Algoritmos para la resolución del problema combinatorio de empaquetamiento

Algorithms for the resolution of Bin Packing Problem

Trabajo Fin de Grado en Matemáticas
Universidad de Málaga

Autor: Rafael Requena Garrido

Área de conocimiento y/o departamento: Lenguajes y Sistemas Informáticos

Fecha de presentación: Septiembre de 2021

Tema: Resolución de problemas de optimización combinatoria

Tipo: (trabajo de revisión bibliográfica, de iniciación a la investigación,...)

Modalidad: Individual

Número de páginas (sin incluir introducción, bibliografía ni anexos):

DECLARACIÓN DE ORIGINALIDAD DEL TFG

D./Dña. *Rafael Requena Garrido*, con DNI (NIE o pasaporte) *76437428X*, estudiante del Grado en *Matemáticas* de la Facultad de Ciencias de la Universidad de Málaga,
DECLARO:

Que he realizado el Trabajo Fin de Grado titulado "*Algoritmos para la resolución del problema combinatorio de empaquetamiento*" y que lo presento para su evaluación. Dicho trabajo es original y todas las fuentes bibliográficas utilizadas para su realización han sido debidamente citadas en el mismo.

De no cumplir con este compromiso, soy consciente de que, de acuerdo con la normativa reguladora de los procesos de evaluación de los aprendizajes del estudiantado de la Universidad de Málaga de 23 de julio de 2019, esto podrá conllevar la calificación de suspenso en la asignatura, sin perjuicio de las responsabilidades disciplinarias en las que pudiera incurrir en caso de plagio.

Para que así conste, firmo la presente en Málaga, el *9 de septiembre de 2021*

Fdo:.....

Índice general

El Título aquí

Resumen

Texto.

Palabras clave:

PONER AQUÍ LAS PALABRAS CLAVE.

El Título (en inglés) aquí

Abstract

Text.

key words:

KEY WORDS.

Capítulo 1

Optimizaiøn combinatoria

1. Conceptos generales

Para comprender con mayor claridad el objetivo de la optimización y los elementos que juegan un papel clave en la misma, procederemos a introducirla mediante un ejemplo:

Supongamos que una compañía fabrica y vende dos modelos de mesas, M_1 y M_2. Para su fabricación, es necesario un trabajo manual de 20 y 30 minutos para los modelos M_1 y M_2, respectivamente, más un trabajo de máquina de 20 minutos para el modelo M_1 y de 10 minutos para el modelo M_2. Para el trabajo manual se dispone de 100 horas al mes, mientras que, para el de máquina, 80 horas. Sabiendo que el beneficio por unidad es de 150 y 100 euros para M_1 y M_2, respectivamente, queremos planificar la producción de manera que obtengamos el beneficio máximo.

Si llamamos x al número de mesas M_1 e y al número de mesas de M_2, podemos definir la función beneficio $f(x, y) = 150x + 100y$. Por otro lado, pasando el tiempo a horas, las condiciones dadas en el enunciado se traducen a:

$$\frac{1}{3}x + \frac{1}{2}y \leq 100$$
$$\frac{1}{3}x + \frac{1}{6}y \leq 80$$

Si juntamos todo para escribirlo como es habitual en los textos de optimización, tenemos que nuestro problema se modela como sigue:

$$f(x, y) = 150x + 100y$$
$$s.a\ \frac{1}{3}x + \frac{1}{2}y \leq 100$$
$$\frac{1}{3}x + \frac{1}{6}y \leq 80$$
$$x \geq 0, y \geq 0, x, y \in \mathbb{Z}$$

De este modo, nuestro problema se reduce a encontrar un par (x, y) que maximice a la función f y verifique las restricciones anteriores.

Formalmente, un problema de optimización se puede describir (**citar tesis Pepe**) como una tupla (D, X, f, R) donde:

1. $D = \{D_1, ..., D_n\}$ es un conjunto de dominios.
2. $X = \{x_1, ..., x_n\}$ es un conjunto de variables tal que para cada $i \in \{1, ..., n\}, x_i \in D_i$.
3. $f : D_1 \times ... \times D_n \longrightarrow \mathbb{R}^+$ se denomina *función objetivo*, para la cual estaremos interesados en conocer sus máximos o mínimos.
4. R es un conjunto de restricciones sobre las variables.

Por otro lado, definimos el conjunto $S \subseteq D_1 \times ... \times D_n$ en el que se verifican las restricciones de R como el *espacio de búsqueda* del problema y, a cada $s \in S$, una *solución posible (o factible)* del problema. Así, una solución del problema es un elemento $s^* \in S$ tal que $f(s^*) \geq f(s), \forall s \in S$. A s^* se le denomina *óptimo global* del problema. Usualmente se sigue la nomenclatura anterior para los problemas de optimización en los que maximizamos la función f, mientras que cuando lo que buscamos es minimizarla, nos referimos a ella como *función de costos*.

Existen diversas formas de clasificar a los problemas de optimización. Como no existe un método único para resolver todos los problemas posibles, es importante analizar en qué categoría entra el problema en cuestión ya que, de esta manera, podremos emplear algoritmos que se ajusten mejor al mismo, ya sea reduciendo el tiempo de cálculo y/o hallando soluciones más aproximadas o exactas, por ejemplo.

Así, podemos realizar una primera clasificación atendiendo a la continuidad o no de las variables. En el caso más general, diremos que estamos frente a un problema de *Optimización Continua* cuando todas las variables del problema sean de tipo continuo. Dentro de este tipo de problemas, cobran especial importancia los problemas de *Optimización Convexa*, en los cuales tenemos que minimizar (en general) una *función convexa* (usalmente llamada *función de costos*) sujeta a un conjunto solución convexo. Cuando la función objetivo y las restricciones son lineales, decimos que estamos frente a un problema de *Optimización Convexa Lineal* o *Programación Lineal*, mientras que cuando no lo son, decimos que el problema es de *Programación no Lineal*.

En cambio, si las variables son de tipo discreto, es decir, solo pueden tomar valores enteros, decimos que el problema es de *Optimización Combinatoria*. Finalmente, decimos que un problema es de *Optimización Mixta* cuando tiene algunas variables de tipo continuo y otras de tipo discreto.

En cuanto a los distintos métodos de resolución de problemas de optimización, aquí también encontramos difersas formas de clasificarlos:

- Resolución mediante cálculo,
- Resolución mediante técnicas de búsqueda.
- Resolución mediante técnicas de convergencia de soluciones

Los métodos de resolución por cálculo hacen uso del cálculo de derivadas para determinar qué valores del dominio de la función presentan máximos y mínimos. Son métodos muy potentes, pero requieren mucha capacidad de cómputo y que la función objetivo y las

restricciones cumplan una serie de condiciones (condiciones de continuidad, derivabilidad, etc.). En la práctica estos métodos no suelen utilizarse, ya que los problemas no suelen cumplir las condiciones necesarias para la aplicación de estos métodos y tienen demasiada variables como para que sean eficientes. Un ejemplo clásico de estos métodos, es el método de los multiplicadores de Lagrange.

En los métodos de resolución mediante técnicas de búsqueda, podemos encontrar desde métodos exactos como el tradicional algoritmo del símplex (para problemas de Programación Lineal) y sus variantes hasta técnicas metaheurísticas como la *búsqueda tabú* o el *recocido simulado (simulated annealing)*, también conocido como algoritmo de cristalización simulada.

Por otro lado, la mayoría de técnicas de convergencia de soluciones son de tipo metaheurístico. Se basan en generar gran cantidad de soluciones, determinar cuáles son las mejores y, a partir de ellas, generar otro conjunto de soluciones a analizar, repitiendo el proceso hasta que estas soluciones que vamos generando converjan a una. Por lo tanto, dentro de este grupo podemos encontrar técnicas de tipo iterativo, como el clásico método de Newton o el método de descenso del gradiente, y hasta los *algoritmos genéticos* (de tipo metaheurístico), que se enmarcan dentro de los *algoritmos evolutivos*, siendo estos dos últimos muy empleados en el área de la inteligencia artificial.

Hecho este breve contexto, estamos en disposición de centrarnos en el caso que nos atañe. La optmización combinatoria es una rama de la optimización relacionada con la investigación operativa, la teoría algorítmica y la teoría de la complejidad computacional (**citar wiki/artículo jgarcia**). Observemos que, para un problema de optimización combinatoria $P = (D, X, f, R)$, el conjunto $S \subseteq D$ de las posibles soluciones de P es finito, lo cual nos lleva a encontrar un método simple con el que hallar el óptimo global de P, que consiste en examinar todas las posibles soluciones del problema. Sin embargo, en la práctica, en la mayoría de problemas de optimización combinatoria no es posible la aplicación de este método debido a que el espacio de búsqueda crece exponencialmente con el tamaño del problema (o tamaño de la instancia del problema), lo cual hace que el coste computacional y el tiempo necesario para examinar cada una de las posibles soluciones sea inviable.

El *tamaño de una instancia* (**citar Brassard**) se corresponde formalmente al número de bits necesarios para representar la instancia en un ordenador, utilizando algún esquema de codificación definido con precisión y razonablemente compacto. No obstante, para que los análisis sean más claros, normalmente emplearemos la palabra "tamaño" para referirnos a cualquier número entero que mida de algún modo el número de componentes de una instancia. Por ejemplo, cuando hablamos de ordenar, usualmente medimos el tamaño de la instancia por el número de elementos a ordenar, independientemente de que dichos elementos necesiten más de un bit para ser representados en un ordenador.

Esta necesidad de resolver instancias de problemas cada vez más grandes con un coste computacional aceptable nos lleva a desarrollar algoritmos que, aunque no nos proporcionan soluciones exactas, si nos permiten obtener soluciones aproximadas razonablemente buenas. Son los ya mencionados algoritmos heurísticos.

2. Complejidad. Problemas P vs NP

Uno de los objetivos de este documento es analizar la eficiencia de una serie de algoritmos en la resolución de un problema de optimización combinatoria. Para ello, resulta necesario la introducción de unos cuantos conceptos con los que analizarlos.

La teoría de la complejidad computacional (o teoría de la complejidad informática) es una rama de la teoría de la computación que trata de clasificar los problemas computacionales en base a si pueden ser o no resueltos con una cantidad determinada de tiempo y memoria, lo que suele denominarse como su *dificultad inherente*.

A continuación presentamos la notación O grande, también conocida como notación de *Landau*, usada frecuentemente para clasificar funciones en base a su velocidad de crecimiento, es decir, su orden de magnitud. Dado que se basa en su comportamiento en casos límite, define lo que se denomina *coste asíntotico* de los algoritmos.

Sean dos funciones $f, g : \mathbb{R} \longrightarrow \mathbb{R}$. Decimos que $f(n)$ es $O(g(n))$ (o que $f(n) = O(g(n))$) si y solo si existen $n_0, c > 0$ tales que $|f(n)| \leq c|g(n)|$ para todo $n > n_0$.

La notación $O(f)$ tiene las siguientes propiedades (**citar webdiis.unizar**), cualesquiera que sean las funciones f, g y h:

1. Para todo $c \in \mathbb{R}^+$,
$$f(n) = O(g(n)) \Longleftrightarrow c \cdot f(n) = O(g(n))$$
2. Si $f(n) = O(g(n))$ y $g(n) = O(h(n))$ entonces $f(n) = O(h(n))$.
3. $O(f+g) = O(max(f, g))$. Se demuestra fácilmente haciendo uso de las desigualdades
$$\left.\begin{array}{l} f \leq max(f,g) \\ g \leq max(f,g) \end{array}\right\} \rightarrow f + g \leq max(f,g) + max(f,g) \leq 2\, max(f,g)$$
Frecuentemente esta propiedad se aplica así: si $f_1 = O(g_1)$ y $f_2 = O(g_2)$, entonces $f_1 + f_2 = O(max(g_1, g_2))$.
4. Si $f_1 = O(g_1)$ y $f_2 = O(g_2)$, entonces $f_1 \cdot f_2 = O(g_1 \cdot g_2)$.
5. Para todo $c \in \mathbb{R}^+$,
$$f = O(g) \Longleftrightarrow c + f = O(g)$$
Es consecuencia inmediata de la regla de la suma.

De este modo, tenemos la siguiente jerarquía para las formas de crecimiento asintótico más importantes:

$$O(1) \subset O(log\, n) \subset O(n) \subset O(n\, log\, n) \subset O(n^2) \subset O(n^3) \subset O(2^n)$$

¿Comentar aquí algo acerca de que hay que tener cuidado a la hora de comparar algoritmos en base a su coste asintótico? Por ejemplo con los algoritmos

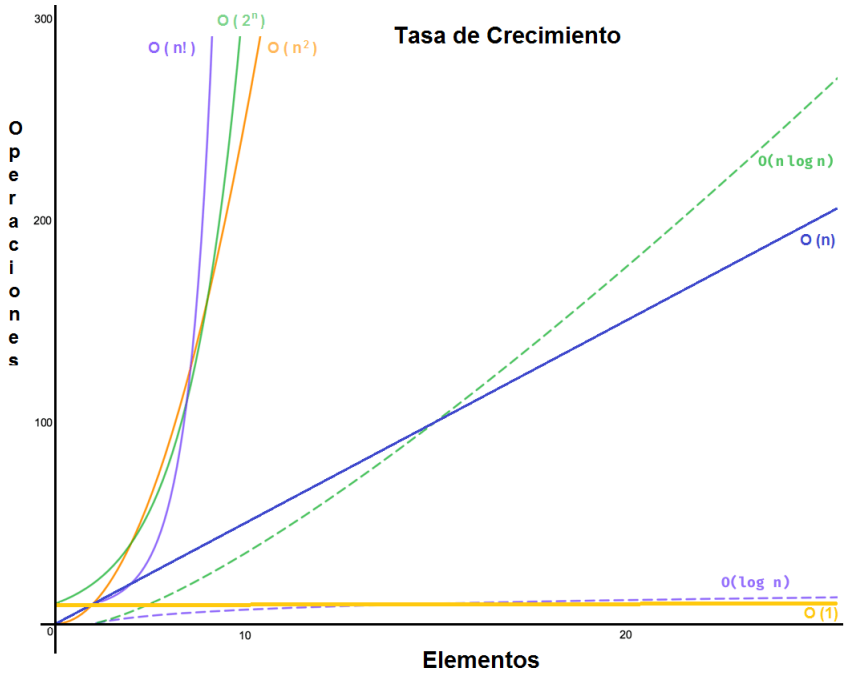

Figura 1.1: Formas de crecimiento asintótico más importantes

de Karatsuba, Toom-Cook y Schönhage-Strassen para el producto.

Sin embargo, ¿qué significa que un algoritmo sea eficiente? ¿Significa que toma un tiempo en $O(n \log n)$? ¿$O(n^2)$? Dependerá del problema a resolver.

Decimos que un algoritmo es eficiente (**citar Brassard**) si existe un polinomio $p(n)$ tal que el algoritmo puede resolver cualquier instancia del problema de tamaño n en un tiempo $O(p(n))$. Se dice entonces que el algoritmo es de *tiempo polinómico* y que los problemas que se resuelven con dicho algoritmo son resolubles en tiempo polinómico. Sin embargo, cuando el tiempo de ejecución de un algoritmo no se puede acotar mediante una fórmula polinómica, se dice que dicho algoritmo y su problema asociado son de *tiempo exponencial*. Cuando solo se conocen algoritmos de tiempos exponenciales para resolver un problema, se dice que el problema es *intratable*.

Para lo que sigue, nos centraremos en el caso de los problemas de decisión, que son aquellos problemas que tienen como respuesta sí o no, o equivalentemente, verdadero o falso.

Todas estas consideraciones anteriores nos sirven de base para la introducción de los siguientes conceptos:

Definición 1.1. *Una clase de complejidad es un conjunto de problemas que poseen la misma complejidad computacional (**citar wiki**).*

Definición 1.2. *La clase de problemas de decisión que pueden ser resueltos por una máquina de Turing determinista en un tiempo polinomial es conocida como clase P (Polynomial-time)*

En términos generales, P corresponde a la clase de problemas que, de forma realista,

se pueden **resolver** con un ordenador. Muchos de los problemas habituales (ordenación, búsqueda, etc.) pertenecen a esta clase.

Definición 1.3. *Llamamos clase NP (Non-Deterministic Polynomial-time) a aquella formada por los problemas de decisión que son* ***verificables*** *por máquinas de Turing no determinista en tiempos polinómicos.*

Una relación evidente entre ambas clases es que $P \subset NP$, ya que si podemos resolver un problema en tiempo polinómico, evidentemente también podemos verificarlo en tiempo polinómico.

Otro importante subconjunto de la clase *NP* son los problemas *NP-completos.*(**citar wiki a continuación**)

Definición 1.4. *Un problema de decisión C es NP-completo si:*

1. $C \in NP$
2. *Todo problema de NP es* ***reducible polinomialmente*** *a C en tiempo polinómico.*

Una reducción polinómica de L en C es un algoritmo de tiempo polinómico que transforma instancias de L en instancias de C, de manera que la respuesta a C es positiva si y solo si lo es la de L. De forma general, la clase *NP-completo* corresponde a la de los problemas que pueden verificarse de forma sencilla pero que, en el caso de que su espacio de soluciones sea muy grande, solo pueden resolverse por fuerza bruta. Algunos de los problemas que pertenecen a esta clase son el **problema de satisfacibilidad booleana** (SAT), el **problema de la mochila** (comúnmente abreviado por KP), el **problema del ciclo hamiltoniano** o el **problema del viajante**. Esta clase tiene la propiedad (**citar wiki)** de que si algún problema *NP-completo* puede ser resuelto en tiempo polinómico, entonces todo problema en *NP* tiene una solución en tiempo polinómico, es decir, $P = NP$.

A pesar de años de investigación, la cuestión de si $P = NP$ continúa aún abierta y es considerado uno de los problemas del milenio. La importancia de este resultado radica en el hecho de que si $P \neq NP$, entonces los problemas *NP-completos* son intratables, ya que si algún problema en *NP* requiere más tiempo que uno polinomial, entonces uno *NP-completo* también.

Una clase más general de problemas no restringida a los problemas de decisión es la clase de complejidad *NP-difícil* (*NP-hard*). (**citar wiki para la def.**)

Definición 1.5. *La clase de complejidad NP-difícil es el conjunto que contiene a los problemas C tales que todo problema L en NP puede ser transformado polinomialmente en C.*

Esta clase contiene a aquellos problemas que son, como mínimo, tan difíciles como un problema de *NP*. De esta forma, la clase de problemas *NP-completo* puede definirse como la intersección entre las clases *NP* y *NP-difícil*. (**imagen de wiki**)

Un ejemplo de problema de optimización combinatoria que es *NP-difícil* y que será el objeto central de estudio de este trabajo es el llamado *problema de empaquetamiento* o ***Bin Packing Problem***.

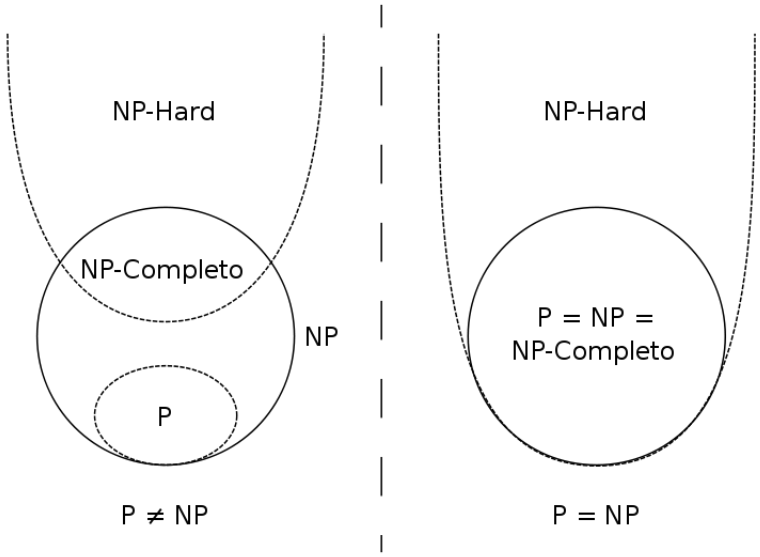

Figura 1.2: Diagrama de Euler de las clases de complejidad más frecuentes

3. Algoritmos heurísticos y metaheurísticos

Como ya se mencionó con anterioridad, para muchos problemas de optimización combinatoria no se conocen algoritmos que sean capaces de obtener una solución óptima en tiempo polinomial. Algunos incluso no admiten el uso de algoritmos de aproximación. En estos casos, nos vemos obligados a usar algoritmos heurísticos.

Los métodos heurísticos son algoritmos que se limitan a proporcionar una "buena" solución del problema, no necesariamente óptima, con un coste computacional razonable. Además, aunque un buen heurístico encuentre muy buenas soluciones para la mayoría de instancias de un problema, no hay garantía de que siempre encuentre una buena solución para todas las instancias del problema.

Existen gran cantidad de métodos heurísticos, lo cual hace que sea complicado dar una clasificación (**citar Rafael Martí**) de los mismos ya que, por ejemplo, muchos de ellos han sido diseñados para resolver un problema concreto. No obstante, podemos dar una clasificación general donde ubicar a los algoritmos heurísticos más conocidos:

- **Métodos de descomposición.** El problema de partida se descompone en subproblemas más sencillos de resolver, sin perder de vista que ambos pertenecen al mismo problema. (**Ejemplo?**)
- **Métodos inductivos.** Estos métodos parten de casos más sencillos del problema general, de manera que analizan propiedades o técnicas que pueden generalizarse al problema completo.
- **Métodos de reducción.** Consiste en seleccionar propiedades que se verifican de forma general en las soluciones consideradas como buenas e introducirlas como restricciones del problema. La finalidad de estos métodos es restringir el espacio de soluciones para simplificar el problema. El inconveniente que presentan estos métodos,

es la posibilidad de dejar fuera del espacio de soluciones nuevo aquellas soluciones óptimas del problema original.

- **Métodos constructivos.** Son métodos que construyen paso a paso una solución del problema. Normalmente son métodos deterministas y suelen estar basados en la mejor elección en cada iteración. (**Ejemplo?** - algoritmos voraces)

- **Métodos de búsqueda local.** A diferencia de los métodos anteriores, los algoritmos de búsqueda o mejora local comienzan con una solución del problema y la mejoran progresivamente. El algoritmo realiza en cada iteración un movimiento de una solución a otra mejor. El método finaliza cuando, para una solución, no existe ninguna otra accesible que la mejore.

Cuando se resuelve un problema por métodos heurísticos, como la optimalidad no está garantizada, se debe medir la calidad de los resultados. Para ello existen diversos procedimientros, entre los cuales podríamos destacar los siguientes:
(**citar Orlando de Antonio Suárez**)

- **Comparación con la solución óptima.** Aunque normalmente se recurre al algoritmo aproximado por no existir un método exacto para obtener el óptimo, o por ser éste computacionalmente muy costoso, en ocasiones puede que dispongamos de un método que proporcione el óptimo para un conjunto limitado de ejemplos. Este conjunto de ejemplos puede servir para medir la calidad del método heurístico. Normalmente se mide, para cada uno de los ejemplos, la desviación porcentual de la solución heurística frente a la óptima, calculando posteriormente el promedio de dichas desviaciones.

- **Comparación con una cota.** En ocasiones el óptimo del problema no está disponible ni siquiera para un conjunto limitado de ejemplos. Un método alternativo de evaluación consiste en comparar el valor de la solución que proporciona el heurístico con una cota del problema (inferior si el problema es de minimizar y superior si es de maximizar). La bondad de esta medida dependerá de la bondad de la cota, es decir, de cómo de cercana se encuentre del óptimo del problema, por lo que de alguna manera, tendremos que tener información de lo buena que es dicha cota. En caso contrario, la comparación propuesta no resulta de utilidad.

- **Comparación con un método exacto truncado.** Para ello elegimos un método exacto que resuelva el problema y establecemos un límite de iteraciones o de tiempo máximo conforme a una serie de criterios que nos garanticen que de esta forma obtenemos una buena solución. Una vez hecho esto, usamos esta solución para compararla con la que obtenemos según el heurístico. Evidentemente, en este caso suponemos que la resolución del problema mediante cualquier método exacto es inabordable por el coste computacional que requiere.

- **Comparación con otros heurísticos.** Es un método usado usalmente en problemas NP-duros para los que se conocen buenos heurísticos.

- **Análisis del peor caso.** Consiste en considerar los ejemplos que sean más desfavorables para el algoritmo y acotar la máxima desviación respecto del óptimo del problema. De esta forma, conseguimos acotar el resultado del algoritmo para cualquier ejemplo. Lo malo es que los resultados no suelen ser representativos del comportamiento medio del algoritmo.

Si bien todos estos métodos han contribuido a ampliar nuestro conocimiento para la resolución de problemas reales, los métodos constructivos y los de búsqueda local constituyen la base de los procedimientos metaheurísticos.

Los metahurísticos son métodos para diseñar y/o mejorar los heurísticos en los que estos métodos clásicos no son efectivos a la hora de resolver un problema difícil de optimización combinatoria. Son algoritmos híbridos que combinan conceptos de distintos campos como la genética, la biología, la física, las matemáticas, la inteligencia artificial o la neurología, por ejemplo. Algunos de los más comunes son los siguientes:

- **Metaheurísticos inspiradas en la física:** Como ya se ha mencionado con anterioridad, el recocido simulado es un ejemplo de este tipo de algoritmos. Es una técnica de búsqueda inspirada en el proceso de calentamiento y posterior enfriamiento de un metal para obtener estados de baja energía en un sólido.
- **Metaheurísticos inspiradas en la evolución:** Son métodos que van construyendo una solución en cada iteración. Consiste en generar, seleccionar, combinar y reemplazar un conjunto de soluciones en la búsqueda de la mejor solución. Un ejemplo de estos métodos son los algoritmos genéticos.
- **Metaheurísticos inspiradas en la biología:** Un ejemplo relativamente reciente de este tipo de algoritmos es la optimización basada en colonias de hormigas (ant colony optimization). Se inspira en el comportamiento estructurado que siguen las colonias de hormigas donde los individuos se comunican entre sí por medio de las feromonas; la repetición de recorridos por los individuos establece el camino más adecuado entre su nido y su fuente de alimentos. El método consiste en simular la comunicación indirecta que utilizan las hormigas para establecer el camino más corto, guardando la información aprendida en una matriz de feromonas.

Capítulo 2

Problema del empaquetamiento (Bin Packing)

1. Introducción

Existen diversas formulaciones para el problema de Bin Packing, al que nos referiremos de ahora en adelante de manera abreviada como BP o BPP. De forma sencilla, podemos enunciarlo como:

Dados n objetos de tamaño $w_1, ..., w_n$, queremos encontrar el menor número de cubos de capacidad c necesario para empaquetar todos los objetos.

A los objetos w_i anteriores se les denomina habitualmente pesos. Son múltiples las aplicaciones que tienen los problemas de tipo BP, además que podemos considerar variantes multidimensionales: desde el llenado de contenedores y/o camiones con restricciones de volumen y peso a la creación de copias de seguridad de archivos o una asignación eficiente de la memoria de un ordenador.

Como vemos, es un problema díficil de resolver debido a que su complejidad crece exponencialmente con el número de objetos a almacenar y variables a considerar, como por ejemplo si dichos objetos fueran tridimensionales y tuviéramos que considerar su volumen y su peso a la hora de imponer las restricciones del problema. Resulta aquí visible la dificultad que presentan los problemas de optimización combinatoria y la necesidad de desarrollar algoritmos suficientemente buenos que puedan resolverlos en tiempos aceptables.

Observación 2.1. *Cuando el número de cubos se limita a uno y cada objeto se caracteriza por su peso, el problema de maximizar el peso de los objetos que pueden caber en el contenedor se conoce como el ya mencionado problema de la mochila.*

El BPP también puede considerarse como un caso especial del *cutting stock problem*, cuyo origen está asociado a la industria maderera: (citar wiki)

Consideremos una lista de m órdenes para las cuales se requiere q_j, $j = 1, ..., m$ piezas para cada una. Posteriormente, se construye una lista de todas las combinaciones posibles de los recortes (frecuentemente llamados *patrones*), asociando a cada uno de ellos una variable entera positiva x_i que representa cuantas veces será utilizado cada patrón. Entonces, el problema de programación lineal entera se modeliza matemáticamente como

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

$$s.a \sum_{i=1}^{n} a_{ij} x_i, \ \forall j = 1, ..., m$$

$$x_i \in \mathbb{Z}^+, \ \forall i = 1, ..., n$$

donde a_{ij} es el número de veces que en la orden j aparece el patrón i y c_i es el costo (a menudo llamado *residuo*) del patrón i.

Así, procedemos a continuación a formular matemáticamente el problema BP.

Sean n objetos (items) y n cubos (bins), donde

$$w_j = \text{peso del item } j,$$
$$c = \text{capacidad de cada cubo,}$$

entonces

$$min \sum_{i=1}^{n} y_i$$

$$s.a \sum_{j=1}^{n} w_j x_{ij} \leq c y_i, \ \forall i = 1, ..., n$$

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

donde

$$y_i = \begin{cases} 1 & \text{si se usa el bin } i \\ 0 & \text{en otro caso} \end{cases}$$

$$x_{ij} = \begin{cases} 1 & \text{si el item } j \text{ se asigna al bin } i \\ 0 & \text{en otro caso} \end{cases}$$

Supondremos, además, que los pesos w_j son enteros positivos. Por lo tanto, sin pérdida de generalidad, podemos suponer que

$$c \text{ es un entero positivo,}$$
$$w_j \leq c, \ \forall j = 1, ..., n.$$

Si algún item no verifica la última suposición, entonces el problema es trivialmente imposible.

En lo que sige, propondremos una serie de algoritmos con los que aproximar las soluciones de distintas instancias del problema BP, a la par que analizaremos cómo de buenos son y qué coste computacional tienen. Para ello, el primer tipo de métodos que analizaremos serán los algoritmos voraces (*greedy alghorithm*).

2. Algoritmos voraces

Por algoritmos voraces se entienden aquellos algoritmos que siguen un esquema de resolución llamado método voraz. Dicho esquema consiste esencialmente en construir una solución de forma incremental, tomando decisiones localmente óptimas en cada paso, lo cual no implica que la solución final sea globalmente óptima. Esquema que siguen algoritmos como, por ejemplo, el método del gradiente, los algoritmos genéticos o los algoritmos de cristalización simulada. Los algoritmos que siguen este método son, en general, los que menos dificultades plantean a la hora de implementar y comprobar su funcionamiento, y suelen aplicarse en problemas de optimización.

Frecuentemente, los algoritmos de tipo greedy y los problemas que pueden resolver, se caracterizan por la mayoría de las siguientes características (**citar Brassard**):

- Tenemos que resolver un problema de optimización y, para construir su solución, tenemos un conjunto de candidatos: en el caso que nos atañe, esos candidatos resultan ser los objetos que queremos introducir en los cubos.
- A medida que el algoritmo avanza, tenemos otros dos conjuntos. Por un lado, tenemos el conjunto formado por los candidatos que ya han sido considerados y elegidos, mientras que por otro tenemos el conjunto con los candidatos que han sido considerados y rechazados.
- Hay una función que verifica si un conjunto particular de candidatos es una solución del problema, independientemente de si dicha solución es la óptima.
- Hay otra función que verifica si un conjunto de candidatos es factible, es decir, si es posible o no completar el conjunto añadiendo más candidatos hasta obtener al menos una solución del problema. De nuevo, esta función tampoco tiene en cuenta la optimalidad de dicha solución.
- Una función más, llamada función de selección, que indica en cada momento cual de los candidatos restantes, que no han sido elegidos ni rechazados, es el que podría ser el mejor.
- Finalmente, una función objetivo que nos proporciona el valor de la solución que hemos encontrado. En nuestro problema de BP, dicha función es el número de bins a minimizar y, a diferencia de las tres funciones anteriores, la función objetivo no aparece explícitamente en el algoritmo.

Un algoritmo voraz avanza paso a paso. Esto quiere decir que, inicialmente, el conjunto de los candidatos elegidos está vacío. Luego, en cada paso, la función de selección elige al mejor candidato restante sin parar a considerar si lo introducimos o no en el conjunto anterior. Si el conjunto ampliado de candidatos elegidos ya no es factible, rechazamos el candidato que estamos considerando actualmente. En este caso, el candidato con el que

hemos probado y que ha sido rechazado no se vuelve a considerar de nuevo. En cambio, si el conjunto ampliado es factible, añadimos el candidato al conjunto de los candidatos elegidos. Cada vez que ampliamos dicho conjunto, verificamos si en ese momento este conjunto es una solución del problema. De esta forma, un algoritmo voraz se vería de la siguiente forma:

```
función greedy (C: set): set
//C es el conjunto de los candidatos
//Construimos la solución en el conjunto
S
S = ∅
while (C ≠ ∅ y no solucion(S)) do
    x ⟵ seleccionar(C)
    C ⟵ C − {x}
    if (factible(S ∪ {x})) then
        S ⟵ S ∪ {x}
    end if
end while
if (solucion(S)) then
    return S
else
    return "No hay solución"
end if
```

A continuación, pasamos a presentar los algoritmos voraces con los que ofreceremos distintas soluciones del problema del BP. Daremos una breve explicación del funcionamiento de los algoritmos, sus pseudocódigos y analizaremos sus complejidades.

2.1. Primeras aproximaciones

Next Fit

Este algoritmo es el más sencillo de implementar y con el que podemos obtener una rápida solución del problema, aunque no muy buena. Funciona de la siguiente manera: comenzamos con una solución en la que no tenemos ningún cubo. A continuación, seleccionamos el primer objeto de la lista de objetos que queremos introducir en los cubos y lo introducimos en un nuevo cubo dado que es el primer elemento seleccionado. Para los siguientes objetos, el algoritmo comprueba si podemos introducir el elemento seleccionado en el último cubo creado. Si se puede, lo introducimos. En otro caso, creamos un cubo nuevo y lo introducimos.

Vemos que, aunque es un algoritmo que nos permite obtener una solución de manera sencilla y rápida (puesto que no hay muchas comprobaciones que realizar), no resulta muy eficiente como veremos. Así, el pseudocódigo del algoritmo sería el siguiente:

```
nextFit (capacity: Int, items: Array[Int]): Array[Bin]
//capacity es la capacidad de los cubos
//items es un array que almacena los elementos que introduciremos en los cubos
//El array solution almacenará los cubos y current será el último cubo creado
solution = ∅
current = new Bin(capacity)
for (item ← items) do
    if (item cabe en current) then
        Añadimos el item a current
    else
        Añadimos current a solution
        current = new Bin(capacity)
        Añadimos el item a current
    end if
end for
Añadimos el último cubo creado a solution
return solution
```

Observemos que, dada una lista de n pesos, el algoritmo hace una única comprobación por cada iteración. Por lo tanto, tenemos que la complejidad del algoritmo es *O(n)*, con lo que el tiempo necesario para la resolución del problema se incrementa de forma lineal conforme lo hace el número de elementos que tenemos que introducir en los cubos. Así pues, como primera aproximación, sería un algoritmo útil, rápido y fácil de resolver para instancias pequeñas del problema en las que no estamos interesados en obtener el óptimo. Además, su sencillez nos permite conseguir resultados como el siguiente:

Teorema 2.1. *Sea M el número de cubos de capacidad c necesarios para empaquetar una lista de n objetos de forma óptima. Entonces el algoritmo Next Fit usará a lo sumo 2M cubos.*

Demostración.

Haremos la demostración para el caso par. Sea k el número de cubos de la solución obtenida al aplicar el algoritmo Next Fit y sea B_i el cubo i-ésimo de la solución. Denotemos por $s(B_i)$ a la suma de los pesos de los objetos empaquetados en el cubo B_i.

Entonces, para cualesquiera dos cubos adyascentes B_i y B_{i+1} se verifica que

$$s(B_i) + s(B_{i+1}) > c$$

Por lo tanto, tenemos que

$$s(B_1) + s(B_2) > c,\ s(B_3) + s(B_4) > c,\ ...,\ s(B_{k-1}) + s(B_k) > c.$$

Con lo que, si sumamos todas estas desigualdades llegamos a

$$\sum_{i=1}^{k} s(B_i) > \frac{k}{2}c.$$

Además, como cualquier cubo de la solución verifica

$$s(B_i) \leq c,$$

llegamos a

$$kc \geq \sum_{i=1}^{k} s(B_i) > \frac{k}{2}c.$$

Por otro lado, observemos que para la solución óptima del problema también se verifica las desigualdades anteriores, es decir,

$$Mc \geq \sum_{i=1}^{M} s(B_i).$$

Por último, teniendo en cuenta que

$$\sum_{i=1}^{M} s(B_i) = \sum_{i=1}^{k} s(B_i),$$

llegamos a

$$Mc \geq \sum_{i=1}^{M} s(B_i) = \sum_{i=1}^{k} s(B_i) > \frac{k}{2}c,$$

de donde deducimos que

$$Mc > \frac{k}{2}c \Longrightarrow 2M > k$$

Podemos ver una prueba más general en [**?**].

□

First Fit

Podríamos considerar a este algoritmo como un primer intento de mejorar el método anterior ya que, aunque no reconsideremos las decisiones tomadas (es decir, si alguno de los items introducidos en los cubos existentes debería introducirse en otro cubo), que es la idea básica de los algortimos voraces, sí que vamos llenando "mejor" los cubos que se van creando en las iteraciones del algoritmo. Además, posteriormente éste será uno de los algortimos que usaremos como base para implemententar mejores técnicas de resolución del problema del BP.

El algoritmo es el siguiente: empezamos igual que en el caso anterior; partimos de una solución que no tiene ninguún cubo. Seleccionamos el primer elemento de nuestra lista de objetos a introducir en los cubos y creamos un primer cubo donde añadimos el elemento. Para los siguientes elementos, recorremos nuestra solución de cubos y añadimos el elemento en el primer cubo que encontremos donde quepa el objeto. Si no hay ningún cubo en el que quepa el elemento seleccionado, creamos un nuevo cubo y lo introducimos.

De esta forma, el pseudocódigo del algoritmo sería el siguiente:

```
firstFit (capacity: Int, items: Array[Int]): Array[Bin]
//capacity es la capacidad de los cubos
//items es un array que almacena los elementos que introduciremos en los cubos
//El array solution almacenará los cubos y current será el último cubo creado
//Inicializamos el algoritmo con el array de soluciones teniendo un cubo vacío
current = new Bin(capacity)
Añadimos el cubo vacío a solution
for (item ← items) do
    i = 0
    while (item no esté añadido y i ≤ tamaño de solution) do
        if (El item cabe en el cubo i-ésimo) then
            Añadimos el item en el cubo i-ésimo
        else
            i = i + 1
        end if
    end while
    if (El item no se ha añadido a ningún cubo) then
        current = new Bin(capacity)
        Añadimos el item a current
        Añadimos el cubo current a solution
    end if
end for
return solution
```

Como mencionamos anteriormente, este algoritmo conseguimos mejorar la solución obtenida según el método Next Fit, pero esto implica que la complejidad algorítmica se incrementa. Dada una lista de n elementos, el bucle for itera sobre cada uno de ellos, por lo que la complejidad introducida a priori por el bucle es $O(n)$. Pero, a su vez, en cada iteración el bucle while recorre en el peor de los casos toda la lista de cubos, con lo que su complejidad es también $O(n)$. Por lo tanto, la complejidad del algoritmo resulta ser $O(n^2)$.

Teorema 2.2. *Sea M el número de cubos de capacidad c necesarios para empaquetar una lista de n objetos de forma óptima. Entonces el algoritmo First Fit no usa más de 1,7M cubos.*

Demostración.
Se omite la demostración de este resultado debido a su larga extensión y a que requiere de unos cuantos resultados previos que se alejan del propósito de lo que aquí tratamos. No obstante, podemos encontrarla en [**?**] □

Best Fit

Este es el primero de los algoritmos que intenta ofrecernos una mejora "considerable" a la hora de llenar los cubos ya que no solo intenta ir llenando todos los cubos en cada iteración, sino que los llena de la mejor forma posible, entendiendo esa forma mejor como aquella en la que el elemento seleccionado rellena mejor en el cubo. Al igual que para First Fit y los métodos posteriores, las mejoras que conseguimos en busca de la solución óptima o una aproximada, aparecen a costa de incrementar la complejidad del algoritmo.

De esta forma, el algoritmo Best Bit es: seleccionamos el primer objeto de nuestra lista, creamos un nuevo cubo con la capacidad dada en el problema e introducimos el objeto. Para los siguientes elementos buscamos el cubo cuya capacidad restante sea menor y en el que, además, quepa el elemento. Si encontramos un cubo que cumpla esta condición introducimos el objeto. En otro caso, creamos un cubo nuevo y lo introducimos.

Obsérvese que el cubo que verifique las condiciones dadas para introducir un nuevo elemento no tiene por qué ser único. Por lo tanto, a la hora de implementar el algoritmo iremos insertando los cubos en nuestra solución de manera ordenada, con lo que cuando vayamos a seleccionar en qué cubo introducimos un nuevo elemento cuando haya más de una elección posible, lo haremos eligiendo el primer cubo que encontremos. Así, el pseudocódigo del algoritmo queda como sigue:

```
bestFit (capacity: Int, items: Array[Int]): Array[Bin]
//capacity es la capacidad de los cubos
//items es un array que almacena los elementos que introduciremos en los cubos
//El array solution almacenará los cubos y current será el último cubo creado
//Inicializamos el algoritmo con el array de soluciones sin ningún cubo
//targetBin es la posición en solution del cubo con la menor capacidad restante que es
mayor o igual que el peso del item seleccionado
//smallerThanTarget es un método de búsqueda binaria con el que, dado un array
de cubos ordenados según sus capacidades restantes de forma creciente y un elemento,
encuentra la posición del menor elemento del array que es mayor o igual que el elemento
dado
solution = ∅
for (item ← items) do
    targetBin = 1 + smallerThanTarget(item, solution)
    if (targetBin ≥ 0 y targetBin ≤ longitud de solution - 1) then
        Añadimos el item en el cubo que se encuentra en la posicion targetBin
        Reordenamos el array solution de menor a mayor capacidad restante
    else
        current = new Bin(capacity)
        Añadimos el item en el cubo current
        Añadimos el cubo current en solution
        Reordenamos el array solution de menor a mayor capacidad restante
    end if
end for
return solution
```

Como ya veremos en el capítulo dedicado a explicar la implementación de los métodos aquí descritos, lo primero que debemos tener en cuenta para analizar la complejidad de este algoritmo es que, en cada iteración, para localizar el cubo donde el objeto seleccionado quede más ajustado realizamos una búsqueda binaria con el método *smallerThanTarget*, que es *O(log n)*. Por otro lado, tras insertar el elemento, la capacidad del cubo disminuirá, con lo que tendremos que desplazarlo a la izquierda (ya que en nuestra implementación ordenaremos los cubos de menor a mayor capacidad restante) para seguir manteniendo los cubos en orden. En el peor de los casos, este desplazamiento será *O(n)* si, por ejemplo, tenemos que desplazar un cubo que está en el extremo derecho al extremo izquierdo.

Por lo tanto, cada iteración en la que insertamos un nuevo elemento será *O(log n) + O(n) = O(n)*. De esta forma, como tenemos que insertar n elementos, el coste total del algoritmo será $n \cdot O(n) = O(n^2)$.

Worst Fit

El método Worst Fit es una variante del algoritmo Best Fit. Mientras que en el método Best Fit vamos introduciendo los pesos en aquellos cubos que quepan y que tengan la menor capacidad restante posible, en el Worst Fit los introducimos en los cubos que quepan y que, además, tengan la mayor capacidad restante de entre todos ellos. También, al igual que en el método Best Fit, a la hora de implementar el algoritmo tenemos la solución con los cubos ordenados de menor a mayor capacidad restante. Así, como tenemos que elegir aquel cubo con la mayor capacidad restante, lo único que tendremos que comprobar en cada iteración del algoritmo es que el peso seleccionado quepa en el último cubo de la solución.

Así, el algoritmo quedaría: inicializamos el algoritmo igual que en Best Fit, partiendo de una solución que tiene un cubo en el cual introducimos el primer peso. A continuación, para cada elemento buscamos aquel cubo con la mayor capacidad restante y en el que quepa el elemento. Si lo encontramos, introducimos el peso. En otro caso creamos un cubo nuevo, introducimos el elemento y añadimos el cubo a la solución. Por lo tanto, el pseudocódigo del algoritmo quedaría como:

```
worstFit (capacity: Int, items: Array[Int]): Array[Bin]
//capacity es la capacidad de los cubos
//items es un array que almacena los elementos que introduciremos en los cubos
//El array solution almacenará los cubos y current será el último cubo creado
//targetBin es la posición en solution del cubo con la menor capacidad restante que es
mayor o igual que el peso del item seleccionado
//Inicializamos el algoritmo con el array de soluciones sin ningún cubo
solution = ∅
for (item ← items) do
    if (solution no está vacío y el item cabe en el último cubo de solution) then
        Añadimos el item en el cubo que se encuentra en la última posición
        Reordenamos el array solution de menor a mayor capacidad restante
    else
        current = new Bin(capacity)
        Añadimos el item en el cubo current
        Añadimos el cubo current en solution
        Reordenamos el array solution de menor a mayor capacidad restante
    end if
end for
return solution
```

El razonamiento que tenemos que seguir para analizar la complejidad de este algoritmo es prácticamente análogo al caso del algoritmo Best Fit. A diferencia del algoritmo anterior, en este método no tenemos que realizar una búsqueda binaria para localizar el

cubo donde insertar el objeto, sino que siempre lo intentamos insertar en el último cubo de nuestra solución, lo cual pareciera que reduce la complejidad del algoritmo. Sin embargo, una vez insertado el elemento en el cubo, ya sea en el último cubo de la solución o creando uno nuevo, de nuevo la capacidad de dicho cubo disminuirá y tendremos que desplazarlo a la izquierda si así fuera necesario. En el peor de los casos, este desplazamiento será $O(n)$ si tenemos un cubo por elemento y tuviéramos que desplazar el cubo (siempre desde el extremo derecho) al extremo izquierdo.

Esto significa que cada iteración en la que insertamos un elemento, su complejidad será $O(n)$ y, dado que tenemos que insertar n n elementos, el coste total del algoritmo será $n \cdot O(n) = O(n^2)$.

De aquí en adelante los métodos que emplearemos para buscar mejores soluciones para el problema del BP utilizarán de base las técnicas anteriores. No obstante, antes de llegar a ellos podemos mencionar una serie de variaciones de los algoritmos anteriores que, en función de la instancia del problema que estemos considerando, pueden ofrecer mejores soluciones que los algoritmos que hemos visto hasta ahora.

2.2. Algunas variaciones de los métodos anteriores

Next k Fit

El método Next k Fit podría considerarse como una generalización del método Next Fit. La diferencia es que mientras que en el método Next Fit solo tenemos "abierto" el último cubo de la solución (es decir, solo miramos el último a la hora de comprobar si podemos introducir un nuevo elemento o si creamos un cubo nuevo donde introducirlo), en este método tenemos abierto los k últimos cubos. Así, dada una solución de cubos de longitud n y un nuevo elemento a introducir en ellos, comenzaríamos comprobando si el elemento cabe en el cubo de posición $n - (k - 1)$; si cabe lo introducimos, en otro caso pasamos al cubo de posición $n - (k - 2)$ y así sucesivamente hasta llegar al último cubo. Como dijimos con anterioridad, el método Next Fit no es más que un caso particular de este método cuando $k = 1$.

Podemos apreciar a simple vista que métodos de este tipo podrían mejorar al algoritmo Next Fit cuando $k > 1$ dado que nos permite "llenar" mejor los cubos de la solución al comprobar si un elemento dado cabe en más de un cubo.

```
nextKFit (k: Int, capacity: Int, items: Array[Int]): Array[Bin]
//capacity es la capacidad de los cubos
//items es un array que almacena los elementos que introduciremos en los cubos
//El array solution almacenará los cubos y current será el último cubo creado
//k es un número entero que representa el número de cubos que permanecerán abiertos
en cada iteración
//Inicializamos el algoritmo con el array de soluciones sin ningún cubo
solution = ∅
current = new Bin(capacity)
for (item ← items) do
    if (solution = ∅) then
        if (puedo añadir el item en el cubo current) then
            Añadimos el item en el cubo current
        else
            Añadimos el cubo current en solution
            current = new Bin(capacity)
            Añadimos el item en el cubo current
            Añadimos el cubo current en solution
        end if
    else if (solution ≠ ∅ y solution.length < k) then
        aux = 0
        while (aux ≤ k − 1 y el item no esté añadido) do
            if (el item cabe en el cubo que se encuentra en la posición aux) then
                Añado el elemento al cubo
            else
                aux += 1
            end if
        end while
        if (el item no se ha añadido en ningún cubo) then
            current = new Bin(capacity)
            Añadimos el item en el cubo current
            Añadimos el cubo current en solution
        end if
    else
        i = 1
        while (el item no esté añadido y no hayamos comprobado los últimos k cubos)
do
            if (podemos añadir el item en el cubo (k-i)-ésimo) then
                Introducimos el item en el cubo (k-i)-ésimo
            else
                i += 1
            end if
        end while
        if (el item no se ha añadido) then
            current = new Bin(capacity)
            Añadimos el item en el cubo current
            Añadimos el cubo current en solution
        end if
    end if
end for
return solution
```

La complejidad de este algoritmo es idéntica al Next Fit. Esto se debe a que el bucle for recorre toda la lista de objetos, luego al igual que en los algoritmos previos, su complejidad es *O(n)*. Por otra parte, el bucle while itera, a lo sumo, k veces, con lo que la complejidad del algoritmo es $k \cdot O(n) = O(n)$ ya que k es una constante.

Algoritmos de tipo decreasing/increasing

Este tipo de algoritmos recoge los métodos ya planteados y les realiza una pequeña variación que es, básicamente, ordenar antes de empezar el proceso la lista de objetos a introducir en los cubos de manera descendente según sus pesos. Así, este tipo de variación da lugar a los algoritmos *First Fit Decreasing*, *Best Fit Decreasing* o incluso *Worst Fit Decreasing*.

Lógicamente, el resto de métodos voraces que presentamos en este trabajo también son susceptibles de introducir este cambio en su algoritmo, pero su uso no suele ser tan común ni tampoco introducen mejoras considerables en la obtención de mejores resultados. Por otro lado, en lugar de ordenar la lista de objetos de manera descendente, también podría hacerse de forma ascendente, lo cual podría llegar a resultar beneficioso según la instancia del problema que se estuviera tratando de resolver.

En cuanto a la complejidad del algoritmo resultante cuando aplicamos este cambio, podemos observar que no varía en absoluto para los métodos que estamos considerando. Esto es así porque la complejidad del algoritmo de reordenación previa es $O(n \log(n))$, que es la que tienen los mejores algoritmos de ordenación. De esta forma la complejidad del nuevo algoritmo no es más que la complejidad de la suma de estos dos algoritmos, que según tratamos en el primer capítulo, resulta que es el máximo de ambas complejidades, es decir, la peor, que es la complejidad del algoritmo de partida.

Teorema 2.3. *Sea M el número de cubos de capacidad c necesarios para empaquetar una lista de n objetos de forma óptima. Entonces el algoritmo First Fit Decreasing no usa más de 1,5M cubos [**?**].*

Demostración.
Sea I el conjunto de los elementos a introducir en los cubos. Sea k el número de cubos hallados según el algoritmo First Fit Decreasing y sea k* la solución óptima del problema de BP.

Consideremos el cubo que se encuentra en la posición $j = [\frac{2}{3}k]$, donde $[\cdot]$ denota la parte entera de un número. Si dicho cubo contiene un elemento i con $s_i > \frac{c}{2}$, donde s_i representa el peso del elemento i, entonces cada cubo $j' < j$ no tiene capacidad restante suficiente para el elemento i. Por lo tanto, a j se le asignó un elemento i' con $i' < i$. Como los elementos se consideran en orden no creciente de tamaño, tenemos $s_{i'} \geq s_i > \frac{c}{2}$. Es decir, hay al menos j elementos de tamaño superior a $\frac{c}{2}$. Estos elementos deben colocarse en cubos individuales. Esto significa que

$$k* \geq j \geq \frac{2}{3}k$$

En caso contrario, el cubo j y cualquier cubo $j' > j$ no contienen ningún elemento de tamaño superior a $\frac{c}{2}$. Por tanto, los cubos que se encuentran en las posiciones $j, j+1, ..., k$ contienen al menos $2(k-j)+1$ elementos, ninguno de los cuales cabe en los cubos que se encuentran en las posiciones $1, ..., j-1$. Por lo tanto, tenemos

$$c \cdot s(I) > c \cdot min\{j-1, 2(k-j)+1\} \geq c \cdot min\{[\frac{2}{3}k]-1, 2(k-(\frac{2}{3}k+\frac{2}{3}))+1\} = c \cdot ([\frac{2}{3}k]-1)$$

y $c \cdot k* \geq c \cdot s(I) > c \cdot ([\frac{2}{3}k]-1)$. Pero esto implica que

$$k* \geq [\frac{2}{3}k] \geq \frac{2}{3}k$$

,

de donde llegamos a lo que queríamos probar. □

Almost Worst Fit

Este algoritmo es una variación del método Worst Fit. La diferencia es que, mientras que en Worst Fit introducimos los pesos en aquel cubo con mayor capacidad restante (cuando cabe el elemento; en otro caso creamos un cubo nuevo y lo introducimos), en Almost Worst Fit comprobamos primero si el peso dado cabe en el cubo con segunda mayor capacidad restante.

Es decir: al igual que en Worst Fit, tenemos la solución de cubos ordenada de menor a mayor capacidad restante por lo que, dado un elemento a introducir en alguno de los cubos de nuestra solución, primero comprobamos si podemos introducirlo en el penúltimo cubo de la solución. Si podemos lo introducimos pero, si no, comprobamos si podemos añadirlo al último cubo de la solución. Si podemos añadirlo en este cubo lo hacemos, pero en otro caso creamos un cubo nuevo, introducimos el elemento y, al igual que en los métodos Best Fit y Worst Fit, hacemos una inserción ordenada del cubo en nuestra solución según su capacidad restante. De esta forma, el pseudocódigo del algoritmo sería el siguiente:

```
almostWorstFit (capacity: Int, items: Array[Int]): Array[Bin]
//capacity es la capacidad de los cubos
//items es un array que almacena los elementos que introduciremos en los cubos
//El array solution almacenará los cubos y current será el último cubo creado
//targetBin es la posición en solution del cubo con la menor capacidad restante que es
mayor o igual que el peso del item seleccionado
//Inicializamos el algoritmo con el array de soluciones sin ningún cubo
solution = ∅
for (item ← items) do
    found = false
    if (solution no está vacío) then
        if (solution.length = 1 y cabe en el cubo) then
            found = true
            targetBin = solution.length - 1
        else if (solution.length > 1) then
            if (item cabe en el penúltimo cubo) then
                found = true
                targetBin = solution.length - 2
            else if (item cabe en el último cubo) then
                found = true
                targetBin = solution.length - 1
            end if
        end if
    end if
    if (found) then
        Añadimos el item al cubo en la posición targetBin
        Reordenamos el array solution de menor a mayor capacidad restante
    else if (solution está vacío o found = false) then
        current = new Bin(capacity)
        Añadimos el item en el cubo current
        Añadimos el cubo current en solution
        Reordenamos el array solution de menor a mayor capacidad restante
    end if
end for
return solution
```

Como puede apreciarse fácilmente, el pseudocódigo de este algoritmo es prácticamente igual al del método Worst Fit, lo cual nos induce a pensar que la complejidad del algoritmo es la misma. Efectivamente esto es así. Al igual que en los métodos anteriores, el bucle for que hay presente tiene un coste *O(n)*. Por otro lado, las comprobaciones que hacemos dentro del bucle no añaden complejidad al algoritmo. Sin embargo, en cada iteración del bucle tenemos que, o bien reordenar el cubo en el que hayamos introducido el nuevo elemento en nuestra solución o bien hacer una inserción ordenada de un nuevo cubo en la misma cuando sea necesario. En ambos casos, como ya se explicó con anterioridad, esta operación tiene un coste *O(n)* por lo que, de nuevo, la complejidad del algoritmo será $O(n^2)$.

Vemos que de esta forma, al igual que sucedía con el algoritmo Next k Fit, introdu-

cimos una variación o posible mejora del algoritmo de partida sin incrementar el coste computacional del método.

Destaquemos de nuevo que estas primeras variaciones que hemos introducido, aunque puedan mejorar el resultado de nuestra solución, no producen una mejora en el coste computacional de los algoritmos que tratamos de mejorar, lo cual resulta fundamental dado que, como hemos visto, la complejidad de estos algoritmos es elevada y resultaría ineficiente y costoso en tiempo el resolver instancias muy grandes del problema haciendo uso de los mismos.

Por lo tanto, deberemos emplear otras técnicas que nos permitan implementar estos mismos algoritmos pero con un coste computacional menor. Esto es precisamente lo que haremos en la siguiente sección, que será implementar estos métodos haciendo uso de los *árboles AVL balanceados*.

2.3. Árboles AVL.

Los árboles son una de las estructuras de datos utilizadas con mayor frecuencia, aunque también de las más complejas. Se caracterizan por almacenar sus elementos de forma jerárquica, a diferencia de los arrays o las listas que lo hacen de manera lineal.

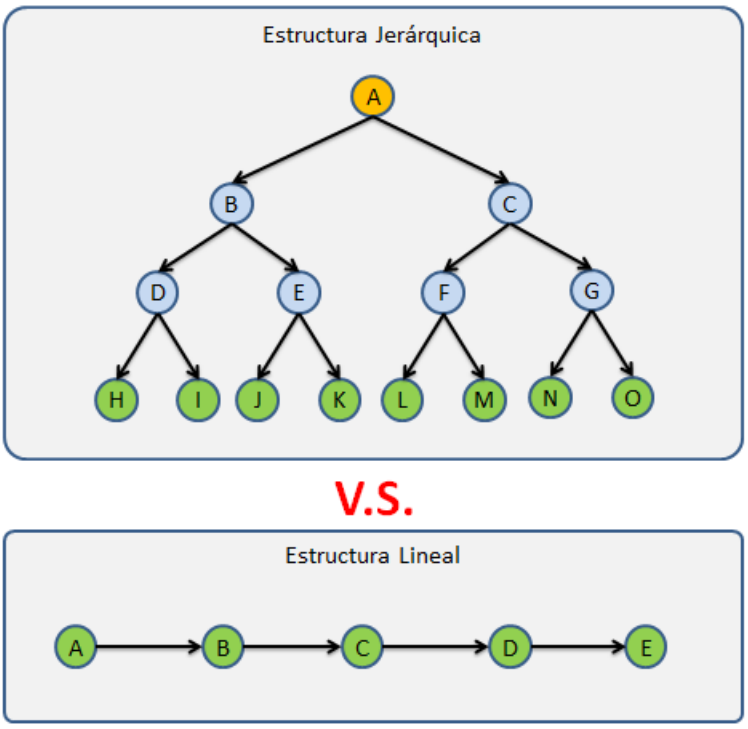

Figura 2.1: Estructura de un árbol vs estructura lineal

De esta forma, podemos enumerar los siguientes elementos que conforman un árbol [?]:

- **Nodos:** Se le llama nodo a cada elemento que contiene un árbol.

- **Nodo raíz:** Se llama nodo raíz o simplemente raíz al primer nodo de un árbol. Solo un nodo del árbol puede ser la raíz.
- **Nodo padre:** Se llaman así a aquellos nodos que tienen al menos un hijo.
- **Nodo hijo:** Son aquellos nodos que tienen un padre.
- **Nodo hermano:** Los nodos hermanos son aquellos nodos que comparten un mismo padre.
- **Nodo hoja:** Son todos los nodos que no tienen hijos, los cuales siempre se encuentran en los extremos del árbol.
- **Nodo rama o internos:** Estos son todos aquellos nodos que no son la raíz y que ademas tiene al menos un hijo.

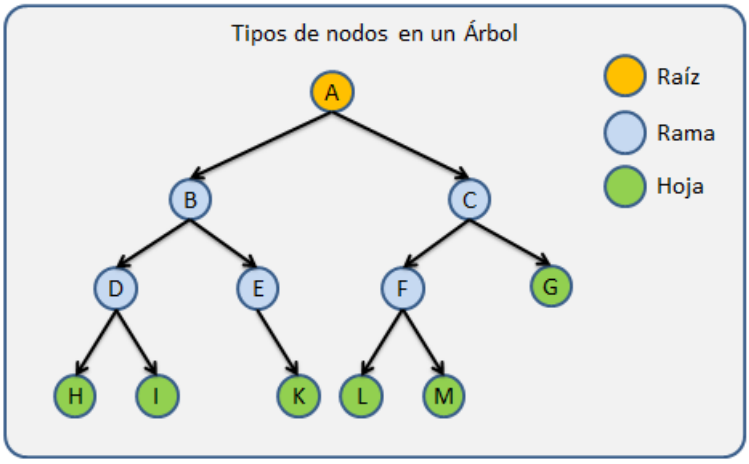

Figura 2.2: Tipos de nodos

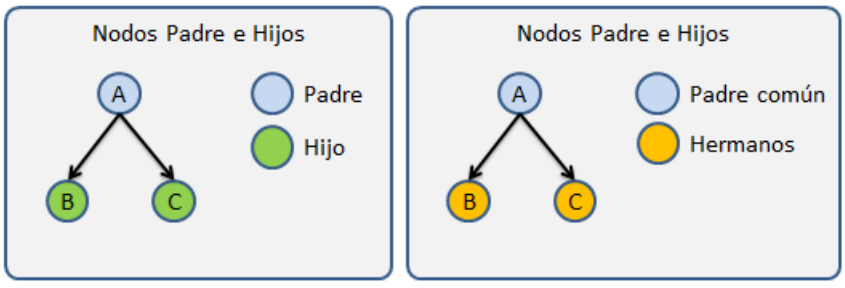

Figura 2.3: Nodos padre, hijos y hermanos

Por otra parte, haciendo uso de estos elementos, definimos los siguientes conceptos o características sobre un árbol:

- **Nivel:** Nos referimos como nivel a cada generación dentro del árbol. Por ejemplo, cuando a un nodo hoja le agregamos un hijo, el nodo hoja pasa a ser un nodo rama pero, además, el árbol crece una generación por lo que tiene un nivel más.Cada generación tiene un nivel distinto que el resto de generaciones. De este modo:

1. Un árbol vacío tiene 0 niveles.
2. El nivel del nodo raíz es 1
3. El nivel de cada nodo se calcula contando cuantos nodos existen sobre él hasta llegar a la raíz más 1. También podrá calcularse de forma inversa, es decir, contando cuantos nodos existen desde la raíz hasta el nodo buscado más 1.

- **Altura:** Llamamos altura al número máximo de niveles de un árbol. Teniendo en cuenta esta idea, podemos calcular también la altura de cada nodo como la altura del subárbol correspondiente a ese nodo. Se calcula de manera recursiva de la siguiente manera:

$$altura = max(altura(hijo1), altura(hijo2), ..., altura(hijoN)) + 1$$

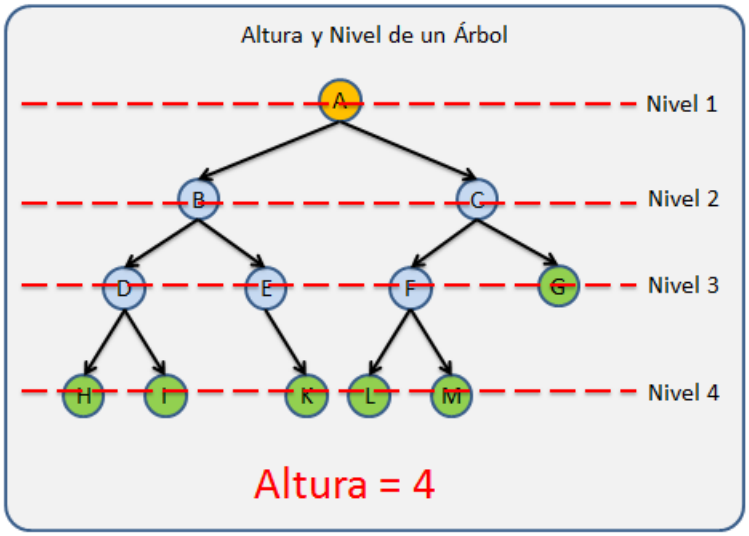

Figura 2.4: Niveles y altura de un árbol

- **Peso:** Llamamos peso al número de nodos que tiene un árbol. Este factor nos da una idea del tamaño del árbol y el tamaño en memoria que nos puede ocupar en tiempo de ejecución. Se puede calcular mediante cualquier algoritmo que recorra todos los nodos del árbol. De forma recursiva, el peso se puede calcular como la suma del peso de los subárboles hijos más 1:

$$peso = peso(hijo1) + peso(hijo2) + ... + peso(hijoN) + 1$$

- **Orden:** Es es el número máximo de hijos que puede tener un nodo.

- **Grado:** Número mayor de hijos que tiene alguno de los nodos del árbol. Observemos que el grado se encuentra limitado por el orden. Para calcularlo, al igual que en los casos anteriores, lo hacemos de forma recursiva contando el número de hijos de cada subárbol y el del nodo actual:

$$grado = max(grado(hijo1), grado(hijo2), ..., grado(hijoN), grado(this))$$

Podemos encontrar distintas clasificaciones para los árboles, pero para el propósito de este trabajo sólo mencionaremos una, dentro de la cual se encuentra el tipo de árboles con los que trabajaremos en el resto de la sección.

Dicho tipo de árboles son los conocidos como *árboles n-arios*. Los árboles n-arios son aquellos árboles donde el número máximo de hijos por nodo es n. En particular, los árboles cuyo número máximo de hijos por nodos es 2 reciben el nombre de *árboles binarios*. Obsérvese que los árboles binarios tienen grado 2.

Definimos entonces los **árboles binarios balanceados**, **árboles binarios de búsqueda balanceados**, o simplemente, **árboles AVL** [**?**] como aquellos árboles binarios en los que las alturas de los dos subárboles de cada nodo difieren, a lo sumo, en 1. Reciben este nombre en honor a sus creadores Adelson-Velski y Landis [**?**] y es la estructura de datos que usaremos a lo largo de esta sección para mejorar algunos de los métodos voraces que vimos con anterioridad.

Sobre este tipo de árboles definimos el *factor de equilibrio* (FE), que no es más que la diferencia entre las alturas de sus subárboles, es decir:

$$FB = altura(hijoDrch) - altura(hijoIzq)$$

De este modo, deducimos que los posibles valores para el FE son -1, 0 y 1. Que un nodo tenga un FE igual a 0 indicará que las alturas del hijo derecho y el izquierdo son iguales. Por otro lado, que el FE sea 1 significa que la altura del hijo derecho es mayor que la del izquierdo, mientras que cuando vale -1 tenemos el caso contrario, es decir, que la altura del hijo izquierdo es mayor a la del hijo derecho.

En caso de que el árbol no esté balanceado, es decir, que se desequilibre a medida que vamos introduciendo nuevos nodos, habrá que rebalancear el árbol para que siga siendo un árbol AVL válido.

Por consiguiente, en el caso de que el árbol esté desequilibrado, existen cuatro operaciones que corrigen el balanceo [**?**]. A saber:

1. Rotación simple a la derecha.
2. Rotación simple a la izquierda.
3. Rotación doble a la derecha.
4. Rotación doble a la izquierda.

Las rotaciones dobles son, básicamente, realizar dos rotaciones simples seguidas. De esta forma, un rotación doble a la derecha es una rotación simple a la derecha seguida de una rotación simple a la izquierda. Por otro lado, una rotación doble a la izquierda se compone de una rotación simple a la izquierda seguida de una rotación simple a la derecha.

Así, por ejemplo, una rotación doble a la derecha resultaría como en la Figura 2.8.

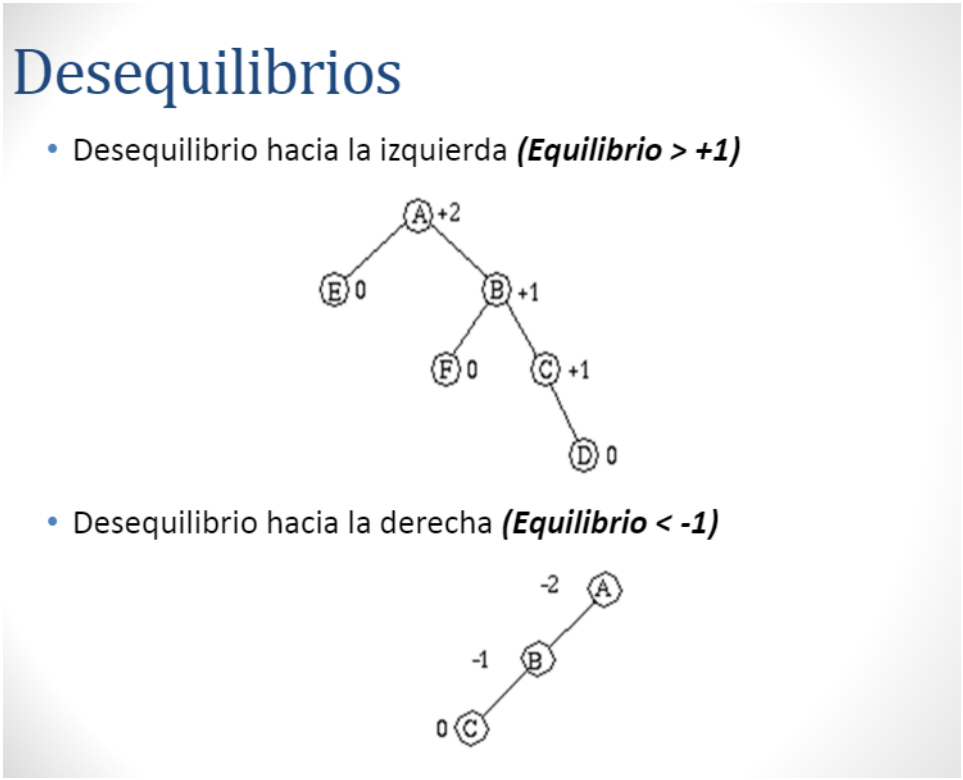

Figura 2.5: Tipos de desequilibrios

Como vemos, la idea detrás de estas operaciones de equilibrado, es desplazar los nodos de la rama más larga a la rama más corta. Debemos mencionar que existen más tipos de equilibrados y que del que estamos haciendo uso es el que se conoce como equilibrado en altura. Una característica importante de este tipo de equilibrado es que se hace en orden ascendente, es decir, sólo en el camino desde el nodo insertado o borrado hacia la raíz.

Hasta ahora hemos dado definido el tipo de árbol que vamos a usar y, sobre él, hemos definido una serie de conceptos entre los cuales el más importante hasta ahora ha sido el de equilibrio. Por otra parte, para garantizar este equilibrio, hemos definido las operaciones de las rotaciones. Por lo tanto, la pregunta que nos hacemos a continuación de manera natural es: ¿por qué estamos interesados en utilizar esta estructura de datos?

Y la respuesta la encontramos en la solución de la cuestión: ¿cuál es la complejidad de la operación de búsqueda en un árbol AVL? Veámoslo.

Supongamos que tenemos un árbol AVL de n elementos y con una altura h. Si queremos buscar un elemento en el árbol en el peor de los casos haremos h comparaciones. Esto es así porque, en este tipo de árboles, tenemos definido un *orden* según el cual cada nodo padre es "mayor" o "igual" que todos los que estás en su hijo izquierdo y "menor" que todos los del hijo derecho.

Por lo tanto, la complejidad para localizar un elemento será $O(h)$ [**?**]. Notemos también que, en cada iteración de la búsqueda, dividimos el espacio de búsqueda a la mitad. Así

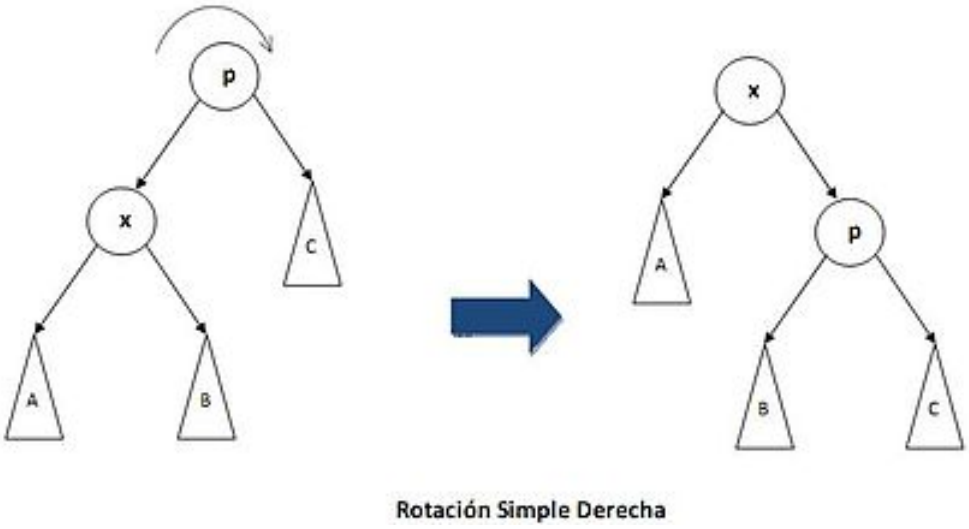

Figura 2.6: Rotación simple a la derecha.

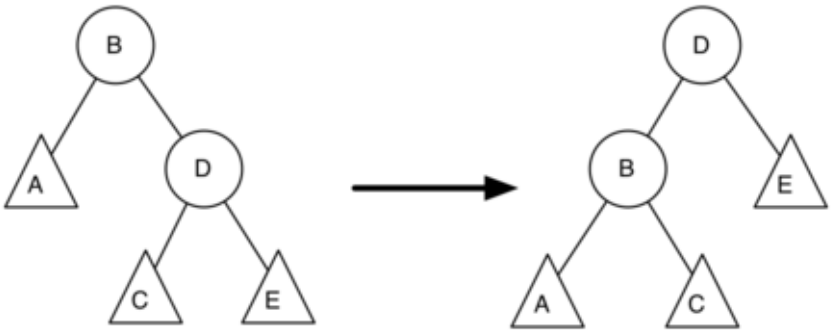

Figura 2.7: Rotación simple a la izquierda

Sea *N(h)* el mínimo número de nodos de un árbol AVL de altura h. Entonces

$$N(h) = 1 + N(h-1) + N(h-2),\ h > 2,$$

donde el primer sumando se debe al nodo que se encuentra en la raíz del árbol, el segundo a que la altura de un hijo debe ser $h-1$ y el tercero a que la diferencia de las alturas de los hijos debe ser menor o igual que 1, y el mínimo de nodos corresponde a $h-2$. Por lo tanto,

$$N(h) = O(\phi^h) \rightarrow h = O(\log_\phi(n))$$

De esta forma vemos que la complejidad de la operación de búsqueda en un árbol AVL es $O(h) = O(\log_\phi(n))$

Así, las mejoras que vamos a incorporar en esta sección para mejorar algunos de los algoritmos anteriores, no es más que representar la solución de nuestro problema mediante una estructura de árbol AVL en lugar de en una lista (array) de cubos.

Los algoritmos que implementaremos haciendo uso de esta estructura serán First Fit, Best Fit, Worst Fit y sus correspondientes algoritmos de tipo decreasing, ya que la im-

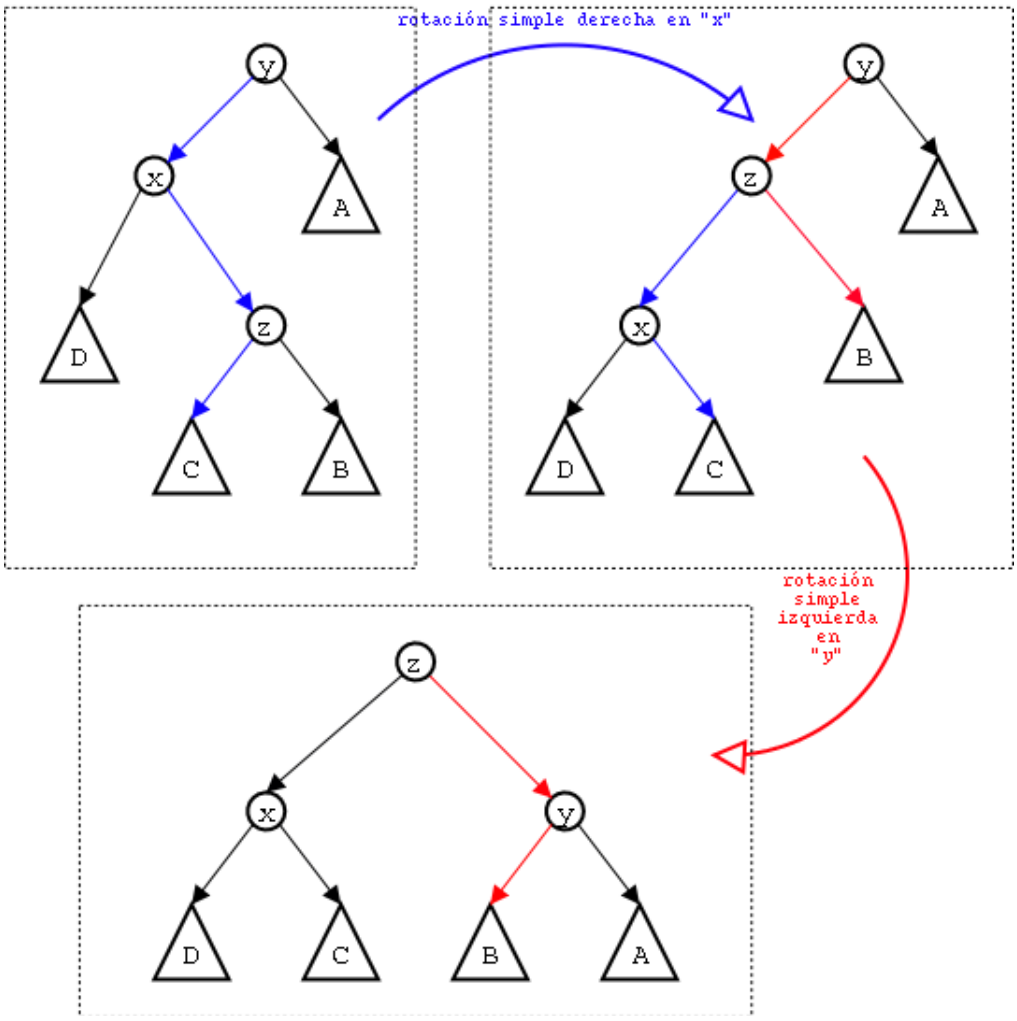

Figura 2.8: Rotación doble a la derecha

plementación de estos últimos no requiere de gran esfuerzo extra para realizarlos.

De nuevo, recordemos que la complejidad de estos algoritmos es $O(n^2)$ debido a que, a la hora de buscar un cubo donde introducir un nuevo elemento y reordenar el array solución, estas operaciones resultaban tener una complejidad $O(n)$, por lo que al tener n elementos a introducir en los cubos la complejidad de los algoritmos finalmente es $O(n^2)$. En cambio, como veremos a medida que vayamos realizando los pseudocódigos de los métodos que implementan estos algoritmos, dado que la búsqueda en árboles AVL tiene complejidad $O(\log_\phi(n))$, los algoritmos tendrán ahora una complejidad $O(n \log_\phi(n))$, lo cual mejora en gran medida la eficiencia de dichos algoritmos a la hora de resolver instancias del BPP muy grandes.

A continuación, vamos a estudiar cuál es la lógica que siguen estos algoritmos cuando hacen uso de esta estructura de datos. Presentaremos el pseudocódigo de los métodos principales mientras que dejaremos para el capítulo siguiente una explicación más exhaustiva de cómo se ha implementado todo el proyecto.

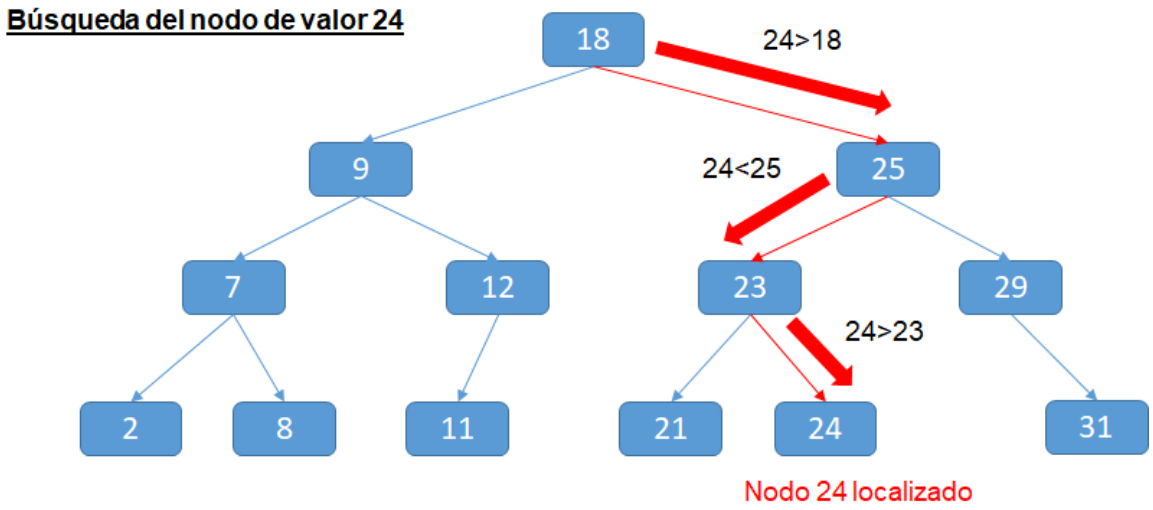

Figura 2.9: Ejemplo de búsqueda en un árbol binario.

First Fit usando árboles AVL

Dado que un árbol está formado por nodos, lo primero que tenemos que establecer es qué información va a almacenar cada nodo del árbol. De este modo, cada nodo contiene:

1. La altura del nodo en el árbol.
2. La capacidad restante máxima del nodo, es decir, la mayor capacidad restante entre la del cubo del nodo actual y la de los cubos de sus dos hijos.
3. Un cubo, que a su vez contiene:
   - Su capacidad restante.
   - Una lista con los pesos que ya se han introducido.
4. Aparte de estos datos, cada nodo también tendrá una referencia a su hijo izquierdo y a su hijo derecho, pero esta información solo resulta relevante a la hora de implementar el algoritmo en el ordenador.

Notemos que, del punto 2, deducimos que la capacidad restante máxima del árbol será la capacidad restante máxima de la raíz. Además, si cuando queremos introducir un nuevo objeto en el árbol vemos que tiene un peso mayor que la capacidad restante máxima de la raíz, esto quiere decir que el objeto no cabe en ninguno de los cubos del árbol, es decir, de la solución, con lo que tendremos que crear un nuevo cubo para este objeto.

Recordemos que anteriormente también mencionamos que sobre los árboles AVL tenemos que definir un orden, pero al igual que en la primera implementación de este algoritmo, el orden que definimos en el árbol no es más que el orden en el que se han ido creando los cubos. De esta forma, deducimos que cada vez que creemos un nuevo cubo lo haremos añadiendo un nuevo nodo con dicho cubo al final de la *espina derecha*, que no es más que el camino que resulta de recorrer el árbol desde la raíz a través de todos sus hijos derechos, hasta llegar a la hoja que se encuentra en el extremo inferior derecho.

Observemos que, como siempre introduciremos los nodos nuevos al final de la espina derecha, para equilibrar el árbol cuando sea necesario solo hará falta realizar rotaciones a la izquierda.

De esta forma, los métodos que resuelven el algoritmo First Fit haciendo uso de la estructura de árboles AVL son los siguientes:

- **addNewBin(bin: Bin)**: Este método toma un cubo y lo añade al final de la espina derecha. Para mantener el invariante de los árboles AVL, en cada nodo de la espina derecha, si la altura resultante del hijo derecho es más de una unidad superior a la del hijo izquierdo, habrá que aplicar una rotación simple a la izquierda.

  Así, el pseudocódigo del método sería

```
def addNewBin(bin: Bin): Unit
//addBinToNode es un método recursivo con el que vamos recorriendo cada hijo derecho
//hasta llegar al final de la espina derecha, que es donde creamos un nuevo nodo con
//el cubo dado
def addBinToNode(node: Node): Node
if (node.right es null) then
    //Esto quiere decir que node no tiene hijo derecho
    Creamos el nodo hijo derecho de node y añadimos el cubo
    Recalculamos la altura de node
    Recalculamos la capacidad restante máxima de node
    Devolvemos el nodo node
else
    //Esto quiere decir que el hijo derecho de node existe y que, por tanto, no estamos
    //al final de la espina derecha
    node.right = addBinToNode(node.right)
    if height(node.right) - height(node.left) > 1 then
        Hacemos una rotación simple a la izquierda de node
    else
        Recalculamos la altura de node
        Recalculamos la capacidad restante máxima de node
        Devolvemos el nodo node
    end if
end if
if root es null then
    //Esto quiere decir que el árbol está vacío
    Creamos el nodo raíz y le añadimos el cubo
else
    root = addBinToNode(root)
end if
```

- **addFirst(initialCapacity: Int, weight: Int): Unit**: Este método toma la capacidad de los cubos del problema, el peso de un objeto a introducir y lo añade al primer cubo que pueda contenerlo o añade un nuevo cubo al final de la espina

derecha si el objeto nuevo no cabe en ningún cubo. El algoritmo funcionaría de la siguiente forma:

- Si el árbol está vacío o el objeto no cabe en ningún cubo, se añadirá un nuevo nodo con un cubo con el objeto al final de la espina derecha.
- En otro caso, si la capacidad restante máxima del hijo izquierdo es mayor o igual al peso del objeto, se añadirá el objeto al primer cubo posible del hijo izquierdo.
- En otro caso, si la capacidad restante del cubo en el nodo raíz (o nodo padre) es mayor o igual al peso del objeto, se añadirá el objeto al cubo en la raíz.
- En otro caso, se añadirá el objeto al primer cubo posible del hijo derecho.

- **addAll(): Unit**: Con este método añadimos todos los objetos de la lista de objetos dada en la instancia del problema y los añadimos al árbol. Es un método muy sencillo cuyo pseudocódigo es

```
def addAll(): Unit
//instance es un objeto que representa una instancia del problema del BP
//El objeto instance tiene dos atributos, capacity e items, que son la capacidad de los
//cubos y la lista de objetos a introducir en los cubos
for (item ← instance.items) do
    addFirst(instance.capacity, item)
end for
```

Para el estudio de la complejidad, supongamos que tenemos n elementos a introducir en los cubos. Dado un elemento nuevo a introducir, en el peor de los casos, tendremos que hacerlo al final de la espina derecha o crear un nodo nuevo aquí también. En ambos casos, por lo explicado con anterioridad acerca de la complejidad de los algoritmos de búsqueda, tenemos que la complejidad de esta parte del algorimo es $O(\log_\phi(n))$. Como tenemos n elementos, la complejidad del algoritmo First Fit usando árboles AVL es $O(n \log_\phi(n))$, como ya habíamos mencionado.

Best Fit y Worst Fit usando árboles AVL

Vamos a englobar estos dos algoritmos en una misma sección dado que su implementación solo se diferencia en un único método.

En ambos casos, la información que almacenaremos en cada nodo será:

1. La altura del nodo en el árbol.
2. Un cubo, el cual contiene los atributos del caso anterior.
3. Al igual que en el caso previo, una referencia a su hijo izquierdo y otra a su hijo derecho.

A diferencia del árbol que construimos para First Fit, en este tenemos que definir un orden diferente, el cual viene dado por la capacidad restante del cubo que hay en cada nodo. Esto es, por ejemplo, un nodo es menor que otro si la capacidad restante del cubo

del primer nodo es menor que la capacidad restante del cubo del segundo nodo.

Otra diferencia con respecto al algoritmo First Fit es cómo introducimos un nuevo elemento. En un principio usamos el mismo criterio que usan los métodos Best Fit y Worst Fit ya definidos. Por ejemplo, para el nuevo método Best Fit, si queremos introducir un nuevo elemento en alguno de los cubos de los nodos del árbol (si es que es posible), tendremos que buscar aquel nodo cuya capacidad restante del cubo sea la menor de todas aquellas que sean mayores o iguales que el peso del elemento dado. Análogamente para el método Worst Fit. La peculiaridad se encuentra en cómo hacemos esa inserción una vez localizado el cubo, porque recordemos que luego tendremos que reordenar los nodos del árbol.

Y es que, lo que hacemos, es guardar el cubo del nodo localizado con el nuevo elemento añadido, borrar dicho nodo y hacer una inserción ordenada del cubo anterior, lo cual se traducirá en crear un nuevo nodo en la posición que le corresponda dentro del árbol que contenga a dicho cubo.

Observemos también que esto último significa que, a diferencia del First Fit en el que solo es necesario hacer rotaciones a la izquierda, aquí tendremos que hacer rotaciones simples tanto a la izquierda como a la derecha. Para ello, como veremos en el capítulo siguiente, implementaremos un método general que realice el balanceo del árbol haciendo uso de los métodos que se encarguen de hacer las rotaciones simples a la derecha o a la izquierda.

De este modo, los algoritmos Best First y Worst Fit comparten los siguientes métodos:

1. **insert(bin: Bin): Unit**: Con este método creamos un nuevo nodo dentro del árbol que contenga al cubo que le indicamos. El pseudocódigo es el siguiente:

```
def insert(bin: Bin): Unit
//insertRec es un método recursivo con el que recorremos los nodos del árbol en orden
//para encontrar el nodo en cuyo hijo crearemos el nodo nuevo con el cubo
def insertRec(node: Node): Node
if node es null then
    //Esto quiere decir que el nodo en el que nos encontramos está vacío y solo ocurrirá
    //cuando lleguemos a la posición donde tenemos que crear el nuevo nodo
    Creamos el nuevo nodo con el cubo bin introducido
else
    //getLeftCapacity es un método que nos devuelve la capacidad restante de un cubo
    cmp = bin.getLeftCapacity - node.bin.getLeftCapacity
    if (cmp < 0) then
        //Esta condición significa que la capacidad restante del cubo a introducir es
        //menor que la capacidad restante del cubo del nodo actual
        //Entonces compruebo el hijo izquierdo
        node.left = insertRec(node.left)
    else
        //La capacidad restante del cubo a introducir es mayor o igual que la del cubo
        //del nodo actual
        //Compruebo entonces el hijo derecho
        node.rigth = insertRec(node.right)
    end if
    //El método balance realiza el balanceo del árbol, haciendo las rotaciones y
    //reajustando las alturas de los nodos
    node.balance
end if
//Empezamos comprobando si podemos insertar el cubo desde la raíz
root = insertRec(root)
```

2. **add(initialCapacity: Int, weight: Int): Unit**: Dada la capacidad de los cubos y el peso de un objeto a introducir en los mismos, este método es el que se encarga de introducirlo en un cubo de algún nodo del árbol. Su funcionamiento ya la describimos cuando se explicó cómo se introducía un nuevo elemento en el árbol. Esto es, primero comprobamos si el elemento cabe en el cubo que almacena algún nodo. Si no cabe en ninguno, creo un nuevo cubo al que le añado el elemento e inserto este cubo en el árbol haciendo una llamada al método anterior pasándole como parámetro este cubo. Si cabe en alguno, copio ese cubo, introduzco el elemento y borro el nodo para, a continuación, llamar de nuevo al método anterior pasándole como parámetro este cubo.

   Así el pseudocódigo sería:

```
def addl(initialCapacity: Int, weight: Int): Unit
//auxBin es una variable auxiliar que me guarda el cubo en el que cabe el elemento si
//lo encuentra o un valor None
auxBin: Option[Bin] = delete(weight)
if (auxBin está vacío) then
    //Creo un cubo con la capacidad inicial dada
    bin = new Bin(initialCapacity)
    Añado el peso del elemento al cubo
    insert(bin)
else
    Le añado el elemento al cubo que hemos copiado y cuyo nodo hemos borrado
    Insertamos de nuevo el cubo en el árbol
end if
```

3. **addAll(): Unit**: Este método es exactamente el mismo que el implementado en First Fit, ya que su única función es, dada una instancia del problema, ir recorriendo la lista de objetos a introducir en la instancia para añadirlos en cubos con la capacidad dada.

Notemos que en estos métodos ya hemos usado indirectamente a aquel que hace que el método Best Fit y Worst Fit sean diferentes, que es la función que borra los nodos que contienen los cubos en donde vamos a introducir un nuevo elemento (método delete que hemos usado en la función add anterior). Esto es así porque estos métodos incorporan los algoritmos de búsqueda de los que hacen uso Best Fit y Worst Fit.

NO SE SI AQUI DEBERIA EXPLICAR LOS DOS METODOS DELETE JUNTO CON SU PSEUDOCODIGO O DEJARLOS PARA EL CAPITULO SIGUIENTE

2.4. Algoritmos evolutivos

Los algoritmos evolutivos son una serie de algoritmos metaheurísticos inspirados en el proceso evolutivo. Por lo tanto, la idea detrás de ellos es sencilla de comprender: partiendo de una población conocida, obtenemos mediante reproducción y selección natural un individuo "mejor". A continuación, este nuevo individuo se incorpora a la población de partida y seguimos obteniendo nuevos individuos mediante los mecanismos anteriores y así, sucesivamente. Evidentemente estos nuevos individuos no tienen por qué ser necesariamente mejores, aunque evidentemente trataremos que así sea. Posteriormente, tendremos que especificar cuáles son estos mecanismos que simulan los procesos de reproducción y selección.

MENCIONAR TFG VIGO

De esta forma, el algoritmo evolutivo funcionaría como describimos a continuación:

0. Antes de arrancar propiamente el algoritmo evolutivo, tenemos que hacer un paso previo que se corresponde con la generación de la población a partir de la cual vamos a ir consiguiendo las posteriores generaciones de individuos (o poblaciones). Esta

población tiene un tamaño prefijado y se encuentra ordenada de mejor a peor, como veremos más adelante.

1. **Fase de selección**. Aquí entramos en el ciclo del algoritmo evolutivo. Aparte de los criterios que establezcamos para que se vayan repitiendo el ciclo, deberemos fijar un tiempo de parada para que el algoritmo finalice una vez se haya alcanzado ese tiempo. Existen diversos métodos de selección, como el método de la ruleta, el método de la ruleta extendido con rangos o el método de selección por torneo. Precisamente será este último el que usaremos para implementar nuestro algoritmo evolutivo. Será mediante estos métodos con los que elegiremos los padres a partir de los cuales obtendremos los nuevos individuos que iremos incorporando a la población.

2. **Fase de cruce**. Tanto esta fase como la siguiente, al igual que ocurre en la naturaleza, son dos procesos que no necesariamente tienen que ocurrir, sino que dependen de una cierta probabilidad prefijada. En el caso de que ocurra, lo que el algoritmo realiza es un cruce entre los padres que hemos obtenidos en la selección anterior para obtener nuevos individuos. Como operaciones de cruce podemos mencionar el cruce de un punto, cruce en dos puntos, cruce uniforme, cruce aritmético, cruce cíclico, el cruce PMX.

3. **Fase de mutación**. Con una cierta probabilidad, los nuevos individuos se someten a alteraciones en su *genoma* según algún patrón. Como operaciones de mutación, podemos citar la mutación por intercambio repetido (que es la que usaremos) o la mutación uniforme.

4. **Fase de ordenación**. Recordemos que la población tiene un tamaño prefijado. Por tanto, en esta fase lo que haremos será evaluar las soluciones que nos proporcionan estos nuevos individuos generados en la iteración actual del algoritmo y sustituir los peores individuos de la población por estos nuevos, de manera que el tamaño de la población siempre sea el mismo y sus individuos se estén actualizando en cada iteración.

5. Terminamos la iteración y volvemos a la fase 1.

Para que el algoritmo quede completamente determinado, aparte de los métodos y variables ya mencionados, tendremos que prefijar el tamaño de la población, el método con el que evaluaremos cómo de bueno es un individuo, las probabilidades de cruce y de mutación y una fuente de aleatoriedad.

En los algoritmos que hemos visto hasta ahora, dada una instancia del problema de BP, construíamos una solución introduciendo cada elemento de la instancia según las reglas que vienen dadas por los algoritmos. Así, las mejoras que hemos ido proponiendo, tenían la finalidad de reducir la complejidad de los algoritmos, pero no mejoraban propiamente la solución. Es decir, por ejemplo, el algoritmo First Fit que usa la implementación de árboles AVL proporciona, en esencia, la "misma" solución que el primer algoritmo First Fit, con lo que aunque hayamos mejorado su eficiencia, no hemos mejorado mucho su optimalidad.

El algoritmo evolutivo pretende compensar este hecho. En nuestro caso, el tipo de algoritmo que implementaremos no constituirá una mejora en cuanto a la eficiencia de los algoritmos anteriores puesto que, de hecho, fijaremos un tiempo durante el cual el

algoritmo se esté ejecutando. Tiempo que no tiene por qué ser necesariamente menor del que necesitan os algoritmos anteriores para ejecutarse. En cambio, sí que supondrá una mejora en cuanto a la calidad de las soluciones.

Para ello, los individuos que formarán la población y aquellos nuevos individuos que iremos generando en cada iteración del algoritmo, no serán más que permutaciones de los elementos de la lista de objetos dada en una instancia determinada del problema de BP. Por lo tanto, cada individuo de la población tendrá dos atributos, que serán la permutación correspondiente y el número de cubos que se obtiene al resolver la instancia del problema mediante el método con el que dijimos que evaluaríamos la calidad de los individuos, Este método será también el que nos permita tener ordenados a los individuos de la población de mejor a peor candidato, ya que esta condición es equivalente a ordenar a los individuos según el número de cubos que produce su solución, de menor a mayor cantidad.

De forma esquemática, el algoritmo evolutivo seguiría el siguiente diagrama:

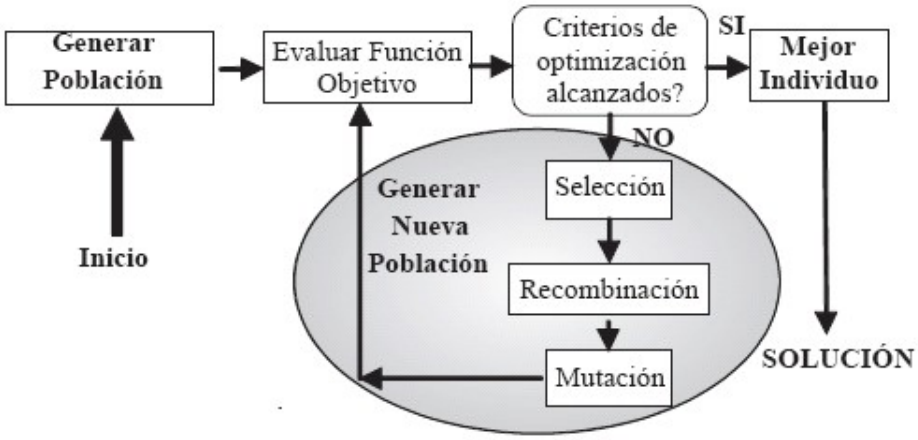

Figura 2.10: Diagrama de flujo de un algoritmo evolutivo.

Así, el pseudocódigo del algoritmo evolutivo quedaría como:

begin
Creamos la población inicial.
Evaluamos la población inicial.
//La condición de parada del bucle while es que alcancemos la SO o que se agote el tiempo.
while (no se verifique la condición de parada) **do**
    Seleccionamos aleatoriamente los padres en la población.
    Cruzamos con cierta probabilidad a los padres para obtener a los hijos.
    Mutamos los descendientes con cierta probabilidad.
    Evaluamos los nuevos individuos generados.
    Sustituimos los peores individuos de la población por los nuevos individuos que hemos generado.
end while

Capítulo 3

Implementación

1. Introducción

En este capítulo vamos a explicar brevemente cómo está estructurada la solución que se propone en este trabajo. En el capítulo anterior hemos explicado los algoritmos que hemos implementado para dar distintas soluciones a un problema de tipo BP por lo que, a continuación, lo que haremos será explicar las clases y métodos que hemos implementado para poder llevar a cabo los algoritmos anteriores. En el siguiente enlace podemos consultar el código completo:

`https://github.com/RafaelRG94/BinPackingProblem`

2. Preparación del problema

En esta primera parte vamos a explicar qué clases hemos implementado para introducir los datos y prepararlos para su posterior uso en los algoritmos que hemos presentado previamente.

2.1. La clase Bin

Lgícamente este es el primer fichero que hemos tenido que implementar. Con él hemos creado un objeto con el que representar la información que guarda un cubo del problema así como los métodos principales para interactuar con el mismo.

En primer lugar tenemos dos únicos atributos, a saber: *items*, que es un array de tipo buffer y es la variable donde almacenaremos los elementos que se vayan introduciendo en el cubo y *leftCapacity*, que almacena un entero cuya clase recibe como constructor y no es más que la capacidad restante del cubo. El motivo por el que se elige el arraybuffer para almacenar los objetos que se van introduciendo es que es una estructura de datos mutable (que es lo que nos permite ir introduciendo nuevos elemento en el cubo) que comparte mucho de los métodos que los arrays tienen predefinidos.

Por otra parte, podemos encontrar los siguientes métodos principales:

- **add**: Este método toma como parámetro un entero y lo introduce en el cubo.

- **canAdd**: Dado un entero, esta función comprueba si el elemento se puede introducir en el cubo.
- **getLeftCapacity**: Con este método podemos consultar la capacidad restante de la instancia del cubo que estemos interesados.

2.2. Las clases ProblemInstance y Solution

Estas dos clases las usamos para crear objetos que representen tanto la instancia de un problema de BP como su solución. Ambos implementan un método **toString** para posteriormente mostrar distintos datos por pantalla. Además, podemos encontrar los dos siguientes métodos:

- **sortDescendingInstance**: Método de la clase ProblemInstance con el que ordenamos de manera descendente la lista de objetos que tenemos en la instancia de un problema BP. Este método es necesario para su posterior uso en los algoritmos de tipo decreasing. Su implementación es muy sencilla dado que lo único que hacemos es copiar la lista de objetos de la instancia del problema, ordenarla y devolver un objeto de tipo ProblemInstance que tenga a esta última lista como objetos a introducir en los cubos.
- **length**: Método de la clase Solution con el que devolvemos la longitud de la solución de una instancia del problema.

2.3. La clase Utils

En el fichero Utils.scala tenemos tres métodos auxiliares de los que posteriormente hacemos uso cuando implementamos los algoritmos:

- **smallerThanTarget**: Se trata de un método de búsqueda binaria [**?**] con el que, dado un entero y un arraybuffer de cubos ordenados según sus capacidades restantes de forma creciente, encuentra la posición del cubo con la menor capacidad restante que es mayor o igual que el entero dado.
- **reorderBufferArrays**: Este método toma como parámetros un arraybuffer de cubos que se encuentran ordenados según sus capacidades restantes y la posición de un elemento del array que se encuentra desordenado y, lo que realiza, es la reordenación de dicho elemento. Su implementación es sencilla dado que, lo que hacemos, es usar el método anterior para encontrar la nueva posición de dicho cubo en el array para luego guardar el cubo en una variable auxiliar y, haciendo uso de que los arraybuffer son estructuras de datos mutables, usar los métodos predefinidos de las librerías de Scala para borrar el elemento del array y reinsertarlo en la posición que habíamos calculado previamente.
- **sortDescending**: Este es un método de ordenación que usamos para ordenar de manera descendente los elementos de la lista de objetos que nos proporciona las distintas instancias del problema. Básicamente hacemos uso del algoritmo QuickSort para realizar tal ordenación [**?**].

3. Algoritmos voraces

Esta parte del código se corresponde con aquellas clases que implementan los algoritmos de tipo greedy. Recordemos que la mayoría de algoritmos propuestos tienen dos implementaciones diferentes en función de qué tipo de estructura de datos usemos para almacenar la solución que vamos calculando. Así, tenemos:

- Ficheros de los algoritmos que almacenan sus soluciones en un **arraybuffer**: .

  1. AlmostWorstFit.
  2. BestFit.
  3. BestFitDecreasing.
  4. FirstFit.
  5. FirstFitDecreasing.
  6. NextFit.
  7. NextKFit.
  8. WorstFit.
  9. WorstFitDecreasing.

  Estos ficheros contienen los métodos de la clase de los objetos que, dada una instancia del problema de BP, resuelven el problema según los algoritmos que hemos descrito en la primera parte del segundo capítulo, almacenando cada nuevo cubo que se va creando en las iteraciones en un arraybuffer.

- Ficheros de los algoritmos que almacenan sus soluciones en un **árbol AVL**:

  1. BFAVLTree.
  2. BFDAVLTree.
  3. FFAVLTree.
  4. FFDAVLTree.
  5. WFAVLTree.
  6. WFDAVLTree.

  Estos ficheros contienen los métodos necesarios para implementar los algoritmos Best Fit, First Fit, Worst Fit y sus respectivos algoritmos de tipo decreasing almacenando los cubos de la solución en árboles AVL.

Como ya mencionamos en el capítulo anterior, los métodos necesarios para implementar los algoritmos Best Fit y Worst Fit mediante árboles AVL son prácticamente iguales, cambiando únicamente el método **delete** de ambos. Por tanto, para tener un código más limpio, se ha creado la clase abstracta AVLTree para implementar todos los métodos de los que hacen uso ambos alogirtmos. Posteriormente, las clases BFAVLTree y WFAVLTree heredan de la clase abstracta anterior para implementar el método delete.

EXPLICO AQUI LAS DIFERENCIAS DE LOS METODOS DELETE?

4. El algoritmo evolutivo

Recordemos que el algoritmo evolutivo va realizando iteraciones hasta que alcanza una condición de parada. En nuestro caso, dicha condición de parada es que se esté ejecutando durante el tiempo que le indiquemos. Para ello, la primera clase que implementamos es la clase **Timer**. Dicha clase contiene los métodos necesarios para obtener el tiempo que marca el reloj interno del ordenador y así, poder establecer un instante inicial y ser capaces de medir intervalos de tiempo [**?**].

4.1. Clases para generar la población inicial

Individual

Los objetos de esta clase pretenden representar a los individuos que forman a la población. Por otro lado, como el objetivo del algoritmo evolutivo es encontrar la permutación de la lista de objetos dada en una instancia del problema de BP con la que obtenemos la mejor solución del problema según un algoritmo concreto, la información que almacenan los objetos de la clase es la permutación correspondiente de la lista de objetos a introducir en los cubos, el número de cubos que obtenemos cuando resolvemos el problema según un algoritmo dado y los métodos necesarios para interactuar con estos datos.

Populations

El fichero **Populations.scala** contiene un objeto de tipo Populations con los métodos auxiliares necesarios para luego crear distintos tipos de poblaciones. Recordemos que la población del algoritmo evolutivo no solo contiene las distintas permutaciones de la lista de objetos dada en la instancia del problema, sino también el número de cubos que obtenemos cuando resolvemos el problema para las distintas permutaciones según el método que le indicamos para, posteriormente, ordenar esas permutaciones de mejor a peor según el número de cubos obtenidos. Por lo tanto, podemos tener distintos algoritmos evolutivos según el método con el que decidamos resolver las distintas instancias que nos porporcionan los individuos de la población.

En este sentido, en Populations podemos identificar dos partes claramente diferenciadas:

- **initPopulation**: Con este método, dada una instancia del problema de BP y una función *Solver* (es decir, una función con la que resolvemos la instancia del problema), generamos una población (que no es más que un array de individuos) para el algoritmo evolutivo. Para ello, lo que hacemos es que vamos generando permutaciones de la lista de objetos de la instancia del problema de BP mediante el algoritmo de Fisher-Yates (función shuffle), luego resolvemos la permutación obtenida con el Solver seleccionado y finalmente ordenamos a los individuos de mejor a peor. La caracterśtica fundamental por la que elegimos el algoritmo de Fisher-Yates es que genera cada permutación según una distribución uniforme.

- Métodos con los que generamos distintas poblaciones según los *Solvers* de los que disponemos (indicamos únicamente que algoritmo usan para resolver las instancias):

    1. **fFPopulation**: First Fit.

2. **bFPopulation**: Best Fit.
3. **wFPopulation**: Worst Fit.
4. **aWFPopulation**: Almost Worst Fit.
5. **fFAVLTreePopulation**: First Fit (según el esquema de árbol AVL).
6. **bFAVLTreePopulation**: Best Fit (según el esquema de árbol AVL).
7. **wFAVLTreePopulation**: Worst Fit (según el esquema de árbol AVL).

A continuación, vamos a probar lo afirmado sobre el algoritmo de Fisher-Yates [**?**]

Lema 3.1. *(Buen funcionamiento del algoritmo Fisher-Yates)*
Sea $v = (v_1, ..., v_n)$ *un vector con* $\{v_i\}_{i=1}^n = \{1, ..., n\}$*, esto es, sus componentes son los elementos 1,...,n permutados. Sean* $i \in \{1, ..., n\}$ *y* $k \in \mathbb{N}^+$*. Definimos la variable aleatoria* X_i^k *como la nueva posición del elemento i-ésimo después de la k-ésima permutación mediante el algoritmo Fisher-Yates (de izquierda a derecha). Entonces* $X_i^k \sim U(1, ..., n)$*.*

Demostración. Nuestro objetivo es el de determinar que, para todos i y k, la posibilidad de que el elemento i-ésimo caiga en cualquier casilla sea $\frac{1}{n}$. Vamos a hacerlo por inducción sobre k.

- Caso $k = 1$. Aprovechamos la recursividad del algoritmo para demostrar que $X_i^k \sim U(1, ..., n)$.

  Para $n = 1$ no hay nada que probar, y para $n = 2$, notamos que el bucle principal tiene una única iteración. Supuesto que j_1 puede valer 1 con probabilidad $\frac{1}{2}$ o 2 con probabilidad $\frac{1}{2}$, $X_1^1, X_1^2 \sim U(1, 2)$ como queríamos ver.

  Suponemos ahora que el resultado es cierto para n y veamos que se cumple para $n+1$. Para aprovechar la hipótesis de inducción, discriminamos la posición 1 para aplicar dicha hipótesis sobre el resto de elementos. Si en la primera iteración $j_1 = 1$, entonces v_1 permanece en la casilla 1. La posición 1 no se ve afectada por las iteraciones siguientes, de manera que v_1 permanecerá en la posición 1 acabada la permutación. Por el contrario, si en la primera iteración $j_1 \neq 1$, al finalizar ésta el elemento v_1 se encontrará en una posición distinta de 1, a la que no podrá regresar. El valor de j_1 determina la posición de v_1. Notamos por A el suceso "$j_1 = 1$" Entonces:

  $$P(X_1^1 = 1) = P(A) = \frac{1}{n+1}$$

  .

  Si vemos también que, dado $x \neq 1, P(X_1^1 = x) = \frac{1}{n+1}$, habremos acabado la inducción (para $k = 1$). Para que v_1 acabe en la posición x, después de ser colocado en la posición j_1 tras la primera iteración, solo queda que el elemento ahora en posición j_1 acabe en la casilla x, a lo que llamamos suceso B. Ahora bien, dejando de lado la primera posición, podemos pensar en el resto del vector como un vector de tamaño n sobre el que se puede emplear la hipótesis de inducción. En tales circunstancias, la probabilidad de que el elemento en posición j_1 acabe en la posición x es $\frac{1}{n}$.

  $$P(X_i^1 = x) = P(\bar{A} \cap B) = P(B|\bar{A})P(\bar{A}) = \frac{1}{n}\frac{n}{n+1}$$

  .

- Caso $k+1$ y supuesto cierto para k. Veamos que $X_i^{k+1} \sim U(1, ..., n)$. Consideremos entonces $m \in \{1, ..., n\}$ y veamos que $P(X_i^{k+1} = m) = \frac{1}{n}$.

$$P(X_i^{k+1} = m) = \sum_{j=1}^{n} P(X_j^{k+1} = m | X_i^k = j) P(X_i^k = j) = \sum_{j=1}^{n} \frac{1}{n} \frac{1}{n} = \frac{1}{n}$$

.

En la primera igualdad hemos hecho uso del teorema de la probabilidad total y en la segunda de dos cosas: la probabilidad condicionada coincide con la probabilidad asociada a permutar los elementos una vez y puede aplicarse el caso $k = 1$; en la otra probabilidad hemos empleado la hipótesis de inducción.

□

Population

El fichero Population.scala consta de una clase abstracta Population que implementa los métodos necesarios a la hora de interactuar con los distintos tipos de poblaciones. Los más importantes:

1. **binaryTournament**: Método que implementa el algoritmo del torneo binario con el cual seleccionamos un individuo de la población aleatoriamente.

2. **binaryInsertion**: Método de inserción binaria con el que, dado un nuevo individuo, lo sustituimos por uno de los que se encuentran en la población para insertarlo.

Además, dentro del fichero tenemos definidas otras clases que extienden a la clase abstracta anterior y que hacen uso de los métodos del objeto Populations para así definir las distintas poblaciones según el *Solver* elegido para resolver la instancia del problema de BP.

4.2. Clases para los cruces y mutaciones

Por una parte tenemos la clase **Crossover**, la cual implementa dos algoritmos de cruce distintos:

1. Algoritmo de cruce según el esquema PMX (Partially Mapped Crossover): **pmx**.

2. Algoritmo de cruce por vectores de inversión: **inversionCrossover**. Se define como vector de inversiones de una permutación $(\pi(i))_{i=1}^n$ un vector $(a_i)_{i=1}^n$ donde a_i indica el número de elementos situados a la izquierda de i que son mayores que i [**?**].

Para más información acerca de estos esquemas así como del código en el que nos hemos basado para desarrollar los algoritmos que usamos, consultar [**?**].

Por otra parte, en la clase **Mutation** tenemos los siguientes algoritmos para representar una mutación en los individuos de la población (notemos que cuando hablemos de los *genes* nos referiremos a los elementos de las permutaciones de la lista de objetos de la instancia del problema de BP) [**?**]:

1. **swapMutation**: Los genes de los extremos izquierdo y derecho intercambian sus posiciones.

2. **insertMutation**: El gen situado en el extremo derecho se coloca yuxtapuesto a la derecha del gen del extremo izquierdo y desplaza a la derecha todos los que se encontraban en la franja central.

3. **scrambleMutation**: Se permutan los genes de la franja central.

4. **inversionMutation**: Se invierte el orden de los genes de la franja central.

Finalmente tenemos la clase **EvolutionaryAlgorithm** la cual, siguiendo el esquema descrito en el capítulo anterior, hace uso de las clases que acabamos de describir y sus métodos para implementar el método **evolve**, el cual resuelve el algoritmo evolutivo.

Capítulo 4

Conclusiones

La finalidad de este trabajo es la de, a través de un problema de optimización combinatoria concreto, pasar del marco teórico de las matemáticas al práctico. Para ello, hemos hecho uso de la potencia de cómputo que nos ofrecen los ordenadores, los cuales nos permiten dar mejores soluciones para este tipo de problemas a medida que la tecnología avanza. Este hecho ha quedado reflejado, sobre todo, en el algoritmo evolutivo, ya que cuanto más potente sea la máquina de la que disponemos, más iteraciones podremos realizar en el mismo intervalo de tiempo.

En realidad, todo lo que se ha expuesto a lo largo del trabajo ha sido para concurrir en el algoritmo evolutivo y ver la cantidad de posibilidades que ofrece. Hemos comenzado explicando una de las versiones más simples del problema de empaquetamiento; posteriormente, hemos presentado una serie de algoritmos que siguen el esquema de los métodos voraces para resolverlo, es decir, el de aquellos métodos que no se replantean las decisiones ya tomadas. A su vez, hemos comprobado que cuanto mejor elección quisieramos hacer a la hora de introducir un elemento en uno de los cubos, los algoritmos resultantes son cada vez menos eficientes.

A continuación, hemos conseguido una mejora significativa en las complejidades de los algotirmos anteriores haciendo uso de la estructura de árboles AVL para almacenar la solución de las instancias del problema. Esto implica, a su vez, que la implementación del algoritmo sea más compleja.

Una vez hecho todo este trabajo, hemos podido pasar propiamente a la implementación del algoritmo evolutivo, pues su funcionamiento se basa en cualquiera de los métodos presentados hasta ahora. De esta forma, vemos que podemos tener un algoritmo evolutivo diferente por cada uno de estos métodos. Además, si tenemos en cuenta los diferentes algoritmos de selección, de cruce y de mutación que hay, vemos que podemos encontrar multitud de combinaciones de todos estos métodos que dan distintos algoritmos evolutivos, lo cual pone de manifiesto la cantidad de opciones existentes a la hora de obtener una "buena" solución de un problema dado en un tiempo "aceptable".

Como trabajo futuro, queda pendiente el comparar y analizar cómo cada uno de estos métodos resuelven distintas instancias cada vez mayores del problema de empaquetamiento para así verificar que, lo que obtenemos, se ajusta a los resultados y conclusiones que aquí se han expuesto.

No obstante, queda también claro la necesidad de encontrar mejores algoritmos que nos permitan resolver problemas cuyo espacio de búsqueda de soluciones sea muy grande en tiempos razonables y no solo en este tipo de problemas. En general, a medida que la tecnología avanza y con ello nuestra capacidad de generar y recopilar datos, se hace más y más necesario que nuestra capacidad de procesar esos datos, analizarlos y obtener conclusiones y resultados a partir de ellos sea cada vez mejor. Es gracias a ello que, al igual que nuestro algoritmo evolutivo, la especie humana consigue adaptarse, mejorar y evolucionar.

Bibliografía

[1] José Enrique Gallardo Ruiz. Hybridization of exact and metaheuristics techniques for the resolution of combinatorial optimization problems. *PhD thesis, Universidad de Málaga*, 2007.