Algoritmia y optimización

Jorge Calvo Zaragoza

Francisco J. Castellanos Regalado

Juan C. Martínez Sevilla

Eric Ayllón Palazón

Universidad de Alicante

2025

Prefacio

El objetivo de este documento es dotar a los alumnos de la asignatura "Algoritmia y optimización" del Grado en Ingeniería en Inteligencia Artificial de la Universidad de Alicante de un material didáctico de consulta. Los contenidos del libro están alineados con los contenidos de la asignatura, aunque en ningún caso sustituyen las sesiones de teoría y prácticas del cuatrimestre.

Índice general

Capítulo 1

Introducción

La optimización es esencial en muchas disciplinas que requieren decisiones eficaces y eficientes, incluyendo disciplinas como la inteligencia artificial. Este campo de estudio trata sobre buscar la *mejor solución*, es decir, la *óptima*, dentro de un conjunto de posibles soluciones a un problema (a veces infinitas). En este sentido, existe el concepto de *optimalidad*, que se refiere a comprobar cada una de todas esas posibles soluciones con el objetivo de encontrar las soluciones óptimas según algún criterio, como maximizar algún beneficio, minimizar costes, o encontrar el mejor balance entre múltiples factores conflictivos, en los que optimizar uno de esos factores puede considerarse equivalente a perjudicar a otros factores, haciendo que deje de ser una solución óptima en su conjunto. A esto último, se le conoce como *optimización multi-objetivo*). En muchos contextos, tratar de optimizar la velocidad de un algoritmo puede provocar que empeore el rendimiento del mismo, mientras que tratar de mejorar el rendimiento puede llevar a una peor eficiencia. La optimización multi-objetivo se refiere a encontrar la solución óptima que mantenga un equilibrio entre ambos.

En informática, los problemas de optimización son omnipresentes, desde diseñar algoritmos que encuentren la ruta más corta en redes de datos hasta modificar los parámetros de una red neuronal para que se ajusten a un conjunto de datos de referencia, lo que se conoce como entrenamiento de un modelo de red neuronal. A lo largo del curso, exploraremos cómo se pueden formular y resolver algunos de estos problemas utilizando estrategias algorítmicas.

1.1. Definición de algoritmo

Un algoritmo comprende una serie finita de instrucciones claramente definidas que, cuando se siguen en orden, resuelven un problema particular. De modo simplista, un algoritmo es una *receta* que proporciona una solución a un problema.

El concepto de algoritmo es antiguo, precediendo por siglos la invención de los ordenadores. Los procedimientos para realizar cálculos matemáticos como multiplicaciones de varios dígitos, extracciones de raíces cuadradas o cálculos geométricos son ejemplos tempranos de algoritmos. En la era moderna, el enfoque se ha expandido para incluir soluciones basadas en programación.

1.2. ¿Por qué estudiar algoritmos?

Existen muchas razones para profundizar en el estudio de los algoritmos, cada una subrayando su relevancia y aplicabilidad en múltiples dominios, incluyendo aquellos más allá de la computación:

1. Los algoritmos son fundamentales para utilizar ordenadores en cualquier campo de las ciencias de la computación, y especialmente críticos en áreas como la inteligencia artificial, donde la capacidad de procesar y analizar grandes volúmenes de datos con eficiencia es esencial.

2. Comprender algoritmos existentes facilita el aprendizaje y la adaptación de técnicas complejas de manera más eficaz. Además, una habilidad algorítmica sólida abre la puerta a la innovación y al desarrollo de nuevos métodos para abordar problemas, además de identificar problemas que compartan solución. En este sentido, el conocimiento sobre un algoritmo eficaz para un problema específico podría facilitar la obtención de una solución para un nuevo problema similar al ya conocido.

3. La algoritmia es fundamental para desarrollar el pensamiento computacional, una habilidad clave que mejora nuestra capacidad de abstracción y la resolución de problemas.

4. Las competencias en algoritmia son a menudo evaluadas en las entrevistas de trabajo de las grandes compañías tecnológicas, donde los retos algorítmicos forman parte integral del proceso de selección.

Capítulo 2

Análisis de algoritmos

El *análisis de algoritmos* es uno de los pilares fundamentales en el campo de la algoritmia, proporcionando herramientas para evaluar y comparar algoritmos en condiciones teóricas y prácticas. Esta evaluación es crítica no solo para el desarrollo de software, sino también para la optimización de recursos en sistemas computacionales complejos.

El análisis de algoritmos se centra en medir la eficiencia con la que un algoritmo opera. La eficiencia puede medirse de múltiples formas, pero su objetivo principal es determinar qué tan bien se comporta un algoritmo bajo diversas condiciones con respecto a unos criterios definidos. Esto incluye identificar si un algoritmo es, en esencia, *eficiente* en términos de los recursos que consume, y si puede considerarse superior a otras alternativas para resolver el mismo problema. Entender cómo y por qué un algoritmo consume recursos específicos es esencial para mejorar y adaptar las soluciones a los requisitos cambiantes de la tecnología y las aplicaciones.

2.1. Complejidad algorítmica

Cuando hablamos de los recursos que consume un algoritmo, nos referimos a su *complejidad*. Tradicionalmente, el estudio de la complejidad algorítmica se centra en dos categorías: complejidad temporal y complejidad espacial, referidas a la cantidad de tiempo y de memoria que el algoritmo requiere, respectivamente. Aun así, puede haber otros recursos relevantes, como el número de accesos a una base de datos o la energía que consume ejecutar el algoritmo, entre otros.

La ejecución de un algoritmo y, por ende, su complejidad, pueden verse afectadas por factores tanto externos como internos. Entre los externos,

encontramos variables como la capacidad del hardware, que puede acelerar o retrasar la ejecución del algoritmo; el compilador, cuyas optimizaciones pueden reducir significativamente el número de operaciones necesarias; y la naturaleza de los datos de entrada. Internamente, la complejidad está influenciada únicamente por el número y tipo de instrucciones del algoritmo.

Siguiendo la idea anterior, vamos a distinguir dos tipos de análisis de complejidad:

- **Análisis empírico**: un análisis empírico consiste en ejecutar el algoritmo para distintos valores de entrada y medir directamente la cantidad de recursos utilizados. Por ejemplo, si nos referimos a recursos temporales, podemos cronometrar el tiempo de ejecución. La ventaja principal es que obtenemos una medida *real* del comportamiento del algoritmo en un entorno concreto. Sin embargo, este análisis puede verse afectado por cuestiones extrínsecas al propio algoritmo.
- **Análisis teórico**: un análisis teórico consiste en obtener una función matemática que represente la complejidad del algoritmo. Tiene la ventaja de que no necesita ejecutar el algoritmo y el resultado depende exclusivamente del diseño del mismo. Sin embargo, es difícil trasladar esta función a términos prácticos de ejecución en un entorno real.

Entre los dos tipos de análisis de complejidad, nos vamos a centrar en la teórica, que es la más común por su independencia con los dispositivos que vayan a ejecutarla. Ten en cuenta que un algoritmo podría ser más eficiente teóricamente que otro, pero si se utiliza un ordenador más potente para ejecutarlo, es posible que en términos de análisis empírico, el que parece menos eficiente teóricamente, resulte ser más eficiente. Sin embargo, esto ocurriría en desigualdad de condiciones, por lo que la complejidad teórica nos puede dar una medida más aproximada de la complejidad del algoritmo (y solo del algoritmo) sin tener en cuenta otros factores que puedan afectar a la eficiencia real, pero que no forman parte del algoritmo.

2.2. Complejidad teórica

El análisis de la complejidad teórica de un algoritmo se basa en la formulación de una función matemática que describe cómo varía el consumo de recursos en función del tamaño de la entrada. Esta medida abstracta proporciona una visión independiente del hardware y del entorno de ejecución, enfocándose en el diseño y la estructura del algoritmo.

Comencemos con un ejemplo. Consideremos el siguiente algoritmo:

```
función es_par(número):
    devuelve módulo(número,2) = 0
```

Se trata de un algoritmo muy simple que determina si un número (recibido por parámetro) es par o no. En el algoritmo se puede ver que devuelve el resultado de comparar el módulo o resto de la división entre el número y 2. Para simplificar, se suele considerar que el coste temporal de las operaciones elementales es unitario (1). En este caso, aunque solo se vea una instrucción a priori, en realidad se compone de varias instrucciones. Por tanto, la complejidad de este algoritmo es de 3: el operador módulo, la comparación con el valor 0 y retorno de la función.

En realidad, el ejemplo anterior no es muy interesante. Normalmente, la complejidad de un algoritmo se calcula en función de su *talla*. La talla de un algoritmo se refiere generalmente a la cantidad o el tamaño de los datos de entrada que el algoritmo debe procesar, la cual puede medirse en términos del número de elementos en una estructura de datos, como un vector o una lista, o bien en términos de algún otro parámetro relevante para el problema en cuestión. En general, es interesante calcular cuánto varía la complejidad de un algoritmo en función de su talla, lo que permite saber cómo escala a medida que se enfrenta a entradas de mayor tamaño. En el ejemplo anterior, la complejidad es 3 independientemente del número que reciba la función. Por lo tanto, se puede considerar que la complejidad es constante. Veamos otro ejemplo dependiente de la talla:

```
función acumulado(vec)
    suma := 0
    para i := 1 hasta |vec|
        suma += vec_i
    devuelve suma
```

En este caso, el algoritmo recibe un vector numérico y calcula la suma de todas los valores que lo componen. La talla del problema es el tamaño del vector `vec`.

Nota: *Por convención, usaremos la letra n para referirnos a la talla del problema y T para referirnos a la función que representa la complejidad del algoritmo.*

Por tanto, la complejidad de este algoritmo `acumulado` se representaría como $T(n) = 1 + 3n + 1 = 3n + 2$ (ver Figura 2.1). El primer 1 corresponde con la inicialización de la variable `suma`, el $3n$ representa que el coste que ocurre en cada iteración del bucle es 3, y que se repite n veces, coincidiendo

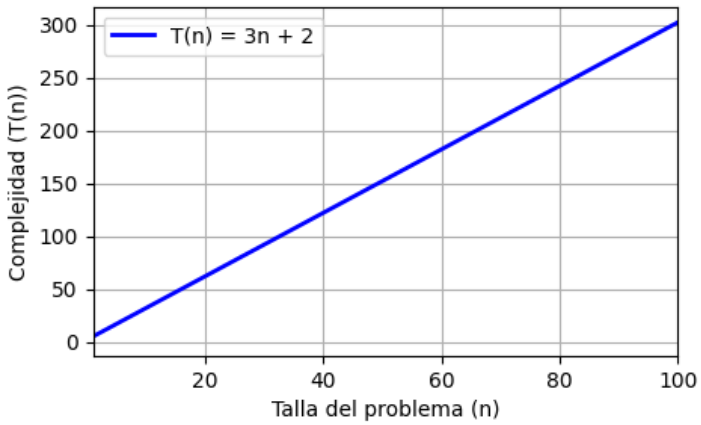


Figura 2.1: Complejidad de `acumulado` en función de la talla del problema.

con la talla del problema (o el tamaño del vector). El último valor 1 es el coste del retorno de la función.

Nota: *Cuando utilicemos los cálculos de complejidad para comparar algoritmos que resuelvan el mismo problema, es importante que en ambos casos la talla se refiera exactamente a la misma magnitud.*

2.2.1. Cotas de complejidad

En ocasiones, un algoritmo conlleva una complejidad diferente en función de variables que no dependen del diseño del algoritmo en sí, sino de los elementos que haya en la entrada. Consideremos el siguiente algoritmo, que busca un valor (`z`) en un vector (`v`) y devuelve su posición:

```
función buscar(v, z)
    para i desde 1 hasta |v|
        si v_i = z
            devuelve i
    devuelve NO_ENCONTRADO
```

En función de los valores concretos de `v` y `z`, podemos definir diferentes complejidades. Analicemos algunas posibilidades:

- El valor `z` es el primer elemento del vector: el interior del bucle solo se ejecutaría una vez. Por tanto, su complejidad podría definirse como $T(n) = 3$ (contando una iteración del bucle, la condición y el retorno).

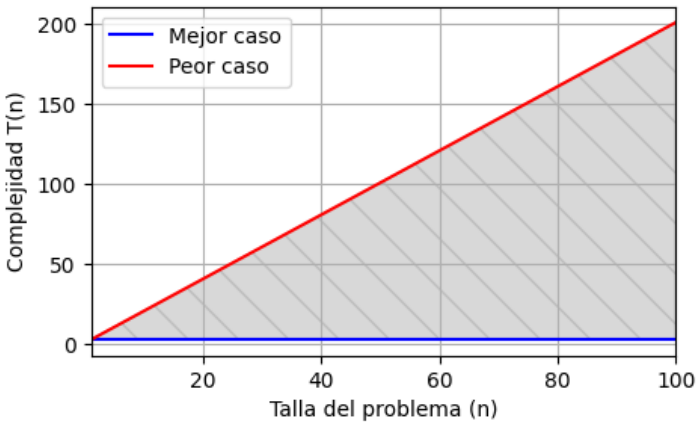


Figura 2.2: Cotas de complejidad de `buscar`.

- El valor `z` no se encuentra: el interior del bucle se ejecutaría n veces (siendo n el número de elementos del vector). Su complejidad se definiría como $T(n) = n(2)+1 = 2n+1$. Ten en cuenta que la instrucción *devuelve i* no se ejecutaría en este caso, y por lo tanto no provoca un coste.

Por tanto, ¿cuál es la complejidad de este algoritmo? Ante este tipo de situaciones, se estiman las llamadas *cotas de complejidad*, es decir, los valores extremos calculados en el mejor y en el peor de los casos. Por un lado, tendríamos la complejidad en el *mejor caso*, es decir, cuando se dan las condiciones más favorables y la complejidad en el *peor caso*, cuando se dan las condiciones más desfavorables. En lugar de obtener un único valor de complejidad, el análisis del algoritmo indiciaría en qué rangos puede encontrarse la complejidad del mismo. En resumen, diríamos que el algoritmo `buscar` tiene una complejidad entre $T(n) = 3$ y $T(n) = 2n + 1$ (ver Figura 2.2).

2.3. Notación asintótica

Para describir la complejidad de un algoritmo, especialmente en términos de su tiempo de ejecución, se utiliza frecuentemente la notación asintótica, conocida popularmente como "Big O" ($\mathcal{O}$). Esta notación proporciona un medio para representar los límites superiores de la complejidad de un algoritmo, enfocándose en cómo se comporta a medida que el tamaño de entrada crece hacia el infinito e ignorando costes superfluos. Este análisis es esencial

para entender el comportamiento del algoritmo en escenarios de gran escala, que es donde las diferencias en complejidad se vuelven más relevantes.

Formalmente, decimos que:

$$T(n) \in \mathcal{O}(g(n)) \iff \exists n_0, c > 0 : T(n) \leq c \cdot g(n) \, \forall n \geq n_0$$

En otras palabras, una función $T(n)$ pertenece al conjunto de funciones $\mathcal{O}(g(n))$ si existen constantes positivas c y n_0 tales que, a partir de $n \geq n_0$, la función $T(n)$ está acotada superiormente por $c \cdot g(n)$. En esta definición:

- $T(n)$ representa la función de complejidad de un algoritmo para un tamaño de entrada n.
- $g(n)$ es una función que proporciona una cota superior simple para $T(n)$ cuando $n \to \infty$.
- c es una constante positiva que escala la función $g(n)$ para que acote a $T(n)$.
- n_0 es el valor a partir del cual la cota se hace válida, indicando que la comparación solo es significativa para valores grandes de n.

La clave para entender la notación "Big O" es que abstrae el comportamiento de un algoritmo para entradas grandes, permitiendo centrarse en lo que realmente importa: cómo escala el algoritmo. Los factores constantes y los términos de menor orden, que también influyen en la complejidad práctica, se ignoran en esta notación, proporcionando una manera simplificada pero eficaz de analizar y comparar algoritmos en escenarios extremos. Por ejemplo, si tuviésemos un algoritmo con complejidad $n + 3$, el escalar 3 siempre se va a mantener igual independientemente de los datos de entrada; sin embargo, lo que realmente hace que el algoritmo pueda llegar a ser muy costoso (por ejemplo, que tarde más tiempo o consuma más recursos) únicamente depende n.

Para aclarar cómo las funciones concretas se clasifican bajo la notación asintótica, consideremos las categorías de complejidad más comunes:[1]

- Complejidad **constante** $\mathcal{O}(1)$: Independientemente del tamaño de la entrada n, la complejidad del algoritmo no cambia. Ejemplo: $T(n) = 3 \in \mathcal{O}(1)$.

[1] En el Apéndice A se muestran algunas demostraciones de pertenencia a órdenes de complejidad.

- Complejidad **logarítmica** $\mathcal{O}(\log n)$: La complejidad aumenta logarítmicamente con el tamaño de entrada. Ejemplo: $T(n) = 4 \log_{10} n + 6 \in \mathcal{O}(\log n)$.
- Complejidad **lineal** $\mathcal{O}(n)$: La complejidad aumenta directamente en proporción al tamaño de la entrada. Ejemplo: $T(n) = 2n + 3 \in \mathcal{O}(n)$.
- Complejidad **lineal-logarítmica** $\mathcal{O}(n \log n)$: Combina el crecimiento lineal y logarítmico de la complejidad. Ejemplo: $T(n) = 2n \log n + 4n + 3 \in \mathcal{O}(n \log n)$.
- Complejidad **cuadrática** $\mathcal{O}(n^2)$: La complejidad aumenta cuadráticamente en función del tamaño de la entrada. Ejemplo: $T(n) = n^2 + n \log n + n + 1 \in \mathcal{O}(n^2)$.
- Complejidad **exponencial** $\mathcal{O}(2^n)$: La complejidad crece exponencialmente, lo cual hace que el algoritmo sea ineficiente para tallas grandes. Ejemplo: $T(n) = 2^n + n^2 \in \mathcal{O}(2^n)$.

Como se puede deducir de algunos ejemplos anteriores, los órdenes de complejidad se pueden organizar jerárquicamente:

$$\mathcal{O}(1) \subset \mathcal{O}(\log n) \subset \mathcal{O}(n) \subset \mathcal{O}(n \log n) \subset \mathcal{O}(n^2) \subset \mathcal{O}(2^n)$$

Entender esta jerarquía ayuda a comparar la eficiencia de distintas alternativas algorítmicas rápidamente. En lo sucesivo, cuando hagamos el cálculo de la complejidad de un algoritmo, estaremos interesados en saber a qué orden de complejidad pertenece.

Antes de acabar la sección, conviene mencionar —aunque no vayamos a tratarlo en la asignatura— que la notación $\mathcal{O}$ es parte de un conjunto más amplio de notaciones que incluye Ω, que representa los límites inferiores, y Θ, que indica un límite estricto tanto superior como inferior.

2.4. Cálculo de complejidades

Calcular la complejidad de un algoritmo es esencial para entender su eficiencia y prever su comportamiento en diferentes escenarios de aplicación. Para realizar este cálculo de manera sistemática, se pueden seguir los siguientes pasos:

1. **Determinar la talla del problema:** Esto implica definir cuál es el tamaño de la entrada del algoritmo, que generalmente se denota como

n. Este tamaño puede representar distintos aspectos, como el número de elementos en una lista, el número de nodos en un grafo, o cualquier otra medida cuantitativa que influya en la ejecución del algoritmo.

2. **Determinar los casos mejor o peor:** Si existen, es crucial identificar el mejor y el peor caso de ejecución. El mejor caso representa la situación en la que el algoritmo realiza el menor gasto posible de recursos, mientras que el peor caso se refiere a la situación opuesta. Los algoritmos también podrían ser analizados bajo el *caso promedio*, que considera un rendimiento típico bajo una distribución estadística de todas las entradas posibles, pero no lo consideraremos en esta asignatura.

3. **Calcular la complejidad asintótica de cada caso:** Una vez identificados los casos relevantes, se analiza la complejidad con respecto a la talla del problema (n). Este análisis se realiza utilizando la notación asintótica para describir estos cambios de forma que se pueda entender el comportamiento del algoritmo en el límite de la talla.

Esta metodología sirve para abordar el cálculo de complejidad de diferentes tipos de algoritmos. En las siguientes secciones, nos centraremos en los algoritmos iterativos y los recursivos.

2.4.1. Complejidad de algoritmos iterativos

Los algoritmos iterativos se caracterizan por su uso de bucles, que repiten secuencias de operaciones hasta que se cumple una condición específica. La complejidad de estos algoritmos viene determinada por el número de veces que se ejecutan los bucles, que en la mayoría de los casos depende directa o indirectamente de la talla del problema.

Para analizar la complejidad de los algoritmos iterativos es importante identificar los bucles principales, esencialmente aquellos que dependen de la talla del problema. Esto incluye bucles anidados, que son particularmente críticos ya que el número total de iteraciones del bucle interno se multiplica por el número de iteraciones del bucle externo. A continuación, se debe prestar atención a cómo el número de iteraciones de cada bucle depende de la talla. Esto puede ser directo, como un bucle que recorre todos los elementos de un vector, o más complejo (como un recorrido no lineal). Por último, hay que identificar el coste de las operaciones realizadas en cada iteración del bucle. Esto puede variar desde una simple operación hasta llamadas a funciones más complejas.

Para ejemplificar cómo se calcula la complejidad en algoritmos iterativos, vamos a utilizar dos ejemplos: la "multiplicación de matrices" y la "ordenación por inserción".

Complejidad de "multiplicación de matrices"

La multiplicación de matrices es una operación fundamental en álgebra lineal que implica combinar dos matrices para producir una tercera matriz. Para multiplicar dos matrices, el número de columnas de la primera matriz debe ser igual al número de filas de la segunda matriz. El elemento en la fila i y columna j de la matriz resultante se calcula como la suma de los productos de los elementos correspondientes de la fila i de la primera matriz y la columna j de la segunda matriz.

Por simplicidad, vamos a asumir que el algoritmo solo multiplica matrices cuadradas. El siguiente código realiza el producto de dos matrices cuadradas A y B, y almacena el resultado en una matriz C:

```
función producto_matrices_cuadradas(A, B)
    n := dimensión(A)
    C := ceros(A)
        para i := 1 hasta n
            para j := 1 hasta n
                suma := 0
                para k := 1 hasta n
                    suma := suma + A_{i,k} * B_{k,j}
                C_{i,j} := suma
    devuelve C
```

Para analizar la complejidad del algoritmo comenzamos identificando la talla del problema. Aquí hay dos alternativas igualmente justificables: utilizar el tamaño de las matrices ($d = n \times n$) o simplemente el tamaño de cada dimensión (n). Dado que los bucles dependen de una única dimensión, parece más conveniente calcular la complejidad en función de n. Por otro lado, se puede observar que no hay caso mejor o peor, sino que el algoritmo ejecuta exactamente los mismos pasos independientemente del contenido de las matrices.

Seguiremos el proceso examinando los pasos del algoritmo, los bucles y las operaciones que se ejecutan dentro de estos. Si seguimos el algoritmo por bloques:

- Inicialización: la primera línea obtiene la dimensión de la matrices, con un coste constante $\mathcal{O}(1)$. A continuación, se inicializa una matriz C

del mismo tamaño que A y rellenada con ceros. Inicializar una matriz de tamaño $n \times n$ podría implicar un coste de orden $\mathcal{O}(n^2)$, ya que se establece cada elemento a cero.

- Bucles anidados: El código tiene tres bucles anidados, cada uno iterando n veces. Cada bucle corresponde a una dimensión: el bucle exterior recorre las filas de A, el bucle medio recorre las columnas de B, y el bucle interior realiza la multiplicación de elementos y suma para calcular un elemento $C_{i,j}$. Dentro del bucle más interno, la operación de suma se realiza n veces por cada combinación de i y j. Cada operación de este tipo implica una multiplicación y una suma, con un coste constante.

Teniendo en cuenta el desglose anterior, podríamos establecer una función de complejidad $T(n) = \mathcal{O}(1) + \mathcal{O}(n^2) + \mathcal{O}(n^3)$. Por lo tanto, la complejidad del algoritmo es de coste cúbico: $T(n) \in \mathcal{O}(n^3)$.

Complejidad de "ordenación por inserción"

Vamos a analizar ahora el algoritmo de "ordenación por inserción", uno de los métodos de ordenación fundamentales en las ciencias de la computación. Analizar algoritmos de ordenación es importante porque la ordenación es un problema ubicuo en la informática, aplicable en áreas que van desde la gestión de bases de datos hasta la inteligencia artificial, así como la optimización de otros algoritmos que requieren datos previamente ordenados para funcionar eficientemente. Además, desde el punto de vista de la algoritmia son interesantes porque existen multitud de alternativas, cada una con sus ventajas y desventajas.[2]

El algoritmo de "ordenación por inserción" simula el proceso de ordenar cartas en una mano. Comienza con un elemento, y en cada paso, toma el siguiente elemento y lo inserta en su posición correcta relativa dentro de la porción ya ordenada de la lista. Este proceso se repite hasta que todos los elementos están ordenados. El algoritmo se ejecuta en linea, lo que significa que no necesita un vector adicional para realizar la ordenación, y es *estable*, es decir, mantiene el orden relativo de los elementos que son iguales.

Consideremos el siguiente pseudocódigo:

```
función ordenación_por_inserción(v):
    para i := 2 hasta |v|:
        valor := v_i
```

[2]Los algoritmos de ordenación tendrán un papel fundamental en el Capítulo 3.

```
j := i - 1
mientras j > 0 y v_j > valor:
    v_{j+1} := v_j
    j := j - 1
v_{j+1} := valor
```

Para calcular su complejidad, comenzamos identificando la talla del problema. En este caso, la talla viene representada por el tamaño del vector a ordenar (n). Después observamos que el algoritmo de ordenación por inserción sí presenta distintos casos:

- Mejor caso: ocurre cuando el vector ya está ordenado. En este caso, dentro del bucle interior, $\texttt{v}_\texttt{j}$ siempre es menor o igual que `valor`. Por lo tanto, la complejidad en el mejor caso se concentra en el bucle exterior, siendo por tanto del orden $\mathcal{O}(n)$.
- Peor caso: sucede cuando el vector está ordenado en orden inverso. Cada elemento `valor` tiene que compararse e intercambiarse con cada uno de los elementos anteriores ya ordenados, lo que lleva a $\frac{n(n-1)}{2}$ comparaciones e intercambios, resultando en una complejidad de $\mathcal{O}(n^2)$.

2.4.2. Complejidad de algoritmos recursivos

A diferencia de los algoritmos iterativos, que generalmente tienen un análisis más directo basado en la cantidad de veces que se ejecutan los distintos bloques del algoritmo, el análisis de algoritmos recursivos presenta un desafío particular.

La complejidad de un algoritmo recursivo depende especialmente del número y la naturaleza de sus llamadas recursivas. Estas llamadas suelen seguir una *relación de recurrencia* que describe cómo el problema original se descompone en otras llamadas —típicamente, reduciendo la talla del problema—, las cuales son resueltas de manera similar. Estas relaciones ayudan a entender cómo la solución a un problema más grande se construye a partir de la solución a problemas más pequeños, proporcionando un marco para analizar la complejidad computacional de un algoritmo recursivo.

Para ilustrar este concepto, utilizaremos dos ejemplos concretos: la búsqueda binaria y el algoritmo de ordenación por selección.

Complejidad de "búsqueda binaria"

La búsqueda binaria es un método eficiente para encontrar un elemento específico en una lista ordenada. Es un ejemplo clásico de cómo la estructura

y el orden en los datos pueden ser explotados para mejorar dramáticamente la eficiencia de los algoritmos de búsqueda. Este enfoque es particularmente útil en aplicaciones donde los datos están ordenados y las búsquedas son frecuentes, como en bases de datos y sistemas de indexación.

El algoritmo de búsqueda binaria opera bajo la premisa (no hay que verificarla) de que la lista o vector está ordenado. Comienza con un intervalo que cubre toda la lista. En cada paso, el algoritmo compara el elemento en el centro del intervalo con el valor objetivo:

- Si el valor en el centro es igual al valor objetivo, la búsqueda concluye.
- Si el valor objetivo es menor que el valor en el centro, la búsqueda continúa en la mitad izquierda de la lista (valores menores).
- Si el valor objetivo es mayor que el valor en el centro, la búsqueda continúa en la mitad derecha de la lista (valores mayores).

Este proceso se repite, reduciendo a la mitad el tamaño del intervalo de búsqueda en cada paso, hasta que el valor objetivo es encontrado o el intervalo se reduce a cero (lo cual indica que el valor objetivo no está en la lista).

El pseudocódigo del algoritmo de búsqueda binaria que refleja esta descripción es el siguiente:

```
función búsqueda_binaria(v, z)
    si |v| = 0
        devuelve NO_ENCONTRADO
    m := |v| / 2
    si v_m = z
        devuelve m
    si v_m > z
        devuelve búsqueda_binaria(v_:m, z)
    si no
        devuelve búsqueda_binaria(v_m:, z)
```

Para analizar su complejidad, comenzamos identificando la talla del problema: el tamaño del vector ($n = |v|$). También podemos observar que tenemos distintos casos. En el mejor caso, el valor objetivo coincide con el valor en el centro del vector (`v_m = z` en la primera comparación). Este caso tiene una complejidad constante $\mathcal{O}(1)$, ya que no depende de la talla sino de algunas operaciones básicas de comparación. En cambio, en el peor caso (el

valor no está en la lista), tenemos que completar la recursión desde el caso general hasta el caso base (`|v| = 0`).

Para resolver la complejidad del peor caso, vamos a obtener primero la relación de recurrencia. Podemos comenzar con el caso base: cuando $n = 0$, únicamente tenemos la comparación para la terminación del algoritmo, por lo que $T(n) \in \mathcal{O}(1)$. Cuando no se da el caso base ($n \geq 1$), y asumiendo el peor caso, siempre se ejecutarán las comparaciones y alguna llamada que divide el vector entre dos. Por lo tanto, la relación de recurrencia para el tiempo de ejecución $T(n)$ de la búsqueda binaria se define formalmente como:

$$T(n) = \begin{cases} \mathcal{O}(1) & \text{si } n = 0 \\ T(\frac{n}{2}) + \mathcal{O}(1) & \text{si } n \geq 1 \end{cases} \tag{2.1}$$

Vamos a ver cómo resolver la complejidad de esta relación de recurrencia usando el método de expansión. Iremos expandiendo la relación paso a paso para obtener una visión más clara de cómo el problema se descompone en cada llamada recursiva y cómo se van acumulando los costes temporales:

- Expansión inicial:
$$T(n) = T\left(\frac{n}{2}\right) + \mathcal{O}(1)$$
- Sustitución de la primera expansión:
$$T(n) = \left(T\left(\frac{n}{4}\right) + \mathcal{O}(1)\right) + \mathcal{O}(1) = T\left(\frac{n}{4}\right) + 2 \cdot \mathcal{O}(1)$$
- Sustitución de la segunda expansión:
$$T(n) = \left(T\left(\frac{n}{8}\right) + \mathcal{O}(1)\right) + 2 \cdot \mathcal{O}(1) = T\left(\frac{n}{8}\right) + 3 \cdot \mathcal{O}(1)$$

Podemos ver que si seguimos expandiendo, el patrón que emerge es:

$$T(n) = T\left(\frac{n}{2^k}\right) + k \cdot \mathcal{O}(1)$$

Sabemos que este proceso se expande hasta que el argumento de T se reduzca a 0. Es decir, k es el número de veces que necesitamos dividir n por 2 hasta llegar a 0, es decir, $\frac{n}{2^k} = 0$. Matemáticamente, esto ocurre en la llamada recursiva realizada en $T(1)$ (cuando $\frac{n}{2^k} = 1$), por lo que necesitaremos $\log_2 n$ llamadas para llegar a $T(1)$ y una más para terminar la búsqueda. Por lo tanto:

$$T(n) = T(1) + \log_2 n = 2 \cdot \mathcal{O}(1) + \log_2 n$$

La forma final de la relación de recurrencia nos muestra que el tiempo total de ejecución del algoritmo de búsqueda binaria es proporcional al logaritmo en base 2 del tamaño de la entrada. Por lo tanto, la complejidad de tiempo de la búsqueda binaria pertenece al orden $\mathcal{O}(\log n)$.

Como se puede observar, la búsqueda binaria reduce significativamente el tiempo de búsqueda al dividir repetidamente a la mitad el rango de búsqueda, en lugar de inspeccionar cada elemento uno por uno como se haría en una búsqueda secuencial, que tendría una complejidad $\mathcal{O}(n)$.

Complejidad de "ordenación por selección"

Vamos a analizar ahora otro algoritmo clásico de ordenación: la ordenación por selección. Este algoritmo funciona de la siguiente manera. Dada una lista:

1. Encuentra el elemento mínimo en la lista y lo intercambia con el elemento de la primera posición (su lugar correcto).
2. Llama recursivamente a ordenar el resto de la lista, excluyendo el primer elemento.

Es un algoritmo muy simple, pudiéndose plantear de manera recursiva de la siguiente manera:[3]

```
función ordenación_por_selección(v):
    si |v| = 1:
        devuelve
    índice_mínimo := 1
    para i:=1 hasta |v|:
        si v_i < v_índice_mínimo:
            índice_mínimo := i
    intercambiar(v_1, v_índice_mínimo)
    ordenación_por_selección(v_2:)
```

Para analizar su complejidad, identificamos primero que el algoritmo de ordenación por selección no tiene un mejor ni peor caso, ya que su complejidad temporal no depende del orden inicial de los elementos en la lista.

[3] La implementación típica de este algoritmo no es recursiva sino iterativa (con otro bucle exterior). Sin embargo, usamos aquí la alternativa recursiva para ejemplificar el cálculo de complejidades de este tipo de algoritmos.

Siempre ejecuta el mismo número de comparaciones y asignaciones sin importar la entrada, sino que únicamente depende de la talla.

Si representamos la complejidad temporal del algoritmo para una lista de n elementos como $T(n)$, podemos descomponer el proceso de la siguiente manera:

1. Encontrar el mínimo en una lista de n elementos requiere $n-1$ comparaciones. Por tanto, este paso pertenece a $\mathcal{O}(n)$.

2. Intercambiar el mínimo con el primer elemento toma un tiempo constante ($\mathcal{O}(1)$).

3. Ordenar el resto de la lista de $n-1$ elementos tiene una complejidad dada por la complejidad de la llamada recursiva; es decir, $T(n-1)$.

En este caso, podemos expresar la relación de recurrencia como:

$$T(n) = \begin{cases} \mathcal{O}(1) & \text{si } n \leq 1 \\ (n-1) + T(n-1) & \text{si } n > 1 \end{cases} \tag{2.2}$$

Al igual que en el ejemplo anterior, podemos resolver esta relación de recurrencia con el método de expansión:

$$T(n) = (n-1) + T(n-1)$$
$$T(n) = (n-1) + (n-2) + T(n-2)$$
$$T(n) = (n-1) + (n-2) + (n-3) + \cdots + 1 + T(1)$$

Dado que $T(1)$ tiene complejidad constante ($\mathcal{O}(1)$), podemos expresar la suma de la siguiente manera:

$$T(n) = (n-1) + (n-2) + (n-3) + \cdots + 1 = \sum_{i=1}^{n-1} i$$

Esta suma aritmética corresponde a:

$$T(n) = \frac{(n-1)n}{2}$$

Por lo tanto, la complejidad temporal del algoritmo de ordenación por selección $T(n) \in \mathcal{O}(n^2)$. O en otras palabras, el tiempo de ejecución del algoritmo de ordenación por selección crece de manera cuadrática en función del número de elementos en la lista.

Capítulo 3

Divide y vencerás

El planteamiento "divide y vencerás" es una estrategia para resolver problemas complejos dividiéndolos en problemas más pequeños y manejables. La idea es bastante intuitiva: en lugar de afrontar el problema completo, se divide en partes más pequeñas, hasta el punto en que su resolución puede ser trivial, y luego se combinan las soluciones de esas partes para obtener la solución final. A pesar de su simpleza conceptual, este enfoque es muy potente porque permite abordar una gran cantidad de problemas, no sólo computacionales.

3.1. Esquema general

Los algoritmos de divide y vencerás se basan en tres etapas principales:

- **División:** Dividir el problema original en subproblemas más pequeños que son instancias del mismo tipo de problema.
- **Resolución:** Cuando un subproblema es lo suficientemente pequeño, se resuelve de forma directa.
- **Combinación:** Se combinan las soluciones de los subproblemas para obtener la solución del problema original.

No todas estas etapas se utilizan en cualquier problema. Como veremos más adelante, en algunos casos, alguna de las etapas no es necesaria o permanece implícita en la solución.

La mayoría de los enfoques computacionales basados en divide y vencerás se plantean mediante recursividad, ya que permite abstraer más fácilmente las distintas etapas. Un ejemplo genérico siguiendo este principio sería:

```
función dyv(n)
    si trivial(n)
        devuelve resolver(n)

    subproblemas := división(n)
    para cada subproblema en subproblemas
        soluciones += dyv(subproblema)

    devuelve combinar(soluciones)
```

donde `n` representa, de manera genérica, la entrada (haciendo referencia a la talla) del problema.

3.1.1. Ejemplo: búsqueda binaria

Vamos a identificar los distintos pasos del esquema de "divide y vencerás" con la ya conocida búsqueda binaria. Como ya vimos en el capítulo anterior, el algoritmo funciona dividiendo repetidamente el rango de búsqueda a la mitad, hasta encontrar el valor objetivo o determinar que no está presente. Recordemos su pseudocódigo:

```
función búsqueda_binaria(v, z):
    si |v| = 0:
        devuelve NO_ENCONTRADO
    m := |v| / 2
    si v[m] = z
        devuelve m
    si v[m] > z
        devuelve búsqueda_binaria(v[:m], z)
    si_no
        devuelve búsqueda_binaria(v[m:], z)
```

En este algoritmo, podemos identificar las tres etapas mencionados anteriormente:

- **División**. Se realiza al calcular el índice medio (`m`) y dividir la lista en dos partes: la parte izquierda y la parte derecha.
- **Resolución**. El problema es trivial cuando la lista tiene tamaño cero (`|v| = 0`) o el elemento objetivo está en la posición central (`si v[m] = z`).

- **Combinar**. Este paso es más o menos implícito, ya que el subproblema resuelto proporciona directamente la solución final. En este caso, la combinación sería simplemente propagar la instrucción `devuelve` en la pila de llamadas recursivas.

La complejidad temporal de la búsqueda binaria corresponde a la relación de recurrencia $T(n) = T(\frac{n}{2}) + O(1)$, donde n es el número de elementos en el vector. Como ya vimos en el capítulo anterior, su complejidad es del orden $\mathcal{O}(\log n)$.

3.2. Algoritmos de ordenación

Los algoritmos de ordenación son de vital importancia en multitud de aplicaciones. Es por ello que a lo largo de los años se han realizado numerosas propuestas para encontrar algoritmos eficientes de ordenación. En este capítulo vamos a explorar dos de los algoritmos de ordenación más conocidos y eficientes, que siguen el esquema de divide y vencerás: MergeSort y QuickSort.

3.2.1. MergeSort: Ordenación por mezcla

La "ordenación por mezcla", más conocida como MergeSort, es un algoritmo de ordenación que utiliza el principio de divide y vencerás. Como veremos después, MergeSort garantiza una complejidad temporal del orden $\mathcal{O}(n \log n)$ en todos los casos.

MergeSort se basa en el principio de que es más fácil construir una lista ordenada a partir de dos sublistas ya ordenadas. Para ello, el algoritmo divide la lista a ordenar recursivamente hasta que llegamos a sublistas cuya ordenación es trivial (listas con máximo un elemento). Después, siguiendo el orden de las llamadas recursivas, va mezclando (operación `mezclar`) las sublistas ordenadas hasta obtener la lista original ordenada.

Un pseudocódigo del algoritmo sería el siguiente:

```
función mergesort(v):
    si |v| <= 1
        devuelve v
    medio = |arr| / 2
    izquierda = mergesort(arr[:medio])
    derecha = mergesort(arr[medio:])
    return mezclar(izquierda, derecha)
```

Con carácter general, podemos identificar las etapas que componen el esquema de divide y vencerás:

1. **División.** Dividir la lista en dos mitades iguales.
2. **Resolución.** Cuando la lista es de tamaño menor o igual a 1, ya está ordenada.
3. **Combinación.** Combinar las dos mitades ordenadas en una sola lista ordenada, mediante la operación *mezclar*.

En el pseudocódigo anterior, hemos utilizado una función de mezcla (`mezclar`). Esta función asume que se reciben dos listas ya ordenadas y devuelve la unión de éstas, también ordenada. Esta función tendría un pseudocódigo como el siguiente:

```
función mezclar(a, b):
    resultado = []
    i, j := 0, 0
    mientras i < |a| y j < |b|
        si a[i] < b[j]
            resultado += a[i]
            i += 1
        else:
            resultado += b[j]
            j += 1
    resultado += a[i:]
    resultado += b[j:]
    devuelve resultado
```

Básicamente, la función `mezclar` va construyendo la lista a devolver (*resultado*) concatenando en cada momento el siguiente menor número de las listas de entrada. Esto se implementa eficientemente manteniendo un índice para cada lista (i, j), que apuntan a la siguiente posición a comparar, respectivamente. Cuando alguno de los índices sobrepasa el tamaño de la lista correspondiente, se concatenan los elementos restantes de la otra lista, que ya estará ordenada (condición del algoritmo).

MergeSort no tiene mejor o peor caso, dado que ni el número de llamadas ni las operaciones a ejecutar dependen del contenido de la lista. Teniendo en cuenta que la operación `mezclar` tiene un coste del orden $\mathcal{O}(n)$, la relación de recurrencia de MergeSort se define como:

$$T(n) = \begin{cases} \mathcal{O}(1) & \text{si } n \leq 1 \\ \mathcal{O}(n) + 2T(\frac{n}{2}) & \text{si } n > 1 \end{cases} \tag{3.1}$$

Esta relación de recurrencia indica un coste perteneciente a $\mathcal{O}(n \log n)$.

3.2.2. Quicksort: Ordenación rápida

La "ordenación rápida", más conocida como *QuickSort*, es un algoritmo de ordenación eficiente que sigue el paradigma de Divide y Vencerás. Este algoritmo divide una lista en dos partes a partir de un elemento "pivote". Este elemento es uno de los valores de la lista y sirve para organizar el resto de elementos en dos listas disjuntas: una para los elementos que son mayores que el pivote y otra para los elementos que son menores que el pivote. Una vez posicionado el pivote en el lugar que le corresponde en la lista ordenada final, QuickSort llama recursivamente para ordenar las dos (sub)listas restantes.

Siguiendo el paradigma de este capítulo, podemos describir el proceso de QuickSort mediante los siguientes pasos:

1. **División:** Elegir un elemento como "pivote" y dividir el vector de manera que todos los elementos menores que el pivote queden a un lado de la lista y los mayores al otro. Esto posiciona el pivote en el lugar que le corresponde en la lista final. A continuación, se ordenan ambos lados de la lista mediante llamadas recursivas.

2. **Resolución:** La resolución es implícita cuando la entrada es un vector de tamaño 1.

3. **Combinación:** Tras la llamada recursiva, el vector queda ordenado puesto que el elemento pivote está en la posición que le corresponde y las otras dos partes se han ordenado recursivamente.

Vamos a asumir por ahora que el pivote se elige aleatoriamente de entre todos los elementos del vector (abajo se comentará más sobre esta decisión). Siendo así, un pseudocódigo para QuickSort sería el siguiente:

```
función qsort(v):
    si |v| <= 1
        devuelve v
    si no
        pivote := aleatorio(1,|v|)
```

```
    izq, der := partición(v,pivote)
    devuelve [qsort(izq), pivote, qsort(der)]
```

El elemento pivote se utiliza para la función `partición`, la cual devuelve dos listas con los valores menores y mayores que el pivote, respectivamente.[1] Esta función tendría un coste perteneciente a $\mathcal{O}(n)$, ya que únicamente tendría que recorrer la lista una vez para dividirla en los dos conjuntos correspondientes.

QuickSort tiene una complejidad temporal diferente según la lista a ordenar. Es decir, sí presenta un mejor y peor caso. El mejor caso ocurre cuando el pivote elegido es siempre la mediana de los elementos a ordenar, lo que permite dividir la lista en dos partes iguales en cada nivel de recursión. Esto conlleva una relación de recurrencia tal que:

$$T(n) = \begin{cases} \mathcal{O}(1) & \text{si } n \leq 1 \\ \mathcal{O}(n) + 2T(\frac{n}{2}) & \text{si } n > 1 \end{cases} \tag{3.2}$$

Este caso arroja una complejidad de orden $\mathcal{O}(n \log n)$. Sin embargo, el peor caso, que ocurre cuando el pivote es siempre el elemento o más grande o más pequeño (casos degenerados), lleva a una relación de recurrencia tal que:

$$T(n) = \begin{cases} \mathcal{O}(1) & \text{si } n \leq 1 \\ \mathcal{O}(n) + T(1) + T(n-1) & \text{si } n > 1 \end{cases} \tag{3.3}$$

Esto se debe a que en cada paso sólo se reduce el tamaño de la lista en uno, lo que genera una profundidad de recursión máxima, que sumado a que los pasos no recursivos tienen complejidad $\mathcal{O}(n)$, lleva a una complejidad temporal perteneciente a $\mathcal{O}(n^2)$.

Elección del pivote

Como acabamos de ver, la elección del pivote es crucial para el rendimiento de QuickSort. En la práctica, se selecciona el pivote de forma aleatoria o casi-aleatoria (por ejemplo, utilizando estrategias como el "mediana de N", que elige el elemento mediana de N elementos seleccionados al azar). Estadísticamente, esto lleva a un caso promedio con una complejidad esperada de $\mathcal{O}(n \log n)$.[2]

[1] Una de las dos listas incluirá los elementos iguales al propio pivote (sin incluirlo). Cuál de las dos lo haga es irrelevante para el algoritmo.

[2] Como vimos en el Capítulo 2, el caso promedio se refiere a la esperanza (matemática) del coste para una instancia cualquiera, y no al promedio entre el mejor y el peor caso.

También es posible forzar la búsqueda de la mediana exacta en cada partición, lo cual garantiza siempre el mejor caso. Sin embargo, encontrar la mediana exacta tiene un coste adicional de $\mathcal{O}(n)$. Asintóticamente, esto asegura siempre el mejor caso pero puede no ser eficiente en muchas aplicaciones.

En resumen, aunque existen métodos para mejorar la elección del pivote, QuickSort es generalmente eficiente con estrategias de pivote simples.

3.3. Consideraciones adicionales

Para poder aplicar el esquema de Divide y Vencerás, se deben identificar una serie de requisitos. El primero es encontrar una forma de descomponer un problema en partes más pequeñas (reduciendo la talla) y que cada descomposición acerque el problema a un punto en el cual se pueda resolver de manera directa. Además, es necesario que exista una forma de combinar las soluciones de los subproblemas de manera que se obtenga la solución del problema original.

Aunque, a priori, la idea de Divide y Vencerás sea muy general, no siempre el problema se puede descomponer o dicha descomposición no implica necesariamente que nos acerquemos a poder resolverlo de manera directa. Del mismo modo, no siempre es posible resolver el problema original a partir de la resolución de los subproblemas.

Un ejemplo clásico de un problema que no es fácilmente resoluble mediante el esquema de Divide y Vencerás es el "Problema del viajante de comercio". Este problema consiste en encontrar el camino más corto que permite a un viajante visitar un conjunto de ciudades, pasando por cada una de ellas exactamente una vez y regresando al punto de partida. En este caso, es imposible dividir el problema en subproblemas sin que haya interdependencias entre estos subproblemas. Es decir, que resolver los subproblemas independientemente y luego combinarlos no garantiza la solución óptima. Veremos este problema en profundidad en el Capítulo 6 (Vuelta atrás).

Capítulo 4

Programación dinámica

La programación dinámica es una técnica de optimización de algoritmos que se basa en la resolución de problemas a partir de subproblemas más simples y en almacenar los resultados de estos subproblemas para evitar cálculos redundantes. Esta técnica es especialmente útil en problemas que pueden ser descritos por relaciones de recurrencia y en los cuales es común que muchos subproblemas se repitan. Por ejemplo, esta última condición no se cumple, o es muy poco frecuente, en los algoritmos de ordenación vistos en el Capítulo 3. Sin embargo, como veremos en este capítulo, es muy común en diversos problemas de optimización.

4.1. Esquema general

La idea central de la programación dinámica es la descomposición del problema en subproblemas solapados, cuya solución se almacena en una estructura de datos, típicamente una tabla, para ser reutilizada posteriormente. Este enfoque, por tanto, reduce drásticamente el tiempo de ejecución de un algoritmo al evitar volver a computar subproblemas ya resueltos.

Este esquema se fundamenta en dos conceptos clave:

- **Subproblemas solapados**: Esta característica implica que un problema puede dividirse en subproblemas que se repiten con frecuencia. Un ejemplo muy claro es el cálculo del número enésimo de Fibonacci (se verá más abajo).
- **Almacenamiento de resultados**: La eficiencia de la programación dinámica se consigue almacenando los resultados de los subproblemas, de manera que sólo se resuelven una única vez y la solución está

disponible cuando vuelve a ser necesario. A su vez, esto se puede implementar de dos formas:

- **Memoización**:[1] Esta implementación sigue un enfoque de arriba hacia abajo (top-down) para resolver mediante **programación dinámica recursiva**, en el cual se almacenan los resultados de las llamadas recursivas ya realizadas.
- **Tabulación**: Esta técnica también se utiliza para evitar cálculos repetidos, pero los subproblemas se resuelven de abajo hacia arriba (bottom-up). Se construye una tabla que almacena los resultados de subproblemas más pequeños, los cuales se utilizan para resolver problemas más grandes de manera iterativa. Este enfoque lleva a la **programación dinámica iterativa**.

Para entender mejor la programación dinámica, veamos un ejemplo sencillo.

4.1.1. Ejemplo: el número de Fibonacci

Un ejemplo clásico y sencillo de programación dinámica es el cálculo de los números de Fibonacci. La secuencia de Fibonacci se define de la siguiente manera:

$$F(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ F(n-1) + F(n-2) & \text{si } n > 1 \end{cases}$$

Un ejemplo de pseudocódigo para esta definición se presenta a continuación:

```
función fibonacci(n):
    si n <= 1
        devuelve n
    si no
        devuelve fibonacci(n-1) + fibonacci(n-2)
```

La implementación recursiva de esta función es ineficiente porque recalcula los mismos valores múltiples veces, lo que lleva a una complejidad

[1] Este término no es un error ortográfico, sino que viene del latín *memorandum* (ser recordado). No debe confundirse con "memorización", que en computación puede tener otros significados.

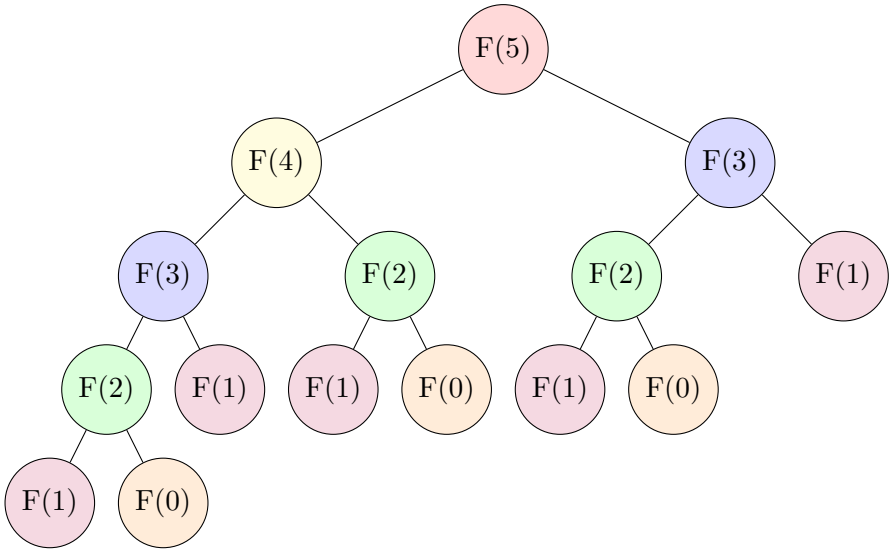


Figura 4.1: Desarrollo de llamadas recursivas para calcular F(5). Cada color se asocia a una llamada con un valor específico del parámetro.

exponencial.[2] Esto se aprecia gráficamente en la Fig. 4.1. Podemos mejorar esto utilizando programación dinámica.

A continuación se presenta el pseudocódigo utilizando memoización (programación dinámica recursiva):

```
función fibonacci(n, memo):
    si n <= 1
        devuelve n
    si n en memo
        devuelve memo[n]
    si no
        memo[n] := fibonacci(n-1, memo) + fibonacci(n-2, memo)
        devuelve memo[n]
```

En esta versión, se utiliza un diccionario (o una estructura similar) llamado *memo* para almacenar los resultados de los subproblemas ya calculados, evitando así el recálculo.

Otra forma de resolver el problema es utilizando tabulación (programación dinámica iterativa). En lugar de resolver los subproblemas recursivamente, se construye una tabla de soluciones desde los casos triviales hasta

[2] Aproximadamente, del orden $\mathcal{O}(1{,}6^n)$.

los más complejos. El pseudocódigo para este enfoque se presenta a continuación:

```
función fibonacci(n):
    si n <= 1
        devuelve n
    tabla[0] := 0
    tabla[1] := 1
    para i desde 2 hasta n:
        tabla[i] := tabla[i-1] + tabla[i-2]
    devuelve tabla[n]
```

En esta versión, se construye una tabla (por ejemplo, un vector) donde cada entrada i contiene el valor de $F(i)$. De esta manera, se evitan los cálculos repetidos y se mejora la eficiencia de la función.

Ambos enfoques de programación dinámica mejoran significativamente la eficiencia del cálculo de los números de Fibonacci. En el caso iterativo, es obvio que la complejidad pertenece a $\mathcal{O}(n)$. El caso recursivo es equivalente, aunque el cálculo de su complejidad sea menos intuitivo.

4.2. El problema de la mochila (discreta)

El "problema de la mochila" (en inglés, *knapsack problem*) es uno de los problemas clásicos en teoría de algoritmos y computación. En términos generales, consiste en seleccionar un subconjunto de elementos, cada uno con un peso y un valor, de manera que se maximice el valor total sin exceder un peso máximo permitido. Aunque aparentemente sencillo, muchos problemas reales pueden reducirse a una instancia del problema de la mochila, como la planificación de proyectos o la distribución de recursos.

El problema de la mochila puede formularse matemáticamente de la siguiente manera: dado un conjunto de n elementos, donde cada elemento i tiene un valor v_i y un peso w_i, y una capacidad máxima de la mochila W, se busca maximizar la suma de los valores de los elementos seleccionados, de modo que la suma de los pesos no exceda W.

$$\text{Maximizar} \sum_{i=1}^{n} v_i x_i$$

sujeto a:

$$\sum_{i=1}^{n} w_i x_i \leq W$$

donde x_i es una variable binaria que indica si el elemento i es seleccionado ($x_i = 1$) o no ($x_i = 0$). Para resolver el problema de la mochila, podemos definir la relación de recurrencia que describe el valor máximo que se puede obtener:

$$K(i, w) = \begin{cases} 0 & \text{si } i = 0 \text{ o } w = 0 \\ K(i-1, w) & \text{si } w_i > w \\ \text{máx}(K(i-1, w), v_i + K(i-1, w-w_i)) & \text{si } w_i \leq w \end{cases}$$

donde $K(i, w)$ representa el valor máximo que se puede obtener utilizando los primeros i elementos con una capacidad de mochila de w. Esta relación de recurrencia tiene dos casos:

1. Caso base: Si no hay elementos ($i = 0$) o la capacidad de la mochila es cero ($w = 0$), entonces el valor máximo que se puede obtener es 0.

2. Exclusión del elemento: Si el peso del i-ésimo elemento es mayor que la capacidad actual de la mochila ($w_i > w$), entonces este elemento no puede incluirse en la solución óptima. En este caso, el valor óptimo es el mismo que el valor óptimo obtenido sin este elemento, es decir, $K(i-1, w)$.

3. Inclusión del elemento: Si el peso del i-ésimo elemento es menor o igual a la capacidad actual de la mochila ($w_i \leq w$), entonces se considera dos posibilidades: no incluir el i-ésimo elemento (valor $K(i-1, w)$) o incluirlo (valor $v_i + K(i-1, w-w_i)$). El valor máximo se obtiene eligiendo la opción que maximiza el valor total.

En este capítulo vamos a estudiar **la versión discreta** del problema de la mochila, en la cuál se añade la restricción de que los pesos de los objetos son discretos (valores enteros). Esto permite indexar en la tabla de programación dinámica los distintos estados en los que puede encontrarse la mochila a medida que se van seleccionando o no objetos.

Para resolver este problema utilizando programación dinámica, podemos definir una tabla `PD` donde `PD[i][w]` representa el valor máximo que se puede obtener utilizando los primeros `i` elementos con una capacidad de mochila de `w`. A continuación, se presenta el pseudocódigo utilizando memoización:

```
función mochila(n, W, pesos, valores, PD):
    si n = 0 o W = 0
        devuelve 0
    si (n, W) en PD
        devuelve PD[(n, W)]
    si pesos[n-1] > W
        PD[(n, W)] := mochila(n-1, W, pesos, valores, PD)
    si no
        PD[(n, W)] := max(
            mochila(n-1, W, pesos, valores, PD),
            valores[n-1] + mochila(n-1, W-pesos[n-1], pesos,
   valores, PD)
        )
    devuelve PD[(n, W)]
```

La versión iterativa utilizando tabulación rellena la tabla `PD` de manera iterativa, comenzando desde el primer elemento hasta el último y para todos los posibles estados de la mochila. Esta versión se presenta a continuación:

```
función mochila(n, W, pesos, valores):
    PD := matriz de ceros de tamaño (n+1, W+1)
    para i desde 1 hasta n:
        para w desde 1 hasta W:
            si pesos[i-1] > w:
                PD[i][w] := PD[i-1][w]
            si no:
                PD[i][w] := max(
                    PD[i-1][w],
                    valores[i-1] + PD[i-1][w-pesos[i-1]]
                )
    devuelve PD[n][W]
```

En esta versión, se construye una tabla `PD` donde cada entrada `PD[i][w]` contiene el valor máximo que se puede obtener utilizando los primeros `i` elementos con una capacidad de mochila de `w`.

Ambos enfoques, memoización y tabulación, tienen sus ventajas y desventajas. La memoización es más intuitiva y fácil de implementar, ya que sigue la estructura recursiva natural del problema. Sin embargo, puede tener una sobrecarga de memoria debido al almacenamiento de la pila de llamadas recursivas. La tabulación, por otro lado, es más eficiente en términos de espacio y evita la sobrecarga de llamadas recursivas, pero puede ser menos intuitiva ya que requiere la construcción explícita de la tabla de soluciones.

Además, la tabulación siempre resuelve todos los subproblemas posibles, incluso los no necesarios, mientras que la memoización solo resuelve los subproblemas que realmente se necesitan.

Al igual que en el ejemplo de Fibonacci, la complejidad es fácil de inferir a partir de la versión iterativa. En este caso, el algoritmo pertenece a $\mathcal{O}(n \cdot W)$.[3]

4.3. La distancia de edición

La *distancia de edición*, también conocida como distancia de Levenshtein, es una métrica fundamental en el campo de la inteligencia artificial y el procesamiento del lenguaje natural. Esta métrica se utiliza para evaluar cuán similares son dos cadenas de texto, midiendo el número mínimo de operaciones necesarias para transformar una cadena en otra. Las operaciones permitidas son la inserción, eliminación y sustitución de caracteres. La importancia de la distancia de edición radica en su amplia aplicación en problemas como la corrección ortográfica, el reconocimiento de voz, la comparación de secuencias genéticas, la búsqueda aproximada de cadenas y la evaluación de sistemas de traducción automática, entre otros.

El problema de la distancia de edición se plantea de la siguiente manera: dadas dos cadenas s_1 y s_2, queremos encontrar el número mínimo de operaciones (inserciones, eliminaciones o sustituciones) necesarias para transformar s_1 en s_2.

Consideremos un ejemplo sencillo para ilustrar la distancia de edición entre dos cadenas. Supongamos que queremos transformar la cadena "casa" en "costa". Para ello:

- Sustituimos 'o' por la primera 'a': "casa" $\rightarrow$ "cosa".
- Insertamos 't' después de 's': "cosa" $\rightarrow$ "costa".

La distancia de edición en este caso es 2, ya que necesitamos una operación de inserción y una de sustitución.

Para resolver la distancia de edición computacionalmente lo podemos plantear como un problema recursivo en el cual se va realizando el cálculo para los prefijos de cada cadena. Para llegar a la relación de recurrencia $d(i, j)$, donde se comparan los prefijos $s_1[1..i]$ y $s_2[1..j]$:

[3] Nótese que en este caso, hay dos variables que definen la talla del problema: el número de elementos n y el peso máximo de la mochila W.

1. Si el último carácter de ambas cadenas es el mismo ($s_1[i] = s_2[j]$), la distancia de edición es la misma que para las subcadenas anteriores, es decir, $d(i-1, j-1)$.

2. Si el último carácter de ambas cadenas es diferente ($s_1[i] \neq s_2[j]$), debemos considerar la operación de menor coste entre:

   - Insertar el carácter $s_2[j]$ en s_1, lo que implica $d(i, j-1) + 1$.
   - Eliminar el carácter $s_1[i]$, lo que implica $d(i-1, j) + 1$.
   - Sustituir el carácter $s_1[i]$ por $s_2[j]$, lo que implica $d(i-1, j-1)+1$.

Por tanto, la relación de recurrencia para la distancia de edición es la siguiente:

$$d(i,j) = \begin{cases} j & \text{si } i = 0 \\ i & \text{si } j = 0 \\ d(i-1, j-1) & \text{si } s_1[i-1] = s_2[j-1] \\ 1 + \min\{d(i-1,j), d(i,j-1), d(i-1,j-1)\} & \text{si } s_1[i-1] \neq s_2[j-1] \end{cases}$$

A continuación se presenta el pseudocódigo para calcular la distancia de edición utilizando una versión iterativa con tabulación, que es la implementación más habitual:

```
función distancia_edición(s1, s2):
    n := |s1|
    m := |s2|
    DP := matriz de tamaño (n+1, m+1)

    para i desde 0 hasta n
        DP[i][0] := i

    para j desde 0 hasta m
        DP[0][j] := j

    para i desde 1 hasta n
        para j desde 1 hasta m
            si s1[i-1] = s2[j-1]
                DP[i][j] := DP[i-1][j-1]
            si no
                DP[i][j] := 1 + min(DP[i-1][j],
```

```
                                DP[i][j-1],
                                DP[i-1][j-1])

    devuelve DP[n][m]
```

Consideremos de nuevo el ejemplo de transformar "casa" en "costa". Veamos cómo se llena la tabla `DP` paso a paso:

		c	o	s	t	a
	0	1	2	3	4	5
c	1	0	1	2	3	4
a	2	1	1	2	3	3
s	3	2	2	1	2	3
a	4	3	3	2	2	2

Para entender la tabla y el cálculo de la distancia de edición, hay que tener en cuenta que:

- La primera fila y la primera columna representan las distancias cuando una de las cadenas es vacía.
- Cada celda `DP[i][j]` se llena utilizando la relación de recurrencia descrita anteriormente. Por ejemplo, `DP[1][1]` es 0 porque los caracteres 'c' coinciden.
- Finalmente, `DP[4][5]` indica que la distancia de edición entre "casa" y "costa" es 2.

Con respecto a su complejidad, la distancia de edición mediante programación dinámica tiene un coste perteneciente a $\mathcal{O}(|s_1| \cdot |s_2|)$.

4.4. Recuperación de soluciones

En programación dinámica, cuando resolvemos un problema utilizando las técnicas que hemos visto (memoización y tabulación), normalmente nos centramos en calcular y almacenar los valores óptimos de los subproblemas, y no las soluciones específicas que los producen. Por ejemplo, en el problema de la mochila hemos calculado qué valor máximo se puede obtener pero el algoritmo no devuelve explícitamente los objetos que se seleccionarían.

Sin embargo, una vez que tenemos los valores óptimos, a menudo necesitamos recuperar la solución específica. Para hacer esto, podemos explorar la tabla de programación dinámica desde el final (donde se encuentra la

solución al problema completo) hacia el principio (donde se encuentran los subproblemas más pequeños).

Este proceso se conoce como **recuperación de soluciones** y típicamente sigue estos pasos:

1. Identificar el valor óptimo: Comenzamos en el elemento de la tabla que contiene el valor óptimo de la solución completa.

2. Rastrear las decisiones: Desde este punto, rastreamos hacia atrás las decisiones que llevaron a este valor óptimo. Esto implica seguir el camino inverso de las decisiones tomadas durante la construcción de la tabla. En cada paso, determinamos si el valor actual fue obtenido incluyendo un cierto elemento o excluyéndolo (en el contexto de problemas de optimización como el de la mochila).

3. Registrar las decisiones: Cada vez que determinamos una decisión, la registramos como parte de la solución específica. Este proceso se repite hasta que llegamos a la base del problema (por ejemplo, cuando el tamaño del subproblema es cero).

Supongamos que estamos resolviendo el problema de la mochila y tenemos una tabla de programación dinámica `DP` donde `DP[i][w]` representa el valor óptimo obtenido considerando los primeros `i` elementos y una restricción de peso restante `w`. La solución final se encuentra en `DP[n][W]`, donde `n` es el número total de elementos y `W` es el peso máximo permitido.

Para recuperar la solución, comenzamos en `DP[n][W]`:

1. Si `DP[i][w]` es igual a `DP[i-1][w]`, entonces el n-ésimo elemento no se incluyó en la solución óptima.

2. Si `DP[i][w]` es diferente de `DP[i-1][w]`, entonces el n-ésimo elemento sí se incluyó en la solución óptima. En este caso, continuamos el rastreo con los $n - 1$ elementos restantes y la nueva restricción $w = w - w_n$.

3. Continuamos este proceso hasta llegar a `DP[0][0]`.

Este método de rastreo inverso asegura que identificamos exactamente qué elementos o decisiones forman parte de la solución óptima, proporcionando no solo el valor del resultado óptimo sino también la composición específica de la solución.

Para finalizar, conviene recordar que, según el problema, es posible que haya varias soluciones igualmente óptimas. El procedimiento anterior seguiría siendo válido siempre y cuándo tan solo necesitáramos recuperar una

de estas posibles soluciones. En caso de que se quieran recuperar todas, habría que hacer ciertas modificaciones durante el rastreo.

4.5. Consideraciones adicionales

En este capítulo hemos visto la programación dinámica a través de dos enfoques: memoización (programación dinámica recursiva) y tabulación (programación dinámica iterativa). En teoría, ambos enfoques son equivalentes, pueden resolver los mismos problemas y tienen complejidades asintóticas iguales. Sin embargo, en la práctica sí presentan diferencias importantes. La memoización, siendo recursiva, puede resultar en un código más intuitivo y natural para problemas definidos recursivamente, pero conlleva una sobrecarga de llamadas recursivas y puede ser menos eficiente en términos de tiempo. En cambio, la tabulación evita la recursión, lo que puede hacerla más eficiente en tiempo al utilizar bucles iterativos, aunque puede requerir más memoria al almacenar todos los subproblemas intermedios (incluso los que no fueran necesarios). La elección entre ambos enfoques depende de la naturaleza del problema y de las restricciones de tiempo y espacio disponibles.

Capítulo 5

Algoritmos voraces

Los algoritmos voraces son un paradigma de diseño de algoritmos que construyen soluciones a problemas mediante una serie de decisiones que son óptimas localmente. La idea principal es escoger, en cada paso, la mejor opción disponible sin reconsiderar las elecciones previas o futuras, con la esperanza de que la solución global así obtenida sea óptima.

En ocasiones puede darse el caso de que una estrategia voraz lleve a la solución óptima para una determinada instancia de un problema, o incluso se puede demostrar que una estrategia voraz es óptima para un determinado problema, pero no es habitual encontrar problemas donde esto ocurra. Además, es posible que un algoritmo voraz ni siquiera encuentre una solución válida para una determinada instancia de problema. No obstante, la bondad de un enfoque voraz es su facilidad de implementación y su baja complejidad (típicamente lineal), lo cual puede ser más determinante si simplemente queremos una *buena* solución aunque no sea óptima.

5.1. Esquema general

Una estrategia voraz se puede resumir en los siguientes pasos:

1. Inicializar la solución.
2. Repetir hasta completar una solución:
   - *a*) Seleccionar el mejor candidato según un criterio voraz.
   - *b*) Comprobar si este candidato puede ser añadido a la solución sin violar las restricciones del problema.
   - *c*) Si se puede: añadir el candidato a la solución.

3. Devolver la solución obtenida.

Como se puede observar, el esquema es muy abstracto puesto que la clave radica en el paso 2a, que hace referencia a "mejor candidato según un criterio voraz". Como veremos en este capítulo, este paso es tremendamente dependiente del problema en cuestión. En la mayoría de los casos, las distintas opciones son ordenadas mediante algún criterio antes de pasar a seleccionarse de manera voraz.

5.1.1. Ejemplo: cambio de monedas

Para ilustrar algunas propiedas del esquema de algoritmos voraces, vamos a plantear el problema del cambio de monedas (*change-making problem*). El problema consiste en, dado un conjunto de tipos de monedas y una cantidad de dinero, determinar el número mínimo de monedas necesarias para alcanzar exactamente esa cantidad.

El problema del cambio de monedas se puede formular matemáticamente de la siguiente manera. Dado un conjunto de tipos de monedas $D = \{d_1, d_2, \ldots, d_n\}$, con $d_1 > d_2 > \ldots > d_n$, donde d_i representa el valor de la i-ésima moneda, y una cantidad de dinero C, se busca un conjunto de enteros no negativos $x_1, x_2, \ldots, x_n$ tal que:

$$\text{mín} \sum_{i=1}^{n} x_i$$

sujeto a la restricción:

$$\sum_{i=1}^{n} x_i \cdot d_i = C$$

donde x_i representa el número de monedas de tipo d_i que se usan en la solución.

El enfoque voraz para resolver este problema consiste en seleccionar, en cada paso, la moneda de mayor valor posible d_i que no exceda la cantidad restante r. Un pseudocódigo para el algoritmo se puede describir de la siguiente manera:

```
función cambio_monedas(C, D):
    n := |D|
    x := vector de tamaño n inicializado a 0
    r := C  # Cantidad restante por alcanzar
    para i desde 1 hasta n:
```

```
        x[i] := r // D[i]
        r := r - x[i] * D[i]
    devuelve x
```

El resultado es el conjunto de valores $x_1, x_2, \ldots, x_n$ que indica cuántas monedas de cada tipo se utilizan para alcanzar C.

A continuación vamos a presentar una serie de ejemplos que muestran que una estrategia voraz puede encontrar la solución óptima, pero también podría encontrar una solución no óptima o incluso no encontrar ninguna solución (aun existiendo). Cada uno de estos casos depende de la instancia concreta del problema:

- Instancia con solución óptima voraz: Supongamos que las monedas disponibles son $\{1, 5, 10, 25\}$ y necesitamos alcanzar $C = 30$. El algoritmo voraz selecciona la moneda de 25, dejando una cantidad restante de 5, para la cual selecciona la moneda de 5. La solución óptima es, efectivamente, $\{25, 5\}$ (2 monedas), que es también la solución que ofrece el algoritmo voraz.

- Instancia con solución subóptima voraz: Consideremos ahora las monedas $\{1, 3, 4\}$ y $C = 6$. El algoritmo voraz selecciona la moneda de 4, dejando una cantidad restante de 2. Luego selecciona dos monedas de 1 para cubrir la cantidad restante, dando la solución $\{4, 1, 1\}$ (3 monedas). Sin embargo, la solución óptima sería $\{3, 3\}$ (2 monedas). El algoritmo voraz encuentra una solución, pero no es óptima.

- Instancia sin solución voraz: Supongamos que las monedas disponibles son $\{3, 4\}$ y necesitamos alcanzar $C = 6$. El algoritmo voraz selecciona primero la moneda de 4, dejando una cantidad restante de 2. Luego, intenta cubrir esa cantidad restante con las monedas disponibles, pero no hay ninguna moneda posible, lo que lleva al algoritmo a no encontrar ninguna solución válida. Sin embargo, la solución óptima es $\{3, 3\}$, pero el algoritmo voraz no la encuentra porque prioriza la mayor disponible sin considerar combinaciones que podrían dar una solución válida.

El problema del cambio de moneda ilustra cómo una estrategia voraz puede ser eficiente y sencilla de implementar, pero también cómo puede llevar a soluciones subóptimas o incluso fallar en encontrar una solución.[1]

[1] En la Sección 5.4 discutiremos sobre si debemos utilizar el término "algoritmo" cuando la estrategia no resuelve el problema o no lo hace de manera óptima.

5.2. El problema de la mochila (continua)

El problema de la mochila continua es una variante del problema de la mochila clásica. En esta variante del problema, se permite fraccionar los objetos para maximizar el valor total que se puede llevar en una mochila con una capacidad limitada.

La variante continua del problema de la mochila puede formularse matemáticamente de la siguiente manera: dado un conjunto de n elementos, donde cada elemento i tiene un valor v_i y un peso w_i, y una capacidad máxima de la mochila W, se busca maximizar la suma de los valores de los elementos seleccionados, de modo que la suma de los pesos de éstos no exceda W.

$$\text{Maximizar } \sum_{i=1}^{n} v_i x_i$$

sujeto a:

$$\sum_{i=1}^{n} w_i x_i \leq W$$

donde x_i es una variable **continua** que indica el porcentaje del elemento i que se incluye $x_i \in [0, 1]$.

Esta variante del problema se puede resolver de manera óptima mediante un algoritmo voraz, basado en la relación valor/peso de los objetos. La idea del algoritmo se describe a continuación:

1. Calcular la relación valor/peso para cada objeto: $\frac{v_i}{w_i}$.
2. Ordenar los objetos en orden descendente de $\frac{v_i}{w_i}$.
3. Inicializar el peso total y el valor total a 0.
4. Para cada objeto en el orden calculado:
   a) Si el peso del objeto actual más el peso total no excede la capacidad W, añadir el objeto completo a la mochila.
   b) Si no, añadir la fracción del objeto que cabe en la mochila y detener el proceso.
5. Devolver el valor total de los objetos en la mochila.

Esta estrategia se presenta a continuación como pseudocódigo:

```
función mochila_continua(W, n, pesos, valores):

    para i desde 1 hasta n:
        ratio[i] := valores[i] / pesos[i]

    ordenar(ratio, valores, pesos)

    peso_total := 0
    valor_total := 0

    para i desde 1 hasta n:
        si peso_total + pesos[i] <= W
            peso_total := peso_total + pesos[i]
            valor_total := valor_total + valores[i]
        si no
            fraccion := (W - peso_total) / pesos[i]
            valor_total := valor_total + valores[i] * fraccion
            peso_total := W

    devuelve valor_total
```

Supongamos que tenemos los siguientes objetos:

$$\{(v_1 = 60, w_1 = 10), (v_2 = 100, w_2 = 20), (v_3 = 120, w_3 = 30)\}$$

y la capacidad de la mochila es $W = 50$.

Una estrategia voraz ordenaría primero los objetos atendiendo a la mejor relación valor/peso:

$$\frac{v_1}{w_1} = \frac{60}{10} = 6, \quad \frac{v_2}{w_2} = \frac{100}{20} = 5, \quad \frac{v_3}{w_3} = \frac{120}{30} = 4$$

La solución óptima sería un valor $240 = 60 + 100 + 80$, correspondiente a seleccionar completamente los objetos 1 y 2, y un $\frac{20}{30}$ del objeto 3.

Como hemos visto en el ejemplo de la sección anterior, los algoritmos voraces no son necesariamente "óptimos". Para verificar que un algoritmo voraz siempre proporciona la solución óptima hay que demostrarlo matemáticamente.

5.2.1. Demostración de optimalidad

El algoritmo voraz para el problema de la mochila continua es óptimo debido a que, en cada paso, se maximiza la cantidad de valor añadido por unidad de peso, es decir, se elige siempre el objeto (o fracción de objeto) con la mejor relación valor/peso. Esta estrategia garantiza que no existe otra forma de llenar la mochila que proporcione un mayor valor total, ya que cualquier otra combinación de objetos incluiría necesariamente un objeto con una menor relación valor/peso, lo que reduciría el valor total alcanzado.

La optimalidad se puede demostrar por contradicción: suponemos que existe otra forma de seleccionar los objetos que proporciona un valor mayor que el obtenido por el algoritmo voraz. En tal caso, esta selección debería incluir un objeto con una menor relación valor/peso antes de que la mochila esté completamente llena, lo cual contradice la suposición de que maximizar el valor/peso en cada paso proporciona la solución óptima. Por lo tanto, el algoritmo voraz es óptimo para esta variante del problema de la mochila.

5.2.2. Estrategia voraz en el caso general del problema de la mochila (discreta)

Una vez que hemos visto que el algoritmo voraz es óptimo para la variante continua del problema de la mochila, surge naturalmente la pregunta: ¿es este algoritmo también óptimo para la versión discreta del problema general de la mochila, donde no se permite fraccionar los objetos?

En otras palabras, si aplicamos el mismo enfoque voraz, que selecciona los objetos en función de su relación valor/peso $\frac{v_i}{w_i}$, ¿obtenemos siempre la solución óptima en el caso de la mochila general?

Aunque la estrategia voraz puede ser una buena aproximación, no siempre garantiza la solución óptima para la mochila general. Para demostrar esto, basta con proporcionar un contraejemplo que muestre una instancia donde la estrategia voraz falla en encontrar la solución óptima.

Consideremos la misma instancia del problema de la mochila que anteriormente pero en este caso formulamos la versión clásica (discreta), donde no se permite fraccionar los objetos.

Siendo así, la estrategia voraz seleccionaría primero v_1 y luego v_2, no quedando espacio para seleccionar v_3, lo que corresponde un valor total de $60 + 100 = 160$. Sin embargo, la solución óptima para esta versión sería elegir v_2 y v_3, lo que da un valor total de $100 + 120 = 220$ y cumple con la restricción de peso.

En este caso, la estrategia voraz falla en encontrar la solución óptima

debido a que prioriza la relación valor/peso sin considerar el impacto global en la solución. Por lo tanto, queda demostrado que la estrategia voraz no es óptima para el problema de la mochila discreta.

5.3. Algoritmo de Kruskal

El algoritmo de Kruskal es un algoritmo clásico utilizado para encontrar *el árbol de expansión mínima* (Minimum Spanning Tree, o MST) en un grafo conexo y no dirigido, minimizando el peso total de las aristas incluidas. Un MST es un subgrafo que conecta todos los vértices del grafo original con el menor peso total posible, y que no contiene ciclos. Este tipo de estructuras son fundamentales en muchas áreas de la computación, como el diseño de redes, la optimización de rutas, y la minimización de costes en sistemas conectados. Además, desde el punto de vista pedagógico, la simplicidad y claridad del algoritmo de Kruskal hacen que sea un excelente ejemplo para aprender cómo diseñar algoritmos para explotar estructuras inherentes a los problemas.

Para ilustrar el problema de encontrar el MST, consideremos un grafo ponderado como el que se muestra en la Fig. 5.1. En este grafo, los vértices A, B, C, D y E están conectados por aristas con diferentes pesos. El objetivo del algoritmo de Kruskal es encontrar un subgrafo que conecte todos los vértices (un árbol) de manera que el peso total de las aristas sea mínimo.

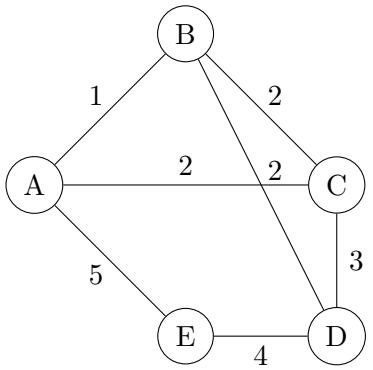


Figura 5.1: Ejemplo de grafo ponderado.

El algoritmo de Kruskal sigue los siguientes pasos:

1. Inicializar un conjunto de árboles, cada uno con un sólo vértice.

2. Crear una lista de todas las aristas del grafo, ordenadas por peso en orden ascendente.

3. Para cada arista en la lista ordenada:

   a) Si la arista conecta dos árboles distintos, añadirla a la solución y unir los dos árboles.

4. Repetir el paso 3 hasta que todos los vértices estén conectados en un único árbol.

Sea un grafo G representado por su conjunto de vertices V y de aristas A, el pseudocódigo para el algoritmo de Kruskal es el siguiente:

```
función Kruskal(V,A):
    MST := conjunto vacío
    T := estructura de tamaño |A|
    para i desde 1 hasta |A|
        T[i] = {i}
    ordenar(A)
    para cada (u, v) en A:
        si T[u] != T[v]
            añadir (u, v) a MST
            union(T[u],T[v])
    devuelve MST
```

Vamos a aplicar el algoritmo de Kruskal paso a paso utilizando el grafo de la Figura 5.1.

1. Ordenar las aristas por peso:

   - A-B (peso 1)
   - B-C (peso 2)
   - A-C (peso 2)
   - B-D (peso 2)
   - C-D (peso 3)
   - D-E (peso 4)
   - E-A (peso 5)

2. Inicialmente, cada vértice es un conjunto propio.

   - Conjuntos: {A}, {B}, {C}, {D}, {E}

3. **3.** Seleccionar iterativamente la arista de menor peso que no forme un ciclo y unir los conjuntos correspondientes:

   a) A-B (peso 1): Une los conjuntos {A} y {B}.
      - Conjuntos: {A, B}, {C}, {D}, {E}

   b) B-C (peso 2): Une los conjuntos {A, B} y {C}.
      - Conjuntos: {A, B, C}, {D}, {E}

   c) A-C (peso 2): Se descarta porque formaría un ciclo con los conjuntos {A, B, C}.
      - Conjuntos: {A, B, C}, {D}, {E}

   d) B-D (peso 2): Une los conjuntos {A, B, C} y {D}.
      - Conjuntos: {A, B, C, D}, {E}

   e) C-D (peso 3): Se descarta porque formaría un ciclo con los conjuntos {A, B, C, D}.
      - Conjuntos: {A, B, C, D}, {E}

   f) D-E (peso 4): Une los conjuntos {A, B, C, D} y {E}.
      - Conjunto final: {A, B, C, D, E}

   g) E-A (peso 5): Se descarta porque formaría un ciclo con el conjunto {A, B, C, D, E}.

4. **Resultado**: El MST está formado por las aristas A-B, B-C, B-D, y D-E con un peso total (mínimo) de $1 + 2 + 2 + 4 = 9$.

5.3.1. Estructura de datos para conjuntos disjuntos

En el contexto del algoritmo de Kruskal, es fundamental contar con una estructura de datos eficiente para gestionar conjuntos disjuntos (T en el pseudocódigo de la sección anterior). Esta estructura se conoce como *disjoint-set*, o como *union-find* por el tipo de operaciones que implementa:

- Union: Unir dos conjuntos disjuntos.
- Find: Determina a qué conjunto pertenece un elemento.

La eficiencia de estas operaciones es crucial para el rendimiento global del algoritmo de Kruskal. En particular, cuando se utiliza la implementación apropiada, ambas operaciones pueden ejecutarse en un tiempo casi constante, específicamente en $\mathcal{O}(\alpha(|A|))$, donde $\alpha(\cdot)$ es la inversa de la función de

Ackermann. En la práctica, $\alpha(n)$ es constante para todos los efectos computacionales, incluso para valores muy grandes de n.[2]

Utilizando esta estructura de datos, el algoritmo de Kruskal refleja una complejidad de $\mathcal{O}(|A| \log |A|)$. Esto es debido al paso de ordenación de las aristas por peso. El resto del algoritmo tiene una complejidad proporcional a $|A|$ operaciones de *union* y *find*. Por tanto, la complejidad total del algoritmo de Kruskal se expresa como $O(|A| \log |A| + |A|\alpha(|A|))$. Dado que $\alpha(|A|)$ es muy pequeño, la complejidad final se simplifica generalmente a $O(|A| \log |A|)$, haciendo que el algoritmo de Kruskal sea una propuesta muy eficiente para encontrar el MST.

5.3.2. Demostración de optimalidad

La demostración de la optimalidad del algoritmo de Kruskal se basa en dos principios:

- Invariante de ciclo mínimo: En cada paso del algoritmo, se selecciona la arista de menor peso que conecta dos subconjuntos de vértices. Si en algún momento se seleccionara una arista que no es de menor peso, entonces existiría otra arista de menor peso que no formaría un ciclo y que podría haber sido incluida en lugar de la arista seleccionada, reduciendo así el peso total del MST.
- Invariante de corte mínimo: El algoritmo asegura que, en cada paso, la arista seleccionada cumple la restricción de un MST porque cruza el corte que separa dos conjuntos de vértices no conectados y es la de menor peso entre todas las aristas que cruzan ese corte.

Por lo tanto, al agregar aristas de manera creciente por peso y solo aquellas que conectan componentes no conectados, Kruskal garantiza que el MST resultante tiene el menor peso total posible, y, por lo tanto, es óptimo.

5.4. Consideraciones adicionales

Cuando hablamos de algoritmos voraces, es importante hacer una distinción clara entre un *algoritmo voraz* y una *estrategia planteada de manera voraz*. Un *algoritmo voraz* es aquel que, utilizando una estrategia de selección localmente óptima (estrategia voraz), garantiza una solución globalmente óptima para un problema dado. Sin embargo, en muchos casos, una

[2]La función de Ackermann es una función que crece extremadamente rápido, por lo que su inversa es casi constante.

estrategia voraz no conduce a una solución óptima; en tales situaciones, no podríamos hablar propiamente de un algoritmo. En lugar de ello, podemos referirnos a una *estrategia planteada de manera voraz*, que sigue un enfoque basado en elecciones locales sin reconsiderar el impacto en la solución global, pero que no necesariamente resuelve el problema de manera óptima.

Recordemos que el término "algoritmo" implica una garantía de resolución correcta y completa del problema según los pasos definidos. Un método que sigue un planteamiento voraz pero no cumple con la condición de optimalidad no resuelve el problema (de optimización) de forma adecuada, y por lo tanto, no cumple con los requisitos para ser considerado un verdadero "algoritmo" en un sentido estricto. En otras palabras, un verdadero algoritmo voraz debe no sólo aplicar una estrategia voraz, sino también proporcionar una solución óptima. Si no lo hace, es simplemente una heurística o un enfoque voraz, pero no un algoritmo. De ahí radica la importancia de las demostraciones de optimalidad que hemos presentado en las secciones anteriores; o, en su defecto, la búsqueda de contraejemplos (como en el caso del cambio de moneda).

Capítulo 6

Vuelta atrás

En diversas ocasiones, un problema de optimización sólo se puede resolver *enumerando* todas las posibles soluciones y eligiendo la mejor. El esquema de "vuelta atrás" proporciona una forma elegante y metódica de recorrer virtualmente todo el espacio de soluciones para encontrar la solución óptima. Además, el esquema proporciona algunas ventajas adicionales que permiten acelerar el proceso si la instancia del problema lo permite.

6.1. Esquema algorítmico

En este capítulo se irá explicando el esquema algorítmico de vuelta atrás de manera incremental, empleando el problema de la mochila como hilo conductor, con el objetivo de dar forma inmediatamente a los conceptos relacionados con este esquema. Al final de esta sección se proporcionará un esquema genérico para situar cada componente del mismo.

6.1.1. El problema de la mochila (general)

La versión del problema de la mochila que se resolverá en este capítulo es la versión general, en el cual solo se permite una unidad de cada objeto (*0-1 knapsack problem*). A diferencia de la versión vista en el Capítulo 4, en esta ocasión los pesos pueden ser continuos, lo que impide (o desaconseja) la utilización de programación dinámica. Además, no se pueden fraccionar, como ocurría en el Capítulo 5.

Formalmente, se dispone de una cantidad de peso máximo W y una serie de objetos $i \in \{1, 2, ..., n\}$. Cada objeto añade peso $w_i \in \mathbb{R}$ y valor $v_i \in \mathbb{R}$ a la mochila. El objetivo es maximizar el valor de la mochila sin superar el

peso máximo de la misma. Es decir:

$$\max_{\mathbf{x}} \sum_{i=1}^{n} v_i x_i \,;\, \sum_{i=0}^{n} w_i x_i < W; \mathbf{x} \in \{0,1\}^n \tag{6.1}$$

Codificación

El primer paso para resolver un problema mediante "vuelta atrás" es encontrar una codificación que permita generar todas las soluciones posibles. Normalmente, una codificación adecuada es un vector que pertenezca al espacio de soluciones posible. Es decir, si el problema que se aborda son decisiones sobre un conjunto de n elementos, un vector $\mathbf{x} \in \mathbb{V}^n$ sería adecuado.[1]

En el caso de la mochila, podemos aprovechar la definición del problema donde se incluye un vector $\mathbf{x}$ binario que indica, para cada objeto, si se añade a la mochila o no.

Recorrido: expansión y validez

El paso clave en "vuelta atrás" es generar todas las soluciones posibles metódicamente. El espacio de soluciones se suele enumerar de manera recursiva, formando una estructura de árbol (*árbol de expansión* o *árbol de recorrido*). Aunque hay diversas formas de realizar la expansión, en esta asignatura utilizaremos el recorrido más sencillo e intuitivo: en profundidad.

El algoritmo que tendríamos para este recorrido es el siguiente:

```
función recorrido(W, Wmax, V, Vcurrent, i):
    si i = |W|
        devuelve

    recorrido(W, Wmax-W[i], V, Vcurrent+V[i], i+1)
    recorrido(W, Wmax, V, Vcurrent, i+1)
```

La Fig. 6.1 representa el recorrido realizado por la función propuesta para una instancia del problema de la mochila con 3 objetos. Este árbol contiene las soluciones parciales que contienen los nodos intermedios y las soluciones completas (llamados *nodos hoja*).

El esquema algorítmico de "vuelta atrás" recibe su nombre debido a la forma de enumerar y recorrer las soluciones. Este recorrido tiene forma de árbol, de manera que explorar las soluciones a partir de cierta solución

[1] Aquí $\mathbb{V}$ representa los posibles valores de cada decisión.

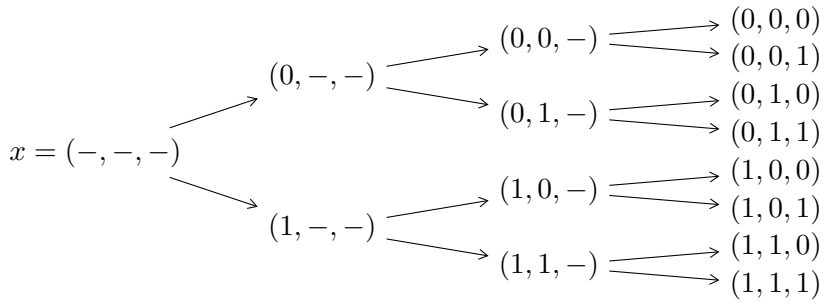


Figura 6.1: Ejemplo de árbol de recorrido para el problema de la mochila con $n = 3$. En cada nodo del árbol se muestra el valor de $\mathbf{x}$.

parcial crea nuevas ramas para este recorrido y para continuar por una rama distinta al evaluar una solución final (nodo hoja) es necesario retroceder a una solución parcial anterior, es decir, *volver atrás* en el recorrido.

Una vez se tiene el recorrido, es importante distinguir entre soluciones factibles (válidas) y no factibles. Esto se puede realizar como una comprobación en los nodos hoja. La condición de factibilidad es independiente del esquema y viene dada específicamente por el problema a resolver.

En el problema de la mochila es necesario descartar (no son factibles) las soluciones cuyo peso supera la capacidad de la mochila. A continuación se muestra nuestro algoritmo con esta comprobación adicional:

```
función recorrido(W, Wmax, V, Vcurrent, i):
    si i = |W|
        si Wmax < 0
            devuelve NO_FACTIBLE
        sino
            devuelve FACTIBLE

    recorrido(W, Wmax-W[i], V, Vcurrent+V[i], i+1)
    recorrido(W, Wmax, V, Vcurrent, i+1)
```

Si el ejemplo de recorrido de la Fig. 6.1 tenemos $W = 30, \mathbf{w} = \{25, 10, 5\}$ nos quedaría el árbol de la Fig. 6.2 con las soluciones no factibles marcadas en rojo.

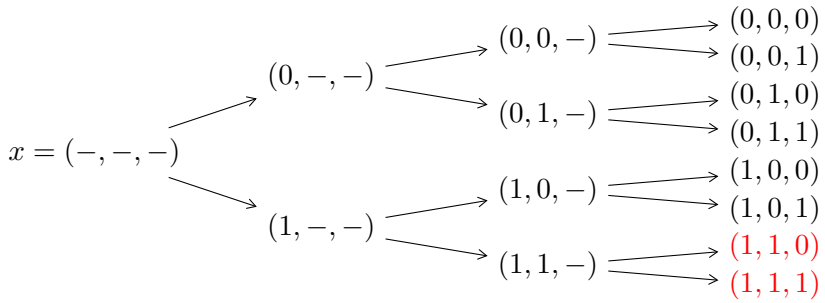


Figura 6.2: Ejemplo de árbol de recorrido para el problema de la mochila con $n = 3, W = 30, \mathbf{w} = \{25, 10, 5\}$. En cada nodo del árbol se muestra el valor de $\mathbf{x}$. En rojo se indican los nodos con soluciones no factibles (no válidas).

Mejor solución

Una vez sabemos que el algoritmo visita todas las soluciones factibles, ya podemos compararlas y guardarnos la mejor posible (optimización). A continuación se muestra el algoritmo de recorrido modificado para guardar la mejor solución. Por simplicidad, asumimos la existencia de un elemento global ajeno a la función llamado "best" y que guardará el mejor valor encontrado hasta el instante actual de la búsqueda. Como la solución que buscamos se guarda en este elemento "best", no es necesario que el algoritmo devuelva el valor.

```
función mejor_solución(W, Wmax, V, Vcurrent, i):
    si i = |W|
        si Wmax < 0
            devuelve
        sino
            si Vcurrent > best
                best := Vcurrent
            devuelve

    mejor_solución(W, Wmax-W[i], V, Vcurrent+V[i], i+1)
    mejor_solución(W, Wmax, V, Vcurrent, i+1)
```

En ocasiones puede ser interesante también buscar la secuencia de decisiones que dan lugar a la solución óptima. Esto puede ser tan sencillo como llevar un registro de las decisiones tomadas en cada llamada y guardar este registro al actualizar la mejor solución:

```
función mejores_decisiones(W, Wmax, V, Vcurrent, i, x):
    si i = |W|
        si Wmax < 0
                devuelve
            sino
                si Vcurrent > best
                    best := Vcurrent
                    bestX := x
                devuelve

    x[i] := 1
    mejores_decisiones(W, Wmax-W[i], V, Vcurrent+V[i], i+1, x)

    x[i] := 0
    mejores_decisiones(W, Wmax, V, Vcurrent, i+1, x)
```

Podas

Con el algoritmo actual podemos asegurar que encontramos la solución óptima, ya que visitamos todas las soluciones posibles y diferenciamos entre las que son factibles y las que no lo son. No obstante, el tiempo de ejecución de este esquema puede llegar a ser muy elevado ya que su complejidad es exponencial respecto a n.

Para visualizar mejor por qué esto es un problema, para obtener una solución al problema de la mochila con 50 objetos usando el algoritmo actual, asumiendo que cada nodo visitado tan solo consume 0.001 segundos, habría que esperar **35.000 años** para obtener la solución óptima. Por esto mismo, evaluar **todas** las soluciones posibles resulta **muy ineficiente**.

En la mayoría de aplicaciones prácticas el tiempo consumido para encontrar la solución óptima es muy importante, ya que los algoritmos se suelen ejecutar con bastante frecuencia. Por este motivo es útil acotar la búsqueda empleando información adicional sobre el problema de forma que se evita visitar caminos del árbol de recorrido que no tengan potencial para mejorar la solución actual. A esta técnica se la denomina "poda" porque consiste en *cortar* ramas del árbol de recorrido.

Validez Sabiendo que, dada una solución parcial, ninguna solución que la contenga va a ser factible, ¿tiene sentido seguir expandiendo a partir de esa solución? Por ejemplo, si mi mochila puede llevar 5 kg y los objetos que

estoy probando a meter acumulan ya más de 5 kg, ¿tiene sentido probar a meter algún otro objeto más?

El primer tipo de poda consiste en restringir la expansión del árbol, evitando explorar caminos que sabemos con antelación que no van a conducir a ninguna solución válida. Normalmente este tipo de poda se puede realizar comprobando las restricciones del problema antes de visitar cada nodo.

En el problema de la mochila esto se puede hacer comprobando si el peso del objeto que se intenta meter en la mochila ya supera el peso restante de la misma. Si no es el caso, se consideran las dos expansiones: cogerlo y no cogerlo; sin embargo, si el objeto no cabe solo se considera la opción de no cogerlo.

A continuación se muestra cómo se puede realizar esta comprobación en el recorrido propuesto, concretamente no se añaden objetos cuando su peso supera el peso disponible:

```
función recorrido_factible(W, Wmax, V, Vcurrent, i):
    si i = |W|
        si Vcurrent > best
            best := Vcurrent
        devuelve

    si Wmax-W[i] >= 0
        recorrido_factible(W, Wmax-W[i], V, Vcurrent+V[i], i
   +1)
    recorrido_factible(W, Wmax, V, Vcurrent, i+1)
```

El árbol de recorrido de la Fig. 6.2 quedaría como se indica en la Fig. 6.3 si se realiza la comprobación de validez durante la expansión, es decir, si se aplica la poda por validez.

Cota optimista Una vez sabemos el valor de una solución factible, ¿de qué nos sirve considerar una solución peor? Por ejemplo, si sabemos que en nuestra mochila podemos reunir un valor total de 8, ¿para qué deberíamos considerar una solución de valor 4 si sabemos que será descartada? Suponiendo que podemos saber qué valor se puede llegar a obtener al expandir un nodo, ¿interesaría expandirlo si ese valor no supera el de la mejor solución encontrada hasta el momento?

El segundo tipo de poda consiste en estimar el mejor valor que se podría alcanzar a partir de una solución parcial. A este valor se llama cota optimista. La cota optimista de un nodo debe ser **igual o mejor** que la mejor solución que se pueda encontrar recorriendo las ramas que salen de dicho nodo ya que

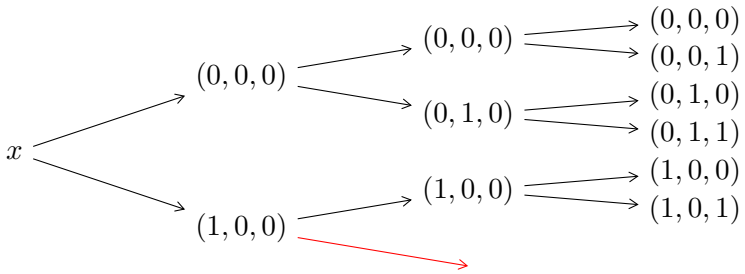


Figura 6.3: Ejemplo de árbol de recorrido para el problema de la mochila con $n = 3, W = 30, \mathbf{w} = \{25, 10, 5\}$ expandiendo únicamente las ramas factibles. En cada nodo del árbol se muestra el valor de $\mathbf{x}$. En rojo se indican los nodos y/o aristas que se han podado.

se empleará para descartar ramificaciones del recorrido. Es decir, si la cota optimista de un nodo es peor o igual que la mejor solución actual, no vale la pena expandir dicho nodo. Si esta propiedad no se cumple, se podrían estar ignorando soluciones potencialmente óptimas.

Para asegurar esta propiedad, normalmente se calculan este tipo de cotas relajando restricciones del problema. A continuación se comentan dos posibles cotas para el problema de la mochila:

- Asumir que se pueden coger todos los objetos restantes, ignorando el límite de peso.
- Asumir que se pueden fragmentar los objetos restantes. Entonces coger, de entre los objetos restantes, los que tengan mayor valor por unidad de peso primero hasta llenar la mochila.

La primera cota relaja la restricción del límite de peso, dando un valor que no puede llegar a ser peor que la mejor solución posible ya que si cupieran todos los objetos restantes, la cota optimista tendría el mismo valor que esta solución, que sería la mejor de su rama. Demostrar que esta cota es optimista resulta trivial ya que cualquier solución que contenga al menos los mismos objetos que otra (objetos específicos, no cantidad de objetos) siempre tendrá el mismo valor o mayor que esta.

La segunda cota relaja la restricción de no fraccionar objetos. Al trabajar con objetos fraccionados, se puede tratar el problema como si se dispusiese de objetos, que son fragmentos de los originales, cuyo peso es una unidad infinitesimal y su valor es proporcional al del objeto al que pertenecen. Al tener

todos los fragmentos de objeto el mismo peso, escoger los de mayor valor primero resulta óptimo. Esto se puede demostrar fácilmente: Asumiendo que tenemos 3 objetos cualesquiera con pesos W_1, W_2, W_3 y valores V_1, V_2, V_3, siempre que se cumpla que el tercer objeto tiene mayor valor por unidad de peso que los otros dos ($\frac{V_1}{W_1} < \frac{V_3}{W_3}$ y $\frac{V_2}{W_2} < \frac{V_3}{W_3}$), ocupar una parte de la mochila W_m con cualquiera de los dos primeros objetos proporcionará menor valor que ocuparla con el tercero ya que la desigualdad se sigue cumpliendo al multiplicar ambos lados por el mismo factor ($W_m \frac{V_1}{W_1} < W_m \frac{V_3}{W_3}$ y $W_m \frac{V_2}{W_2} < W_m \frac{V_3}{W_3}$). Además, cualquier combinación lineal de los dos primeros objetos para rellenar el seguirá aportando menos valor a la mochila que introducir el tercer objeto: $W_{m1} \frac{V_1}{W_1} + W_{m2} \frac{V_2}{W_2} < W_m \frac{V_3}{W_3}$ para $W_{m1} + W_{m2} = W_m$.

Al conocer una cota optimista del valor que se puede obtener expandiendo determinadas ramas, es posible evitar expandir el árbol de recorrido en determinadas situaciones para ahorrar recursos (temporales y espaciales). Normalmente se suele comparar la cota optimista con el mejor valor encontrado hasta el momento ya que no interesa encontrar soluciones que empeoren la actual, solo aquellas que puedan mejorarla. Cuanto más ajustada sea la cota (más cercana al valor real) más recursos ahorrará. A continuación se muestran las cotas mencionadas en esta sección en el mismo orden:

```
función optimista_peso(V, Vcurrent, i):
    devuelve Vcurrent + sum(V[i:])
```

```
función optimista_fracción(W, Wmax, V, Vcurrent, i):
    ordenar W[i:] y V[i:] de mayor a menor según V[j]/W[j] de
    j=i a j=|W|

    mientras que i <= |W| y Wmax > 0:
        si Wmax >= W[i]
            Vcurrent := Vcurrent+V[i]
            Wmax := Wmax-W[i]
        sino
            Vcurrent := Vcurrent + Wmax * V[i]/W[i]
            Wmax := 0

    devuelve Vcurrent
```

El algoritmo que resuelve el problema de la mochila quedaría como se muestra a continuación:

```
función mochila_con_poda(W, Wmax, V, Vcurrent, i):
    si i = |W|
```

```
        si Vcurrent > best
            best := Vcurrent
        devuelve

    si optimista_fracción(W, Wmax, V, Vcurrent, i) <= best
        devuelve

    si Wmax-W[i] >= 0
        mejor_solución(W, Wmax-W[i], V, Vcurrent+V[i], i+1)
    mejor_solución(W, Wmax, V, Vcurrent, i+1)
```

Cota pesimista La poda por cota optimista no empieza a tener efecto hasta que no se conoce el valor de una solución factible al problema. Pero, ¿es realmente necesario explorar el espacio de soluciones hasta encontrar una solución factible para empezar a utilizar la cota optimista? Es posible aprovechar la cota optimista desde el inicio de la búsqueda?

A pesar del beneficio de emplear una cota optimista, por muy ajustada que sea, no empieza a aplicarse hasta que se encuentra una solución lo suficientemente buena, como ya se ha comentado arriba. Por este motivo, es también bastante común iniciar la búsqueda con una solución válida conocida. A esto se llama cota pesimista ya que "como muy mal, la solución que encuentre tiene que ser esta". Esta solución debe ser válida, por lo que no se debe encontrar relajando las restricciones del problema. Al igual que con la cota optimista, cuanto más se acerque la cota pesimista a la solución óptima, más recursos ahorrará durante el recorrido ya que permitirá podar más nodos.

Una posible cota pesimista puede ser coger los objetos en orden de mayor valor por unidad de peso a menor, sin fragmentarlos:

```
función pesimista_greedy(W, Wmax, V, Vcurrent, i):
    ordenar W[i:] y V[i:] de mayor a menor según V[j]/W[j] de
    j=i a j=|W|

    mientras que i <= |W| y Wmax > 0:
        si Wmax >= W[i]
            Vcurrent := Vcurrent+V[i]
            Wmax := Wmax-W[i]
```

Hay cotas pesimistas que se pueden calcular durante la búsqueda, pero normalmente el coste que añaden a cada nodo no compensa el tiempo de

búsqueda que ahorran. Esta cota, por ejemplo, se podría calcular en cada nodo, pero añadiría un coste bastante elevado, de manera que es mejor usar esta cota para obtener una solución inicial para adelantar el efecto de la cota optimista. Los parámetros iniciales de esta función serán en este caso *Wmax=Wmax*, *Vcurrent=0* e *i=1*.

Es posible también calcular varias cotas pesimistas y quedarse la de mayor valor, cuando el espacio de búsqueda es potencialmente muy elevado, emplear operaciones más complejas antes de empezar la búsqueda para reducir más el tiempo que esta pueda consumir se vuelve más interesante.

Una posible cota pesimista es generar varias soluciones **válidas** aleatorias y quedarse con la de mejor valor. No obstante, esta cota no asegura una solución inicial suficientemente buena, por lo que no suele emplearse si se dispone de otra cota pesimista que aproveche las propiedades del problema.

Otros

A parte de las estrategias ya comentadas, es posible emplear conocimiento más específico del problema para acortar el tiempo de búsqueda y reducir el número de nodos visitados. Dos estrategias bastante conocidas son los pre-cálculos y el orden de expansión o de recorrido.

Cálculos previos Los cálculos previos (o pre-cálculos) consisten en adelantar el cálculo de todo aquello que no cambie durante la búsqueda. Muchas cotas optimistas es posible calcularlas antes de iniciar la búsqueda ya que no dependen del recorrido actual, tan solo del recorrido restante.

Al realizar estos cálculos antes de iniciar la búsqueda, se puede evitar la repetición de determinados cálculos, potencialmente ahorrando una gran cantidad de recursos.

Orden de recorrido Alterar el orden de recorrido es otra técnica bastante usada. En el problema de la mochila es posible ahorrar tiempo de cómputo si se evalúan los objetos en distinto orden. Por ejemplo, se podrían recorrer en orden decreciente según el valor por unidad de peso de los objetos. Este orden también podría ahorrar tiempo a la hora de calcular la cota optimista "optimista_fracción" propuesta anteriormente y la cota pesimista voraz propuesta.

Poda			O	Nodos		Tiempo	
V	O	P		Total	Hojas	Pesimista	Búsqueda
✗	✗	✗	✗	67108863	33554432	0.00e+00	1.58e+01
✓	✗	✗	✗	65359897	32110044	0.00e+00	1.57e+01
✗	✓	✗	✗	557	132	0.00e+00	1.36e-03
✓	✓	✗	✗	354	21	0.00e+00	1.30e-03
✗	✓	✓	✗	175	10	8.76e-06	6.60e-04
✓	✓	✓	✗	171	6	9.12e-06	6.61e-04
✗	✓	✗	✓	221	80	0.00e+00	1.52e-04
✓	✓	✗	✓	81	3	0.00e+00	1.01e-04
✗	✓	✓	✓	221	80	2.86e-06	1.60e-04
✓	✓	✓	✓	81	3	2.74e-06	1.01e-04

Cuadro 6.1: Tabla comparativa de la eficiencia empírica usando (✓) o sin usar (✗) cada una de las componentes del esquema algorítmico de vuelta atrás. Las componentes son: Poda (V-validez, O-optimista, P-pesimista) y Orden. La cota optimista considerada en esta tabla es "optimista_fracción".

Eficiencia

En esta sección se muestra una comparación empírica de la eficiencia del algoritmo empleando distintas versiones. La tabla 6.1 muestra distintas métricas como el número de nodos visitados y el tiempo de ejecución (media de 4 ejecuciones) de cada una de los componentes del algoritmo.

Como se puede comprobar, la cota optimista es extremadamente importante ya que permite reducir en gran cantidad el número de nodos visitados. La cota pesimista es bastante importante también, pero no se puede aplicar sin una cota optimista. Por otro lado, expandir únicamente los nodos que cumplen las restricciones del problema puede acortar bastante la búsqueda si se emplea un orden de recorrido apropiado. No obstante, este orden de recorrido puede llegar a anular el efecto de algunos componentes, por ejemplo evaluar los objetos en orden descendente de valor por unidad de peso anula el efecto de la cota pesimista ya que la primera solución probada es igual que la cota pesimista.

6.1.2. Esquema general

A continuación se muestra una versión general de este esquema algorítmico con los distintos mecanismos explicados anteriormente. En este pseudocódigo se pueden ver 4 partes: gestión de hojas, poda con almacén, poda

con cota optimista y expansión. La cota pesimista se calcularía antes de iniciar la búsqueda.

```
función vuelta_atrás(...):
    si es hoja
        si es mejor que best
            best := solución actual
        devuelve

    si este nodo está en el almacén
        si el valor almacenado es mejor o igual que el actual
            return
    almacenar el valor actual asociado al nodo actual

    si cota optimista es peor o igual que best
        devuelve

    para cada expansión posible:
        si es válida
            vuelta_atrás(...)
```

6.2. Camino óptimo en un laberinto

En esta sección se propone otro problema de ejemplo y una solución al mismo, para afianzar los conceptos explicados en este capítulo.

Supongamos que queremos resolver un laberinto de tamaño $n \times m$, empezando en el nodo (Sx, Sy) y acabando en el nodo (Ex, Ey). Tan solo se permite el desplazamiento vertical y horizontal, y no se pueden atravesar paredes. El coste de atravesar las distintas casillas (entrar en ellas) puede ser distinto para cada una, este coste se puede consultar en la matriz $L_{n \times m}$. Las paredes tienen coste -1 y ninguna casilla puede tener coste 0, además, el coste de la casilla inicial se puede ignorar al inicio del recorrido, pero no las siguientes visitas a esta casilla. El objetivo es buscar el camino de coste mínimo. Para resolver este problema (y los demás) con este esquema algorítmico se recomienda seguir la misma lógica que en la sección anterior, empezando por construir una búsqueda exhaustiva que visite todas las soluciones o las soluciones válidas y se guarde la mejor, a continuación diseñar una o varias cotas optimistas, elaborar alguna cota pesimista y finalmente evaluar la posibilidad de emplear un almacén o cambiar el orden de recorrido. También

se recomienda realizar los cálculos previos que se puedan justificar (ahorren más cómputo total del que necesitan para ser calculados al principio).

La codificación de la solución de este problema puede ser una matriz de $n \times m$ que contenga 1 en las casillas que pertenezcan al camino de la solución y 0 en las demás. Otra posible codificación es una lista con las posiciones (x, y) en el orden en que aparecen en la solución. Finalmente, otra posible codificación es una lista de direcciones (norte, sur, este, oeste) que conformen el camino hacia la casilla final desde la casilla inicial. Por simplicidad se ha decidido emplear la primera opción, ya que permite comprobar en tiempo constante si se ha visitado ya una casilla durante el recorrido.

A continuación se muestra la función que realiza la búsqueda:

```
función laberinto(L, x, y, c):
    si (x = Ex) y (y = Ey)
        si c < best
            best := c
        devuelve

    si (almacén[x][y] != -1) y (almacén[x][y] <= c)
            devuelve

    almacén[x][y] := c

    para cada par (vx, vy) en [(1, 0), (0, 1), (-1, 0), (0,
   -1)]:
        nx, ny := x+vx, y+vy

        si no 0 <= nx < |L|: salta esta iteración
        si no 0 <= ny < |L[nx]|: salta esta iteración
        si L[nx][ny] = -1: salta esta iteración
        si visitado[nx][ny]: salta esta iteración

        nc := c+L[nx][ny]

        si nc+optimista[nx][ny] < best
            visitado[nx][ny] := True
            laberinto(L, nx, ny, nc)
            visitado[nx][ny] := False
```

Donde *visitado* es una matriz de valores booleanos que indican si ya se ha pasado por una casilla durante el recorrido, para evitar bucles y porque si

ya has pasado por una casilla, volver a pasar por la misma tan solo aumenta el coste sin aportar progreso hacia la casilla objetivo.

Almacén contiene el coste mínimo de llegar a cada casilla desde la casilla inicial. Si ya se ha visitado anteriormente una casilla, el coste total de alcanzar la solución es el coste mínimo desde esa casilla hasta la final sumado al coste de alcanzar la misma. La primera parte es la misma para ambas visitas, por lo que si se ha alcanzado la casilla anteriormente con un coste menor, el coste actual no puede ser mejor que el que se alcanzó a raíz de la anterior visita por lo que se puede detener la expansión del nodo en cuestión.

La cota optimista es el coste de alcanzar la solución en el número mínimo de movimientos. Para cada movimiento en horizontal se suma el coste mínimo de la columna que se visita (sin contar las paredes, que cuestan -1) y cada movimiento en vertical suma el coste mínimo de la fila que se visita (también sin contar las paredes). De esta forma, se asegura que la cota es optimista ya que si el camino óptimo contiene un movimiento que aumenta la distancia euclídea entre la casilla actual y el destino, debe contener otro movimiento que lo contrarreste (que lo acerque al destino), la cota optimista considera el mínimo de esos dos, o incluso un valor menor y tan solo una vez. La restricción que relaja esta cota es la de que la solución debe ser un camino válido (esta cota considera la posibilidad de moverse a casillas no contiguas). Un ejemplo de esta cota en un laberinto de 3 filas y 3 columnas, empezando en la esquina superior izquierda y acabando en la esquina inferior derecha, en el que los costes sean los siguientes:

1	10	1
10	10	1
10	1	10

El valor de la cota optimista sería 4: derecha (coste 1 por la casilla (3, 2)), derecha (coste 1 por la casilla (1, 3)), abajo (coste 1 por la casilla (2, 3)), abajo (coste 1 por la casilla (3, 2)). Si en una de las columnas o filas todos los valores fueran 10, atravesar dicha fila o columna siempre tendrá coste 10, por lo que la cota no se alejaría tanto como una cota trivial como puede ser el coste mínimo multiplicado por la distancia euclídea entre cada casilla y la casilla final.

La cota pesimista es el algoritmo de Dijkstra. Este algoritmo resuelve el problema de encontrar el camino de coste mínimo entre dos nodos de un grafo. Como este problema se puede resolver planteándolo como una búsqueda de camino de menor coste, esta cota elimina completamente la necesidad de realizar la búsqueda. No obstante, el tiempo total de ejecución

puede llegar a ser más elevado en determinadas ocasiones ya que el algoritmo de Dijkstra tiene complejidad $O(|V|^2)$ donde $|V| = n \cdot m$ es el número de vértices del grafo (casillas del laberinto), por lo que la complejidad de esta cota sería $O(n^2 \cdot m^2)$.

Es importante destacar que en este ejemplo, la cota pesimista puede llegar a consumir bastante tiempo. A menudo hay que valorar la posibilidad de emplear distintas estrategias en función del tamaño del problema. En la Sección **??** se comenta en más detalle en esta cuestión y se ponen ejemplos para clarificar esta idea.

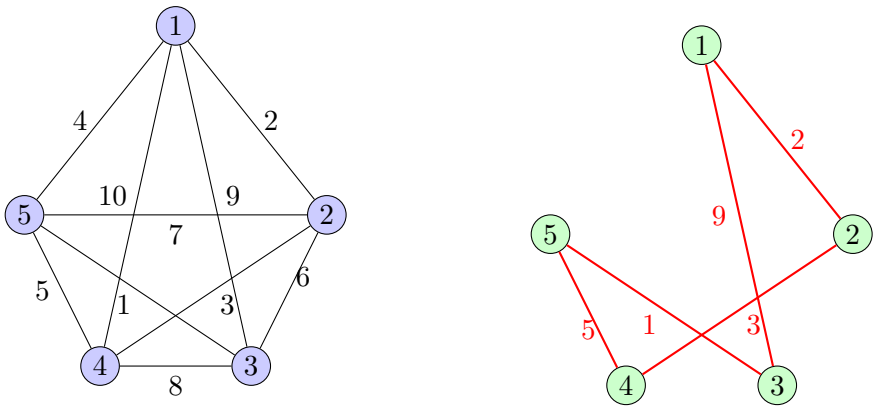
Por otro lado, también es importante tener en cuenta que un correcto análisis previo del problema a resolver es imprescindible. Este problema se puede resolver empleando el algoritmo de Dijkstra, un algoritmo voraz empleado para encontrar caminos de coste mínimo. A menudo hay problemas que se pueden resolver con esquemas algorítmicos más sencillos y es importante identificar cuándo y si es más eficiente emplear dichos esquemas antes de aplicar algoritmos de vuelta atrás.

6.3. El viajante de comercio

El problema del viajante de comercio (Travelling Salesman Problem, o TSP) trata de resolver la siguiente situación: Un comerciante quiere visitar una serie de ciudades y volver a la ciudad inicial. La intención del comerciante es visitar cada ciudad únicamente una vez. El coste de desplazamiento entre cada par de ciudades puede ser distinto. El objetivo del comerciante es minimizar el coste total (suma de los costes de desplazamiento) del camino que cumple las condiciones ya comentadas.

Este problema se suele modelar con grafos. Cada ciudad es un vértice y el camino entre cada par de ciudades es una arista. Dependiendo de qué versión del problema se trate de resolver, el coste del camino entre dos ciudades puede ser el mismo en ambas direcciones (grafo no dirigido) o ser distinto (grafo dirigido). Además, también es posible que no haya un camino entre todos los pares de ciudades, pero siempre debe haber al menos un camino de entrada y uno de salida a cada ciudad. Para no complicar mucho las cotas propuestas, se considerará la versión modelada con un grafo no dirigido. Un ejemplo del problema formalizado con grafos y su solución se presenta en la Fig. 6.4.

La solución de este problema se puede codificar como una lista que contenga las ciudades visitadas en orden cronológico. Esta lista debe empezar y acabar por la ciudad inicial y no tener otras ocurrencias de la misma, el



(a) Grafo original (b) Solución.

Figura 6.4: Ejemplo visual del problema del viajante de comercio. El nodo de partida es (1).

resto de ciudades deben aparecer exactamente una vez y para cada par de ciudades contiguas (c_i, c_{i+1}), debe existir un camino que lleve de la ciudad c_i a la ciudad c_{i+1}. El coste de la solución es la suma de los costes de cada par de ciudades (c_i, c_{i+1}). En el ejemplo de la Fig. 6.4, el recorrido de mínimo coste es el formado por $1 \to 2 \to 4 \to 5 \to 3 \to 1$, con peso 20.

Una propuesta de solución a este problema mediante vuelta atrás es la siguiente:

```
función viajante(C, actual, coste, restantes):
    si restantes = 0
        si coste < best
            best := coste
        devolver

    para cada destino aún por visitar:
        si existe camino de ciudad actual a ciudad destino
            si restantes = 1 o destino != inicio
                marcar destino como visitado
                nuevoCoste := coste + C[actual][destino]
                si nuevoCoste + optimista < best
                    viajante(C, destino, nuevoCoste, restantes
   -1)
                marcar destino como no visitado
```

En este caso, C es una matriz cuadrada que contiene el coste de desplazarse de una ciudad origen a una ciudad destino para cada entrada de la matriz $C[origen][destino]$. Durante la expansión tan solo se permite visitar la ciudad inicial si no quedan ciudades por visitar.

Una posible cota optimista puede ser la suma del coste mínimo de salir de cada ciudad aún por visitar, sin contar la que se está a punto de visitar, para la cual se usa el coste de desplazarse a dicha cuidad desde la ciudad actual. Otra posible cota optimista podría ser calcular el árbol de expansión de coste mínimo antes de iniciar la búsqueda, utilizando el algoritmo de Kruskal (ver Capítulo 5).

Por otro lado, una posible cota pesimista para este problema podría ser calculada mediante el siguiente proceso: empezar con un grafo vacío. Para cada ciudad que tenga únicamente dos ciudades conectadas, añadir al grafo estas aristas y los nodos que conectan. Estos nodos y aristas deben pertenecer al camino óptimo obligatoriamente ya que el camino debe contener todos los nodos y cada ciudad debe ser visitada una única vez, por lo que si un nodo solo tiene una arista en este grafo, el camino final contendrá dos veces el otro nodo que conecta dicha arista. En este instante se pueden tener una o varias componentes conexas. En estas componentes conexas deben haber uno o dos nodos que solo tengan una arista, estos nodos se llamarán "punta" de la componente conexa a la que pertenecen. Para añadir el resto de las ciudades se procede de la siguiente manera: de entre las aristas que no pertenecen aún al grafo y conecten un nodo punta con otro de otra componente conexa o con una ciudad que aún no está en el grafo, se añade la de menor coste. Cuando no queden ciudades por añadir, se añaden las aristas de menor coste que unan dos componentes conexas a través de sus puntas, cuando solo quede una componente conexa, se añadirá la arista de menor coste que una sus puntas. Si en algún momento del proceso ninguna arista se puede añadir, no existe solución.

Con las estructuras de datos apropiadas se puede calcular esta cota pesimista sin acarrear una complejidad computacional exponencial. Por ejemplo, si en todo momento se conserva una lista que contenga los nodos punta y se conozca a qué componente conexa pertenece cada uno, es posible mantener una lista ordenada con las aristas que conectan estos nodos con otros pero aún no pertenecen al grafo.

6.4. Consideraciones adicionales

Es importante tener en cuenta algunas consideraciones al diseñar algoritmos con el esquema de vuelta atrás. La primera de las consideraciones es que el recorrido, y por tanto la eficiencia, puede depender bastante de la codificación que se haya empleado. El número de nodos hoja de la búsqueda exhaustiva inicial se puede calcular, por lo que es recomendable asegurar que las expansiones realizadas cubran todas las soluciones posibles empleando tallas pequeñas ya que si algún detalle de la expansión produce como consecuencia la discriminación de soluciones válidas de manera no intencionada, es posible que se descarte la solución óptima y este hecho se pase por alto. Es necesario por tanto, que se puedan recorrer todas las soluciones válidas de manera estructurada y ordenada, evitando visitar soluciones repetidas o dejar soluciones prometedoras sin visitar.

En cuanto a la complejidad, los algoritmos de vuelta atrás suelen tener una cota superior de complejidad exponencial ($O(x^n)$) que no se puede reducir de ninguna forma ya que a fin de cuentas el espacio de soluciones se expande exponencialmente en función de la talla del problema. No obstante, con las cotas apropiadas, la complejidad promedio puede llegar a disminuir en gran cantidad. Por este motivo es importante evaluar adecuadamente los algoritmos implementados. Además, como ya se ha comentado en capítulos anteriores, las cotas de complejidad son límites asintóticos, por lo que es posible que para ciertas tallas, un algoritmo cuya cota de complejidad sea mayor resuelva el problema en menos tiempo.

Capítulo 7

Métodos heurísticos

Los métodos heurísticos representan una familia de estrategias para abordar problemas que, debido a su complejidad inherente, resisten las soluciones por métodos convencionales o exactos. Estos métodos son particularmente útiles cuando tenemos restricciones de tiempo o recursos computacionales, situaciones en las que aplicar algoritmos exactos sería inútil o directamente impracticable.

Los métodos heurísticos, por su naturaleza, no garantizan encontrar la solución óptima a un problema; sin embargo, contienen estrategias para encontrar soluciones "suficientemente" buenas dentro de un margen aceptable de eficiencia. Su diseño se basa en reglas prácticas o experimentales, conocidas como heurísticas, que guían el proceso de búsqueda de soluciones. Estas reglas se derivan del conocimiento del problema específico y la experiencia previa en dominios similares, proporcionando una aproximación más pragmática que prioriza la obtención de resultados en tiempos razonables. Un ejemplo de este tipo de aproximaciones son los *algoritmos voraces* (Capítulo 5).

Debido a su generalidad, los métodos heurísticos no pueden estudiarse como un esquema general (a diferencia de los vistos en anteriores capítulos), sino que deben conocerse uno a uno.

7.1. Algoritmo aleatorio

Una estrategia simple para resolver un problema combinatorio es tomar decisiones al azar. Es decir, escoger en cada momento una opción disponible de manera aleatoria.

Aunque depende del problema concreto, en la mayoría de casos de in-

terés un algoritmo aleatorio no sólo no nos dará la solución óptima sino que dará una solución pobre. Sin embargo, la complejidad de cada una de estas soluciones es lineal. Por tanto, uno puede repetir K veces la ejecución del algoritmo, aumentando así la probabilidad de encontrar una buena solución (con complejidad $\mathcal{O}(nK)$). Nótese que cuando K tiende a infinito, la probabilidad de encontrar la solución óptima se aproxima a 1.

Esta opción también puede ser muy útil para complementar un esquema de vuelta atrás, siendo una alternativa sencilla como solución inicial o cota pesimista cuando no se dispone de otra que emplee un conocimiento o entendimiento mayor del problema.

7.2. Búsqueda local

La búsqueda local es un método heurístico que se basa en la exploración iterativa de las soluciones vecinas de una solución dada. Comienza con una solución inicial y, en cada iteración, se examinan las soluciones que están en su "vecindad", es decir, aquellas que se pueden alcanzar mediante pequeñas modificaciones de la solución actual. Si se encuentra una mejor solución en esta vecindad, el algoritmo se mueve a ella y repite el proceso.

Este método es simple y eficiente para problemas donde la estructura del espacio de soluciones permite una exploración rápida de las soluciones vecinas. Sin embargo, uno de los mayores desafíos de la búsqueda local es evitar quedar atrapado en óptimos locales, es decir, soluciones que son mejores que todas las vecinas, pero no necesariamente cercana a la solución globalmente óptima.

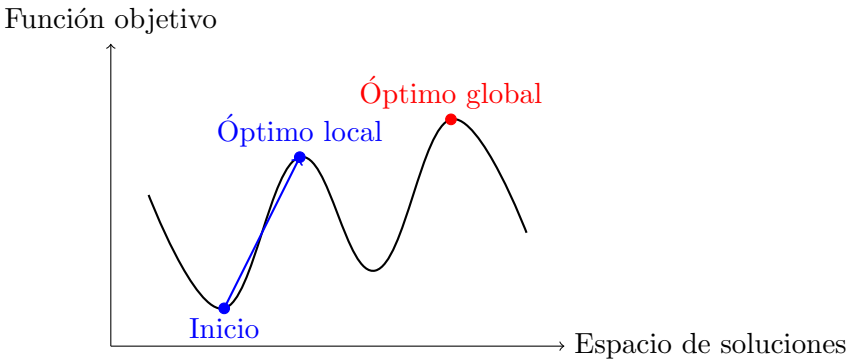


Figura 7.1: Ejemplo de búsqueda local en un espacio de soluciones.

En la Figura 7.1, se muestra un ejemplo de cómo funciona la búsqueda

local. Partiendo de un punto inicial, el algoritmo encuentra un óptimo local, pero no necesariamente alcanza el óptimo global, ya que este se encuentra más allá de la vecindad explorada.

7.3. Heurísticas avanzadas

Además de los métodos heurísticos discutidos anteriormente, existen otros enfoques destacados que se utilizan en la optimización y la toma de decisiones en problemas particularmente complejos. A continuación, se presentan breves reseñas de algunos de estos métodos, incluyendo los más conocidos en el campo de la Inteligencia Artificial:

- **Temple simulado (Simulated Annealing)**: Este algoritmo se inspira en el proceso físico de templado, donde un material se enfría lentamente para alcanzar un estado de mínima energía. En optimización, el temple simulado busca soluciones explorando el espacio y aceptando, ocasionalmente, soluciones peores para evitar quedar atrapado en óptimos locales. Con el tiempo, la probabilidad de aceptar peores soluciones disminuye (se va *templando*), lo que permite converger hacia una solución cercana al óptimo global.

- **Búsqueda Tabú (Tabu Search)**: Este método es una extensión de la búsqueda local que utiliza una memoria adaptativa para evitar ciclos y ayudar a escapar de los óptimos locales. La "lista tabú" almacena movimientos recientes o soluciones que no deben ser revisadas en un número determinado de iteraciones, lo que fuerza al algoritmo a explorar nuevas áreas del espacio de soluciones.

- **Optimización por enjambre de partículas (Particle Swarm Optimization, PSO)**: Inspirado en el comportamiento social de los animales, como bandadas de pájaros o bancos de peces, PSO optimiza una función al mover un conjunto de partículas (soluciones) a través del espacio de búsqueda. Cada partícula ajusta su posición en función de su experiencia individual y la experiencia de sus vecinos, buscando el equilibrio entre la exploración del espacio de soluciones y la explotación de las mejores soluciones encontradas.

- **Colonia de Hormigas (Ant Colony Optimization, ACO)**: Este algoritmo está inspirado en el comportamiento de las hormigas para encontrar caminos hacia fuentes de alimento. Las hormigas exploran el espacio de soluciones, y las mejores soluciones son reforzadas mediante

la “deposición de feromonas”, lo que guía a las siguientes generaciones de hormigas hacia soluciones de mayor calidad.

- **Algoritmos Genéticos**: Inspirados en la evolución biológica, los algoritmos genéticos optimizan una población de soluciones mediante procesos de selección, cruce y mutación. A través de iteraciones sucesivas, estos algoritmos buscan mejorar la aptitud de la población hacia soluciones más cercanas al óptimo global. Son especialmente útiles en problemas donde el espacio de soluciones es grande y complejo, y donde es útil explorar una amplia variedad de soluciones potenciales aunque nunca se alcance la óptima.

- **Monte Carlo Tree Search (MCTS)**: MCTS es un algoritmo utilizado principalmente en la toma de decisiones secuenciales, como en juegos de estrategia (por ejemplo, ajedrez o Go). El método combina la búsqueda sistemática en un árbol de decisiones con simulaciones aleatorias, evaluando el resultado de cada decisión a través de múltiples simulaciones. MCTS equilibra la exploración de nuevas opciones con la explotación de las mejores opciones conocidas, y es especialmente poderoso para problemas donde las posibles decisiones y resultados son vastos y complejos.

Estos métodos heurísticos, aunque variados en su enfoque y aplicación, comparten el objetivo de encontrar soluciones eficientes a problemas de optimización complejos. Cada uno ofrece ventajas particulares según la naturaleza del problema y las restricciones computacionales.

Capítulo 8

Programación lineal

La programación lineal es una técnica matemática utilizada para optimizar un objetivo lineal sujeto a un conjunto de restricciones también lineales. Este paradigma se utiliza especialmente en la investigación operativa, una de las disciplinas que estudian cómo tomar decisiones óptimas bajo ciertas condiciones. Los problemas de programación lineal pueden modelar situaciones donde se busca maximizar o minimizar una función objetivo, como maximizar ganancias o minimizar costes, dado un conjunto de recursos limitados.

8.1. Formulación

Un problema de programación lineal se puede formular matemáticamente mediante los siguientes elementos:

- Las variables $x_1, x_2, \ldots, x_n \geq 0$.
- Una función lineal a maximizar:[1]

$$Z = c_1x_1 + c_2x_2 + \cdots + c_nx_n$$

- Un conjunto de restricciones lineales:

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \leq b_1$$

$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \leq b_2$$

[1] Nótese que cualquier problema cuyo objetivo sea minimizar se puede convertir a uno análogo de maximización.

$$\vdots$$

$$a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \leq b_m$$

En resumen, el objetivo es encontrar los valores de $x_1, x_2, \ldots, x_n$ que maximicen Z, satisfaciendo todas las restricciones.

Como ejemplo, podemos considerar una empresa que produce dos productos: A y B. La empresa desea maximizar sus ganancias. El producto A genera una ganancia de 40 euros por unidad y el producto B genera 30 euros por unidad. Sin embargo, la producción de ambos productos está limitada por la disponibilidad de recursos:

- Cada unidad de A requiere 2 horas de trabajo y 3 unidades de materia prima.
- Cada unidad de B requiere 1 hora de trabajo y 2 unidades de materia prima.

Si la empresa dispone de 100 horas de trabajo y 180 unidades de materia prima, el problema de programación lineal se formula de la siguiente manera:

$$\text{Maximizar } Z = 40x_1 + 30x_2$$

Sujeto a:

$$2x_1 + x_2 \leq 100$$

$$3x_1 + 2x_2 \leq 180$$

$$x_1, x_2 \geq 0$$

En este caso, la solución al problema es $x_1 = 0$ y $x_2 = 90$, que cumple con las restricciones del problema y arroja un valor de Z máximo (2700).

La programación lineal es una formulación interesante porque puede resolverse de manera exacta mediante algoritmos diseñados específicamente para este propósito. Estos métodos no solo garantizan encontrar la solución óptima cuando existe, sino que también identifican situaciones en las que el problema no tiene solución, ya sea porque es *inviable* (no hay valores que cumplan simultáneamente todas las restricciones) o porque es *no acotado* (la función objetivo puede aumentar o disminuir indefinidamente). Esta capacidad de detectar y clasificar el estado del problema es fundamental en aplicaciones prácticas, donde es crucial conocer si las restricciones planteadas son coherentes y manejables.

En la siguiente sección, se explicará en detalle el método "Simplex", uno de los algoritmos más populares y robustos para programación lineal, ampliamente utilizado por su eficiencia en la práctica.

8.2. El método Simplex

El método Simplex es uno de los algoritmos más utilizados para resolver problemas de programación lineal. Desarrollado por George Dantzig en 1947, este método explora las soluciones factibles del problema de programación lineal, buscando maximizar (o minimizar) la función objetivo. Su principal ventaja radica en su capacidad para encontrar la solución óptima de manera eficiente en la mayoría de los casos prácticos.

De manera intuitiva, el método Simplex aprovecha la estructura geométrica de los problemas de programación lineal. En un espacio definido por n variables y m restricciones, las soluciones factibles forman un politopo, es decir, una figura geométrica de múltiples caras. El Simplex comienza en un vértice de este politopo y avanza hacia vértices adyacentes que mejoran el valor de la función objetivo, deteniéndose cuando encuentra el vértice que maximiza (o minimiza) la función objetivo.

Desde un punto de vista geométrico, el método Simplex explora el politopo formado por las restricciones del problema de programación lineal. Cada vértice de este politopo representa una solución factible básica, que satisface exactamente n restricciones en igualdad. La función objetivo se representa como un plano que se desplaza sobre el politopo, buscando el vértice más alto (o más bajo, en caso de minimización). Esta conexión geométrica explica por qué el Simplex avanza solo entre vértices, descartando regiones que no contienen la solución óptima.

Consideremos el problema de programación lineal en dos dimensiones, con las siguientes restricciones y función objetivo:

$$\text{Maximizar } Z = 3x_1 + 2x_2$$

Sujeto a:

$$x_1 + x_2 \leq 4$$

$$2x_1 + x_2 \leq 5$$

$$x_1 \geq 0, \quad x_2 \geq 0$$

Como se ha mencionado, las restricciones definen un area factible de soluciones, que en este caso es un polígono en el plano x_1–x_2. En la Fig. 8.1,

el área sombreada representa el conjunto de soluciones factibles, es decir, el polígono donde todas las restricciones son satisfechas. Los vértices de este polígono son puntos clave, ya que el método Simplex explora estos vértices para encontrar la solución óptima.

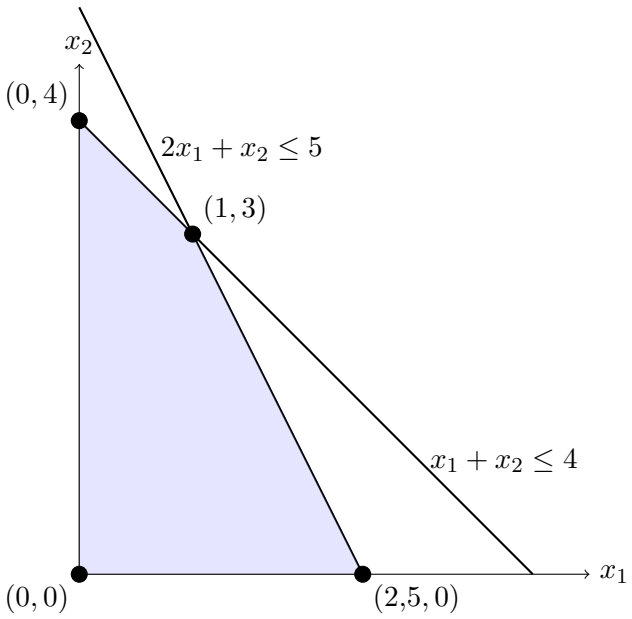


Figura 8.1: Conjunto factible en dos dimensiones.

8.2.1. Funcionamiento general

El algoritmo Simplex es un método iterativo que permite resolver problemas de programación lineal encontrando el valor óptimo de la función objetivo. Su funcionamiento se basa en la exploración de los vértices del politopo que define el conjunto de soluciones factibles, es decir, el espacio donde se cumplen todas las restricciones del problema.

El algoritmo comienza con una solución inicial, llamada **solución básica inicial**, que corresponde a uno de los vértices del politopo. Esta solución se construye seleccionando un subconjunto de las variables del problema (llamadas **variables básicas**), mientras que el resto de las variables se fijan a cero (denominadas **variables no básicas**). El objetivo del algoritmo es mejorar esta solución moviéndose hacia vértices adyacentes que ofrezcan un mejor valor de la función objetivo, hasta alcanzar el vértice óptimo.

En cada paso, el Simplex identifica una **variable de entrada**, que es una variable no básica que tiene el potencial de mejorar el valor de la función objetivo. Esta variable entra en la base, lo que significa que se le asignará un valor positivo en la solución. Para mantener la factibilidad de la solución, es necesario seleccionar también una **variable de salida**, que es una de las variables básicas que se reemplazará por la nueva variable en la base. Este proceso se realiza utilizando el llamado *pivoteo*, que actualiza las ecuaciones del problema para reflejar el cambio.

El proceso de pivoteo garantiza que la nueva solución siga siendo factible (cumple todas las restricciones) y, al mismo tiempo, mejora o mantiene el valor de la función objetivo. El algoritmo continúa iterando de esta forma hasta que no se pueda mejorar más la función objetivo, es decir, hasta que todas las posibles variables de entrada dejarían la función objetivo igual o peor. En este punto, se ha alcanzado el vértice óptimo.

El Simplex también es capaz de detectar si el problema no tiene solución. Si el valor de la función objetivo puede aumentar indefinidamente (en el caso de maximización) o disminuir sin límite (en el caso de minimización), el problema se considera **no acotado**. Si no hay ninguna solución factible que cumpla con todas las restricciones, el problema se clasifica como **inviable**.

De manera geométrica, siguiendo con el ejemplo de la sección anterior, el Simplex podría comenzar en el vértice $(0, 0)$ y luego moverse a lo largo del borde del polígono (ver Fig. 8.2):

1. Primera iteración: Mueve de $(0, 0)$ a $(0, 4)$, aumentando Z a 8.
2. Segunda iteración: Se mueve de $(0, 4)$ a $(1, 3)$, mejorando aún más el valor de Z a 9.
3. Conclusión: $(1, 3)$ es el punto óptimo para este problema de programación lineal dado que maximiza la función objetivo dentro del conjunto factible.

8.2.2. Complejidad

El método Simplex es uno de los algoritmos más utilizados y estudiados en el campo de la optimización, debido a su capacidad para resolver problemas de programación lineal de manera eficiente en la práctica. Sin embargo, la complejidad teórica del Simplex ha sido un tema de estudio y debate durante muchos años.

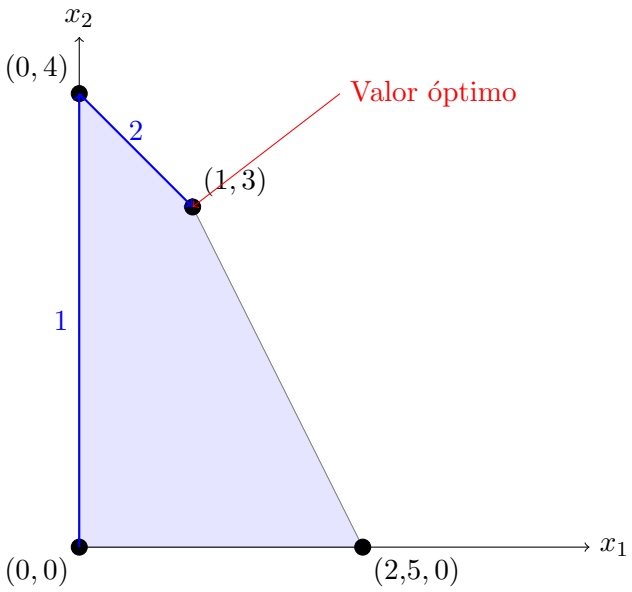


Figura 8.2: Iteraciones del Simplex en el polígono factible.

En la práctica, el método Simplex es extremadamente eficiente y resuelve la mayoría de los problemas de programación lineal en un tiempo razonable. Esto se debe a que, en la mayoría de los casos, el número de iteraciones necesarias para alcanzar la solución óptima es lineal con respecto al número de restricciones y variables. Por esta razón, el Simplex es ampliamente utilizado en aplicaciones reales, donde la velocidad y la eficiencia son cruciales.

A pesar de su excelente rendimiento práctico, el método Simplex tiene una complejidad teórica en el peor caso que es exponencial. Esto significa que, en ciertas configuraciones extremas del problema, el número de iteraciones necesarias puede crecer exponencialmente con respecto al número de variables y restricciones. Ejemplos de problemas en los que el Simplex experimenta este comportamiento son muy raros, pero existen y han sido objeto de investigación.

A raíz de la complejidad exponencial en el peor caso, surgieron otros algoritmos para resolver problemas de programación lineal, como el método de puntos interiores. Este método, introducido por Karmarkar en 1984, tiene una complejidad polinómica en el peor caso, lo que lo hace teóricamente más atractivo. Sin embargo, el método Simplex sigue siendo preferido en muchas aplicaciones debido a su simplicidad y rendimiento eficiente en la mayoría de los problemas prácticos.

8.3. Implementación

En la actualidad, la resolución de problemas de programación lineal no requiere realizar los cálculos manualmente, ya que existen multitud de librerías y herramientas computacionales que implementan el método Simplex y otros algoritmos eficientes.[2] Por este motivo, la mayor dificultad en la programación lineal no radica en la resolución computacional del problema, sino en su correcta formulación. Es esencial identificar con precisión la función objetivo, las variables de decisión y las restricciones que modelan el problema real. Antes de implementar un problema en código, es crucial entender a fondo su estructura y asegurarse de que representa correctamente la situación que se desea optimizar.

[2]En Python, una de las librerías más utilizadas es `SciPy`, que incluye el módulo `scipy.optimize.linprog`, el cual permite resolver problemas de programación lineal de manera rápida y sencilla. Otras librerías, como `PuLP` y `OR-Tools`, también son ampliamente utilizadas.

Apéndice A

Funciones Big O

Vamos a realizar demostraciones sencillas utilizando la definición formal de la notación Big O para dos de los ejemplos del Capítulo 2, especificando las constantes c y n_0 y mostrando cómo se aplican para demostrar que cada función pertenece a su respectiva clase de complejidad.

A.1. Ejemplo 1

- Complejidad lineal $T(n) = 2n + 3$
- **Afirmación:** $T(n) \in \mathcal{O}(n)$

Para demostrar que $T(n) \in \mathcal{O}(n)$, necesitamos encontrar constantes c y n_0 tales que:

$$0 \leq f(n) \leq c \cdot n \text{ para todo } n \geq n_0.$$

Tomemos $c = 5$ y $n_0 = 1$. Entonces para $n \geq n_0$:

$$T(n) = 2n + 3 \leq 2n + 3n = 5n.$$

Por lo tanto, tenemos que:

$$0 \leq 2n + 3 \leq 5n \text{ para todo } n \geq 1.$$

Esto muestra que $T(n) \in \mathcal{O}(n)$ con $c = 5$ y $n_0 = 1$.

A.2. Ejemplo 2

- Complejidad cúbica $T(n) = 2n^3 + n + 3$
- **Afirmación:** $T(n) \in \mathcal{O}(n^3)$

Para demostrar que $T(n) \in \mathcal{O}(n^3)$, necesitamos encontrar constantes c y n_0 tales que:

$$0 \leq f(n) \leq c \cdot n^3 \text{ para todo } n \geq n_0.$$

Tomemos $c = 4$ y $n_0 = 1$. Entonces para $n \geq n_0$:

$$f(n) = 2n^3 + n + 3 \leq 2n^3 + n^3 + n^3 = 4n^3.$$

Por lo tanto, tenemos que:

$$0 \leq 2n^3 + n + 3 \leq 4n^3 \text{ para todo } n \geq 1.$$

Esto muestra que $f(n) \in \mathcal{O}(n^3)$ con $c = 4$ y $n_0 = 1$.

A.2.1. Ejemplo 3

- Complejidad exponencial $T(n) = 2^n + n^2$
- **Afirmación:** $T(n) \in \mathcal{O}(2^n)$

Vamos a realizar la demostración de que una función exponencial con un aditivo cuadrático pertenece a la clase de complejidad $\mathcal{O}(2^n)$.

Para demostrar que $T(n)$ es $\mathcal{O}(2^n)$, necesitamos encontrar constantes c y n_0 tales que:

$$0 \leq f(n) \leq c \cdot 2^n \text{ para todo } n \geq n_0.$$

Observamos que n^2 crece mucho más lentamente que 2^n cuando n es grande. Para hacer esto más formal, podemos considerar cómo se comportan los términos cuando n aumenta:

- 2^n crece exponencialmente,
- n^2 crece polinomialmente.

Elegiremos $c = 3$ y $n_0 = 10$. Esto es algo arbitrario, pero son valores que van a hacer que claramente el término exponencial domine.

Para $n \geq n_0 = 10$, necesitamos mostrar que:

$$2^n + n^2 \leq 3 \cdot 2^n$$

Dividimos todo por 2^n:

$$1 + \frac{n^2}{2^n} \leq 3$$

El término $\frac{n^2}{2^n}$ tiende a cero a medida que n aumenta debido al crecimiento exponencialmente más rápido de 2^n comparado con el crecimiento polinomial de n^2. Específicamente, para $n = 10$, el cálculo muestra:

$$\frac{100}{2^{10}} \approx 0{,}0977$$

Para valores mayores de n, este término será aún menor, asegurando que:

$$1 + \frac{n^2}{2^n} \leq 3$$

Así queda ejemplificado que para $n \geq 10$, $f(n) = 2^n + n^2$ se mantiene por debajo de $3 \cdot 2^n$, confirmando que $f(n)$ es $\mathcal{O}(2^n)$ con $c = 3$ y $n_0 = 10$.

Apéndice B

Optimalidad de algoritmos voraces

B.1. Algoritmo voraz para el problema de la mochila continua

Supongamos que existe una solución mejor que la obtenida por el algoritmo voraz, es decir, una selección de fracciones de objetos que proporciona un valor total mayor al que obtenemos utilizando la estrategia voraz. Llamemos a esta supuesta solución óptima S^* y al valor total de S^* como $V(S^*)$. Denotemos por S la solución generada por el algoritmo voraz y su valor total como $V(S)$. Supongamos entonces que $V(S^*) > V(S)$.

El algoritmo voraz selecciona los objetos en orden descendente de la relación valor/peso $\frac{v_i}{w_i}$. Así, si tomamos los objetos seleccionados por S^* y S, podemos dividir los objetos en los que ambos algoritmos tienen en común (o fracciones de ellos) y aquellos que son diferentes. Como el algoritmo voraz selecciona en cada paso el objeto con la mayor relación $\frac{v_i}{w_i}$ disponible, cualquier objeto i que S^* incluya y S no haya seleccionado ya (o no haya seleccionado en su totalidad) debe tener una relación $\frac{v_i}{w_i}$ menor que al menos un objeto que S haya seleccionado.

Formalicemos esto:

1. Sea j un índice tal que j es el primer objeto donde S^* y S difieren, es decir, j es un objeto incluido en S^* que no está completamente incluido en S.

2. Sea i un objeto en S que fue seleccionado antes que j por el algoritmo voraz, de manera que $\frac{v_i}{w_i} \geq \frac{v_j}{w_j}$.

3. Ahora, comparemos el valor adicional que S^* podría obtener sobre S. Como S incluye objetos con una mayor relación $\frac{v_i}{w_i}$ que $\frac{v_j}{w_j}$, cualquier fracción de peso que S^* asigne a un objeto de menor relación $\frac{v_j}{w_j}$ debería haber sido asignada a un objeto con una mayor relación $\frac{v_i}{w_i}$ para maximizar el valor total.

Matemáticamente, si consideramos un pequeño incremento de peso Δw, el incremento en el valor para S sería mayor o igual que para S^* porque:

$$\Delta v(S) = \frac{v_i}{w_i} \cdot \Delta w \geq \frac{v_j}{w_j} \cdot \Delta w = \Delta v(S^*)$$

Debido a que $\frac{v_i}{w_i} \geq \frac{v_j}{w_j}$, el valor total generado por S al seleccionar primero los objetos con mayor $\frac{v_i}{w_i}$ no puede ser inferior al valor total generado por S^*, lo cual contradice la suposición de que $V(S^*) > V(S)$.

Por lo tanto, no puede existir una solución S^* con un valor total mayor que S, lo que implica que el algoritmo voraz genera la solución óptima.

B.2. Optimalidad del algoritmo de Kruskal

Consideremos un grafo G con vértices V y aristas E. Un corte en G es una partición de V en dos subconjuntos S y $V - S$. Una arista cruza el corte si conecta un vértice en S con un vértice en $V - S$. La arista de menor peso que cruza entre S y $V - S$ (es decir, entre los vértices dentro de S y los que están fuera de S) siempre pertenece a algún árbol de expansión mínima (MST) de G.

El algoritmo de Kruskal selecciona aristas en orden ascendente de peso. En cada paso, la arista seleccionada es la de menor peso que conecta dos componentes distintos (dos subconjuntos de vértices que aún no están conectados por las aristas seleccionadas hasta ese momento). Esta arista, por definición, es la de menor peso que cruza el corte entre esos dos componentes. Por el principio del corte mínimo, esta arista debe pertenecer a algún MST. Dado que el algoritmo de Kruskal siempre selecciona aristas de esta manera, todas las aristas seleccionadas pertenecen a un MST.

Supongamos, por contradicción, que el árbol T construido por el algoritmo de Kruskal no es un MST, y que existe un árbol T^* que es un MST con un peso total menor que T. Dado que T^* es un MST, debe contener todas las aristas seleccionadas por Kruskal, ya que cada arista seleccionada es la mínima posible para conectar dos componentes distintos. Sin embargo, si existiera una arista en T^* con un peso menor que alguna arista en T,

esta arista habría sido seleccionada por Kruskal antes de la otra, lo cual contradice la suposición de que T^* tiene un peso total menor que T.

Por lo tanto, T debe ser un MST, y el algoritmo de Kruskal es óptimo.