Trabajo de Fin de Grado

Biblioteca Java para algoritmos heurísticos en optimización matemática

Java library for heuristics in mathematic optimization .

Sabato Ceruso

La Laguna, 6 de junio de 2016

D. **Jesús M. Jorge Santiso**, con N.I.F. 42.097.398-S profesor Titular de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor

C E R T I F I C A

Que la presente memoria titulada:

"*Biblioteca Java para algoritmos heurísticos en optimización matemática.*"

ha sido realizada bajo su dirección por D. **Sabato Ceruso.**, con N.I.E. X-8912496-L

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 6 de junio de 2016

Agradecimientos

A mis padres

A mi tutor, Jesús M. Jorge Santiso.

Licencia

Resumen

El objetivo de este trabajo ha sido implementar una biblioteca de clases en Java con las principales técnicas metaheurísticas para la resolución de problemas de optimización matemática.

Se ha puesto especial énfasis en el diseño de la biblioteca y la generalización de las clases que implementan las distintas heurísticas consideradas.

De esta forma se ha conseguido que los diversos algoritmos soportados por la biblioteca sean independientes de los problemas a resolver y que, en la medida de lo posible, los algoritmos y técnicas particulares puedan ser obtenidos a través de la configuración de distintos parámetros de las clases de la biblioteca (tipo de búsqueda y estrategia, criterio de aceptación, ...).

Palabras clave: Metaheurísticas, Optimización, Librería Java, *Framework*

Abstract

The objective of this work is the implementation of a Java library with a collection of metaheuristic techniques for solving mathematical optimization problems .

Special emphasis was placed on the design of the library and the generalization of the classes that implement the various heuristics.

In this way, we managed to implement various algorithms that are independant to the problem to solve. This implementation has the characteristic that, configuring diverse parameters, we can obtain various algorithms depending on acceptance criteria, strategy of the search, etc.

Keywords: *Metaheuristics, Optimization, Java Library, Framework*

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1. Introducción.

En el contexto científico, optimizar es el proceso por el cual se encuentra la mejor solución a un determinado problema. En este tipo de problemas el objetivo es, como se ha dicho antes, encontrar la mejor solución, utilizando una función objetivo y que a su vez, esta solución cumpla una serie de restricciones; de modo que, se podrá decidir si una solución es mejor que otra si tiene mejor valor objetivo y no infringe ninguna restricción.

Existen una gran cantidad problemas de optimización: de cálculo de caminos mínimos, flujo en redes, asignación, etc. De manera formal, un problema de optimización se puede describir de la siguiente manera:

$$\begin{aligned} &\text{Minimizar } f(x) \\ &\quad \text{Sujeto a} \\ &\qquad \text{Condicion 1} \\ &\qquad \text{Condicion 2} \\ &\qquad \vdots \end{aligned} \tag{1.1}$$

Estos problemas se pueden resolver de dos modos distintos: de forma aproximada o de forma exacta. La resolución exacta de estos problemas, exceptuando algunos casos que se conoce la forma de calcular la solución óptima en tiempo polinómico, suele ser muy costosa en términos computacionales, alcanzando muchas veces un tiempo exponencial en su resolución.

Los métodos aproximados, también llamados *heurísticos*, ganan importancia gracias al coste que se tiene al utilizar métodos exactos. Estos métodos consiguen encontrar una solución óptima considerada de "buena calidad" en un tiempo mucho menor que el que tardaría un método exacto, normalmente en tiempo polinómico. Se pretende que esta biblioteca implemente y genere esta clase de algoritmos.

Existe una clase de problemas que son costosos de resolver, los de optimización combinatoria [11]. La principal característica de estos problemas es que el espacio de soluciones donde se ha de buscar es discreto o infinito numerable (región factible del problema); y, lo que los vuelve particularmente difíciles es la gran cantidad de soluciones que contiene este espacio.

Un modo de resolver este tipo de problemas de forma exacta (aunque claramente ineficiente debido al gran número de soluciones factibles) puede ser muestrear toda la región factible (si se trata de un conjunto finito) y obtener así la solución con mejor valor objetivo. Sin embargo, está claro que este método es inviable para la mayor parte de los problemas.

1.2. Importancia de las heurísticas.

Como se ha comentado anteriormente, existen problemas de optimización que, dadas sus características, son muy difíciles de resolver siguiendo métodos exactos, y es aquí donde toman importancia las heurísticas.

En contraposición con los métodos exactos, los métodos heurísticos no garantizan encontrar la solución óptima del problema, sin embargo, consiguen resolverlo y encontrar soluciones "de buena calidad". Está claro que, al no proporcionar una garantía de que la solución es óptima (que en la mayoría de los casos no lo es), el método heurístico deberá conseguir encontrar esa solución en un tiempo mucho menor al que tardaría el método exacto.

Los métodos heurísticos no solo se utilzan gracias a su velocidad, si no que también ganan importancia a la hora de resolver problemas para los cuales no se conoce un método exacto (o el único método posible es la fuerza bruta).

Por último, otra razón para utilizar estos métodos es su fácil capacidad de adaptación a los problemas (sobre esta característica se basa esta biblioteca), es decir, un mismo método, misma idea, puede utilizarse en una variedad distinta de problemas, lo que permite que sea muy fácil realizar un método para un determinado problema, y luego modificarlo para que contemple distintos matices que se quieran agregar al mismo (como pueden ser más restricciones o distintas funciones objetivo).

1.3. Importancia de la biblioteca.

Dadas las características de los métodos heurísticos, existen una gran variedad de ellos, muchos son ideados a partir de la observación de la naturaleza (algoritmos genéticos),a partir de métodos muy sencillos que pueden ser generalizados (métodos greedy), como combinación de varios métodos, etc.. También, muchos métodos heurísticos tienen la característica que dependen de muchos parámetros, y una mala elección de los mismos podría hacer que no funcionen correctamente (como puede ser la decisión del tamaño de la lista restringida de candidatos en un GRASP). También se debe tener en cuenta que la validez y utilidad de un método dependerá del problema al que se aplique, por lo que no existe un método que sea en términos absolutos mejor que otro (teorema *No Free Lunch*).

Dadas estas características, resulta interesante poder disponer de un *software* capaz de generar distintos métodos heurísticos para cualquier problema y realizar combinaciones entre ellos; de modo que se pueda encontrar qué método consigue mejores resultados para un determinado problema, y con que parámetros se debería ejecutar. Alguna de las propiedades deseables en este tipo de *software* son:

- Que incluyan los principales algoritmos del estado del arte de las metaheurísticas.
- Que sea puedan modificar, crear e implementar nuevos algoritmos e ideas de manera fácil y rápida.
- Que contengan *sets* de problemas de estudio, como puede ser el TSPLIB.

1.4. Antecedentes.

Durante los últimos años, han surgido varios *frameworks* distintos que permiten implementar algoritmos heurísticos, como es el caso de jMetal [5] o MOEAT Frameowrk [10]. Estas herramientas están centradas en la resolución de problemas multi-objetivo, y contienen la implementación de varios de los algoritmos del estado del arte para afrontar este tipo de problemas (SPEA2, NSGA-II...) [3] [7] y varios *sets* de problemas clásicos como ZDT [2] o DTLZ [6]. Para la evaluación de la calidad de los frente de Pareto obtenidos, estos *frameworks* proporcionan varios de los indicadores de calidad más utilizados, como el indicador de hipervolumen, el indicador epsylon o el Spread.

Nuestro *software* toma de ejemplo estas herramientas ya existentes para el diseño e implementación. Si bien las similitudes son evidentes (dado que se trata de herramientas pensadas para el mismo contexto), existen diferencias sustanciales entre nuestra biblioteca y dichas herramientas.

En primer lugar, nuestra biblioteca se centra en algoritmos mono-objetivos, mientras que los *frameworks* citados anteriormente están centrados en algoritmos y problemas multi-objetivo. Si bien, un algoritmo multi-objetivo puede resolver problemas mono-objetivo planteándolos como problemas multi-objetivo con una sola función objetivo (por definición, un problema mono-objetivo); las características de problemas con varias funciones objetivos frente a problemas con una única función son muy diferentes, empezando por que en el primer caso se busca un frente eficiente de soluciones, mientras que en el segundo se persigue encontrar el óptimo global. Por esta razón, aunque se pueda utilizar un algoritmo multi-objetivo en un problema mono-objetivo, no sería de extrañar que la calidad de la solución obtenida y eficiencia del método sea inferior a la de un algoritmo pensado para problemas de un solo objetivo.

Otra diferencia importante es el tipo de algoritmos que permite implementar cada *software*. Los *frameworks* mencionados anteriomente están basados en métodos poblacionales, mientras que nuestra biblioteca implementará tanto métodos poblacionales (algoritmos genéticos, búsquedas dispersas, meméticos, etc..), como métodos constructivos (*greedy*, GRASP), basados en multi-arranque o métodos de mejora (búsquedas locales, búsquedas tabú, etc..), además de la posibilidad de combinar cualquiera de estos métodos entre sí, facilitando la creación de algoritmos híbridos.

Capítulo 2

Problemas de estudio.

Como se ha dicho, existen una gran variedad de problemas distintos de optimización combinatoria, de los cuales, se ha hecho una selección de 3 problemas, para poder probar los distintos algoritmos que implementa esta biblioteca. Los 3 problemas seleccionados fueron: problema de la mochila, problema de asignación y problema del viajante de comercio (*Knapsack Problem*, *Assingment Problem*, *Travelling Salesman Problem*).

Estos 3 problemas han sido profusamente estudiados, existiendo una gran cantidad de métodos, tanto exactos como heurísticos, que los resuelven. Además, estos 3 problemas son muy fáciles de comprender y de representar, con lo que resultan ser ideales para realizar las pruebas. En este capítulo los describiremos brevemente.

2.1. Problema de la mochila.

En este problema, se tiene una mochila con una determinada capacidad y una serie de objetos, cada uno con un peso y un valor determinado. El problema consiste en seleccionar qué objetos se insertará en la mochila, de modo que el peso total de los objetos a insertar no sobrepase la capacidad de la misma, y se maximice el valor total de los elementos seleccionados.

Su definición formal es:

$$\begin{aligned} &\text{Maximizar } \sum_{i=1}^{n} x_i * v_i \\ &\quad \text{Sujeto a:} \\ &\qquad \sum_{i=1}^{n} x_i * w_i \leq W \\ &\qquad x_i \in \{0,1\} \quad \forall i \in \{1...n\} \end{aligned} \tag{2.1}$$

Cabe destacar que en este modelo del problema, las variables son binarias; si fuera el problema de la mochila fraccionario, donde la variable "x" puede tomar valores fraccionarios entre [0, 1], entonces el problema sería muy fácil de resolver en tiempo polinomial.

Algunas aplicaciones para este problema pueden ser:

- Determinar artículos a almacenar para maximizar el valor.
- El problema del *Cutting Stock* [1].

- Maximizar el beneficio de inversiones cuando solo hay una restricción.

2.2. Problema de asignación.

En el problema de asignación, se tiene una serie de tareas a realizar por una serie de personas. Para cada persona y tarea, se tiene un coste asociado que indica cuanto cuesta asociar esa persona a esa tarea. El problema consiste en decidir, teniendo en cuenta que se tiene igual número de personas que de tareas, qué persona realiza qué tarea, de tal modo que se minimice el coste total de realizar todas las tareas.

Su definición formal es:

$$\text{Minimizar } \sum_{i=1}^{n}\sum_{j=1}^{n} x_{i,j} * c_{i,j}$$

Sujeto a:

$$\sum_{i=1}^{n} x_{i,j} = 1 \forall j \in \{1...n\} \tag{2.2}$$

$$\sum_{j=1}^{n} x_{i,j} = 1 \forall i \in \{1...n\}$$

$$x_{i,j} \in \{0,1\} \quad \forall i,j \in \{1...n\}$$

En este modelo una tarea puede ser realizada por solo una persona, y cada persona solo puede estar asignada a una tarea. Este es el modelo más básico del problema de asignación y existe un algoritmo exacto capaz de resolverlo rápidamente (Método Húngaro [8]). Sin embargo, existen muchas variantes de este problema, que, por ejemplo, cambiando simplemente la función objetivo ya deja de funcionar el algoritmo exacto.

Algunas aplicaciones para este problema pueden ser:

- Asignar tareas a personas.
- Asignación de trabajos a procesadores.

2.3. Viajante de comercio.

El problema también es conocido como *Travelling Salesman Problem* (TSP), y ha sido uno de los problemas más estudiados en Investigación Operativa.

El problma se formula de la siguiente manera: "Un viajante de comercio ha de visitar una serie de ciudades comenzando y finalizando en su ciudad. Conociendo el coste de ir de una ciudad a otra, se debe determinar el recorrido de coste mínimo."

Esto es lo que se conoce en teoría de grafos como un ciclo hamiltoniano, y, lo que se debe encontrar en este problema, es el ciclo hamiltoniano de coste mínimo.

El problema puede ser formulado mediante un modelo de programación lineal entera con variables binarias. Para ello, basta considerar que las las variables $x_{i,j}$ valen 1 si el viajante va de la ciudad i a la j y 0 en otro caso, mientras que $c_{i,j}$ es el coste de ir de la ciudad i a la j. Su

definición formal es:

$$\begin{aligned} &\text{Minimizar } \sum_{i<j} x_{i,j} * c_{i,j} \\ &\text{Sujeto a:} \\ &\sum_{i<j} x_{i,j} + \sum_{j<i} x_{j,i} = 2 \quad \forall i \in \{1...n\} \\ &\sum_{(i,j)\in\delta(S)} x_{i,j} \geq 2 \quad \forall S \subseteq \{1...n\} \quad 3 \leq |S| \leq [n/2] \\ &x_{i,j} \in \{0,1\} \quad \forall i < j \end{aligned} \tag{2.3}$$

Donde $\delta(S)$ representa el conjunto de aristas incidentes con exactamente un vértice de S.

Las restricciones que aparecen en segundo lugar reciben el nombre de *restricciones de eliminación de subtour*. La principal dificultad es el número exponencial de restricciones, y esta es la razón por la que este problema es particularmente difícil de resolver.

Algunas aplicaciones para este problema pueden ser:

- Fabricación de circuitos integrados
- Rutas de vehículos.
- Instalación de componentes de ordenadores.

Capítulo 3

Algoritmos.

En este capitulo se detallarán las distintas heurísticas seleccionadas que implementa la biblioteca. Empezaremos hablando sobre las diferentes formas de codificar las soluciones y después describiremos las principales metaheurísticas.

3.1. Codificación de las soluciones.

Cada solución debe ser representada. La forma en que se represente determinará en gran medida el desempeño del algoritmo, por lo que la elección de la codificación resulta ser un paso crucial a la hora de diseñar un algoritmo.

Hay que tener en cuenta varios factores a la hora de seleccionar como codificar las soluciones. Se deberá tener en cuenta el resto de operaciones del algoritmo así como el modo en que se evalúan las soluciones; de modo que se pueda escoger una codificación que facilite dichas operaciones. Un ejemplo de una codificación inteligente puede ser una que, independientemente de los valores que tomen los elementos de la solución, siempre se tenga una solución factible; esto, en muchos casos agiliza en gran medida los algoritmos.

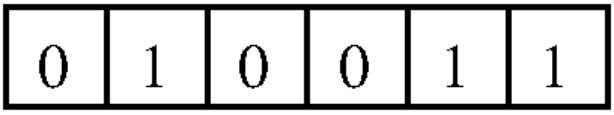

Figura 3.1: Codificación de una solución.

Una forma de codificar una solución, puede ser con una ristra de números binarios. Por ejemplo, en la figura 3.1 se muestra como se podría codificar una solución para el problema de la mochila, con tantos bits como elementos tenga el problema, donde una solución con el i-ésimo elemento a 1 significará que el elemento i forma parte de la solución.

Otra forma de codificar soluciones puede ser, por ejemplo, con permutaciones de enteros. Con esa codificación se podría representar fácilmente una solución para el problema del TSP, dónde la lista de números indica el orden en que se visita cada ciudad.

Esas son dos de las codificaciones más básicas. Existen muchas otras desde ristras de números reales hasta representaciones con árboles.

3.2. Métodos poblacionales.

Los métodos poblacionales son algoritmos que en lugar de trabajar sobre una única solución, trabajan sobre un conjunto de soluciones (población) que van modificando iterativamente, haciendo interactuar las soluciones de la misma entre ellas, para así conseguir explorar el espacio de soluciones en busca del óptimo global.

En las siguientes secciones se detallarán los algoritmos poblacionales implementados en esta biblioteca.

3.2.1. Algoritmos genéticos.

Los Algoritmos Genéticos (*GA-Genetic Algorithm*)[4, p. 55] son métodos adaptativos que pueden usarse para resolver problemas de búsqueda optimización. Están basados en el proceso genético de los organismos vivos, inspirándose en la teoría de la evolución de Darwin. A lo largo de las generaciones, las poblaciones evolucionan en la naturaleza de acorde con los principios de la selección natural y la supervivencia de los más fuertes. Por imitación de este proceso, los Algoritmos Genéticos son capaces de ir creando soluciones para problemas del mundo real.

En el mundo real, los individuos compiten entre sí para conseguir alimentos, refugio, recursos y por reproducirse. Los individuos que consiguen adaptarse mejor al medio conseguirán reproducirse con mayor probabilidad que aquellos individuos con más dificultades para adaptarse; de modo que, generación tras generación, se irán propagando los genes de aquellos individuos mejores adaptados. De esta forma, se irán produciendo combinaciones de buenas características que podrán dar lugar a individuos cada vez mejores adaptados al medio.

Los algoritmos genéticos usan una analogía directa con el comportamiento natural. Trabajan sobre una población de individuos, donde cada uno de ellos será una solución. A cada individuo se le asigna un valor de adaptación, es decir, qué tan buena es la solución para el problema; de esta forma se puede simular el nivel de adaptación al medio del individuo. Y, al igual que en la naturaleza, los algoritmos genéticos van, iteración tras iteración, seleccionando los individuos a cruzar su información (basándose en su nivel de adaptación al medio), cruzándolos para generar una nueva generación, y, finalmente, decidiendo que individuos formarán parte de la nueva población en la siguiente iteración.

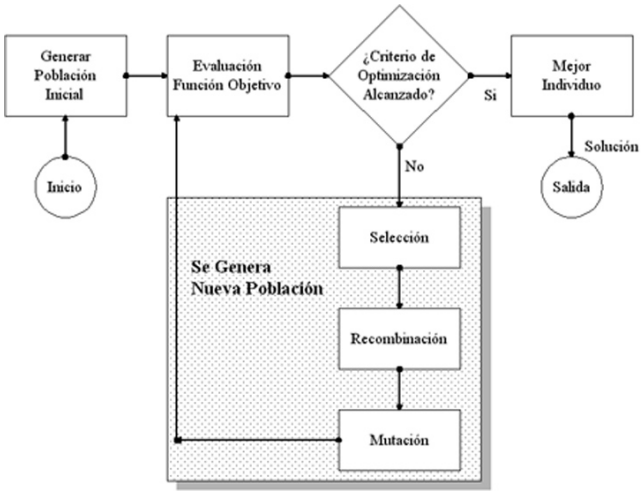

Figura 3.2: Diagrama de un algoritmo genético.

El procedimiento que realiza para llevar a cabo la búsqueda se representa en la figura 3.2. Como se puede apreciar, hace falta determinar el significado de las operaciones de "selección", "recombinación" y "mutación"; así como la forma en que se evalúa y representa cada solución para poder ser un individuo que forme parte de la población.

```
Function GeneticAlgorithm(populationSize)
    actualPopulation ← generateStartingPopulation(populationSize);
    while not stop criterion met do
        parents ← selectParents (actualPopulation);
        children ← cross (parents);
        children ← mutate (children);
        actualPopulation ← reduce (actualPopulation, children);
        bestSolution ← getBestSolution (actualPopulation ∪ bestSolution);
    end
    return bestSolution;
end
```

Algoritmo 1: Algoritmo genético

Función de *fitness*.

La función de *fitness* o de adaptación será la encargada de establecer una medida de calidad de las soluciones; es decir, qué tan buena es una solución para el problema. Esta medida es importante para guiar el proceso de búsqueda, permitiendo que las soluciones de alta calidad se reproduzcan con más frecuencia para poder propagar sus genes a generaciones futuras.

En un problema de optimización combinatoria, la función de fitness podría ser la función objetivo del problema; sin embargo, podría utilizarse una medida distinta si se quisiera tener aspectos distintos en la evaluación, como puede ser su densidad respecto a la población (como lo hace el algoritmo SPEA2 [3]).

Selección.

La selección es el proceso por el cual se eligen las parejas de individuos a combinarse para producir los descendientes. La idea es seleccionar los individuos dando cierto favoritismo a aquellos mejor adaptados pero sin denegar la posibilidad de ser elegidos a los individuos con valores de adaptación bajos.

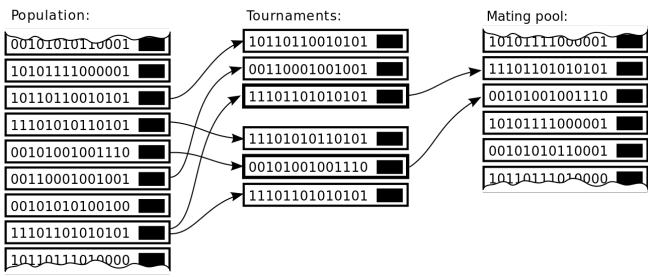

Figura 3.3: Selección por torneo.

Existen muchos métodos para realizar la selección, uno de ellos es la selección por torneo. En la figura 3.3 se muestra el ejemplo de un torneo de tamaño 3; donde se selecciona aleatoriamente 3 individuos de la población y de esos 3, se elige el mejor para ser añadido en la lista de soluciones a reproducirse. Este procedimiento se repite hasta haber encontrado todos los individuos que se espera que se reproduzcan en cada generación.

Recombinación.

La recombinación es el proceso de generar la población descendiente a partir de los padres seleccionados a reproducirse. Esto se logra al mezclar la información de cada padre, generando así hijos que conserven información de cada progenitor, con la idea de que mantengan "las buenas características" de cada padre.

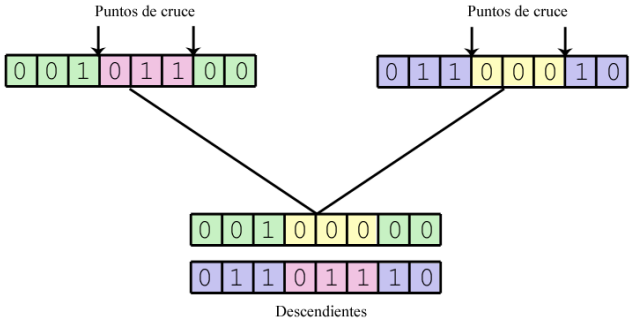

Figura 3.4: Operador de cruce.

La elección del operador de cruce será muy dependiente del problema que resuelve, así como de la codificación. En la figura 3.4 se muestra un cruce basado en 2 puntos para dos soluciones del problema de la mochila. Este operador selecciona aleatoriamente 2 puntos y crea 2 hijos, el primer hijo tendrá la primera y última parte del primer padre y la parte intermedia del segundo; el segundo hijo la primera y última parte con la información del segundo padre y la parte intermedia con la información del primero.

Mutación.

La mutación se le aplicará a las soluciones hijas antes de insertarlas en la población. Este operador pretende imitar a la mutación en la naturaleza, de modo que añada cierta diversidad en la población con el objetivo de permitir aparecer ciertos rasgos, que no se hubieran visto anteriormente, que puedan mejorar las soluciones de la población.

Al igual que el resto de operadores, existen muchos métodos de mutación. Para el problema de la mochila, se puede definir un operador de mutación como añadir un elemento a la solución con una determinada probabilidad (usualmente baja).

Reducción.

Una vez que se han generado los descendientes, se deberá decidir qué soluciones formarán parte de la siguiente generación, y cuales "morirán" en el proceso de selección natural. A esta operación se le conoce como operador de reducción, que es capaz de reducir la población actual y la nueva generación, a una única población que iniciará la siguiente generación.

Como todos los demás operadores, existen varias formas de realizar la reducción, una muy sencilla es descartar la población anterior, y considerar como nueva población a los descendientes, independientemente de su nivel de adaptación al medio.

3.2.2. Algoritmos meméticos.

Los algoritmos meméticos (*MA-Memetic Algorithm*) [4, p. 105] son algoritmos inspirados en los algoritmos genéticos que pretenden mejorar su desempeño. La idea del algoritmo memético es crear un algoritmo genético donde cada individuo se comporte de manera activa; que cada uno sea un agente que mejore realizando una búsqueda propia con información adicional del entorno. Esto se logra implementando técnicas de búsquedas locales que empleará cada individuo.

Como es de esperar, estas técnicas de búsqueda son dependientes del problema a resolver, con lo que, cada agente se valdrá de la información del problema para mejorar su nivel de adaptación al medio, en lugar de simplemente ser generado a partir de dos progenitores y luego mutado.

```
Function MemeticAlgorithm(populationSize, preservedSize)
    actualPopulation ← generateStartingPopulation(populationSize);
    actualPopulation ← improvePopulation (actualPopulation);
    while not stop criterion met do
        parents ← selectParents (actualPopulation);
        children ← cross (parents);
        children ← mutate (children);
        children ← improvePopulation (children);
        actualPopulation ← reduce (actualPopulation, children);
        bestSolution ← getBestSolution (actualPopulation ∪ bestSolution);
        if actualPopulation converged then
            actualPopulation ← reset (actualPopulation,
             populationSize,preservedSize);
        end
    end
    return bestSolution;
end
```

Algoritmo 2: Algoritmo memético

En la figura 2 se muestra el pseudocódigo de un algoritmo memético. Como se puede comprobar, difiere en dos cosas respecto al algoritmo genético; en primer lugar, se realiza una mejora de la población mediante técnicas de búsqueda tanto luego de generar la población inicial como luego de generar la población de descendientes. La otra diferencia, es la capacidad de reiniciar el algoritmo si se detecta que la población ha convergido. Se puede definir la convergencia de la población como un estado el que alcanza si se cumplen una serie determinada de condiciones, como puede ser por ejemplo, que el valor de adaptación de sus individuos es similar en un determinado porcentaje.

Una vez que se detecta la convergencia de la población, el algoritmo reinicia; pero en lugar de crear simplemente una nueva población desde cero, se crea una nueva preservando algunos de los mejores individuos de la última generación. Esto se detalla en el algoritmo 3.

```
Function reset(actualPopulation, populationSize, preservedSize)
    newPopulation ←
     generateStartingPopulation(populationSize − preservedSize);
    newPopulation ←improvePopulation(newPopulation);
    for i = 0 to preservedSize do
        newPopulation ← newPopulation ∪ extractBest(actualPopulation);
    end
    return newPopulation;
end
```

Algoritmo 3: Función de reinicio

3.2.3. Búsqueda dispersa.

La búsqueda dispersa (*SS- Scatter Search*) [4, p. 1] es un método de búsqueda que actúa sobre un conjunto de soluciones, llamado conjunto de referencia en lugar de población dado que este conjunto está compuesto por soluciones consideradas "buenas", que han sido obtenidas a partir de cálculos anteriores. Por soluciones "buenas" no solo se consideran aquellas que tienen un valor de *fitness* elevado, sino que también se considerarán soluciones "buenas" como para entrar en este conjunto de referencia aquellas soluciones que cumplan con una buena dispersión frente al resto de soluciones, entendiendo por dispersión, la distancia entre cada solución al resto de soluciones del conjunto de referencia. La medida de distancia varía según la representación. En una representación binaria, una medida de distancia entre dos soluciones puede ser el número de elementos en que son diferentes.

La búsqueda dispersa funciona de manera muy similar a un algoritmo memético, con la diferencia que en lugar de trabajar sobre una población, se trabaja con un conjunto de referencia que suele ser de un tamaño reducido (normalmente 10 veces más pequeño de lo que sería una población), donde una parte de este conjunto son soluciones consideradas "dispersas".

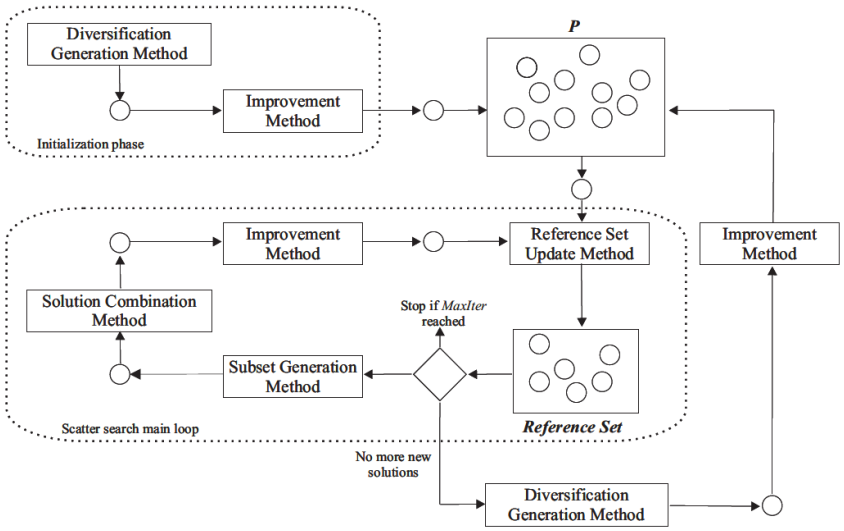

Figura 3.5: Búsqueda dispersa.

En la figura 3.5 se muestra una traza de una búsqueda dispersa. En él, se pueden identificar las 5 partes principales del método: método de diversificación, método de mejora, método de generación de subconjuntos, método de combinación de soluciones y método de actualización

del conjunto de referencia.

```
Function ScatterSearch(refSetSize)
    pool ← generateScatterPool;
    pool ← improve (pool);
    refSet ← updateRefSet (pool, refSetSize);
    while not stop criterion met do
        parents ← selectParents (refSet);
        children ← cross (parents);
        pool ← improve (children);
        refSet ← updateRefSet (pool, refSetSize);
        bestSolution ← getBestSolution (refSet ∪ bestSolution);
        if no new solutions in reference set then
            pool ← generateScatterPool;
            pool ← improve (pool);
            refSet ← updateRefSet (pool, refSetSize);
        end
    end
    return bestSolution;
end
```

Algoritmo 4: Búsqueda dispersa.

Método de diversificación.

Este método es utilizado al inicio y cada vez que se quiere reiniciar el algoritmo. Es el encargado de generar un conjunto de soluciones dispersas, que serán la entrada a la posterior fase de mejora del algoritmo.

Método de mejora.

Es el método encargado de mejorar las soluciones que reciba como entrada. Al igual que en los algoritmos meméticos, este método está encargado de llevar cada solución que reciba a un óptimo local. Con esto se consigue que las soluciones que vayan a entrar el conjunto de referencia sean siempre óptimos locales.

Método de generación de subconjuntos.

La generación de subconjuntos es el análogo al operador de selección en los algoritmos genéticos; ambos se encargan de seleccionar qué soluciones se deberán combinar en el paso posterior. La principal diferencia es, que al seleccionar sobre un conjunto de referencia de tamaño relativamente pequeño, este método suele escoger como subconjuntos a todas las posibles combinaciones de 2 soluciones del conjunto de referencia. Otra diferencia que tiene frente al operador de selección clásico en un algoritmo genético, es la posibilidad de permitir seleccionar subconjuntos con cardinalidad mayor a 2.

Método de combinación de soluciones.

Análogo al operador de cruce en un algoritmo genético. Sin embargo, la combinación de soluciones en una búsqueda dispersa tiene dos diferencias principales: la primera, puede permitir combinar más de dos soluciones, y, la segunda, la combinación no se realiza de forma completamente aleatoria; sino que para combinar las soluciones se tienen en cuenta los valores de adaptación de dichas soluciones. Existen varias formas de realizar la combinación de las soluciones teniendo en cuenta el valor de adaptación de las mismas, una de ellas es generar los rasgos de los descendientes de forma probabilística, en función del valor de adaptación del subconjunto a combinar; por ejemplo, generando una única solución, donde la probabilidad de que cada rasgo de dicha solución hija pertenezca a la i-ésima solución del conjunto a combinar viene dada por la ecuación 3.1.

$$p(i) = \frac{f(i)}{\sum_{j=1}^{n} f(j)} \tag{3.1}$$

Donde $f(i)$ es el valor objetivo de la solución i y $p(i)$ es la probabilidad que tiene cada rasgo de la solución hija a ser igual al de la i-ésima solución padre.

Método de actualización del conjunto de referencia.

Este método se encarga de reducir el *pool* de soluciones a un conjunto de referencia. La característica principal es que, además de tener en cuenta únicamente el valor de fitness de cada solución, tiene en cuenta su dispersión.

```
Function updateRefSet(pool,refSetSize)
    newRefSet ← extract refSetSize /2 bests solutions in pool;
    for i = 0 to refSetSize/ 2 do
        newRefSet ← newRefSet ∪ extractMostDisperse(pool);
    end
    return newRefSet;
end
```

Algoritmo 5: Método de actualización del conjunto de referencia.

En la figura 5 se muestra el método clásico de actualización de conjunto de referencia en una búsqueda dispersa. Este método, primero rellena la mitad del conjunto de referencia con las mejores soluciones que encuentra en el *pool*, y, la otra mitad la rellena con las soluciones más distantes a las soluciones que ya forman parte al conjunto de referencia.

3.3. Métodos basados en trayectorias.

Los métodos basados en trayectorias son aquellos métodos que, a diferencia de los métodos poblacionales, mejoran o construyen iterativamente una única solución hasta alcanzar distintos óptimos locales.

En las siguientes secciones se detallarán los algoritmos basados en trayectorias implementados en esta biblioteca.

3.3.1. Métodos *greedy*.

Quizás, los métodos *greedy* (Voraces) son los métodos más conocidos, dada su simplicidad y eficacia. Estos métodos se basan en una idea muy simple, mejorar una solución de forma iterativa, tomando en cada paso la mejor decisión, hasta que no sea posible mejorar la solución, en cuyo caso se considerará esa solución como un óptimo local y el método finalizará.

De la idea de realizar siempre el mejor movimiento viene el principal defecto, es un algoritmo "miope", es decir, no tiene en cuenta el comportamiento futuro, sino que se limita a ver únicamente el horizonte inmediato, lo que hará que caiga en óptimos locales sin poder salir de ellos.

Un ejemplo de un algoritmo *greedy* es el del vecino más cercano para el problema del viajante de comercio. Este método construye la solución seleccionando la siguiente ciudad a visitar como la siguiente ciudad más cercana a la última ciudad visitada (obviamente, sin visitar dos veces una misma ciudad). Este algoritmo da un resultado aproximado, pero claramente dista de ser el óptimo global.

Aunque los algoritmos *greedys* no den normalmente un óptimo global, suelen dar soluciones con al menos una mínima calidad y, es por eso que son usados para construir la solución inicial de muchos otros algoritmos más complejos.

3.3.2. Búsqueda local.

Las búsquedas locales (*LS-Local Search*) son métodos de mejora de soluciones de búsqueda por entornos, pues, parten de una solución inicial dada y la mejoran hasta alcanzar un óptimo local. En el proceso de búsqueda, estos algoritmos analizan el entorno de la solución, en busca de una mejor solución vecina, y pasan a ella para luego repetir el procedimiento hasta alcanzar algún óptimo local. De hecho, un *greedy* es un caso particular de una búsqueda local.

```
Function LocalSearch(startingSolution, neighborhood)
    actualSolution ← startingSolution;
    while better solution found do
        newSolution ← obtainSolutionFromNeighborhood (actualSolution,
         neighborhood);
        if newSolution is better than actualSolution then
            actualSolution ← newSolution;
        end
    end
    return actualSolution;
end
```

Algoritmo 6: Búsqueda local.

En algoritmo 6 se muestra el pseudocódigo de una búqueda local. Como se puede apreciar, hace falta especificar, en primer lugar, qué es un entorno, como se elijen las soluciones de un entorno y como se determina si una solución es mejor que otra (función de evaluación).

Función de evaluación.

Es la función que determinará si una solución es mejor que otra. Al igual que en los algoritmos genéticos, por ejemplo, esta función puede ser la misma que la función objetivo del problema;

aunque puede variar y podría ser una versión simplificada de la misma en caso de que la función objetivo real sea costosa de calcular; o incluso, existen métodos más sofisticados que usan varias funciones de evaluación, una función heurística fácil de calcular para decidir qué solución del entorno obtener, y la función objetivo del problema, presumiblemente más costosa de calcular, para decidir si moverse o no a la solución vecina.

Muestreo del entorno.

El muestreo del entorno es el paso en que se decide qué solución vecina examinar para luego decidir si moverse a ella o no. Como es de esperar, existen varias formas distintas de muestrear el entorno, siendo más conveniente utilizar una u otra en función de varios factores, como tamaño del entorno, coste de evaluar cada solución, tipo de problema a resolver, su necesidad de exactitud en las evaluaciones, etc..

Algunos métodos de muestreo comunes:

- **Muestreo *greedy*:** En este muestreo, se evalúan todas las soluciones del entorno y se obtiene la mejor vecina, convirtiendo así a la búsqueda local en un método *greedy*.
- **Muestreo ansioso:** Con el muestreo ansioso, la búsqueda en el entorno finaliza cuando se obtiene la primera solución que mejora la actual.
- **Muestreo aleatorio:** Se obtiene una solución aleatoria del entorno, o la mejor de una muestra seleccionada del entorno.

Estructura de entorno.

Lo único que resta para poder tener completa una búsqueda local es la definición de la estructura de entorno. En general, las estructuras de entorno son dependientes del problema y de la codificación de la solución. Sin embargo, el entorno de una solución se suele definir como el conjunto de soluciones resultantes de aplicar un movimiento a una determinada solución; entendiendo por movimiento algún tipo de operación sobre la misma (intercambiar dos elementos, sumar un valor a algún elemento, etc.). Esto da lugar a algunas estructuras de entornos predefinidas para cierto tipo de problemas, como puede ser el entorno con movimiento 2-Opt o el 3-Opt para el TSP (en general estos dos entornos valen para cualquier problema que codifique sus soluciones como permutaciones de enteros).

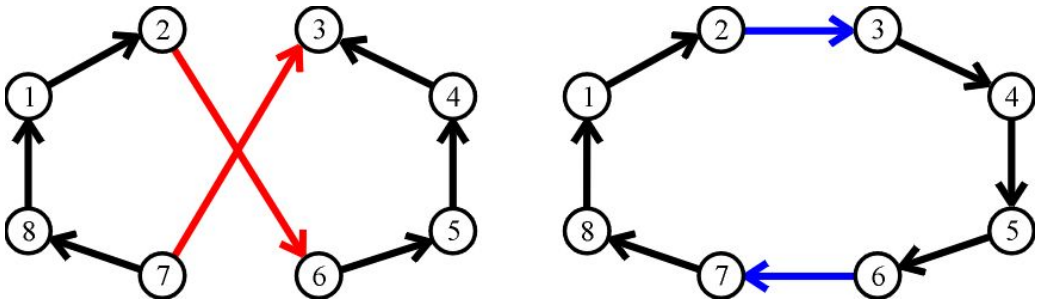

Figura 3.6: Movimiento 2-Opt para el problema del viajante de comercio.

3.3.3. Recocido simulado.

El método de recocido simulado (*SA-Simulated Annealing*) [4, p. 287] es una modificación de la búsqueda local, que permite escapar de los óptimos locales moviendo la solución actual a soluciones peores con mayor probabilidad al inicio del algoritmo, mientras que a medida que se acerca al enfriamiento (número máximo de iteraciones) disminuye la probabilidad de empeorar la solución actual hasta no permitir dicho empeoramiento.

El nombre e inspiración viene del proceso de recocido del acero y cerámicas, una técnica que consiste en calentar y luego enfriar lentamente el material para variar sus propiedades físicas. El calor causa que los átomos aumenten su energía y que puedan así desplazarse de sus posiciones iniciales (un mínimo local de energía); el enfriamiento lento les da mayores probabilidades de recristalizar en configuraciones con menor energía que la inicial (mínimo global).

```
Function SimulatedAnnealing(startingSolution, neighborhood)
    actualSolution ← startingSolution;
    while system not cool do
        newSolution ← obtainRandomSolutionFromNeighborhood (actualSolution,
         neighborhood);
        if newSolution is better than actualSolution then
            actualSolution ← newSolution;
        end
        else if can accept new solution then
            actualSolution ← newSolution;
        end
        decrease system temperature;
    end
    return actualSolution;
end
```

Algoritmo 7: Recocido simulado.

En el algoritmo 7 se muestra el pseudocódigo de un recocido simulado. Para comprender el funcionamiento de este algoritmo, hace falta especificar el parámetro de "temperatura", cuando se considera que el sistema está frio y cómo se aceptan las soluciones que empeoran la actual.

La temperatura es el parámetro que utiliza el algoritmo tanto para determinar en qué momento parar (a modo de criterio de parada) como para decidir si aceptar o no una solución peor. La temperatura puede ser simplemente un número que, con el paso de las iteraciones disminuye hasta alcanzar un determinado valor que será considerado como punto en el que el sistema se enfría. El modo en que disminuye puede variar de una implementación a otra, desde un simple decremento hasta una función más compleja que dependa del número de iteraciones del algoritmo.

La decisión de aceptar o no una solución peor se realiza de forma probabilística en función de la temperatura del sistema y del grado de empeoramiento de dicha solución respecto a la actual. Al depender tanto de la temperatura como de la calidad de la solución nueva, se podrá diseñar una función probabilística que se adapte al problema a resolver que permita aceptar soluciones peores en las primeras iteraciones del algoritmo, mientras que hacia las últimas iteraciones, la probabilidad de aceptar una solución peor sea muy baja.

3.3.4. Búsqueda local rápida.

La búsqueda local rápida (*FLS- Fast Local Search*) [4, p. 188] es una mejora de la búsqueda local que pretende disminuir los tiempos para la ejecución del algoritmo. Uno de los factores que más afecta a la eficiencia de las búsquedas locales es el tamaño de la estructura de entornos. Buscar en entornos muy grandes resulta ser muy lento, sobre todo si el coste de evaluar una solución es elevado. Una forma de resolver esto puede ser implementar un muestreo ansioso, de modo que no sea necesario explorar todo el entorno de cada solución, si no que baste con explorar hasta que se encuentre una solución mejor que la actual; sin embargo, esto puede llevar a una convergencia precipitada a un óptimo local próximo.

La búsqueda local rápida divide el entorno en sub-entornos. Estos sub-entornos no pueden ser conjuntos disjuntos, y la unión de todos ellos debe dar como resultado el entorno al completo. En cada iteración, el algoritmo buscará únicamente en los sub-entornos donde se prevee que hayan vecinos prometedores e ignorando el resto; logrando así reducir considerablemente el tamaño del entorno a muestrear. La búsqueda finalizará cuando se considere que en ninguno de los sub-entornos se pueda encontrar soluciones prometedoras.

```
Function FLS(startingSolution)
    actualSolution ← startingSolution;
    activate all sub-neighborhoods;
    while active sub-neighborhood exist do
        subNeighborhood ← get next active sub neighborhood;
        newSolution ← obtainBestFromSubNeighborhood (actualSolution,
         subNeighborhood);
        if newSolution is better than actualSolution then
            actualSolution ← newSolution;
            activate sub-neighborhoods that contains newSolution;
        end
        else
            deactivate subNeighborhood;
        end
    end
    return actualSolution;
end
```

Algoritmo 8: Búsqueda local rápida.

En el algoritmo 8 se muestra el pseudocódigo de una búsqueda local rápida. La búsqueda, en su primera iteración, búscara en todos los sub-entornos si es necesario hasta dar con una solución mejor, desactivando aquellos que no contuvieron ninguna solución capaz de mejorar la actual (en el caso peor, en la primera iteración se comportará incluso peor que un *greedy*). Una vez encontrada la solución vecina a la que moverse, se activan todos los sub-entornos que también contengan a dicha solución, de este modo, se podrán reactivar sub-entornos previamente desactivados, para así evitar una convergencia precipitada. Como la mejora de este algoritmo radica en la división del entorno, es deseable que dicha división no sea costosa y que mantenga la coherencia con la semántica del problema, además de permitir realizar la operación de verificar si una solución pertenece o no a un sub-entorno de manera eficiente.

El objetivo de esta modificación es poder realizar búsquedas locales de manera mucho más rápida, sin embargo, esto se obtiene a costa de ignorar ciertos sub-entornos que puedan contener mejoras. Por esta razón, este algoritmo por si solo no suele obtener soluciones de alta calidad;

no obstante, al combinarlo con algoritmos basados en multi-arranque se puede compensar la pérdida de calidad con la ganancia en velocidad de cada búsqueda.

3.3.5. Métodos *GRASP*.

Los métodos *GRASP (Greedy Randomized Adaptative Search Procedures)* [4, p. 219] son métodos de búsqueda que constan de dos fases: una primera fase de construcción, donde a través de una estrategia *greedy* aleatorizada construyen una solución, y una segunda fase de post-procesado, donde se mejora la solución obtenida en el paso anterior.

```
Function GRASP()
    solution ← greedyRandomizedConstruction ();
    solution ← improveSolution (solution);
    return solution;
end
```

Algoritmo 9: *GRASP* básico.

En el algoritmo 9 se muestra el pseudocódigo de un método *GRASP* básico, en él, se puede apreciar claramente las dos fases: la constructiva y la de mejora.

Fase constructiva.

La fase constructiva se realiza utilizando un método *greedy* aleatorizado, donde en cada paso de la estrategia *greedy*, en lugar de realizar la mejor acción posible, se realiza una de las mejores acciones encontradas seleccionada aleatoriamente, lo que permite suavizar el efecto "miope" que tienen los métodos *greedy*.

La forma de agregar el componente aleatorio puede variar de una implementación a otra. Una forma es seleccionar un parámetro de entrada k que será el tamaño de la llamada "lista restringida de candidatos". Esta lista estará formada por los k mejores candidatos encontrados durante la exploración, y sobre ella se realizará la selección aleatoria.

También se puede trabajar con una lista restringida de candidatos de tamaño variable, en cuyo caso, se deberá especificar un parámetro α que indicará el grado de diferencia máxima que puede tener una solución respecto a la mejor encontrada para poder formar parte de la lista restringida de candidatos.

Otro enfoque también puede ser trabajar con todo el entorno como lista restringida de candidatos y realizar la selección aleatoria utilizando un método de ruleta sesgada, dando mayor probabilidad a los candidatos más prometedores.

Método de mejora.

En la fase de post-procesado, se aplica una mejora a la solución encontrada por el método *greedy* aleatorizado permitiendo así asegurar que la solución final será un óptimo local. El método de mejora a utilizar puede variar mucho, desde una simple búsqueda local, a una búsqueda tabú, por entornos variables, etc..

Posibles mejoras.

Dado el componente aleatorio de los métodos *GRASP*, es posible aplicar una mejora fácil y rápida al realizar varias ejecuciones del algoritmo y recordando la mejor solución encontrada. Esta es una mejora muy común y fácil de realizar, convirtiendo el método *GRASP* en un método multi-arranque.

3.3.6. Multi-arranque.

Los métodos multi-arranque (*MS-Multi Start*) [4, p. 355] son métodos basado en una idea sencilla, realizar múltiples búsquedas partiendo de distintos puntos o siguiendo distintos caminos. Estos métodos constan de dos fases, una primera fase generadora de una solución, la fase constructiva, y una segunda fase de mejora (no siempre presente). De esta forma, se pueden realizar varias búsquedas desde distintos puntos del espacio de soluciones, intentando conseguir aproximarse al óptimo global.

```
Function MultiStart()
    for i = 0 to maxIterations do
        solution ← generateSolution ();
        solution ← improveSolution (solution);
        updateBest (solution);
    end
    return bestSolution;
end
```

Algoritmo 10: Multi-arranque básico.

En el algoritmo 10 se muestra un multi-arranque básico. Un multi-arranque puede formarse con cualquier combinación de métodos de generación y mejora de soluciones. Un multi-arranque muy básico puede ser generar una solución aleatoria y mejorarla con una búsqueda local, de este modo, con cada reinicio se partirá de una solución distinta, permitiendo alcanzar diversos óptimos locales.

El objetivo principal de reiniciar la búsqueda es poder escapar de los óptimos locales, para ello, se pueden aplicar varias mejoras que permitan reiniciar la búsqueda desde puntos más prometedores del espacio de soluciones.

Una forma puede ser aplicar algún tipo de "memoria" al algoritmo, es decir, permitir que recuerde las "buenas características" de los óptimos locales encontrados, de modo que, se pueda reiniciar la búsqueda a partir de una solución que contenga alguna de dichas características, esperando que el punto de partida sea más próximo al óptimo global y así poder alcanzarlo.

También se puede puede aplicar métodos de diversificación a cada reinicio. En lugar de utilizar un reinicio completamente aleatorio, se puede utilizar un reinicio "guiado" donde la solución se construye teniendo en cuenta las características de las soluciones vistas anteriormente, intentando que la nueva solución sea distinta, para así explorar zonas del espacio de soluciones no exploradas aún.

Estas dos formas de plantear el reinicio plantean una dualidad para este tipo de algoritmo: intensificación frente a diversificación. Se debe decidir sobre qué zonas intensificar las búsquedas, y en qué momento diversificar para poder explorar nuevas zonas. Como es de esperar, cualquiera de estos métodos son dependientes del problema a resolver y de la codificación que se haga de

las soluciones, con lo que no existe una única forma eficiente de diseñar un algoritmo multi-arranque, dando lugar a múltiples propuestas.

3.3.7. *VNS*. Búsqueda por entorno variable.

La búsqueda por entorno variable (*VNS-Variable Neighborhood Search*) [4, p. 145] es un método de búsqueda iterativo que se basa en un principio muy simple: cambiar de entorno para escapar de óptimos locales. Es un método que realiza varias búsquedas locales, variando el entorno en cada óptimo local, hasta encontrar un óptimo local que no pueda ser mejorado con ninguna estructura de entorno. Para definir un *VNS*, hace falta una lista de entornos sobre los que buscar y un método de mejora de las soluciones, normalmente una búsqueda local.

```
Function VNS(startingSolution, neighborhoods)
    actualSolution ← startingSolution;
    while not all neighborhoods explored do
        newSolution ← obtainRandomFromNeighborhood (actualSolution,
         actualNeighborhood);
        newSolution ← localSearch (newSolution, actualNeighborhood);
        if newSolution is better than actualSolution then
            actualSolution ← newSolution;
            actualNeighborhood ← first neighborhood ;
        end
        else
            actualNeighborhood ← next neighborhood ;
        end
    end
    return actualSolution;
end
```

Algoritmo 11: VNS.

El algoritmo comienza recibiendo una solución inicial y una lista de entornos que explorar. El primer paso de cada iteración, llamado *shaking* (sacudida), obtiene una solución vecina aleatoria utilizando el entorno actual para luego mejorar utilizando una búsqueda local. Una vez finalizada la búsqueda local, se decide si aceptar la solución si mejora la actual, volviendo a explorar a partir del primer entorno, o si rechazar la solución encontrada por el método de mejora y continuar explorando el siguiente entorno.

Variante: *VND*.

La variante *VND* (*Variable Neighborhood Descendant*) [4, p. 147] se obtiene al omitir la fase de *shaking* y en lugar de ello se obtiene siempre la mejor solución del entorno.

Como se muestra en el algoritmo 12, tampoco se realiza la fase de búsqueda local, pues esa fase realiza las mismas operaciones que el primer paso del algoritmo de obtener la mejor solución del entorno.

```
Function VND(startingSolution, neighborhoods)
    actualSolution ← startingSolution;
    while not all neighborhoods explored do
        newSolution ← obtainBestFromNeighborhood (actualSolution,
         actualNeighborhood);
        if newSolution is better than actualSolution then
            actualSolution ← newSolution;
            actualNeighborhood ← first neighborhood ;
        end
        else
            actualNeighborhood ← next neighborhood ;
        end
    end
    return actualSolution;
end
```

Algoritmo 12: VND.

Variante: *RVNS*.

La variante *RVNS* (*Reduced VNS*) [4, p. 147] se obtiene al omitir la fase de mejora despues de obtener una solución aleatoria del entorno (fase de *shaking*). Tal y como se puede apreciar en el algoritmo 13, se debe añadir un bucle exterior al algoritmo para no parar aunque se hayan visitado todos los entornos, pues, encontrar una solución peor durante un muestreo aleatorio será algo muy frecuente, con lo que se deberá repetir el procedimiento varias veces hasta obtener una solución de calidad (convirtiendo así a la búsqueda en un multi-arranque).

```
Function RVNS(startingSolution, neighborhoods)
    actualSolution ← startingSolution;
    while stop condition not met do
        return to first neighborhood ;
        while not all neighborhoods explored do
            newSolution ← obtainRandomFromNeighborhood (actualSolution,
             actualNeighborhood);
            if newSolution is better than actualSolution then
                actualSolution ← newSolution;
                actualNeighborhood ← first neighborhood ;
            end
            else
                actualNeighborhood ← next neighborhood ;
            end
        end
        update bestSolution;
    end
    return bestSolution;
end
```

Algoritmo 13: RVNS.

3.3.8. Búsqueda local guiada.

La búsqueda local guiada (*GLS-Guided Local Search*) [4, p. 185] es una técnica heurística generalizada y extendida a partir de un método basado en una red neuronal llamado *GENET*, un método ponderado para la satisfacción de restricciones. La búsqueda local guiada intenta distribuir el esfuerzo de búsqueda por las áreas más prometedoras del espacio de soluciones. Para ello, se vale de una función heurística que es modificada en cada iteración para poder así favorecer (o penalizar) la exploración de determinadas zonas del espacio.

Como se ha dicho, *GLS* se vale de una función heurística diferente a la usada en una búsqueda local convencional. Esta función está formada en parte por la función a optimizar (la que se utilizaría como función heurística en una búsqueda local) y por una medida de penalización de cada elemento de la solución a evaluar. Entendiendo por elemento de una solución, por ejemplo, en el problema de la mochila el elemento i-ésimo de la solución puede ser el indicador si el objeto i está dentro de la mochila. Formalmente, la función de evaluación $h(s)$ (suponiendo un problema de minimización), se define en 3.2.

$$h(s) = g(s) + \lambda * \sum (p_i * l_i(s)) \tag{3.2}$$

En la ecuación 3.2 s es la solución a evaluar, $g(s)$ es el valor objetivo de la solución s, λ es el parámetro de "peso" de las penalizaciones del *GLS*, el índice i varía entre 1 y el número total de elementos de la solución, p_i es el número de veces que se ha penalizado i, mientras que $l_i(s)$ indica si la solución s contiene al elemento i.

$$l_i(s) = \begin{cases} 1 & \text{si s contiene el elemento i} \\ 0 & \text{en otro caso.} \end{cases} \tag{3.3}$$

El método *GLS* modifica las penalizaciones de los elementos luego de cada ejecución de la búsqueda local, penalizando los elementos presentes en el óptimo local que son más desfavorables. La intención es seleccionar para penalizar aquellos elementos que tengan un mayor coste y que menos hayan sido penalizados. De modo que, para decidir qué elemento penalizar, se calcula la utilidad de penalizar cada elemento, y se penalizan aquellos elementos de máxima utilidad. La utilidad de penalizar el elemento i, bajo el óptimo local s_* se define en la ecuación 3.4.

$$util_i(s_*) = l_i(s_*) * \frac{c_i}{1 + p_i} \tag{3.4}$$

Donde c_i es el coste del elemento $_i$. En otras palabras, si el elemento i no está presente, entonces la utilidad de penalizarlo es 0, si está presente, la utilidad dependerá del coste de i y de las veces que haya sido penalizado. Teniendo en cuenta el coste y la penalización actual, la búsqueda se centrará en aquellas áreas más prometedoras del espacio de soluciones, áreas donde se prevee que se encuentren soluciones con mejores características, a la vez que se evita centrar todo el esfuerzo de búsqueda en una zona en concreto.

En el algoritmo 14 se muestra el pseudocódigo del *GLS*. Claramente, la elección de lo que se considera "elemento" de una solución (decisión que normalmente va ligada a la codificación de las soluciones), elección del parámetro λ, criterio de parada y solución de partida de las búsquedas locales afectan la eficiencia del algoritmo. Puede aplicarse como criterio de parada un límite de iteraciones, mientras que la elección de los elementos y su coste suele extraerse de la función objetivo y del problema mientras que el parámetro λ suele afectar en menor medida al desempeño del algoritmo. En cuanto a la solución inicial, puede calcularse de manera aleatoria,

```
Function GLS(startingSolution)
    set all penalties to 0 ;
    while stop condition not met do
        newSolution ← localSearch (newSolution, heuristicFunction) ;
        updatePenalties (newSolution);
        updateHeuristic (heuristicFunction);
        if newSolution is better than bestSolution then
            bestSolution ← newSolution;
        end
    end
    return bestSolution;
end
```

Algoritmo 14: GLS.

a través de alguna heurística como un *greedy*, o bien se puede crear un algoritmo más complejo al modificar la solución de partida en cada iteración, creando así un híbrido entre un *ILS* y *GLS*.

3.3.9. Búsqueda local iterativa.

La búsqueda local iterativa, *ILS* [4, p. 321] es una mejora a la búsqueda local. La idea básica es concatenar búsquedas desde distintos puntos pará así encontrar óptimos locales diferentes. Obsérvese que cada vez que inicia una búsqueda local, si esta es determinista, al empezar en una misma solución acabará siempre en el mismo óptimo local. Pero también, si empieza de una solución diferente pero que está más cercana a ese óptimo local que a ningún otro, la búsqueda acabará igualmente en el mismo óptimo local.

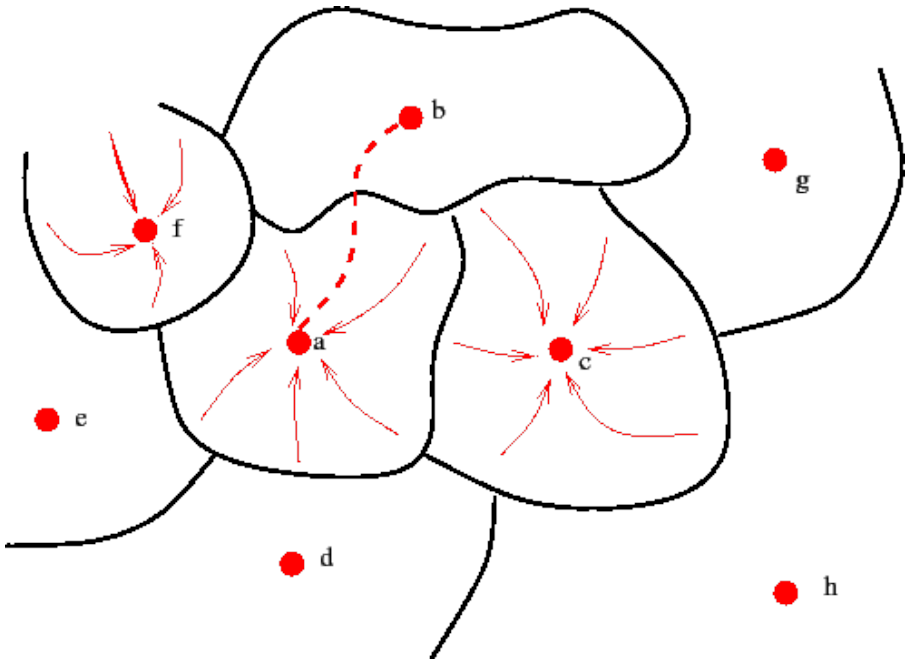

Figura 3.7: Espacio de soluciones.

En la figura 3.7 se muestra una representación del espacio de soluciones. Donde los puntos denotados con letras representan el óptimo local del área al que pertenecen. De modo que,

como indican las flechas, cualquier solución dentro de una determinada área, tenderá al óptimo local de dicha área. De todos los óptimos locales que existen en el espacio de soluciones, solo uno representa al global; de modo que, para encontrar ese óptimo global se deberá iniciar una búsqueda local desde una solución que pertenezca al área de dicho óptimo.

Para ello se reinician las búsquedas, cada vez desde un punto distinto. El modo en que se reinicia la búsqueda es determinante para el desempeño del algoritmo. Una forma simple de hacerlo es reiniciar la búsqueda desde un punto aleatorio del espacio en cada iteración. Con ello se consigue sin duda explorar mejor el espacio de soluciones, pero se desperdicia la información que se hubiera podido obtener a partir de la ejecución de las búsquedas.

Idealmente, si se considera como entorno de una solución todos aquellas soluciones que son óptimos locales vecinas a la misma, se podría alcanzar el óptimo global. Sin embargo, definir de forma exacta este entorno es complicado por varias razones, entre ellas está que para alcanzar un óptimo local primero hay que realizar una búsqueda local. *ILS* alcanza esto de manera heurística, de modo que a partir de un óptimo local s_* genera una solución s' con la intención de que esa nueva solución esté lo bastante alejada como para pertenecer a un área diferente, pero a su vez lo bastante cerca como para no diferenciarse mucho de s_*. A este paso de modificar s_* se le llama perturbación.

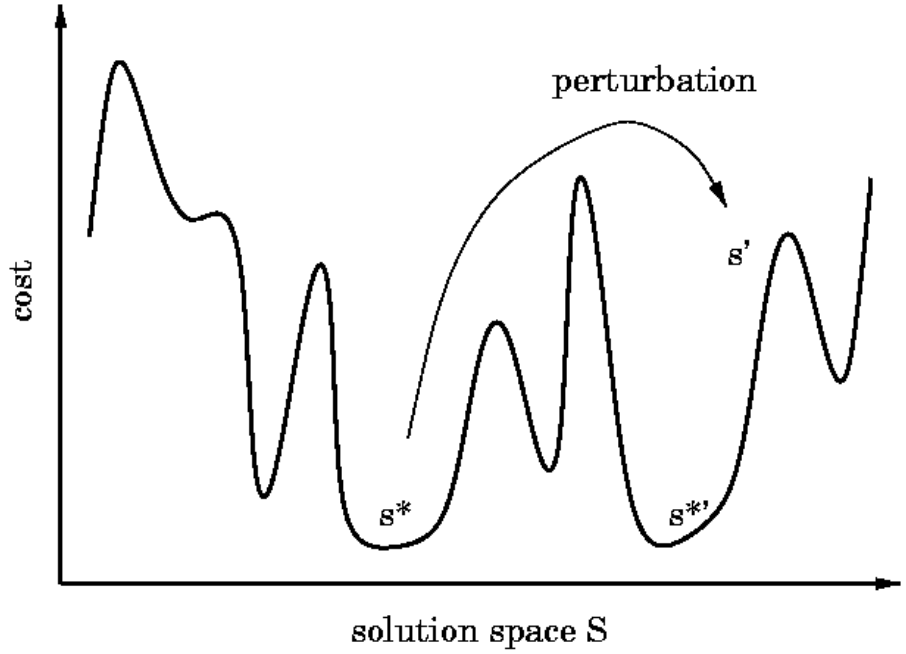

Figura 3.8: ILS.

En la figura 3.8 se muestra el resultado de aplicar una perturbación ideal, donde s_* se modifica lo suficiente, como para poder alcanzar un óptimo diferente en la siguiente búsqueda local. Como es de esperar, el modo de aplicar la perturbación determinará la eficacia del algoritmo. Además, se debe tener en cuenta que el tamaño de las áreas del espacio de soluciones es variable, con lo que aplicar una perturbación determinista no dará resultados de calidad. En cambio, se aplican perturbaciones adaptativas que dependen del resultado de la última búsqueda realizada, para poder cambiar durante la ejecución el tamaño de los saltos y así poder encontrar distintos óptimos en cada búsqueda.

En el algoritmo 15 se muestra el pseudocódigo de un *ILS*. La forma de realizar las perturbaciones se basa en el historial de soluciones encontradas, de modo que si se detecta que se ha caido en el mismo óptimo varias veces, se va incrementando el tamaño de la perturbación

```
Function ILS(startingSolution)
    actualSolution ← localSearch (startingSolution);
    while stop condition not met do
        newSolution ← perturbation (actualSolution, history);
        newSolution ← localSearch (newSolution);
        if acceptanceCriterion (newSolution, actualSolution, history) then
            actualSolution ← newSolution;
        end
        if newSolution is better than bestSolution then
            bestSolution ← newSolution;
        end
    end
    return bestSolution;
end
```

Algoritmo 15: ILS.

hasta poder escapar. Cabe destacar también que el criterio de aceptación de un óptimo depende también del historial de soluciones encontradas. En la literatura, la mayor parte de las implementaciones han sido realizadas sin la memoria proporcionada por el historial, utilizando un método de aceptación que utiliza únicamente la solución actual y el óptimo local generado por la búsqueda local.

3.3.10. Búsqueda tabú.

La búsqueda tabú (*TS-Tabu Search*) [4, p. 37] es otra mejora de la búsqueda local. La idea básica de la búsqueda tabú es recordar una serie de movimientos realizados durante la búsqueda, convirtiéndolos en "tabú" o prohibidos. Con esta memoria que se añade y aceptando siempre la mejor solución no tabú del entorno, se consigue una búsqueda tabú que permite escapar de óptimos locales utilizando la memoria tanto para escapar de ellos como para evitar ciclarse.

```
Function TabuSearch(startingSolution, neighborhood)
    actualSolution ← startingSolution;
    while not reached maximum iterations do
        newSolution ← obtainBestNonTabuSolutionFromNeighborhood
         (actualSolution, neighborhood);
        add to tabu newSolution;
        actualSolution ← newSolution;
        if newSolution is better than bestSolution then
            bestSolution ← newSolution;
        end
    end
    return bestSolution;
end
```

Algoritmo 16: Búsqueda tabú.

En el algoritmo 16 se muestra el pseudocódigo de una búsqueda tabú. La lista de elementos tabú dura durante un número determinado de iteraciones, una vez pasado ese número de iteraciones, el elemento deja de ser tabú, pudiendo ser elegido en las próximas búsquedas.

La forma en que se hace tabú un elemento depende tanto de la codificación de las soluciones como del problema en concreto. Una forma de crear una lista tabú es añadir qué elementos de la solución no pueden ser modificados, de esta forma, si se tiene la solución {0, 1, 1, 0} para el problema de la mochila y el elemento 2 es tabú, cualquier solución que no tenga un "1" en la segunda posición será tabú.

Otro modo de implementar la lista tabú es prohibiendo movimientos que reviertan al último realizado. Por ejemplo, en el problema del viajante de comercio, si se tiene la solución {1, 2, 3, 4} y se pasa a la solución {2, 1, 3, 4}; se puede prohibir el movimiento de intercambio del primer con el segundo elemento.

Durante la ejecución del algoritmo, es posible que se encuentren soluciones tabú que pueda ser interesante visitar, aún siendo tabú, para mejorar la calidad de la mejor solución encontrada. Para permitir ignorar la restricción tabú y visitar dichas soluciones, se establecen criterios de aspiración. Estos criterios son una serie de condiciones que, de ser cumplidas, permiten ignorar el criterio tabú para aceptar una solución. Un criterio de aspiración muy común es permitir aceptar una solución que sea mejor que la mejor solución encontrada hasta el momento.

La lista tabú también es llamada memoria a corto plazo, en contraposición con la memoria a largo plazo *LTM (Long term memory)*. La *LTM* se utiliza para aplicar una estrategia de diversificación. Cuando el algoritmo llega a su criterio de parada, en lugar de finalizar y devolver la mejor solución encontrada, se puede reiniciar la búsqueda desde un punto diferente, este punto diferente puede ser calculado utilizando la *LTM*. La *LTM* guarda la frecuencia con la que han aparecido ciertas características en las soluciones que se han ido visitando, y perdura a lo largo de las ejecuciones, a diferencia de la memoria a corto plazo; con esta información, se puede crear una solución de partida que contenga los elementos menos frecuentemente vistos para así explorar zonas del espacio de soluciones poco visitadas (diversificación de la búsqueda).

Capítulo 4

La biblioteca.

En este capítulo se detallará la biblioteca de clases. Se explicará su estructura, cada uno de los módulos y métodos implementados, así como ejemplos de uso de este *framework*.

4.1. Estructura del *framework*.

Esta biblioteca está diseñada siguiendo una arquitectura orientada a objetos, lo que facilita la extensión de la misma mediante la inclusión de módulos adicionales. Provee al usuario de una base con algoritmos, métodos, operadores y estructuras ya creadas y preparadas para su utilización, así como la capacidad de crear nuevas estructuras, operadores o métodos tan complejos como se requiera, que podrán ser añadidos al *framework* y utilizados del mismo modo que el resto de componentes.

4.1.1. Estructura de las soluciones.

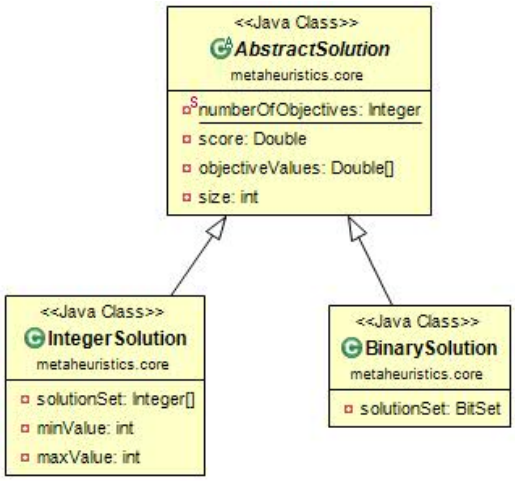

Figura 4.1: Diagrama de las soluciones.

En la figura 4.1 se muestra la estructura utilizada para representar las soluciones a los problemas. Todas las soluciones deberán heredar de la clase base abstracta *AbstractSolution* e implementar los métodos específicos a cada representación de soluciones, como son los métodos para obtener elementos de una solución según un índice, o de modificación de un elemento.

Además, toda solución tiene la información de su tamaño (cuantos elementos o variables posee), cantidad de objetivos que ha de cumplir (dependiente del problema, en nuestro caso será siempre 1), valor de la evaluación de la solución para cada objetivo y puntuación heurística. Se diferencia el concepto de puntuación con el del valor objetivo, pues, la puntuación de una solución es una valoración heurística de su *fitness* que será modificada por los evaluadores heurísticos, mientras que los valores objetivos son la medida de desempeño para cada función objetivo, y serán modificados por las funciones objetivo.

Esta biblioteca, en su estado actual proporciona la implementación de dos tipos de soluciones: soluciones binarias y enteras (incluyendo permutaciones de enteros).

Las soluciones binarias son soluciones que son representadas mediante un *array* de bits; especialmente útiles para representar las soluciones de los problemas de asignación o de la mochila. Las soluciones enteras son soluciones representadas con un *array* de números enteros que varían en un rango determinado; son ideales para la representación de las soluciones del TSP.

4.1.2. Diseño para los problemas.

En la figura 4.2 se muestra la estructura de clases para la representación de problemas. Todos los problemas deberán heredar de la clase *Abstractproblem*. Esta clase está compuesta por una serie de objetivos de tipo *MultiobjectiveFunction*, que en nuestro caso será únicamente un objetivo, es solo un *placeholder* para dejar el diseño preparado para una posible extensión del *framework*; unos datos que serán específicos para cada problema (por ejemplo, el grafo de ciudades en un TSP) y una serie de restricciones, también específicas para cada problema.

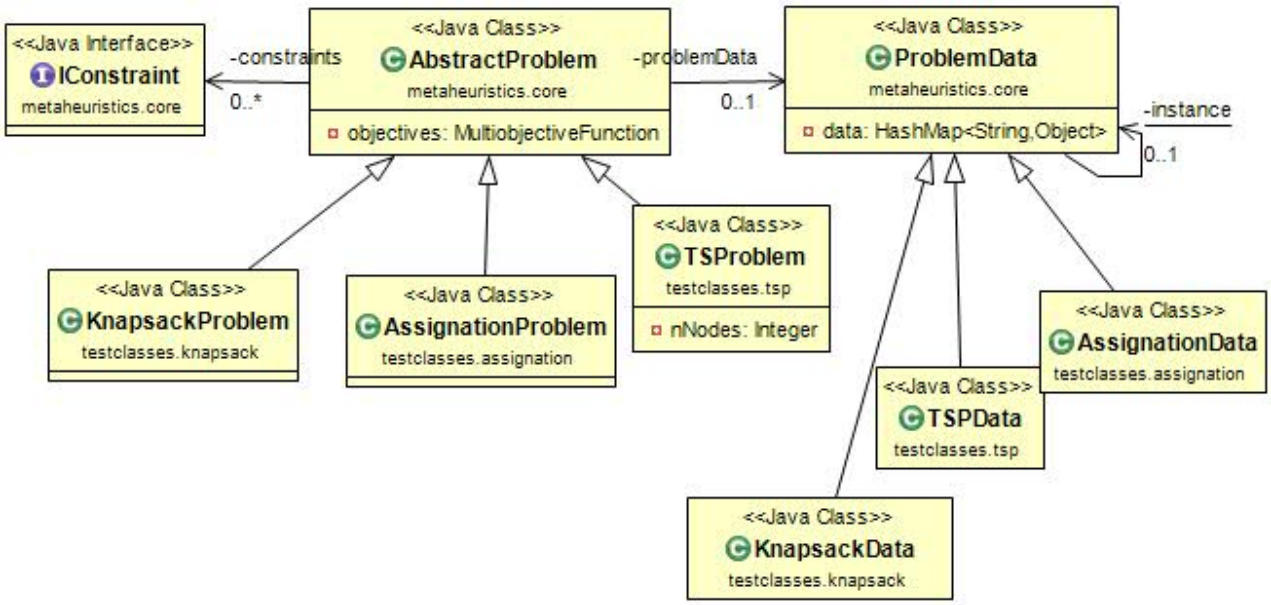

Figura 4.2: Diagrama de los problemas.

Nuestra biblioteca proporciona ya implementados los problemas del TSP, mochila y asignación. Cada clase heredara de *AbstractProblem* y se ocupa de asignar correctamente las funciones objetivo, restricciones y datos para que la clase represente al problema deseado.

4.1.3. Diseño para las funciones de evaluación.

En la figura 4.3 se muestra el diagrama de clases de las funciones de evaluación. De forma directa o indirecta, todas las funciones de evaluación deberán heredar de la clase base *AbstractEvaluationFunction* e implementar los siguientes métodos:

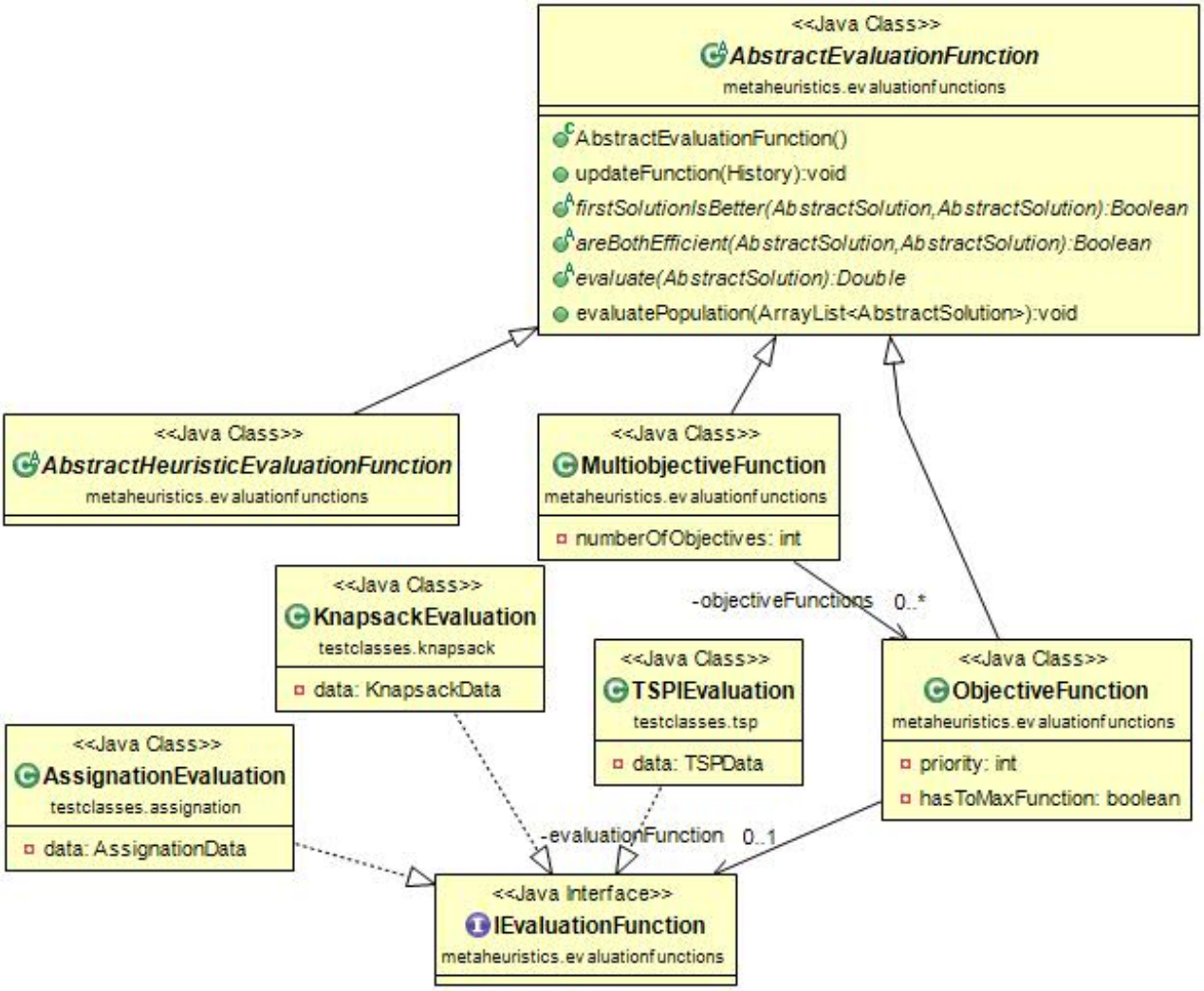

Figura 4.3: Diagrama de las funciones de evaluación.

- El método de evaluación de soluciones, *evaluate*, que será el encargado de modificar los valores objetivos o puntuación de una solución en función de si se trata de un método heurístico o de una función objetivo. Este método se utiliza también para evaluar poblaciones, por defecto evaluará las soluciones una a una, pero se puede reimplementar para modificar su comportamiento y que evalúe una población completa.
- El método *firstSolutionIsBetter* que será el encargado de determinar si una solución es mejor que otra.
- El método que determina si dos soluciones son equivalentes: *areBothEfficient*.
- El método de actualización de la función objetivo *updateFunction*. Por defecto este método no hace nada, sin embargo, en algunos casos puede ser interesante modificar la función objetivo, como puede ser en el caso de una búsqueda local guiada.

De la clase base, heredan las clases de función heurística abstracta, función multi-objetivo y función objetivo (*AbstractHeuristicFunction*, *MultiobjectiveFunction* y *ObjectiveFunction* respectivamente).

La clase función heurística abstracta será la base para todas las funciones heurísticas que modifiquen la puntuación de una solución durante su evaluación.

La clase función multi-objetivo estará compuesta por varias funciones objetivo. De momento solo se utiliza a modo de *placeholder*, y sus métodos de evaluación, comparación y actualización dependen de los mismos métodos de la única función objetivo que contendrá.

La clase función objetivo será la encargada de comparar las soluciones modificando su valor objetivo. Tiene como atributo la forma en que tiene que comparar (si maximizando o minimizando) y el modo en que se evalúa las soluciones mediante la interfaz *IEvaluationFunction*.

La interfaz de evaluación de soluciones es la única parte específica a cada problema de esta estructura, y será la encargada de evaluar una solución en función de la misma y de los datos del problema a resolver.

Con esta estructura, para obtener una función de evaluación específica para un determinado problema, basta con implementar la interfaz de evaluación junto con la estructura de datos del problema y cargarla en una clase función objetivo especificando si la función es de maximizar o minimizar.

4.1.4. Diseño para los algoritmos.

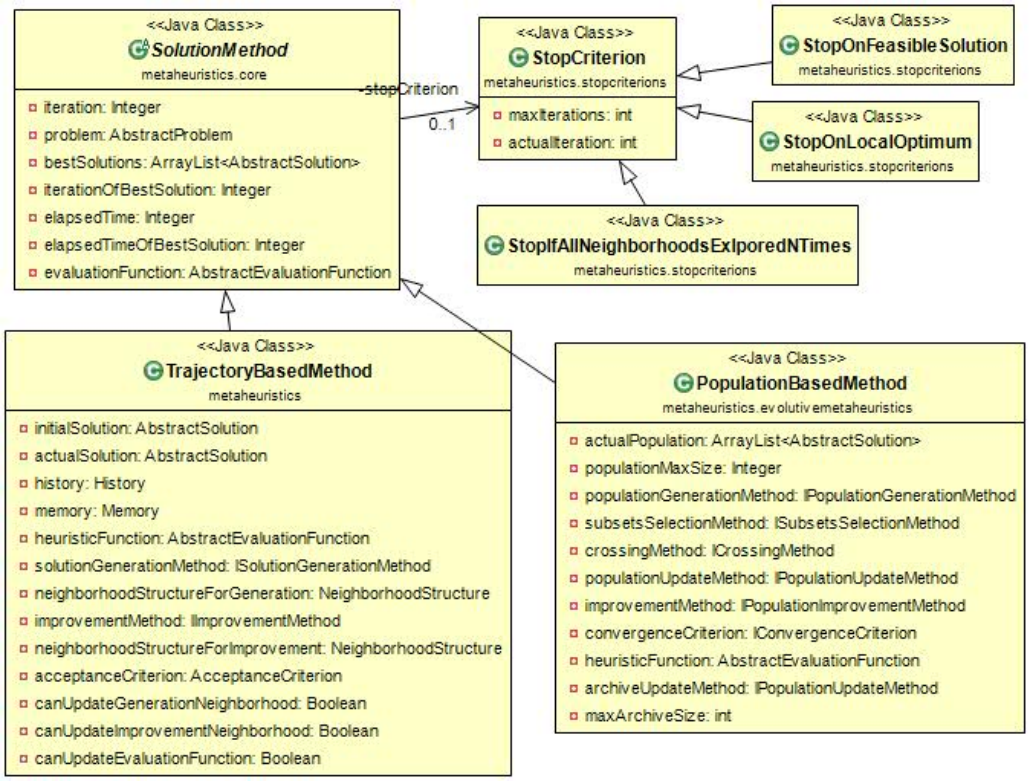

Figura 4.4: Diagrama de clases para los algoritmos.

En la figura 4.4 se muestra el diagrama de clases de los algoritmos. Todos los algoritmos que se quieran implementar deberán heredar de la clase base *SolutionMethod* que contiene

la información genérica a todo tipo de algoritmo, como puede ser el problema que resuelve, las mejores soluciones encontradas, la función de evaluación que utiliza para determinar si una solución es mejor que otra (normalmente utiliza la misma que la función objetivo del problema), criterio de parada e información adicional de datos estadísticos como la iteración en la que se encuentra o el tiempo que ha tardado la última ejecución.

El método principal de esta clase base es el método *run*, que ejecutará el algoritmo hasta alcanzar con el criterio de parada especificado (como puede ser un criterio de parada por iteraciones, o una parada en óptimo local, o si se ha explorado todo el entorno, etc.).

Como se puede ver en el diagrama, de la clase base heredan las dos clases principales que serán usadas como esqueletos para implementar y combinar todos los algoritmos de esta biblioteca, las clases *PopulationBasedMethod* y *TrajectoryBasedMethod*.

Diseño de los métodos poblacionales.

En la figura 4.5 se muestra el diseño de clases de los algoritmos poblacionales, así como algunos de los módulos implementados en la librería para los distintos operadores y métodos.

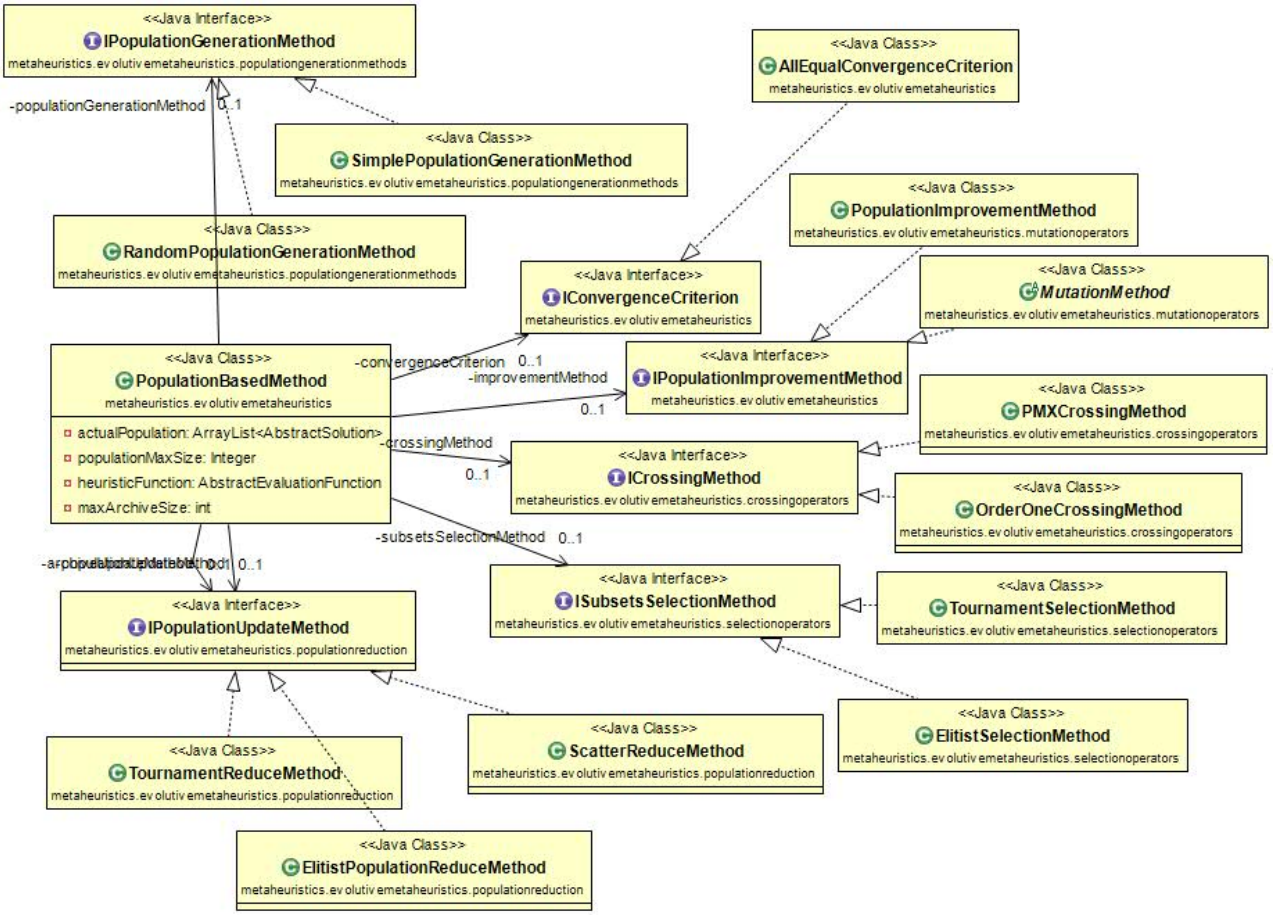

Figura 4.5: Diagrama de clases para los algoritmos poblacionales.

La clase *PopulationBasedMethod* pretende ser el esqueleto genérico para cualquier algoritmo basado en poblaciones; de modo que, combinando convenientemente los distintos módulos, se pueda obtener un simple algoritmo genético, un memético o incluso una búsqueda dispersa. Los módulos combinables de esta clase son los siguientes:

- **Método de generación de poblaciones.**: Es el modo en que se genera tanto la población inicial como las siguientes poblaciones a partir de un reinicio del algoritmo después de una convergencia.
- **Método de selección de subconjuntos**: Análogo al operador de selección en un algoritmo genético. Selecciona los subconjuntos a cruzar. Los subconjuntos pueden ser de cardinalidad superior a 2.
- **Método de cruce**: Modo en que se cruzan dos soluciones, desde simples cruces en un punto, hasta cruces *PMX* u *HUX* (estos métodos de explicarán en la siguiente sección).
- **Método de post-procesado**: Análogo al operador de mutación en algoritmos genéticos. Si está definido, se encarga de modificar las soluciones del conjunto generado por el operador de cruce.
- **Método de reducción**: Método que se encarga de reducir la población actual y la generada a una sola para dar lugar a la nueva generación.
- **Criterio de convergencia**: Criterio que determina si la población a llegado al punto de convergencia y se debe reiniciar siguiendo el método de generación de poblaciones.
- **Función heurística**: Función de *fitness*, puede ser distinta a la función objetivo del problema.

Diseño de los métodos basados en trayectorias.

En la figura 4.6 se muestra el diseño de la estructura de clases para la generación de algoritmos basados en trayectorias junto con algunos de los módulos que implementa este *framework*. Este tipo de algoritmo comprende a todos aquellos algoritmos que consisten en mejorar iterativamente una única solución hasta alcanzar algún criterio de parada. Partiendo de una solución inicial, permite iteración tras iteración generar nuevas soluciones a partir de la actual, mejorarla y decidir si aceptarla o no como nueva solución actual. Este esqueleto permite combinar sus distintos módulos para así obtener desde simples algoritmos *greedy*, hasta VNS, multi-arranques, algoritmos ILS o cualquier combinación que se pueda diseñar.

Los módulos que se pueden combinar dentro de esta clase son los siguientes:

- **Método de generación de soluciones**: Es el modo en que se obtiene una solución a partir de otra, considerada la "actual". Pueden ser tanto selecciones de una solución dentro de un entorno, generaciones aleatorias o incluso el resultado de aplicar algún algoritmo como un *greedy*.
- **Método de mejora**: Método que se aplica, si se ha definido uno, a la solución emergente del paso de generación de soluciones. Normalmente son otros algoritmos basados en trayectorias, como por ejemplo, búsquedas locales.
- **Criterio de aceptación**: Criterio que determina si aceptar una nueva solución emergente y considerarla la nueva actual o descartarla y generar una nueva. Pueden ser desde simples criterios como aceptar siempre o aceptar solo si mejora la actual, hasta criterios más complejos como pueden ser los criterios probabilistas o los que usan memoria como en las búsquedas tabú.
- **Historial**: Estructura que se utiliza para recordar las últimas soluciones encontradas por el algoritmo.

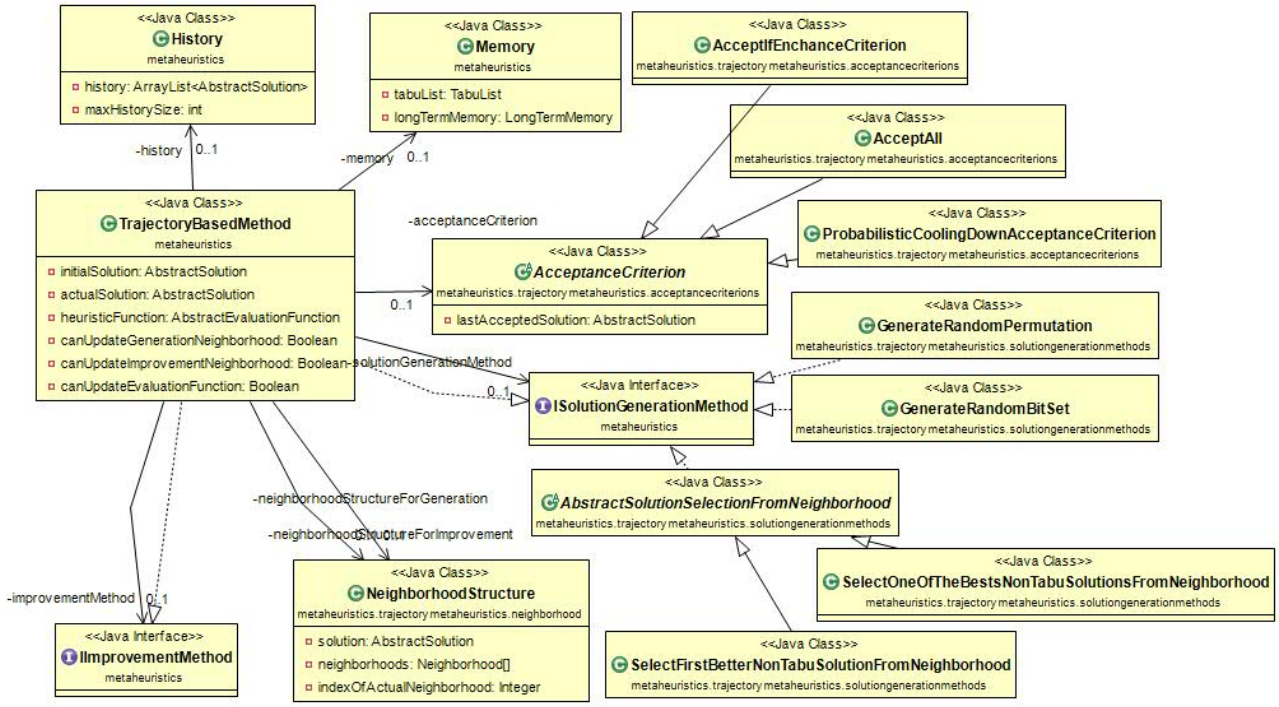

Figura 4.6: Diagrama de clases para los algoritmos basados en trayectorias.

- **Memoria**: Estructura de memoria utilizada en la generación de soluciones. Es muy dependiente de la representación de soluciones y lo que se quiera recordar de cada movimiento, por lo que existen diversas implementaciones.
- **Función heurística**: Función utilizada para comparar las soluciones. Puede ser una función dinámica si el *flag* de *canUpdateHeuristicFunction* está activado.
- **Estructuras de entorno**: Se diferencian dos tipos de estructuras de entorno: para la generación de soluciones y para la mejora. Esto permite mayor flexibilidad al poder usar estructuras distintas en cada paso. Si se desea utilizar estructuras dinámicas (como en el caso de los algoritmos VNS) se deberán activar los *flags* correspondientes de *canUpdateGenerationNeighborhood* o *canUpdateImprovementNeighborhood*

Diseño de las estructuras de entorno.

Una parte muy importante y posiblemente la más complicada de diseñar y generalizar en los algoritmos basados en trayectoria es la forma en que se representan los entornos. Existe una gran variedad de algoritmos que utilizan entornos (VNS, LS, FLS, etc.). Los entornos de cada uno de ellos tienen determinadas características y exigencias; por ejemplo, un VNS tiene que poder cambiar de un entorno a otro, mientras que un FLS tiene que ser capaz de dividir los entornos en sub-entornos, o un LS con muestreo ansioso deberá ser capaz de muestrear el entorno sin la necesidad de generarlo todo. Estos son requisitos ambiciosos y claves para nuestra biblioteca.

El diseño creado pretende cumplir simultáneamente con los siguientes objetivos:

- Permitir cambiar de una estructura de entorno a otra.

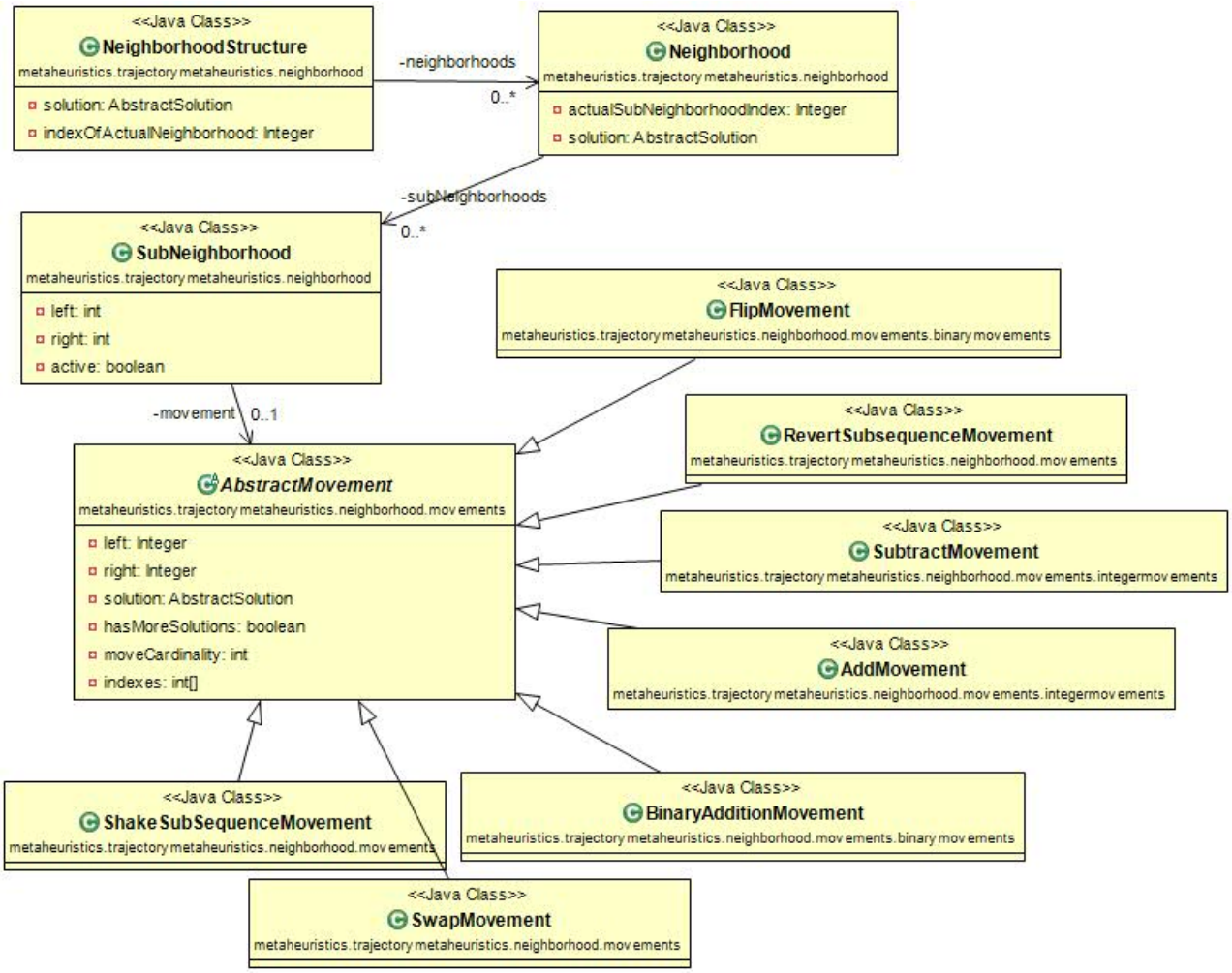

Figura 4.7: Diagrama de clases para las estructuras de entorno.

- Poder muestrear el entorno solución a solución, es decir, sin la necesidad de generarlo todo.
- Poder dividir cada entorno en sub-entornos.
- Poder determinar si una solución pertenece o no a un sub-entorno de forma rápida y eficiente.
- Permitir extender los tipos de entorno para una posible ampliación de la biblioteca.

Para lograr estos objetivos, en primer lugar se tiene la clase *NeighborhoodStructure*, que no es más que una lista de entornos que provee la interfaz de cambiar de uno a otro, cumpliendo así con el primer objetivo.

Cada entorno está formado por una serie de sub-entornos. Cada sub-entorno podrá aplicar un movimiento en una determinada región de la solución. De este modo se tiene la posibilidad de dividir un entorno en varios sub-entornos que pueden actuar en distintas partes de la solución, pudiendo así tener combinaciones de varios sub-entornos interceptados entre sí, e incluso varios sub-entornos con distintos movimientos dentro de un mismo entorno. Los movimientos son la operación que se ha de aplicar a una determinada solución para generar una solución vecina. Cada movimiento deberá poder generar soluciones vecinas a partir de una solución dada, deberá poder generar soluciones diferentes una a una hasta haber generado todas las soluciones

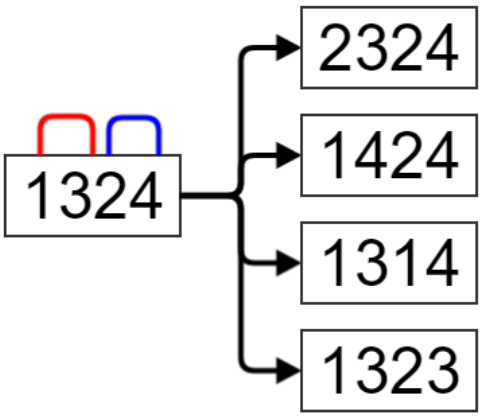

Figura 4.8: Ejemplo de un entorno sobre una solución de *array* de enteros.

distintas que podía generar, y además, deberá poder determinar si, dadas dos soluciones, este movimiento hubiera podido generar una solución a partir de otra. De este modo se cumplen los últimos dos objetivos propuestos, muestrear los entornos de solución en solución, y determinar si una solución pertenece a un sub-entorno o no.

En la figura 4.8 se muestra un ejemplo de un entorno formado por dos sub-entornos. El primero, un movimiento que suma 1 a los elemento que actúa en los índices 1 y 2 de la solución, mientras que el segundo es un movimiento que resta 1 a los elementos que actúa sobre los índices 3 y 4.

4.2. Módulos implementados.

Como se ha dicho anteriormente, una de las características fundamentales de esta biblioteca es la capacidad que tiene para generar nuevos algoritmos a partir de la combinación de los distintos módulos implementados. En esta sección se presentarán los distintos módulos ya implementados para cada tipo de búsqueda (sea basa en poblaciones o trayectorias) y una breve descripción de cada uno de ellos.

4.2.1. Módulos para algoritmos basados en poblaciones.

Los algoritmos poblacionales pueden ser generados mediante las distintas combinaciones de los métodos de generación de poblaciones, selección, cruce, post-procesado, reducción y criterios de convergencia. A continuación se presentarán los distintos métodos implementados en esta biblioteca.

Métodos de generación de poblaciones.

- **Generación aleatoria.** Genera aletoriamente una población de individuos.
- **Generación dispersa.** Genera de forma procedural una población dispersa de individuos.

- **Generación mejorada.** Genera de forma aleatoria una población de individuos y luego los mejora cada uno de ellos hasta alcanzar un óptimo local.

Métodos de selección.

- **Selección por ruleta sesgada.** Selecciona las parejas de padres a cruzar de forma probabílistica, dando mayor probabilidad de selección a los individuos mejor adaptados.
- **Selección elitista.** Elige los padres de forma determinista, seleccionando los mejores individuos.
- **Selección por torneo.** Selecciona los padres a cruzar haciendo competir a los individuos entre sí mediante una serie de torneos, eligiendo como candidato a la reprodución al ganador de cada torneo. Los torneo pueden ser de tamaño mayor a 2 y no tienen por qué ser deterministas.
- **Selección de todas las combinaciones.** Selecciona todas las combinaciones de 2 individuos.

Métodos de cruce.

- **Cruce en un punto.** Método básico de cruce, Una vez seleccionados dos individuos se cortan sus cromosomas por un punto seleccionado aleatoriamente para generar dos segmentos diferenciados en cada uno de ellos: la cabeza y la cola. Se intercambian las colas entre los dos individuos para generar los nuevos descendientes. De esta manera ambos descendientes heredan información genética de los padres
- **Cruce en dos puntos.** Se trata de una generalización del cruce de 1 punto. En vez de cortar por un único punto los cromosomas de los padres como en el caso anterior se realizan dos cortes. Deberá tenerse en cuenta que ninguno de estos puntos de corte coincida con el extremo de los cromosomas para garantizar que se originen tres segmentos. Para generar la descendencia se escoge el segmento central de uno de los padres y los segmentos laterales del otro padre.
- **Cruce por máscara.** Se genera una máscara de bits que indicará de qué padre heredará la información cada hijo.
- **HUX.** En el esquema de recombinación uniforme media (HUX del inglés *Half Uniform Crossover*), exactamente la mitad de los bits que son diferentes se intercambian. Por esto se necesario calcular la distancia de *Hamming* (número de bits diferentes). Este número se divide entre dos, y el número resultante es la cantidad de bits diferentes que tiene que ser intercambiada entre los parentales.
- **PMX.** *Partial Mapped Crossover*. Método de cruce para codificaciones que representan permutaciones; trata de combinar ambos padres manteniendo la información de orden de una determinada sub-secuencia de cada padre.

Post-procesado.

- **Mutación.** Modifica cada individuo en base a una determinada probilidad.
- **Mejora de soluciones.** Mejora cada individuo hasta alcanzar un óptimo local. Se puede elegir cualquier algoritmo basado en trayectorias como método de mejora.

Métodos de reducción.

- **Reducción a la nueva generación.** Descarta a los padres y permanecen en la nueva generación unicamente los hijos.
- **Reducción elitista.** Permanecen en la nueva generación los mejores individuos.
- **Reducción por torneo.** Realiza una selección por torneo para decidir qué individuo formará parte de la nueva generación.
- **Reducción por ruleta sesgada.** Realiza una selección ruleta sesgada para decidir qué individuo formará parte de la nueva generación.
- **Reducción dispersa.** La nueva generación estará formada por un lado por los mejores individuos, y por otro por los más dispersos.

Criterios de convergencia.

- **Convergencia si todos los individuos son iguales.** Se considerará que se ha llegado a un punto de convergencia si todos los individuos son iguales.
- **Convergencia si la calidad de las soluciones es la misma.** En el momento que todos los individuos tengan igual nivel de adaptación, la población se considerará que ha convergido.
- **Convergencia si no se encuentran mejores individuos.** Se considera que la población converge si durante un determinado número de generaciones no se actualiza el archivo con los mejores individuos encontrados.

4.2.2. Módulos para algoritmos basados en trayectorias.

Los algoritmos basados en trayectorias pueden ser generados mediante las distintas combinaciones de los métodos de generación de soluciones, mejora de la misma, criterio de aceptación, estructura de entorno y estructuras de memoria e historial. A continuación se presentarán los distintos métodos implementados en esta biblioteca.

Métodos de generación de soluciones.

Son todos aquellos métodos capaces de generar una solución, ya sea seleccionándola a partir de otra solución y un entorno, generándola aleatoriamente, como resultado de un método más complejo de generación como puede ser mismo un algoritmo constructivo como un *greedy*, etc.

- **Generación aleatoria.** Genera una solución aleatoria, ya sea una ristra aleatoria de bits, una permutación aleatoria o un *array* de enteros aleatorio.
- **Generación de ristras de bits dispersa.** Genera de forma dispersa una ristra de bits.
- **Selección aleatoria de un entorno.** Elije de forma aleatoria una solución a partir de otra y un entorno.
- **Selección ansiosa.** Elije la primera solución del entorno que mejora la solución actual.

- **Selección ansiosa con memoria.** Elije la primera solución no *tabú* del entorno de la solución actual.
- **Selección *greedy*.** Elije la mejor solución del entorno de la solución actual.
- **Selección *greedy* con memoria.** Elije la mejor solución no *tabú* del entorno de la solución actual.
- **Selección *GRASP*.** Elije aleatoriamente una de las mejores soluciones del entorno de la solución actual.
- **Selección *GRASP* con memoria.** Elije aleatoriamente una de las mejores soluciones no *tabú* del entorno de la solución actual.
- **Selección por sub-entornos.** Aplica una selección ansiosa sobre los sub-entornos activos y reactivando los que se deban activar según la solución seleccionada.
- **Selección por sub-entornos con memoria.** Aplica una selección ansiosa con memoria sobre los sub-entornos activos y reactivando los que se deban activar según la solución seleccionada.
- **Perturbación de soluciones.** Dada la solución actual y un historial, aplica una perturbación para modificar la solución actual y así generar una nueva solución.
- **Generación a partir de un método basado en trayectorias.** Genera una nueva solución utilizando un algoritmo basado en trayectorias (*greedy* por ejemplo).

Para todos los métodos de selección que usen memoria se deberá especificar qué módulo de memoria ha de utilizar.

Métodos de mejora.

Los métodos de mejora que podrán utilizar los algoritmos basados en trayectorias son otros algoritmos basados en trayectorias que se dedican a mejorar una solución partiendo de una solución, entorno y función heurística dada. Algunos de los métodos pre-implementados en esta librería son los siguientes.

- **Búsqueda local.**
- **Búsqueda local rápida.**
- **Búsqueda guiada.**
- **Búsqueda VNS.**
- **VND.**
- **RVNS.**
- **Búsqueda tabú.**
- **Búsqueda local iterativa.**

Criterios de aceptación.

Los criterios de aceptación son los métodos que determinan si, dada una nueva solución emergente, el algoritmo puede pasar a considerar como actual la nueva solución, o descartarla y permanecer con la que tenía antes. Los criterios de aceptación implementados son los siguientes.

- **Acepta todo.** Acepta todas las soluciones emergentes.
- **Acepta solo si mejora.** Aceptará la solución solo si mejora la actual (aceptación usada en una búsqueda local).
- **Aceptación probabilista .** Aceptará la solución siempre que la mejore, o bien, de forma probabilista en función del grado de empeoramiento respecto la actual. Esta aceptación es la utilizada en un recocido simulado.

Tipos de entorno.

Los tipos de entorno vienen definidos principalmente por los movimientos que realizan.

Los movimientos definidos tanto para soluciones enteras como binarias son:

- **Revertir sub-secuencia.** Dados dos puntos, revierte el orden de la sub-secuencia.
- **"Agitar" sub-secuencia.** Dados dos puntos, aleatoriza dicha ristra.
- **Intercambio.** Dados dos puntos, intercambia el valor de ambos.

Los movimientos definidos únicamente para soluciones binarias son:

- **Añadir un elemento.** Pone a 1 un bit que esté a 0.
- **Quitar un elemento.** Pone a 0 un bit que esté a 1.
- **Invertir un elemento.**Pone a 0 un bit que esté a 1, o a 1 un bit que esté a 0.
- **Intercambio binario.** Intercambia un bit que esté a 0 por otro que esté a 1 o vice-versa.

Los movimientos definidos únicamente para soluciones enteras son:

- **Sumar a un elemento.** Suma una cierta cantidad a un elemento.
- **Restar a un elemento.** Resta una cierta cantidad a un elemento.
- **Invertir un elemento.**Pone a 0 un bit que esté a 1, o a 1 un bit que esté a 0.
- **Modificar un elemento.** Modifica el valor de un elemento a algún otro dentro del rango permitido por las soluciones.

4.3. Ejemplo de uso de la librería.

En la figura 4.9 se muestra el código de ejemplo de un algoritmo multi arranque para el problema del tsp. Este algoritmo en concreto generará en cada iteración una solución aleatoria y luego la mejorará con una búsqueda local utilizando movimientos 2-opt.

En la figura 4.10 se muestra el código de ejemplo de un algoritmo híbrido para el problema del tsp. Este algoritmo generará en cada iteración una solución nueva aplicando una perturbación a la actual, luego la mejorará con un VNS, y aceptará las soluciones siguiendo un criterio probabilista que utiliza como función de probabilidad una exponencial.

```
6  public class Example {
7      public static void main(String[] args) {
8
9          SolutionMethod localSearch;
10         SolutionMethod multiStart;
11         TSProblem problem = new TSProblem("XQF131.tsp");
12
13         localSearch = AlgorithmMaker.createLocalSearch()
14                     .forProblem(problem)
15                     .withMaxIterations(500)
16                     .withNeighborhood("2-Opt")
17                     .whithSelection("Greedy")
18                     .create();
19
20         multiStart = AlgorithmMaker.createMultiStart()
21                     .forProblem(problem)
22                     .withMaxIterations(500)
23                     .withGeneration("Random")
24                     .withImprovementMethod(localSearch)
25                     .create();
26
27         multiStart.runSearch();
28     }
29 }
30
```

Figura 4.9: Ejemplo de la creación de un multi arranque para el TSP.

Por último, en la figura 4.11 se muestra un ejemplo de como crear un algoritmo evolutivo para el problema del viajante de comercio. Este algoritmo utiliza una población de 100 individuos generada aleatoriamente, un operador de selección por torneo binario, cruce PMX, un post-procesado que aplica una búsqueda local a cada individuo con probabilidad 0.1, y, un método de reducción elitista.

```java
public class Example {
    public static void main(String[] args) {

        SolutionMethod vns;
        SolutionMethod hybrid;
        TSProblem problem = new TSProblem("XQF131.tsp");

        vns = AlgorithmMaker.createVNS()
                .forProblem(problem)
                .withMaxIterations(500)
                .withLocalSearch("Greedy")
                .addNeighborhood("Swap")
                .addNeighborhood("2-Opt")
                .create();

        hybrid = AlgorithmMaker.createHybridAlgorithm()
                .forProblem(problem)
                .withMaxIterations(600)
                .withHistorySize(10)
                .whithSelection("Random perturbation")
                .withImprovementMethod(vns)
                .withAcceptanceCriterion("Probabilistic")
                .withAcceptanceFunction("exp")
                .create();

        hybrid.runSearch();

    }
}
```

Figura 4.10: Ejemplo de la creación de un algoritmo híbrido para el TSP.

```java
public class Example {
    public static void main(String[] args) {

        SolutionMethod localSearch;
        SolutionMethod ga;
        TSProblem problem = new TSProblem("XQF131.tsp");

        localSearch = AlgorithmMaker.createLocalSearch()
                .forProblem(problem)
                .withMaxIterations(500)
                .withNeighborhood("2-Opt")
                .withSelectionOperator("Greedy")
                .create();

        ga = AlgorithmMaker.createGeneticAlgorithm()
                .forProblem(problem)
                .withMaxIterations(500)
                .withPopulationSize(100)
                .withPopulationGeneration("Random")
                .withSelectionOperator("BinaryTournament")
                .withCrossingOperator("PMX")
                .withImprovementMethod(localSearch)
                .withChanceOfPostProcessing(0.1)
                .withReductionMethod("Elitist")
                .create();

        ga.runSearch();

    }
}
```

Figura 4.11: Ejemplo de la creación de un algoritmo genético para el TSP.

Capítulo 5

Resultados.

5.1. Resultados.

En este capitulo se mostrarán algunos de los resultados obtenidos. Las pruebas se han realizado con un ordenador con las siguientes características:

- Procesadore Intel Core i5-3330 6Mb de Cache, 3.2 GHz, 4 núcleos.
- 4 GB de RAM.
- GPU Nvidia Geforce GT640.

Las pruebas se han realizado sobre el problema del viajante de comercio con las instancias proporcionadas por la TSPLIB. En las siguientes tablas se muestran los valores objetivo obtenidos para los distintos problemas usando diferentes algoritmos.

Problema	**Nodos**	**Greedy**	**GRASP1**	**GRASP2**	**Exacto**
Att48	48	12.842	10.713	10.785	10.628
Ch130	130	7.575	6.543	6.388	6.110
U159	159	56.369	45.709	45.672	42.080

Tabla 5.1: Tabla resultados 1.

En la tabla 5.1 los algoritmos que se han utilizado son:

- *Greedy*: Algoritmo de vecinos más cercano.
- GRASP1: Algoritmo vecino más cercano con lrc de tamaño 2 con una búsqueda local *greedy*.
- GRASP2: Algoritmo vecino más cercano con lrc de tamaño 2 con una búsqueda local ansiosa.

En la tabla 5.2 se muestran los resultados de ejecutar un algoritmo genético con distintos parámetros para el problema "att48". Se puede concluir que una buena combinación de parámetros puede ser usar torneo binario, mutación con probabilidad no superior al 10 % y una reducción elitista. Además, como era de esperar, a mayor tamaño de población, en general, se obtiene una solución de mejor calidad.

Población	**Selección**	**Cruce**	**Mutación**	**Reducción**	**Valor**
100	Torneo binario	PMX	0.1	Nueva generación.	13.794
500	Torneo binario	PMX	0.1	Nueva generación.	12.465
1000	Torneo binario	PMX	0.1	Nueva generación.	11.659
100	Torneo binario	PMX	0.1	Torneo binario	15.876
100	Torneo binario	PMX	0.1	Elitista	14.830
500	Torneo binario	PMX	0.1	Elitista	12.420
500	Torneo binario	Order one	0.2	Elitista	12.497
500	Torneo ternario	Order one	0.2	Elitista	12.349
1000	Torneo ternario	Order one	0.2	Elitista	12.044

Tabla 5.2: Tabla resultados 2.

Capítulo 6

Conclusiones y líneas futuras

6.1. Conclusiones.

Tras haber realizado este trabajo, se puede apreciar la importancia de las metaheurísticas en la resolución de problemas reales. Dada esta importancia, se puede concluir que la existencia de una librería que de facilidades para diseñar, implementar y testear diversos métodos resulta interesante y útil.

Desde un principió se decidió implementar este *software* en Java, gracias a la flexibilidad del lenguaje y la capacidad de crear *software* de alta calidad de forma más rápida que utilizando otros lenguajes, más eficientes como C++. Si bien se puede criticar la pérdida de eficiencia respecto a una implementación en C++ por ejemplo, esta decisión de usar Java se defiende argumentando que esta biblioteca está enfocada a prototipar, diseñar y testear diversas metaheuristicas y configuraciones, para luego, una vez encontrado el método y configuración deseada, se pueda implementar de forma especifica según el contexto; agilizando, de ese modo, la fase de búsqueda del método idóneo para un determinado problema específico.

La razón por la que se pretende que esta biblioteca se utilice para prototipar y diseñar más que para resolver problemas reales es intrínseca a su diseño; se ha preparado para ser genérica, parametrizable, extensible, para cualquier algoritmo y problema. Estas propiedades tienen un precio, y se paga en eficiencia; por ganar generalidad y tiempo de implementación muchas veces se pierde parte de la eficiencia que se podría tener con un método implementado *ad hoc*.

6.2. Líneas futuras.

Como líneas futuras de trabajo, se pueden proponer las siguientes:

- Una mejor comprobación de cada uno de los métodos y módulos.
- Implementación de una interfaz gráfica de usuario.
- Añadir una *suite* de métodos estadísticos para el análisis de rendimiento de algoritmos, como por ejemplo, el test de *Wilkinson*.
- Extensión al caso multi-objetivo.
- Incorporación de nuevos problemas a la biblioteca de problemas.

- Nuevas formas de codificar las soluciones.
- Rediseño del código.

Capítulo 7

Summary and Conclusions

7.1. Conclusions.

Having done this work , we can appreciate the importance of metaheuristics for solving real problems. Given this importance, it can be concluded that the existence of a library which facilitate the design, implementation and testing of the various methods is interesting and useful.

From the begining it was decided to implement this software in Java, thanks to the flexibility of language and the capability to create high quality software way faster than using other languages, such as C++ . While we can criticize the loss of efficiency with respect to an implementation in C++ for example, we can defend the decision we made arguing that this library is focused to prototye, design and test instead of solving real world problems.

The reason why this library is intended to be used for prototype and design rather than to solve real problems is intrinsic to its design; It has been prepared to be generic for any algorithm and problem. These properties are not free, and the price is paid in efficiency. So, an algorithm made using this library will be probably slower than an algorithm made to solve a specific problem under specific circumstances.

7.2. Future work.

As future work lines , we can propose the following :

- A better testing phase.
- A graphic user interface.
- Add a some statistical methods to compare algorithms.
- Extension to multi-objective.
- Add more problems to the library.
- New ways to codificate solutions.
- Re-design of the implementation.

Capítulo 8

Presupuesto

Descripción	**Precio**
280 horas de trabajo.	14.000 €
Ordenador de computación	2.000 €
Total	16.000 €

Tabla 8.1: Presupuesto

Bibliografía

[1] Harald Dyckhoff. A typology of cutting and packing problems. *ELSEVIER*, 44:145–159, 1990.

[2] Lothar Thiele Eckart Zitzler, Kalyanmoy Deb. Comparison of multiobjective evolutionary algorithms: Empirical results. *Evolutionary computation.*, 8(2):173–195, 2000.

[3] Lothar Thiele Eckart Zitzler, Marco Laumanns. Spea2: Improving the strenght pareto evolutionary algorithm. Technical report, 2001.

[4] Gary A. Kochenberger Fred Glover. *Handbook of metaheuristics.* 2003.

[5] Antonio J.Nebro Juan J.Durillo. jmetal: A java framework for multi-objective optimization. *ELSEVIER*, 42:760–771, 2011.

[6] Marco Laumanns Eckart Zitzler Kalyanmoy Deb, Lothar Thiele. Scalable test problems for evolutionary multi-objective optimization. Technical report, july 2001.

[7] Sameer Agarwal Kalyanmoy Deb, Amrit Pratap. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on evolutionary computation.*, 6(2):182–197, april 2002.

[8] Harold W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.

[9] Rafael Martí. Procedimientos Metaheurísticos en Optimización Combinatoria. Technical report, 2001.

[10] Mehmet Cunkas Tahir Sag. A tool for multiobjective evolutionary algorithms. *ELSEVIER*, 40:902–912, 2009.

[11] William R.Pulleyblank William J.Cook, William H. Cunningham and Alexander Schrijver. *Combinatorial Optimization.* 1997.