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.

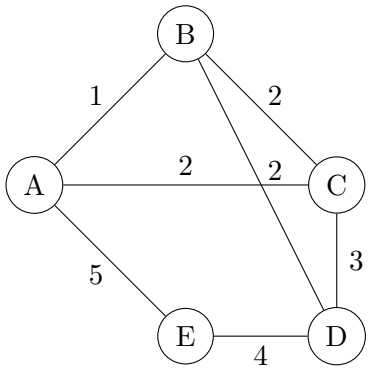

Figura 5.1: Ejemplo de grafo ponderado.

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).