Título del tfg en español

Título del tfg en inglés

Trabajo Fin de Grado en Matemáticas
Universidad de Málaga

Autor: Rafael Requena Garrido

Área de conocimiento y/o departamento:

Fecha de presentación: (mes y año)

Tema:

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

Modalidad: (individual o grupal)

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

DECLARACIÓN DE ORIGINALIDAD DEL TFG

D./Dña. *(nombre del autor)*, con DNI (NIE o pasaporte) *(DNI, NIE o pasaporte)*, estudiante del Grado en *(titulación)* de la Facultad de Ciencias de la Universidad de Málaga, **DECLARO:**

Que he realizado el Trabajo Fin de Grado titulado "*(Título)*" 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 *(fecha)*

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.

Introducción

Capítulo 1

Optimizaiǿn 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 $x =$ número de mesas M_1 e $y =$ 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 mamximice 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) dinde:

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 Conveza*, 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 búsquedas.
- 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 búsquedas, 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 tiempo 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 hasta los *algoritmos genéticos* (de tipo metaheurístico), que se enmarcan dentro de los *(*algoritmos evolutivos), diendo estos dos últimos muy empleados en el áre 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 aproxiamadas 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

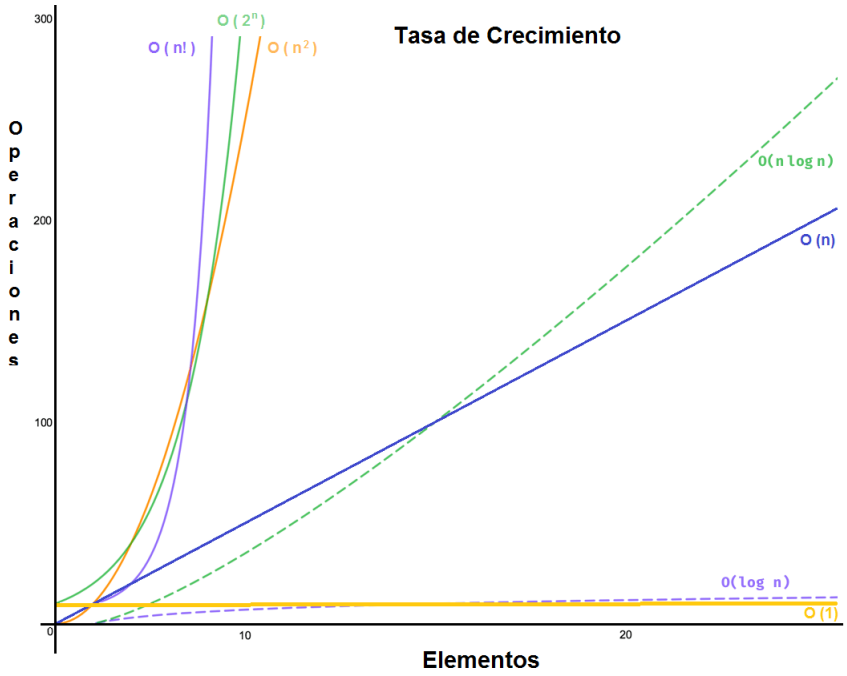

Figura 1.1: Formas de crecimiento asintótico más importantes

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 expresar mediante una fórmula polinómica, se dice que dicho algoritmo y su problema asociado son de *tiempo exponencial*. Cuando un problema solo se puede resolver mediante algoritmos de tiempos exponenciales, se dice que el problema es *intratable*.

Para el propósito de este trabajo, 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. Un problema de decisión puede considerarse como la definición de un conjunto de instancias en los que la respuesta correcta es sí.

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. La mayoría de 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-completo.*(**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 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 abieta 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-completo* 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 *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***.

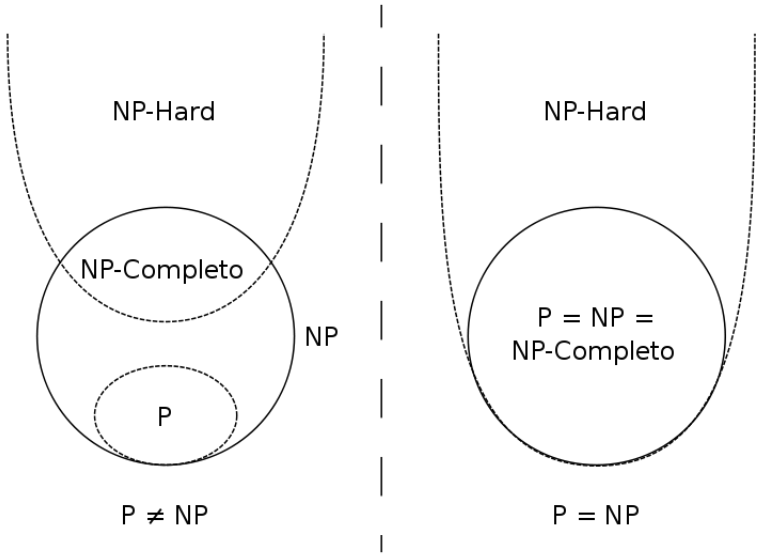

Figura 1.2: Diagrama de Euler de las clases de complejidad más frecuentes

3. Algoritmos heurísticos y metaheurísticos. Algoritmos evolutivos?

Como ya se mencionó con anterioridad, para muchos problemas de optimización combinatoria no se conocen algoritmos que sean capaces de obtener una solución 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. (**Ejemplo?**)
- **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 so-

luciones 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. (**Ejemplo?**)

- **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. (**Ejemplo?** - cualquier algoritmo iterativo?)

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 https://core.ac.uk/download/pdf/236383515.pdf**)

- **Comparación con la solución óptima.** Aunque normalmente se recurre al algoritmo aproximado por no existir un método exacto para obtener el ótimo, 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 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 tamaño c en donde se coloquen 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 y su volument, 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. Cuando $c_i = 1$ la función objetivo minimiza el número de elementos utilizados y, si larestricción de los elementos a producir se sustituye por la igualdad, obtenemos el BPP (**esta parte no la entiendo**). Así, procedemos a continuación a formular matemáticamente el problema BP.

Sea 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 a qué coste. Para ello, el primer tipo de métodos que analizaremos serán los algoritmos voraces (*greedy alhorithm*).

2. Algoritmos voraces

Por algoritmos voraces se entienden aquellos algoritmos que siguen un esquema de resolución llamado método voraz. Dicho esquema forma parte de una familia de algoritmos más amplia denominada *algoritmos de búsqueda local*, de la que también forman parte, 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 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
        Añadimo current a soltion
        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 la velocidad de 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.

QUEDA POR ANALIZAR SU APPROXIMATION RATIO

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 este será uno de los algortimos que usaremos como base para implmententar 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. Dado 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 $O(n)$.

QUEDA POR ANALIZAR SU APPROXIMATION RATIO

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 llena mejor en el cubo. Al igual que el 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 fit 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 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 aquel cubo donde se verifiquen 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 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
        Añadimo 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)
        Añadimo el item en el cubo current
        Añadimo 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
//smallerThanTarget es un método con el que, dado un array y un elemento, encuentra
la posición del menor elemento del array que es mayor o igual que el elemento dado
//Inicializamos el algoritmo con el array de soluciones sin ningún cubo
solution = ∅
for (item ← items) do
    targetBin = smallerThanTarget(item, solution)
    if (targetBin es menor que la longitud de solution) then
        //Esto equivale a comprobar que item cabe en el último cubo
        Añadimo 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)
        Añadimo el item en el cubo current
        Añadimo el cubo current en solution
        Reordenamos el array solution de menor a mayor capacidad restante
    end if
end for
return solution
```

QUEDA POR ANALIZAR LA COMPLEJIDAD DEL ALGORITMO

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. Subsección

Una sección dentro de una sección se denomina subsección.

Subsubsección

Esto es una sección dentro de una subsección, o sea, una subsubsección. Esto es un ejemplo de cita [1]

Bibliografía

[1] E. I. Zelmanov. Lie algebras with finite gradation. *Mat. Sb. (N.S.)*, 124(166)(3):353–392, 1984.