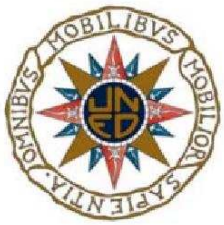


Universidad Nacional de Educación a Distancia

Escuela Técnica Superior de Ingeniería Informática

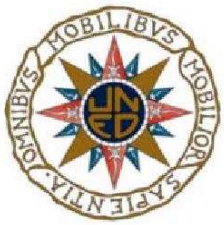
Proyecto de Fin de Carrera de Ingeniero Informático

Diseño, Aplicación y Evaluación de un Algoritmo Evolutivo

Manuel Ruiz de Quintanilla Foncubierta

Director: Severino Fernández Galán

Curso 2008/2009. Convocatoria de Diciembre/Enero.



Diseño, Aplicación y Evaluación de un Algoritmo Evolutivo

Proyecto de Fin de Carrera de modalidad Oferta General.

Realizado por: Manuel Ruiz de Quintanilla Foncubierta

Dirigido por: Severino Fernández Galán

Tribunal calificador:

Presidente: D/D^a. .....

Secretario: D/D^a. .....

Vocal: D/D^a. .....

Fecha de lectura y defensa: .....

Calificación: .....

Resumen

La *computación evolutiva* es una rama de las *ciencias de la computación* que ha sido ampliamente utilizada en campos como la *inteligencia artificial*. Los *algoritmos evolutivos* usan métodos de búsqueda estocástica basados en la evolución natural de cara a resolver problemas de adaptación en dominios como: optimización, diseño, aprendizaje o asignación de horarios. La computación evolutiva utiliza procesos iterativos como el crecimiento o desarrollo de una población. Dicha población es posteriormente seleccionada mediante un procedimiento aleatorio guiado que lo condiciona para converger a la solución buscada. Estos procedimientos vienen inspirados por los mecanismos biológicos de la evolución. Existen distintas clases de algoritmos evolutivos: algoritmos genéticos, estrategias evolutivas, programación evolutiva y programación genética.

El *problema de la mochila* plantea un escenario donde disponemos de un contenedor con una capacidad máxima y una colección de objetos con dos características fundamentales: *valor* y *volumen*. El problema consiste en rellenar el contenedor con los objetos, de forma que la suma del valor total sea máxima sin sobrepasar la capacidad de la mochila.

En el presente proyecto de fin de carrera planteamos la resolución del *problema de la mochila* por medio de un *algoritmo genético*. Para ello debemos definir tanto la representación de los individuos como el conjunto de operadores necesarios para su ejecución. En el caso de la función de evaluación, pieza fundamental de nuestro algoritmo, hemos desarrollado distintas alternativas con el fin de realizar posteriormente una evaluación de las mismas.

Se ha hecho especial énfasis en la elaboración de una aplicación a través de la cual resolver el problema de la mochila. Como característica fundamental de la aplicación debemos destacar el esfuerzo dedicado a crear una interfaz intuitiva con la que, a través de gráficos y animaciones, se pueda simular y visualizar el proceso de cálculo del algoritmo genético para un amplia variedad de casos.

Para terminar, ha sido realizada una extensa evaluación empírica del comportamiento de las tres funciones de evaluación por medio de problemas generados aleatoriamente. De todas las simulaciones realizadas, mostramos determinados casos

representativos que describen el comportamiento de nuestro algoritmo bajo distintas condiciones de ejecución.

Palabras clave

computación evolutiva, algoritmos genéticos, problema de la mochila, función reparadora, función decodificadora, función penalizadora, evaluación empírica

Summary

Evolutionary computing is a branch of *computer science* that has been widely used in *artificial intelligence*. *Evolutionary algorithms* use stochastic searching methods based in natural evolution. These methods are useful in adaptative problems such as optimization, design, learning and scheduling. *Evolutionary computing* uses an iterative process, such as the growth or the development of a population. This population is later selected by a guided random search to achieve the desired end. This process is often inspired by biological mechanisms of evolution. There are many evolutionary computing approaches: genetic algorithms, evolutionary strategies, evolutionary programming and genetic programming.

As far as the *knapsack problem* is concerned, there is a container with a maximun capacity and a collection of different objects which can be used to fill the container. Each object has two attributes: value and weight. The main task consists of filling the container with the objects so that the sum of the total value would be the maximum achievable without exceeding the container's limit.

The knapsack problem has been solved in the present project by means of a *genetic algorithm*. In order to achieve this, we have to define the representation of the individuals as well as the operators needed for its implementation. In the case of the *evaluation function*, which is an important element of the algorithm, a variety of alternatives have been developed so as to carry out an evaluation of them.

Special emphasis has been given to the creation of an application in order to solve the knapsack problem. The main characteristic of this application is the ability to create an intuitive interface that is able to simulate and visualize the calculation process of genetic algorithms for a variety of cases by using graphics and animation.

To sum up, an empirical wide assessment of the three evaluation functions has been executed by means of radomly generated problems. Of the many simulations which have been carried out, only the most representative ones that describe the behaviour of algorithms under different execution conditions, have been shown.

Keywords

evolutionary computation, genetic algorithms, knapsack problem, repair function, decoder function, penalty function, empirical evaluation

"Owing to this struggle for life, variations, however slight and from whatever cause proceeding, if they be in any degree profitable to the individuals of a species, in their infinitely complex relations to other organic beings and to their physical conditions of life, will tend to the preservation of such individuals, and will generally be inherited by the offspring. The offspring, also, will thus have a better chance of surviving, for, of the many individuals of any species which are periodically born, but a small number can survive. I have called this principle, by which each slight variation, if useful, is preserved, by the term Natural Selection."

Charles Darwin, The Origin of Species, 1859.

Índice general

1. Introducción **21**

1.1. Objetivos 21

1.2. Metodología 22

1.3. Organización de la Memoria 23

2. Computación Evolutiva **25**

2.1. Algoritmos Evolutivos 25

2.2. Desarrollo Histórico de la Computación Evolutiva 26

2.3. Estructura Genérica de un Algoritmo Evolutivo 28

2.3.1. Inicialización de la Población 28

2.3.2. Evaluación de la Población 29

2.3.3. Selección de Padres 30

2.3.4. Recombinación 31

2.3.5. Mutación 31

2.3.6. Evaluación de los Descendientes 32

2.3.7. Selección de Supervivientes 32

2.3.8. Condición de Terminación 33

2.4. Ramas de la Computación Evolutiva 33

2.4.1. Algoritmos Genéticos 33

2.4.2. Estrategias Evolutivas 34
2.4.3. Programación Evolutiva 34
2.4.4. Programación Genética 35

3. Aplicación al Problema de la Mochila **37**

3.1. Justificación Matemática 38
3.1.1. Programación Lineal 38
3.1.2. Programación Lineal Entera y Programación Lineal Entera Binaria o 0-1 39
3.1.3. Definición del Problema de la Mochila 39
3.2. Métodos de Resolución del Problema de la Mochila 40
3.2.1. Búsqueda Exhaustiva 41
3.2.2. Programación Dinámica 41
3.2.3. Algoritmos Voraces 41
3.2.4. Algoritmos Genéticos 43
3.3. Resolución Mediante un Algoritmo Genético 44
3.3.1. Representación de Individuos 44
3.3.2. Inicialización de la Población 46
3.3.3. Función de Evaluación 46
3.3.3.1. Función Decodificadora 47
3.3.3.2. Función Reparadora 50
3.3.3.3. Función Penalizadora 51
3.3.4. Selección de Padres 53
3.3.5. Operador de Cruce 54
3.3.6. Operadores de Mutación 55
3.3.7. Selección de Supervivientes 55

4. Entorno de Aplicación Diseñado **57**

4.1. Jerarquía de Clases 57

4.1.1. Paquete mochila 58

4.1.2. Paquete mochila.interfaz 58

4.2. Interfaz Gráfico 59

4.2.1. Arranque de la Aplicación 59

4.2.2. Pantalla Principal de la Aplicación 62

4.2.3. Barra de Menú 64

4.2.4. Botones e Información Básica 64

4.2.5. Configuración de Objetos 66

4.2.6. Configuración de Preferencias 66

4.2.7. Gráfico de Evaluación 68

4.2.8. Gráfico de Distribución de Objetos 68

4.2.9. Gráfico de Distribución de Objetos en la Mochila 69

4.2.10. Formatos de Visualización de Objetos y Mochila 70

5. Evaluación Empírica **73**

5.1. Metodología Para las Simulaciones 73

5.2. Experimentos Realizados 74

5.2.1. Comparativa Entre las Tres Funciones. 74

5.2.2. Diferencias Entre la Función Reparadora y la Decodificadora . 78

5.2.3. Análisis Para Tamaños Grandes 83

5.3. Resultados 88

5.3.1. Desempeño de la Función Penalizadora 89

5.3.2. Funciones Reparadora y Decodificadora 90

6. Conclusiones **91**

6.1. Interfaz Gráfico 91
6.2. Evaluación 92
6.3. Posibles Mejoras 92

Bibliografía **95**

Índice de figuras

4.1. Pantalla principal de la aplicación 63
4.2. Botones e información básica de la aplicación 65
4.3. Configuración de objetos 66
4.4. Parámetros de la aplicación 67
4.5. Gráfico de evaluación de generaciones 68
4.6. Gráfico de distribución de objetos 69
4.7. Gráfico de distribución de objetos en la mochila 69
4.8. Gráfico de distribución de objetos en la mochila superando la capacidad 69
4.9. Distribución asimétrica del ratio valor/volumen 70
4.10. Formatos de visualización de los objetos y la mochila 70
4.11. Distribución normalizada para el ratio valor/volumen 71

5.1. Capacidad: 500, Objetos: 100, Generaciones: 100. Escala Lineal . . . 75
5.2. Capacidad: 500, Objetos: 100, Generaciones: 100. 76
5.3. Capacidad: 1000, Objetos: 100, Generaciones: 100. 76
5.4. Capacidad: 2000, Objetos: 100, Generaciones: 100. 77
5.5. Capacidad: 3000, Objetos: 100, Generaciones: 100. 77
5.6. Capacidad: 4000, Objetos: 100, Generaciones: 100. 77
5.7. Capacidad: 8000, Objetos: 100, Generaciones: 100. 78
5.8. Capacidad: 500, Objetos: 100, Generaciones: 100. 79

5.9. Capacidad: 4000, Objetos: 100, Generaciones: 100. 79

5.10. Capacidad: 8000, Objetos: 100, Generaciones: 100. 80

5.11. Capacidad: 500, Objetos: 100, Generaciones: 1000. 80

5.12. Capacidad: 4000, Objetos: 100, Generaciones: 1000. 80

5.13. Capacidad: 8000, Objetos: 100, Generaciones: 1000. 81

5.14. Comparativa en torno a una capacidad de 500 unidades. Capacidad 300 y 400. 81

5.15. Comparativa en torno a una capacidad de 500 unidades. Capacidad 600 y 700. 82

5.16. Comparativa para capacidad de 500 unidades con 1000 objetos 82

5.17. Comparativa para capacidad de 500 unidades con 10000 objetos . . . 83

5.18. Comparativa para 10000 objetos. Capacidad entre 10000 y 10000000 unidades 84

5.19. Comparativa para 10000 objetos. Capacidad 200000 y 300000 84

5.20. Capacidad: 250000, Objetos: 10000, Generaciones: 100. 85

5.21. Capacidad: 100000, Objetos: 10000, Generaciones: 10000. 86

5.22. Capacidad: 200000, Objetos: 10000, Generaciones: 10000. 86

5.23. Capacidad: 300000, Objetos: 10000, Generaciones: 10000. 87

5.24. Capacidad: 10000 y 50000, Objetos: 10000, Generaciones: 100. 87

5.25. Capacidad: 100000 y 500000, Objetos: 10000, Generaciones: 100. . . . 88

5.26. Capacidad: 1000000 y 10000000, Objetos: 10000, Generaciones: 100. . 88

Lista de algoritmos

1. Estructura general de un Algoritmo Evolutivo 29
2. Implementación voraz del problema de la mochila 42

Índice de tablas

3.1. Ejemplo de lista de objetos 45

3.2. Tamaño del espacio de soluciones en función del número de objetos para el problema de la mochila. 46

3.3. Colección de objetos con ratio valor/volumen calculado 48

3.4. Ejemplo de ejecución de función de evaluación decodificadora 49

4.1. Formato de fichero de parámetros 60

4.2. Formato de fichero de objetos 62

Capítulo 1

Introducción

El presente Proyecto de Fin de Carrera culmina y completa la formación que el alumno ha estado recibiendo durante estos últimos años en la Escuela Técnica Superior de Ingeniería Informática de la U.N.E.D. como futuro Ingeniero Superior en Informática.

Con la realización del presente Proyecto de Fin de Carrera, el autor pretende poner en práctica los conocimientos y técnicas adquiridos durante estos años de estudio, aplicados a un problema concreto. Para ello será decisiva la capacidad de análisis y de síntesis, así como el ser capaz de resolver problemas prácticos de una forma rápida, eficaz y profesional. Cabe destacar también la figura del Tutor de Proyecto, que, además de ejercer una labor de ayuda indispensable en la elaboración del mismo, posibilita el desarrollo de un aspecto tan destacable (y valorado profesionalmente) como la capacidad del trabajo en equipo. Por último, y no menos importante, resaltar la labor de documentación e investigación para profundizar en determinados aspectos clave para el desarrollo eficaz del Proyecto de Fin de Carrera.

1.1. Objetivos

Mediante el presente Proyecto de Fin de Carrera se pretende estudiar la aplicación de técnicas de computación evolutiva en la resolución de problemas de optimización.

Cuando la solución clásica se encuentra con un espacio de soluciones excesivamente extenso, cobra sentido realizar una exploración dirigida sobre dicho espacio para no sufrir los efectos de la explosión combinatoria. En determinados problemas complejos, resulta interesante obtener una buena solución en un espacio de tiempo acotado. Es preferible obtener dicha solución, a explorar todo el espacio de soluciones para obtener la mejor, con el coste prohibitivo que esto supone. La computación evolutiva constituye una técnica que aglutina estas ventajas.

Como ejemplo práctico de resolución de problemas de optimización mediante algoritmos evolutivos [ES07], se ha elegido el denominado «problema de la mochila». El mismo constituye uno de los problemas clásicos en computación y tiene un gran número de aplicaciones prácticas como: contratación de personal [BIKK07a], planificación [KS07], reparto de inversiones [BIKK07b], diseño de circuitos [Ist02], control de capacidad industrial, cuidado de la salud, redes de computadoras [BS02], entre otras muchas.

La implementación del algoritmo genético se realizará por medio de un entorno que permita simular diferentes problemas y modificar las características fundamentales del algoritmo que los resuelve. Este entorno estará claramente enfocado al usuario, de forma que mostrará la ejecución del algoritmo de forma clara y concisa por medio de gráficos explicativos y animaciones.

Tras el diseño e implementación del algoritmo evolutivo que resuelva el problema de la mochila, se evaluará empíricamente el efecto que la modificación de determinadas características fundamentales del mismo tiene sobre su comportamiento.

1.2. Metodología

Basándonos en la descripción general de un *algoritmo evolutivo*, se determinó que la variante idónea para nuestro implementación sería la de un *algoritmo genético*. Se determinaron qué características de los algoritmos genéticos podían influir en mayor medida en su comportamiento al ser aplicados al problema de la mochila. Como característica fundamental y determinante se estableció la *función de evaluación* de los individuos de la población. Se han implementado tres aproximaciones diferentes para la evaluación de individuos en el problema de la mochila: una función decodi-

ficadora, una función reparadora y una función penalizadora. Uno de los objetivos principales de este proyecto es comparar el desempeño de estos tres tipos de función de evaluación.

Para realizar la implementación del algoritmo, se planteó realizar una aplicación capaz de proveer al usuario de un entorno en el que pudiera comprobar la evolución del algoritmo para una colección diversa de problemas y casos. Cabe destacar el especial esfuerzo que se ha hecho en la representación gráfica de la ejecución del algoritmo, de forma que se pueda observar fácilmente cómo van generándose nuevas soluciones e interpretar los resultados mostrados. También se proporciona al usuario de un entorno de experimentación en el que puede comprobar como afecta a la ejecución del algoritmo la modificación de diversos parámetros del mismo.

Tras la implementación de la solución propuesta se ha realizado una exhaustiva evaluación empírica a partir de una amplio abanico de problemas de la mochila generados aleatoriamente. El análisis de dichos resultados pretende establecer cuál de las tres funciones de evaluación consideradas es la más adecuada para resolver el problema de la mochila, así como determinar qué configuración de los parámetros propios del algoritmo genético favorece en mayor medida la eficiencia y eficacia del mismo

1.3. Organización de la Memoria

La presente memoria está estructurada en cuatro partes fundamentales:

Introducción a la computación evolutiva. Analizamos las características fundamentales de un *algoritmo evolutivo*, su utilidad, campos de aplicación, resultados esperados, así como una reseña sobre la evolución histórica que ha llevado al desarrollo de este tipo de algoritmos.

Aplicación al problema de la mochila. Nos centramos en el problema de la mochila, su enunciado, su características fundamentales, su representación, sus puntos críticos y sus aplicaciones. Enlazamos la descripción genérica del problema con la implementación concreta que se ha realizado mediante algoritmos genéticos, deteniéndonos en cada una de las partes fundamentales del algorit-

mo.

Entorno de aplicación diseñado. Seguiremos con la descripción de la aplicación desarrollada para poder comprobar el funcionamiento y comportamiento de nuestro algoritmo genético, su representación gráfica, sus características y su forma de uso.

Evaluación. Mostramos en este apartado los resultados obtenidos en la evaluación empírica de nuestro algoritmo. Comparamos su comportamiento en problemas de distinta complejidad. Además, para un mismo problema, comprobamos cómo afecta a nuestro algoritmo la modificación de sus características fundamentales (función de evaluación, tamaño de la población, etc)

Conclusiones. Recopilación de resultados obtenidos, discusión sobre dichos resultados y elaboración de conclusiones justificadas sobre el comportamiento del algoritmo.

Capítulo 2

Computación Evolutiva

La *computación evolutiva* es una rama de las ciencias de la computación que ha sido ampliamente utilizada en campos como la *inteligencia artificial*. Los algoritmos evolutivos usan métodos de búsqueda estocástica basados en la evolución natural de cara a resolver problemas de adaptación en dominios como: optimización, diseño, aprendizaje o asignación de horarios, entre otros muchos. Una completa revisión de las aplicaciones de interés de los algoritmos evolutivos se puede encontrar en [BIKK07a], [KS07], [BIKK07b], [Ist02], [WPRF98] y [BS02].

La computación evolutiva utiliza procesos iterativos como el crecimiento o desarrollo de una población. Esta población es posteriormente seleccionada mediante un procedimiento aleatorio guiado que lo condiciona para converger a la solución buscada. Estos procedimientos vienen inspirados por los mecanismos biológicos de la evolución [Jon06].

2.1. Algoritmos Evolutivos

Los algoritmos evolutivos incorporan técnicas inspiradas en la evolución biológica como la reproducción, mutación, recombinación, selección natural y supervivencia del más apto. Las soluciones candidatas del problema juegan el papel de individuos dentro de la población, y una función de evaluación es la encargada de determinar la aptitud de cada elemento y cómo evoluciona el sistema. La evolución de la población tiene lugar repitiendo la aplicación de los operadores anteriormente reseñados.

En este proceso coexisten dos operaciones fundamentales que conforman la estructura básica de un *sistema evolutivo*: *recombinación* y *mutación*. Éstas son las que posibilitan que haya *diversidad en la población*, mientras que la *selección* actúa como operación que incrementa la calidad de la población [TH93].

Muchos aspectos de estos procesos evolutivos son *estocásticos*. Las distintas piezas de información resultado de la recombinación y mutación son seleccionadas aleatoriamente. Por otra parte, los operadores de selección pueden ser bien deterministas o estocásticos. En el caso general, los miembros con evaluación más alta tienen más posibilidades de ser seleccionados que aquellos con una evaluación peor pero, aún así, incluso los elementos débiles tienen alguna posibilidad de convertirse en padre para la siguiente generación o sobrevivir en dicha generación.

2.2. Desarrollo Histórico de la Computación Evolutiva

Las primeros intentos de lo que hoy conocemos como *algoritmos genéticos* aparecieron a finales de los años cincuenta y principio de los sesenta en forma de programas realizados por biólogos que intentaban construir un modelo artificial de la evolución natural [Mar04]. En un principio no se estimó que este tipo de estrategias pudieran ser usadas de una forma más genérica, aunque no tardó en surgir la idea.

En 1965, Ingo Rechenberg introdujo una técnica que denominó *estrategia evolutiva*, aunque era más similar a un procedimiento de escalada que a un *algoritmo genético*. En esta técnica, no existe una población ni operación de cruce entre los elementos. Un padre es mutado para obtener una determinada descendencia y el mejor de ellos dos se conserva y convierte en el padre de la siguiente ronda de mutación [HH98]. Posteriores versiones introdujeron la idea de *población*. Las *estrategias evolutivas* son empleadas aún hoy en día por ingenieros y científicos, especialmente en Alemania.

El siguiente desarrollo de importancia en este campo hizo su aparición en 1966, cuando L. J. Fogel, A. J. Owens y M. J. Walsh introdujeron en Estados Unidos una técnica denominada *programación evolutiva*. En este método, las soluciones candidatas del problema eran representadas como simples *máquinas de estado finito*. Al

igual que en la estrategia evolutiva de Rechenberg, su algoritmo funcionaba mediante una mutación aleatoria de estas máquinas simuladas y mantenía la mejor de cada dos ([Mit96] y [Gol89]). Al igual que las *estrategias evolutivas*, una formulación más extensa de la *programación evolutiva* es objeto de estudio hoy en día. Aún así, se acusa en estas dos metodologías de la falta de reconocimiento de la importancia del *cruce de individuos*.

Durante 1962, John Hollan sentó las bases de *sistemas adaptativos* que constituyeron la base de futuros desarrollos. Aún más destacable, Holland fue el primero que propuso el cruce de individuos y otro tipo de operadores de recombinación. Sin embargo, el trabajo fundamental en el campo de algoritmos genéticos hizo su aparición en 1975, con la publicación del libro «Adaptation in Natural and Artificial Systems». Basándose en investigaciones y artículos tanto de Holland como de sus colegas de la Universidad de Michigan, este libro fue el primero que de manera sistemática y rigurosa presentó el concepto de sistemas adaptativos digitales usando mutación, selección y operación de cruce, simulando procesos de la evolución biológica y una estrategia dirigida a la resolución de problemas. Ese mismo año, la tesis de Kenneth De Jong's estableció el potencial de los *algoritmos genéticos* mostrando que podían tener un buen desempeño en una amplia variedad de funciones, incluyendo espacios de búsqueda con ruido, discontinuos y multimodales [Gol89].

Estos trabajos propiciaron la extensión del interés en la *computación evolutiva*. A principios de la década de 1980, los algoritmos genéticos fueron usados en un amplio espectro de aplicaciones, desde modelos matemáticos abstractos como el problema de la mochila o el coloreado de grafos a problemas más tangibles de la rama de ingeniería tales como control de flujo en tuberías, reconocimiento de patrones y clasificación y optimización de estructuras. Ejemplos de estas aplicaciones pueden consultarse en [Gol89].

Al principio, estas aplicaciones fueron principalmente teóricas. Sin embargo, a medida que proliferaron las investigaciones, los algoritmos genéticos migraron al sector comercial. Hoy en día, la *computación evolutiva* es un campo muy próspero, y los algoritmos genéticos se encargan de resolver problemas cotidianos en áreas de estudio tan diversas como la predicción de mercados de valores y gestión de carteras, ingeniería aeroespacial, diseño de microchips, bioquímica y biología molecular, planificación, gestión de líneas de ensamblaje, etc.

2.3. Estructura Genérica de un Algoritmo Evolutivo

Los algoritmos evolutivos son algoritmos genéricos, basados en poblaciones y optimizaciones heurísticas que usan mecanismos basados en la biología tales como mutación, recombinación, selección natural y supervivencia del más apto de cara a refinar un conjunto de soluciones candidatas iterativamente.

En la naturaleza continuamente se produce una competición por acaparar los recursos disponibles. Sólo los individuos más aptos sobreviven. La capacidad para adaptarse a un entorno cambiante es uno de los factores determinantes de la aptitud. Las características de los individuos que ayudan a su supervivencia están determinadas por el contenido de sus genes. El conjunto de genes que controla las características de los individuos se denomina cromosoma.

La evolución está dirigida por la acción conjunta de la selección natural y la recombinación de material genético. Debido a la selección natural, los miembros más aptos dominan el acceso a los recursos disponibles y tienen más posibilidades de procrear, contribuyendo a la supervivencia de su genoma. Esto conlleva que el material genético más apto es el que prevalece sobre el resto.

Cualquiera de la gran cantidad de algoritmos evolutivos propuestos hasta hoy en día se basa en el proceso de selección natural. Todos ellos tienen en común el concepto de simular la evolución de estructuras individuales usando operadores genéticos como la selección, mutación y recombinación (reproducción). Los operadores usados en la computación evolutiva son sólo una simplificación del conjunto de procesos biológicos. Aun así, son suficientemente complejos para proporcionar una mecanismo de búsqueda adaptativa robusto.

La estructura genérica de un algoritmo evolutivo podría plantearse en forma de pseudocódigo tal y como se puede ver en el Algoritmo 1.

2.3.1. Inicialización de la Población

La inicialización de la población proporciona un primer conjunto de soluciones. Generalmente se realiza de manera aleatoria, dependiendo estrechamente de la repre-

Algoritmo 1 Estructura general de un Algoritmo Evolutivo

```
Inicialización de la población;
Evaluación de la población;
Mientras no se llegue a la condición de fin {
   Selección de Padres;
   Recombinación;
   Mutación;
   Evaluación;
   Selección de Supervivientes;
}
```

sentación elegida para nuestro algoritmo evolutivo. El conjunto de estas soluciones iniciales conformarán nuestra *población* y a cada uno de los elementos que componen los denominaremos *individuos*.

Al ser la generación de soluciones iniciales un procedimiento aleatorio, podemos obtener distintas soluciones que no tengan sentido en nuestro problema. Cuando definimos el problema y su representación, definimos también una serie de restricciones que nos permitan establecer si una solución es válida en el espacio de soluciones del problema o no lo es. Si establecemos que una de nuestras variables debe ser siempre mayor que otra y generamos una solución donde esto no se cumpla, esta solución no será válida. Las soluciones que consideramos válidas para nuestro problema las denominaremos *soluciones factibles* mientras que al resto las denominaremos *soluciones no factibles*.

En la generación inicial de soluciones es lógico que obtengamos varias soluciones no factibles. Cabrá la posibilidad de modificar estas soluciones no factibles para convertirlas en factibles, proyectarlas al espacio de soluciones factibles realizando una traducción, tratarlas convenientemente en nuestro algoritmo o, directamente, no permitir soluciones no factibles y seguir generando soluciones hasta que demos con el número deseado de soluciones factibles.

2.3.2. Evaluación de la Población

Necesitamos de una medida para cuantificar la bondad de cada una de nuestras soluciones. La función de evaluación representa los requisitos a los que la población debe adaptarse. Desde el punto de vista de un problema de optimización, se trata

de la tarea a ser resuelta. Una función de evaluación asocia generalmente un valor numérico a cada individuo, con el fin determinar la calidad de la solución que dicho individuo aporta al problema planteado.

La evaluación de los elementos de la población permite realizar comparaciones entre estos para saber qué elemento es más apto y, por tanto, deberá tener más opciones de transmitir su información genética a las siguientes generaciones, más probabilidades de sobrevivir o ambas.

2.3.3. Selección de Padres

La selección es uno de los operadores más importantes para los algoritmos evolutivos [BBM00]. El objetivo principal del operador de selección es enfatizar las mejores soluciones de una población. Este operador no genera ninguna nueva solución, sino que selecciona repetidamente buenas soluciones de una población y descarta las demás.

El proceso para identificar buenas o malas soluciones normalmente se realiza haciendo uso de la función de evaluación. La idea principal es que una solución con mejor función de evaluación debe tener una mayor probabilidad de ser seleccionada. Aún así, los distintos operadores de selección de padres se diferencian en la forma en que eligen los miembros de la población que serán elegidos para la reproducción.

Algunos operadores ordenan la población por el valor obtenido al aplicar la función de evaluación y escoge las mejores soluciones [Deb00]. Otros, sin embargo, asignan una probabilidad de selección proporcional a su evaluación a cada miembro de la población y realizan la selección usando esa función de probabilidad. En esta función probabilística, existe la posibilidad, aunque pequeña, de descartar una buena solución y escoger una peor en su lugar. Sin embargo, el operador se diseña de forma que la probabilidad de que esto suceda sea muy pequeña.

Existe también una ventaja a la hora de utilizar el operador probabilístico. Puede darse el caso, debido a una población inicial muy pequeña o una selección incorrecta de los parámetros iniciales, que en un problema de optimización no lineal, los mejores individuos pertenezcan todos a una misma región donde exista un óptimo local. Si en esta situación hacemos uso del operador determinista, la solución tenderá a converger

al máximo local. Sin embargo, en la versión estocástica del operador de selección, la diversidad de la población se mantiene eligiendo esporádicamente soluciones no tan buenas. Esta característica previene a los algoritmos genéticos de converger a máximos locales.

2.3.4. Recombinación

La *recombinación* permite obtener nuevos individuos a partir de los elementos de nuestra población [Esh00]. Los elementos originales serán los *padres*, mientras que los nuevos que generemos serán sus *descendientes*. Estos descendientes serán creados realizando una mezcla de las características de los padres .

En el caso ideal, un descendiente aglutinará las mejores características de sus padres, mejorando su evaluación con respecto a estos. Sin embargo, al no saber qué características proporcionan una mejor evaluación, no podemos determinar la forma óptima de aglutinar estas características. La mejor solución que podemos adoptar es mezclar las características de los padres aleatoriamente. Es a esta operación a la que denominamos recombinación [Spe00].

Al ser un proceso aleatorio, la recombinación genera tanto elementos que mejoran la evaluación como elementos que la empeoran. Esto no implica mayor problema ya que se confía en el operador de selección para que escoja de entre todos estos descendientes aquellos que presentan una mejor adaptación.

2.3.5. Mutación

Todos los algoritmos evolutivos funcionan a base de combinar el operador de selección con un mecanismo para producir variaciones. El mejor mecanismo para producir dichas variaciones es la *mutación* [Esh00].

El *operador de mutación* se basa en la modificación aleatoria del valor del alguno de los genes del individuo sobre el que lo aplicamos. Estos valores se denominan *alelos* y su representación puede variar en función del tipo de algoritmo evolutivo que estemos utilizando pudiendo ser valores lógicos, enteros, reales, etc.

La mutación genera nuevas soluciones haciendo pequeñas modificaciones en la

representación de los individuos. La tasa de dichas modificaciones es otra de las características fundamentales de un algoritmo evolutivo, siendo su valor más extendido el de un gen por cada individuo. Por ejemplo, si el individuo tiene un cromosoma (cadena de genes) de longitud cien, la probabilidad de que un gen concreto mute será de 0.01 (1 %) [Spe00].

2.3.6. Evaluación de los Descendientes

Una vez hemos generado los nuevos elementos de la población, debemos evaluarlos para saber el grado de aptitud de cada uno. Para ello utilizamos la *función de evaluación* que hemos definido, asignando el valor oportuno a cada uno de los nuevos miembros generados.

Es importante destacar que el reevaluar un mismo miembro no arrojará un resultado distinto del original. Solo tiene sentido evaluar elementos nuevos o que hayan mutado. El aplicar la función de evaluación es una operación costosa que se repite multitud de veces, por lo que conviene minimizar el número de estas ejecuciones.

2.3.7. Selección de Supervivientes

El fin primordial de la selección de supervivientes es diferenciar a los elementos de la población en función de su adecuación al medio. En cierta forma se parece a la selección de padres, pero es usada en otra etapa del ciclo evolutivo. El mecanismo de *selección de supervivientes* es usado tras la creación de una nueva generación. Como el tamaño de la población suele ser constante, nos encontramos en la diatriba de seleccionar qué elementos pasarán a la siguiente generación. Esta elección normalmente se hace basándonos en la evaluación de cada uno de los miembros, favoreciendo a aquellos que tienen una evaluación mejor. En algunas ocasiones también se suele utilizar el concepto de edad a la hora de escoger a los elementos a descartar [ES07].

El concepto de selección de supervivientes está muy relacionado con el solapamiento entre generaciones. En un modelo sin solapamiento, más conocido como generacional, los padres nunca compiten con su descendencia. Toda la población de padres es sustituida por sus descendientes. En un sistema con solapamiento, los

padres y su descendencia compiten por la supervivencia [Esh00].

A diferencia de la selección de padres, que suele ser un procedimiento estocástico, la selección de supervivientes tiende a ser completamente determinista.

2.3.8. Condición de Terminación

Un algoritmo genético se ejecuta a lo largo de un número de generaciones. Por tanto, se necesita una condición de terminación que explicite bajo qué circunstancias consideramos que la mejor solución encontrada hasta el momento es «suficientemente correcta».

Algunas de las condiciones de terminación más ampliamente utilizadas son [ES07]:

1. Número máximo de ciclos de CPU ocupados.
2. El número total de generaciones alcanza un límite prefijado.
3. La evaluación no mejora más allá de un determinado rango durante un periodo de tiempo prefijado.
4. La diversidad de la población cae por debajo de un determinado límite.

2.4. Ramas de la Computación Evolutiva

2.4.1. Algoritmos Genéticos

Los *algoritmos genéticos* son una subclase de los algoritmos evolutivos en el que los elementos del espacio de búsqueda son cadenas binarias, vectores o algún otro tipo elemental.

Entre las aplicaciones habituales de los algoritmos genéticos se encuentran: problemas de planificación, ingeniería química, medicina, data mining y análisis de datos, geometría y física, economía y finanzas, redes y comunicaciones, ingeniería eléctrica y diseño de circuitos, procesamiento de imágenes, optimización combinatoria, etc.

2.4.2. Estrategias Evolutivas

Las *estrategias evolutivas*, introducidas por Rechenberg son una técnica de optimización heurística basada en las ideas de adaptación y evolución, una forma especial de algoritmos evolutivos. Las estrategias evolutivas presentan las siguientes características:

- Normalmente usan vectores de números reales como soluciones candidatas.

- Los operadores de mutación y selección son los más utilizados y la recombinación es menos común.

- La mutación frecuentemente cambia los elementos del vector de la solución candidata por medio de una distribución normal.

- Los valores de los parámetros se van ajustando por medio de auto-adaptación al medio.

- En los demás aspectos, su comportamiento es semejante al del resto de algoritmos evolutivos.

Entre las aplicaciones de las estrategias evolutivas encontramos: minería y análisis de datos, problemas de planificación, química e ingeniería química, minimización de recursos y protección del medioambiente, optimización combinatoria, geometría y física, óptica y procesamiento de imágenes, etc.

2.4.3. Programación Evolutiva

Al contrario que el resto de algoritmos evolutivos analizados, no existe una especificación clara de la variedad denominada «*programación evolutiva*». Aún así, hay una característica diferenciadora clara: mientras que los individuos de una determinada especie son la metáfora biológica de las soluciones candidatas en otros algoritmos evolutivos, en programación evolutiva una solución candidata es tratada como si ella misma representara a una especie independiente. En este contexto, no tiene sentido hablar de recombinación entre individuos ya que no podemos recombinar elementos de distintas especies. De esta forma, la mutación y la selección son los únicos operadores usados en programación evolutiva y la recombinación casi nunca suele usarse.

Las aplicaciones de la programación evolutiva son diversas: aprendizaje automático, autómatas celulares y máquinas de estado finito, comportamientos evolutivos (agentes), química, ingeniería química y bioquímica, ingeniería eléctrica y diseño de circuitos, data mining y análisis de datos, robótica, etc.

2.4.4. Programación Genética

El término «*programación genética*» tiene dos posibles significados. Es usado habitualmente para contener a todos aquellos algoritmos evolutivos que utilizan árboles como estructuras de datos para los genotipos. Además, también puede ser definido como el conjunto de todos los algoritmos evolutivos que generan programas, algoritmos y construcciones similares.

Tradicionalmente, el concepto de procesamiento de información implica una instancia de un programa que recibe determinada información como entrada y devuelve el resultado de transformarla mediante una salida. En programación genética, normalmente determinadas entradas o situaciones se hacen corresponder con ejemplos de respuesta que son previamente conocidos o pueden ser producidos o simulados. El objetivo en este caso es encontrar un programa que sea capaz de conectar entrada y salida, o que muestre un determinado comportamiento ante ciertos estímulos o situaciones.

Entre las aplicaciones de la programación genética, podemos destacar: regresión simbólica y síntesis de funciones, inducción de gramáticas, data mining y análisis de datos, ingeniería eléctrica y diseño de circuitos, medicina, economía y finanzas, geometría y física, autómatas celulares y máquinas de estado finito, programación automatizada, robótica, redes y comunicaciones, comportamientos evolutivos (agentes), reconocimiento de patrones, bioquímica, aprendizaje automático, etc.

Capítulo 3

Aplicación al Problema de la Mochila

Como ejemplo práctico a resolver mediante un algoritmo evolutivo, hemos seleccionado el problema de la mochila. Dicho problema es uno de los ejemplos clásicos de optimización matemática.

Su enunciado plantea una colección de objetos definidos por dos características: *valor* y *volumen*. Disponemos asimismo de un contenedor de capacidad limitada, la mochila. La resolución consiste en rellenar la mochila con objetos de forma que su capacidad máxima no sea sobrepasada y el valor total del conjunto sea el máximo posible.

Este tipo de problemas, en el que debemos proporcionar una solución entera a un problema de reparto, aparece comúnmente en tareas de asignación de personal, planificación y logística, entre otras (véase [BIKK07a, KS07]).

A lo largo de las siguientes secciones vamos a ver las características principales de un problema de optimización. Analizaremos también distintas aproximaciones a la resolución del problema de la mochila como pueden ser la búsqueda exhaustiva, programación dinámica, algoritmos voraces y algoritmos genéticos. Posteriormente nos centraremos en el análisis pormenorizado de nuestro algoritmo genético y su aplicación al problema.

3.1. Justificación Matemática

Como indicamos anteriormente, el problema de la mochila es un problema de optimización matemática, más concretamente de *optimización combiatoria*. Este tipo de problemas trata de encontrar una solución x^* de un conjunto de posibles soluciones S. Esta solución debe representar el valor óptimo para una determinada función de evaluación f. Dicho de otra forma, para un problema de maximización tratamos de encontrar una x^* de forma que si $x \neq x^* \Rightarrow f(x^*) \geq f(x)$. Es habitual que nuestra solución deba cumplir también una serie de *restricciones* establecidas por el problema.

Dentro de la rama de optimización matemática encontramos distintos tipos y especializaciones. El problema de la mochila pertenece a la rama de programación lineal, más concretamente se trata de un problema de programación lineal entera binaria.

3.1.1. Programación Lineal

La *programación lineal* engloba todos aquellos modelos de optimización donde las funciones que lo componen, es decir, *función objetivo* y *restricciones*, son funciones lineales.

La forma general de la función objetivo en un problema de optimización lineal es:

$$max \sum_{i=1}^{n} c_i x_i$$

Donde x_i representa cada una de las variables de decisión y c_i es el coeficiente i-ésimo. Además de la función objetivo, existen una serie de restricciones que nuestra solución debe cumplir, pudiendo ser de alguna de las siguientes formas:

$$A_j \quad = \quad \sum_{i=1}^{n} a_{ij} x_i$$

$$B_j \leq \sum_{i=1}^{n} b_{ij} x_i$$

$$C_j \geq \sum_{i=1}^{n} c_{ij} x_i$$

Donde j representa el número de la restricción, A, B y C son valores conocidos, a, b y c son coeficientes conocidos y x_i es la variable de decisión [AL07, GGT04].

3.1.2. Programación Lineal Entera y Programación Lineal Entera Binaria o 0-1

Determinados problemas presentan una característica extra: que la solución esté compuesta por valores enteros para todas o algunas de las variables. El obtener una solución no entera no resulta conveniente porque es incongruente con la situación que planteada. Ejemplo de este tipo de problemas puede ser el reparto de una serie de bienes indivisibles.

Esta nueva característica se traduce en añadir la restricción $x_i \in \mathbb{Z}$ para forzar que la variable de decisión i-ésima sea entera.

Además de este tipo de problemas de programación lineal entera, tenemos una tercera especialización: los problemas de programación lineal entera binaria o 0-1. La principal característica de estos es que todas las variables de decisión x_i son obligatoriamente 0 o 1. Se usan en problemas de selección, donde las posibilidades se reducen a escoger o descartar.

3.1.3. Definición del Problema de la Mochila

El *problema de la mochila* es un problema de *programación lineal entera binaria*. Plantea la existencia de un contenedor de capacidad finita (la mochila) y una colección de objetos con dos características fundamentales: *valor* y *volumen*. Necesitamos seleccionar un conjunto de objetos tal que la suma de sus valores individuales sea la máxima posible y que, al mismo tiempo, el volumen total del conjunto no sobrepase la capacidad máxima de la mochila.

Los datos iniciales del problema serán:

- n: número de objetos entre los que se puede elegir.

- vo_i: volumen del objeto i-ésimo.

- va_i: valor asignado al objeto i-ésimo.

- C: capacidad de la mochila que determina el volumen máximo de objetos que puede contener.

La colección de objetos a introducir en la mochila serán las variables a tener en cuenta (x_i) y solo tendrán dos valores posibles: 1, en el caso en que el objeto vaya a ser introducido en la mochila o 0 en caso en que quede fuera de la selección final.

La restricción vendrá marcada por la capacidad máxima de la mochila, de tal forma que la suma de todos los objetos multiplicados por el espacio que ocupan en la mochila no podrá exceder dicha capacidad máxima.

La formulación matemática del problema queda:

$$max \sum_{i=1}^{N} va_i x_i$$

$$s.a.\, x_i \quad \in \quad \{0,1\}\, \forall i = 1...N \wedge \sum_{i=1}^{N} vo_i x_i \leq C$$

3.2. Métodos de Resolución del Problema de la Mochila

A continuación vamos a revisar alguno de los métodos más conocidos para la resolución del problema de la mochila. Nuestro objetivo es comparar sus virtudes y debilidades frente al método basado en computación evolutiva que desarrollaremos en el resto de secciones.

3.2.1. Búsqueda Exhaustiva

En una primera aproximación, la solución exacta del *problema de la mochila* pasaría por calcular todas las posibles configuraciones de objetos dentro de la mochila. Para cada una de estas configuraciones, podemos calcular el volumen total del conjunto y su valor. La solución óptima será aquella que tenga mayor valor, manteniendo un volumen menor o igual a la capacidad de la mochila.

En el caso de n objetos, habrá 2^n combinaciones posibles de objetos en la mochila. La complejidad de este método crece exponencialmente con el número de objetos inicial y solo será válido para instancias de pequeño tamaño del problema de la mochila.

3.2.2. Programación Dinámica

La *programación dinámica* es una técnica utilizada en la resolución de problemas cuyas soluciones satisfacen relaciones de recurrencia. Trata de resolver un determinado problema descomponiéndolo en partes más pequeñas que, una vez agrupadas, representan al problema original.

Como método de resolución utiliza una tabla interna donde va guardando las soluciones parciales a medida que son calculadas. De esta forma se elimina la posibilidad de volver a resolver un subproblema que ya ha sido resuelto anteriormente. Una vez completada la tabla con las soluciones de todos los subproblemas, se reconstruye a partir de ellas la solución al problema original.

La principal ventaja de la aproximación mediante programación dinámica es la sencillez de su implementación. Por contra, necesita construir en memoria la tabla que sirve como base a la resolución del problema. El tamaño de dicha tabla crece con el número total de objetos y la capacidad de la mochila, por lo que las necesidades de memoria para el problema crecen con estos.

3.2.3. Algoritmos Voraces

Los *algoritmos voraces* utilizan alguna heurística para generar una secuencia de soluciones subóptimas que, en el mejor de los casos, se espera que converjan a la

solución óptima.

El término voraz se deriva de la forma en que los datos de entrada se van tratando, realizando la elección de desechar o seleccionar un determinado elemento una sola vez. Para ello puede utilizar distintas estrategias dentro del contexto del problema de la mochila:

1. Escoger el elemento de mayor valor de los que quedan.
2. Escoger el elemento de menor volumen de los restantes.
3. Escoger el objeto que tiene mejor relación valor / volumen.

Los algoritmos voraces tienden a ser bastante eficientes y pueden implementarse de forma relativamente sencilla. Su eficiencia se deriva de la forma en que tratan los datos, llegando a alcanzar muchas veces una complejidad de orden lineal.

Vamos a ver un ejemplo de implementación de algoritmo voraz que resuelve el problema de la mochila.

Algoritmo 2 Implementación voraz del problema de la mochila

```
mochila_voraz (va[1...n], vo[1...n])
begin
for (i = 1;i <n; i++) {
   sol[i] = 0; }
valor_obten = 0;
j = 0;
volumen_ac = 0;
while (volumen_ac <C) {
   if (volumen[j] + volumen_ac <C) {
      valor_obten = valor_obten + valor[j];
      sol[j] = 1;
      volumen_ac = volumen_ac + volumen[j]
   }
   j = j+1;
}
benef = valor_obten;
```

Las variables y estructuras usadas son:

m capacidad de la mochila
va almacena el valor de cada objeto
vo almacena el volumen de cada objeto
sol devuelve la lista de objetos seleccionados
benef devuelve el beneficio total
valor_obten almacena el beneficio parcial
volumen_ac almacena el volumen parcial.

Podemos observar como la heurística seguida en esta implementación lo que hace es recorrer la lista de objetos, añadiendo aquellos que caben en la mochila mientras que descarta a los que hacen que sobrepase el volumen total admitido. En el caso en el que queramos mejorar la heurística para ir introduciendo los objetos de mayor valor, menor volumen o mejor ratio, tan solo habrá que realizar una ordenación previa de los vectores utilizados.

Los algoritmos voraces son rápidos y con pocas necesidades de memoria para guardar las estructuras de datos que necesitan. Como contrapartida, no podemos garantizar que se alcance el óptimo para cualquier problema.

3.2.4. Algoritmos Genéticos

Los *algoritmos genéticos* parten de una serie de soluciones iniciales generadas aleatoriamente a las que llamamos población. Cada una de estas soluciones puede ser cuantificada por medio de una función de evaluación que nos indicará su grado de aptitud. El algoritmo genético va generando nuevas poblaciones con la esperanza de que vaya mejorando su evaluación global y acercándose a la solución óptima.

Una nueva generación se crea a partir de la anterior eligiendo a los miembros con mejor evaluación, mezclando sus características y escogiendo a los más aptos. Aquellos elementos de la población con mejor evaluación tendrán más posibilidades de ser escogidos para formar parte de la nueva generación. Es importante hacer notar que debemos establecer un criterio de parada, ya que un algoritmo genético no tiene un punto de finalización definido tal y como los anteriores.

Con respecto a los anteriores métodos, los algoritmos genéticos tienen la característica de que sus requisitos temporales y espaciales no crecen con el aumento de

capacidad de la mochila. El espacio de memoria necesario para su implementación será independiente de la capacidad de la mochila. Por contra, su implementación es más compleja. Con respecto al método voraz, los algoritmos genéticos obtienen por regla general mejores soluciones que estos.

Cabe la posibilidad de aunar varios de estos métodos de resolución para resolver un problema. En nuestro caso se ha elegido la aproximación mediante algoritmos genéticos como herramienta para resolver el problema de la mochila, aunque se hará uso de una estrategia voraz en la implementación de la función de evaluación.

En la siguiente sección analizaremos cada uno de los aspectos fundamentales de un algoritmo genético y cuál ha sido la implementación concreta elegida en nuestro caso.

3.3. Resolución Mediante un Algoritmo Genético

Un algoritmo genético implementa un gran número de operaciones y procedimientos, basados en la representación concreta elegida para modelarlo. En cada uno de los apartados en los que se divide la implementación de un algoritmo de este tipo, disponemos de distintas posibilidades de cara a su diseño. Vamos a ir analizando y justificando cada una de las decisiones de diseño tomadas en la construcción de nuestra implementación.

3.3.1. Representación de Individuos

Un punto crítico para la resolución del problema de la mochila por medio de un algoritmo genético es la elección de la *representación de individuos*. En función de la representación escogida, variará completamente la implementación del algoritmo y de todos sus operadores.

Como ya vimos en los apartados anteriores, el problema de la mochila parte de una colección de objetos, definidos por su valor y su volumen, y la capacidad total que tendrá nuestra mochila. El resultado esperado será un subconjunto de esta colección de objetos, representando la configuración final de la mochila.

Resulta natural pensar en la representación de una solución cualquiera por medio

de una cadena de texto, de longitud igual al número total de objetos. Cada posición de esta cadena representará a uno de estos objetos que estarán almacenados en una lista ordenada para que sean fácilmente referenciables por su posición. En el caso en que en una posición dada haya un «1», implicará que el objeto ha sido seleccionado para pertenecer a la mochila en esta solución, mientras que un «0» significa que se ha desechado. A continuación podemos ver un ejemplo de problema de la mochila, con sus soluciones asociadas.

Posición	Objeto	Valor	Volumen
1	Objeto 1	6	3
2	Objeto 2	10	8
3	Objeto 3	3	5
4	Objeto 4	7	9
5	Objeto 5	9	4
6	Objeto 6	4	10
7	Objeto 7	6	1
8	Objeto 8	2	3
9	Objeto 9	8	4
10	Objeto 10	9	7

Tabla 3.1: Ejemplo de lista de objetos

Una solución cualquiera podría ser:

0	1	1	0	1	0	0	0	1	1

Que nos indicaría que dicha solución estima que los objetos de las posiciones 2, 3, 5, 9 y 10 deben pertenecer a la mochila, mientras que los objetos 1, 4, 6, 7 y 8 no pertenecerán a la misma.

La cantidad total de soluciones posibles será 2^n, para un número de objetos igual a n. Veamos como varía el tamaño del espacio de búsqueda en función del número de objetos considerado:

Objetos	Número de Soluciones
10	1024
100	$1,27 \cdot 10^{30}$
1000	$1,07 \cdot 10^{301}$
10000	$2 \cdot 10^{3010}$

Tabla 3.2: Tamaño del espacio de soluciones en función del número de objetos para el problema de la mochila.

3.3.2. Inicialización de la Población

Nuestro algoritmo genético necesita una población inicial. Esta población estará compuesta por una serie de soluciones para nuestro problema. Como vimos en el anterior apartado, cada una de las soluciones se traduce en una cadena de «0» y «1», determinando de esta forma la colección de objetos que forma parte de nuestra solución.

Inicialmente no tenemos ninguna guía para generar buenas soluciones, así que nos limitaremos a generar soluciones aleatorias. Para cada uno de los miembros de la población se asignarán distribuciones aleatorias de objetos esperando que, a medida que el algoritmo vaya calculando las siguientes generaciones, la evaluación global de las soluciones vaya mejorando.

3.3.3. Función de Evaluación

Un aspecto crucial de nuestro algoritmo genético es el procedimiento mediante el cual cuantificamos la bondad de cada uno de los elementos de la población. Esto se consigue por medio de la *función de evaluación*. La idea general es que cuanto mejor sea un elemento, mejor resultado debe obtener al aplicarle la función de evaluación. Una buena evaluación de los elementos de nuestra población permite compararlos con efectividad y escoger aquellos que mejor se adaptan al entorno.

Las posibilidades en cuanto a la elección de la función de evaluación son prácticamente ilimitadas. El punto más importante a tener en cuenta a la hora de elegir una función de evaluación es que función escogida debe guiar la evolución de nuestra población hacia soluciones que converjan al óptimo.

El objetivo en el problema que nos ocupa es conseguir maximizar el valor del contenido de la mochila, sin sobrepasar la capacidad máxima de la misma. Una buena aproximación para construir nuestra función de evaluación sería sumar el valor de todos los objetos que incluimos dentro de la mochila.

Junto a esta medida del valor total de la mochila, debemos además tener en cuenta que el volumen total de los objetos seleccionados no puede sobrepasar la capacidad máxima. Dada una solución de nuestra población tenemos dos posibilidades:

- Que la solución no sobrepase la capacidad total de la mochila, esto es, que sea *factible*.

- Que la solución sobrepase la capacidad máxima de la mochila y sea *no factible*.

Una solución no factible sale del dominio de nuestro problema y no podemos considerarla como una solución válida. Tenemos distintas posibilidades con respecto a las soluciones no factibles: ignorarlas, transformarlas, corregirlas, etc.

En la implementación de nuestro algoritmo genético, se estimó que era bastante interesante desarrollar distintas funciones de evaluación para comparar su desempeño. Es por ello que tenemos tres aproximaciones distintas para nuestra función de evaluación a las que hemos llamado decodificadora, reparadora y penalizadora.

3.3.3.1. Función Decodificadora

La función a la que hemos denominado *decodificadora* trata las soluciones no factibles como si fueran factibles. Para ello, dada una solución determinada, escoge de aquella los objetos más apropiados para rellenar la mochila de manera óptima.

En el caso en que la solución fuera inicialmente factible, todos los objetos pertenecientes a esta solución serán tomados en cuenta. En caso de no ser factible, hemos de escoger qué objetos consideraremos. Para tomar los objetos que pertenecerán a nuestra solución se ha hecho uso de una estrategia voraz. La heurística empleada para implementación devoradora ha sido considerar aquellos objetos con mejor relación de valor por unidad de volumen.

Dispondremos para ello de una estructura que representa a nuestros objetos ordenados en función del criterio valor / volumen. Recorreremos los objetos en el orden determinado por esta estructura. Para cada uno de ellos, comprobamos que

al incluirlo en la mochila la capacidad de la misma no sea excedida. Si es así, será incluido. Si sobrepasa la capacidad de la mochila, no será tomado en consideración y se pasa al siguiente.

Vamos a analizar la ejecución del algoritmo para un problema concreto. Disponemos de una mochila de capacidad 20 unidades y una colección de objetos tal y como sigue.

Objeto	Valor	Volumen	Ratio Valor/Volumen
1	1	10	0.10
2	2	5	0.40
3	3	2	1.50
4	2	9	0.22
5	8	3	2.66
6	6	1	6.00
7	7	9	0.77
8	4	2	2.00
9	7	6	1.17
10	1	7	0.44

Tabla 3.3: Colección de objetos con ratio valor/volumen calculado

Asimismo, una de las soluciones generadas tiene la siguiente forma:

0	1	1	0	1	1	1	0	1	0

El volumen total quedaría: $0*10+1*5+1*2+0*9+1*3+1*1+1*9+0*2+1*6+0*9=26$. Al ser la capacidad de la mochila de 20 unidades, nos encontramos ante una solución no factible.

En casos como este, la función de evaluación irá recorriendo la solución en el orden establecido por los ratios, ordenados de mayor a menor. En nuestro ejemplo el orden en que tomaremos los elementos será: 6, 5, 8, 3, 9, 7, 10, 2, 4, 1.

Vamos a comprobar cómo se evalúa la solución por medio de la siguiente tabla, donde se ha trasladado la solución del ejemplo ordenada en función del ratio.

Paso	Elemento	Solución	Valor	Volumen	Valor Total	Volumen Total
1	6	1	6	1	6	1
2	5	1	8	3	14	4
3	8	0	4	2	14	4
4	3	1	3	2	17	6
5	9	1	7	6	24	12
6	7	1	7	9	24	12
7	10	0	1	7	24	12
8	2	1	2	5	26	17
9	4	0	2	9	26	17
10	1	0	1	10	26	17

Tabla 3.4: Ejemplo de ejecución de función de evaluación decodificadora

Las acciones realizadas en cada paso serían:

Paso 1: La mochila inicialmente está vacía y la solución nos indica que el elemento 6 debe ir en la mochila. Como tenemos espacio, introducimos el objeto 6 en la mochila. El valor total será 6 y el volumen total 1 (exactamente los del objeto).

Paso 2: El volumen del objeto es 3, luego cabe en la mochila. Además, la solución incluye a este objeto. Introducimos el objeto en la mochila, quedando el valor total como 6+8=14 y el volumen 1+3=4.

Paso 3: Este objeto no pertenece a la solución. En la posición que le corresponde aparece un «0» en la cadena de solución por lo que ni siquiera lo consideramos.

Paso 4: El volumen del objeto es 2 y, siendo el volumen ocupado de la mochila de 4, podemos incluirlo. El nuevo volumen de la mochila quedará 4+2=6 y el valor total 14+3=17.

Paso 5: El volumen en este caso, para el objeto 5, es de 6 unidades. Aún cabe en la mochila. Lo incluimos y queda el volumen total en 6+6=12 unidades y el valor total será 17+7=24.

Paso 6: Este paso es realmente importante. El objeto 7 pertenece a la solución, por lo que deberíamos incluirlo en la mochila. Sin embargo, el volumen ocupado actualmente es de 12 unidades, y el del objeto es de 9. Tendríamos por

tanto un volumen total de 21 unidades, lo cual es imposible porque estaríamos excediendo la capacidad de la mochila. La solución sería no factible. Para transformarla en factible lo que hacemos es no considerar este objeto como perteneciente a la mochila y pasar al siguiente. Por tanto el valor total sigue siendo 24 y el volumen 12.

Paso 7: Este objeto no es considerado en la solución, luego no lo incluimos.

Paso 8: El objeto 2 pertenece a la solución y su volumen es 5. Como el volumen actualmente ocupado en la mochila es 12, aún disponemos de 7 unidades de volumen libres. Introducimos el objeto en la mochila quedando un valor total de 24+2 = 26 y un volumen de 12+5 = 17.

Paso 9: El objeto no pertenece a la solución.

Paso 10: El objeto tampoco pertenece a la solución.

Por tanto, aunque la solución nos indica que debemos introducir en la mochila los objetos 6, 5, 3, 9, 7 y 2, finalmente hemos descartado el objeto 7 para hacer que la solución sea factible.

Es importante destacar que la función de evaluación asignará finalmente a la solución un valor de 17, que sería el correspondiente a la suma del valor de los objetos introducidos en la mochila para la versión factible de la solución.

3.3.3.2. Función Reparadora

La función reparadora sigue la misma filosofía de la decodificadora. Podemos tener soluciones factibles y no factibles. En el caso de que la solución sea no factible, se realiza un procedimiento prácticamente idéntico al anterior, con la salvedad de que la solución se modifica.

En el ejemplo anterior, con la función decodificadora, teníamos la siguiente solución:

0	1	1	0	1	1	1	0	1	0

Vimos como en el paso 6 del procedimiento, descartábamos el objeto 7 porque superábamos el valor máximo de capacidad de la mochila. No se tenía en cuenta ese objeto para asignar la evaluación a la solución.

La filosofía de la función reparadora es exactamente la misma, sólo que en este caso, la solución es modificada. A la hora de evaluar el individuo que aporta esta solución, al considerar el objeto 7, determinaríamos asimismo que no puede pertenecer a la mochila y *corregiríamos* la solución para que no aparezca dicho objeto quedando de la siguiente forma:

0	1	1	0	1	1	0	0	1	0

La evaluación sería semejante a la función decodificadora, es decir, 17. La diferencia fundamental radica en que, al haber modificado la solución, sólo tendremos soluciones factibles a la hora de aplicar el resto de operadores como, por ejemplo, el de recombinación. El efecto que esta modificación de las soluciones pudiera tener sobre el desempeño global de nuestra solución será motivo de análisis en el capítulo dedicado a la evaluación.

3.3.3.3. Función Penalizadora

La principal diferencia de la función penalizadora con respecto a las dos anteriores es que no hace distinción entre soluciones factibles y no factibles. Esta función trata a todas las soluciones como si fueran válidas. Aún así necesitamos un mecanismo para promocionar aquellas soluciones válidas sobre las que no lo sean. Para conseguir esto, la función penalizadora asigna valores negativos a la evaluación en el caso en que una solución no sea factible.

Dada una solución no factible, la suma total del volumen de los objetos que plantea introducir en la mochila será mayor que la capacidad de la misma. En esta situación, la función penalizadora asigna una evaluación a dicha solución igual a la capacidad de la mochila menos el volumen total ocupado por los objetos. De este modelo de evaluación podemos destacar distintas características:

- Todas las soluciones no factibles tendrán una evaluación negativa.

- Todas las soluciones factibles tienen una evaluación positiva.

- Dadas dos soluciones no factibles, aquella que esté más cerca de ser factible (la que sobrepase la capacidad de la mochila por un valor menor) tendrá mejor evaluación que aquella que dista mucho de una solución factible.

- Dadas dos soluciones factibles, la evaluación se corresponderá al valor total de los objetos introducidos en la mochila. Por tanto, cuanto mayor sea el valor mejor será la evaluación.

Veamos un ejemplo basado en el anterior que ya hemos analizado. La capacidad de la mochila en este caso era de 20 unidades y la solución que analizábamos era:

0	1	1	0	1	1	1	0	1	0

Ya vimos que dicha solución no era factible. La función penalizadora calculará el valor total de todos los objetos considerados y el volumen total de la colección. En nuestro caso el valor total sería de 33 unidades y el volumen 26. Como el volumen total sobrepasa la capacidad de la mochila en seis unidades, la evaluación asignada a dicha solución será:

$$Evaluacion = capacidad - volumen\_total = 20 - 26 = -6$$

Si «empeoramos» la solución introduciendo algún otro objeto, podríamos tener la siguiente solución:

0	1	1	0	1	1	1	0	1	1

Añadiendo el objeto número 10 a la solución, tendríamos que el valor total sería 34 unidades y el volumen ocupado de 36 unidades. La evaluación quedará por tanto:

$$Evaluacion = capacidad - volumen\_total = 20 - 36 = -16$$

Comprobamos como, estando más alejada de la capacidad máxima de la mochila, esta solución tiene una evaluación peor que la anterior. Ambas serían no factibles, pero a la hora de compararlas, la primera sería proporcionalmente mejor que la segunda.

Por último veamos el caso en que la solución sea la equivalente factible que analizamos en los dos apartados anteriores. Dicha solución sería de la forma:

0	1	1	0	1	1	0	0	1	0

El volumen total sería 17 y el valor que le correspondería sería 26. La evaluación en este caso, al ser el volumen ocupado menor que la capacidad, se correspondería con el valor total de la colección de objetos, esto es, 26 unidades.

3.3.4. Selección de Padres

Por medio de la *selección de padres*, tomaremos aquellos elementos de nuestra población original que vamos a recombinar para generar una nueva población. Es de vital importancia tomar una muestra significativa de los mejores miembros intentando, al mismo tiempo, no eliminar la diversidad de la población.

En nuestro caso vamos a ir tomando los elementos de nuestra población por grupos de manera aleatoria. Para cada uno de estos grupos, se elegirá el elemento más apto por medio de un método de selección denominado «selección por torneo».

La selección por torneo enfrenta a una serie de elementos de la población entre ellos devolviendo aquel que tiene mejor evaluación. La cantidad de elementos de la población que participarán en el torneo será variable y la denominaremos *tamaño del torneo*. Cuanto mayor sea el tamaño del torneo, la probabilidad de escoger los mejores elementos de la población será mayor. La implementación realizada para calcular el vencedor de un grupo ha sido enfrentar a todos los elementos uno a uno, contabilizando el número total de victorias. Si finalmente hay algún empate en el número de victorias dando más de un vencedor, se vuelve a realizar un torneo entre estos vencedores.

Cabe destacar que el enfrentamiento entre dos individuos cualquiera en el torneo escogerá al que tenga mejor evaluación con un 95 % de posibilidades. De esta forma dejamos una pequeña posibilidad de pertenecer al grupo de padres a elementos con peor evaluación para mantener la diversidad.

Por cada torneo realizado, escogeremos un padre para la siguiente generación calculada. El total de padres que vamos a necesitar será igual al tamaño de la po-

blación, por lo que repetimos el procedimiento de selección por torneo tantas veces como sea necesario. Cabe destacar que la selección se realiza siempre sobre el conjunto completo de elementos de nuestra población. Siendo así, un mismo elemento puede llegar a ser seleccionado varias veces para formar parte del conjunto de padres, lo cual es lógico si pensamos que un elemento especialmente apto tendrá más posibilidades de procrear que otro menos apto.

Una vez tenemos el conjunto completo de padres, podemos pasar a generar su descendencia.

3.3.5. Operador de Cruce

El *operador de cruce* permitirá crear nuevos miembros para nuestra población recombinando los padres que previamente hemos seleccionado. Por cada pareja de padres generamos dos descendientes. Estos descendientes son el resultado de mezclar las características de sus progenitores.

Vimos anteriormente que, para cada elemento de nuestra población, su representación viene dada por una cadena de ceros y unos. Cada uno de los elementos de la cadena representa un objeto y su valor indica si el objeto i-ésimo pertenecerá o no a la configuración final de la mochila. En el caso del problema de la mochila para n objetos, podemos representar a los progenitores como sigue:

a_1	a_2	...	a_{i-1}	a_i	a_{i+1}	...	a_n

b_1	b_2	...	b_{i-1}	b_i	b_{i+1}	...	b_n

En nuestra implementación vamos a realizar un cruce entre los progenitores por un único punto. Dados los dos ejemplos de padres anteriores, el cruce por el punto i-ésimo daría como resultado los siguientes descendientes:

a_1	a_2	...	a_{i-1}	b_i	b_{i+1}	...	b_n

b_1	b_2	...	b_{i-1}	a_i	a_{i+1}	...	a_n

De esta forma tendríamos dos nuevas soluciones que añadir a nuestro conjunto de descendientes. La implementación realizada lleva a cabo este tipo de cruce entre los padres para la recombinación. Además de esto, el punto sobre el que hacer el cruce es escogido aleatoriamente para cada pareja de padres a recombinar.

Cada uno de los descendiente generados pasan a formar parte de la nueva generación.

3.3.6. Operadores de Mutación

El *operador de mutación* proporciona una nueva alternativa para generar individuos diferenciados de los originales. Estos cambios ayudan a mantener la población convenientemente diversificada. En el caso en que los mejores individuos pertenezcan todos a la misma región del espacio de búsqueda y se encuentren en torno a un máximo local, la mutación puede propiciar que algunas de las soluciones se muevan a otras localizaciones del espacio de búsqueda [Esh00].

Para nuestro algoritmo genético, la mutación consistirá en cambiar aleatoriamente valores de cada elemento. Al ser una cadena de ceros y unos, elegiremos arbitrariamente el elemento a modificar y alternaremos su valor. La cantidad de elementos que modificaremos vendrá definida por los parámetros que le pasemos a nuestro algoritmo. Típicamente se suele escoger una tasa de mutación de un elemento por cadena [Spe00]. Para el problema genérico de n objetos, las cadenas serán de longitud n y la tasa de mutación sería $1/n$.

Es importante señalar, que una vez que hemos mutado los nuevos elementos, debemos volver a calcular su evaluación que se habrá visto modificada por los cambios producidos.

3.3.7. Selección de Supervivientes

La *selección de supervivientes* es la encargada de tomar los elementos que conformarán la siguiente generación. Disponemos de dos tipos de estrategias:

- *Generacional*. Toda la población original es sustituida por los nuevos elementos generados. Debemos generar al menos tantos descendientes como elementos tenga

nuestra población.

- *De Estado Estacionario* («steady state»). Mediante esta estrategia, enfrentamos a los elementos que acabamos de crear con los elementos actuales de nuestra población. De todos ellos, solo los más aptos pasarán a la nueva generación. Con este método no necesitamos generar un número de descendientes tal que sea capaz de sustituir completamente la población sino que podemos generar un número arbitrario de nuevos elementos, pudiendo ser inferior al tamaño de la población.

Además de estas dos aproximaciones, podemos implementar otro tipo de estrategias como la penalización por edad de los miembros de nuestra población.

En nuestro caso concreto hemos decidido aplicar una *estrategia generacional*. Por cada nueva generación creamos un número de descendientes igual al tamaño de la población y la sustituimos completamente. Aún así, tenemos la opción de establecer por medio de los parámetros del algoritmo una estrategia *elitista*. En este caso, se sustituye la población completamente pero se conserva el mejor individuo de la población anterior.

Capítulo 4

Entorno de Aplicación Diseñado

A lo largo de este capítulo vamos a analizar la aplicación desarrollada para resolver el *problema de la mochila* por medio de un *algoritmo genético*. Como parte fundamental del presente proyecto se planteó la necesidad de realizar una aplicación que pudiese servir al usuario para visualizar gráficamente la ejecución y el comportamiento de nuestro algoritmo genético. Gracias a esta aplicación podremos comprobar el desempeño de nuestro algoritmo ante distintos problemas y distintas configuraciones. Podemos modificar, de igual forma, parte de los parámetros fundamentales del algoritmo y comprobar la respuesta que obtenemos.

El desarrollo ha sido realizado en lenguaje Java y se ha mantenido durante todo su desarrollo la filosofía de primar el aspecto gráfico para que sea fácil de entender y eficaz a la hora de interpretar el comportamiento del algoritmo.

4.1. Jerarquía de Clases

Nuestro programa ha sido dividido en dos paquetes fundamentales:

- mochila: Incluye todas las clases encargadas de gestionar la mochila, el algoritmo genético, los operadores, la lista de objetos, etc.
- mochila.interfaz: Donde se encuentra el apartado gráfico del proyecto. En él incluimos el código de todas las pantallas, así como de los gráficos que representan la ejecución de nuestro algoritmo.

4.1.1. Paquete mochila

El paquete mochila está compuesto por los siguientes ficheros:

AG.java: En este fichero se encuentra todo el código referente al algoritmo genético. En él se generan las soluciones iniciales, se evalúa la población (incluyendo las tres funciones evaluadoras desarrolladas) y se engarzan los distintos procedimientos para calcular la siguiente generación. Además incluye determinadas funciones para calcular el mejor y peor individuo, el valor medio de la población, la representación de los elementos... etc.

Comparador.java: Una clase auxiliar que nos sirve para poder comparar dos genotipos.

Genotipo.java: Esta clase implementa la representación de nuestros individuos. Cada uno de ellos viene representado por una cadena con la colección de objetos que representa y su evaluación. Además, incluye los operadores de torneo y mutación.

Mochila.java: Esta clase recrea el objeto mochila. Incluye y gestiona la referencia al algoritmo genético que va a dirigir la ejecución de nuestro problema. Además, controla la lista de objetos sobre los que vamos a trabajar.

Objeto.java: Clase representativa de los distintos objetos que va a componer nuestra colección. Cada uno de estos objetos vendrá definido por su valor y su volumen.

Simulacion.java: Clase auxiliar utilizada para simular la ejecución del algoritmo y poder exportar los resultados en texto para su posterior análisis.

4.1.2. Paquete mochila.interfaz

El paquete con el apartado gráfico de la aplicación está compuesto por las siguientes clases:

acercaUI.java: Pantalla con información general sobre el proyecto.

etiquetaGrafica.java: Clase para representar etiquetas en formato gráfico. Esto nos permite modificarlas para que se adapten a nuestra representación.

GraficoAG.java: Gráfico que muestra la evolución de nuestro algoritmo genético. Muestra en todo momento el valor máximo alcanzado en la evaluación de nuestra población a lo largo del tiempo.

GraficoMochila.java: Representa la disposición de los objetos dentro de la mochila. Para ello toma el mejor elemento de la población actual.

GraficoObjeto.java: Muestra la colección completa de objetos, de forma que su anchura representa el volumen que ocupan y la intensidad del color el ratio valor/volumen. Podemos observar asimismo en el gráfico qué objetos han sido seleccionados al ser su altura el doble que la del resto de objetos.

mochilaUI.java: Clase principal de nuestra aplicación. Gestiona los parámetros pasados como argumentos y el interfaz gráfico para el usuario.

objetosUI.java: Código de la pantalla para la gestión de los objetos. Desde ella podemos añadir o eliminar objetos de nuestra colección.

preferenciasUI.java: Esta clase controla las preferencias de nuestro algoritmo genético y de la mochila. Podemos modificar características como la capacidad de la mochila, el tamaño de la población, el tamaño del torneo, la función de evaluación, el ratio de mutación, el tamaño del torneo y el elitismo.

4.2. Interfaz Gráfico

4.2.1. Arranque de la Aplicación

Como comentamos anteriormente, nuestra aplicación ha sido desarrollada en lenguaje Java. Concretamente, la versión del compilador utilizada ha sido la 1.6, aunque funciona con versiones posteriores a la 1.5 de JRE (Java Runtime Environment). El código ha sido compilado en un fichero jar autocontenido.

La ejecución puede realizarse desde la línea de comandos con el comando:

```
$ java -jar mochila.jar
```

De esta forma, el programa cargaría con una colección de objetos generada aleatoriamente y con los parámetros del algoritmo genético y la mochila por defecto. Podemos detallar una serie de parámetros por línea de comandos para restaurar una entorno de trabajo determinado, siendo todos ellos opcionales. Los parámetros son:

```
$ java -jar mochila.jar [-p nombre_fichero] [-o nombre_fichero]
[-c capacidad]
```

-p Fichero que contiene la configuración para el algoritmo genético.

-o Fichero con la descripción de la colección de objetos a utilizar.

-c Capacidad de la mochila.

Veamos el formato de los distintos ficheros, comenzando por los parámetros del algoritmo genético. Podemos ver un ejemplo del mismo en la tabla de la Tabla 4.1.

```
# Numero de elementos en la población
TamPoblacion = 10

# Función de evaluación
# 1: Decodificadora
# 2: Reparadora
# 3: Penalizadora
FunEval = 2

# Tamaño del torneo. Rango: [2, TamPoblacion].
# En caso de ser mayor que el tamaño de la población quedara
# fijado al tamaño de la misma
TamTorneo = 2

# Ratio de mutación
RatioMutacion = 0.01333

# Elitismo
# 1: Con Elitismo
# 2: Sin Elitismo
Elitismo = 1
```

Tabla 4.1: Formato de fichero de parámetros

TamPoblacion Corresponde al tamaño de nuestra población. Por cada generación se crearán tantas soluciones como indique este parámetro.

FunEval Indica la función de evaluación utilizada en nuestro algoritmo genético. Un 1 corresponderá a la función decodificadora, 2 a la reparadora y el 3 a la penalizadora.

TamTorneo Tamaño que deseamos para el torneo. El rango estimado para el mismo será de dos elementos como mínimo y el tamaño total de la población como máximo. En caso de especificar un valor menor, quedará fijado a dos y, en caso de que sea mayor que el tamaño de la población, quedará fijado a dicho tamaño.

RatioMutacion Ratio de mutación. Típicamente será de un elemento por tamaño de la cadena (que corresponde, a su vez, al número de objetos). Aún así podemos modificarlo para comprobar cómo afecta a la ejecución del algoritmo. El rango será de cero a uno.

Elitismo Un valor de 1 indica que se aplique elitismo. De esta forma, el mejor elemento de la generación precedente pasará a la siguiente generación. En caso que establezcamos su valor a 2, nuestro algoritmo genético será puramente generacional, sustituyendo completamente una generación por la siguiente.

Visto el fichero de parámetros, veamos el formato del fichero de descripción de los objetos. Lo usaremos para guardar colecciones de objetos previamente definidos. El fichero tiene la siguiente forma:

```
Objeto_1_n = Objeto_1
Objeto_1_v = 1
Objeto_1_p = 66
Objeto_2_n = Objeto_2
Objeto_2_v = 33
Objeto_2_p = 78
Objeto_3_n = Objeto_3
Objeto_3_v = 25
Objeto_3_p = 53
Objeto_4_n = Objeto_4
Objeto_4_v = 86
Objeto_4_p = 71
Objeto_5_n = Objeto_5
Objeto_5_v = 78
Objeto_5_p = 42
[...]
```

Tabla 4.2: Formato de fichero de objetos

Los objetos se definen en grupos de tres líneas. Cada una de estas líneas empiezan con un descriptor del objeto de la forma Objeto_num donde el número variará desde 1 hasta el número total de objetos que tengamos. Cada una de estas tres líneas están diferenciadas por el sufijo que tendrá el descriptor del objeto de la siguiente forma:

Objeto_num_n Nombre que daremos al objeto en la aplicación

Objeto_num_v Valor del objeto

Objeto_num_p Volumen ocupado por el objeto

Junto a la aplicación se incluyen varios ficheros de ejemplo con colecciones de objetos que van desde los 100 a los 100000 elementos.

4.2.2. Pantalla Principal de la Aplicación

Una vez cargada la aplicación, tendremos a nuestra disposición un interfaz como el que se muestra a continuación.

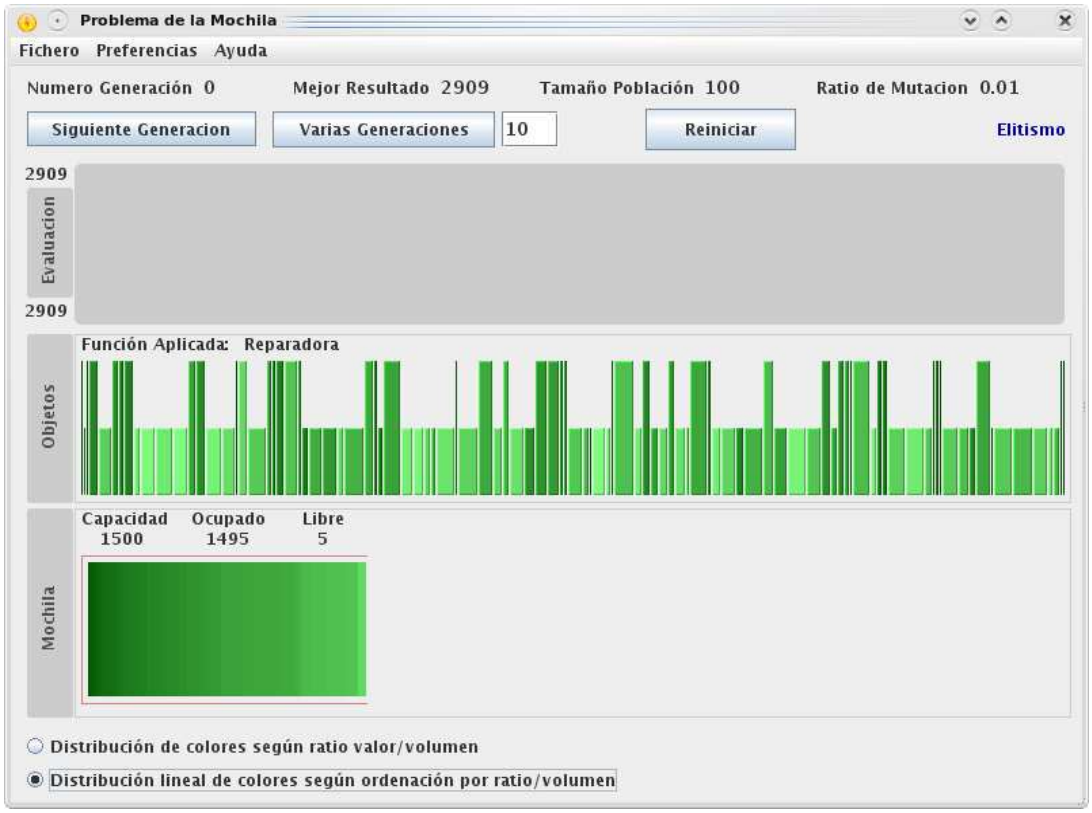


Figura 4.1: Pantalla principal de la aplicación

A través de esta pantalla podemos realizar las operaciones básicas de nuestro algoritmo genético. Podemos distinguir varias partes claramente diferenciadas:

Barra de menú: En ella tenemos los menús que nos permitirán modificar preferencias, cargar y guardar colecciones de objetos, etc.

Información básica y botones de operación: Con los botones podemos avanzar en el cálculo de nuevas generaciones de nuestro algoritmo. Además, mostramos la información básica sobre los parámetros que estamos considerando para la simulación.

Gráfico de Evaluación: Es un gráfico que muestra la evaluación del mejor miembro de nuestra población a lo largo del tiempo.

Gráfico de Objetos: Muestra una representación gráfica de la colección de objetos sobre la que estamos trabajando.

Gráfico de Mochila: Representación gráfica de nuestra mochila, con la disposición de los objetos que hemos introducido en ella hasta el momento y un detalle sobre el espacio ocupado y disponible. A cada objeto le asignamos un color en función del ratio valor/volumen para poder visualizar el valor ponderado de cada uno de ellos.

Modo de visualización: Dos botones de selección nos permiten variar la forma en que los datos serán presentados gráficamente.

En los siguientes subapartados comentaremos en detalle cada una de estas partes.

4.2.3. Barra de Menú

La barra de menú contiene tres entradas:

- Fichero
- Preferencias
- Ayuda

Desde el menú «Fichero» podemos «Guardar» la lista de objetos actual para poder recuperarlos más tarde, «Cargar» una lista de objetos previamente guardada, y «Salir» de la aplicación.

El menú «Preferencias» nos permite acceder a las pantallas de «Parámetros» y «Objetos». Estas pantallas nos permiten configurar los parámetros básicos de la mochila y el algoritmo genético, así como la lista de objetos sobre la que estamos trabajando, respectivamente.

El menú «Ayuda» muestra información básica sobre la aplicación.

4.2.4. Botones e Información Básica

En la parte superior de la aplicación, disponemos de la información básica sobre la ejecución de nuestro algoritmo genético, así como los botones básicos de operación.

Figura 4.2: Botones e información básica de la aplicación

Los campos de información son:

Numero Generación: Nos indica el número de generaciones que hemos calculado. Comenzamos por la generación 0 y vamos incrementando este número por cada paso de la aplicación.

Mejor Resultado: Evaluación del mejor elemento de la población actual. En el caso en que no tengamos activada la función de elitismo, puede no ser la mejor solución encontrada hasta ahora. Esta información la podemos consultar en el gráfico de evaluación de la población.

Tamaño Población: Número de elementos que compone nuestra población.

Ratio de mutación: Ratio de mutación escogido para nuestro algoritmo genético.

Elitismo: Esta etiqueta tiene dos estados. Si es de color azul significa que el elitismo está activado y, por tanto, el mejor elemento de una generación pasará a la siguiente. En el caso en que lo encontremos de color rojo significará que la opción de elitismo está desactivada.

En cuanto a los botones de operación son:

Siguiente Generación: Por cada pulsación en este botón se calculará la siguiente generación para nuestro problema, actualizándose la información pertinente.

Varias Generaciones: Calcula tantas generaciones como nos indique el número a su derecha. El cálculo de varias generaciones se hace a modo de animación, de forma que podamos observar en los gráficos cómo va variando el estado de nuestro algoritmo.

Reiniciar: Restablecemos el estado original de nuestra simulación.

4.2.5. Configuración de Objetos

La pantalla de configuración de objetos nos permite añadir y eliminar objetos de nuestra colección.

Figura 4.3: Configuración de objetos

Para añadir un objeto introducimos su nombre, el valor y el volumen que queremos asignarle, pulsando posteriormente el botón de «añadir». Si lo que queremos es borrar un objeto, seleccionamos con el ratón el objeto de la lista y pulsamos el botón de «borrar». Podemos seleccionar rangos de objetos marcándolos mientras pulsamos la tecla «shift», o seleccionar varios objetos alternos dejando pulsada la tecla «control» a la hora de seleccionarlos con el ratón.

4.2.6. Configuración de Preferencias

Desde la pantalla de configuración de preferencias podemos modificar las características de nuestra mochila y del algoritmo genético.

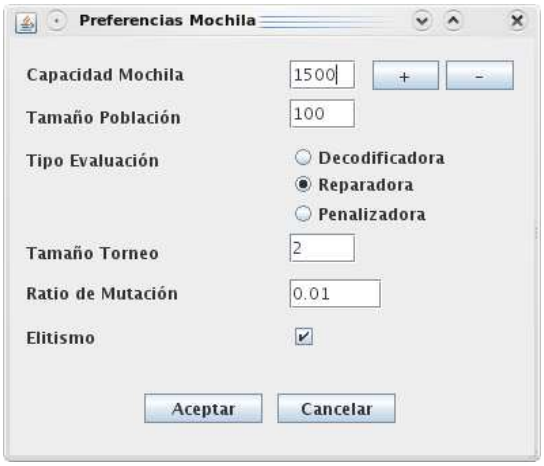


Figura 4.4: Parámetros de la aplicación

Los parámetros que podemos modificar son:

Capacidad Mochila: Capacidad de la mochila. Se corresponde con la capacidad aceptada por la mochila. Variará de 1 a 9999 unidades de volumen.

Tamaño Población: Cantidad de elementos que componen nuestra población.

Tipo Evaluación: Escogemos la función de evaluación que vamos a utilizar de las tres implementadas: decodificadora, reparadora o penalizadora.

Tamaño Torneo: Número de elementos que formarán un grupo del que saldrá cada uno de los padres en la etapa de selección de padres de nuestro algoritmo. El tamaño mínimo será 2. El máximo depende del tamaño de la población. Si asignamos un tamaño mayor al de la población, el tamaño del torneo quedará fijado a esta.

Ratio de Mutación: Tasa de mutación para los elementos de la población. Típicamente se asigna el inverso del tamaño de la cadena, siendo el tamaño de ésta igual al número de objetos con el que estemos trabajando. Para 100 objetos sería 0.01, aunque podemos modificarlo para comprobar el rendimiento con distintos valores. Los valores aceptados van de 0 a 1 (que equivaldrían al 0 y el 100 %).

Elitismo: Marcamos si deseamos que se aplique elitismo, con lo que el elemento más apto de una generación pasará a la siguiente, o queremos desactivarlo.

4.2.7. Gráfico de Evaluación

El gráfico de evaluación muestra el valor de la evaluación del mejor elemento de nuestra población a lo largo de todo el ciclo de vida de nuestro algoritmo genético. En el eje de ordenadas representamos el valor de la evaluación, mientras que cada punto del eje de abcisas representa una generación. A la izquierda podemos observar en todo momento el mínimo y máximo histórico registrados durante todo el ciclo de vida del algoritmo.

Figura 4.5: Gráfico de evaluación de generaciones

4.2.8. Gráfico de Distribución de Objetos

El gráfico de distribución de objetos muestra una representación gráfica de la colección de objetos que consideramos para rellenar la mochila. Cada una de las barras verticales representa un objeto. El ancho de cada barra corresponde al volumen ocupado por cada objeto. A mayor ancho, mayor volumen. El color de la barra representa el ratio valor/volumen, de forma que los objetos más oscuros representan objetos que en principio resultarán más ventajosos de cara a añadirlos a la mochila. Más adelante explicaremos los dos modos de representación implementados.

El tamaño de las barras representa la elección o no del objeto. Referidos siempre al elemento con mejor evaluación de nuestra población, aquellos objetos que hayan sido seleccionados para pertenecer a la mochila quedan representados con una altura doble. Los objetos con altura media son aquellos que no han sido seleccionados para pertenecer a la mochila.

A modo informativo incluimos la descripción de la función de evaluación que estamos utilizando en un momento determinado.

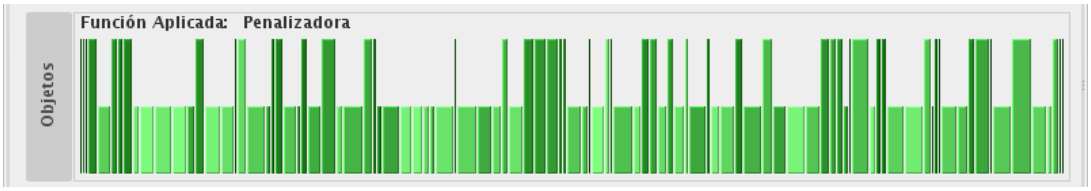


Figura 4.6: Gráfico de distribución de objetos

4.2.9. Gráfico de Distribución de Objetos en la Mochila

El gráfico de distribución de objetos en la mochila muestra los distintos objetos que hemos incluido como solución del problema de la mochila. Además, incluye información sobre la ocupación de la mochila como la capacidad total, el espacio ocupado y el espacio libre. Al igual que el gráfico de objetos, representa al elemento con mejor evaluación de nuestra población.

En este gráfico representamos los objetos que en el gráfico de objetos tienen una altura doble, es decir, que han sido seleccionados para formar parte de la solución. Estos objetos son ordenados en función del ratio valor/volumen para que podamos observar mejor la evolución global de la mochila.

Figura 4.7: Gráfico de distribución de objetos en la mochila

En el caso en que la capacidad de la mochila sea sobrepasada por los objetos seleccionados, el gráfico tendrá tonos rojizos en lugar del verde habitual:

Figura 4.8: Gráfico de distribución de objetos en la mochila superando la capacidad

4.2.10. Formatos de Visualización de Objetos y Mochila

La representación del color de los objetos tanto en el gráfico de la colección completa como en el de la mochila se basa en el ratio valor/volumen. Este ratio es el que hemos usado en la heurística de selección de los objetos, por lo que es una buena medida para saber el valor relativo que un objeto aporta a la mochila a la hora de añadirlo. Un objeto más oscuro tendrá gran valor y pequeño volumen, por lo que resulta muy interesante su inclusión en la solución final.

Pueden aparecer problemas de visualización en el caso de distribuciones del ratio descompensadas. Por ejemplo, la aplicación al generar un problema aleatorio, escoge para el valor y el volumen valores aleatorios entre 1 y 100. Ambos valores por separado tienden a distribuirse uniformemente, pero al dividirlos la distribución deja de tener este comportamiento. La distribución resultante tiende a concentrar la mayoría de sus valores en torno al valor 1, estando la mitad de los elementos en el rango (0,1], y la otra mitad en el rango (1,100). Es por ello que podemos encontrarnos representaciones gráficas de la siguiente forma:

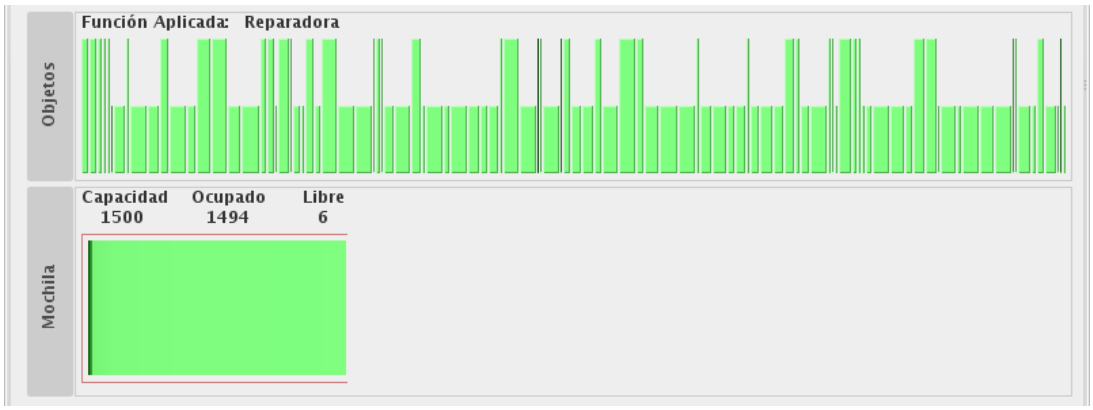


Figura 4.9: Distribución asimétrica del ratio valor/volumen

Observamos como tenemos unos pocos objetos con muy buen ratio, cercano al máximo, mientras que el resto tiene ratios en torno a 1. En estos casos conviene cambiar el modo de visualización utilizado.

Figura 4.10: Formatos de visualización de los objetos y la mochila

La vista seleccionada por defecto es «Distribución de colores según ratio valor/volumen». En situaciones como la anterior, si cambiamos la vista a «Distribución lineal de colores según ordenación por ratio/volumen», lo que hacemos es normalizar la distribución, asignando los colores a los objetos en función de la posición que ocuparían si los ordenáramos por ratios. El anterior ejemplo quedaría al cambiar la vista tal y como sigue:

Figura 4.11: Distribución normalizada para el ratio valor/volumen

Capítulo 5

Evaluación Empírica

Como parte fundamental del presente Proyecto de Fin de Carrera, incluimos una comparativa entre las distintas funciones de evaluación elegidas. Vamos a evaluar el comportamiento del algoritmo para cada una de estas funciones, modificando asimismo la cantidad de objetos, tamaño de la mochila y parámetros del algoritmo genético.

5.1. Metodología Para las Simulaciones

Para cada una de las funciones de evaluación, hemos realizado las correspondientes simulaciones sobre una base de cien ejecuciones. Al finalizar dichas ejecuciones, se ha calculado la media de la evaluación del mejor individuo obtenida para cada una de las generaciones. Los parámetros que se han ido modificando para comprobar los resultados han sido:

Número de Objetos El listado de objetos utilizado ha sido generado aleatoriamente. Por medio de un programa auxiliar se han generado tres ficheros de objetos con 100, 1000 y 10000 objetos respectivamente. El rango tanto para el valor como para el volumen de cada uno de los objetos se corresponde a un número aleatorio en el rango 1 a 100. Los ficheros con los objetos generados se incluyen en el CDROM de la memoria.

Capacidad de la Mochila En función del número de objetos, hemos variado la

capacidad de la mochila desde 500 unidades a varios millones. La capacidad ha sido ajustada de manera empírica para cada uno de los problemas de manera que no sea demasiado pequeña en problemas grandes ni demasiado grande para problemas con pocos objetos.

Tamaño de la Población Hemos jugado con dos valores fundamentales para las simulaciones, 10 elementos y 100 elementos.

Para cada una de las simulaciones, hemos generado cuatro gráficas:

1. Comparativa lineal entre las tres funciones de evaluación.
2. Comparativa logarítmica entre las tres funciones de evaluación.
3. Comparativa lineal entre la función reparadora y la decodificadora.
4. Comparativa logarítmica entra la función reparadora y la decodificadora.

El motivo de hacer esta distinción ha sido la gran diferencia existente en la mayoría de los casos entre las dos primeras funciones y la penalizadora. Para cada caso concreto mostraremos las gráficas más representativas. Las representaciones logarítmicas serán útiles para mostrar la evolución de la simulación para un número elevado de generaciones. Por la naturaleza del problema, nuestra evaluación tiende a crecer rápidamente durante las primeras generaciones para estabilizarse posteriormente. Por medio de la representación logarítimica podemos apreciar las diferencias entre las funciones tanto en las primeras generaciones como en la región en la que tienden a crecer más lentamente.

5.2. Experimentos Realizados

5.2.1. Comparativa Entre las Tres Funciones.

Presentamos para comenzar un problema sencillo. Disponemos de una colección de 100 objetos y vamos a ir incrementando la capacidad de la mochila, partiendo de 500 unidades para comprobar el comportamiento de las funciones.

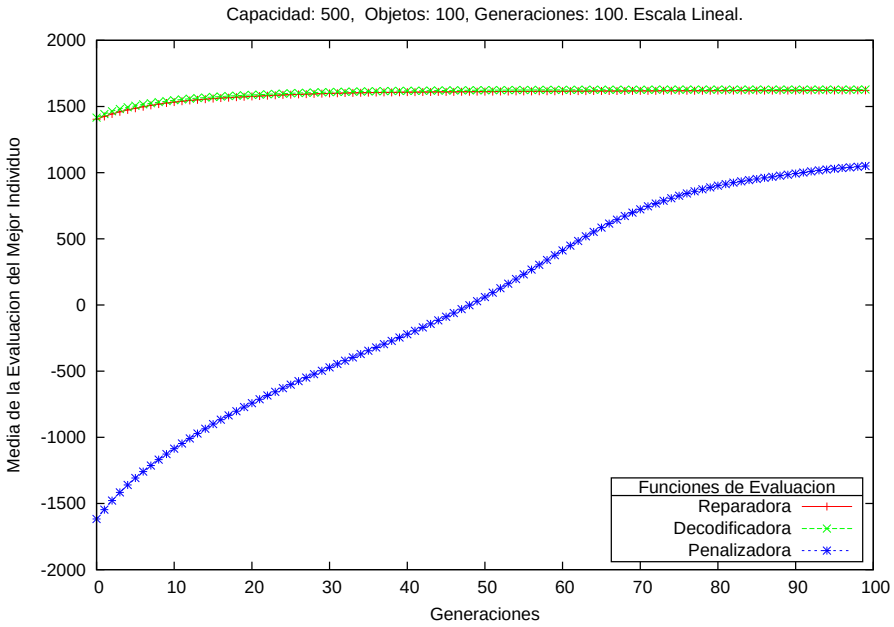


Figura 5.1: Capacidad: 500, Objetos: 100, Generaciones: 100. Escala Lineal

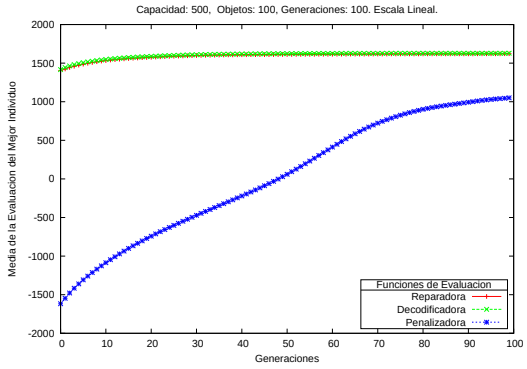
Comprobamos como las funciones reparadora y decodificadora son prácticamente indistinguibles. Por contra, la función penalizadora parte de lo que podríamos calificar de una situación de desventaja ya que, al ser la población pequeña y no haber ninguna solución inicial factible, comenzamos teniendo una evaluación negativa tal y como analizamos en el apartado correspondiente a la función de evaluación penalizadora.

El crecimiento de la función penalizadora varía en función de dos parámetros fundamentales:

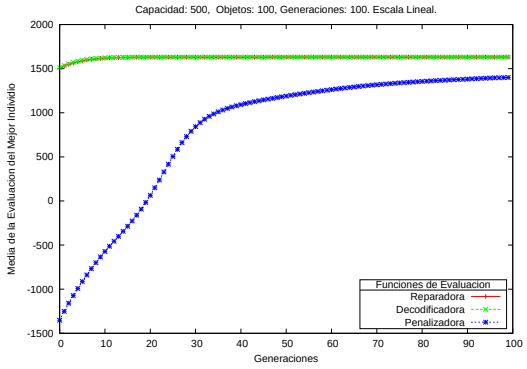
1. Tamaño de la población: ya que cuanto mayor sea el tamaño, más posibilidades habrá de que se llegue pronto a una solución factible y, por tanto, la evaluación sea positiva.

2. Capacidad de la mochila: Cuanto mayor es la capacidad de la mochila, igualmente antes aparecerán las primeras soluciones factibles. Es más fácil que una solución sea factible cuando la capacidad de la mochila es elevada y, de esta forma, una solución aleatoria tendrá más posibilidades de no sobrepasar la capacidad máxima.

Vamos a comparar la evolución de la función penalizadora, comparándola con la decodificadora y la reparadora. Para ello vamos a representar en paralelo dos casos idénticos pero con poblaciones de 10 y 100 elementos respectivamente. Bajo este planteamiento, vamos a comprobar cómo se ve modificada la función penalizadora cuando vamos aumentando la capacidad de la mochila.

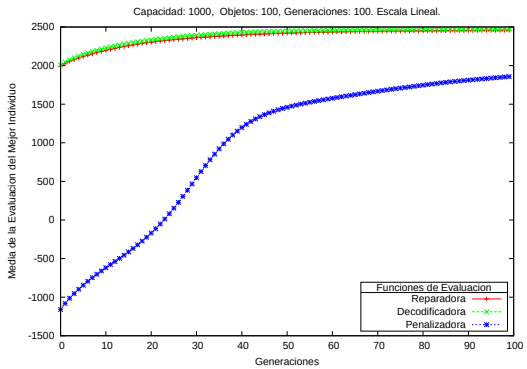


Población de 10 elementos

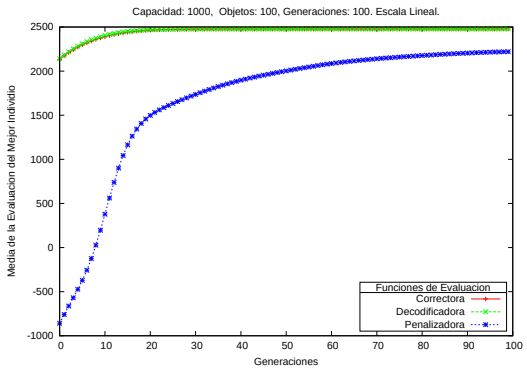


Población de 100 elementos

Figura 5.2: Capacidad: 500, Objetos: 100, Generaciones: 100.

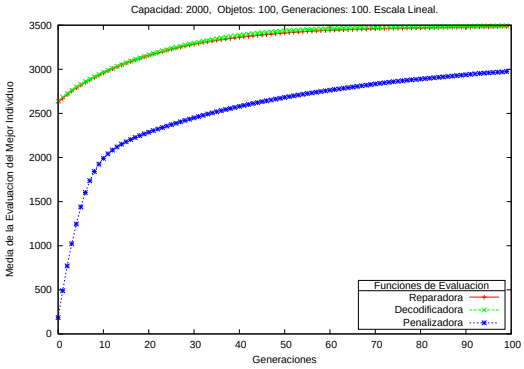


Población de 10 elementos

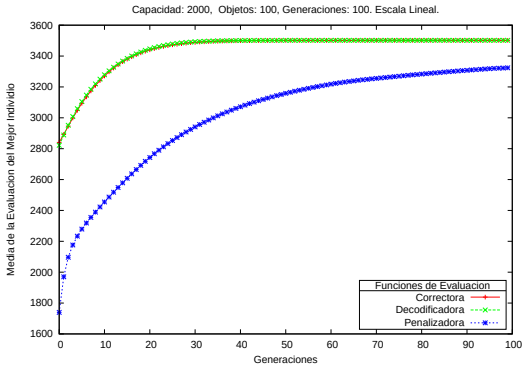


Población de 100 elementos

Figura 5.3: Capacidad: 1000, Objetos: 100, Generaciones: 100.

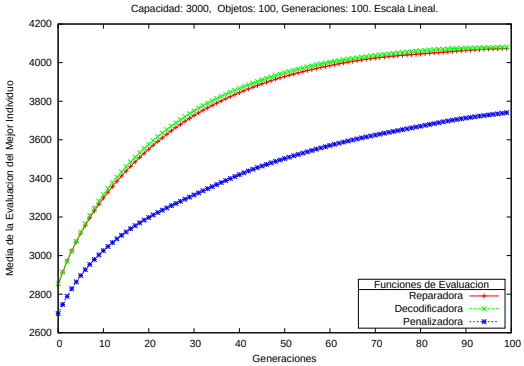


Población de 10 elementos

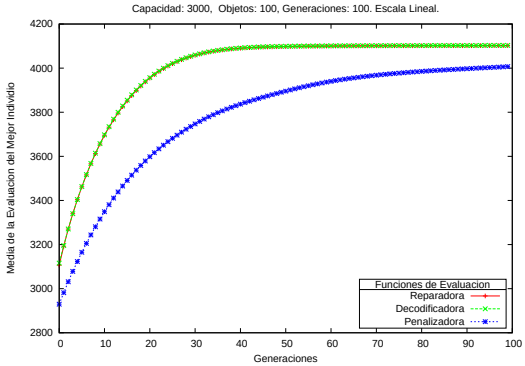


Población de 100 elementos

Figura 5.4: Capacidad: 2000, Objetos: 100, Generaciones: 100.

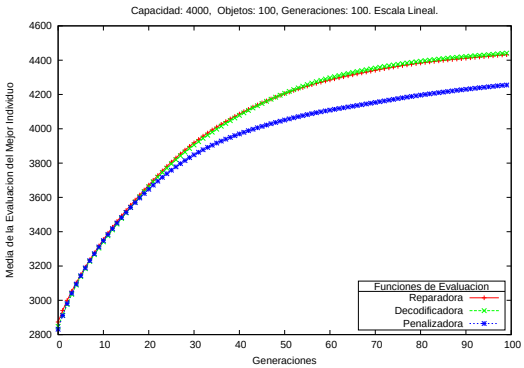


Población de 10 elementos

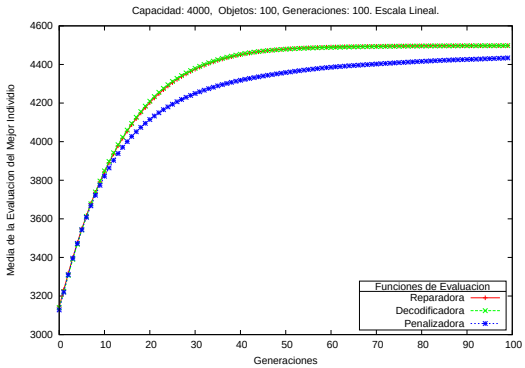


Población de 100 elementos

Figura 5.5: Capacidad: 3000, Objetos: 100, Generaciones: 100.



Población de 10 elementos



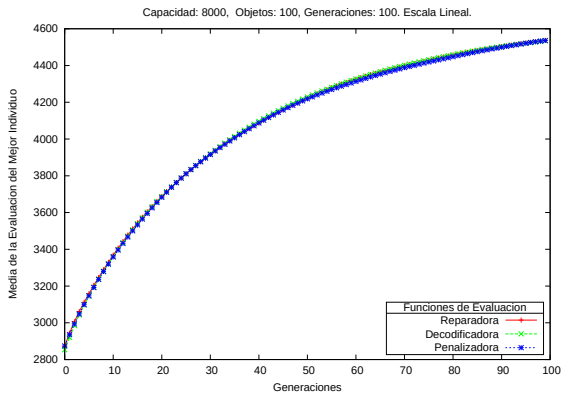
Población de 100 elementos

Figura 5.6: Capacidad: 4000, Objetos: 100, Generaciones: 100.

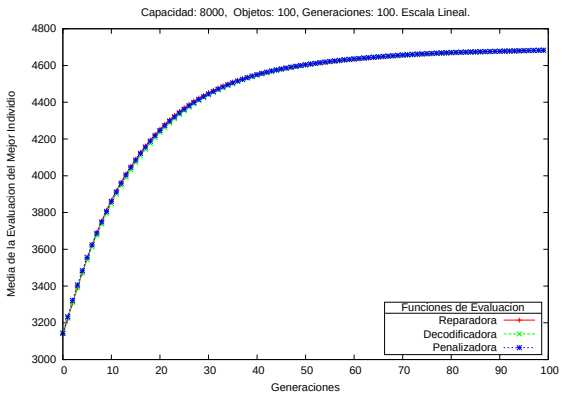
A medida que crece la población, más acusado es el crecimiento de la función

de evaluación penalizadora. Al tener más elementos en la población, hay más posibilidades de que alguno obtenga una buena evaluación. Además, cuanto mayor es capacidad de la mochila, antes se alcanzará el punto crítico en el que obtenemos soluciones factibles.

Como podemos ver, en torno a una capacidad de 4000 unidades de volumen, llegamos a un punto en el que la evaluación inicial es positiva para algunos miembros de la población. Llegados a este punto, el aumento en la capacidad de la mochila conlleva que la función se ajuste más aún al comportamiento de la reparadora y la decodificadora. Veamos como quedaría la distribución de evaluaciones aumentando la capacidad a 8000 unidades.



Población de 10 elementos



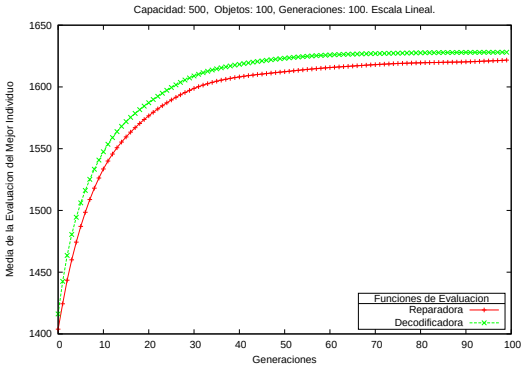
Población de 100 elementos

Figura 5.7: Capacidad: 8000, Objetos: 100, Generaciones: 100.

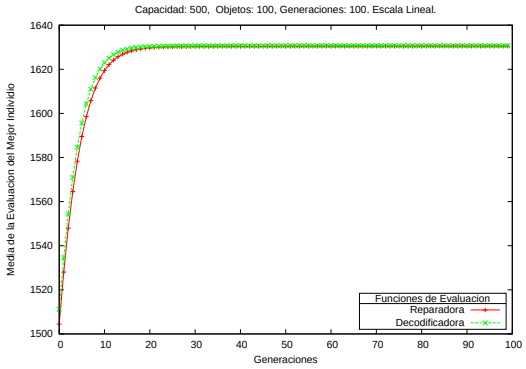
A partir de este punto las tres funciones son prácticamente indistinguibles unas de otras.

5.2.2. Diferencias Entre la Función Reparadora y la Decodificadora

Veamos ahora la misma simulación, pero comparando las funciones reparadora y decodificadora.



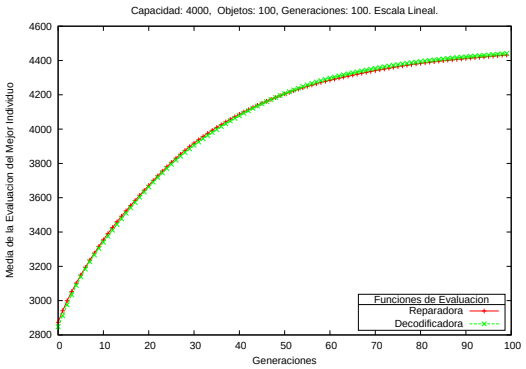
Población de 10 elementos



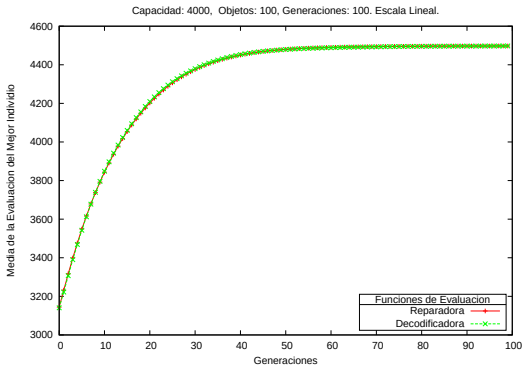
Población de 100 elementos

Figura 5.8: Capacidad: 500, Objetos: 100, Generaciones: 100.

Parece que hay una ligera diferencia a favor de la función decodificadora. Sobre todo es apreciable cuando la población es de diez elementos por contraposición a la de cien. Vamos comprobar otros casos aumentando la capacidad de la mochila en las figuras 5.9 y 5.10.



Población de 10 elementos

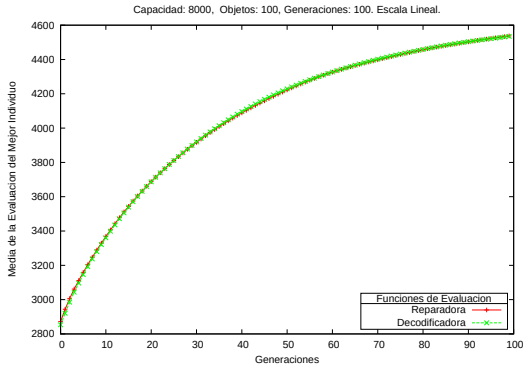


Población de 100 elementos

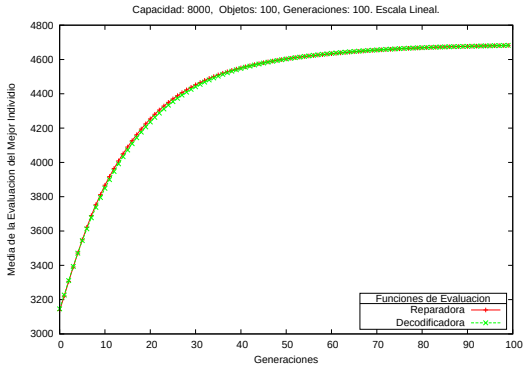
Figura 5.9: Capacidad: 4000, Objetos: 100, Generaciones: 100.

No hay ninguna diferencia destacable entre ambas funciones, ambas se alternan en ser cota superior de la otra, seguramente debido a que hayan resultado especialmente ventajosas las cien simulaciones realizadas. Aún así, ambas funciones crecen de manera idéntica y convergen al mismo punto.

Para comprobar que la convergencia de ambas funciones sigue siendo semejante, simulamos un número mayor de generaciones (figuras 5.11, 5.12 y 5.13):

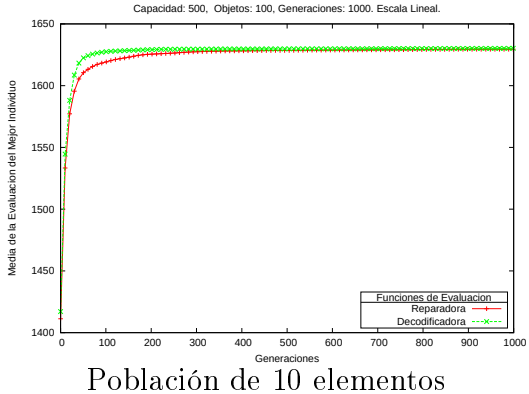


Población de 10 elementos

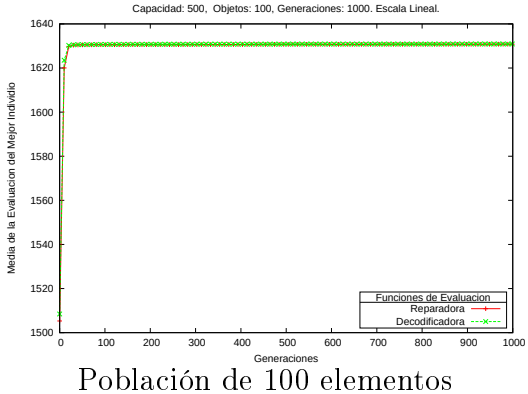


Población de 100 elementos

Figura 5.10: Capacidad: 8000, Objetos: 100, Generaciones: 100.

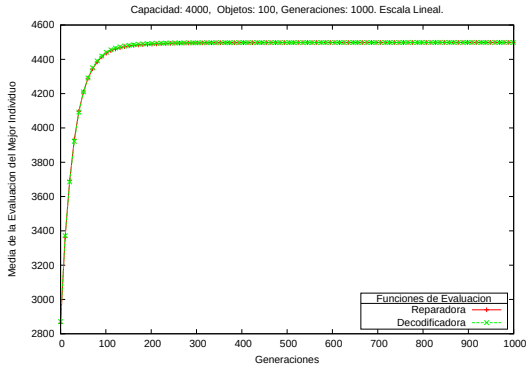


Población de 10 elementos

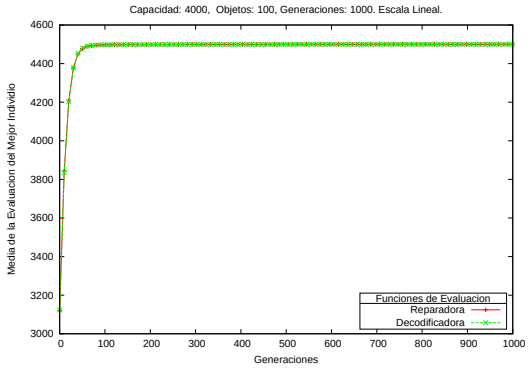


Población de 100 elementos

Figura 5.11: Capacidad: 500, Objetos: 100, Generaciones: 1000.

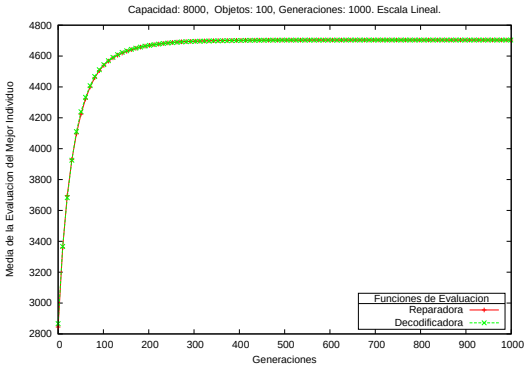


Población de 10 elementos

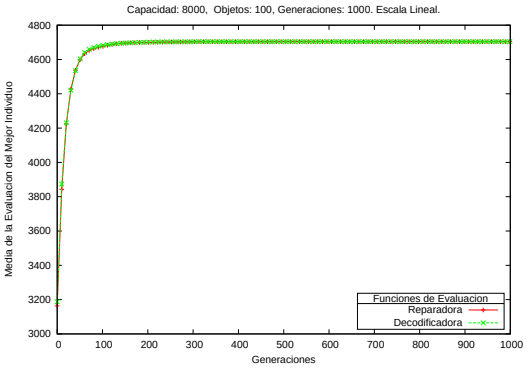


Población de 100 elementos

Figura 5.12: Capacidad: 4000, Objetos: 100, Generaciones: 1000.



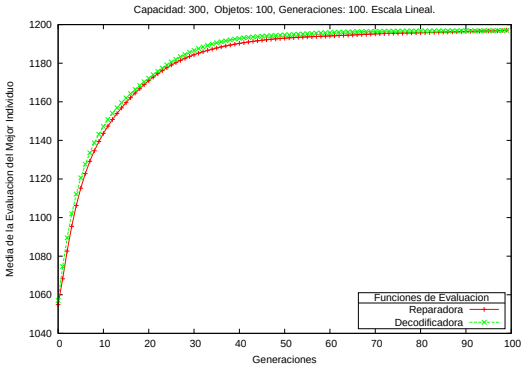
Población de 10 elementos



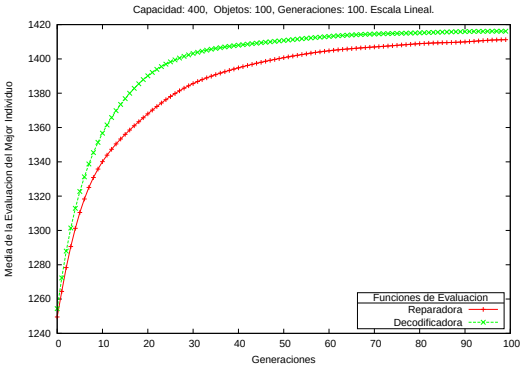
Población de 100 elementos

Figura 5.13: Capacidad: 8000, Objetos: 100, Generaciones: 1000.

A medida que hemos aumentado la capacidad de la mochila, las funciones se hacen prácticamente indistinguibles. Vamos a volver por tanto al caso de una capacidad de 500 unidades para comprobar cómo se comportan las funciones en torno a ella. Nos centraremos en la población de 10 elementos que muestra mayores diferencias y vemos las soluciones obtenidas para capacidades de 300, 400, 600 y 700 unidades de volumen.

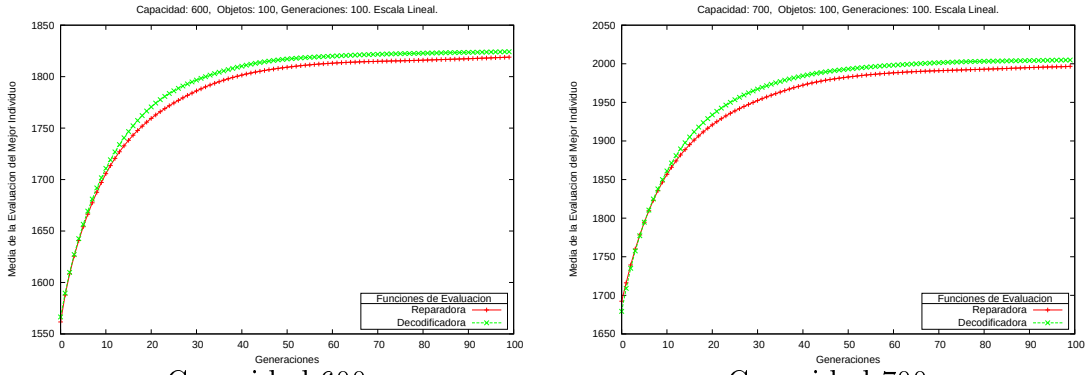


Capacidad 300



Capacidad 400

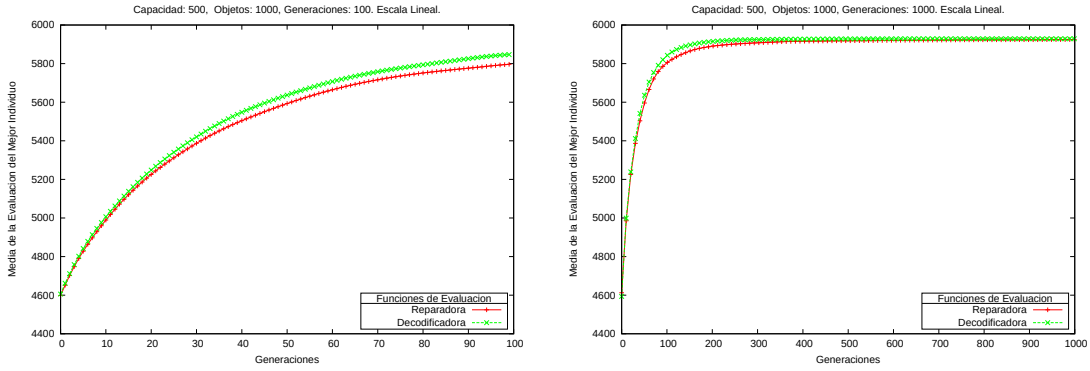
Figura 5.14: Comparativa en torno a una capacidad de 500 unidades. Capacidad 300 y 400.



Capacidad 600 Capacidad 700

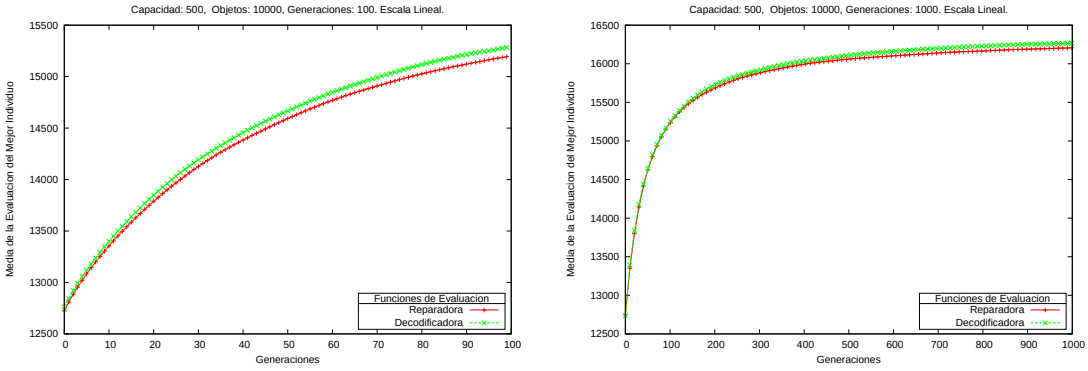
Figura 5.15: Comparativa en torno a una capacidad de 500 unidades. Capacidad 600 y 700.

A la vista de los resultados, no podemos extrapolar ningún comportamiento específico de las funciones en torno a una capacidad de 500 unidades, tan sólo que para pequeños tamaños de la mochila, la función decodificadora parece ser ligeramente superior a la reparadora. Vamos a variar ahora, para esa misma capacidad de 500 unidades el tamaño del problema, aumentando el número de objetos para observar como se comportan ambas funciones.



100 generaciones 1000 generaciones

Figura 5.16: Comparativa para capacidad de 500 unidades con 1000 objetos



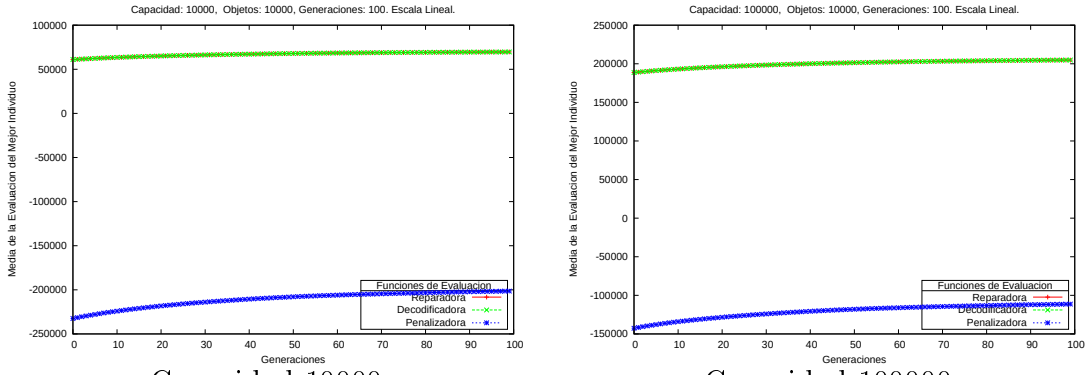
100 generaciones 1000 generaciones

Figura 5.17: Comparativa para capacidad de 500 unidades con 10000 objetos

Podemos determinar con estos resultados que, a esta escala, tanto la función reparadora como la decodificadora son semejantes. Hemos realizado distintas pruebas para el caso detectado en el que más diferencia parecía haber. Aun existiendo una ligera ventaja de la función decodificadora sobre la reparadora, los resultados son muy semejantes para ambas funciones como para determinar que la función decodificadora sea significativamente superior a la reparadora. Por tanto debemos concluir que el desempeño de ambas funciones es prácticamente idéntico para problemas de este tamaño.

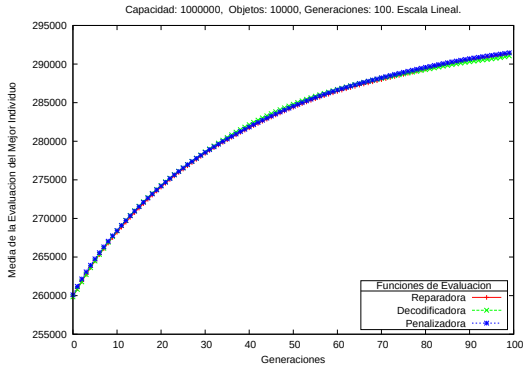
5.2.3. Análisis Para Tamaños Grandes

Al igual que hicimos en el apartado 5.2.1, vamos a comparar las tres funciones pero, en vez de realizar dicha comparación para una colección de 100 objetos, vamos a tomar una colección de 10000 objetos. Para ello, hemos generado las simulaciones correspondientes a mochilas de una capacidad comprendida entre 10000 y 30000000 unidades de volumen. Mostramos a continuación las correspondientes a 10000, 100000, 1000000 y 10000000 unidades de volumen.

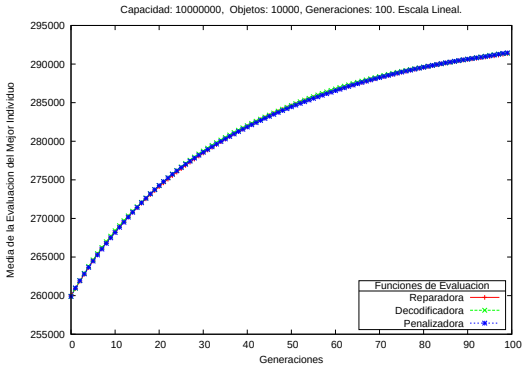


Capacidad 10000

Capacidad 100000



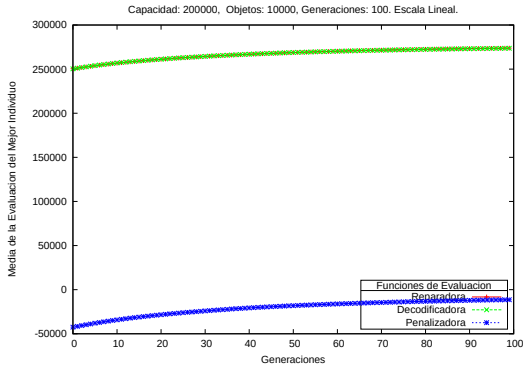
Capacidad 1000000



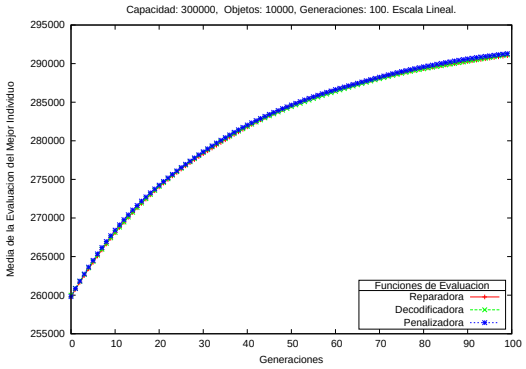
Capacidad 10000000

Figura 5.18: Comparativa para 10000 objetos. Capacidad entre 10000 y 10000000 unidades

Vemos como la función penalizadora pasa a equipararse a las otras dos en un punto intermedio entre la capacidad de 100000 unidades y 1000000 unidades. Vamos a centrarnos en ese rango para localizar el entorno en el que esto se produce.



Capacidad 200000



Capacidad 300000

Figura 5.19: Comparativa para 10000 objetos. Capacidad 200000 y 300000

El punto en que la función penalizadora comienza a generar soluciones iniciales factibles lo encontramos para capacidades de la mochila entre 200000 y 300000 unidades de volumen. Analizando la evolución de la función penalizadora en torno a una capacidad de 250000 unidades, vemos que su comportamiento es similar al caso anteriormente analizado para 100 objetos y capacidad de 500 unidades.

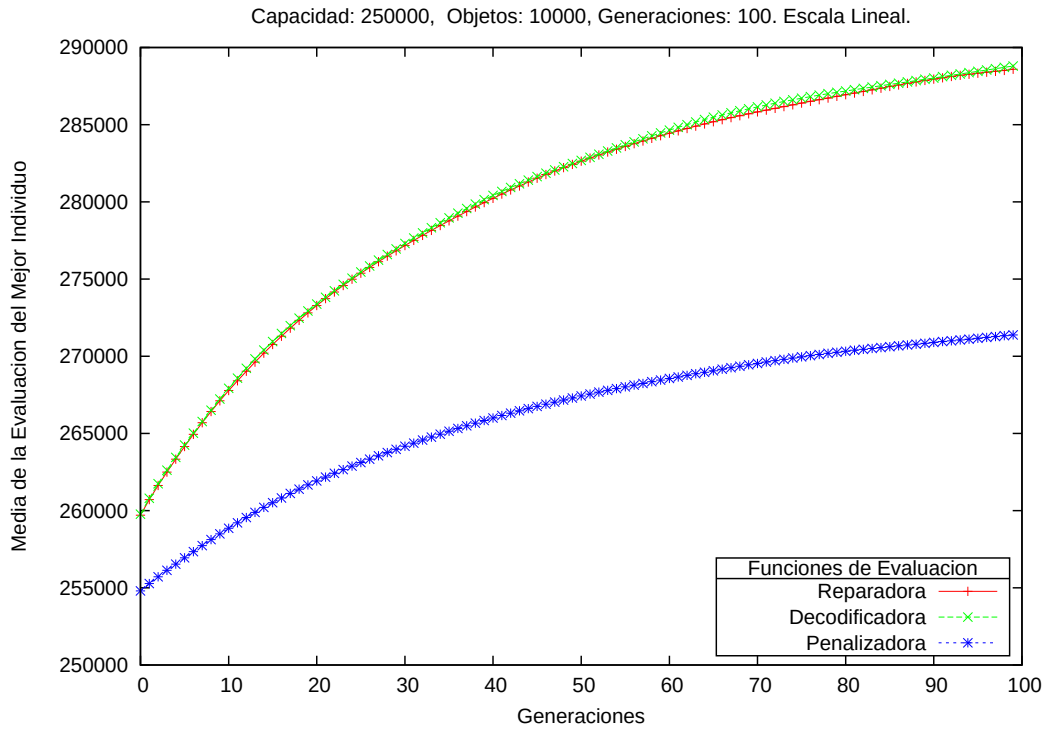
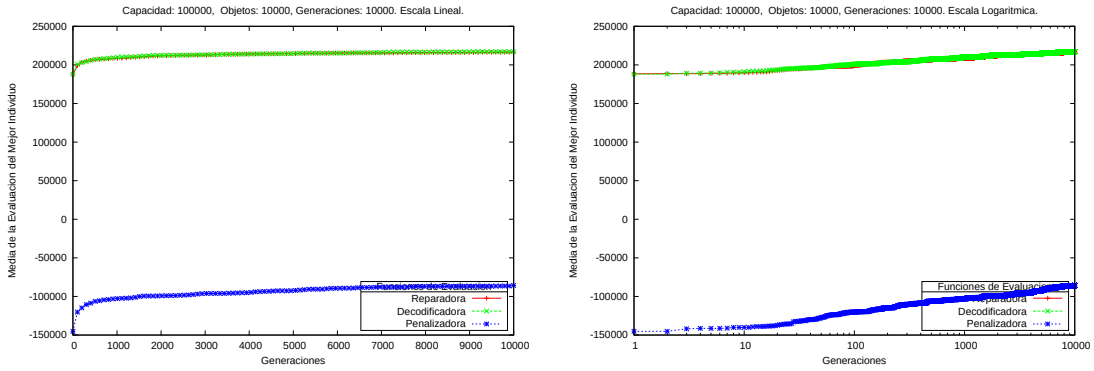


Figura 5.20: Capacidad: 250000, Objetos: 10000, Generaciones: 100.

Podemos determinar, por tanto, que el comportamiento de la función penalizadora es semejante. Se equipara a la función decodificadora y a la reparadora cuando sobrepasa un determinado límite en la capacidad que propicia que la mayor parte de las soluciones generadas sean factibles y, por tanto, tengan evaluación positiva.

Nos hemos centrado hasta ahora en el análisis de las primeras 100 generaciones. Pasamos ahora a comprobar el comportamiento de la función penalizadora a medida que aumenta el número de generaciones para ver si termina convergiendo con las funciones reparadora y decodificadora. Para tener una mejor visión del crecimiento de las funciones, utilizamos para la representación una escala lineal junto con su correspondiente logarítmica.

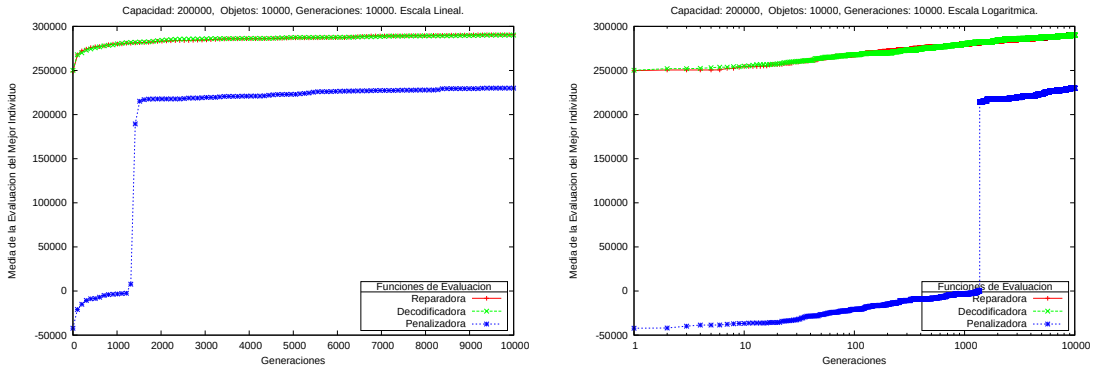
Vamos a analizar tres casos bastantes característicos con unas capacidades para la mochila de 100000, 200000 y 300000 unidades respectivamente.



Escala lineal Escala logarítmica

Figura 5.21: Capacidad: 100000, Objetos: 10000, Generaciones: 10000.

En el primer caso, para 100000 unidades de volumen de capacidad, vemos como la función penalizadora, no llega ni siquiera a generar soluciones factibles en las primeras 10000 generaciones. Aún así, podemos comprobar en la escala logarítmica como la funcion penalizadora tiene una tasa de crecimiento mayor que las otras dos.

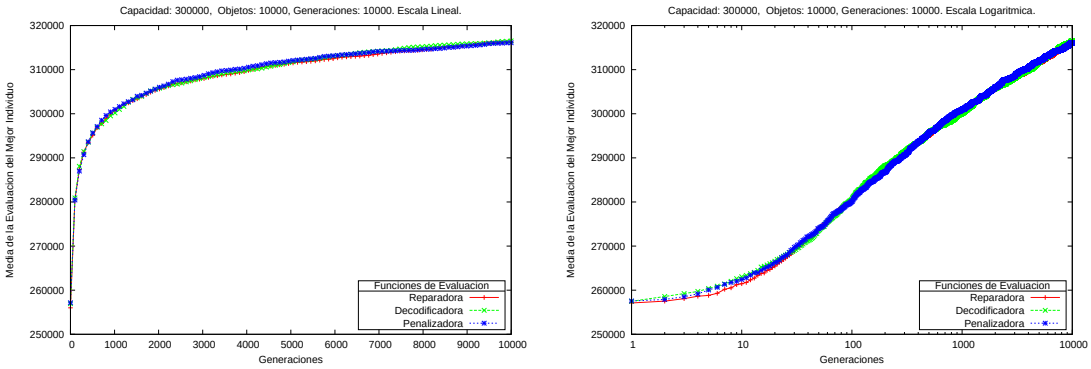


Escala lineal Escala logarítmica

Figura 5.22: Capacidad: 200000, Objetos: 10000, Generaciones: 10000.

Para el caso de 200000 unidades de volumen, podemos comprobar como la función penalizadora comienza a tener elementos factibles entre las generaciones 1000 y 2000. En este punto sufre un salto cualitativo ya que empieza a evaluar sus soluciones por el valor de los objetos incluidos en la mochila en lugar de penalizar la capacidad

sobrepasada. Luego sigue creciendo a un ritmo ligeramente superior a las funciones reparadora y decodificadora. Aún así, no llega a converger al mismo ritmo de estas. Podría ocurrir que la función tenga un crecimiento más lento que las otras dos o que haya caído en un óptimo local.

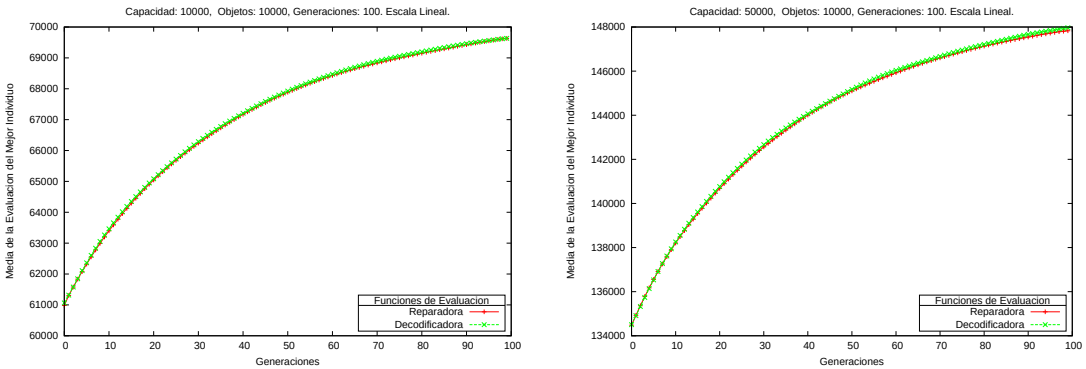


Escala lineal Escala logarítmica

Figura 5.23: Capacidad: 300000, Objetos: 10000, Generaciones: 10000.

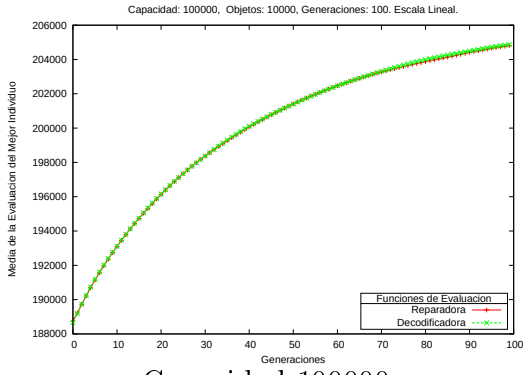
En este caso ya hemos superado la barrera de capacidad para la cual la mayoría de las soluciones iniciales de la función penalizadora son factibles. A partir de este punto, las tres funciones son prácticamente indistinguibles.

Vamos a pasar a comparar, de la misma manera que hicimos anteriormente, las funciones reparadora y decodificadora.

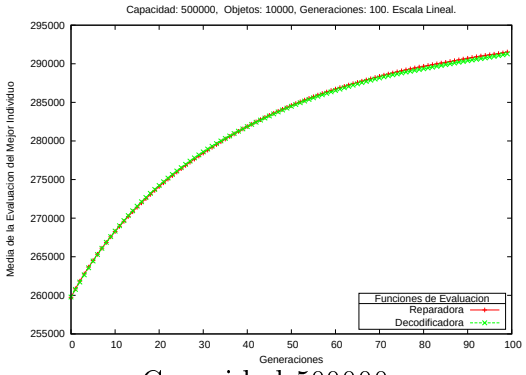


Capacidad 10000 Capacidad 50000

Figura 5.24: Capacidad: 10000 y 50000, Objetos: 10000, Generaciones: 100.

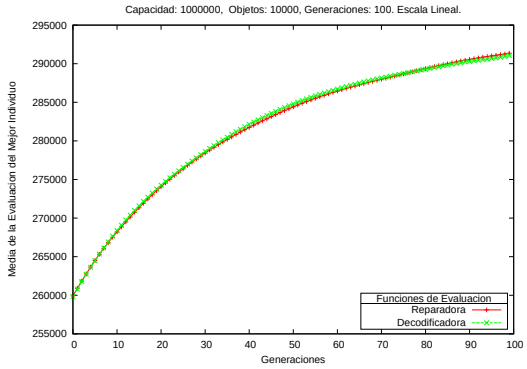


Capacidad 100000

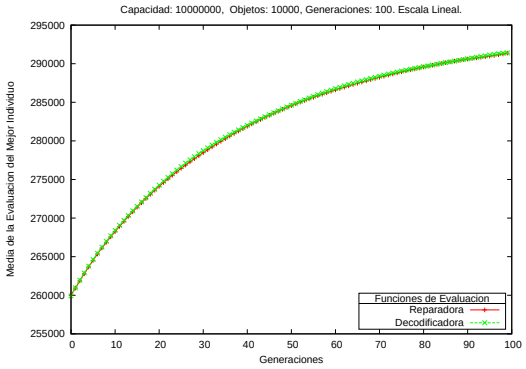


Capacidad 500000

Figura 5.25: Capacidad: 100000 y 500000, Objetos: 10000, Generaciones: 100.



Capacidad 1000000



Capacidad 10000000

Figura 5.26: Capacidad: 1000000 y 10000000, Objetos: 10000, Generaciones: 100.

Al igual que en anteriores simulaciones, las diferencias entre ambas funciones son despreciables. Es más, a esta escala, las diferencias se hacen ínfimas, por lo que podemos determinar que ambas funciones son prácticamente semejantes.

5.3. Resultados

Por medio de las simulaciones y evaluaciones que hemos ido realizando a lo largo de este capítulo, hemos podido comprobar dos puntos fundamentales:

1. La función penalizadora tiene peor rendimiento que la reparadora y decodificadora para tamaños pequeños de la mochila. En problemas de mayor tamaño,

cuando la mochila es asimismo pequeña en proporción al volumen total del conjunto de objetos, la función penalizadora tiene un ritmo de crecimiento muy pobre.

2. Las funciones reparadora y decodificadora son prácticamente semejantes.

5.3.1. Desempeño de la Función Penalizadora

La función de evaluación penalizadora tiene un desempeño mucho peor que la reparadora y la decodificadora para tamaños pequeños de la mochila. Sin embargo, a medida que aumentamos la capacidad de la misma, llega un momento en la que las tres funciones comienzan a asemejarse. El punto en que la función penalizadora empieza a equipararse a las otras dos se corresponde con aquella capacidad que consigue que parte de las soluciones iniciales tengan una evaluación positiva.

Para justificar este comportamiento hemos de recordar que, en el caso de la función penalizadora, la evaluación será negativa siempre que la capacidad de la mochila sea superada por los objetos seleccionados para pertenecer a una determinada solución. La evaluación en estos casos será igual a la diferencia entre la capacidad de la mochila y el volumen total de la solución. Cuanto más negativa sea la evaluación de la solución, significará que más lejos se encuentra de ser factible.

Nuestra implementación del algoritmo genético crea las soluciones iniciales asignando aleatoriamente a cada uno de los elementos de cada solución un 1, si dicho objeto pertenece a la solución o un 0 si el objeto se descarta. La capacidad total ocupada por una solución inicial en nuestra implementación, para una distribución normal de objetos, estará por término medio en torno al 50 % del volumen total de todos los objetos. En el caso en que el tamaño de la mochila esté cerca de este tamaño, tendremos varias soluciones factibles y la función generará rápidamente soluciones factibles. En caso contrario, la función penalizadora tardará un tiempo en generar soluciones factibles.

Para problemas de mayor tamaño, con muchos objetos y capacidad pequeña para la mochila, además del problema de evaluación recién comentado, encontramos que la convergencia a la solución se hace extremadamente lenta, siendo la función penalizadora muy inferior en rendimiento a la reparadora y decodificadora.

5.3.2. Funciones Reparadora y Decodificadora

Las funciones reparadora y decodificadora tienen idéntico comportamiento. En todas las simulaciones realizadas se ha comprobado como las funciones son prácticamente indistinguibles en su representación. Hemos detectado y analizado algún caso donde era apreciable una cierta diferencia. Aunque en estas simulaciones la función decodificadora tienda a ser cota superior de la reparadora, no se observa un comportamiento extrapolable del que se pueda deducir una diferencia de rendimiento apreciable entre ambas.

Las diferencias puntuales en las simulaciones podrían estar causadas por la implementación concreta del algoritmo, por la distribución de objetos utilizada en las simulaciones o por una ligera ventaja de la función decodificadora. Esta ventaja provendría del hecho de que, al no modificar la función decodificadora los genes de las soluciones en su evaluación, se permite que dos soluciones ligeramente distintas tengan la misma evaluación. En el caso expuesto, tendríamos dos soluciones distintas, cuyo volumen total sobrepasa la capacidad de la mochila. En cada una de estas soluciones tendríamos un objeto con un ratio bastante pobre y cuya eliminación haría que la solución fuera factible. La función reparadora, a la hora de evaluar dichas soluciones, eliminaría ambos objetos de cada una de las soluciones, consiguiendo que las dos soluciones sean idénticas y tengan la misma evaluación. La función decodificadora, sin embargo, no modifica los genes de la solución. Devolvería la misma evaluación para las dos soluciones, pero los genes de ambas seguirían siendo distintos. Esto implicaría la existencia de una mayor diversidad de soluciones y de ahí la posible ventaja.

Capítulo 6

Conclusiones

A lo largo de la toda la memoria hemos ido desgranando cada uno de los puntos fundamentales del trabajo realizado durante la elaboración del proyecto de fin de carrera. Partiendo de un análisis del contexto histórico en el que hizo su aparición y cómo se ha ido desarrollando a lo largo de los años la computación evolutiva, hemos visto las distintas ramas de las que se compone y el campo de aplicación de cada una de ellas. Tras analizar la estructura básica de un algoritmo genético hemos introducido el problema de la mochila así como determinados métodos de resolución clásicos. Pasamos posteriormente a realizar un análisis pormenorizado de la implementación de nuestro algoritmo genético, que resuelve el problema de la mochila, acabando con una explicación del interfaz gráfico y las distintas evaluaciones que hemos realizado.

6.1. Interfaz Gráfico

Uno de los objetivos fundamentales planteados al inicio del Proyecto fue la elaboración de un interfaz gráfico para la visualización de la ejecución del algoritmo genético. Este apartado ha sido el que ha requerido un mayor esfuerzo. La idea del interfaz gráfico era conseguir que la representación planteada mostrase, de la manera más fidedigna posible, la resolución del problema de la mochila. Asimismo debía ser lo suficientemente intuitivo como para que su funcionamiento pudiera ser entendido tras una pequeña introducción al problema de la mochila. Ambos apartados han sido

alcanzados cuidando especialmente el apartado gráfico y añadiendo la posibilidad de ver la ejecución del procedimiento de resolución por medio de animaciones.

Otra característica de la implementación gráfica es la posibilidad de modificar los distintos parámetros que guían la ejecución del algoritmo genético. Para ello disponemos de una ventana auxiliar para modificarlos, dando la posibilidad de realizar distintas ejecuciones para un mismo problema para comprobar cómo afecta cada uno de los parámetros a su resolución. Estos problemas pueden ser posteriormente guardados en disco, de forma que se posibilita su reutilización posterior.

6.2. Evaluación

En cuanto a la evaluación de nuestro algoritmo, hemos prestado especial atención a la función de evaluación. Siendo esta función fundamental en cualquiera de las especializaciones de la computación evolutiva, parecía lógico realizar distintas implementaciones y comprobar el desempeño de las mismas. Hemos incluido tres funciones, a las que hemos denominado reparadora, decodificadora y penalizadora. Tras el análisis del desempeño de las tres, hemos determinado el peor rendimiento de la penalizadora, excepto en casos muy concretos, y la similitud entre las funciones decodificadora y reparadora.

6.3. Posibles Mejoras

Durante la elaboración del trabajo, se han detectado determinados puntos que, por motivos temporales, salían del alcance original del proyecto y que destacamos a continuación:

1. *Sustitución de la aproximación generacional por una no generacional.* La primera implementación realizada era no generacional, es decir, el procedimiento para escoger a los elementos de la nueva generación consistía en un torneo entre los padres y sus descendientes. De todos ellos, los más aptos eran los que pasaban a formar parte de la siguiente generación. Esta aproximación parecía tener mejor rendimiento que la generacional. Aún así se sustituyó por

una estrategia generacional por motivos de claridad y simplicidad. No se ha podido comprobar la diferencia de rendimiento entre ambas aproximaciones empíricamente.

2. *Mejora del rendimiento.* Todo el desarrollo ha sido realizado pensando en la simplicidad de su implementación para facilitar el entendimiento de la misma. Por tanto se han desechado todas las modificaciones que, aun pudiendo mejorar el rendimiento, hicieran el código más oscuro e incomprensible. Es posible mejorar algún apartado del código en este aspecto.

3. *Evaluación temporal.* Todas las medidas realizadas se basan en la evaluación del mejor individuo para cada generación. Esta es una aproximación apropiada, clara y sencilla. Aún así, es posible que alguna de las aproximaciones que a este nivel de detalle no parezca ser demasiado ventajosa, sí lo sea a la hora de realizar el correspondiente análisis temporal. Quizás la función decodificadora o la reparadora tenga mucho mejor rendimiento analizándola en una escala temporal con respecto a la otra y esta sea una diferencia fundamental a tener en cuenta. Por tanto, sería interesante realizar un análisis del comportamiento del algoritmo con respecto a parámetros temporales.

4. *Optimización centrada en el problema.* La implementación del algoritmo ha sido lo más genérica posible. No se ha pretendido realizar un algoritmo genético específico y optimizado para el problema de la mochila. Caben muchas posibilidades en este aspecto, incluyendo nuevas aproximaciones como, por ejemplo, una estrategia memética que guíe la convergencia de la función. La aproximación memética añade un proceso de búsqueda local a cada una de las fases de inicialización, recombinación y mutación de un algoritmo genético.

5. *Distintos métodos de representación de objetos.* La representación gráfica de objetos realizada, por su propia concepción, funciona muy bien para una determinada escala respecto al tamaño de los problemas. En el caso de tener una cantidad grande de objetos, no es posible representarlos todos en pantalla asignando tamaños proporcionales en función de su volumen. Para simulaciones grandes, manejamos colecciones de objetos que ni siquiera asignando un ancho de un punto por objeto cabrían en la pantalla. En el caso en que quisiera realizarse la representación gráfica de estos casos, habría que pensar en una

nueva representación, quizás mostrando una serie de datos estadísticos sobre la distribución de la colección de objetos y la organización de los mismos en la mochila.

6. *Exportación del trabajo a la web.* Podría elaborarse un applet java partiendo del trabajo desarrollado de forma que fuera un recurso fácilmente utilizable desde la web.

Bibliografía

[AL07] Andreas Antoniou and Wu-Sheng Lu. *Practical Optimization: Algorithms and Engineering Applications.* Springer, 2007.

[BBM00] Thomas Back, David B.Fogel, and Zbigniew Michalewicz. *Evolutionary Computation 1. Basic Algorithms and Operators.* Institute of Physics Publishing, 2000.

[BIKK07a] Moshe Babaio, Nicole Immorlica, David Kempe, and Robert Kleinberg. A knapsack secretary problem with applications. *10-th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems 2007*, 2007.

[BIKK07b] Moshe Babaioff, Nicole Immorlica, David Kempe, and Robert Kleinberg. Approximation, randomization, and combinatorial optimization. algorithms and techniques. *10th International Workshop, APPROX 2007, and 11th International Workshop, RANDOM 2007, Princeton, NJ, USA, August 20-22, 2007*, pages 16–28, 2007.

[BS02] Kurt M. Bretthauera and Bala Shetty. The nonlinear knapsack problem âĂŞ algorithms and applications. *Elsevier Science B.V.*, 2002.

[Deb00] Kalyanmoy Deb. *Evolutionary Computation 1. Basic Algorithms and Operators*, chapter Introduction to selection. Institute of Physics Publishing, 2000.

[ES07] A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computing.* Springer, 2007.

[Esh00] Larry J. Eshelman. *Evolutionary Computation 1. Basic Algorithms and Operators*, chapter Genetic Algorithms. Institute of Physics Publishing, 2000.

[GGT04] G. Giorgi, Angelo Guerraggio, and J. Thierfelder. *Mathematics of Optimization: Smooth and Nonsmooth Case*. Elsevier Science & Technology Books, 2004.

[Gol89] David Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[HH98] Randy Haupt and Sue Ellen Haupt. *Practical Genetic Algorithms*. John Wiley & Sons, 1998.

[Ist02] Aho Isto. *Interactive Knapsacks: Theory and Applications*. Tampereen Yliopisto, 2002.

[Jon06] K.A.De Jong. *Evolutionary computation: a unified approach*. MIT Press, Cambridge MA, 2006.

[KS07] Stavros G. Kolliopoulos and George Steiner. *Discrete Applied Mathematics*, chapter Partially ordered knapsack and applications to scheduling. Elsevier Science Publishers B. V., 2007.

[Mar04] Adam Marczyk. Genetic algorithms and evolutionary computation. *http://www.talkorigins.org/faqs/genalg/genalg.html*, 2004.

[Mit96] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1996.

[Spe00] William M. Spears. *Evolutionary Algorithms. The Role of Mutation and Recombination*. Springer, 2000.

[TH93] Th.Back and H.P.Schwefel. An overview of evolutionary algorithms for parameter optimization. *Evolutionary Computation*, pages 1–23, 1993.

[WPRF98] W.Banzhaf, P.Nordin, R.E.Keller, and F.D.Francone. *Genetic Programming - An Introduction*. Morgan Kaufmann, 1998.

Índice alfabético

adaptación, 31, 34
agente, 35
alelo, 31
algoritmo evolutivo, 22, 23, 25, 28, 33–35
algoritmo genético, 22, 26, 27, 37, 43, 44, 46, 57–61, 65, 66, 73, 89, 91–93
algoritmo voraz, 37, 41–43
algoritmos genéticos, 33
análisis temporal, 93
aplicación, 59
applet, 94
aprendizaje automático, 35
argumento, 59
asignación de personal, 37
autómata celular, 35
ayuda, 64

beneficio, 43
búsqueda estocástica, 25
búsqueda exhaustiva, 37, 41
búsqueda local, 93

capacidad, 40, 43, 44, 47, 48, 51, 52, 59, 60, 67, 69, 74, 75, 81, 82, 86, 89, 90
capacidad de la mochila, 67, 73–75, 78
capacidad máxima, 39, 40, 47
capacidad total, 47, 69
características, 66
colección de objetos, 40, 44, 63
comparativa lineal, 74
comparativa logarítmica, 74
compilador, 59
complejidad, 41
complejidad lineal, 42
comportamiento evolutivo, 35
computación evolutiva, 25, 27, 28, 33, 91, 92
condición de terminación, 33
configuración, 60, 66
contenedor, 37
convergencia, 41, 46, 89, 93
cota superior, 79, 90
criterio de parada, 43
cruce, 55
código, 58

decodificadora, 47, 50, 51, 60, 61, 67
descendiente, 31, 54, 92
descriptor del objeto, 62
devoradora, 47
distribución, 70, 78, 94
distribución aleatoria, 46
distribución de objetos, 90
distribución normal, 34, 89
distribución uniforme, 70
diversidad, 26, 53, 55
diversidad de la población, 33, 53

eficiencia, 42
elemento más apto, 67
elitismo, 56, 59–61, 65, 67
escala logarítmica, 86
espacio de búsqueda, 27, 33, 45
espacio de soluciones, 22
espacio libre, 69
espacio ocupado, 69
estado estacionario, 56
estocástico, 26, 33
estrategia evolutiva, 26, 27, 34
estrategia generacional, 56
estrategia memética, 93
estrategia voraz, 47

Evaluación, 65
evaluación, 32, 46, 51, 58, 59, 63, 85, 88–90, 92, 93
evaluación de la población, 58, 65
evaluación global, 43
evaluación inicial, 78
evaluación temporal, 93
evolución, 34, 59
exploración dirigida, 22
explosión combinatoria, 22

factible, 47, 50, 51, 85, 87, 89, 90
fichero, 58, 64
funcion penalizadora, 86
función de evaluación, 22, 25, 29, 30, 32, 38, 43, 46, 48, 59, 61, 67, 73, 74, 92
función decodificadora, 50, 61, 74, 75, 78, 79, 82, 83, 85, 87–90, 92, 93
función lineal, 38
función objetivo, 38
función penalizadora, 51, 52, 61, 74–76, 78, 84–89, 92
función probabilistica, 30
función reparadora, 50, 51, 61, 74, 75, 78, 82, 83, 85, 87–90, 92, 93

gen, 90
generacional, 32, 55, 61, 92
generación, 32, 43, 55, 61, 65, 67
genotipo, 35, 58
grado de aptitud, 43
gráfico de distribución de objetos, 69
gráfico de evaluación, 68
gráfico de objetos, 69

heurística, 41, 43, 47, 70

individuo, 29, 51, 53, 58
inicialización, 93
inicialización de la población, 28, 46
inteligencia artificial, 25
interfaz, 62
interfaz gráfico, 59

jar, 59
java, 57, 59, 94

lista de objetos, 58
logística, 37

media de la evaluación, 73
mejor elemento, 59, 65, 68
mejor evaluación, 68, 69
mejor individuo, 58, 73, 93
mejor solución, 33, 65
memética, 93
mochila, 37, 39, 45, 48, 49, 51, 52, 57–60, 64, 67–69, 75, 85, 86, 88
modelo de optimización, 38
modo de visualización, 70
mutación, 25, 26, 28, 31, 34, 55, 58, 93
máquina de estado finito, 26, 35
máximo, 39
máximo local, 55

no factible, 47, 48, 50, 51
no generacional, 92
número de generaciones, 65
número de objetos, 40, 41, 73

objeto, 43, 45, 47, 50, 57, 58, 60, 64, 66, 68
objetos, 66
operador de mutación, 31, 55
operador de selección, 30, 31
operador probabilístico, 30
operadores, 57
operadores de cruce, 54
optimización, 21
optimización combiatoria, 38
optimización heurística, 34
optimización matemática, 37, 38

padre, 31, 32, 53, 54, 67, 92
pantalla, 58, 63
parámetro, 34, 59, 60, 63, 92
parámetros temporales, 93
penalización por edad, 56
penalizadora, 47, 60, 61, 67
peor individuo, 58

planificación, 33, 34, 37
población, 26, 43, 46, 53, 59, 63, 65, 67–69, 75, 79, 81
población inicial, 46
preferencias, 59, 64, 66
probabilidad de selección, 30
problema de la mochila, 22, 23, 37, 39–42, 57, 69, 91
problema de maximización, 38
problema de optimización, 29, 37
problema de reparto, 37
problema de selección, 39
proceso iterativo, 25
programación automatizada, 35
programación dinámica, 37, 41
programación evolutiva, 26, 27, 34
programación genética, 35
programación lineal, 38
programación lineal entera, 39
programación lineal entera binaria, 38, 39

ratio, 48, 59, 64, 68–70, 90
ratio de mutación, 59–61, 65, 67
recombinación, 25, 26, 28, 31, 34, 51, 53, 55, 93
reconocimiento de patrones, 35
recurrencia, 41
regresión simbólica, 35
rendimiento, 90, 93
reparadora, 47, 60, 61, 67
reparto, 39
representación, 59, 93
representación de individuos, 44
reproducción, 25, 28
restricción, 38–40
robótica, 35

selección, 26, 28, 30, 34
selección de padres, 53
selección de supervivientes, 32, 55
selección natural, 25, 28
selección por torneo, 53, 54
simulación, 58, 63, 73, 74, 88, 90
sistema adaptativo, 27
sistema generacional, 32
solapamiento, 32
soluciones iniciales, 29, 58, 89
soluciones iniciales factibles, 85
solución, 46–50, 52, 69, 89, 90
solución aleatoria, 46
solución candidata, 25, 34
solución clásica, 22
solución entera, 37, 39
solución exacta, 41
solución factible, 29, 50, 75, 78, 86, 89
solución inicial, 43, 89
solución inicial factible, 75
solución no factible, 29
solución parcial, 41
solución subóptima, 41
solución válida, 47, 51
solución óptima, 41, 42
steady state, 56
subproblema, 41
superviviente, 32, 55
síntesis de funciones, 35

tabla interna, 41
tamaño de la mochila, 82, 89
tamaño de la población, 32, 53, 56, 59, 61, 67, 74, 75
tamaño del torneo, 53, 59–61, 67
tasa de mutación, 55, 67
torneo, 58, 92

valor, 37, 39–41, 43, 44, 47, 58, 62, 66, 70
valor / volumen, 47
valor máximo, 59
valor óptimo, 38
valor/volumen, 59, 64, 68–70
variable de decisión, 38, 39
volumen, 37, 39, 40, 43, 44, 47–51, 58, 59, 62, 66, 68, 70, 81, 89, 90, 93
volumen máximo, 40
volumen total del conjunto, 41
voraz, 42

árboles, 35

óptimo, 43, 46
óptimo local, 30, 87