tiene grandes desventajas. P. ej. la estimación presupone que los efectos determinados del parámetro son los mismos cambiando los demás parámetros. No se pueden determinar las interacciones. En [BB06] se presentan mejores alternativas (*factorial-design*) que aquí se descartan para no exceder el volumen de este proyecto.

Los parámetros a determinar de los algoritmos son los siguientes [1]: tamaño poblacional, probabilidad de recombinación y de mutación, y la frecuencia de generaciones para la detección de bloques (GABB). Además en los problemas de las funciones numéricas $f_1 - f_{23}$ se puede escoger el tipo de codificación de las variables de objeto (código Gray o binario).

Otros parámetros no se han ajustado sino que se les dieron valores corrientes porque se consideran poco relevantes para el comportamiento de los diferentes algoritmos. Estos son el número de puntos para el *multi-point-crossover*, el mapeo y el factor de escalamiento.

Aparte de estos parámetros también hay que fijar unos que tienen que valer para todos los algoritmos, para hacer una comparación justa. Estos son la precisión de las variables de los problemas numéricos, es decir, el número de bits para cada variable y el elitismo.

En la fase de *pretesting* se han escogido los parámetros siguientes para los algoritmos estudiados:

Parámetro	**Valor**
Tamaño poblacional	200
Elitismo	1
Codificación	Gray
Puntos de cruce	2
Probabilidad de recombinación	0.60
Probabilidad de mutación	0.01
Factor de escalamiento	2.0
Número de bits por variable numérica	16

La frecuencia de detección de bloques constructivos es 500 para $f_1 - f_{13}$, f_{15}, JHRR,

[1] No todos los parámetros aparecen en todos los algoritmos

PFSP y CKBD; y para el resto 25. El número de bits para representar cada una de las probabilidades de variación en el GASA es 20.

El mapeo g' para los problemas de minimización es

$$g'(x) = c_{max} - g(x)$$

donde c_{max} es el máximo de los valores de la población actual.

Para problemas de maximización se mapea

$$g'(x) = g(x) - c_{min}$$

donde c_{min} es el mínimo de los valores de la población actual.

Luego queda determinar el número de evaluaciones como criterio de terminación. Este depende del instante en el cual el valor de evaluación se estanca. Los algoritmos genéticos tienen como criterio de terminación un número de generaciones. Este se calcula a partir del número de evaluaciones determinado.

Problema	**Número de evaluaciones**
$f_1 - f_{13}$, JHRR, PFSP, CKBD	2000000
f_{14}	4000
f_{15}	1920000
f_{16} y f_{18}	32000
f_{17}	6000
f_{19}	10000
f_{20}	12000
f_{21} - f_{23}	25600

5.1.4 Modelo de validación

Para validar los algoritmos según la definición de los objetivos (5.1.1), se tiene en cuenta la bondad de las soluciones y la rapidez de encontrarlas. Para obtener un muestreo representativo de resultados cada algoritmo se ejecuta 30 veces (para cada problema de optimización). El sistema desarrollado proporciona la mejor solución de cada optimización.

Validación de la bondad

Para comparar las bondades de cada algoritmo en cada problema, se consideran la mediana y el mejor de los 30 valores de evaluación del muestreo. Además se visualizan los valores mediante un *boxplot* (sin *outliers*). No siempre se puede garantizar que los valores de diferentes algoritmos se distinguen significativamente y así favorecer a uno. Para asegurarse de que hay una diferencia significativa entre las medianas de las 30 soluciones

de cada algoritmo, se aplica el *Mann-Whitney test*.

El *Mann-Whitney test* (véase también [Kan99]) verifica la igualdad de las medianas de dos poblaciones y calcula un punto de estimación correspondiente y el intervalo de confianza para la diferencia de las medianas. Las hipótesis son:

$$H_0 : \quad \eta_1 = \eta_2 \Leftrightarrow \eta_1 - \eta_2 = 0$$

$$H_1 : \quad \eta_1 \neq \eta_2 \Leftrightarrow \eta_1 - \eta_2 \neq 0$$

donde η es la mediana poblacional.

Una suposición para el *Mann-Whitney test* es que los datos son muestreos aleatorios independientes de dos poblaciones que tienen el mismo aspecto con la misma varianza y una escala que es continua u ordinal al ser discreta. Este test es un poco menos potente que el *2-sample test* pero no supone una distribución normal de las medias.

Para rechazar la hipótesis nula el nivel de riesgo α es 0.05. Cuando el p-valor del test sea menor que α se rechaza la hipótesis nula. Por lo tanto se consideraría la diferencia de las medianas como significativa. Consecuentemente se podrá decidir cual de los algoritmos comparados tiene la mayor bondad.

A partir de este test se puede agrupar aquellos algoritmos que no provocan rechazar la hipótesis nula, de manera que para cada problema de optimización se podrá presentar el rango de estas agrupaciones de algoritmos según su bondad.

Validación de la rapidez

Para comparar la rapidez de cada algoritmo en cada problema, se visualizan para cada problema los valores de evaluación en dos gráficos. Los gráficos tienen como eje horizontal el número de evaluaciones y como eje vertical el valor de evaluación. Por cada algoritmo hay una curva. En el primer gráfico las curvas representan los mejores valores de evaluación hasta el instante. En el segundo gráfico las curvas son los valores medianos de evaluación. Además en el segundo gráfico se muestran los cuartiles de los valores. Con esta información se puede averiguar el comportamiento de convergencia de los valores de evaluación de cada algoritmo.

5.2 Resultados experimentales

5.2.1 Resultados de la bondad de las soluciones

Cuadros de comparación

Se muestran los cuadros de comparación de mejores valores (5.5) y medianas (5.4).

Boxplots

Se muestran los *boxplots* de cada problema descartando *outliers*. En caso de no poder distinguir bien los cuartiles a causa de grandes diferencias entre los valores de diferentes algoritmos, se mostrara en la parte derecha otro *boxplot* de los algoritmos mejores.

Agrupación y rangos de algoritmos en cada problema

Las agrupaciones del cuadro 5.6 se forman si no se puede rechazar la hipótesis nula en el test de *Mann-Whitney* lo cual quiere decir que no existen evidencias significativas de que las medianas poblacionales se distinguen. Los p-valores de cada test se encuentran en el CD del proyecto.

En los rangos del cuadro 5.7 se tiene en cuenta estas agrupaciones. Si unos algoritmos forman una agrupación, entonces obtienen el mismo rango.

Cuadro 5.4: Medianas de los muestreos

	SGA	**GACI**	**GASA**	**GABBTO**	**GABBRA**
f_1	$2.55335 \cdot 10^{-1}$	$6.09575 \cdot 10^{1}$	$4.42857 \cdot 10^{-3}$	$5.07385 \cdot 10^{0}$	$7.41868 \cdot 10^{-2}$
f_2	$6.53391 \cdot 10^{-1}$	$8.91646 \cdot 10^{0}$	$7.23633 \cdot 10^{1}$	$3.11315 \cdot 10^{0}$	$2.50858 \cdot 10^{-1}$
f_3	$2.26780 \cdot 10^{3}$	$9.06293 \cdot 10^{3}$	$2.53414 \cdot 10^{4}$	$7.36426 \cdot 10^{3}$	$5.96194 \cdot 10^{3}$
f_4	$3.21813 \cdot 10^{0}$	$1.56420 \cdot 10^{1}$	$6.26383 \cdot 10^{0}$	$8.11627 \cdot 10^{0}$	$7.61578 \cdot 10^{0}$
f_5	$1.45868 \cdot 10^{2}$	$4.06732 \cdot 10^{3}$	$7.13204 \cdot 10^{1}$	$5.04978 \cdot 10^{2}$	$7.67511 \cdot 10^{1}$
f_6	$1.20000 \cdot 10^{1}$	$9.20000 \cdot 10^{1}$	$5.00000 \cdot 10^{0}$	$3.90000 \cdot 10^{1}$	$4.30000 \cdot 10^{1}$
f_7	$6.94214 \cdot 10^{-2}$	$3.34758 \cdot 10^{-1}$	$8.33285 \cdot 10^{-3}$	$1.99013 \cdot 10^{-1}$	$2.12739 \cdot 10^{-1}$
f_8	$2.38641 \cdot 10^{2}$	$4.94099 \cdot 10^{2}$	$2.27012 \cdot 10^{3}$	$1.07151 \cdot 10^{3}$	$1.12144 \cdot 10^{3}$
f_9	$1.66691 \cdot 10^{1}$	$3.59505 \cdot 10^{1}$	$5.37286 \cdot 10^{1}$	$3.96389 \cdot 10^{1}$	$3.96045 \cdot 10^{1}$
f_{10}	$1.04965 \cdot 10^{-1}$	$2.64858 \cdot 10^{0}$	$2.43802 \cdot 10^{-3}$	$1.01983 \cdot 10^{0}$	$4.52276 \cdot 10^{-2}$
f_{11}	$3.26117 \cdot 10^{-1}$	$1.44946 \cdot 10^{0}$	$8.84301 \cdot 10^{-2}$	$1.06552 \cdot 10^{0}$	$1.53835 \cdot 10^{-1}$
f_{12}	$1.04341 \cdot 10^{-1}$	$3.23792 \cdot 10^{0}$	$1.03670 \cdot 10^{-1}$	$1.05167 \cdot 10^{0}$	$1.03919 \cdot 10^{-1}$
f_{13}	$4.69403 \cdot 10^{-1}$	$7.37099 \cdot 10^{0}$	$4.97878 \cdot 10^{-1}$	$1.43577 \cdot 10^{0}$	$4.50581 \cdot 10^{-1}$
f_{14}	$9.98008 \cdot 10^{-1}$	$9.98004 \cdot 10^{-1}$	$1.01046 \cdot 10^{0}$	$9.98006 \cdot 10^{-1}$	$9.98005 \cdot 10^{-1}$
f_{15}	$8.63826 \cdot 10^{-3}$	$8.51442 \cdot 10^{-3}$	$8.87057 \cdot 10^{-3}$	$8.68313 \cdot 10^{-3}$	$8.61934 \cdot 10^{-3}$
f_{16}	$2.39766 \cdot 10^{-6}$	$5.64532 \cdot 10^{-8}$	$5.07233 \cdot 10^{-4}$	$1.94011 \cdot 10^{-4}$	$2.26166 \cdot 10^{-5}$
f_{17}	$3.98705 \cdot 10^{-1}$	$3.97907 \cdot 10^{-1}$	$4.01820 \cdot 10^{-1}$	$3.99880 \cdot 10^{-1}$	$4.00936 \cdot 10^{-1}$
f_{18}	$3.00006 \cdot 10^{0}$	$3.00000 \cdot 10^{0}$	$3.01469 \cdot 10^{0}$	$3.00991 \cdot 10^{0}$	$3.00164 \cdot 10^{0}$
f_{19}	$7.34051 \cdot 10^{-3}$	$7.26051 \cdot 10^{-3}$	$7.55039 \cdot 10^{-3}$	$7.86630 \cdot 10^{-3}$	$8.16493 \cdot 10^{-3}$
f_{20}	$5.00518 \cdot 10^{-2}$	$1.33627 \cdot 10^{-1}$	$1.39503 \cdot 10^{-1}$	$1.34008 \cdot 10^{-1}$	$1.32447 \cdot 10^{-1}$
f_{21}	$5.94502 \cdot 10^{0}$	$8.31717 \cdot 10^{0}$	$5.68821 \cdot 10^{0}$	$3.95058 \cdot 10^{0}$	$1.00533 \cdot 10^{0}$
f_{22}	$5.99290 \cdot 10^{-1}$	$8.24809 \cdot 10^{0}$	$5.95855 \cdot 10^{0}$	$1.29876 \cdot 10^{0}$	$6.34662 \cdot 10^{-1}$
f_{23}	$4.66716 \cdot 10^{-1}$	$8.12890 \cdot 10^{0}$	$6.20070 \cdot 10^{0}$	$5.94329 \cdot 10^{-1}$	$4.99777 \cdot 10^{-1}$
JHRR	$1.04600 \cdot 10^{1}$	$1.28000 \cdot 10^{1}$	$3.64000 \cdot 10^{0}$	$8.42000 \cdot 10^{0}$	$8.68000 \cdot 10^{0}$
PFSP	$4.03900 \cdot 10^{3}$	$4.05600 \cdot 10^{3}$	$4.04200 \cdot 10^{3}$	$4.12400 \cdot 10^{3}$	$4.06900 \cdot 10^{3}$
CKBD	$1.15200 \cdot 10^{3}$	$1.10100 \cdot 10^{3}$	$1.17800 \cdot 10^{3}$	$1.13400 \cdot 10^{3}$	$1.18800 \cdot 10^{3}$

Cuadro 5.5: Mejores valores de los muestreos

	SGA	**GACI**	**GASA**	**GABBTO**	**GABBRA**
f_1	$1.30421 \cdot 10^{-1}$	$1.51481 \cdot 10^{1}$	$6.98513 \cdot 10^{-5}$	$1.64721 \cdot 10^{0}$	$7.29713 \cdot 10^{-3}$
f_2	$3.61333 \cdot 10^{-1}$	$4.81849 \cdot 10^{0}$	$1.26378 \cdot 10^{1}$	$1.13832 \cdot 10^{0}$	$1.00099 \cdot 10^{-1}$
f_3	$1.12460 \cdot 10^{3}$	$5.71526 \cdot 10^{3}$	$1.70802 \cdot 10^{4}$	$4.02110 \cdot 10^{3}$	$3.34394 \cdot 10^{3}$
f_4	$2.32700 \cdot 10^{0}$	$9.31563 \cdot 10^{0}$	$1.57625 \cdot 10^{0}$	$5.59243 \cdot 10^{0}$	$4.12756 \cdot 10^{0}$
f_5	$6.27755 \cdot 10^{1}$	$1.12859 \cdot 10^{3}$	$2.38338 \cdot 10^{0}$	$2.51759 \cdot 10^{2}$	$1.80543 \cdot 10^{1}$
f_6	$0.00000 \cdot 10^{0}$	$2.20000 \cdot 10^{1}$	$0.00000 \cdot 10^{0}$	$1.50000 \cdot 10^{1}$	$9.00000 \cdot 10^{0}$
f_7	$2.33298 \cdot 10^{-2}$	$1.56407 \cdot 10^{-1}$	$7.24649 \cdot 10^{-4}$	$9.75692 \cdot 10^{-2}$	$8.94234 \cdot 10^{-2}$
f_8	$1.43804 \cdot 10^{0}$	$4.11241 \cdot 10^{1}$	$8.29103 \cdot 10^{2}$	$4.61396 \cdot 10^{2}$	$4.84369 \cdot 10^{2}$
f_9	$6.22735 \cdot 10^{0}$	$2.36192 \cdot 10^{1}$	$3.48235 \cdot 10^{1}$	$1.74412 \cdot 10^{1}$	$2.11675 \cdot 10^{1}$
f_{10}	$4.97160 \cdot 10^{-2}$	$1.14065 \cdot 10^{0}$	$1.96585 \cdot 10^{-3}$	$2.69578 \cdot 10^{-1}$	$1.59550 \cdot 10^{-2}$
f_{11}	$1.38592 \cdot 10^{-1}$	$1.15751 \cdot 10^{0}$	$1.68048 \cdot 10^{-4}$	$9.73039 \cdot 10^{-1}$	$4.69525 \cdot 10^{-2}$
f_{12}	$1.17551 \cdot 10^{-3}$	$9.00984 \cdot 10^{-1}$	$4.41288 \cdot 10^{-8}$	$3.22912 \cdot 10^{-2}$	$7.90931 \cdot 10^{-6}$
f_{13}	$2.55733 \cdot 10^{-1}$	$3.60316 \cdot 10^{0}$	$2.49565 \cdot 10^{-1}$	$6.09298 \cdot 10^{-1}$	$2.52815 \cdot 10^{-1}$
f_{14}	$9.98003 \cdot 10^{-1}$	$9.98003 \cdot 10^{-1}$	$9.98003 \cdot 10^{-1}$	$9.98003 \cdot 10^{-1}$	$9.98003 \cdot 10^{-1}$
f_{15}	$8.46483 \cdot 10^{-3}$	$8.45922 \cdot 10^{-3}$	$8.75794 \cdot 10^{-3}$	$8.48598 \cdot 10^{-3}$	$8.45943 \cdot 10^{-3}$
f_{16}	$5.64532 \cdot 10^{-8}$	$5.64532 \cdot 10^{-8}$	$2.87924 \cdot 10^{-5}$	$8.22257 \cdot 10^{-7}$	$5.64532 \cdot 10^{-8}$
f_{17}	$3.97892 \cdot 10^{-1}$	$3.97887 \cdot 10^{-1}$	$3.97921 \cdot 10^{-1}$	$3.97911 \cdot 10^{-1}$	$3.98107 \cdot 10^{-1}$
f_{18}	$3.00000 \cdot 10^{0}$	$3.00000 \cdot 10^{0}$	$3.00025 \cdot 10^{0}$	$3.00060 \cdot 10^{0}$	$3.00007 \cdot 10^{0}$
f_{19}	$7.21981 \cdot 10^{-3}$	$7.22046 \cdot 10^{-3}$	$7.22166 \cdot 10^{-3}$	$7.26011 \cdot 10^{-3}$	$7.23884 \cdot 10^{-3}$
f_{20}	$1.09141 \cdot 10^{-2}$	$1.17807 \cdot 10^{-2}$	$1.53121 \cdot 10^{-2}$	$1.74217 \cdot 10^{-2}$	$2.04904 \cdot 10^{-2}$
f_{21}	$8.46887 \cdot 10^{-1}$	$8.31714 \cdot 10^{0}$	$8.47846 \cdot 10^{-1}$	$8.59182 \cdot 10^{-1}$	$8.47462 \cdot 10^{-1}$
f_{22}	$5.97113 \cdot 10^{-1}$	$8.23412 \cdot 10^{0}$	$6.11675 \cdot 10^{-1}$	$6.05536 \cdot 10^{-1}$	$5.98274 \cdot 10^{-1}$
f_{23}	$4.63733 \cdot 10^{-1}$	$8.12885 \cdot 10^{0}$	$4.70636 \cdot 10^{-1}$	$4.69180 \cdot 10^{-1}$	$4.68417 \cdot 10^{-1}$
JHRR	$1.06800 \cdot 10^{1}$	$1.28000 \cdot 10^{1}$	$4.38000 \cdot 10^{0}$	$9.72000 \cdot 10^{0}$	$9.72000 \cdot 10^{0}$
PFSP	$3.97800 \cdot 10^{3}$	$4.00000 \cdot 10^{3}$	$3.96600 \cdot 10^{3}$	$4.06200 \cdot 10^{3}$	$4.03000 \cdot 10^{3}$
CKBD	$1.19300 \cdot 10^{3}$	$1.13500 \cdot 10^{3}$	$1.22600 \cdot 10^{3}$	$1.16900 \cdot 10^{3}$	$1.21400 \cdot 10^{3}$

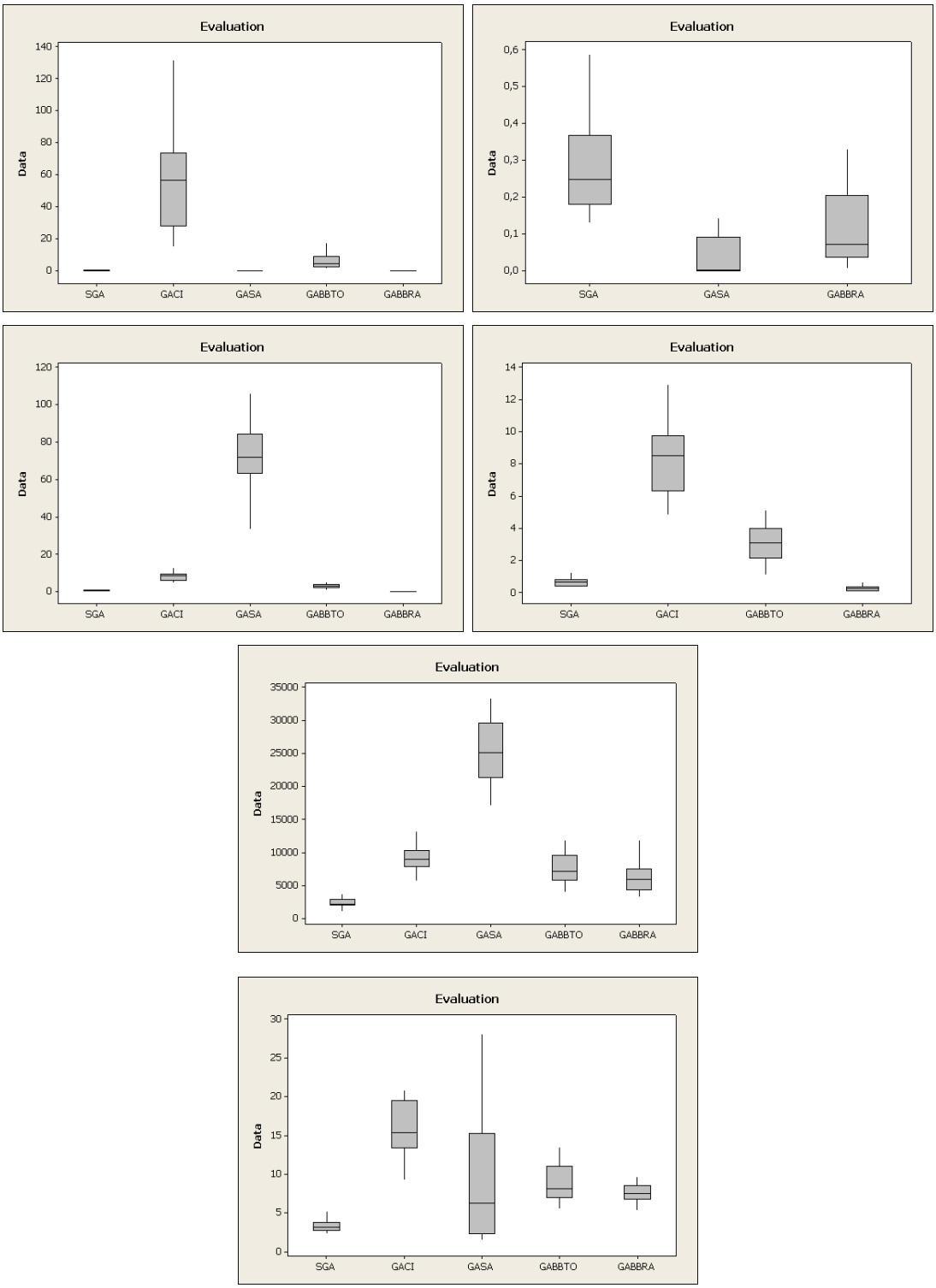

Figura 5.1: Funciones $f_1 - f_4$

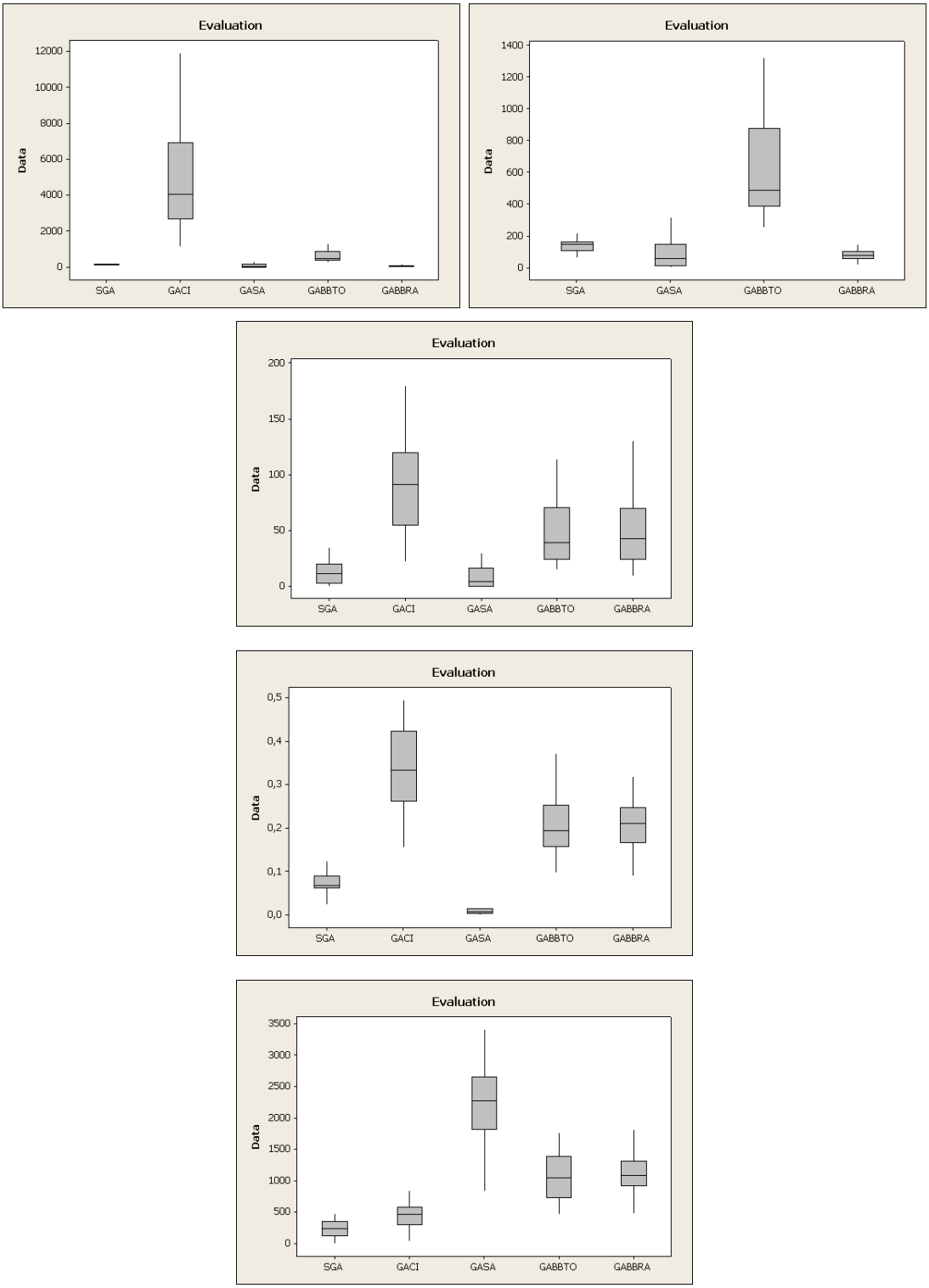

Figura 5.2: Funciones $f_5 - f_8$

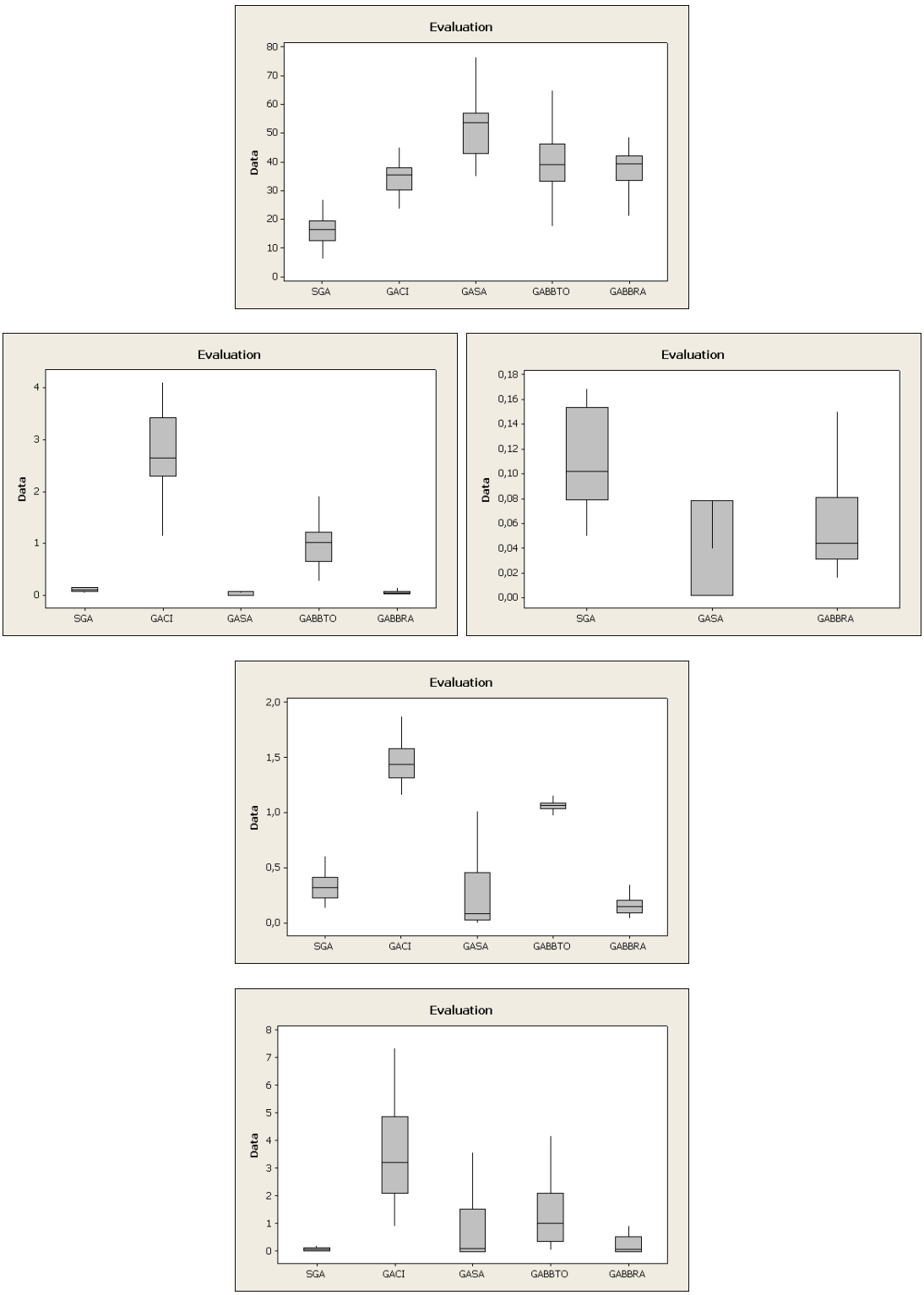

Figura 5.3: Funciones $f_9 - f_{12}$

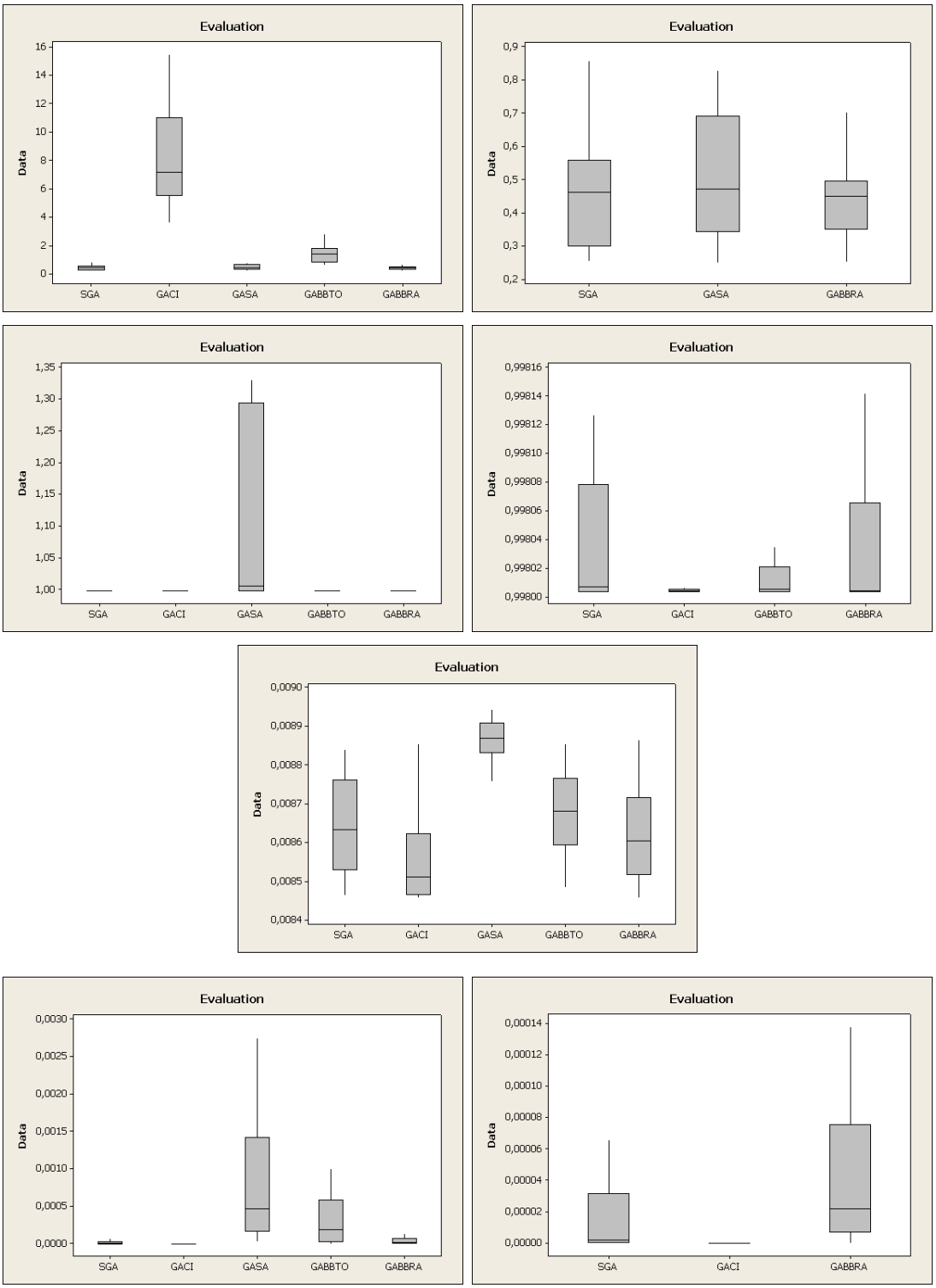

Figura 5.4: Funciones $f_{13} - f_{16}$

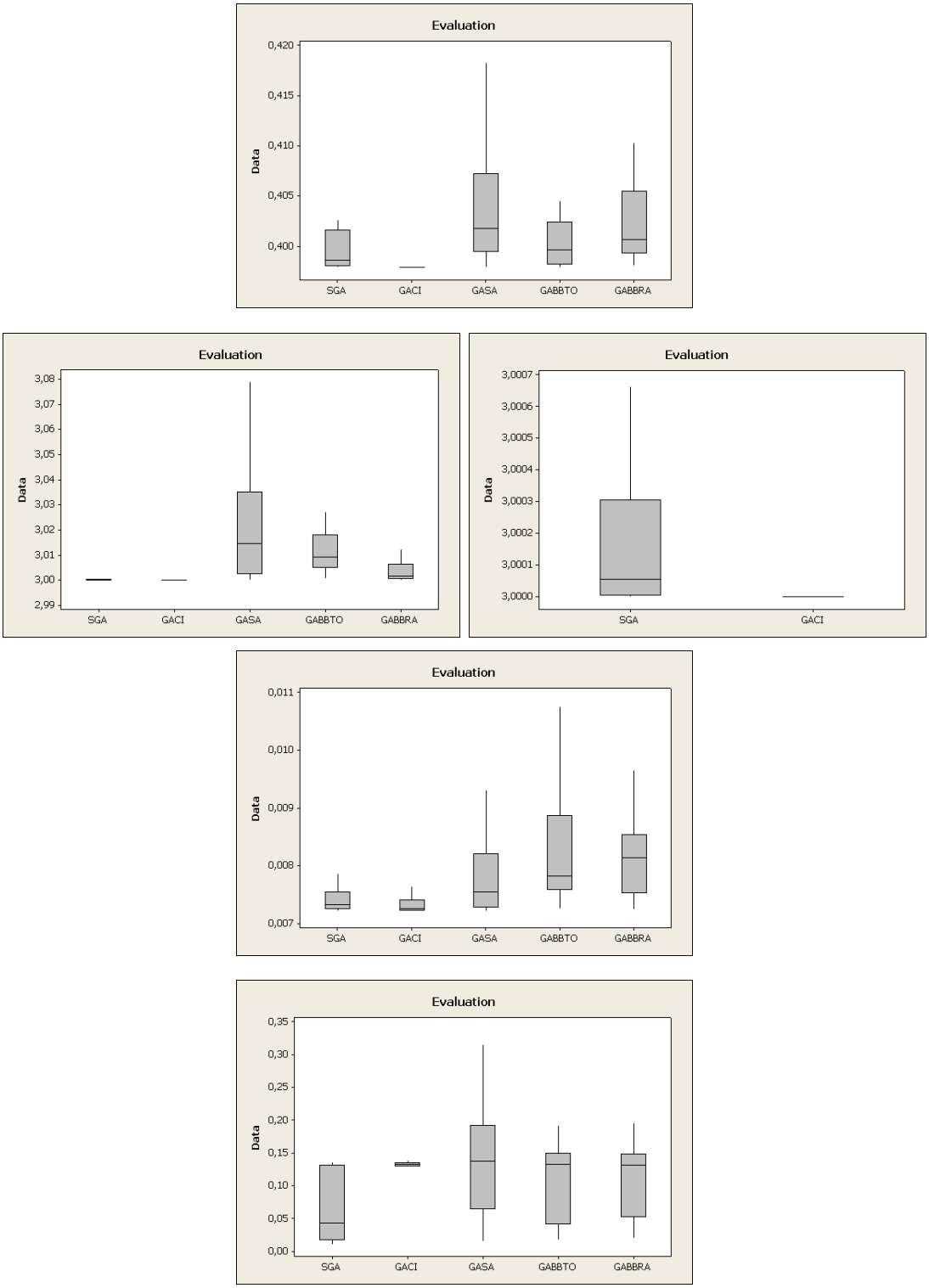

Figura 5.5: Funciones $f_{17} - f_{20}$

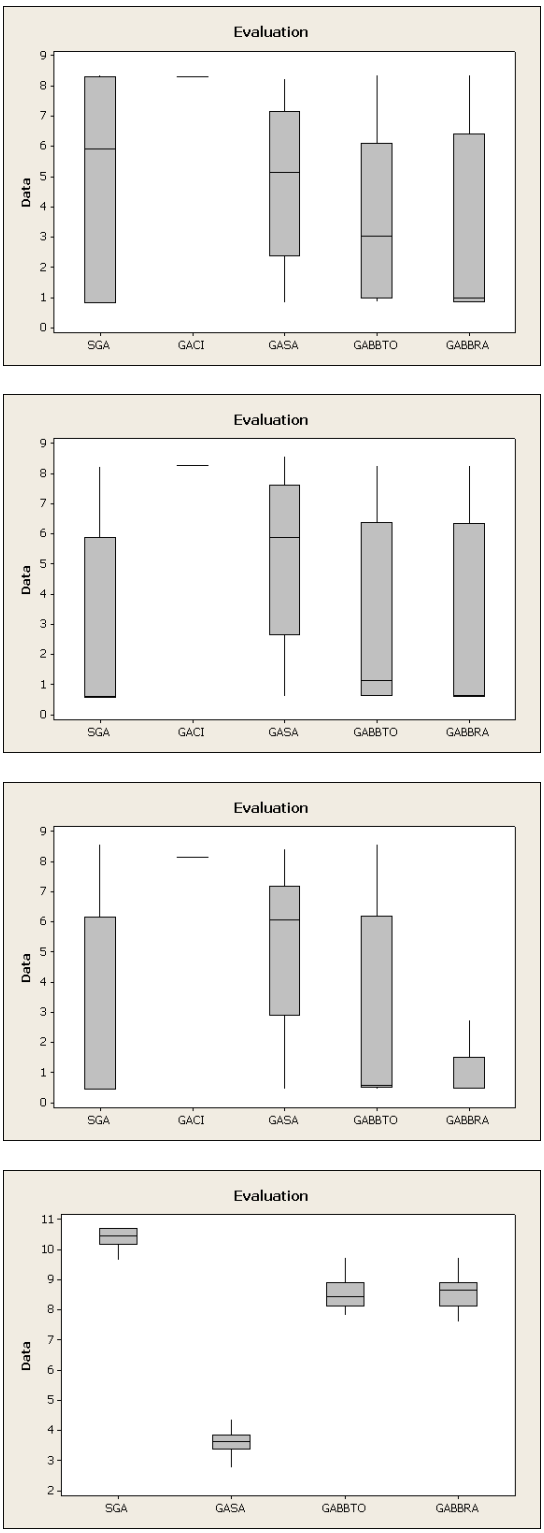

Figura 5.6: Funciones $f_{21} - f_{23}$ y John Holland Royal Road

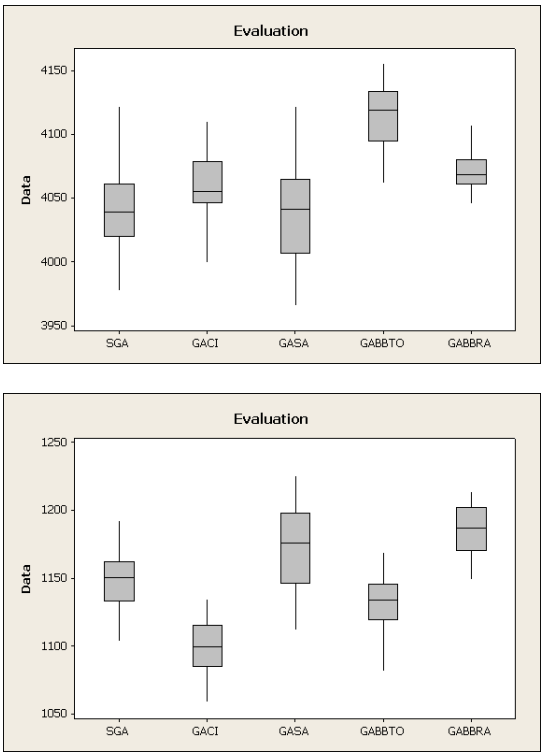

Figura 5.7: Permutation Flowshop y Checkerboard

Cuadro 5.6: Agrupaciones en orden de mejor bondad a partir de *Mann-Whitney*

	Agrupaciones
f_1	{GASA} {GABBRA} {SGA} {GABBTO} {GACI}
f_2	{GABBRA} {SGA} {GABBTO} {GACI} {GASA}
f_3	{SGA} {GABBRA} {GABBTO} {GACI} {GASA}
f_4	{SGA} {GASA,GABBRA} {GABBTO,GABBRA} {GACI}
f_5	{GASA,GABBRA} {SGA} {GABBTO} {GACI}
f_6	{SGA,GASA} {GABBTO,GABBRA} {GACI}
f_7	{GASA} {SGA} {GABBTO,GABBRA} {GACI}
f_8	{SGA} {GACI} {GABBTO,GABBRA} {GASA}
f_9	{SGA} {GACI} {GABBTO,GABBRA} {GASA}
f_{10}	{GASA} {GABBRA} {SGA} {GABBTO} {GACI}
f_{11}	{GASA,GABBRA} {SGA} {GABBTO} {GACI}
f_{12}	{SGA,GASA,GABBRA} {GABBTO} {GACI}
f_{13}	{SGA,GASA,GABBRA} {GABBTO} {GACI}
f_{14}	{SGA,GACI,GABBTO,GABBRA} {GASA}
f_{15}	{GACI} {SGA,GABBTO,GABBRA} {GASA}
f_{16}	{GACI} {SGA} {GABBRA} {GABBTO} {GASA}
f_{17}	{GACI} {SGA,GABBTO} {GASA,GABBTO,GABBRA}
f_{18}	{GACI} {SGA} {GABBRA} {GASA,GABBTO}
f_{19}	{SGA,GACI} {GASA,GABBRA} {GABBTO,GABBRA}
f_{20}	{SGA} {GACI,GASA,GABBTO,GABBRA}
f_{21}	{GABBRA,GABBTO,SGA} {SGA,GASA} {SGA,GACI}
f_{22}	{SGA} {GABBRA} {GABBTO} {GASA} {GACI}
f_{23}	{SGA} {GABBRA} {GABBTO} {GASA} {GACI}
JHRR	{GACI} {SGA} {GABBTO,GABBRA} {GASA}
PFSP	{SGA,GASA} {GACI,GASA} {GABBRA} {GABBTO}
CKBD	{GASA,GABBRA} {SGA} {GABBTO} {GACI}

Cuadro 5.7: Rangos en cada problema

	SGA	**GACI**	**GASA**	**GABB TO**	**GABB RA**
f_1	3	5	1	4	2
f_2	2	4	5	3	1
f_3	1	4	5	3	2
f_4	1	5	2	4	2
f_5	3	5	1	4	1
f_6	1	5	1	3	3
f_7	2	5	1	3	3
f_8	1	2	5	3	3
f_9	1	2	5	3	3
f_{10}	3	5	1	4	2
f_{11}	3	5	1	4	1
f_{12}	1	5	1	4	1
f_{13}	1	5	1	4	1
f_{14}	1	1	5	1	1
f_{15}	2	1	5	2	2
f_{16}	2	1	5	4	3
f_{17}	2	1	4	2	4
f_{18}	2	1	4	4	3
f_{19}	1	1	3	5	3
f_{20}	1	2	2	2	2
f_{21}	1	5	4	1	1
f_{22}	1	5	4	3	2
f_{23}	1	5	4	3	2
JHRR	2	1	5	3	3
PFSP	1	3	1	5	4
CKBD	3	5	1	4	1

5.2.2 Resultados del comportamiento de convergencia

Se muestran los gráficos para comparar el comportamiento de convergencia de cada algoritmo en cada problema. En la parte izquierda de cada página los gráficos muestran las curvas de los mejores (de los 30) valores de evaluación de cada algoritmo hasta el instante. En la parte derecha se muestran las curvas de las medianas con cuartiles. El CD del proyecto contiene los ficheros de los gráficos.

La asignación de los colores de las curvas a los algoritmos es la siguiente:

Algoritmo	**Color**
SGA	negro
GACI	verde
GASA	rojo
GABBTO	azul
GABBRA	amarillo

La escala del eje vertical es logarítmica (en base 10) para las funciones $f_1 - f_{23}$.

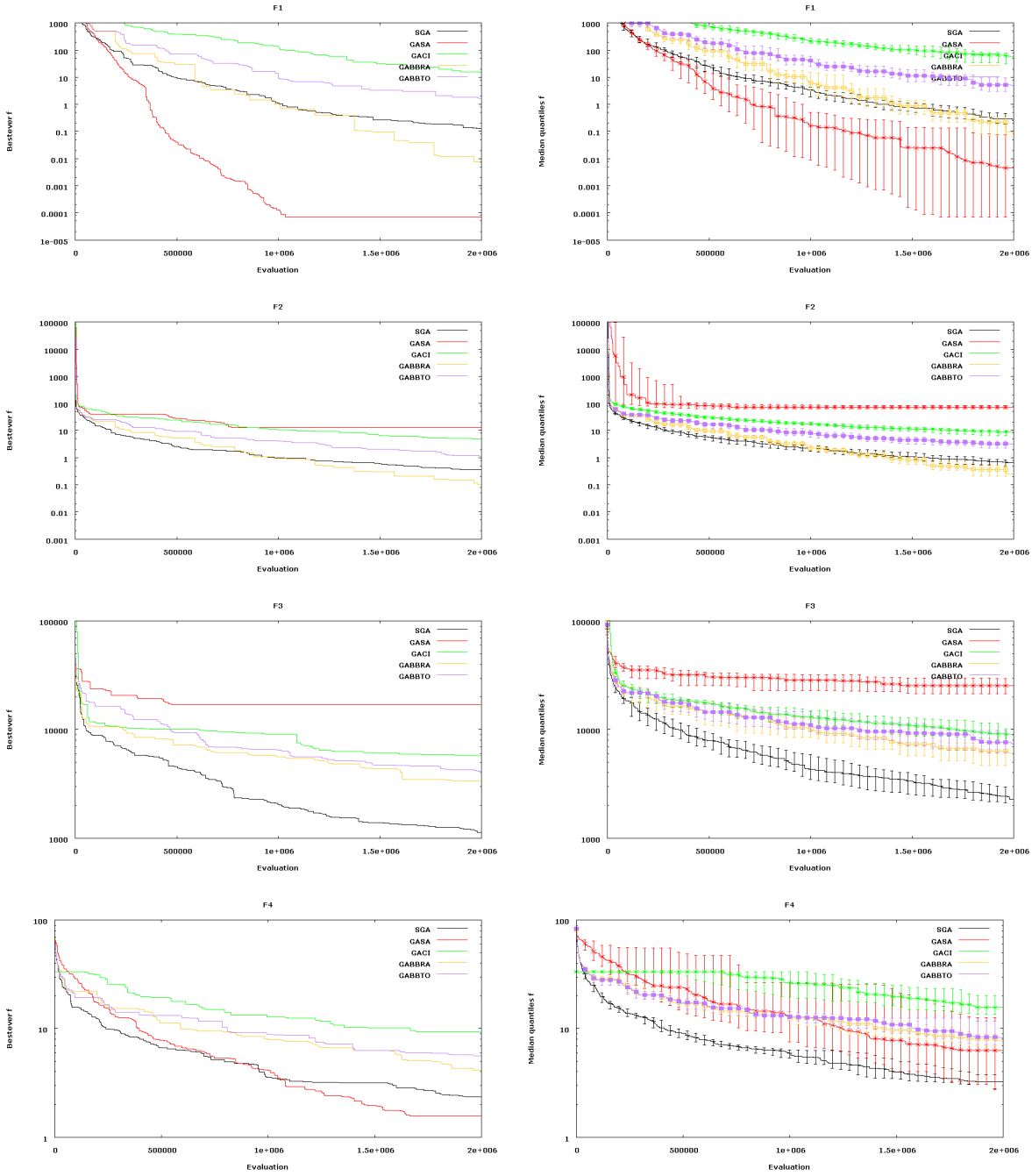

Figura 5.8: Funciones $f_1 - f_4$

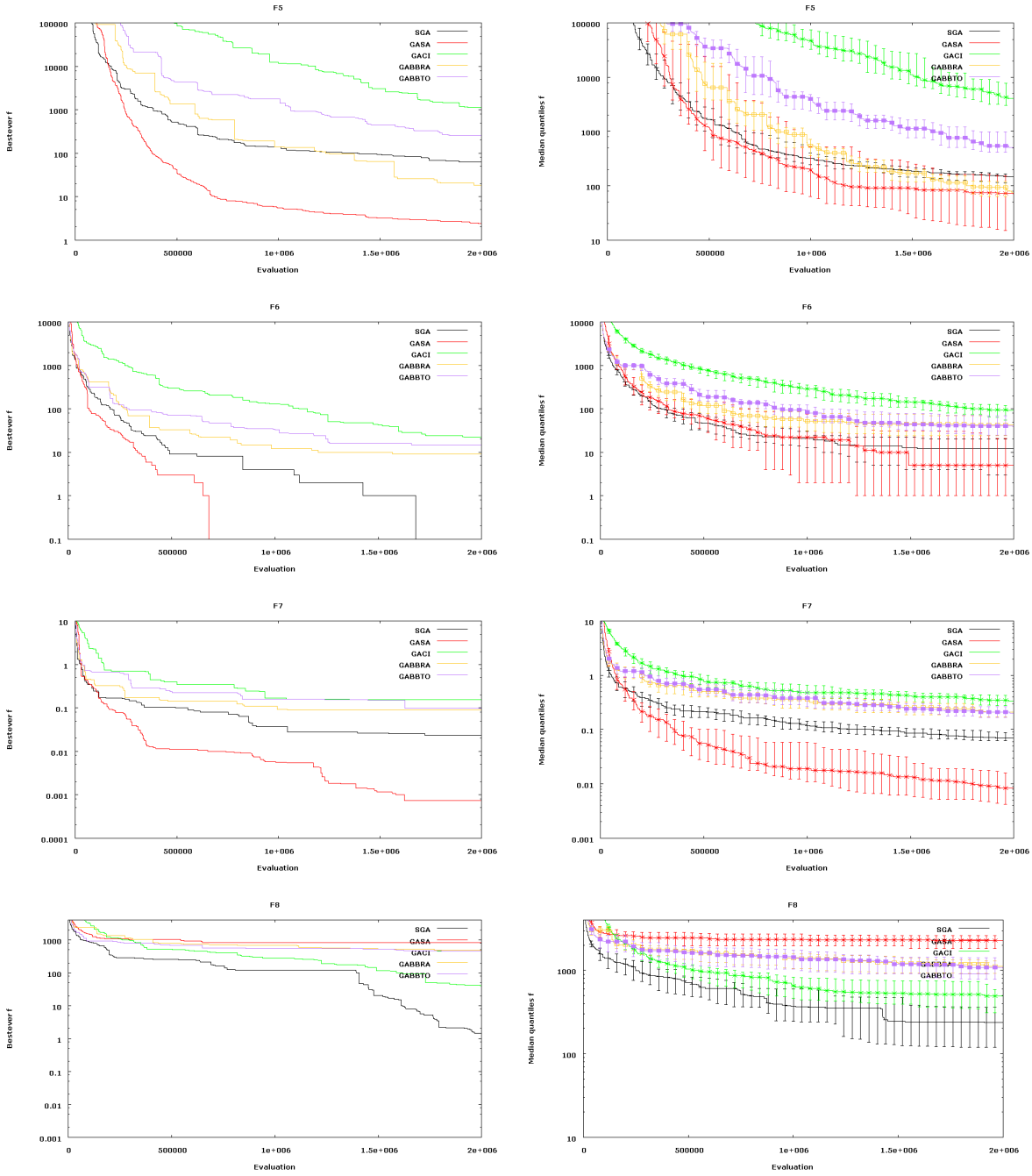

Figura 5.9: Funciones $f_5 - f_8$

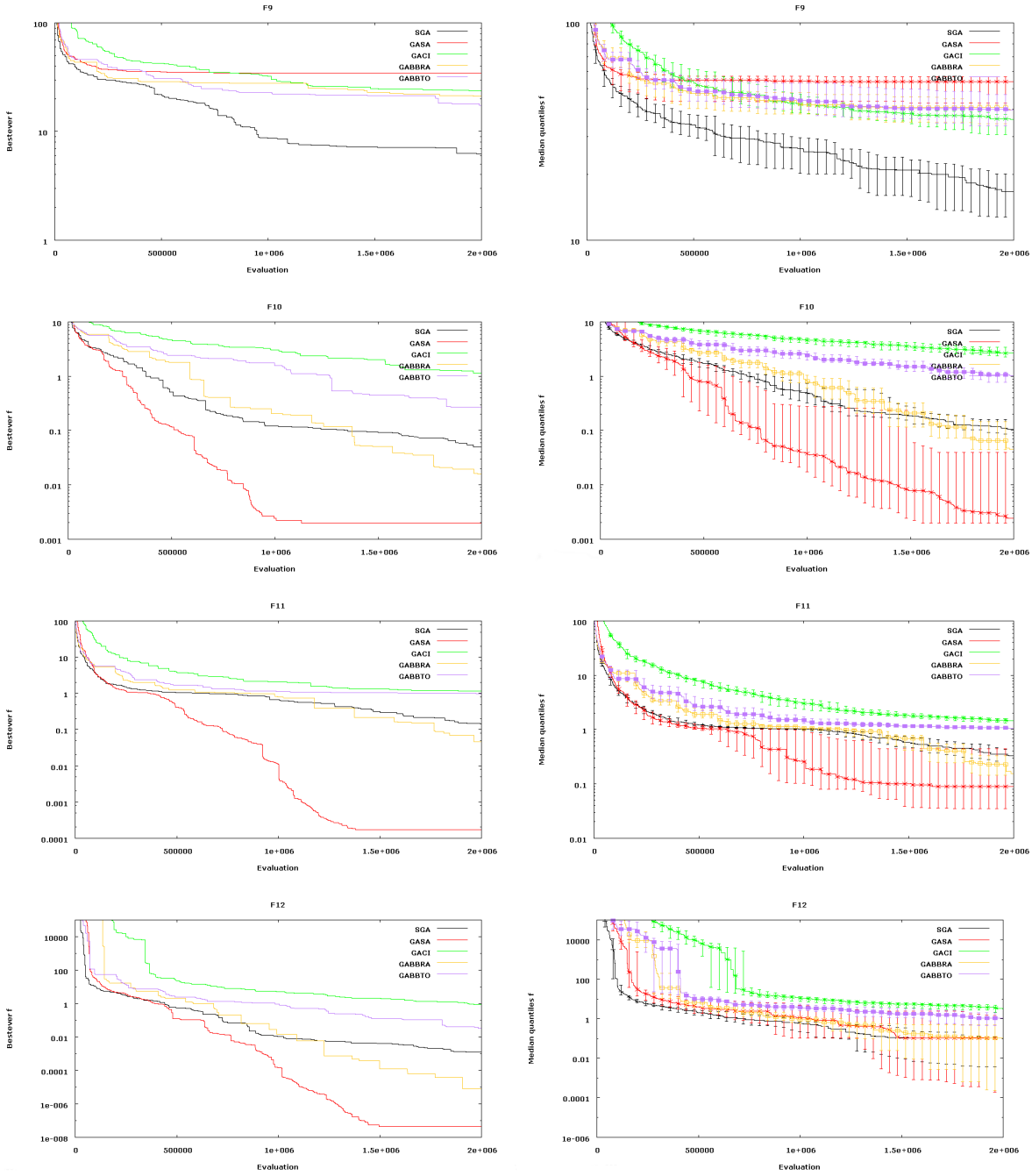

Figura 5.10: Funciones $f_9 - f_{12}$

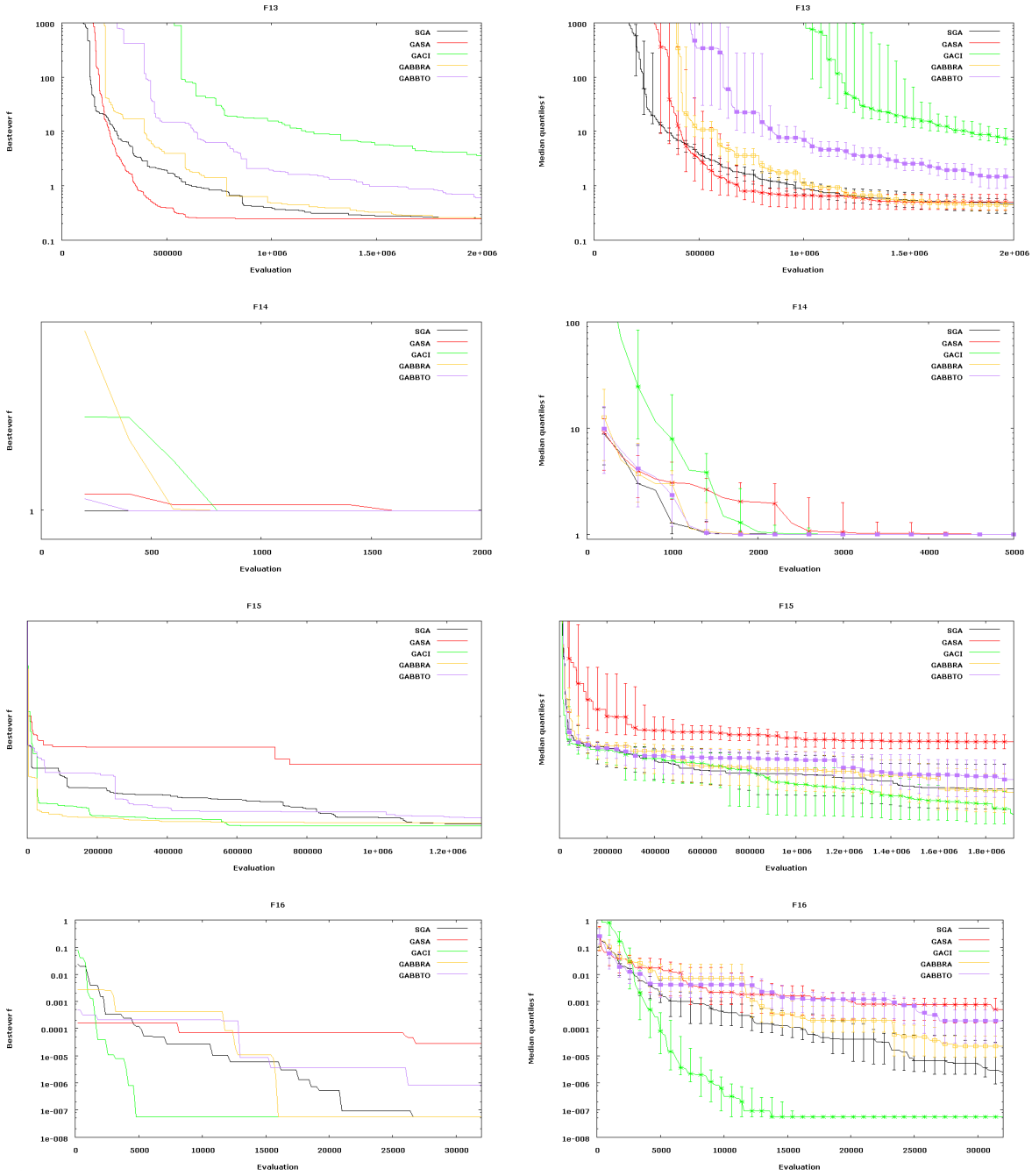

Figura 5.11: Funciones $f_{13} - f_{16}$

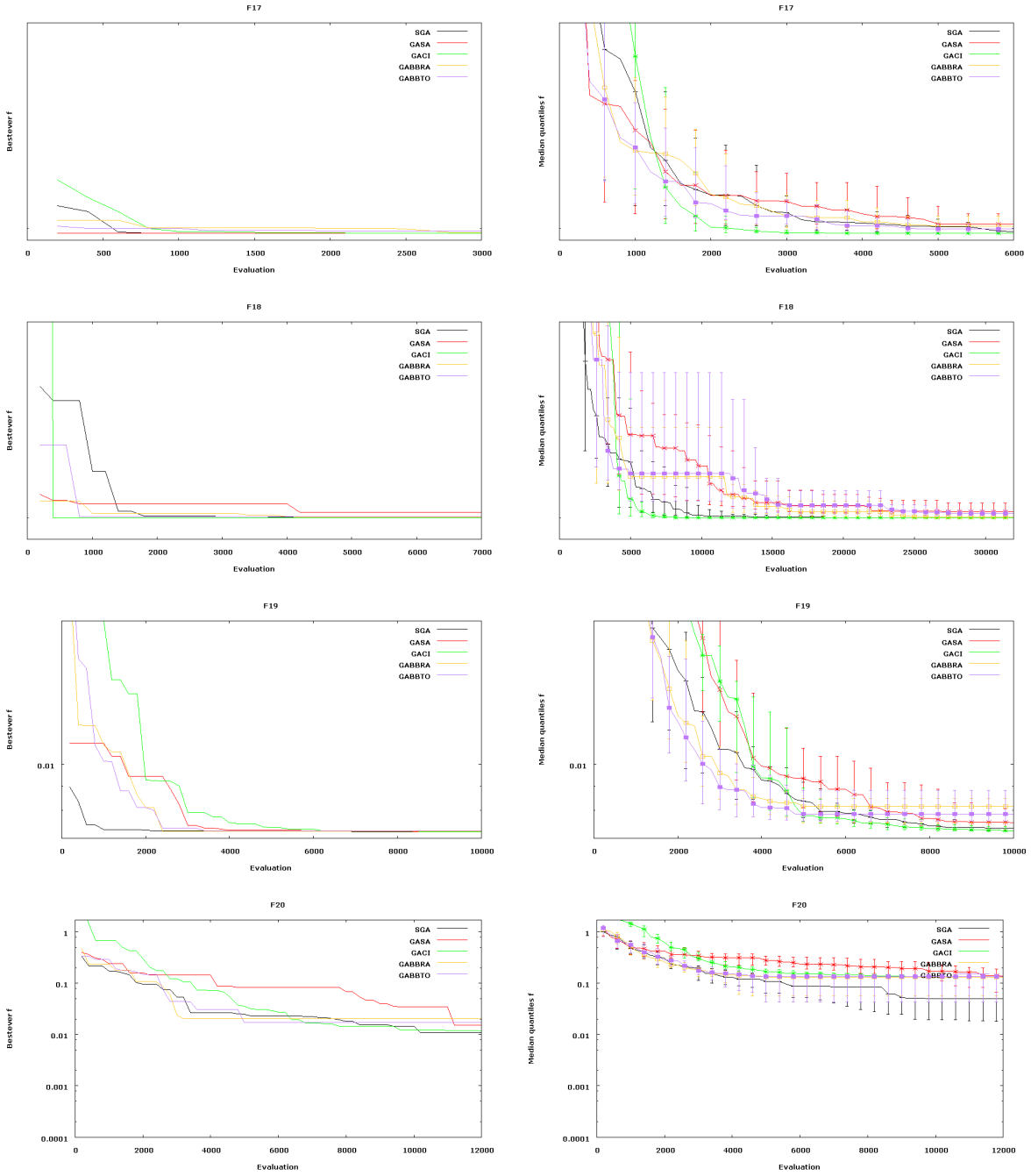

Figura 5.12: Funciones $f_{17} - f_{20}$

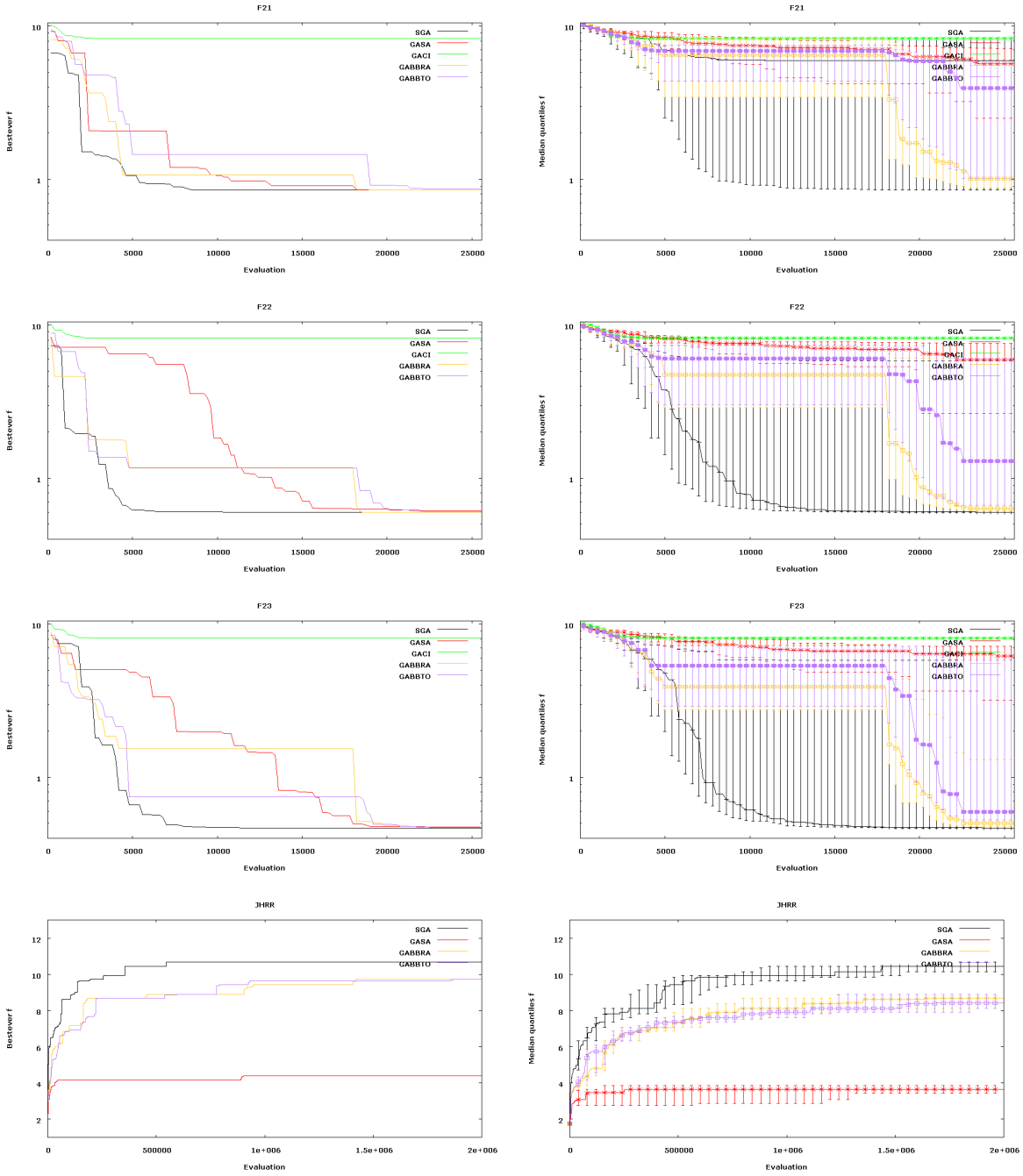

Figura 5.13: Funciones $f_{21} - f_{23}$ y John Holland Royal Road

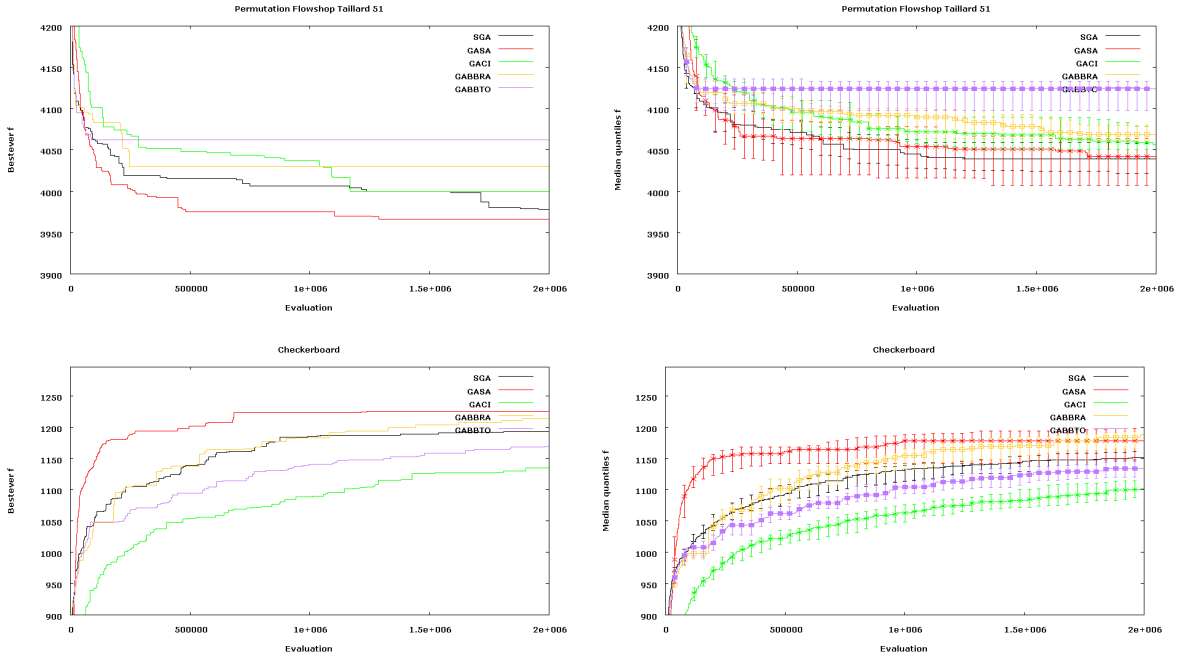

Figura 5.14: Permutation Flowshop y Checkerboard

5.3 Discusión de resultados

5.3.1 Observaciones generales

- De todos los algoritmos estudiados no aparece ninguno que sea superior en todos los problemas del experimento. A veces hay varios ganadores en cuanto a los valores de evaluación.
- El algoritmo GABBTO tiene un rendimiento bajo en casi todos los problemas.
- Los valores del algoritmo GASA suelen variar más que los demás. Posiblemente en las diferentes ejecuciones las probabilidades de variación tardan más en un caso que en otro en adaptarse.
- En el GABBRA los gráficos muestran escalones. Esto ocurre cuando se tiene que hacer varias evaluaciones para detectar bloques constructivos y se cambian los bits fuera de los bloques, lo cual puede favorecer la bondad.

5.3.2 Observaciones de la bondad

Funciones unimodales

- No hay un ganador consistente.
- Aunque el GASA sea superior en la mayoría de los problemas, en otros es el peor (f_2 y f_3). Esto también se observa en las demás clases de problemas.
- El GACI es bastante inferior. A causa de las dimensiones medias de estas funciones, el tamaño poblacional del GACI aumenta exageradamente (véase 3.1). Se necesitan muchas más evaluaciones en cada generación que en los otros algoritmos. Aunque la intención del GACI es facilitar el encuentro de bloques constructivos mediante su inicialización canónica, el coste resultante de tantas evaluaciones le perjudica más a lo largo de su transcurso y no vale la pena, sobre todo en funciones unimodales que no son difíciles de resolver por los demás algoritmos. Además el tamaño poblacional del GACI solamente se asigna a causa de su inicialización. Después de varias generaciones, cuando ya se hayan desarrollado los bloques, el tamaño poblacional pierde su sentido al permanecer, ya que la población contendrá muchos individuos redundantes muy similares entre sí. Esto ya se pudo observar en la fase de *pretesting*, donde - a partir de un cierto tamaño poblacional - ya no mejoraba el rendimiento.
- El GABBTO y el GABBRA tienen el mismo tamaño poblacional que el SGA, pero para la detección de bloques constructivos, en la media también necesitan muchas más evaluaciones por generación que el SGA.

Funciones multimodales con muchos mínimos locales

- Se observan resultados similares a las unimodales.
- En f_8 y f_9 el GACI tiene resultados buenos, aunque inferiores al SGA. Quizás para estas funciones conviene tener muchos individuos en la población. Esta suposición se confirma si también se contempla el comportamiento de los algoritmos en el gráfico que muestra los valores de evaluación por generación.

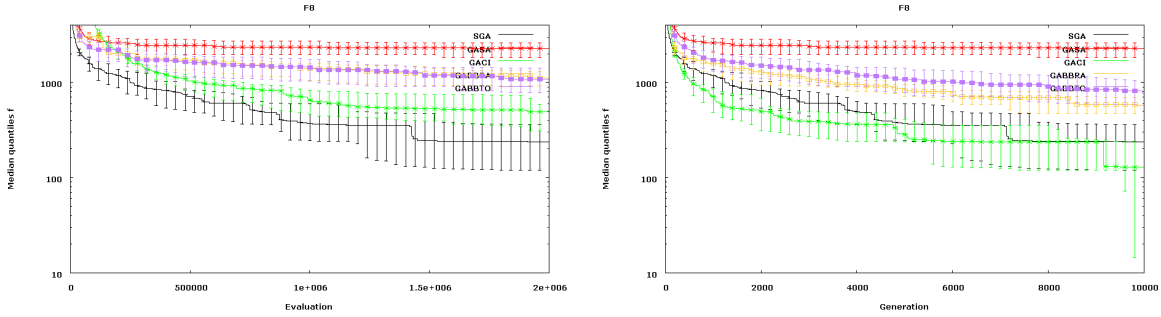

Figura 5.15: Función f_8. En la parte izquierda están los valores por evaluación, y en la derecha por generación.

Podemos observar en figura 5.15 que en f_8 el GACI incluso supera al SGA si comparamos los valores por generación.

Funciones multimodales con pocos mínimos locales

- Son de una dimensión baja. Esto provoca que la población del GACI tenga un tamaño adecuado aún después de haberse desarrollado los bloques a partir de su inicialización. En $f_{14} - f_{19}$ tiene un rendimiento bastante mayor a los demás. En $f_{22} - f_{23}$ se observa lo contrario. En esta familia de problemas los algoritmos modificados son peores que el SGA.

Problemas de optimización combinatoria

- El GACI gana en JHRR ya que su inicialización casualmente siempre contiene el óptimo global. Los demás algoritmos modificados son inferiores al SGA. Posiblemente la razón es que el JHRR es un problema altamente engañoso (véase [Gol89, "deceptive problems"]). Entonces los bloques detectados y apoyados por la máscara de cortes se ven "engañados" por el aspecto engañoso de la función, es decir, los bloques no conducen más rápido el algoritmo hacia el óptimo.
- Por lo contrario, el CKBD favorece los bloques. Consecuentemente el GASA y el GABBRA dan mejores resultados que el SGA.

- El bajo rendimiento de los algoritmos modificados en el problema PFSP puede resultar de la codificación utilizada. La búsqueda o el intercambio de bloques parece inútil ya que después de una recombinación, los valores decodificados en general ya no son los mismos que antes, es decir, los identificadores de las tareas permutadas cambian de manera absurda.

5.3.3 Observaciones del comportamiento de convergencia

- En los casos donde el GASA comparte su superioridad de la bondad, se suele observar que encuentra antes mejores soluciones que los demás algoritmos aunque estos al final consiguen una bondad similar. Esto ocurre p. ej. con f_5, f_{11}, f_{13} y CKBD.

- También en f_{10} y f_{12} el GASA encuentra muy rápido soluciones buenas.

- Mientras que otros converjan el GASA puede adaptar su distancia de paso por las probabilidades de variación. Esto puede resultar útil en funciones como f_6, donde hacen falta grandes pasos para salir de un óptimo local.

- Aunque las medianas del GASA en f_4, f_6, f_{11} y f_{12} son inferiores o iguales que las de otro algoritmo, su mejor valor es superior.

- En los gráficos de f_5, f_{10}, f_{11} y f_{12} se puede observar que al principio el SGA tiene un mejor rendimiento que GABBRA pero bastante más tarde el GABBRA le supera cuando el SGA ya casi converge. Parece ser que en algunos casos, en generaciones elevadas el intercambio de bloques favorece el rendimiento.

- Respecto al GACI se observa que en los problemas que domina, suele encontrar soluciones buenas muy rápido.

6 Conclusiones

Objetivos conseguidos

Se han conseguido los objetivos del estudio. Se ha diseñado e implementado un entorno software para experimentar con los diferentes algoritmos genéticos y luego compararlos al analizar los resultados. Este entorno permite modificar y añadir fácilmente nuevas componentes de cualquier algoritmo evolutivo. Para la experimentación se han usado problemas típicos de *test* con diferentes características en todos los algoritmos del estudio, que se centran en aprovechar bloques constructivos. Luego se comparó el rendimiento de la bondad de las soluciones de cada algoritmo. Además se ha tenido en cuenta el comportamiento de convergencia. En el análisis de los resultados se ha observado que algunos de los algoritmos estudiados dan mejores resultados en determinados casos que se han intentado destacar.

Trabajo futuro

En el análisis también se ha intentado encontrar posibles causas para explicar el comportamiento y los resultados de cada algoritmo. A partir de ellas se puede proporcionar unas posibles mejoras para los algoritmos modificados. Se debe modificar el GACI tal que durante el transcurso de la optimización su tamaño poblacional se aproxime a un valor razonable. En los GABB se debe detectar los bloques realizando menos evaluaciones. P. ej. se puede estimar valores de evaluación en ciertos bits o aplicar algún modelo de minería de datos. Para ello se puede estudiar, p. ej. , cómo se comportan los bits de la máscara de cortes. Otras posibles mejoras serían la incorporación de un vector de *fitness* (en vez de el *fitness* solo) para problemas de optimización con varios objetivos. También se puede mejorar algo el rendimiento del sistema optimizando algunos operadores como la selección, o incorporando *multi-threading* evaluando los individuos en varios flujos.

Beneficio personal

El proyecto me ayudó a aprender mucho sobre algoritmos evolutivos. También me hizo profundizar conocimientos sobre ingeniería de software, experimentación y programación

en C++. Al tener que implementar la interfaz gráfica de usuario he aprendido programar con la librería Qt.

A Manual de usuario

A.1 GUI

Para utilizar el sistema mediante el interfaz gráfico de usuario, hay que llamar al programa sin parámetros, es decir, introduciendo *algevo*. Entonces aparece la ventana de figura A.1.

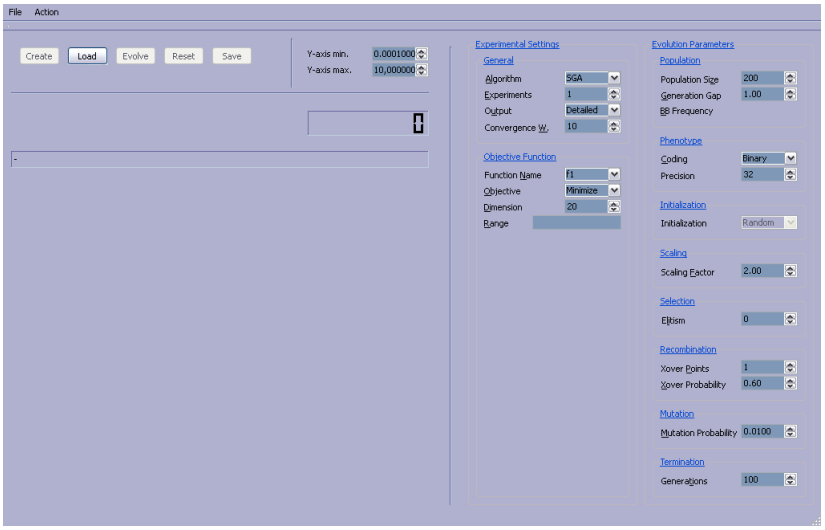

Figura A.1: Creación de un experimento

El programa está listo para crear o cargar un experimento. En la parte derecha están los parámetros para especificar un experimento. En la parte izquierda hay un espacio donde aparecerá un gráfico mostrando la media de los valores de evaluación por generación. Además se muestra arriba el mejor valor de evaluación jamás encontrado. Arriba a la izquierda hay unos botones que tienen el siguiente significado:

- **Create**
  Se crea un experimento a partir de los parámetros especificados en la parte derecha de la ventana. Al crear un experimento se pedirá al usuario un nombre de fichero para el mismo.

- **Load**
  Se carga un experimento desde un fichero. Se pedirá al usuario especificar el nombre del fichero (figura A.2).

- **Evolve**
  Se comienza o sigue con el experimento (figura A.3). Si el experimento se está ejecutando, entonces con este botón se pausa el experimento.

- **Reset**
  Se inicializa un experimento pausado.

- **Save**
  Se guarda el experimento. Si el experimento estaba pausado, se podrá cargar más adelante para seguir con él.

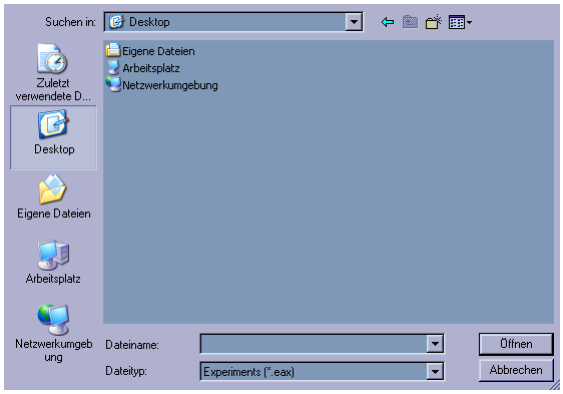

Figura A.2: Carga de un experimento

Para introducir los parámetros de un experimento nuevo, se utiliza la parte derecha de la ventana (figura A.4). Para los problemas de optimización numéricas hay que especificar un rango para todas las variables de objeto o un rango para cada una de ellas. Un ejemplo para un rango que sirve para todas las variables de objeto es "[-10.0;7.0)". Como se ve hay que indicar con los paréntesis correspondientes la inclusión o exclusión de los márgenes. Si se quiere especificar un rango para cada variable, simplemente se introducen secuencialmente. Para obtener información sobre algún parámetro determinado, se puede mover el ratón por encima y leer el *tooltip* correspondiente.

A.2 Línea de comandos

Mediante el comando *algevo -h* se recibe una explicación de cómo especificar un experimento mediante la línea de comandos. Opcionalmente se puede especificar y ejecutar un experimento indicando por la línea de comandos el fichero que contiene su especificación

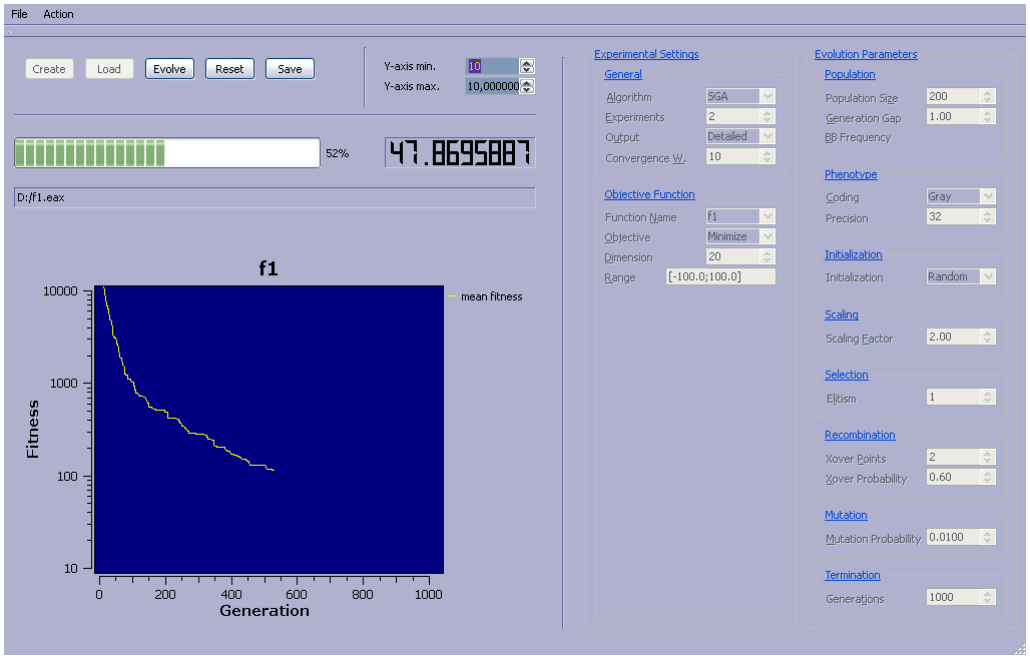

Figura A.3: Experimento en progreso

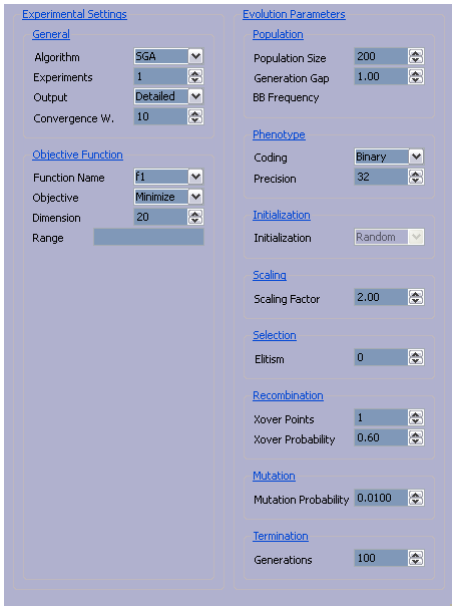

Figura A.4: Introducción de parámetros

como parámetro (único).

La sintaxis para especificar un experimento con un fichero se presenta mediante un ejemplo. El símbolo '#' trata la línea como comentario:

```
Experiment
{
  # numero de experimentos
  Experiments 30

  # especificacion de la funcion de evaluacion
  Evaluation f1
  {
    Dimension 30

    # se puede indicar un rango para todas la variables
    # o uno por cada variable (con varios Range)
    Range [-100.0;100.0]
  }

  # Output 1 guarda toda la informacion del transcurso del experimento
  # Output 0 solo guarda la mejor solucion encontrada
  Output 1

  # especificacion del algoritmo evolutivo
  # puede ser SGA, GACI, GASA, GABBTO o GABBRA
  EA SGA
  {
    # el objetivo puede ser Minimization o Maximization
    Objective Minimization
    # para GABBTO y GABBRA hay que especificar aqui la
    # frecuencia de deteccion, p.ej.: Frequency 200
    Generations 10000
    PopulationSize 200
    KPoints 2

    # para los numeros reales tambien se puede usar esta sintaxis
    MutationProbability 1.0E-002
    CrossoverProbability 6.0E-001
    ConvergenceWindow 10
    ScaleFactor 2.0000000000000000E+000
```

```
    Elitism 1
    GenerationGap 1.0000000000000000E+000
    Phenotype
    {
      # Coding puede ser Gray o Binary
      Coding Gray
      Precision 16
    }
  }
}
```

No todos los parámetros tienen sentido en todos los algoritmos. Solamente se permiten los que tienen sentido, es decir, p. ej. no se permite especificar el tamaño poblacional en el algoritmo GACI. A continuación se muestra un ejemplo para especificar el problema de optimización JHRR (solo se muestra la parte de *Evaluation*):

```
Evaluation JHRR
{
  K 4
  Block 8
  Gap 7
  M* 4
  V 2.0E-002
  U* 1.0E+000
  U 3.0E-001
}
```

Cuando se haya realizado el experimento, el fichero contiene la información siguiente en sus columnas (en este orden):

- Número de experimento
- Número de generación
- Número de evaluaciones realizadas
- Mejor valor de evaluación jamás encontrado
- Actual mejor valor de evaluación
- *Offline performance*
- *Online performance*

- Desviación estándar del valor de evaluación
- Valor de convergencia de la evaluación (véase 4.2.1)

Además al final del fichero se guarda la especificación del experimento.

Mediante los siguientes comandos se puede convertir el fichero obtenido a otro con información estadística más compacta:

Comando para convertir el fichero a datos estadísticos por generación:

```
algevo -genconv <ficheroexp> <ficherostat> <cuantil>
```

donde *ficheroexp* es el fichero obtenido por el experimento, *ficherostat* es el fichero de salida y *cuantil* define los límites de los cuantiles, es decir, para cuartiles suele ser 0.25.

El fichero de salida contiene la información siguiente en sus columnas (en este orden):

- Número de generación
- Mejor valor de evaluación de todos los experimentos
- Media de los mejores valores de todos los experimentos
- Media de los mejores valores actuales de todos los experimentos
- Media de los *Offline performance*
- Media de los *Online performance*
- Media de desviación estándar del valor de evaluación
- Media de convergencia de la evaluación
- Mediana de los mejores valores de todos los experimentos
- Cuantil Q1
- Cuantil Q3

Comando para convertir el fichero a datos estadísticos por evaluación:

```
algevo -evlconv <ficheroexp> <ficherostat> <frecuencia> <cuantil>
```

donde *frecuencia* es el número de evaluaciones cada cual aparecen los datos estadísticos.

El fichero de salida solamente se distingue en la primera columna, donde - en vez de la generación - se indica el número de evaluaciones realizadas.

B Análisis económico global y temporal del proyecto

Cuadro B.1: Tiempo dedicado para cada tarea (en horas)

Tarea	**Duración prevista**	**Duración real**
Análisis	35	60
Diseño	50	200
Implementación	480	700
Experimentación	240	300
Documentación	150	200
Total	955	1460

Cuadro B.2: Costes para software y hardware

Producto	**Coste (en euros)**
Visual C++ .Net	105.00
MS Visio	543.00
Minitab	850.00
PC	600.00
Total	2098.00

Cuadro B.3: Recursos humanos

Perfil	**Horas**	**Precio/hora (en euros)**	**Total Total**
Director	25	60.00	1500.00
Analista	260	45.00	11700.00
Programador	700	20.00	14000.00
Experimentador	300	13.00	3900.00
Total			31100.00

Cuadro B.4: Costes total (en euros)

Tipo	**Coste**
Software y hardware	2098.00
Recursos humanos	31100.00
Total	33198.00

Bibliografía

[Bäc96] T. Bäck. *Evolutionary Algorithms in Theory and Practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford University Press US, 1996.

[Bal94] Shumeet Baluja. Population-based incremental learning: A method for integrating genetic search based function optimization and competitive learning,. Technical Report CMU-CS-94-163, Carnegie Mellon University, Pittsburgh, PA, 1994.

[BB06] Thomas Bartz-Beielstein. *Experimental Research in Evolutionary Computation*. Natural Computing Series, Springer-Verlag, 2006.

[BJR99] Grady Booch, Ivar Jacobson, and James Rumbaugh. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[FOW66] L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence through Simulated Evolution*. John Wiley & Sons, New York, 1966.

[GHJV00] Gamma, Helm, Johnson, and Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 2000.

[Gol89] D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1989.

[Hol75] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[Kan99] Gopal K. Kanji. *100 Statistical Tests*. SAGE Publications, 1999.

[Kea95] A. J. Keane. A brief comparison of some evolutionary optimization methods. In *Proceedings of the Conference on Applied Decision Technologies (ADT '95). Volume 2: Modern Heuristic Search Methods*, pages 125–138, Uxbridge, UK, April 1995. Unicom Seminars.

[Koz92] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[Lar98] Craig Larman. *Applying UML and Patterns, An Introduction to Object-Oriented Analysis and Design.* Prentice-Hall, 1998.

[Rec73] Ingo Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution.* Frommann-Holzbog, Stuttgart, 1973. German.

[RJB98] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language – Reference Manual.* Addison-Wesley, 1998.

[Sch77] H. P. Schwefel. Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie. *Interdisciplinary Systems research*, 26, 1977. Birkhauser,Basel.

[SM87] J. David Schaffer and Amy Morishima. An adaptive crossover distribution mechanism for genetic algorithms. In *Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, pages 36–40. Lawrence Erlbaum, 1987.

[Tai93] E. Taillard. Benchmarks for basic scheduling problems. *European Journal of Operations Research*, 64:278–285, 1993.

[WM95] David H. Wolpert and William G. Macready. No free lunch theorems for search. Technical Report SFI-TR-95-02-010, Santa Fe Institute, 1995.

[YL97] Xin Yao and Yong Liu. Fast evolution strategies. In Peter J. Angeline, Robert G. Reynolds, John R. McDonnell, and Russ Eberhart, editors, *Evolutionary Programming VI*, pages 151–161, Berlin, 1997. Springer. Lecture Notes in Computer Science 1213.