Complejidad computacional de problemas y el análisis y diseño de algoritmos

Satu Elisa Schaeffer

30 de septiembre de 2014

Prefacio

Este documento ha sido preparado para la unidad de aprendizaje *Análisis y Diseño de Algoritmos* (MECBS5126) del Programa de Posgrado en Ingeniería de Sistemas (PISIS) de la Facultad de Ingeniería Mecánica y Eléctrica (FIME) de la Universidad Autónoma de Nuevo León (UANL), ubicada en San Nicolás de los Garza, en el estado de Nuevó León en México. Es una unidad de nivel doctoral, aunque también se aceptan estudiantes de maestría de buen nivel con el permiso de sus asesores de tesis. La mayoría de los estudiantes de PISIS no son computólogos, y aún menos del campo de teoría de computación, por lo cuál la presentación ha sido simplificada de lo usual de programas de posgrado con enfoque exclusivo en la computación.

Mucho de este material se debe a los materiales que use yo misma cuando estudié los mismos temas en la Universidad Politécnica de Helsinki (TKK; ahora se llama Aalto University) es Espoo, Finlandia. En el año 2000 trabajé como asistente de enseñanza en un curso de análisis y diseño de algoritmos bajo la instrucción del Dr. Eljas Soisalon-Soininen; en aquel entonces, en colaboración con otro asistente de enseñanza Riku Saikkonen compilamos un documento en finés utilizando LaTeX (basado en las diapositivas escritas a mano por el Dr. Soisalon-Soininen y en el libro de texto de Cormen et al. [4]). Al empezar a compilar el presente documento, mi primer recurso era el documento antiguo en finés, por lo cual varios ejemplos son inspirados por el trabajo que hicimos Riku y yo en el año 2000.

Otra inspiración importante fue la docencia del Dr. Ilkka Niemelä durante mis estudios de posgrado en el TKK, específicamente el curso de complejidad computacional que impartió, basado en el libro de texto clásico de Christos Papadimitriou [15]. Dr. Niemelä me proporcionó todas sus diapositivas del curso para la preparación de este documento, por lo cual le agradezco mucho.

Otras partes del documento están prestadas y modificadas de trabajos anteriores míos, en la revisión de los cuales he contado con el apoyo de muchas personas, los más importantes siendo el asesor de mi tesis doctoral, Dr. Pekka Orponen, y el oponente de la misma tesis, Dr. Josep Díaz.

Índice general

Capítulo 1

Definiciones matemáticas y computacionales

1.1. Conjuntos y permutaciones

Si los elementos a, b y c forman un *conjunto*, se asigna una letra mayúscula para denotar el conjunto y ofrece una lista de los elementos según la siguiente notación:

$$A = \{a, b, c\}\,. \tag{1.1}$$

Si el elemento a pertenece a un *conjunto* A, se escribe $a \in A$. Si el elemento a *no* pertenece al conjunto A, se escribe $a \notin A$.

$\mathbb{Z}$ es el conjunto de números enteros y $\mathbb{R}$ es el conjunto de números reales.

Si un conjunto está *ordenado*, se puede definir para cada par de elementos si uno está mayor, menor o igual a otro según el orden definido.

El *cardinalidad* de un conjunto A es el número de elementos que pertenecen al conjunto. Se denota por $|A|$. Por ejemplo, la cardinalidad del conjunto $A = \{a, b, c\}$ es tres, $|a| = 3$. La *unión* de dos conjuntos A y B contiene todos los elementos de A y todos los elementos de B y se denota por $A \cup B$. La *intersección* de dos conjuntos A y B se denota por $A \cap B$ y contiene solamente los elementos que pertenecen a ambos conjuntos:

$$c \in (A \cap B) \Leftrightarrow \big((c \in A) \wedge (c \in B)\big), \tag{1.2}$$

donde $\Leftrightarrow$ significa que la expresión a la izquierda siendo verdadera implica que la expresión a la derecha también tiene que ser válida y viceversa.

Nota que necesariamente $|A \cup B| \geq \max\{|A|, |B|\}$ y $|A \cap B| \leq \min\{|A|, |B|\}$. El *conjunto vacío* que no contiene ningún elemento se denota con $\emptyset$.

1.1.1. Subconjuntos

Un *subconjunto* $B \subseteq A$ es un conjunto que contiene algún parte de los elementos del conjunto A. Si no se permite incluir todos los elementos de A en B, escribimos $B \subset A$ y aplica $|B| < |A|$. Por ejemplo, si $A = \{a, b, c\}$, $B = \{b, c\}$ y $C = \{c, d\}$, tenemos que $B \subseteq A$, pero C no es un subconjunto: $C \nsubseteq A$.

El *complemento* de un conjunto A es el conjunto $\bar{A}$ que contiene cada elemento del universo que no está incluido en A,

$$\bar{A} = \{a \mid a \notin A\}. \tag{1.3}$$

El *complemento* de un subconjunto $B \subseteq A$ se denota por $A \setminus B$ (leer: A sin B) y es el conjunto de todos elementos $a \in A$ tales que $a \notin B$. En general, la *diferencia* de dos conjuntos A y B es el conjunto de los elementos que están en A pero no en B,

$$A \setminus B = \{a \mid a \in A, a \notin B\}. \tag{1.4}$$

Dado un conjunto A, el conjunto de *todos sus subconjuntos* se denota con 2^A. El número total de tales subconjuntos es $2^{|A|}$. Por ejemplo, si $A = \{a, b, c\}$, tenemos $|A| = 3$, $2^3 = 8$ y

$$2^A = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{b, c\}, \{a, c\}, A\}.$$

Un k-subconjunto de un conjunto A es un subconjunto de k elementos de A. Si $|A| = n$, el número de los diferentes combinaciones de k elementos posibles es el *coeficiente binomio*

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}, \tag{1.5}$$

donde $n!$ es el *factorial*. El factorial está definido a todo número entero $k \in \mathbb{Z}$ tal que

$$k! = k \cdot (k-1) \cdot (k-2) \cdot \ldots \cdot 2 \cdot 1. \tag{1.6}$$

Una aproximación muy útil del factorial es la *aproximación de Stirling*

$$k! \approx k^k e^{-k} \sqrt{2\pi k} \tag{1.7}$$

y una versión mejorada por Gosper es

$$k! \approx \sqrt{\pi(2k + \frac{1}{3})} k^k e^k, \tag{1.8}$$

donde $e \approx 2,718$ es el número neperiano.

El coeficiente binomio permite una definición recursiva:

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k}, \text{ si } 0 < k < n \\ 1, \text{ en otro caso.} \end{cases} \tag{1.9}$$

El rango (cerrado) es un subconjunto de un conjunto ordenado desde el elemento a hasta el elemento b se marca con $[a, b]$. El rango *abierto* que no incluye el primer ni el último elemento sino todos los elementos intermedios c tales que $a < c < b$ se denota con (a, b). Si uno de los puntos extremos sí se incluye y el otro no, la notación es $(a, b]$ o $[a, b)$, dependiendo de cuál extremo está incluido.

El conjunto $A \times B$ es el conjunto de todos los pares ordenados (a, b) de elementos de los conjuntos A y B tal que $a \in A$ y $b \in B$,

$$A \times B = \{(a, b) \mid a \in A, b \in B\}. \tag{1.10}$$

El conjunto $A \times B$ se llama el *producto cartesiano*. Se puede generalizar a más de dos conjuntos: para n conjuntos $A_1, A_2, \ldots, A_n$, es el conjunto de todos los n-eadas ordenadas

$$A_1 \times A_2 \times \ldots \times A_n = \{(a_1, a_2, \ldots, a_n) \mid a_i \in A_i, i \in [1, n]\}. \tag{1.11}$$

1.1.2. Relaciones

Una *relación* $\mathcal{R} \subseteq A \times B$ es un subconjunto de pares. Una relación entre un conjunto $A = \{a_1, a_2, \ldots, a_n\}$ y otro conjunto $B = \{b_1, b_2, \ldots, b_m\}$ es un subconjunto de asociaciones de elementos de A con los elementos de B. Se escribe $(a, b) \in \mathcal{R}$ o alternativamente $a\mathcal{R}b$.

Una relación se puede representar en la forma de una $n \times m$ matriz, donde $n = |A|$ y $m = |B|$. La representación es una matriz binaria $\mathbf{A}$ tal que el elemento a_{ij} de la matriz $\mathbf{A}$ tiene el valor uno si y sólo si los elementos a_i y b_j forman uno de los pares incluidos en la relación $\mathcal{R}$:

$$a_{ij} = \begin{cases} 1, & \text{si } (a_i, b_j) \in \mathcal{R}, \\ 0, & \text{si } (a_i, b_j) \notin \mathcal{R}. \end{cases} \tag{1.12}$$

Una relación $\mathcal{R} \subseteq A \times A$ entre un conjunto y si mismo se dice una relación *en el conjunto* A. Tal relación es *transitiva* si $\forall a, b, c \in A$ tales que $a\mathcal{R}b$ y $b\mathcal{R}c$, también aplica que $a\mathcal{R}c$. Una relación $\mathcal{R}$ en A es *reflexiva* si para todo $a \in A$, aplica que $a\mathcal{R}a$. Una relación $\mathcal{R}$ en A es *simétrica* si aplica que

$$(a_i, a_j) \in \mathcal{R} \Leftrightarrow (a_j, a_i) \in \mathcal{R}. \tag{1.13}$$

Esto implica que la $n \times n$ matriz $\mathbf{A}$, donde $n = |A|$, es también simétrica: $\mathbf{A} = \mathbf{A^T}$.

El *clausura transitiva* de una relación $\mathcal{R} \subseteq A \times A$ es la relación transitiva $\mathcal{R}_T \subseteq A \times A$ de *cardinalidad menor* que contiene $\mathcal{R}$. Entonces, $a\mathcal{R}_T b$ para $a, b \in A$ sí y sólo si $\exists c_0, c_1, \ldots, c_k \in A$ tales que $c_0 = a$, $c_k = b$ y $c_i \mathcal{R}_T c_{i+1}$ para todo $i \in [0, n)$.

La *clausura reflexiva* de la relación $\mathcal{R}$ en A es la relación mínima reflexiva $\mathcal{R}_R$ en A que contiene $\mathcal{R}$: para todo $a, b \in A$ aplica $a\mathcal{R}_R b$ si y sólo si $a = b$ o $a\mathcal{R}b$.

En la *clausura reflexiva y transitiva* $\mathcal{R}^*$ aplica que $(a, b) \in \mathcal{R}^*$ si y sólo si o $a = b$ o existe un $c \in A$ tal que $(a, c) \in \mathcal{R}$ y $(c, b) \in \mathcal{R}^*$.

1.1.3. Permutaciones

Una *permutación* de un conjunto A es un orden de los elementos. Si $|A| = n$, A tiene $n!$ permutaciones. Por ejemplo, si $A = \{a, b, c, d\}$, las $4! = 24$ permutaciones de A son

$$\begin{array}{cccccc} abcd & abdc & adbc & adcb & acbd & acdb \\ bacd & badc & bdac & bdca & bcad & bcda \\ cabd & cadb & cdab & cdba & cbad & cbda \\ dabc & dacb & dcab & dcba & dbac & dbca \end{array}$$

El número de k-subconjuntos *ordenados* del conjunto A de orden $|A| = n$ es

$$k! \cdot \binom{n}{k} = \frac{n!}{(n-k)!}. \tag{1.14}$$

Por ejemplo, si $A = \{a, b, c, d\}$ y $k = 3$, los 12 ordenes son

$$\begin{array}{cccccc} abc & abd & acd & bac & bad & bcd \\ cab & cad & cbd & dab & dbc & dcb \end{array}$$

1.2. Mapeos y funciones

Un *mapeo* (inglés: map o mapping) $g : A \to B$ es un tipo de relación que asocia algunos elementos de A a elementos de B, pero no necesariamente todos. Un mapeo es técnicamente equivalente a una relación $\mathcal{R}_g$, pero típicamente se escribe $g(a) = b$ para significar que $(a, b) \in \mathcal{R}_g$. Nota que se puede asignar varios elementos de A a un elemento de B y viceversa.

El conjunto A se llama el *dominio* de la relación y el conjunto B el *rango*. El conjunto de los elementos $b_i \in B$ para las cuales aplica $(a, b_i) \in \mathcal{R}_g$ (es decir $g(a) = b_i$) son la *imagen* de a en B. El conjunto de los elementos de A que corresponden a un cierto elemento $b \in B$ tal que $(a, b) \in \mathcal{R}_g$ es la *imagen inversa* de b, $g^{-1}(b)$.

Un mapeo $g : A \to B$ es *sobreyectivo* (también *epiyectivo*; inglés: surjection o onto) si para cada elemento del rango está asignada algún elemento del dominio:

$$\forall b \in B \exists a \in A \text{ tal que } g(a) = b. \tag{1.15}$$

Un mapeo es *inyectivo* (inglés: injection o one-to-one) si

$$\forall a_1, a_2 \in A \text{ tales que } g(a_1) = g(a_2) \Rightarrow a_1 = a_2, \tag{1.16}$$

o sea, la imagen inversa de todo $b \in B$ tiene cardinalidad menor o igual a uno. Aquí $\Rightarrow$ significa que la expresión a la izquierda siendo verdadera implica que la expresión a la derecha también tiene que ser válida, pero no es necesariamente así en la dirección contraria: $g(a_1) = g(a_2)$ no implica que necesariamente sea $a_1 = a_2$, puede ser que $a_1 \neq a_2$ también.

Un mapeo es *biyectivo* si es inyectivo y sobreyectivo: todo elemento de A tiene una imagen distinta y la unión de las imágenes cubren todo el rango B.

Una *función* $f : A \to B$ es un mapeo sobreyectivo pero no necesariamente una inyectiva. El símbolo f refiere a la función misma, mientras $f(x)$ refiere al valor de la función en x, o sea, el elemento del rango B a lo cual corresponde el elemento del dominio $x \in A$.

Una función $f : \mathbb{R} \to \mathbb{R}$ es *convexa* si para todo x_1, x_2 y $\alpha \in [0, 1]$

$$f(\alpha x_1 + (1 - \alpha)x_2) \leq \alpha f(x_1) + (1 - \alpha) f(x_2). \tag{1.17}$$

El *valor absoluto* de $x \in \mathbb{R}$ es una función de valores reales a valores reales positivos:

$$|x| = \begin{cases} x, & \text{si } x \geq 0 \\ -x, & \text{si } x < 0. \end{cases} \tag{1.18}$$

1.3. Función exponencial

La función exponencial se define por un serie (la expansión de Taylor) como

$$\exp(x) = e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots \tag{1.19}$$

donde $e \approx 2,718281828$ es la *constante de Napier*. La figura 1.1 muestra su forma. Una definición alternativa es por límite:

$$e^x = \lim_{n \to \infty} \left(1 + \frac{x}{n}\right)^n. \tag{1.20}$$

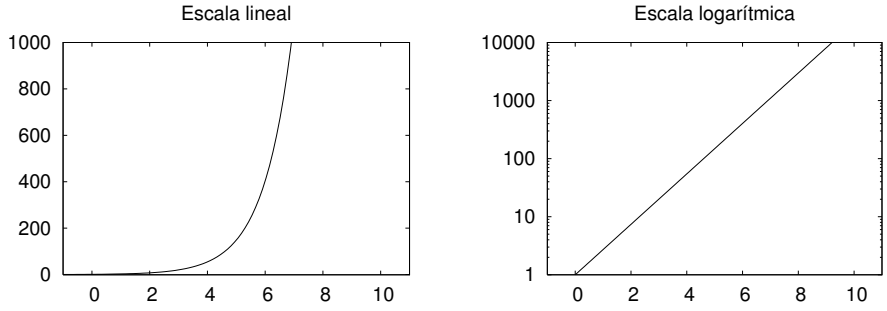

Figura 1.1: Gráficas de la función exponencial $f(x) = \exp(x)$: a la izquierda, en escala lineal, mientras a la derecha, el eje y tiene escala *logarítmica* (ver la sección 1.4).

La función exponencial comparte las propiedades de exponentes sin importar la base b, que en el caso de $\exp(x)$ es e:

$$\begin{aligned} b^0 &= 1 \\ b^1 &= b \\ b^{a+c} &= b^a b^c \\ b^{ac} &= (b^a)^c \\ b^{-a} &= \left(\tfrac{1}{b}\right)^a = \tfrac{1}{b^a} \end{aligned} \tag{1.21}$$

mientras también tiene otras propiedades interesantes: es su propio derivativo

$$\mathrm{D}(e^x) = e^x. \tag{1.22}$$

Aplican las siguientes cotas y aproximaciones que resultan útiles en varios contextos de análisis de algoritmos:

$$\begin{gathered} \left(1 - \tfrac{1}{x}\right)^k \approx e^{-\frac{k}{x}} \\ 1 - x \leq e^{-x} \\ \frac{x^x}{x!} < e^x \\ \text{para } |x| \leq 1 : \; e^x(1 - x^2) \leq 1 + x \leq e^x \\ \text{para } k \ll x : \; 1 - \tfrac{k}{x} \approx e^{-\frac{k}{x}}. \end{gathered} \tag{1.23}$$

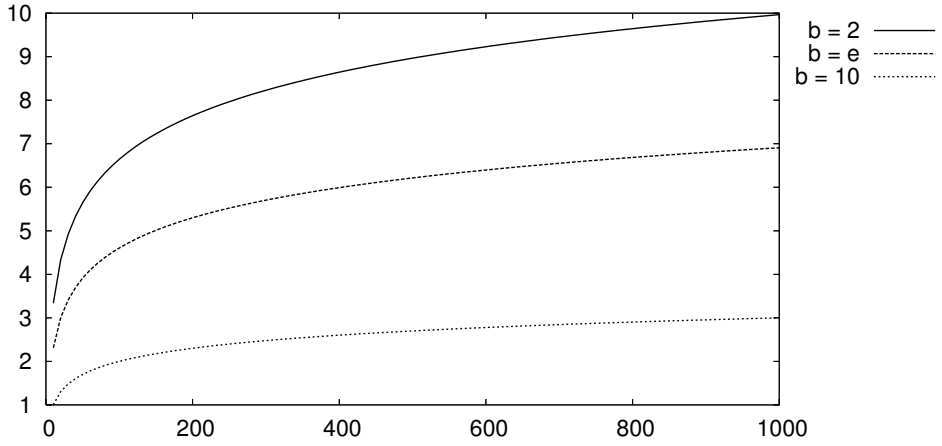

Figura 1.2: Las funciones logarítmicas por tres valores típicos de la base: $b \in \{2, e, 10\}$, donde $e \approx 2,718$ es el número neperiano.

1.4. Logaritmos

El *logaritmo* en base b es una función que se define por la ecuación

$$b^{\log_b x} = x, \tag{1.24}$$

donde $x > 0$. Eso quiere decir que si $\log_b x = y$, $b^y = x$. Se define además que $\log_b 1 = 0$. Se puede realizar cambios de base b a otra base b':

$$\log_{b'}(x) = \frac{\log_b(x)}{\log_b(b')}. \tag{1.25}$$

El logaritmo con la base e se llama el *logaritmo natural* o *neperiano*. En la figura 1.2, se muestra las funciones para tres valores de base. Típicamente se escribe $\ln x$ para $\log_e x$ y $\log x$ sin base normalmente se interpreta como $\log_{10}$, aunque en computación la interpretación $\log x = \log_2 x$ es muy común cuando $x \in \mathbb{Z}$.

Típicamente en el análisis de algoritmos se utiliza $b = 2$. El logaritmo es una función *inyectiva*,

$$\log_b x = \log_b y \Longrightarrow x = y, \tag{1.26}$$

y también es una función *creciente*

$$x > y \Longrightarrow \log_b x > \log_b y. \tag{1.27}$$

Cuenta con algunas propiedades de interés que ayuden a simplificar fórmulas:

$$\begin{aligned}
&\text{Logaritmo de multiplicación:} && \log_b(x \cdot y) = \log_b x + \log_b y \\
&\text{Logaritmo de división:} && \log_b\left(\tfrac{x}{y}\right) = \log_b x - \log_b y \\
&\text{Logaritmo con potencia:} && \log_b x^c = c \log_b x \\
&\text{Logaritmo en el exponente:} && x^{\log_b y} = y^{\log_b x}, \\
&\text{Logaritmo de un factorial:} && \log_b(n!) = \textstyle\sum_{i=1}^{n} \log_b i
\end{aligned} \tag{1.28}$$

la segunda igualdad siendo consecuencia de la primera, la penúltima igualdad siendo consecuencia de la tercera y la última siendo consecuencia de la primera.

Una variación de la aproximación de Stirling para el factorial cuenta con logaritmos:

$$\ln k! \approx k \ln k - k \tag{1.29}$$

que aplica para valores grandes de k. Una forma que aplica más generalmente es

$$\ln k! \approx \frac{2k+1}{2} \ln k - k + \frac{\ln(2\pi)}{2}. \tag{1.30}$$

1.5. Sucesiones

Dado una *sucesión* de números, asignamos un índice a cada número de la sucesión. Por ejemplo, el primer número será x_1, el segundo x_2, etcétera, hasta el último número, que marcamos con x_n, n siendo el largo de la sucesión. Cada uno de los $x_1, x_2, \ldots$ se llama un *término*. Para hablar de un término arbitrario cualquiera, usamos x_i (o alguna otra letra de índice), donde se supone de una manera implícita que $1 \leq i \leq n$.

La suma de los términos de una sucesión $x_1, x_2, \ldots, x_n$ se llama una *serie* S y la sumación tiene su notación especial donde un índice i corre desde el valor uno hasta el valor n:

$$S = x_1 + x_2 + x_3 + \ldots + x_n = \sum_{i=1}^{n} x_i. \tag{1.31}$$

Una serie *parcial* S_k es la suma hasta el término x_k:

$$S_k = \sum_{i=1}^{k} x_i. \tag{1.32}$$

Tal que $S_n = S$. Depende la sucesión de qué manera calcular el valor de la serie S o un subserie S_k.

También se puede definir el producto de una sucesión:

$$P = x_1 \cdot x_2 \cdot x_3 \cdot \ldots \cdot x_n = \prod_{i=1}^{n} x_i. \tag{1.33}$$

1.5.1. Series aritméticas

En las series *aritméticas*, la diferencia entre cada par de términos subsecuentes x_i y x_{i+1} es una constante d:

$$x_{i+1} - x_i = d \tag{1.34}$$

para todo $i \in [0, n-1]$ si es una sucesión finita y para todo $i \in [0, \infty)$ en el otro caso. La suma de una sucesión infinita es infinita positiva ∞ si la diferencia $d > 0$ y infinita negativa $-\infty$ si la diferencia $d < 0$ es negativa. La subserie de los n términos primeros de una sucesión aritmética es

$$S_n = \sum_{i=0}^{n-1}(x_1 + i \cdot d) = \frac{n(x_1 + x_n)}{2}. \tag{1.35}$$

Se puede derivar la fórmula por pensar en la suma como un triángulo de encima de columnas que todos tienen una base de altura x_1, y en cada columna, el cambio de altura es d a en comparación con la columna anterior. Se deja como un ejercicio hacer el cálculo que corresponde.

Otra manera de derivar el resultado es a través de la *ecuación recursiva* de los términos:

$$x_{i+1} = x_i + d. \tag{1.36}$$

Aplicamos esta ecuación de una manera repetitiva para encontrar una ecuación para x_n:

$$\begin{aligned} x_2 &= x_1 + d, \\ x_3 &= x_2 + d = (x_1 + d) + d = x_1 + 2d, \\ x_4 &= x_3 + d = (x_1 + 2d) + d = x_1 + 3d, \\ &\vdots \\ x_n &= x_1 + (n-1)d. \end{aligned} \tag{1.37}$$

En la otra dirección, tenemos

$$\begin{aligned} x_{n-1} &= x_n - d, \\ x_{n-2} &= x_{n-1} - d = (x_n - d) - d = x_n - 2d, \\ x_{n-3} &= x_{n-2} - d = (x_n - 2d) - d = x_n - 3d, \\ &\vdots \\ x_{n-k} &= x_{n-k-1} - d = x_n - (n-k)d, \\ x_1 &= x_{n-(n-1)} = x_n - \Big(n - \big(n - (n-1)\big)\Big)d = x_n - (n-1)d. \end{aligned} \tag{1.38}$$

Al sumar $x_1 + x_2 + \ldots + x_n$ escrito por la formulación de la ecuación 1.37 con lo mismo pero escrito por la formulación de la ecuación 1.38 nos da $2S_n$ y se simplifica a dar el mismo resultado. El cálculo se deja como ejercicio.

También hay que notas que transformaciones del índice hacen que las series se ven más complejas:

$$\sum_{i=1}^{n}(n - i + 1) = \sum_{i=1}^{n} i. \tag{1.39}$$

1.5.2. Series geométricas

En una serie *geométrica*, la *proporción* de un término con el siguiente es constante:

$$x_{i+1} = x_i \cdot d. \tag{1.40}$$

Si $|d| > 1$, la serie infinita tiene un valor infinita el signo de cual depende de los signos de x_1 y d. Cuando $|d| < 1$, incluido las series infinitas tienen una suma finita:

$$S = \sum_{i=0}^{\infty} d^i x_i = \frac{x_1}{1-d}. \tag{1.41}$$

Nota que hay que empezar del índice $i =$ para lograr que el primer término tenga el valor $x_1 = d^0 x_1$.

Las sumas parciales hasta término n cuando $d \neq 1$ tienen la fórmula siguiente:

$$S_n = \sum_{i=0}^{n} d^i x_i = \frac{x_1(1 - d^{n+1})}{1-d}. \tag{1.42}$$

Cuando $d = 1$, todos los términos son iguales a x_1 y la suma parcial es simplemente $S_n = n \cdot x_1$.

Un ejemplo es la subserie donde $x_1 = 1$ y $d \neq 1$, o sea, la suma parcial de la sucesión, $1, d, d^2, d^3, \ldots$:

$$S_n = \sum_{i=0}^{n} d^i = \frac{1(1-d^{n+1})}{1-d} = \frac{d^{n+1}-1}{d-1}. \tag{1.43}$$

Entonces, si $d = 2$ y $x_1 = 1$, tenemos la suma de potencias de dos:

$$S_n = \sum_{i=0}^{n} 2^i = \frac{2^{n+1}-1}{2-1} = 2^{n+1} - 1. \tag{1.44}$$

1.5.3. Otras series

Hay algunas series interesantes donde el índice de la sumación aparece en el término. Aquí incluyamos unos ejemplos:

$$\begin{aligned}
\sum_{i=1}^{n} i^2 &= \frac{n(n+1)(2n+1)}{6} \\
\sum_{i=0}^{n} i \cdot d^i &= \frac{n \cdot d^{n+2} - (n+1) \cdot d^{n+1} + d}{(d-1)^2}, \text{ donde } d \neq 1 \\
\sum_{i=0}^{\infty} d^i &= \frac{1}{1-d}, \text{ donde } d \neq 1 \\
\sum_{i=n}^{\infty} k^i &= \frac{k^n}{1-k}, \text{ donde } x \in (0,1) \\
\sum_{i=1}^{\infty} i^{-2} &= \frac{\pi^2}{6}.
\end{aligned} \tag{1.45}$$

1.6. Demostraciones

Para analizar si o no algo es valido en todo caso, se necesita utilizar los conceptos de sistemas matemáticos: los hechos universales se llaman *axiomas*. Además se cuenta con *definiciones* donde se fija el sentido de algún formalismo, notación o terminología. La meta es derivar de los axiomas y las definiciones, algunos *teoremas*, que son proposiciones verdaderas. Cada teorema trata de una cierta propiedad de interés. Los teoremas "auxiliares" que uno tiene que demostrar para llegar al resultado final deseado se llaman *lemas*. La cadena de pasos que establecen que un teorema sea verdad es llama la *demostración* (o también *prueba*) del teorema.

Una técnica esencial de demostración es la *inducción matemática*, donde primero se establece que una condición inicial c_1 es válida y verdadera (el paso base), y después deriva que si c_k es válida y verdadera, también c_{k+1} lo es (el paso inductivo).

Por ejemplo, para verificar que la suma de la ecuación 1.35 (en la página 9) es la fórmula correcta para la sucesión $3, 5, 8, 11, 14, 17, 20, \ldots, 213$, primero sacamos los valores $x_1 = d = 3$ y mostramos que $S_1 = x_1 + 0 \cdot d = 3 + 0 = 3 = x_1$ es válida (el paso base). Después hay que mostrar que $S_n + x_{n+1} = S_{n+1}$ (el paso inductivo). Cuando uno establece esto, la demostración de que la validez de la fórmula de la ecuación 1.35 está completa.

1.7. Lógica matemática

1.7.1. Lógica booleana

La lógica booleana de trata de un conjunto X de *variables* que también se llama *átomos* $x_1, x_2, \ldots$. Una variable se interpreta a tener el valor "verdad" (se denota con el símbolo $\top$) o "falso" (se denota con el símbolo $\bot$). La *negación* de una variable x_i se denota con $\neg x_i$:

$$\neg x_i = \begin{cases} \top, & \text{si } x_i = \bot, \\ \bot, & \text{si } x_i = \top. \end{cases} \tag{1.46}$$

Se puede formar *expresiones* de las variables con los símbolos. Las expresiones básicas son los *literales* x_i y $\neg x_i$. Además se puede formar expresiones con los *conectivos* $\vee$ ("o"), $\wedge$ ("y") — también $\neg$ se considera un conectivo. Si ϕ_1 y ϕ_2 son expresiones booleanas, también $(\phi_1 \vee \phi_2)$, $(\phi_1 \wedge \phi_2)$ y $\neg\phi_1$ lo son. Por ejemplo, $((x_1 \vee x_2) \wedge \neg x_3)$ es una expresión booleana pero $((x_1 \vee x_2)\neg x_3)$ no lo es. Para simplificar las expresiones, existen las convenciones siguientes:

$$\begin{aligned} \bigvee_{i=1}^{n} \varphi_i \quad &\text{significa} \quad \varphi_1 \vee \cdots \vee \varphi_n \\ \bigwedge_{i=1}^{n} \varphi_i \quad &\text{significa} \quad \varphi_1 \wedge \cdots \wedge \varphi_n \\ \phi_1 \to \phi_2 \quad &\text{significa} \quad \neg\phi_1 \vee \phi_2 \\ \phi_1 \leftrightarrow \phi_2 \quad &\text{significa} \quad (\neg\phi_1 \vee \phi_2) \wedge (\neg\phi_2 \vee \phi_1). \end{aligned} \tag{1.47}$$

Lo de $\to$ se llama una *implicación* y lo de $\leftrightarrow$ se llama una *equivalencia*.

Es siempre recomendable marcar con paréntesis la precedencia deseada de los operadores lógicos, pero en su ausencia, la precedencia se interpreta en el orden (de la más fuerte al más débil): $\neg$, $\vee$, $\wedge$, $\to$, $\leftrightarrow$. Por ejemplo, la expresión

$$\neg x_1 \vee x_2 \to x_3 \leftrightarrow \neg x_4 \wedge x_1 \vee x_3 \tag{1.48}$$

debería ser interpretada como

$$((((\neg x_1) \vee x_2) \rightarrow x_3) \leftrightarrow ((\neg x_4) \wedge (x_1 \vee x_3))). \tag{1.49}$$

Como las variables se interpreta como verdaderas o falsas, también es posible evaluar si o no es verdadera cualquier expresión booleana. Una *asignación de verdad* T es un mapeo de una subconjunto finito $X' \subset X$ al conjunto de valores $\{\top, \bot\}$. Denota por $X(\phi)$ el conjunto de variables booleana que aparezcan en una expresión ϕ. Una asignación de valores $T : X' \rightarrow \{\top, \bot\}$ es *adecuada* para ϕ si $X(\phi) \subseteq X'$. Escribimos $x_i \in T$ si $T(x_i) = \top$ y $x_i \notin T$ si $T(x_i) = \bot$.

Si T *satisface* a ϕ, lo que se denota por $T \models \phi$ tiene siguiente la definición inductiva:

(I) Si $\phi \in X'$, aplica $T \models \phi$ si y sólo si $T(\phi) = \top$.

(II) Si $\phi = \neg\phi'$, aplica $T \models \phi$ si y sólo si $T \not\models \phi'$ (lee: T no satisface a ϕ').

(III) Si $\phi = \phi_1 \wedge \phi_2$, aplica $T \models \phi$ si y sólo si $T \models \phi_1$ y $T \models \phi_2$.

(IV) Si $\phi = \phi_1 \vee \phi_2$, aplica $T \models \phi$ si y sólo si $T \models \phi_1$ o $T \models \phi_2$.

Una expresión booleana ϕ es *satisfactible* si existe una asignación de verdad T que es adecuada para ϕ y además $T \models \phi$.

Una expresión booleana ϕ es válida si para toda T imaginable aplica que $T \models \phi$. En este caso, la expresión es una *tautología* y se lo denota por $\models \phi$. En general aplica que $\models \phi$ si y sólo si $\neg\phi$ es *no satisfactible*.

Dos expresiones ϕ_1 and ϕ_2 son *lógicamente equivalentes* si para toda asignación T que es adecuada para las dos expresiones aplica que

$$T \models \phi_1 \text{ si y sólo si } T \models \phi_2. \tag{1.50}$$

La equivalencia lógica se denota por $\phi_1 \equiv \phi_2$.

Por ejemplo, si tenemos la asignación $T(x_1) = \top$ y $T(x_2) = \bot$, es válido que $T \models x_1 \vee x_2$, pero $T \not\models (x_1 \vee \neg x_2) \wedge (\neg x_1 \wedge x_2)$. Unos ejemplos de expresiones lógicamente equivalentes son los siguientes:

$$\begin{aligned}
\neg\neg\phi &\equiv \phi \\
(\phi_1 \vee \phi_2) &\equiv (\phi_2 \vee \phi_1) \\
((\phi_1 \wedge \phi_2) \wedge \phi_3) &\equiv (\phi_1 \wedge (\phi_2 \wedge \phi_3)) \\
((\phi_1 \wedge \phi_2) \vee \phi_3) &\equiv ((\phi_1 \vee \phi_3) \wedge (\phi_2 \vee \phi_3)) \\
\neg(\phi_1 \wedge \phi_2) &\equiv (\neg\phi_1 \vee \neg\phi_2) \\
(\phi_1 \vee \phi_1) &\equiv \phi_1.
\end{aligned} \tag{1.51}$$

Existen *formas normales* de escribir expresiones booleana. Las dos formas normales son la *forma normal conjuntiva* (CNF) que utiliza puramente el conectivo $\wedge$ y literales y la *forma normal disyunctiva* (DNF) que utiliza puramente el conectivo $\vee$ y literales. Una conjunción de literales se llama un *implicante* y una disyunción de literales se llama una *cláusula*. Se puede asumir que ninguna cláusula ni implicante sea repetido en una forma normal, y tampoco se repiten literales dentro de las cláusulas o los implicantes. Las reglas para transformar una expresión booleana a una de las dos formas normales son los siguientes: primero eliminamos las notaciones auxiliares $\leftrightarrow$ y $\rightarrow$ y movemos las negaciones a formar literales:

(I) **Eliminar la notación auxiliar de** $\leftrightarrow$: La expresión $\phi_1 \leftrightarrow \phi_2$ es lógicamente equivalente a la expresión $(\neg\phi_1 \vee \phi_2) \wedge (\neg\phi_1 \vee \phi_2)$ por lo cual puede ser reemplazado con la expresión más larga que ya no contiene el símbolo auxiliar $\leftrightarrow$ no permitido en las formas normales.

(II) **Eliminar la notación auxiliar de** $\rightarrow$: La expresión $\phi_1 \rightarrow \phi_2$ es lógicamente equivalente a la expresión $\neg\phi_1 \vee \phi_2$ por lo cual puede ser reemplazado con la segunda expresión que ya no contiene el símbolo auxiliar $\rightarrow$ no permitido en las formas normales.

(III) **Mover negaciones donde las variables para formar literales**: utiliza las siguientes equivalencias lógicas para cambiar la ubicación de los símbolos de negación $\neg$ tal que no habrá redundancia y que cada negación forma un literal y ya no aplica a una expresión compleja:

$$\begin{aligned} \neg\neg\phi \quad &\text{es equivalente a} \quad \phi \\ \neg(\phi_1 \vee \phi_2) \quad &\text{es equivalente a} \quad \neg\phi_1 \wedge \neg\phi_2 \\ \neg(\phi_1 \wedge \phi_2) \quad &\text{es equivalente a} \quad \neg\phi_1 \vee \neg\phi_2. \end{aligned} \tag{1.52}$$

Después de aplicar esas reglas tantas veces como posible, queda una expresión de puros literales con los conectivos $\wedge$ y $\vee$. Para lograr una forma normal conjunctiva, hay que mover los conectivos $\wedge$ afuera de los disyunciones aplicando las equivalencias lógicas siguientes:

$$\begin{aligned} \phi_1 \vee (\phi_2 \wedge \phi_3) \quad &\text{es equivalente a} \quad (\phi_1 \vee \phi_2) \wedge (\phi_1 \vee \phi_3) \\ (\phi_1 \wedge \phi_2) \vee \phi_3 \quad &\text{es equivalente a} \quad (\phi_1 \vee \phi_3) \wedge (\phi_2 \vee \phi_3). \end{aligned} \tag{1.53}$$

Si la meta es lograr la forma normal disyuntiva, hay que aplicar las equivalencias siguien-

tes para mover los conectivos $\vee$ afuera de los conjunciones:

$$\begin{aligned} \phi_1 \wedge (\phi_2 \vee \phi_3) \quad &\text{es equivalente a} \quad (\phi_1 \wedge \phi_2) \vee (\phi_1 \wedge \phi_3) \\ (\phi_1 \vee \phi_2) \wedge \phi_3 \quad &\text{es equivalente a} \quad (\phi_1 \wedge \phi_3) \vee (\phi_2 \wedge \phi_3). \end{aligned} \tag{1.54}$$

Es importante tomar en cuenta que las expresiones en forma normal pueden en el peor caso tener un largo exponencial en comparación con el largo de la expresión original. Para dar un ejemplo de la transformación, consideramos el caso de la expresión $(x_1 \vee x_2) \rightarrow (x_2 \leftrightarrow x_3)$ y su transformación a la forma normal conjuntiva utilizando las reglas mencionadas. Cada línea en el proceso es lógicamente equivalente con todas las versiones anteriores por las equivalencias ya definidas; para ahorrar espacio, utilizamos la notación siguiente para las variables: $a = x_1$, $b = x_2$ y $c = x_3$.

$$\begin{gathered} (a \vee b) \rightarrow (b \leftrightarrow c) \\ \neg(a \vee b) \vee (b \leftrightarrow c) \\ \neg(a \vee b) \vee ((\neg b \vee c) \wedge (\neg c \vee b)) \\ (\neg a \wedge \neg b) \vee ((\neg b \vee c) \wedge (\neg c \vee b)) \\ (\neg a \vee ((\neg b \vee c) \wedge (\neg c \vee b))) \wedge (\neg b \vee ((\neg b \vee c) \wedge (\neg c \vee b))) \\ ((\neg a \vee (\neg b \vee c)) \wedge (\neg a \vee (\neg c \vee b))) \wedge \ (\neg b \vee ((\neg b \vee c) \wedge (\neg c \vee b))) \\ ((\neg a \vee (\neg b \vee c)) \wedge (\neg a \vee (\neg c \vee b))) \wedge \ ((\neg b \vee (\neg b \vee c)) \wedge (\neg b \vee (\neg c \vee b))) \end{gathered} \tag{1.55}$$

Una *función booleana* de n-dimensiones f es un mapeo de $\{\top, \bot\}^n$ al conjunto $\{\top, \bot\}$. El conectivo $\neg$ corresponde a una función unaria $f^{\neg} : \{\top, \bot\} \rightarrow \{\top, \bot\}$ definida por la ecuación 1.46. Los conectivos $\vee$, $\wedge$, $\rightarrow$ y $\leftrightarrow$ definen cada uno una función binaria $f : \{\top, \bot\}^2 \rightarrow \{\top, \bot\}$.

Cada expresión booleana se puede interpretar como una función booleana con la dimensión $n = |X(\phi)|$. Se dice que una expresión booleana *expresa* una función f si cada n-eada de valores de verdad $\tau = (t_1, \ldots, t_n)$ aplica que

$$f(\tau) = \begin{cases} \top, & \text{si } T \models \phi, \\ \bot, & \text{si } T \not\models \phi, \end{cases} \tag{1.56}$$

donde T es tal que $T(x_i) = t_i$ para todo $i = 1, \ldots, n$.

Las funciones booleanas se puede representar a través de *grafos* (ver sección 1.8 como *circuitos booleanos*: los vértices son "puertas" y el grafo es dirigido y no cíclico. Un grafo dirigido no cíclico siempre se puede etiquetar tal que cada vértice está representado por

un número entero, $V = \{1, 2, \ldots, n\}$, de tal manera que aplica para cada arista dirigida $\langle i, j\rangle \in E$ que $i < j$. La asignación de tales etiquetas se llama *sorteo topológico*.

Cada puerta es de un cierto tipo: la puerta puede representar una variable x_i, una valor $\top$ o $\bot$ o un conectivo ($\wedge$, $\vee$ o $\neg$). Los vértices que corresponden a variables o los valores de verdad deben tener grado de entrada cero. Las puertas de tipo negación tienen grado de entrada uno y las puertas con $\wedge$ o $\vee$ tienen grado de entrada dos. El último vértice n representa la salida del circuito.

Los valores de verdad de las distintas puertas se determina con un procedimiento inductivo tal que se define el valor para cada puerta todas las entradas de la cual ya están definidos. Cada circuito booleano corresponde a una expresión booleana ϕ. Los circuitos pueden ser representaciones más "compactas" que las expresiones, porque en la construcción del circuito se puede "compartir" la definición de un subcircuito, mientras en las expresiones habrá que repetir las subexpresiones que aparecen en varias partes de la expresión.

1.7.2. Lógica proposicional

La *lógica de primer orden* es una lógica donde se *cuantifica* variables individuales. Cada variable individual corresponde a una oración. Los símbolos de cuantificación son la cuantificador $\exists$ para la cuantificación *existencial* y la cuantificador $\forall$ para la cuantificación *universal*.

La cuantificación existencial quiere decir que en un conjunto indicado existe por lo menos una variable que cumpla con una cierta requisito, mientras la cuantificación universal dice que alguna requisito se aplica para todas las variables del conjunto indicado. Por ejemplo,

$$\forall x \in \mathbb{R}, \exists y \text{ tal que } x = 2y. \tag{1.57}$$

Se puede cuantificar varias variables al mismo tiempo:

$$\forall x, y \in \mathbb{Z}, \exists z, z' \in \mathbb{Z} \text{ tal que } x - y = z \text{ y } x + y = z'. \tag{1.58}$$

1.7.3. Información digital

Bit

El *bit* es la unidad básica de información digital. Un bit es una *variable binaria*: tiene dos valores posibles que se interpreta como los valores lógicos "verdad" (1) y "falso" (0).

En la memoria de una computadora, se expresa los números enteros con sumas de *potencias de dos* (ver el cuadro 1.1 y la figure 1.3), empezando con la potencia más grande que quepa en el número e iterando hasta llegar a la suma correcta.

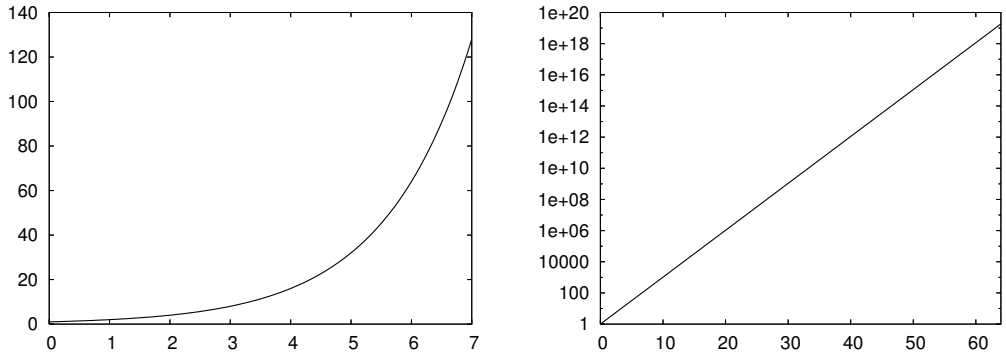

Figura 1.3: Gráficas de las potencias de dos: a la izquierda, con ejes lineales hasta $x = 7$, y a la derecha, con ejes de escala logarítmica hasta $x = 64$.

Se representa la presencia de una cierta potencia con un bit de valor 1 y su ausencia por un bit de valor 0, empezando con la potencia más grande presente:

$$\begin{aligned} 61 &= 32 + 29 = 32 + 16 + 13 = 32 + 16 + 8 + 5 \\ &= 32 + 16 + 8 + 4 + 1 \\ &= 25 + 24 + 23 + 22 + 20 \\ &\Rightarrow 111101. \end{aligned}$$

Entonces se necesita seis bits para representar el valor 61. Por lo general, la cantidad b de bits requeridos para representar un valor $x \in \mathbb{Z}^+$ está el exponente de la mínima potencia de dos mayor a x,

$$b = \min_{k \in \mathbb{Z}} \left\{ k \mid 2^k > x \right\}. \tag{1.59}$$

Para incluir los enteros negativos, se usa un bit auxiliar de signo.

Byte

El *byte* es la unidad básica de capacidad de memoria digital. Un byte (se pronuncia "bait") es una sucesión de ocho bits. El número entero más grande que se puede guardar en un solo byte es

$$2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 2^8 - 1 = 255$$

y entonces, contando cero, son 256 valores posibles por un byte. Nota que aplica en general la igualdad

$$2^{k+1} - 1 = \sum_{i=0}^{k} 2^i. \tag{1.60}$$

Cuadro 1.1: Algunas potencias del número dos, $2^k = x$.

2^{1}	=	2	2^{20}	=	1 048 576	2^{40}	=	1 099 511 627 776
2^{2}	=	4	2^{21}	=	2 097 152	2^{41}	=	2 199 023 255 552
2^{3}	=	8	2^{22}	=	4 194 304	2^{42}	=	4 398 046 511 104
2^{4}	=	16	2^{23}	=	8 388 608	2^{43}	=	8 796 093 022 208
2^{5}	=	32	2^{24}	=	16 777 216	2^{44}	=	17 592 186 044 416
2^{6}	=	64	2^{25}	=	33 554 432	2^{45}	=	35 184 372 088 832
2^{7}	=	128	2^{26}	=	67 108 864	2^{46}	=	70 368 744 177 664
2^{8}	=	256	2^{27}	=	134 217 728	2^{47}	=	140 737 488 355 328
2^{9}	=	512	2^{28}	=	268 435 456	2^{48}	=	281 474 976 710 656
2^{10}	=	1 024	2^{29}	=	536 870 912	2^{49}	=	562 949 953 421 312
2^{11}	=	2 048	2^{30}	=	1 073 741 824	2^{50}	=	1 125 899 906 842 624
2^{12}	=	4 096	2^{31}	=	2 147 483 648	2^{51}	=	2 251 799 813 685 248
2^{13}	=	8 192	2^{32}	=	4 294 967 296	2^{52}	=	4 503 599 627 370 496
2^{14}	=	16 384	2^{33}	=	8 589 934 592	2^{53}	=	9 007 199 254 740 992
2^{15}	=	32 768	2^{34}	=	17 179 869 184	2^{54}	=	18 014 398 509 481 984
2^{16}	=	65 536	2^{35}	=	34 359 738 368	2^{55}	=	36 028 797 018 963 968
2^{17}	=	131 072	2^{36}	=	68 719 476 736	2^{56}	=	72 057 594 037 927 936
2^{18}	=	262 144	2^{37}	=	137 438 953 472	2^{57}	=	144 115 188 075 855 872
2^{19}	=	524 288	2^{38}	=	274 877 906 944	2^{58}	=	288 230 376 151 711 744
			2^{39}	=	549 755 813 888	2^{59}	=	576 460 752 303 423 488

Un *kilobyte* es 1024 bytes, un *megabyte* es 1024 kilobytes (1048576 bytes) y un *gigabyte* es 1024 megabytes (1073741824 bytes). Normalmente el prefix kilo implica un mil, pero como mil no es ningún potencia de dos, eligieron la potencia más cercana, $2^{10} = 1024$, para corresponder a los prefixes.

Representación de punto flotante

Para representar números reales por computadora, hay que definir hasta que exactitud se guarda los decimales del número, como el espacio para guardar un número entero está limitada a un tamaño constante. El método común de lograr tal representación es lo de *punto flotante* (también conocido como *coma flotante*) donde la representación se adapta al orden magnitud del valor $x \in \mathbb{R}$. La idea es trasladar la coma decimal hacia la posición de la primera cifra significativa de x mediante un exponente γ:

$$x = m \cdot b^{\gamma}, \tag{1.61}$$

donde m se llama la *mantisa* y contiene los dígitos significativos de x – es común normalizar la mantisa tal que su parte entera consta de solamente la primera cifra significativa de x. La mantisa típicamente tiene un tamao máximo limitado a una cierta cantidad fija de bytes.

El parámetro b en ecuación 1.61 es la *base* del sistema de representación. Los números binarios tienen base $b = 2$ y comúnmente en cálculo utilizamos $b = 10$. También existen sistemas en base $b = 8$ (el sistema *octal*) y $b = 16$ (el sistema *hexadecimal*).

Lo que determina el rango de valores posibles que se puede representar en punto flotante es la cantidad de memoria reservada para el exponente $\gamma \in \mathbb{Z}$ de la ecuación 1.61.

La otra opción sería simplemente reservar una cierta cantidad de bytes para la representación y *fijar* a posición en la cual se supone que esté la coma decimal. La representación de *punto fija* es mucho más restrictiva con respeto al rango de valores posibles de guardar. En comparación, el método de punto flotante causa variaciones en la exactitud de la representación, mientras permite guardar valores de un rango mucho más amplio.

1.7.4. Redondeo

Funciones techo y piso

La función *piso* $\lfloor x \rfloor : \mathbb{R} \to \mathbb{Z}$ define el número entero *máximo menor* a $x \in \mathbb{R}$:

$$\lfloor x \rfloor = \max_{y \in \mathbb{Z}} \{y \mid x \geq y\}. \tag{1.62}$$

Por definición, siempre aplica para todo $x \in \mathbb{R}$ que

$$\lfloor x \rfloor \leq x < \lfloor x + 1 \rfloor \tag{1.63}$$

donde la igualdad ocurre solamente cuando $x \in \mathbb{Z}$. Igualmente, tenemos

$$x - 1 < \lfloor x \rfloor \leq x \tag{1.64}$$

También, para todo $k \in \mathbb{Z}$ y $x \in \mathbb{R}$ aplica

$$\lfloor k + x \rfloor = k + \lfloor x \rfloor. \tag{1.65}$$

Una sucesión de combinar las propiedades del logaritmo con la función piso es que siempre aplica también para $x > 0$, $x \in \mathbb{R}$ que

$$\frac{x}{2} < 2^{\lfloor \log_2 x \rfloor} \leq x, \tag{1.66}$$

la verificación de que se deja como ejercicio.

La función *techo* $\lceil x \rceil : \mathbb{R} \to \mathbb{Z}$ define el número entero *mínimo mayor* a $x \in \mathbb{R}$:

$$\lceil x \rceil = \min_{y \in \mathbb{Z}} \{ y \mid x \leq y \}. \tag{1.67}$$

Aplica por definición que

$$x \leq \lceil x \rceil < x + 1. \tag{1.68}$$

Se deja como ejercicio verificar que también para $x \in \mathbb{R}$, $x > 0$ aplica

$$x \leq 2^{\lceil \log_2 x \rceil} < 2x. \tag{1.69}$$

Función parte entera

Varios lenguajes de programación incorporan una manera de convertir un número en formato punto flotante a un valor entero. Por ejemplo, en C se permite la construcción siguiente

```
float a = 3.76;
int b = (int)a;
```

donde la regla de asignar un valor a la variable `b` es una mezcla de la funciones piso y techo: denota el valor de la variable `a` por a. Si $a \geq 0$, se asigna a `b` el valor $\lfloor a \rfloor$, y cuando $a < 0$, se asigna a `b` el valor $\lceil a \rceil$.

La función que captura esta comportamiento es la función *parte entera*

$$[x] : \mathbb{R} \to \mathbb{Z}. \tag{1.70}$$

El *redondeo* típico de $x \in \mathbb{R}$ al entero más próximo es equivalente a $[x + 0{,}5]$.

Hay que tener mucho cuidado con la operación de parte entera en programación, como implica pérdida de datos. Acumulando uno después de otro, los errores de redondeo se pueden amplificar y causar comportamiento no deseado en el programa. Por ejemplo, en algunas implementaciones una sucesión instrucciones como

```
float a = 0.6/0.2;
int b = (int)a;
```

puede resultar en b asignada al valor 2, porque por la representación binaria de punto flotante de los valores $0{,}6$ y $0{,}2$, su división resulta en el valor punto flotante $2{,}99999999999999955591079014993 7$. Entonces, aplicar la función parte entera al valor de la variable a resulta en el valor 2, aunque la respuesta correcta es 3.

1.8. Teoría de grafos

Un *grafo* (también se dice *gráfica*) G es un par de conjuntos $G = (V, E)$, donde V es un conjunto de n *vértices* (o sea, nodos), $u, v, w \in V$ y E es un conjunto de *aristas* m (o sea, arcos). Utilizamos la notación $|V| = n$, $|E| = m$

Las aristas son típicamente pares de vértices, $\{u, v\} \in E$, o sea, las aristas definen una relación entre el conjunto V con si mismo:

$$E \subseteq V \times V, \tag{1.71}$$

pero también se puede definir grafos donde el producto es entre más de dos "copias" del conjunto V, el cual caso se habla de *hípergrafos*.

El *complemento* de un grafo $G = (V, E)$ es un grafo $\bar{G} = (V, \bar{E})$ donde

$$\forall v \neq u : \big(\{v, u\} \in \bar{E} \Leftrightarrow \{v, u\} \notin E\big). \tag{1.72}$$

1.8.1. Clases de grafos

Un grafo es *plano* si se puede dibujar en dos dimensiones tal que ninguna arista cruza a otra arista. En un grafo *no dirigido*, los vértices v y w tienen un papel igual en la arista $\{v, u\}$. Si las aristas tienen *dirección*, el grafo G es *dirigido* (también *digrafo*) y el vértice v es el *origen* (o inicio) de la arista dirigida $\langle v, w\rangle$ y el vértice w es el destino (o fin) de la arista. Un *bucle* es una arista *reflexiva*, donde coinciden el vértice de origen y el vértice de destino: $\{v, v\}$ o $\langle v, v\rangle$. Si un grafo G no cuente con ninguna arista reflexiva, el grafo es *no reflexivo*.

En el caso general, E puede ser un *multiconjunto*, es decir, es posible que haya más de una arista entre un par de vértices. En tal caso, el grafo se llama *multigrafo*. Si no se permiten aristas múltiples, el grafo es *simple*.

Si se asignan *pesos* $\omega\,(v)\,w$ (o costos o longitudes) a las aristas, el grafo es *ponderado*. Si se asigna *identidad* a los vértices o las aristas, es decir que sean distinguibles, el grafo es *etiquetado*.

1.8.2. Adyacencia

Dos **aristas** $\{v_1, v_2\}$ y $\{w_1, w_2\}$ de un grafo son *adyacentes* si tienen un vértice en común:

$$|\{v_1, v_w\} \cap \{w_1, w_2\}| \geq 1 \tag{1.73}$$

(el valor puede ser mayor a uno solamente en un multigrafo). Una arista es *incidente* a un vértice si ésta lo une a otro vértice.

Dos **vértices** v y w son adyacentes si una arista los une:

$$\{v, w\} \in E. \tag{1.74}$$

Vértices adyacentes son llamados *vecinos* y el conjunto de vecinos del vértice v se llama su *vecindario* y se denota con $\Gamma(v)$.

La matriz $\mathbf{A}$ que corresponde a la relación E se llama la *matriz de adyacencia* del grafo; para construir la matriz, es necesario etiquetar los vértices para que sean identificados como $v_1, v_2, \ldots, v_n$. La matriz de adyacencia de un grafo no dirigido es simétrica.

Para representar la adyacencia de multigrafos, es mejor abandonar la matriz binaria y construir otra matriz entera $\mathbf{A}'$ donde el elemento $a'_{ij} \geq 0$ contiene el número de aristas entre v_i y v_j. Para grafos ponderados, es mejor usar una matriz (real) $\mathbf{A}''$ donde el elemento a''_{ij} contiene el peso de la arista $\{v_i, v_j\}$ o cero si no hay tal arista en el grafo.

El *grado* $\deg(v)$ de un vértice v es el número de aristas incidentes a v. Para grafos dirigidos, se define el *grado de salida* $\overrightarrow{\deg}(v)$ de un vértice v como el número de aristas que tienen su origen en v y el *grado de entrada* $\overleftarrow{\deg}(v)$ de v como el número de aristas que tienen su destino en v. El grado total de un vértice de un grafo dirigido es

$$\deg(v) = \overleftarrow{\deg}(v) + \overrightarrow{\deg}(v). \tag{1.75}$$

En un grafo simple no dirigido, el grado $\deg(v_i)$ del vértice v_i es la suma de la $i$jésima fila de $\mathbf{A}$.

Nota que siempre aplica

$$\sum_{v \in V} \deg(v) = 2m. \tag{1.76}$$

La sumación sobre $v \in V$ quiere decir que cada vértice v del conjunto V se toma en cuenta una vez en el cálculo.

En un grafo simple no reflexivo, aplica

$$\deg(v) = |\Gamma(v)|. \tag{1.77}$$

Si *todos* los vértices tienen el mismo grado k, el grafo es *regular*, o mejor dicho, k-regular.

En un grafo $n-1$-regular, cada vértice está conectado a cada otro vértice por una arista. Se llama un grafo *completo* y se denota por K_n.

1.8.3. Grafos bipartitos

Un grafo *bipartito* es un grafo $G = (V, E)$ cuyos vértices se pueden separar en dos conjuntos disjuntos U y W,

$$U \cap W = \emptyset,\, U \cup W = V \tag{1.78}$$

tal que las aristas solamente unen vértices de un conjunto con algunos vértices del otro:

$$\{u, w\} \in E \Rightarrow (u \in U \wedge w \in W) \vee (u \in W \wedge w \in U). \tag{1.79}$$

En un grafo *bipartito completo* están presentes *todas* las aristas entre U y W. Se denota tal grafo por $K_{a,b}$ donde $a = |U|$ y $b = |W|$. Nota que para $K_{a,b}$, siempre aplica que $m = a \cdot b$.

1.8.4. Densidad

El número máximo posible de aristas en un grafo simple es

$$m_{\text{máx}} = \binom{n}{2} = \frac{n(n-1)}{2}. \tag{1.80}$$

Para K_n, tenemos $m = m_{\text{máx}}$. La *densidad* $\delta(G)$ de un G se defina como

$$\delta(G) = \frac{m}{m_{\text{máx}}} = \frac{m}{\binom{n}{2}}. \tag{1.81}$$

Un grafo *denso* tiene $\delta(G) \approx 1$ y un grafo *escaso* tiene $\delta(G) \ll 1$.

1.8.5. Caminos

Una sucesión de aristas adyacentes que empieza en v y termina en w se llama un *camino* de v a w. El *largo* de un camino es el número de aristas que contiene el camino. La *distancia* $\operatorname{dist}(v, w)$ entre v y w es el largo mínimo de todos los caminos de v a w. La distancia de un vértice a si mismo es cero. El *diámetro* $\operatorname{diam}(G)$ de un grafo G es la distancia máxima en todo el grafo,

$$\operatorname{diam}(G) = \max_{\substack{v \in V \\ w \in V}} \operatorname{dist}(v, w). \tag{1.82}$$

Un camino *simple* solamente recorre la misma arista una vez máximo, nunca dos veces o más. Un *ciclo* es un camino que regresa a su vértice inicial. Un grafo que no cuente con ningún ciclo es *acíclico*. Un grafo no dirigido acíclico es necesariamente un árbol, pero en grafos dirigidos la ausencia de ciclos no implica que sea un árbol. Un grafo es bipartito si y sólo si no tiene ningún ciclo de largo impar.

1.8.6. Conectividad

Un grafo G es *conexo* si *cada* par de vértices está conectado por un camino. Si por algunos vértices v y w no existe ningún camino de v a w en el grafo G, el grafo G es *no conexo*, la distancia entre los dos vértices no está definido, y en consecuencia el diámetro $\operatorname{diam}(G)$ del grafo tampoco es definido. Un grafo G es *fuertemente conexo* si cada par de vértices está conectado por *al menos dos* caminos disjuntos, es decir, dos caminos que no comparten ninguna arista.

Un grafo no conexo se puede dividir en dos o más *componentes conexos* que son formados por tales conjuntos de vértices de distancia definida.

1.8.7. Subgrafos

Un grafo $G(S) = (S, F)$ es un *subgrafo* del grafo $G = (V, E)$ si $S \subseteq V$ y $F \subseteq E$ tal que

$$\{v, w\} \in F \Rightarrow \big((v \in S) \wedge (w \in S)\big). \tag{1.83}$$

Cada componente conexo es un subgrafo conexo maximal, o sea, a cual no se puede aadir ningún otro vértice sin romper conectividad.

Un subgrafo que completo se dice una *camarilla* (inglés: clique).

Un grafo G_1 es *isomorfo* a otro grafo G_2 si y sólo si existe una función $f : V_1 \to V_2$ de los vértices V_1 de G_1 a los vértices V_2 de G_2 tal que para en conjunto de aristas de G_1, denotado por E_1, y lo de las aristas de G_2, denotado por E_2 aplica que

$$\{v_1, u_1\} \in E_1 \Leftrightarrow \{f(v_1), f(u_1)\} \in E_2. \tag{1.84}$$

1.8.8. Árboles

Un *árbol* es un grafo conexo acíclico. Un *árbol cubriente* (también: un árbol de *expansión*) de un grafo $G = (V, E)$ es un subgrafo de $grafo$ que es un árbol y contiene todos los vértices de G. Si el grafo es ponderado, el árbol cubriente *mínimo* es cualquier árbol donde la suma de los pesos de las aristas incluidas es mínima. Un grafo G no conexo es un *bosque* si cada componente conexo de G es un árbol.

Capítulo 2

Problemas y algoritmos

La meta de este documento que el lector aprendra a analizar de complejidad de dos conceptos diferentes de computación: *problemas* y *algoritmos*. Un problema es un conjunto (posiblemente infinito) de *instancias* junto con una pregunta sobre alguna propiedad de las instancias. Un algoritmo es un proceso formal para encontrar la respuesta correcta a la pregunta de un problema para una cierta instancia del problema.

2.1. Problema

Problemas en general son conjuntos de instancias al cual corresponde un conjunto de soluciones, junto con una relación que asocia para cada instancia del problema un subconjunto de soluciones (posiblemente vaco).

Los dos tipos de problemas son los problemas de *decisión* y los problemas de *optimización* . En los primeros, la respuesta es siempre "sí" o "no", mientras en la segunda clase de problemas la pregunta es del tipo "cuál es el mejor valor posible" o "con qué configuración se obtiene el mejor valor posible".

2.1.1. Problema de decisión

En un problema de decisión, la tarea es decidir si o no la relación entre instancias y soluciones asigna un subconjunto vacío a una dada instancia. Si existen soluciones, la respuesta a la pregunta del problema es "sí", y si el subconjunto es vacío, la respuesta es "no".

2.1.2. Problema de optimización

Para problemas de optimización, la instancia está compuesta por un conjunto de configuraciones, un conjunto de restricciones, y además una *función objetivo* que asigna un valor (real) a cada instancia. Si las configuraciones son discretas, el problema es *combinatorial*.

La tarea es identificar cuál de las configuraciones *factibles*, es decir, las que cumplen con todas las restricciones, tiene el mejor valor de la función objetivo. Depende del problema si el mejor valor es el mayor (problema de *maximización*) o el menor (problema de *minimización*). La configuración factible con el mejor valor se llama la *solución óptima* de la instancia.

2.2. Algoritmo

Un *algoritmo* es un método de solución para resolver una dada instancia de un cierto problema. En computación, por lo general, se escribe el algoritmo en un lenguaje de programación para ser ejecutado por una computadora. Ejemplos de algoritmos son los métodos sistemáticos de resolver los problemas siguientes:

- Cómo encontrar un nombre en la guía telefónica?
- Cómo llegar de mi casa a mi oficina?
- Cómo determinar si un dado número es un número primo?

Para definir un algoritmo, hay que definir primero dos conjuntos:

(I) un conjunto $\mathcal{E}$ de las *entradas* del algoritmo, que representan las instancias del problema y

(II) un conjunto $\mathcal{S}$ de las *salidas*, que son los posibles resultados de la ejecución del algoritmo.

Para un problema, por lo general existen varios algoritmos con diferente nivel de eficiencia (es decir, diferente tiempo de ejecución con la misma instancia del problema). En algoritmos *deterministas*, la salida del algoritmo depende únicamente de la entrada de lo mismo, por lo cual se puede representar el algoritmo como una función $f : \mathcal{E} \to \mathcal{S}$. Existen también algoritmos *probabilistas* o *aleatorizados* donde esto **no** es el caso.

Los algoritmos se escribe como sucesiones de *instrucciones* que procesan la entrada $\rho \in \mathcal{E}$ para producir el resultado $\xi \in \mathcal{S}$. Cada instrucción es una operación simple, produce un resultado intermedio único y es posible ejecutar con eficiencia.

La sucesión S de instrucciones tiene que ser finita y tal que para toda $\rho \in \mathcal{E}$, si P está ejecutada con la entrada ρ, el resultado de la computación será $f(\rho) \in \mathcal{S}$. Sería altamente deseable que para todo $\rho \in \mathcal{E}$, la ejecución de S terminará después de un tiempo finito.

Los algoritmos se implementa como programas de cómputo en diferentes lenguajes de programación. El mismo algoritmo se puede implementar en diferentes lenguajes y para diferentes plataformas computacionales. En este curso, los ejemplos siguen las estructuras básicas de programación procedural, en pseudocódigo parecido a C y Java.

2.2.1. Algoritmo recursivo

Un algoritmo *recursivo* es un algoritmo donde una parte del algoritmo utiliza a si misma como subrutina. En muchos casos es más fácil entender la función de un algoritmo recursivo y también demostrar que funcione correctamente. Un algoritmo que en vez de llamarse a si mismo repite en una manera cíclica el mismo código se dice *iterativo*. En muchos casos, el pseudocódigo de un algoritmo recursivo resulta más corto que el pseudocódigo de un algoritmo parecido pero iterativo para el mismo problema.

Es un hecho universal que cada algoritmo recursivo puede ser convertido a un algoritmo iterativo (aunque no viceversa), aunque típicamente hace daño a la eficiencia del algoritmo hacer tal conversión. Depende del problema cuál manera es más eficiente: recursiva o iterativa.

Como ejemplo, veremos dos algoritmos para verificar si o no una dada palabra (una sola palabra de puras letras; en español, típicamente ignorando los acutes de las letras) es un *palíndromo*, o sea, que se lee igual hacia adelante que hacia atrás. Un ejemplo de tal palabra es "reconocer". Un algoritmo recursivo simplemente examinaría la primera y la última letra. Si son iguales, el algoritmo los quita de la sucesión de letras y llama a si mismo para ver si el resto también lo es. Al recibir una sucesión de un símbolo o vacío, el algoritmo da la respuesta "sí":

procedimiento rpal(palabra $P = \ell_1\ell_2 \ldots \ell_{n-1}\ell_n$)
  si $n \leq 1$
    devuelve verdadero ;
  si $\ell_1 = \ell_n$
    devuelve rpal($\ell_2\ell_3 \ldots \ell_{n-2}\ell_{n-1}$);
  en otro caso
    devuelve falso

Una versión iterativa examinaría al mismo tiempo desde el inicio y desde el fin verificando para cada letra si son iguales:

procedimiento ipal(palabra $P = \ell_1\ell_2 \dots \ell_{n-1}\ell_n$)
  $i = 1$;
  $j = n$;
  mientras $i < j$
    si $\ell_i \neq \ell_j$
      devuelve falso ;
    en otro caso
      $i := i + 1$;
      $j := j - 1$;
  devuelve verdadero ;

2.3. Calidad de algoritmos

Las dos medidas más importantes de la calidad de un algoritmo son

(I) el tiempo total de computación, medido por el número de operaciones de cómputo realizadas durante la ejecución del algoritmo, y

(II) la cantidad de memoria utilizada.

La notación para capturar tal información es a través de *funciones de complejidad*.

Para eliminar el efecto de una cierta computadora o un cierto lenguaje de programación, se considera que el algoritmo se ejecuta en una máquina "modelo" virtual tipo RAM (inglés: random access machine) que no tiene límite de memoria ni límite de precisión de representación de números enteros o reales.

2.3.1. Operación básica

Para poder contar las operaciones que ejecuta un algoritmo, hay que definir cuáles operaciones se cualifican como operaciones básicas. Típicamente se considera básicas los siguientes tipos de operaciones:

(I) operaciones simples aritméticas ($+, -, \times, /,$ $\mod$),

(II) operaciones simples lógicas ($\wedge, \vee, \neg, \rightarrow, \leftrightarrow$),

(III) comparaciones simples ($<, >, =, \neq, \leq, \geq$),

(IV) asignaciones de variables (:=),

(V) instrucciones de salto (`break`, `continue`, etcétera).

2.3.2. Tamaño de la instancia

Para un cierto problema computacional, existen típicamente varias si no una cantidad infinita de instancias. Para definir el *tamaño* de una dicha instancia, hay que fijar cuál será la unidad básica de tal cálculo. Típicamente se utiliza la cantidad de bits, bytes, variables enteras, etcétera que se necesita ocupar para representar el problema en su totalidad en la memoria de una computadora.

Para un algoritmo de ordenar una lista de números, el tamaño de la instancia es la cantidad de números que tiene como entrada. Por ejemplo, si la instancia es un grafo, su tamaño es bien capturado en la suma $n + m$, como ninguno de los dos números sólo puede capturar el tamaño de la instancia completamente. En realidad, para guardar cada arista, se necesita guardar su punto de inicio y su punto final, sumando en $2m$, pero por lo general se suele ignorar multiplicadores constantes en tal análisis, como veremos en sección 2.3.5.

2.3.3. Funciones de complejidad

Incluso se fijamos el tamaño de la instancia, todavía hay variaciones en la cantidad de tiempo requerido para la ejecución del algoritmo. Por ejemplo, es más difícil ordenar la lista $[3, 5, 2, 9, 1]$ que la lista $[1, 3, 5, 6, 7]$, como la segunda ya está ordenada. Lo que queremos nosotros es tal caracterización de la calidad de un algoritmo que nos facilita hacer comparaciones entre algoritmos. Las soluciones incluyen el uso del *caso peor*, *caso promedio* y el *caso amortizado*.

Función del peor caso

Formamos una función de complejidad $f : \mathbb{Z}^+ \to \mathbb{Z}^+$ tal que para un valor n, el valor $f(n)$ representa el número de operaciones básicas para el *más* difícil de todas las instancias de tamaño n. La única dificultad es identificar o construir la instancia que es el peor posible para la ejecución del algoritmo.

Función del caso promedio

Formamos una función de complejidad $f : \mathbb{Z}^+ \to \mathbb{R}$ tal que para un valor n, el valor $f(n)$ representa el número *promedio* de operaciones básicas sobre todas las instancias

de tamaño n. Aquí la parte con posible dificultad es la estimación de la distribución de probabilidad: con qué probabilidad ocurren diferentes tipos de instancias? En práctica, si el peor caso es muy raro, resulta más útil estudiar el caso promedio, si es posible.

Complejidad amortizada

En algunos casos, es posible que el tiempo de ejecución de un algoritmo depende de las ejecuciones anteriores. Este ocurre cuando uno procesa una serie de instancias con algún tipo de dependencia entre ellas. En tal caso, las funciones de peor caso y caso promedio pueden resultar pesimistas. La *complejidad amortizada* sirve para evaluar la eficiencia de un algoritmo en tal caso. La idea es ejecutar el algoritmo varias veces en secuencia con diferentes instancias, ordenando las instancias de la peor manera posible (para consumir más recursos), calculando el tiempo total de ejecución, y dividiendo por el número de instancias. En muchos casos, la computación de la complejidad amortizada resulta razonable y no demasiado compleja.

2.3.4. Consumo de memoria

Igual como el tiempo de ejecución, el *consumo de memoria* es una medida importante en la evaluación de calidad de algoritmos. Hay que definir cuál será la unidad básica de memoria para hacer el análisis: puede ser un bit, un byte o una variable de tamaño constante de un cierto tipo. El caso peor de consumo de memoria es la cantidad de unidades de memoria que el algoritmo tendrá que ocupar simultaneamente en el peor caso imaginable. Se define el caso promedio y el caso amortizado igual como con el tiempo de ejecución.

2.3.5. Análisis asintótico

La meta del análisis de algoritmos es evaluar la calidad de un algoritmo en comparación con otros algoritmos o en comparación a la complejidad del problema o alguna cota de complejidad conocida. Sin embargo, típicamente el conteo de "pasos de computación" falta precisión en el sentido que no es claro que cosas se considera operaciones básicas. Por eso normalmente se caracteriza la calidad de un algoritmo por la *clase de magnitud* de la función de complejidad y no la función exacta misma. Tampoco son interesantes los tiempos de computación para instancias pequeñas, sino que instancias grandes — con una instancia pequeña, normalmente todos los algoritmos producen resultados rápidamente.

Para definir clases de magnitudes, necesitamos las definiciones siguientes. Para funciones $f : \mathbb{Z}^+ \to \mathbb{R}$ y $g : \mathbb{Z}^+ \to \mathbb{R}$, escribimos

(I) $f(n) \in \mathcal{O}(g(n))$ si $\exists c > 0$ tal que $|f(n)| \leq c\,|g(n)|$ para suficientemente grandes valores de n,

(II) $f(n) \in \Omega(g(n))$ si $\exists c > 0$ tal que $|f(n)| \geq c\,|g(n)|$ para suficientemente grandes valores de n,

(III) $f(n) \in \Theta(g(n))$ si $\exists c, c' > 0$ tales que $c \cdot |g(n)| \leq |f(n)| \leq c' \cdot |g(n)|$ para suficientemente grandes valores de n y

(IV) $f(n) \in o(g(n))$ si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

El símbolo $\in$ se reemplaza frecuentemente con $=$. La función $\mathcal{O}(f(n))$ es una *cota superior asintótica* al tiempo de ejecución, mientras la $\Omega(f(n))$ es una *cota inferior asintótica*. La tercera definición quiere decir que las dos funciones crecen asintóticamente iguales. Las definiciones se generalizan para funciones de argumentos múltiples.

Estas definiciones son *transitivas*:

$$(f(n) \in \mathcal{O}(g(n)) \wedge g(n) \in \mathcal{O}(h(n))) \Rightarrow f(n) \in \mathcal{O}(h(n)). \tag{2.1}$$

Igualmente, si tenemos que

$$(f(n) \in \Omega(g(n)) \wedge g(n) \in \Omega(h(n))) \Rightarrow f(n) \in \Omega(h(n)). \tag{2.2}$$

Como este aplica para $\Omega(f(n))$ y $\mathcal{O}(f(n))$ los dos, aplica por definición también para $\Theta(f(n))$.

Otra propiedad útil es que si

$$(f(n) \in \mathcal{O}(h(n)) \wedge g(n) \in \mathcal{O}(h(n))) \Rightarrow f(n) + g(n) \in \mathcal{O}(h(n)). \tag{2.3}$$

Si $g(n) \in \mathcal{O}(f(n))$, $f(n) + g(n) \in \mathcal{O}(f(n))$. Esto nos permite fácilmente formar $\mathcal{O}(f(n))$ de polinomios y muchas otras expresiones. Por definición, podemos ignorar coeficientes numéricos de los términos de la expresión. Además con el resultado anterior nos permite quitar todos los términos salvo que el término con exponente mayor. Por lo general, $\mathcal{O}(f(n))$ es la notación de complejidad asintótica más comúnmente utilizado.

Una observación interesante es la complejidad asintótica de funciones *logarítmicas*: para cualquier base $b > 0$ y *cada* $x > 0$ tal que $x \in \mathbb{R}$ (incluso números muy cercanos a cero), aplica que $\log_b(n) \in \mathcal{O}(n^x)$ (por la definición de logaritmo). Utilizando la definición de ecuación 1.25 para cambiar la base de un logaritmo, llegamos a tener

$$\log_a(n) = \frac{1}{\log_b(a)} \log_b(n) \in \Theta(\log_b n), \tag{2.4}$$

porque $\log_b(a)$ es una constante. Entonces, como $\log_a(n) = \Theta(\log_b n)$, no hay necesidad de marcar la base en una expresión de complejidad asintótica con logaritmos.

Otra relación importante es que para todo $x > 1$ y todo $k > 0$, aplica que $n^k \in \mathcal{O}(x^n)$ — es decir, cada polinomial crece asintóticamente más lentamente que cualquiera expresión exponencial.

En términos no muy exactos, se dice que un algoritmo es *eficiente* si su tiempo de ejecución tiene una cota *superior* asintótica que es un *polinomio*. Un problema que cuenta con por lo menos un algoritmo eficiente es un problema *polinomial*. Un problema es *intratable* si no existe ningún algoritmo eficiente para resolverlo. También se dice que un problema *sin solución* si no cuenta con algoritmo ninguno. El el siguiente capítulo formulamos estes tipos de conceptos formalmente.

Para ilustrar el efecto del tiempo de ejecución, el cuadro 2.1 (adaptado de [10]) muestra la dependencia del tiempo de ejecución del número de operaciones que se necesita.

Cuadro 2.1: En el cuadro (originalmente de [10]) se muestra para diferentes valores de n el tiempo de ejecución (en segundos (s), minutos (min), horas (h), días (d) o años (a)) para un algoritmo necesita exactamente $f(n)$ operaciones básicas del procesador para encontrar solución y el procesador es capaz de ejecutar un millón de instrucciones por segundo. Si el tiempo de ejecución es mayor a 10^{25} años, lo marcamos simplemente como $\approx \infty$, mientras los menores a un segundo son ≈ 0. El redondeo con tiempos mayores a un segundo están redondeados a un segundo entero mayor si menos de un minuto, al minuto entero mayor si menores a una hora, la hora entera mayor si menores a un día, y el año entero mayor si medidos en años.

$f(n)$ $(\rightarrow)$ n $(\downarrow)$	n	$n \log_2 n$	n^2	n^3	$1,5^n$	2^n	$n!$
10	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	4 s
30	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	18 min	10^{25} a
50	≈ 0	≈ 0	≈ 0	≈ 0	11 min	36 a	$\approx \infty$
100	≈ 0	≈ 0	≈ 0	1 s	12,892 a	10^{17} años	$\approx \infty$
1000	≈ 0	≈ 0	1 s	18 min	$\approx \infty$	$\approx \infty$	$\approx \infty$
10000	≈ 0	≈ 0	2 min	12 d	$\approx \infty$	$\approx \infty$	$\approx \infty$
100000	≈ 0	2 s	3 h	32 a	$\approx \infty$	$\approx \infty$	$\approx \infty$
1000000	1 s	20 s	12 d	31,710 a	$\approx \infty$	$\approx \infty$	$\approx \infty$

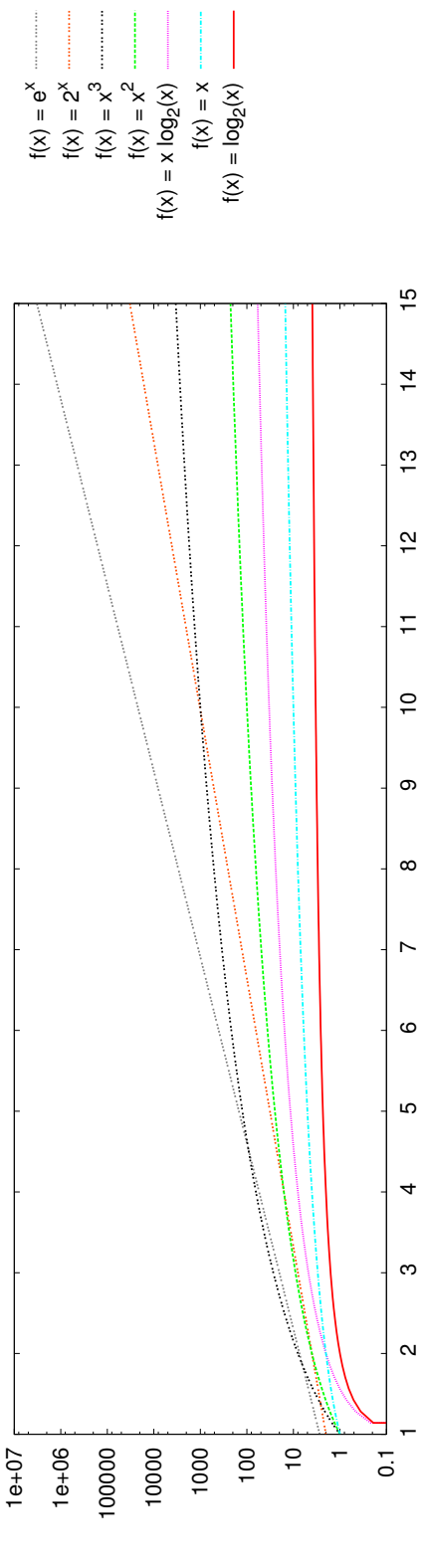

Figura 2.1: Crecimiento de algunas funciones comúnmente encontrados en el análisis de algoritmos. Nota que el ordenación según magnitud cambia al comienzo, pero para valores suficientemente grandes ya no hay cambios.

Capítulo 3

Modelos de la computación

3.1. Máquinas Turing

Un modelo formal de la computación fundamental es la *máquina Turing*. Las máquinas Turing pueden simular cualquier algoritmo con prdida de eficiencia insignificante utilizando una sola estructura de datos: una sucesión de símbolos escrita en una cinta (infinita) que permite borrar e imprimir símbolos. Formalmente, se define una máquina Turing $M = (K, \Sigma, \delta, s)$ por

(I) un conjunto finito de *estados* K,

(II) un conjunto finito de *símbolos* Σ, que se llama el *alfabeto* de M que contiene dos símbolos espacieles $\sqcup, \triangleright \in \Sigma$,

(III) una *función de transición*

$$\delta : K \times \Sigma \to (K \cup \{\text{“alto”}, \text{“sí”}, \text{“no”}\}) \times \Sigma \times \{\to, \leftarrow, -\}, \tag{3.1}$$

(IV) un estado de *alto* “alto”, un estado de *aceptación* “sí” y un estado de *rechazo* “no” y

(V) *direcciones* del *puntero*: $\to$ (derecha), $\leftarrow$ (izquierda) y $-$ (sin mover).

La función δ captura el “programa” de la máquina. Si el estado actual es $q \in K$ y el símbolo actualmente bajo el puntero es $\sigma \in \Sigma$, la función δ nos da

$$\delta(q, \sigma) = (p, \rho, D), \tag{3.2}$$

donde p es el estado nuevo, ρ es el símbolo que será escrito en el lugar de σ y

$$D \in \{\to, \leftarrow, -\} \tag{3.3}$$

es la dirección a la cual moverá el puntero. Si el puntero mueve fuera de la sucesión de entrada a la derecha, el símbolo que es leído es siempre $\sqcup$ (un símbolo blanco) hasta que la máquina lo reemplaza por imprimir algo en esa posición de la cinta. El largo de la cinta en todo momento es la posición más a la derecha leído por la máquina, es decir, la mayor cantidad de posiciones utilizada por la máquina hasta actualidad.

Cada programa comienza con la máquina en el estado inicial s con la cinta inicializada a contener $\triangleright x$, donde x es una sucesión finita de símbolos en $(\Sigma - \{\sqcup\})^*$, y el puntero puntando a $\triangleright$ en la cinta. La sucesión x es la entrada de la máquina.

Se dice que la máquina se ha *detenido* si se ha llegado a uno de los tres estados de alto $\{$"alto", "sí", "no"$\}$. Si la máquina se detuvo en "sí", la máquina *acepta* la entrada. Si la máquina se detuvo en "no", la máquina *rechaza* su entrada. La *salida* $M(x)$ de la máquina M con la entrada x se define como

(I) $M(x) =$ "sí" si la máquina acepta x,

(II) $M(x) =$ "no" si la máquina rechaza x,

(III) $M(x) = y$ si la máquina llega a "alto" y $\triangleright y \sqcup \sqcup \ldots$ es la sucesión escrita en la cinta de M en el momento de detenerse y

(IV) $M(x) = \nearrow$ si M nunca se detiene con la entrada x.

Un ejemplo es la máquina siguiente para el cómputo de $n+1$ dado $n \in \mathbb{Z}$, $n > 0$, en representación binaria. Se tiene dos estados s y q (además del estado "alto") y cuatro símbolos: $\Sigma = \{0, 1, \sqcup, \triangleright\}$. La función de transición δ se define como

$p \in K$	$\sigma \in \Sigma$	$\delta(p, \sigma)$
s,	0	$(s, 0, \rightarrow)$
s,	1	$(s, 1, \rightarrow)$
s,	$\sqcup$	$(q, \sqcup, \leftarrow)$
s,	$\triangleright$	$(s, \triangleright, \rightarrow)$
q,	0	("alto", $1, -$)
q,	1	$(q, 0, \leftarrow)$
q,	$\triangleright$	("alto", $\triangleright, \rightarrow$)

Una *configuración* (q, w, u) es tal que $q \in K$ es el estado actual y $w, u \in \Sigma^*$ tal que w es la parte de la sucesión escrita en la cinta a la *izquierda* del puntero, incluyendo el símbolo debajo del puntero actualmente, y u es la sucesión a la derecha del puntero.

La relación "rinde en un paso" $\overset{M}{\to}$ es la siguiente:

$$(q, w, u) \overset{M}{\to} (q', w', u') \tag{3.4}$$

así que si σ es el último símbolo de w y $\delta(q, \sigma) = (p, \rho, D)$, aplica que $q' = p$ y los w' y u' se construye según (p, ρ, D). Por ejemplo, si $D = \to$, tenemos w' igual a w con el último símbolo reemplazado por ρ y el primer símbolo de u concatenado. Si u es vacía, se adjunta un $\sqcup$ a w. Si u no es vacía, u' es como u pero sin el primer símbolo, y si u es vacía, u' tambin la es.

Para la relación "rinde en k pasos", se define

$$(q, w, u) \overset{M}{\to}^{k} (q', w', u') \tag{3.5}$$

si y sólo si existen configuraciones $(q_i, w_i, u_i), i = 1, \ldots, k+1$ así que

(I) $(q, w, u) = (q_1, w_1, u_1)$,

(II) $(q_i, w_i, u_i) \overset{M}{\to} (q_{i+1}, w_{i+1}, u_{i+1}), i = 1, \ldots, k$ y

(III) $(q', w', u') = (q_{k+1}, w_{k+1}, u_{k+1})$.

La relación de "rinde en general" se define como

$$(q, w, u) \overset{M}{\to}^{*} (q', w', u') \tag{3.6}$$

si y sólo si $\exists k \geq 0$ tal que $(q, w, u) \overset{M}{\to}^{k} (q', w', u')$. La relación $\overset{M}{\to}^{*}$ es la clausura transitiva y reflexiva de $\overset{M}{\to}$.

Las máquinas Turing son una representación bastante natural para resolver muchos problemas sobre sucesiones, como por ejemplo reconocer lenguajes: sea $L \subset (\Sigma - \{\sqcup\})^*$ un lenguaje. Una máquina Turing M *decide* el lenguaje L si y sólo si para toda sucesión $x \in (\Sigma - \{\sqcup\})^*$ aplica que si $x \in L$, $M(x) =$ "sí" y si $x \notin L$, $M(x) =$ "no". La clase de lenguajes decididos por alguna máquina Turing son los lenguajes *recursivos*.

Una máquina Turing *acepta* un lenguaje L si para toda sucesión $x \in (\Sigma \setminus \{\sqcup\})^*$ aplica que si $x \in L$, $M(x) =$ "sí", pero si $x \notin L$, $M(x) = \nearrow$. Los lenguajes aceptados por algún máquina Turing son *recursivamente numerables*. Nota que si L es recursivo, tambin es recursivamente numerable.

Para resolver un problema de decisión por una máquina Turing, lo que se hace es decidir un lenguaje que consiste de representaciones de las instancias del problema que corresponden a la respuesta "sí". Los problemas de optimización están resueltos por máquinas Turing que hagan el cómputo de una función apropiada de sucesiones a sucesiones, representando tanto la entrada como la salida en formato de sucesiones con un alfabeto adecuado.

Tcnicamente, cualquier "objeto matemático finito" puede ser representado por una sucesión finita con un alfabeto adecuado. Por lo general, números siempre deberían estar representados como números binarios.

Por ejemplo, para representar grafos, se puede preparar una sucesión con información de n y despus los elementos de su matriz de adyacencia. Si existen varias maneras de hacer la codificación, todas las representaciones tienen largos polinomialmente relacionados: siendo N el largo de una representación de una dada instancia, las otras tienen largo no mayor a $p(N)$ donde $p()$ es algún polinomio. La única excepción es la diferencia entre la representación singular (ingls: unary) en comparación con la representación binaria: la singular necesita una cantidad exponencial de símbolos en la cinta.

3.2. Máquinas de acceso aleatorio

Una pregunta interesante es si se puede implementar *cualquier* algoritmo como una máquina Turing. Hay una conjectura que dice que cualquier intento razonable de modelado matemático de algoritmos computacionales y su eficacia (en trminos de tiempo de ejecución) resultará en un modelo de computación y costo de operaciones que es equivalente, con diferencia polinomial, a lo de máquinas Turing.

Un modelo ideal de computación es lo de máquinas de acceso aleatorio (ingls: random access machines, RAM) . Un RAM es capaz de manejar números enteros de tamao arbitrario. La estructura de datos de un RAM es un arreglo de *registros* $R_0, R_1, R_2, \ldots$, cada uno con capacidad de un entero cualquiera, posiblemente negativo. Un programa RAM es una sucesión finita de instrucciones de tipo assembler, $\Pi = (\pi_1, \pi_2, \ldots, \pi_m)$. La entrada al programa está guardada en un arreglo finito de registros de entrada $I_1, I_2, \ldots, I_n$.

El primer registro r_0 sirve como una acumuladora y las instrucciones posibles se muestra en el cuadro 3.1

Una *configuración* es un par $C = (\kappa, \mathcal{R})$, donde κ es el contador del programa que determina cual instrucción de Π se está ejecutando y $\mathcal{R} = \{(j_1, r_{j_1}), (j_2, r_{j_2}), \ldots, (j_k, r_{j_k})\}$ es un conjunto finito de pares registro-valor. La configuración inicial es $(1, \emptyset)$. Se define una relación de un paso

$$(\kappa, \mathcal{R}) \overset{\Pi, I}{\to} (\kappa', \mathcal{R}') \tag{3.7}$$

entre las configuraciones para un programa RAM Π y una entrada I de n instrucciones: κ' es el valor nuevo de κ despus de haber ejecutado la instrucción en posición κ (nota que no es necesariamente $\kappa' = \kappa + 1$) y $\mathcal{R}'$ es una versión posiblemente modificada de $\mathcal{R}$ donde algún par (j, x) puede haber sido removido y algún par (j', x') aadido según la instrucción en posición κ del programa Π. Esta relación induce otras relaciones $\overset{\Pi, I^k}{\to}$ ("da en k pasos") y $\overset{\Pi, I^*}{\to}$ ("da eventualmente") como en el caso de las máquinas Turing.

Cuadro 3.1: Instrucciones de las máquinas de acceso aleatorio (RAM). Aquí j es un entero, r_j es el contenido actual del registro R_j, i_j es el contenido del registro de entrada I_j. La notación x significa que puede ser reemplazado por cualquier de los tres operadores j, $\uparrow j$ o $= j$ — x' es el resultado de tal reemplazo. κ es el contador del programa que determina cual instrucción de Π se está ejecutando.

Instrucción	**Operando**	**Semántica**
READ	j	$r_0 := i_j$
READ	$\uparrow j$	$r_0 := i_{r_j}$
STORE	j	$r_j := r_0$
STORE	$\uparrow j$	$r_{r_j} := r_0$
LOAD	x	$r_0 := x'$
ADD	x	$r_0 := r_0 + x'$
SUB	x	$r_0 := r_0 - x'$
HALF		$r_0 := \lfloor \frac{r_0}{2} \rfloor$
JUMP	j	$\kappa := j$
JPOS	j	si $r_0 > 0$, $\kappa := j$
JZERO	j	si $r_0 = 0$, $\kappa := j$
JNEG	j	si $r_0 < 0$, $\kappa := j$
HALT		$\kappa := 0$

Si D es una sucesión finita de enteros, se dice que una máquina de acceso aleatorio *compute* una función $\phi : D \to \mathbb{Z}$ si y sólo si para toda $I \in D$ aplica que

$$(1, \emptyset) \overset{\Pi, I^*}{\to} (0, \mathcal{R}) \tag{3.8}$$

así que $(0, \phi(I)) \in \mathcal{R}$. Como un ejemplo, considere la RAM que computa la función $\phi(x, y) = |x - y|$ en el cuadro 3.2.

El modelo de tiempo de ejecución es el siguiente: la ejecución de cada instrucción cuenta como un paso de computación. La abstracción normalmente considerala sumación de enteros grandes como es algo que no se puede hacer rápidamente. Tambin hay que tomar en cuenta que multiplicación no está incluida como instrucción y que se implementa por sumación repetida. El tamao de la entrada se considera en *logaritmos*: sea b_i una representación binaria del valor absoluto un entero i sin ceros no significativos al comienzo.

Cuadro 3.2: Una RAM para computar la función $\phi(x, y) = |x - y|$ para enteros arbitrarios x, y. La entrada del ejemplo es $I = (6, 10)$, o sea, $x = 6$ y $y = 10$, lo que nos da como resultado $\phi(I) = 4$. Las configuraciones para el ejemplo se muestra a la derecha, mientras el programa general está a la izquierda.

Programa		Configuración
READ	2	$(1, \emptyset)$
STORE	2	$(2, \{(0, 10)\})$
READ	1	$(3, \{(0, 10), (2, 10)\})$
STORE	1	$(4, \{(0, 6), (2, 10)\})$
SUB	2	$(5, \{(0, 6), (2, 10), (1, 6)\})$
JNEG	8	$(6, \{(0, -4), (2, 10), (1, 6)\})$
HALT		$(8, \{(0, -4), (2, 10), (1, 6)\})$
LOAD	2	$(9, \{(0, 10), (2, 10), (1, 6)\})$
SUB	1	$(10, \{(0, 4), (2, 10), (1, 6)\})$
HALT		$(0, \{(0, 4), (2, 10), (1, 6)\})$

Para valores negativos, se supone que haya un bit "gratuito" para el signo. El largo del entero en el contexto RAM es el número de bits en b_i. El largo de la entrada entera $\mathcal{L}(I)$ es la suma de los largos de los enteros en estos trminos.

Se dice que un programa RAM Π *computa* una función $\phi : D \to \mathbb{Z}$ en tiempo $f(n)$ donde $f : \mathbb{N}^+ \to \mathbb{N}^+$ si y sólo si para toda $I \in D$ aplica que

$$(1, \emptyset) \overset{\Pi, I}{\to}{}^{k} (0, \mathcal{R}) \tag{3.9}$$

así que $k \leq f(\mathcal{L}(I))$.

Con una RAM, se puede simular una máquina Turing M con un alfabeto $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$ con prdida lineal de eficiencia. El rango de entradas posibles para la RAM simulando a M es

$$D_\Sigma = \{(i_1, \ldots, i_n, 0) \mid n \geq 0,\ 1 \leq i_j \leq k,\ j = 1, \ldots, n\}. \tag{3.10}$$

Para un lenguaje $L \subset (\Sigma - \{\sqcup\})^*$, hay que definir una función $\phi_L : D_\Sigma \mapsto \{0, 1\}$ así que $\phi_L(i_1, \ldots, i_n, 0) = 1$ si y sólo si $\sigma_{i_1} \cdots \sigma_{i_n} \in L$. La tarea de decidir el lenguaje L

es equivalente a la tarea de computar ϕ_L. La idea de la construcción es preparar una subrutina RAM para simular cada transición de una máquina Turing M que decide L. Eso se puede probar formalmente, llegando a la teorema siguiente:

Teorema 1. *ramturing*

Dado un lenguaje L que se puede decidir en tiempo $f(n)$ por una máquina Turing, existe una RAM que computa la función ϕ_L en tiempo $\mathcal{O}(f(n))$.

Tambin funciona viceversa: cada RAM se puede simular con una máquina Turing con prdida de eficiencia polinomial. Primero hay que representar la sucesión de entrada $I = (i_1, \ldots, i_n)$ de enteros como una cadena $b_I = b_1; b_2; \ldots; b_n$ donde b_j es la representación binaria del entero i_j. Ahora sea D un conjunto de sucesiones finitas de enteros y $\phi : D \to \mathbb{Z}$. Se dice que una máquina Turing M computa ϕ si y sólo si para cualquier $I \in D$ aplica que $M(b_I) = b_{\phi(I)}$, donde $b_{\phi(I)}$ es la representación binaria de $\phi(I)$.

La prueba de este resultado utiliza máquinas Turing con *siete* cintas. Para muchos casos, es más fácil considerar máquinas Turing de *cintas múltiples*, aunque todas esas máquinas pueden ser simuladas por una máquina Turing ordinaria. Las siete cintas sirven los propósitos siguientes:

(I) una cinta para la entrada,

(II) una cinta para representar los contenidos de los registros en su representación binaria: pares de número de registro y su contenido separados por el símbolo ";",

(III) una cinta para el contador de programa κ,

(IV) una cinta para el número de registro que se busca actualmente y

(V) tres cintas auxiliares para usar en la ejecución de las instrucciones.

Teorema 2. *Si un programa RAM compute la función ϕ en tiempo $f(n)$, existe necesariamente una máquina Turing M de siete cintas que compute la misma función ϕ en tiempo $\mathcal{O}(f(n)^3)$.*

La idea es que cada instrucción del programa RAM Π se implementa por un grupo de estados de M. En la segunda cinta de la máquina hay $\mathcal{O}(f(n))$ pares. Se necesita $\mathcal{O}(\ell f(n))$ pasos para simular una instrucción de Π, donde ℓ es el tamao máximo de todos los enteros en los registros. Entonces, simulación de Π con M requiere $\mathcal{O}(\ell f(n)^2)$ pasos.

Todavía habrá que establecer que $\ell = \mathcal{O}(f(n))$ para llegar al teorema 2. Cuando se ha ejecutado t pasos del programa Π con la entrada I, el contenido de cada registro tiene largo máximo $t + \mathcal{L}_i + b_j$ donde j es el entero mayor referido a el cualquier instrucción de Π. Esto es válido para $t = 0$. Por inducción, probamos que si esto es verdad hasta el paso número $t - 1$, será válido tambin en el paso número t. Para establecer la

prueba por inducción, hay que analizar los casos de diferentes tipos de instrucciones. Las instrucciones que son saltos, HALT, LOAD, STORE o READ no crean valores nuevos, por lo cual no alteran la validez de la proposición. Para la operación aritmtica ADD de dos enteros i y j, el largo del resultado es uno más el largo del operando mayor (por aritmtica binaria), y como el operando mayor tiene largo máximo $t-1+\mathcal{L}_I+b_j$, hemos establecido

$$1+t-1+\mathcal{L}_I+b_j = t+N(I)+N(B) \tag{3.11}$$

y así llegado a validar el teorema 2.

3.3. Máquinas Turing no deterministas

El modelo de máquinas Turing *no deterministas* (NTM) no es un modelo realista de de computación, pero resulta de alta utilidad al definir complejidad computacional. Se puede simular cada máquina Turing no determinista con una máquina Turing determinista con una pérdida exponencial de eficiencia. La pregunta interesante — y abierta — es si es posible simularlas con prdida solamente polinomial de eficiencia.

Una máquina Turing no determinista es un cuádruple $N=(K,\Sigma,\Delta,s)$ donde la diferencia a una máquina Turing determinista es que ahora Δ no es una función de transición, pero una *relación de transición*:

$$\Delta \subset (K\times\Sigma)\times[(K\cup\{\text{“alto”},\text{“sí”},\text{“no”}\})\times\Sigma\times\{\rightarrow,\leftarrow,-\}]. \tag{3.12}$$

La definición de una configuración no cambia, pero la relación de un paso se generaliza: $(q,w,u)\overset{N}{\rightarrow}(q',w',u')$ si y sólo si la transición está representada por algún elemento que pertenezca en Δ.

Cualquier computación de N es una secuencia de selecciones no deterministas. Una máquina Turing no determinista *acepta* su entrada si existe alguna secuencia de selecciones no deterministas que resulta en el estado de aceptación “sí”. Una máquina Turing no determinista *rechaza* su entrada si no existe ninguna secuencia de selecciones no deterministas que resultara en “sí”. El *grado de no determinismo* de una NTM N es la cantidad máxima de movimientos para cualquier par estado-símbolo en Δ.

Se dice que una NTM N decide un lenguaje L si y sólo si para todo $x\in\Sigma^*$ aplica lo siguiente:

$$x\in L \text{ si y sólo si } (s,\triangleright,x)\overset{N}{\rightarrow}{}^{*}\ (\text{“sí”},w,u) \tag{3.13}$$

para algunas cadenas w y u.

Además, una NTM N decide un lenguaje L *en tiempo* $f(n)$ si y sólo si N decide L y para todo $x\in\Sigma^*$ aplica lo siguiente:

$$\big((s,\triangleright,x)\overset{N}{\rightarrow}{}^{k}\ (q,w,u)\big)\Rightarrow\big(k\leq f(|x|)\big). \tag{3.14}$$

Esto quiere decir que todos los caminos de computación en el árbol de las decisiones no deterministas tiene largo máximo $f(|x|)$.

3.4. Máquina Turing universal

Como las hemos definido, cada máquina Turing es capaz de resolver un cierto problema. Ahora definamos una máquina Turing *universal* U la entrada de la cual es *la descripción de una máquina Turing* M junto con una entrada x para M y la función de U es tal que U simula a M con x así que $U(M;x) = M(x)$.

Habrá que ver cómo representar una máquina Turing por una cadena se símbolos de algún alfabeto. Dada una máquina Turing $M = (K, \Sigma, \delta, s)$, podemos codificarlo utilizando números enteros de la manera siguiente, donde $\varsigma = |\Sigma|$ y $k = |K|$:

$$\begin{aligned} \Sigma &= \{1, 2, \ldots, \varsigma\}, \\ K &= \{\varsigma + 1, \varsigma + 2, \ldots, \varsigma + k\}, \\ \text{dónde } s &= \varsigma + 1 \text{ y finalmente} \\ \{\leftarrow, \rightarrow, -, \text{“alto”}, \text{“sí”}, \text{“no”}\} &= \{\varsigma + k + 1, \varsigma + k + 2, \ldots, \varsigma + k + 6\}. \end{aligned} \tag{3.15}$$

Ahora, para lograr un alfabeto finito, codificamos todo esto con las representaciones binarias y el símbolo “;”: $M = (K, \Sigma, \delta, s)$ está representada por $M_b = b_\Sigma; b_K; b_\delta$ donde cada entero i está representado como b_i con exactamente $\lceil \log(\varsigma + k + 6) \rceil$ bits y la codificación de δ es una secuencia de pares $((q, \sigma), (p, \rho, D))$ de la función de transición δ utilizando las representaciones binarias de sus componentes.

La máquina universal U necesita además de la cinta de entrada donde tiene x, una cinta S_1 para guardar la descripción M_b de M y otra cinta S_2 para guardar la configuración actual de M, (q, w, u). Cada paso de la simulación consiste de tres fases:

(I) Buscar en S_2 para el entero que corresponde al estado actual de M.

(II) Buscar en S_1 para la regla de δ que corresponde al estado actual.

(III) Aplicar la regla.

(IV) En cuanto M para, tambin U para.

3.5. Máquina Turing precisa

Sea M una máquina Turing con múltiples cintas, determinista o no, con o sin entrada y salida. La máquina M es *precisa* si existen funciones f y g tales que para todo $n \geq 0$,

para toda entrada x del largo $|x| = n$ y para cada computación de M aplica que M para despus de exactamente $f(|x|)$ pasos de computación y todas sus cintas que no son reservados para entrada o salida tienen largo exactamente $g(|x|)$ cuando M para.

Dada una máquina M (determinista o no) que decide el lenguaje L en tiempo $f(n)$ donde f es una función correcta de complejidad. Entonces necesariamente existe una máquina Turing precisa M' que decide L en tiempo $\mathcal{O}(f(n))$. La construcción de tal máquina M' es así que M' usa M_f para computar una "vara de medir" $\sqcap^{f(|x|)}$ y simula M o por exactamente $f(|x|)$ pasos. Lo mismo se puede definir para espacio en vez de tiempo, alterando la construcción de M' así que en la simulación de M, se ocupa exactamente $f(|x|)$ unidades de espacio.

Capítulo 4

Complejidad computacional de problemas

4.1. Clases de complejidad de tiempo

Para definir *clases de complejidad* computacional , se procede a través de las máquinas Turing deterministas y no deterministas:

Definición 1. TIMEUna clase de complejidad $\mathbf{TIME}\left(f(n)\right)$ es el conjunto de lenguajes L tales que una máquina Turing (determinista) decide L en tiempo $f(n)$.

Definición 2. NTIMEUna clase de complejidad $\mathbf{NTIME}\left(f(n)\right)$ es el conjunto de lenguajes L tales que una máquina Turing *no determinista* decide L en tiempo $f(n)$.

Definición 3. PEl conjunto $\mathbf{P}$ contiene todos los lenguajes decididos por máquinas Turing (deterministas) en tiempo *polinomial*,

$$\mathbf{P} = \bigcup_{k>0} \mathbf{TIME}\left(n^k\right). \tag{4.1}$$

Definición 4. NPEl conjunto **NP** contiene todos los lenguajes decididos por máquinas Turing *no deterministas* en tiempo *polinomial*,

$$\mathbf{NP} = \bigcup_{k>0} \mathbf{NTIME}\left(n^k\right). \tag{4.2}$$

Teorema 3. *Supone que una* NTM N *decide un lenguaje* L *en tiempo* $f(n)$*. Entonces existe una máquina Turing determinista* M *que decide* L *en tiempo* $\mathcal{O}\left(c_N^{f(n)}\right)$*, donde* $c_N > 1$ *es una constante que depende de* N*. Eso implica que*

$$\mathbf{NTIME}\left(f(n)\right) \subseteq \bigcup_{c>1} \mathbf{TIME}\left(c^{f(n)}\right). \tag{4.3}$$

Para comprobar el teorema 3, sea $N = (K, \Sigma, \Delta, s)$ una NTM y denota por d el grado de no determinismo de N. Numeramos las opciones disponibles a cada momento de decisión con los enteros $0, 1, \ldots, d-1$. Entonces, la secuencia de selecciones en un camino de computación de N se representa como una secuencia de t enteros. La máquina M que puede simular a N necesita considerar *todas las secuencias* posibles en orden de largo creciente y simula el conducto de N para cada secuencia fijada en turno, es decir, con la secuencia $(c_1, c_2, \ldots, c_t)$ la máquina M simula las acciones que hubiera tomado N si N hubiera elegido la opción c_i en su selección número i durante sus primeros t pasos de computación.

Si M llega a simular una secuencia que resulta a "sí" en N, M también termina con "sí". Si una secuencia no resulta en "sí" para N, la máquina M continua con la siguiente secuencia en su lista. Después de haber simulado todas las secuencias sin llegar a "sí" (o sea, siempre ha llegado a "no" o "alto"), M rechaza la entrada.

La cota de tiempo de ejecución $\mathcal{O}\left(c^{f(n)}\right)$ resulta como la suma del número total de secuencias posibles

$$\sum_{t=1}^{f(n)} d^t = \mathcal{O}\left(d^{f(n)+1}\right) \tag{4.4}$$

y el costo de simular cada secuencia $\mathcal{O}\left(2^{f(n)}\right)$.

4.2. Clases de complejidad de espacio

Para considerar complejidad del espacio, o sea, la necesidad de memoria, hay que definir una NTM con cintas múltiples. Dada una NTM N con k cintas, se dice que N decide a un lenguaje L *dentro de espacio* $f(n)$ si N decide L y para todo $x \in (\Sigma \setminus \{\sqcup\})^*$ aplica que

$$\begin{aligned} &\left((s, \triangleright, x, \triangleright, \epsilon, \ldots, \triangleright, \epsilon,) \overset{N}{\rightarrow}{}^{*} (q, w_1, u_1, \ldots, w_k, u_k)\right) \Rightarrow \\ &\qquad \sum_{i=2}^{k-1} |w_i u_i| \leq f(|x|), \end{aligned} \tag{4.5}$$

donde $|w_i u_i|$ es el largo (en cantidad de símbolos) de la cadena concatenada de las dos cadenas w_i y u_i.

4.3. Problemas sin solución

Existen más lenguajes que máquinas de Turing. Entonces, habrá que existir lenguajes que no son decididos por ninguna máquina Turing. Un problema que no cuente con una

máquina Turing se llama un problema una problema *sin solución* (inglés: undecidable problem). Un problema sin solución muy famoso es el problema HALTING:

Problema 1 (HALTING)**.** *Dada:* una descripción M_b de una máquina Turing M y una entrada x. *Pregunta:* ¿va a parar M cuando se ejecuta con x?

El lenguaje que corresponde al HALTING es simplemente

$$H = \{M_b; x \mid M(x) \neq \nearrow\}, \tag{4.6}$$

donde M_b es una descripción de M. Resulta que HALTING es recursivamente numerable, lo que se puede comprobar por modificar un poco la máquina Turing universal U. La máquina modificada U' es tal que cada estado de alto de U está reemplazada con "sí". Entonces, si $M_b; x \in H$, por definición $M(x) \neq \nearrow$ por lo cual $U(M_b; x) \neq \nearrow$, que resulta en $U'(M_b; x) =$ "sí". Al otro lado, si $M_b; x \notin H$, aplica que $M(x) = U(M_b; x) = U'(M_b; x) = \nearrow$.

Sin embargo, HALTINGño es recursivo: supone que H sea recursivo, o sea, alguna máquina Turing M_H decide H. Dada la máquina D con entrada M tal que

$$D(M_b) = \begin{cases} \nearrow, & \text{si } M_H(M_b; M_b) = \text{"sí"}, \\ \text{"sí"}, & \text{en otro caso.} \end{cases} \tag{4.7}$$

tomamos la representación de D misma, D_b. ¿Qué es el resultado de $D(D_b)$? Si $D(D_b) = \nearrow$, tenemos $M_H(D_b; D_b) =$ "sí" que implica que $D(D_b) \neq \nearrow$ y llegamos a una contradicción. Entonces, si $D(D_b) \neq \nearrow$, tenemos $M_H(D_b, D_b) \neq$ "sí". Pero como M_H es la máquina que decide H, $M_H(D_b, D_b) =$ "no", por lo cual $D_b; D_b \notin H$, o sea, $D(D_b) = \nearrow$, que nos da otra contradicción. Entonces H no puede ser recursivo.

4.4. Problema complemento

Dado un alfabeto Σ y un lenguaje $L \subseteq \Sigma^*$, el *complemento* del lenguaje L es el lenguaje

$$\bar{L} = \Sigma^* \setminus L. \tag{4.8}$$

En el contexto de problemas de decisión, si la respuesta para el problema A con la entrada x es "sí", la respuesta para su *problema complemento* A Complement es "no" y viceversa:

$$\text{A Complement}(x) = \text{"sí" si y sólo si } A(x) = \text{"no"}. \tag{4.9}$$

Teorema 4. *Si un lenguaje L es recursivo, su complemento $\bar{L}$ también lo es.*

Teorema 5. *Un lenguaje L es recursivo si y sólo si los ambos lenguajes L y $\bar{L}$ son recursivamente numerables.*

Cada lenguaje recursivo es también recursivamente numerable. En la otra dirección la demostración necesita un poco más atención: se va a simular con una máquina S las dos máquinas M_L y $M_{\bar{L}}$ con la misma entrada x tomando turnos paso por paso. Si M_L acepta a x, S dice "sí" y si $M_{\bar{L}}$ acepta a x, S dice "no".

Entonces, el complemento $\bar{H}$ de H (el lenguaje del problema HALTING) no es recursivamente numerable. Resulta que cualquier propiedad no trivial de las máquinas Turing no tiene solución:

Teorema 6 (Teorema de Rice)**.** *Sea R el conjunto de lenguajes recursivamente numerables y $\emptyset \neq C \subset R$. El siguiente problema es sin solución: dada una máquina Turing M, ¿aplica que $L(M) \in C$ si $L(M)$ es el lenguaje aceptado por M?*

4.5. Algunos problemas fundamentales

4.5.1. Problemas de lógica booleana

El problema de *satisfiabilidad* (SAT) es el siguiente:

Problema 2 (SAT)**.** *Dada:* una expresión booleana ϕ en CNF. *Pregunta:* ¿es ϕ satisfactible?

Utilizando tablas completas de asignaciones, se puede resolver el problema SAT en tiempo $\mathcal{O}\left(n^2 \cdot 2^n\right)$. Aplica que SAT $\in$ **NP**, pero no se sabe si SAT está también en $\mathbf{P}$.

Para demostrar que SAT $\in$ **NP**, formulamos una máquina Turing no determinista para examinar si ϕ es satisfactible: asignamos para cada variable $x_i \in X(\phi)$ de una manera no determinista un valor $T(x) := \top$ o alternativamente $T(x) := \bot$. Si resulta que $T \models \phi$, la respuesta es "sí". En el otro caso, la respuesta es "no". El problema complemento de SAT es:

Problema 3 (SAT Complement)**.** *Dada:* una expresión booleana ϕ en CNF. *Pregunta:* ¿es ϕ no satisfactible?

Una clase especial de interés de SAT es la versión donde solamente *cláusulas Horn* están permitidos. Una cláusula Horn es una disyunción que contiene por máximo un literal positivo (es decir, todas menos una variable tienen que ser negadas). Si una cláusula Horn contiene un literal positivo, se llama *implicación* por la razón siguiente: la cláusula Horn

$$\left((\neg x_1) \vee (\neg x_2) \vee \ldots (\neg \vee) x_k \vee x_p\right) \tag{4.10}$$

es lógicamente equivalente a la expresión

$$(x_1 \wedge x_2 \wedge \ldots \wedge x_l) \rightarrow x_p \tag{4.11}$$

por las reglas de transformación de la sección 1.7.1. El problema HORNSAT es el siguiente:

Problema 4 (HORNSAT)**.** *Dada:* una expresión booleana ϕ que es una conjunción de cláusulas Horn. *Pregunta:* ¿es ϕ satisfactible?

Resulta que HORNSAT sí tiene un algoritmo polinomial, por lo cual HORNSAT $\in \mathbf{P}$. El algoritmo para determinar satisfiabilidad para una expresión ϕ que consiste de cláusulas Horn es lo siguiente:

(I) Inicializa $T := \emptyset$ y el conjunto S para contener las cláusulas.

(II) Si hay una implicación

$$\phi_i = (x_1 \wedge x_2 \wedge \cdots \wedge x_n) \rightarrow y \in S$$

tal que $(X(\phi_i) \setminus \{y\}) \subseteq T$ pero $y \notin T$, haz que y sea verdadera por asignar $T := T \cup \{y\}$.

(III) Si T cambió, vuelve a repetir el paso II.

(IV) Cuando ya no haya cambios en T, verifica para todas las cláusulas en S que consisten de puros literales negados, $\phi_i = \neg x_1 \vee \cdots \vee \neg x_n$:

- Si existe un literal $\neg x_i$ tal que $x_i \notin T$, el resultado es que ϕ_i *es* satisfactible.
- Si no existe tal literal, el resultado es que ϕ_i es *no satisfactible* y así ϕ tampoco lo es.

(V) Si todas las cláusulas negativas son satisfactibles, ϕ también lo es.

Problema 5 (CIRCUITSAT)**.** *Dado:* un circuito booleano C. *Pregunta:* ¿existe una asignación $T : X(C) \rightarrow \{\top, \bot\}$ tal que la salida del circuito tenga el valor verdad?

Problema 6 (CIRCUITVALUE)**.** *Dado:* un circuito booleano C que no contenga variables.*Pregunta:* ¿tiene el valor $\top$ la salida del circuito?

CIRCUITSAT $\in$ **NP**, pero CIRCUITVALUE $\in \mathbf{P}$. Para CIRCUITVALUE, no es necesario definir un T como $X(C) = \emptyset$.

4.5.2. Problemas de grafos

En esta sección se formulan algunos de los problemas computacionales fundamentales que involucran grafos como parte de su entrada.

Problema de alcance

Un problema básico de grafos es el problema de *alcance* :

Problema 7 (REACHABILITY)**.** *Dado:* un grafo $G = (V, E)$ y dos vértices $v, u \in V$. *Pregunta:* ¿existe un camino de v a u?

Su problema complemento es obviamente el siguiente:

Problema 8 (REACHABILITY Complement)**.** *Dado:* un grafo $G = (V, E)$ y dos vértices $v, u \in V$. *Pregunta:* ¿es verdad que no existe ningún camino de v a u?

Un algoritmo básico para REACHABILITY es el algoritmo Floyd-Warshall que además resuelve no solamente la existencia de los caminos pero en grafos ponderados encuentra los caminos más cortos. El algoritmo compara todos los caminos posibles entre todos los pares de vértices en un grafo de entrada. El algoritmo construye de una manera incremental estimaciones a los caminos más cortos entre dos vértices hasta llegar a la solución óptima.

Etiquetamos los vértices de $G = (V, E)$ tal que $V = \{1, 2, \ldots, n\}$. Utilizamos como subrutina $\mathtt{cc}(i, j, k)$ que construye el camino más corto entre los vértices i y j pasando *solamente* por vértices con etiqueta menor o igual a k. Para construir un camino de i a j con solamente vértices intermedios con menores o iguales a $k+1$, tenemos dos opciones: la primera es que el camino más corto con etiquetas menores o iguales a $k + 1$ utiliza solamente los vértices con etiquetas menores o iguales a k o que existe algún camino que primero va de i a $k + 1$ y después de $k + 1$ a j tal que la combinación de estos dos caminos es más corto que cualquier camino que solamente utiliza vértices con etiquetas menores a $k + 1$. Esta observación nos ofrece una formulación recursiva de $\mathtt{cc}()$:

$$\mathtt{cc}(i, j, k) \;=\; \min\left\{\mathtt{cc}(i, j, k-1), \mathtt{cc}(i, k, k-1) + \mathtt{cc}(k, j, k-1)\right\}, \tag{4.12}$$

con la condición inicial siendo para grafos ponderados que $\mathtt{cc}(i, j, 0) = w(i, j)$ donde $w(i, j)$ es el peso de la arista $(i, j) \in E$ y para grafos no ponderados $\mathtt{cc}(i, j, 0) = 1$ para cada arista de $G = (V, E)$. En ambos casos, para los pares que *no* corresponden a una arista, $\mathtt{cc}(i, j, 0) = \infty$. Los pesos tienen que ser no negativos para que funcione el algoritmo.

Iterando esto para computar $\mathtt{cc}(i, j, k)$ primero con $k = 1$, después con $k = 2$, continuando hasta $k = n$ para cada par. Lo conveniente es que la información de la iteración

k se puede reemplazar con la de la iteración $k+1$. Entonces, el uso de memoria es lineal. El tiempo de computación es $\mathcal{O}(n^3)$ — es un ejercicio fácil ver porqué.

Ciclos y caminos de Hamilton

Los problemas de decidir si un grafo de entrada contiene un camino o un ciclo de Hamilton son los siguientes:

Problema 9 (HAMILTON PATH)**.** *Dado:* un grafo $G = (V, E)$. *Pregunta:* ¿existe un camino C en G tal que C visite cada vértice exactamente una vez?

Problema 10 (HAMILTON CYCLE)**.** *Dado:* un grafo $G = (V, E)$. *Pregunta:* ¿ existe un ciclo C en G tal que C visite cada vértice exactamente una vez?

Un problema de optimización basada en el problema de decisión HAMILTON CYCLE es el *problema del viajante (de comercio)* (inglés: travelling salesman problem) TSP es una versión en un grafo ponderado: habrá que encontrar un ciclo de Hamilton con costo mínimo, donde el costo es la suma de los pesos de las aristas incluidas en el ciclo.

La problema de decisión que corresponde a la versión ponderada es TSPD:

Problema 11 (TSPD)**.** *Dado:* un grafo ponderado $G = (V, E)$ con pesos en las aristas y una constante c. *Pregunta:* ¿existe un ciclo C en G tal que C visite cada vértice exactamente una vez y que la suma de los pesos de las aristas de C sea menor o igual a c?

Camarilla y conjunto independiente

El problema de camarilla (CLIQUE) trata de la existencia de subgrafos completos, o sea, conjuntos de vértices $C \subseteq V$ tales que $\forall v, u \in C$ aplica que $\{v, u\} \in E$:

Problema 12 (CLIQUE)**.** *Dado:* grafo no dirigido $G = (V, E)$ y un entero $k > 0$. *Pregunta:* ¿existe un subgrafo completo inducido por el conjunto $C \subseteq V$ tal que $|C| = k$?

Un problema muy relacionado es el problema de *conjunto independiente* (INDEPENDENT SET) que es un conjunto $I \subseteq V$ tal que $\forall v, u \in I$ aplica que $\{v, u\} \notin E$:

Problema 13 (INDEPENDENT SET)**.** *Dado:* un grafo no dirigido $G = (V, E)$ y un entero $k > 0$. *Pregunta:* ¿existe un subgrafo inducido por el conjunto $I \subseteq V$ tal que $|I| = k$ y que no contenga arista ninguna.?

Una observación importante es que si C es una camarilla en $G = (V, E)$, C es un conjunto independiente en el grafo complemento $\bar{G}$.

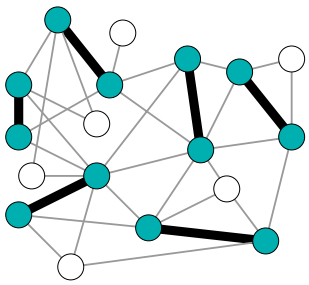

Figura 4.1: Un acoplamiento de un grafo pequeño: las aristas gruesas negras forman el acoplamiento y los vértices azules están acoplados.

Acoplamiento

Un *acoplamiento* (inglés: matching) es un conjunto $\mathcal{M} \subseteq E$ de *aristas disjuntas no adyacentes*. Un vértice v está *acoplado* si $\mathcal{M}$ contiene una arista incidente a v. Si no está acoplado, el vértice está *libre*. La figura 4.1 muestra un ejemplo simple.

Un *acoplamiento máximo* $\mathcal{M}_{\text{máx}}$ es uno que contiene el número máximo posible de aristas. No es necesariamente único. Un *acoplamiento maximal* tiene la propiedad de todas las aristas que *no pertenecen* a $\mathcal{M}$ están adyacentes a por lo menos una arista en $\mathcal{M}$. Entonces, si se añade una arista a tal $\mathcal{M}$, el conjunto que resulta ya no es un acoplamiento. Todos los acoplamientos máximos deben ser maximales, pero no todos los maximales deben de ser máximos.

El *número de acoplamiento* de $G = (V, E)$ es $|\mathcal{M}_{\text{máx}}|$. El número de vértices libres (dado G y $\mathcal{M}$) se llama el *déficit* del acoplamiento en G. Un *acoplamiento perfecto* cubre todos los vértices del grafo. Cada acoplamiento perfecto es máximo y maximal y tiene un número de acoplamiento igual a $\frac{|n|}{2}$.

Para mejorar un acoplamiento existente, se utiliza el método de *caminos aumentantes*. Un *camino alternante* es un camino en G al cual sus aristas alternativamente pertenecen y no pertenecen a $\mathcal{M}$. Un *camino aumentante* $\mathcal{A}$ es un camino alternante de un vértice libre v a otro vértice libre u.

Dado un camino aumentante $\mathcal{A}$, podemos realizar un intercambio de las aristas de $\mathcal{A}$ incluidas en $\mathcal{M}$ para las aristas de $\mathcal{A}$ no en $\mathcal{M}$ para construir un acoplamiento nuevo $\mathcal{M}'$ tal que $|\mathcal{M}'| = |\mathcal{M}| + 1$, donde

$$\mathcal{M}' = \big(\mathcal{M} \setminus (\mathcal{M} \cap \mathcal{A})\big) \cup \big(\mathcal{A} \setminus (\mathcal{M} \cap \mathcal{A})\big). \tag{4.13}$$

Un acoplamiento $\mathcal{M}$ es máximo si y sólo si no contiene ningún camino aumentante. La figura 4.2 muestra un ejemplo del intercambio por un camino aumentante.

El *algoritmo Húngaro* es un algoritmo de tiempo $\mathcal{O}(n^3)$ que construye en base de la matriz se adyacencia de un grafo de entrada $G = (V, E)$ un subgrafo que contiene las

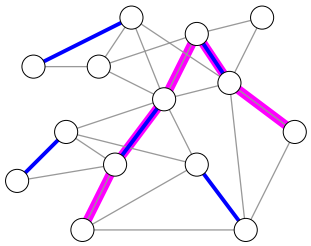

Figura 4.2: Un ejemplo del método de camino aumentante para mejorar un acoplamiento. En azul se muestra el acoplamiento actual. Se puede aumentar el tamaño intercambiando la pertenencia con la no pertenencia al acoplamiento de las aristas del camino aumentante indicado.

aristas de un acoplamiento máximo Es un algoritmo iterativo para eliminar aristas que no formarán parte del acoplamiento y se acredita a Harold Kuhn (1955).

Para problemas de acoplamiento, muy típicamente el grafo de entrada es bipartito. En acoplamientos de grafos ponderados, se considera la suma de los valores de las aristas en $\mathcal{M}$ y no solamente el número de aristas incluidas en el acoplamiento.

Cubiertas

Una *cubierta* (inglés: cover) es un subconjunto de vértices (o aristas) que de una manera "cubre" todas las aristas (resp. todos los vértices).

Formalmente, una *cubierta de aristas* es un conjunto $\mathcal{C}_E$ de aristas tal que para cada vértice $v \in V$, $\mathcal{C}$ contiene una arista incidente a v. Similarmente, una *cubierta de vértices* es un conjunto $\mathcal{C}_V$ de vértices tal que para cada arista $\{v, w\}$, por lo menos uno de los vértices incidentes está incluido en $\mathcal{C}_V$. La meta de las problemas relacionadas suele ser encontrar un conjunto de cardinalidad *mínima* que cumpla con la definición. La figura 4.3 tiene ejemplos de los dos tipos de cubiertas.

Para un acoplamiento máximo $\mathcal{M}$ y una cubierta de aristas mínima $\mathcal{C}_E$, aplica siempre que $|\mathcal{C}_E| \geq |\mathcal{M}|$. Además, dado un $\mathcal{M}$ máximo, se puede fácilmente construir una cubierta $\mathcal{C}_E$ óptima. Un problema de decisión relacionado es el siguiente:

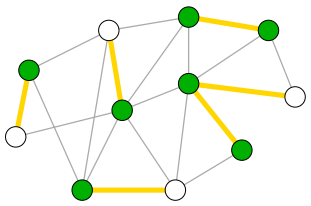

Figura 4.3: Ejemplos de cubiertas: los vértices verdes cubren todas las aristas y las aristas amarillas cubren todos los vértices.

Problema 14 (VERTEX COVER)**.** *Dado:* un grafo $G = (V, E)$ no dirigido y un entero $k > 0$. *Pregunta:* ¿existe un conjunto de vértices $C \subseteq V$ con $|C| \leq k$ tal que $\forall\{v, u\} \in E$, o $v \in C$ o $u \in C$.

Existe una conexión fuerte entre conjuntos independientes y camarillas y cubiertas de vértices : Un conjunto $I \subseteq V$ de un grafo $G = (V, E)$ es un conjunto independiente si y sólo si I es una camarilla en el grafo complemento de G. Además, I es un conjunto independiente en G si y sólo si $V \setminus I$ es una cubierta de vértices G.

Flujos

Para problemas de flujos, típicamente se considera grafos ponderados (y posiblemente dirigidos). Además se fija *dos vértices especiales*: un vértice *fuente* s y un vértice *sumidero* t. El grafo necesita ser conexo en un sentido especial: $\forall v \in V$, existe un camino (dirigido en en caso de grafos dirigidos) del fuente s al sumidero t que pasa por el vértice v. Vértices que no cumplan con este requisito pueden ser eliminados en un paso de preprocesamiento para preparar una instancia de un problema de flujos.

En el grafo de entrada ponderado, los valores de las aristas se llaman *capacidades* $c(v, w) \leq 0$. Para generalizar la definición de capacidad a ser una función sobre todos los pares de vértices $c : V \times V \to \mathbb{R}$, se define: $\{v, w\} \notin E \Rightarrow c(v, w) = 0$. La figura 4.4 muestra un ejemplo de un grafo que puede ser una instancia de un problema sobre flujos.

Un *flujo positivo* es una función $f : V \times V \to \mathbb{R}$ que satisface dos restricciones: la *restricción de capacidad*:

$$\forall u, v \in V : 0 \leq f(u, v) \leq c(u, v) \tag{4.14}$$

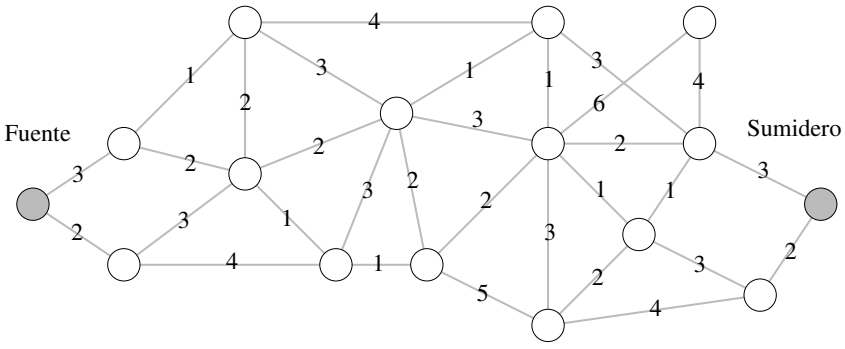

Figura 4.4: Una instancia de ejemplo para problemas de flujos: los vértices grises son el fuente (a la izquierda) y el sumidero (a la derecha). Se muestra la capacidad de cada arista.

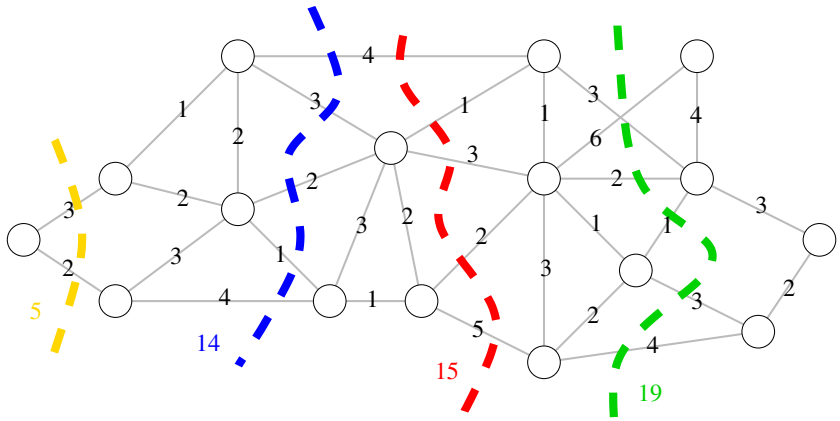

Figura 4.5: Un ejemplo de un grafo que es una instancia de un problema de flujos. Se muestra cuatro cortes con sus capacidades respectivas.

y la restricción de *conservación de flujo*: el flujo que entra es igual al flujo que sale,

$$\forall u \in V \setminus \{s,t\} : \sum_{v \in V} f(v,u) = \sum_{v \in V} f(u,v). \tag{4.15}$$

Cortes

Un *corte* $C \subseteq V$ de G es una *partición* del conjunto de vértices V en dos conjuntos: C y $V \setminus C$. En el caso especial de cortar un grafo de flujo, se exige que $s \in C$ y $t \notin C$. La *capacidad de un corte* C en un grafo no ponderado es el número de las aristas que cruzan de C a $V \setminus C$,

$$\left|\{\{v,u\} \in E \mid v \in C, w \notin C\}\right|. \tag{4.16}$$

Para un grafo ponderado la capacidad del corte es la suma de los pesos de las aristas que cruzan de C a $V \setminus C$,

$$\sum_{\substack{v \in C \\ w \notin C}} c(v,w). \tag{4.17}$$

La figura 4.5 muestra una instancia de un problema de flujos con cuatro cortes y sus capacidades.

Un *corte mínimo* de $G = (V,E)$ es un corte cuya *capacidad es mínima* entre todos los cortes de G. El problema de encontrar el corte mínimo tiene solución polinomial. Además, la capacidad del corte mínimo entre dos vértices s y t es *igual* al flujo máximo entre s y t. Cuando establecido un flujo en el grafo, la cantidad de flujo que cruza un corte es igual a cada corte del grafo.

En la mayoría de las aplicaciones de cortes de grafos, los tamaños de los dos "lados" del corte, $|C|$ y $|V \setminus C|$ no suelen ser arbitrarios. El problema de *la máxima bisección* es el siguiente:

Problema 15 (MAX BISECTION)**.** *Dado:* un grafo $G = (V, E)$ (tal que n es par) y un entero $k > 0$. *Pregunta:* ¿existe un corte C en G con capacidad mayor o igual a k tal que $|C| = |V \setminus C|$.

Coloreo

Problema 16 (k-COLORING)**.** *Dado:* un grafo no dirigido $G = (V, E)$ y un entero $k > 0$. *Pregunta:* ¿existe una asignación de colores a los vértices de V tal que ningún par de vértices $v, u \in V$ tal que $\{v, u\} \in E$ tenga el mismo color?

Capítulo 5

Clases de complejidad

Es posible definir diferentes tipos de clases de complejidad computacional. Los requisitos para definir una clase son

(I) fijar un modelo de computación (es decir, qué tipo de máquina Turing se utiliza),

(II) la modalidad de computación: determinista, no determinista, etcétera,

(III) el recurso el uso del cual se controla por la definición: tiempo, espacio, etcétera,,

(IV) la cota que se impone al recurso (es decir, una función f *correcta* de complejidad).

Una función $f : \mathbb{N} \to \mathbb{N}$ es una función correcta de complejidad si es no decreciente y existe una máquina Turing M_f de k cintas con entrada y salida tal que con toda entrada x aplica que $M_f(x) = \sqcap^{f(|x|)}$, donde $\sqcap$ es un símbolo "casi-blanco", y además M_f para después de $\mathcal{O}(|x| + f(|x|))$ pasos de computación y utiliza $\mathcal{O}(f(|x|))$ espacio en adición a su entrada. Algunos ejemplos de funciones de complejidad son

$$c,\ \sqrt{n},\ n,\ \lceil \log n \rceil,\ \log^2 n,\ n \log n,\ n^2,\ n^3 + 3n,\ 2^n,\ n!, \tag{5.1}$$

donde n es el parámetro de la función, $f(n)$, y c es un constante. Dada dos funciones correctas de complejidad f y g, también $f + g$, $f \cdot g$ y 2^f son funciones correctas.

Formalmente, una *clase de complejidad* es el conjunto de todos los lenguajes decididos por alguna máquina Turing M del tipo I que opera en el modo de operación de II tal que para todo entrada x, M necesita al máximo $f(|x|)$ unidades del recurso definido en III, donde f es la función de IV.

Dada una función correcta de complejidad f, obtenemos las clases siguientes:

$$\begin{cases} \text{tiempo} \begin{cases} \text{determinista} & \mathbf{TIME}(f) \\ \text{no determinista} & \mathbf{NTIME}(f) \end{cases} \\ \text{espacio} \begin{cases} \text{determinista} & \mathbf{SPACE}(f) \\ \text{no determinista} & \mathbf{NSPACE}(f) \end{cases} \end{cases} \tag{5.2}$$

Para definiciones más amplias, podemos utilizar una familia de funciones f cambiando un parámetro $k \geq 0$, $k \in \mathbb{Z}$: las clases más importantes así obtenidas son

$$\begin{array}{rcccl} & \text{Definición de la clase} & & & \text{Nombre} \\ \mathbf{TIME}\left(n^k\right) & = & \bigcup_{j>0} \mathbf{TIME}\left(n^j\right) & = & \mathbf{P} \\ \mathbf{NTIME}\left(n^k\right) & = & \bigcup_{j>0} \mathbf{NTIME}\left(n^j\right) & = & \mathbf{NP} \\ \mathbf{SPACE}\left(n^k\right) & = & \bigcup_{j>0} \mathbf{SPACE}\left(n^j\right) & = & \mathbf{PSPACE} \\ \mathbf{NSPACE}\left(n^k\right) & = & \bigcup_{j>0} \mathbf{NSPACE}\left(n^j\right) & = & \mathbf{NPSPACE} \\ \mathbf{TIME}\left(2^{n^k}\right) & = & \bigcup_{j>0} \mathbf{TIME}\left(2^{n^j}\right) & = & \mathbf{EXP}. \end{array} \tag{5.3}$$

Otras clases que se encuentra frecuentemente son $\mathbf{L} = \mathbf{SPACE}(\log(n))$ y $\mathbf{NL} = \mathbf{NSPACE}(\log(n))$

Para cada clase de complejidad C, existe una clase $\mathbf{co}C$ que es la clase de los complementos,

$$\left\{\bar{L} \mid L \in C\right\}. \tag{5.4}$$

Todas las clases deterministas de tiempo y espacio son *cerradas bajo el complemento*. Es decir, la operación de tomar el complemento de un lenguaje $L \in C$ resulta en un lenguaje $L' = \bar{L}$ tal que también $L' \in C$. Por ejemplo, $\mathbf{P} = \mathbf{coP}$. Lo único que hay que hacer para mostrar esto es intercambiar los estados "sí" y "no" en las máquinas Turing que corresponden.

Se puede mostrar también que las clases de *espacio* no deterministas están cerradas bajo complemento, pero es una pregunta *abierta* si también aplica para clases no deterministas de tiempo.

5.1. Jerárquias de complejidad

Si permitimos más tiempo a una máquina Turing M, logramos que M puede tratar de tareas más complejas. Para una función correcta de complejidad $f(n) \geq n$, definimos

Cuadro 5.1: La jerarquía de complejidad computacional con ejemplos: arriba están las tareas más fáciles y abajo las más difíciles.

Tiempo	**Clase**	**Ejemplo**
	Problemas con algoritmos eficientes	
$\mathcal{O}(1)$	$\mathbf{P}$	si un número es par o impar
$\mathcal{O}(n)$	$\mathbf{P}$	búsqueda de un elemento entre n elementos
$\mathcal{O}(n \log n)$	$\mathbf{P}$	ordenación de n elementos
$\mathcal{O}(n^3)$	$\mathbf{P}$	multiplicación de matrices
$\mathcal{O}(n^k)$	$\mathbf{P}$	programación lineal
	Problemas difíciles	
$\mathcal{O}(n^2 2^n)$	**NP**-completo	satisfiabilidad

un lenguaje nuevo:

$$H_f = \{M_b; x \mid M \text{ acepta a } x \text{ después de no más de } f(|x|) \text{ pasos } \} \tag{5.5}$$

El lenguaje H_f es una versión "cortada" de H, el lenguaje de HALTING.

Se puede mostrar que $H_f \in \mathbf{TIME}((f(n))^3)$ con una máquina Turing U_f de cuatro cintas que está compuesta de la máquina universal U, un simulador de máquinas de cintas múltiples con una sola cinta, una máquina de aumento de rapidez lineal y la máquina M_f que computa la "vara de medir" del largo $f(n)$, donde $n = |x|$ es el largo de la entrada x. La operación de la máquina U_f es la siguiente:

(I) M_f computa la vara de medir $\sqcap^{f(|x|)}$ para M en la cuarta cinta.

(II) La descripción binaria de M, M_b está copiada a la tercera cinta.

(III) La segunda cinta está inicializada para codificar el estado inicial s.

(IV) La primera cinta está inicializada para contener la entrada $\triangleright x$.

(V) U_f simula con una sola cinta a M.

(VI) Después de cada paso de M simulado, U_f avanza la vara de medir por una posición.

(VII) Si U_f descubre que M acepta a x en $f(|x|)$ pasos, U_f acepta su entrada.

(VIII) Si se acaba la vara de medir sin que M acepte a x, U_f rechaza su entrada.

Cada paso de la simulación necesita $\mathcal{O}\left(f(n)^2\right)$ tiempo, por lo cual con los $f(|x|)$ pasos simulados al máximo, llegamos a tiempo de ejecución total $\mathcal{O}\left(f(n)^3\right)$ para U_f.

También se puede mostrar que $H_f \notin \mathbf{TIME}\left(f(\lfloor \frac{n}{2} \rfloor)\right)$. Para comprobar esto, suponga que existe una máquina Turing M que decide a H_f en tiempo $f(\lfloor \frac{n}{2} \rfloor)$. Consideramos otra máquina D que dice "no" cuando $M(M_b; M_b) =$ "sí" y en otros casos D dice "sí". Con la entrada M_b, D necesita

$$f(\lfloor \frac{2|M_b|+1}{2} \rfloor) = f(|M_b|) \tag{5.6}$$

pasos. Si $D(D_b) =$ "sí", tenemos que $M(D_b; D_b) =$ "no", y entonces $D; D \notin H_f$. En este caso, D no puede aceptar la entrada D_b dentro de $f(|D_b|)$ pasos, que significa que $D(D_b) =$ "no", que es una contradicción.

Entonces $D(D_b) \neq$ "sí". Esto implica que $D(D_b) =$ "no" y $M(D_b; D_b) =$ "sí", por lo cual $D_b; D_b \in H_f$ y D acepta la entrada D_b en no más que $f(|D_b|)$ pasos. Esto quiere decir que $D(D_b) =$ "sí", que es otra contradicción y nos da el resultado deseado.

Utilizando estos dos resultados, podemos llegar a probar al teorema siguiente:

Teorema 7. *Si $f(n) \geq n$ es una función correcta de complejidad, entonces la clase $\mathbf{TIME}\left(f(n)\right)$ está estrictamente contenida en $\mathbf{TIME}\left((f(2n+1))^3\right)$.*

Es bastante evidente que

$$\mathbf{TIME}\left(f(n)\right) \subseteq \mathbf{TIME}\left((f(2n+1))^3\right) \tag{5.7}$$

como f es no decreciente. Ya sabemos que

$$H_{f(2n+1)} \in \mathbf{TIME}\left((f(2n+1))^3\right) \tag{5.8}$$

y que

$$H_{f(2n+1)} \notin \mathbf{TIME}\left(f\left(\lfloor \frac{2n+1}{2} \rfloor\right)\right) = \mathbf{TIME}\left(f(n)\right). \tag{5.9}$$

Este resultado implica también que $\mathbf{P} \subset \mathbf{TIME}\left(2^n\right) \subseteq \mathbf{EXP}$ por $n^k = \mathcal{O}\left(2^n\right)$ y por el resultado anterior que nos da

$$\mathbf{TIME}\left(2^n\right) \subset \mathbf{TIME}\left((2^{2n+1})^3\right) \subseteq \mathbf{TIME}\left(2^{n^2}\right) \subseteq \mathbf{EXP}. \tag{5.10}$$

Teorema 8. *Si $f(n) \geq n$ es una función correcta de complejidad, entonces aplica que $\mathbf{SPACE}\left(f(n)\right) \subset \mathbf{SPACE}\left(f(n)\log f(n)\right)$.*

En el teorema 8 es esencial el requisito de que f sea una función correcta de complejidad. Por ejemplo, existe una función *no correcta* $f : \mathbb{Z}^+ \to \mathbb{Z}^+$ tal que $\mathbf{TIME}\left(f(n)\right) = \mathbf{TIME}\left(2^{f(n)}\right)$. El truco es definir f tal que ninguna máquina Turing M con entrada x, $|x| = n$, para después de k pasos tal que $f(n) \leq k \leq 2^{f(n)}$.

Relaciones entre clases

Sea $f(n)$ una función de complejidad correcta. Naturalmente $\mathbf{SPACE}(f(n)) \subseteq \mathbf{NSPACE}(f(n))$ y $\mathbf{TIME}(f(n)) \subseteq \mathbf{NTIME}(f(n))$ por el hecho simple que una máquina Turing determinista es técnicamente también una NTM, con "buenas adivinanzas". Un resultado menos obvio es el siguiente:

Teorema 9. $\mathbf{NTIME}(f(n)) \subseteq \mathbf{SPACE}(f(n))$.

La demostración se construye por simular todas las opciones: dado un lenguaje $L \in \mathbf{NTIME}(f(n))$, existe una máquina Turing precisa no determinista M que decide L en tiempo $f(n)$. Sea d el grado de no determinismo de M; cada computación de M es una sucesión de selecciones no deterministas. El largo de la sucesión es $f(n)$ y cada elemento se puede representar por un entero $\in [0, d-1]$.

Construyamos una máquina M' para simular a M, considerando cada sucesión de selecciones posibles a su turno. Como M' simula las acciones de M, si con una sucesión M para con "sí", M' también para con "sí". Si ninguna sucesión acepta, M' rechaza su entrada.

Aunque el número de simulaciones es exponencial, el *espacio* que necesitamos es solamente $f(n)$ como lo hacemos una por una. El estado y todo lo anotado durante la simulación de una sucesión puede ser borrado cuando empieza la simulación de la sucesión siguiente. Además, como $f(n)$ es una función correcta de complejidad, la primera secuencia se puede generar en espacio $f(n)$.

La demostración del teorema siguiente se encuentra en el libro de texto de Papadimitriou [15]:

Teorema 10. $\mathbf{NSPACE}(f(n)) \subseteq \mathbf{TIME}\left(c^{\log n + f(n)}\right)$.

Una consecuencia del teorema es que

$$\mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} \subseteq \mathbf{EXP}. \tag{5.11}$$

La pregunta interesante con ese corolario es donde aplica $A \subset B$ en vez de $A \subseteq B$, es decir, entre que clases podemos probar que existe una diferencia: algunos elementos pertenecen a una pero no a la otra. Con los resultados ya establecidos, tenemos $\mathbf{L} \subset \mathbf{PSPACE}$ por

$$\begin{aligned} \mathbf{L} &= \mathbf{SPACE}(\log(n)) \subset \mathbf{SPACE}(\log(n)\log(\log(n))) \\ &\subseteq \mathbf{SPACE}(n^2) \subseteq \mathbf{PSPACE}. \end{aligned} \tag{5.12}$$

Es comúnmente aceptado que las inclusiones inmediatas en ecuación 5.11 sean todos de tipo $A \subset B$, pero todavía no se ha establecido tal resultado. Lo que sí se ha establecido

que por lo menos una de las inclusiones entre $\mathbf{L}$ y $\mathbf{PSPACE}$ es de ese tipo, y también que por lo menos una entre $\mathbf{P}$ y $\mathbf{EXP}$ es de ese tipo. El problema es que no se sabe cuál.

Espacio y no determinismo

De los resultados de la sección anterior tenemos que

$$\mathbf{NSPACE}\left(f(n)\right) \subseteq \mathbf{TIME}\left(c^{\log n + f(n)}\right) \subseteq \mathbf{SPACE}\left(c^{\log n + f(n)}\right), \tag{5.13}$$

pero existen resultados aún más estrictas: máquinas no deterministas con límites de espacio se puede simular con máquinas deterministas con espacio cuadrático. El resultado se basa en Reachability.

Teorema 11 (Teorema de Savitch)**.** Reachability$\in \mathbf{SPACE}\left(\log^2 n\right)$.

La demostración del teorema está en el libro de texto de Papadimitriou [15]. Se llega a probar que para toda función correcta $f(n) \geq \log n$, aplica que

$$\mathbf{NSPACE}\left(f(n)\right) \subseteq \mathbf{SPACE}\left(\left(f(n)\right)^2\right). \tag{5.14}$$

Además, se demostración que $\mathbf{PSPACE} = \mathbf{NPSPACE}$. Entonces máquinas no deterministas con respeto a espacio son menos poderosos que las máquinas no deterministas con respeto a tiempo; todavía no se sabe si **P** $=$ **NP**o no.

5.2. Reducciones

Una clase de complejidad es una colección infinita de lenguajes. Por ejemplo, la clase **NP** contiene problemas como Sat, Hornsat, Reachability, Clique, etcétera. No todas las problemas en la misma clase parecen igualmente difíciles. Un método para establecer un ordenamiento de problemas por dificultad es la construcción de *reducciones*: un problema A es por lo menos tan difícil como otro problema B si existe una *reducción del problema B al problema A*.

Un problema B *reduce* al problema A si existe una transformación R que para toda entrada x del problema B produce una entrada *equivalente* y de A tal que $y = R(x)$. La entrada y de A es equivalente a la entrada x de B si la respuesta ("sí" o "no") al problema A con la entrada y es la misma que la del problema B con la entrada x,

$$x \in B \text{ si y sólo si } R(x) \in A. \tag{5.15}$$

Entonces, para resolver el problema B con la entrada x, habrá que construir $R(x)$ y resolver el problema A para descubrir la respuesta que aplica para los dos problemas con

sus entradas respectivas. Es decir, si contamos con un algoritmo para el problema A y un algoritmo para construir $R(x)$, la combinación nos da un algoritmo para el problema B.

Entonces, si $R(x)$ es fácilmente construida, parece razonable que A es por lo menos tan difícil como B. Para poder establecer resultados formales, hay que clasificar las reducciones según los recursos computacionales utilizados por sus transformaciones. Las *reducciones Cook* permiten que $R(x)$ sea computada por una máquina Turing polinomial, mientras una *reducción Karp* es una reducción con una función de transformación de tiempo polinomial y las *reducciones de espacio logarítmico* son la clase de reducciones que utilizamos en este curso: un lenguaje L es *reducible* a un lenguaje L', denotado por $L \leq_{\mathrm{L}} L'$, si y sólo si existe una función R de cadenas a cadenas que está computada por una máquina Turing determinista en espacio $\mathcal{O}(\log n)$ tal que para toda entrada x

$$x \in L \text{ si y sólo si } R(x) \in L'. \tag{5.16}$$

La función R se llama una *reducción de L a L'.*

Teorema 12. *Si R es una reducción computada por una máquina Turing determinista M, para toda entrada x, M para después de un número polinomial de pasos.*

La demostración sigue las ideas siguientes: denota el largo de la entrada x con $|x| = n$. Como M utiliza $\mathcal{O}(\log n)$ espacio, el número de configuraciones posibles de M es $\mathcal{O}\left(nc^{\log n}\right)$. El hecho que M es determinista y para con toda entrada, no es posible que se repita ninguna configuración. Entonces, M para después de

$$c'nc^{\log n} = c'nn^{\log c} = \mathcal{O}\left(n^k\right) \tag{5.17}$$

pasos de computación con algún valor de k. Además, como la cinta de salida de M, que al final contiene $R(x)$, está computada en tiempo polinomial, el largo de $R(x)$, $|R(x)|$, es necesariamente polinomial en $n = |x|$.

Ejemplos de reducciones

Si queremos mostrar que un problema A no tenga solución, tenemos que mostrar que si existe un algoritmo para A, necesariamente tiene que existir un algoritmo para HALTING. La técnica para mostrar tal cosa es por construir una *reducción* del problema HALTING al problema A. Una reducción de B a A es una transformación de la entrada I_B del problema B a una entrada $I_A = t(I_B)$ para A tal que

$$I_B \in B \Leftrightarrow I_A \in A \tag{5.18}$$

y que exista una máquina Turing T tal que $t(I_B) = T(I_B)$.

Entonces, para mostrar que A no tiene solución, se transforma la entrada $M_b; x$ de HALTING a una entrada $t(M_b; x)$ de A tal que $M_b; x \in H$ si y sólo si $t(M_b; x) \in A$. Algunos ejemplos de lenguajes no recursivos son

(I) $A = \{M_b \mid M$ para con cada entrada$\}$

(II) $B = \{M_b; x \mid \exists y : M(x) = y\}$

(III) $C = \{M_b; x \mid$ para computar $M(x)$ se usa todos los estados de $M\}$

(IV) $D = \{M_b; x; y \mid M(x) = y\}$

Para el primero, la idea de la reducción es la siguiente: dada la entrada $M_b; x$, se construye la siguiente máquina M'

$$M'(y) = \begin{cases} M(x), & \text{si } x = y, \\ \text{“alto”} & \text{en otro caso.} \end{cases} \tag{5.19}$$

para la cual aplica que

$$M_b; x \in H \Leftrightarrow M \text{ para con } x \Leftrightarrow M' \text{ para con toda entrada } \Leftrightarrow M' \in A. \tag{5.20}$$

Ahora veremos ejemplos de reducciones entre dos problemas: de un problema B a otro problema A. En cada caso, se presenta una reducción R del lenguaje L_B del problema B al lenguaje L_A del problema A tal que para cada cadena x del alfabeto de B

(I) $x \in L_B$ si y sólo si $R(x) \in L_A$ y

(II) se puede computar $R(x)$ en espacio $\mathcal{O}(\log n)$.

Como un ejemplo, hacemos ahora una reducción de Hamilton Path a Sat. Para mostrar que Sat es por lo menos tan difícil que Hamilton Path, hay que establecer una reducción R de Hamilton Path a Sat: para un grafo $G = (V, E)$, el resultado $R(G)$ es una conjunción de cláusulas (o sea, una expresión booleana en Cnf) tal que G contiene un camino de Hamilton si y sólo si $R(G)$ es satisfactible.

La construcción es la siguiente y no es nada obvia sin conocer los trucos típicos del diseño de reducciones. Etiquetamos los n vértices de $G = (V, E)$ con los enteros: $V = \{1, 2, \ldots, n\}$. Representamos cada *par de vértices* con una variable booleana x_{ij} donde $i \in V$ y $j \in [1, n]$.

Asignamos a la variable x_{ij} el valor $\top$ si y sólo si el vértice número j en el camino C construido en Hamilton Path es el vértice i. Entonces, son en total n^2 variables, como el largo del camino es necesariamente n. Las cláusulas necesarias son las siguientes:

(I) En cada posición del camino C hay algún vértice:

$$\forall j \in V : x_{1j} \vee \cdots \vee x_{nj}. \tag{5.21}$$

(II) Ningún par de vértices puede ocupar la misma posición en el camino C:

$$\forall i, k \in V, \forall j \in [1, n], i \neq k : \neg x_{ij} \vee \neg x_{kj}. \tag{5.22}$$

(III) Cada vértice necesita estar incluido en el camino C:

$$\forall i \in V : x_{i1} \vee \cdots \vee x_{in}. \tag{5.23}$$

(IV) Cada vértice solamente puede estar incluida una sola vez en el camino C:

$$\forall i \in V, \forall j, k \in [1, n], j \neq k : \neg x_{ij} \vee \neg x_{ik}. \tag{5.24}$$

(V) El camino tiene que seguir las aristas del grafo: un vértice solamente puede seguir otro vértice en el camino C si esos vértices están conectados por una arista en $G = (V, E)$:

$$\forall (i, j) \notin E, \forall k \in [1, n-1] : \neg x_{ki} \vee \neg x_{(k+1)j}. \tag{5.25}$$

Ahora hay que mostrar que si $R(G)$ tiene una asignación T que satisface a $R(G)$, este corresponde al camino C del problema HAMILTON PATH:

- Por las cláusulas III y IV existe un sólo vértice i tal que $T(x_{ij}) = \top$.
- Por las cláusulas I y II existe una sola posición j tal que $T(x_{ij}) = \top$.
- Entonces, T representa una a permutación $\pi(1), \ldots, \pi(n)$ de los vértices tal que $\pi(i) = j$ si y sólo si $T(x_{ij}) = \top$.
- Por las cláusulas V, aplica que $\{\pi(k), \pi(k+1)\} \in E$ para todo $k \in [1, n-1]$.
- Entonces, las aristas que corresponden a la secuencia $(\pi(1), \ldots, \pi(n))$ de visitas a los vértices es un camino de Hamilton.

También hay que establecer que si $(\pi(1), \ldots, \pi(n))$ es una secuencia de visitas a los vértices del grafo $G = (V, E)$ que corresponde a un camino de Hamilton, definido por la permutación π, necesariamente está satisfecha la expresión $R(G)$ por una asignación T tal que

$$T(x_{ij}) = \begin{cases} \top & \text{si } \pi(i) = j, \\ \bot & \text{en otro caso.} \end{cases} \tag{5.26}$$

Además, queda mostrar que la computación de $R(G)$ ocupa $\mathcal{O}(\log n)$ espacio. Dada G como la entrada de una máquina Turing M, M construye a $R(G)$ de la manera siguiente: primero, se imprime las cláusulas de las primeras cuatro clases uno por uno a través de

tres contadores i, j y k. La representación binaria de cada contador con el rango $[1, n]$ es posible en $\log n$ espacio. Está parte no depende de la estructura del grafo, solamente de su número de vértices.

Además, M genera las cláusulas V por considerar cada par (i, j) en su turno: M verifica si $(i, j) \in E$, y si no lo es, se añade para todo $k \in [1, n-1]$ la cláusula $\neg x_{ki} \vee \neg x_{(k+1)j}$. Aquí espacio adicional aparte de la entrada misma es solamente necesario para los contadores i, j y k.

En total, el espacio ocupado simultaneamente es al máximo $3 \log n$, por lo cual la computación de $R(G)$ es posible en espacio $\mathcal{O}(\log n)$.

Como otro ejemplo, hacemos una reducción de REACHABILITY a CIRCUITVALUE: Para un grafo G, el resultado $R(G)$ es un circuito tal que la salida del circuito $R(G)$ es $\top$ si y sólo si existe un camino del vértice 1 al vértice n en $G = (V, E)$. Las puertas de $R(G)$ son de dos formas:

(I) g_{ijk} donde $1 \leq i, j \leq n$ y $0 \leq k \leq n$ y

(II) h_{ijk} donde $1 \leq i, j, k \leq n$.

La puerta g_{ijk} debería funcionar tal que tenga el valor $\top$ si y sólo si existe un camino en G del vértice i al vértice j sin pasar por ningún vértice con etiqueta mayor a k. La puerta h_{ijk} debería funcionar tal que tenga el valor $\top$ si y sólo si existe un camino en G del vértice i al vértice j sin usar ningún vértice intermedio mayor a k pero sí utilizando k. La estructura del circuito $R(G)$ es tal que

- Para $k = 0$, la puerta g_{ijk} es una entrada en $R(G)$.
- La puerta g_{ij0} es una puerta tipo $\top$ si $i = j$ o $\{i, j\} \in E$ y en otro caso una puerta tipo $\bot$.
- Para $k = 1, 2, \ldots, n$, las conexiones entre las puertas en $R(G)$ son las siguientes:
  - Cada h_{ijk} es una puerta tipo $\wedge$ con dos entradas: la salida de $g_{ik(k-1)}$ y la salida de $g_{kj(k-1)}$.
  - Cada g_{ijk} es una puerta tipo $\vee$ con dos entradas: la salida de $g_{ij(k-1)}$ y la salida de h_{ijk}.
- La puerta g_{1nn} es la salida del circuito $R(G)$.

El circuito $R(G)$ es no cíclico y libre de variables. Llegamos a una asignación correcta de valores a h_{ijk} y g_{ijk} por inducción en $k = 0, 1, \ldots, n$. El caso básico $k = 0$ aplica por definición. Para $k > 0$, el circuito asigna

$$h_{ijk} = g_{ik(k-1)} \wedge g_{kj(k-1)}. \tag{5.27}$$

Nuestro hipótesis de inducción es que h_{ijk} sea $\top$ si y sólo si haya un camino del vértice i al vértice k y además un camino del vértice k al vértice j sin usar vértices intermedios con etiquetas mayores a $k-1$, lo que aplica si y sólo si haya un camino del vértice i al vértice j que no utiliza ningún vértice intermedio mayor a k pero sí pasa por el mismo vértice k.

Para $k > 0$, el circuito asigna por definición $g_{ijk} = g_{ij(k-1)} \vee h_{ijk}$. Por el hipótesis, g_{ijk} es $\top$ si y sólo si existe un camino del vértice i al vértice j sin usar ningún vértice con etiqueta mayor a $k-1$ o si existe un camino entre los vértices que no utilice ningún vértice con etiqueta mayor a k pero pasando por k mismo. Entonces, tiene el valor $\top$ solamente en el caso que existe una camino del vértice i al vértice j sin pasar por ningún vértice con etiqueta mayor a k.

Para comprobar que es correcta la reducción, hacemos el análisis siguiente, por cuál resulta que el circuito $R(G)$ de hecho implementa el algoritmo Floyd-Warshall para el problema de alcance. La salida del circuito $R(G)$ es $\top$ si y sólo si g_{1nn} es $\top$, lo que aplica si y sólo si existe un camino de 1 a n en G sin vértices intermedios con etiquetas mayores a n (que ni siquiera existen) lo que implica que existe un camino de 1 a n en G. Dado dos vértices $v, u \in V$ para el problema REACHABILITY, lo único es etiquetar $v = 1$ y $u = n$ y la reducción está completa.

Entonces, lo único que necesitamos son tres contadores para computar el circuito $R(G)$: se puede hacer esto en $\mathcal{O}(\log n)$ espacio. Una observación importante es que $R(G)$ es un circuito *monótono*: no contiene ninguna puerta tipo $\neg$.

Cada circuito C sin variables de puede convertir a un circuito monótono: la idea es utilizar las leyes De Morgan

$$\begin{aligned} \neg(a \wedge b) &\Leftrightarrow (\neg a) \vee (\neg b) \\ \neg(a \vee b) &\Leftrightarrow (\neg a) \wedge (\neg b) \end{aligned} \tag{5.28}$$

para "empujar" todas las negaciones hasta las puertas de entrada. Como cada puerta de entrada es de tipo $\top$ o de tipo $\bot$ en la ausencia de variables, aplicamos $\neg\top = \bot$ y $\neg\bot = \top$ y ya no hay puertas de negación.

Un circuito monótono solamente puede computar *funciones booleanas monótonas*: si el valor de una se sus entradas cambia de $\bot$ a $\top$, el valor de la función *no puede cambiar* de $\top$ a $\bot$.

Continuamos con los ejemplos de reducciones con una reducción de CIRCUITSAT a SAT: Dado un circuito booleano C, habrá que construir una expresión booleana $R(C)$ en CNF tal que C es satisfactible si y soló si $R(C)$ lo es.

La expresión $R(C)$ usará todas las variables x_i de C y incorpora una variable adicional y_j para cada puerta de C. Las cláusulas son las siguientes:

(I) Si la puerta número j es una puerta tipo variable con la variable x_i, las dos variables y_j y x_i en $R(C)$ tienen que tener el mismo valor, $y_j \leftrightarrow x_i$; en CNF contendrá las dos cláusulas

$$(y_j \vee \neg x_i) \text{ y } (\neg y_j \vee x_i). \tag{5.29}$$

(II) Si la puerta j es una puerta tipo $\top$, habrá que poner la cláusula (y_j).

(III) Si la puerta j es una puerta tipo $\bot$, habrá que poner la cláusula $(\neg y_j)$.

(IV) Si la puerta j es una puerta tipo $\neg$ y su puerta de entrada es la puerta h, habrá que asegurar que y_j tiene el valor $\top$ si y sólo si y_h tiene el valor $\bot$, $y_j \leftrightarrow \neg y_h$; en CNF contendrá las dos cláusulas

$$(\neg y_j \vee \neg y_h) \text{ y } (y_j \vee y_h). \tag{5.30}$$

(V) Si la puerta j es una puerta tipo $\wedge$ con las puertas de entrada h y k, habrá que asegurar que $y_j \leftrightarrow (y_h \wedge y_k)$; en CNF contendrá las tres cláusulas

$$(\neg y_j \vee y_h), (\neg y_j \vee y_k) \text{ y } (y_j \vee \neg y_h \vee \neg y_k). \tag{5.31}$$

(VI) Si la puerta j es una puerta tipo $\vee$ con las puertas de entrada h y k, habrá que asegurar que $y_j \leftrightarrow (y_h \vee y_k)$; en CNF contendrá las tres cláusulas

$$(\neg y_j \vee y_h \vee y_k), (y_j \vee \neg y_k) \text{ y } (y_j \vee \neg y_h). \tag{5.32}$$

(VII) Si la puerta j es una puerta de salida, habrá que poner la cláusula (y_j).

Es relativamente directa la demostración de que sea correcto.

Nuestra última reducción de la sección es de CIRCUITVALUE a CIRCUITSAT; de hecho, CIRCUITVALUE es un caso especial de CIRCUITSAT. Todas las entradas de CIRCUITVALUE son también entradas validas de CIRCUITSAT. Además, para las instancias de CIRCUITVALUE, las soluciones de CIRCUITVALUE y CIRCUITSAT coinciden. Entonces, CIRCUITSAT es una generalización de CIRCUITVALUE. La reducción es trivial: la función de identidad $I(x) = x$ de CIRCUITVALUE a CIRCUITSAT.

Reducciones compuestas

En la sección anterior, establecemos una sucesión de reducciones,

$$\textsc{Reachability} \leq_{\mathrm{L}} \textsc{Circuitvalue} \leq_{\mathrm{L}} \textsc{Circuitsat} \leq_{\mathrm{L}} \textsc{Sat}. \tag{5.33}$$

Una pregunta interesante es si se puede componer reducciones, o sea, si la relación $\leq_{\mathrm{L}}$ es transitiva. Por ejemplo, aplica que $\textsc{Reachability} \leq_{\mathrm{L}} \textsc{Sat}$?

Teorema 13. *Si R es una reducción del lenguaje L a lenguaje L' y R' es una reducción del lenguaje L' al lenguaje L'', la composición $R \cdot R'$ es una reducción de L a L''.*

Por definición de reducciones, aplica que

$$x \in L \Leftrightarrow R(x) \in L' \Leftrightarrow R'(R(x)) \in L''. \tag{5.34}$$

Lo que hay que mostrar es que es posible computar $R'(R(x))$ en espacio $\mathcal{O}(\log n)$, $|x| = n$.

Tomamos las dos máquinas Turing M_R y $M_{R'}$. Construyamos una máquina Turing M para la composición $R \cdot R'$. Necesitamos cuidar *no guardar* el resultado intermedio de M_R porque eso puede ser más largo que $\log n$, por lo cual no se puede guardar sin romper el requisito de espacio logarítmico.

La máquina Turing M necesita simular a $M_{R'}$ con la entrada $R(x)$ por recordar la *posición* del cursor i en la cinta de entrada de $M_{R'}$. La cinta de entrada de $M_{R'}$ es la cinta de salida de M_R. Guardando solamente el índice i en binaria y el símbolo actualmente leído, pero no toda la cinta, podemos asegurar usar no más que $\mathcal{O}(\log n)$ espacio.

Inicialmente $i = 1$ y por definición de las máquinas Turing, el primer símbolo escaneado es $\triangleright$.

Cuando $M_{R'}$ mueve *a la derecha*, M corre a M_R para generar el símbolo de salida de esa posición y incrementa $i := i + 1$.

Si $M_{R'}$ mueve *a la izquierda*, M asigna $i := i - 1$ y vuelve a correr a M_R con x *desde el comienzo*, contando los símbolos de salida y parando al generar la posición i de la salida.

El espacio necesario para tal seguimiento de la salida de M_R con la entrada x es $\mathcal{O}(\log n)$: estamos guardando un índice binario de una cadena del largo $|R(x)| = \mathcal{O}\left(n^k\right)$.

Como las ambas máquinas M_R (con la entrada x) y $M_{R'}$ (con la entrada $R(x)$) operan en espacio $\mathcal{O}(\log n)$, así también M necesita $\mathcal{O}(\log n)$ espacio, $|x| = n$.

5.3. Problemas completos

La relación transitiva y reflexiva de reducciones $\leq_{\mathrm{L}}$ nos ofrece un orden de problemas. De interés especial son los problemas que son elementos maximales de tal orden de complejidad.

Sea C una clase de complejidad computacional y $L \in \mathrm{C}$ un lenguaje. El lenguaje L es C-*completo* si para todo $L' \in \mathrm{C}$ aplica que $L' \leq_{\mathrm{L}} L$.

En palabras: un lenguaje L es *completo* en su clase C si cada otro lenguaje $L' \in C$ se puede reducir a L.

Teorema 14. HALTING *es completo para lenguajes recursivamente numerables.*

La demostración es fácil: sea M una máquina Turing que acepta L. Entonces

$$x \in L \Leftrightarrow M(x) = \text{“sí”} \Leftrightarrow M_L(x) \neq \nearrow \Leftrightarrow M_L; x \in H. \tag{5.35}$$

Un lenguaje L es C-*duro* (también se dice: C-difícil) si se puede reducir cada lenguaje $L' \in \mathrm{C}$ a L, pero no se sabe si es válido que $L \in \mathrm{C}$.

Todas las clases principales como **P**, **NP**, **PSPACE** y **NL**tienen problemas completos naturales. Se utiliza los problemas completos para caracterizar una clase de complejidad. Se puede aprovechar los resultados de completitud para derivar *resultados de complejidad negativos*: un problema completo $L \in \mathrm{C}$ es el *menos probable* de todos los problemas de su clase para pertenecer a una clase “más débil” $\mathrm{C}' \subseteq \mathrm{C}$.

Si resulta que $L \in \mathrm{C}'$, aplica $\mathrm{C}' = \mathrm{C}$ si C' está *cerrada bajo reducciones*. Una clase C' está cerrada bajo reducciones si aplica que siempre cuando L es reducible a L' y $L' \in \mathrm{C}'$, también $L \in \mathrm{C}'$. Las clases **P**, **NP**, **coNP**, **L**, **NL**, **PSPACE** y **EXP** todas están cerradas bajo reducciones. Entonces, si uno pudiera mostrar que un problema **NP**-completo pertenece a la clase $\mathbf{P}$, la implicación sería que **P** = **NP**.

Entonces, habrá que establecer cuáles problemas son completos para cuáles clases. Lo más difícil es establecer el primero: hay que capturar la esencia del modo de cómputo y la cota del recurso de la clase en cuestión en la forma de un problema. El método de hacer esto se llama el *método de tabla*. En esta sección lo aplicamos para las clases **P** y **NP**.

Sea $M = (K, \Sigma, \delta, s)$ una máquina Turing de tiempo polinomial que computa un lenguaje L basado en Σ. La computación que ejecuta M con la entrada x se puede representar como si fuera una *tabla de cómputo* T de tamaño $|x|^k \times |x|^k$, donde $|x|^k$ es la cota de uso de tiempo de M.

Cada fila de la tabla representa un paso de computación: desde el primer paso 0 hasta el último $|x|^{k-1}$. Cada columna es una cadena del mismo largo tal que el elemento $T_{i,j}$ tiene el símbolo contenido en la posición j de la cinta de M después de i pasos de computación de M. Para continuar, hay que hacer algunas suposiciones sobre M:

- M cuenta con una sola cinta.
- M para con toda entrada x después de un máximo de $|x|^k - 2$ pasos de cómputo, con el valor de k elegido tal que se garantiza esto para cualquier $|x| \geq 2$.
- Las cadenas en la tabla todos tienen el mismo largo $|x|^k$, con cada posición que falta un símbolo llenada con un $\sqcup$.
- Si en el paso i, el estado de M es q y el cursor está escaneando el símbolo j que es σ, el elemento $T_{i,j}$ no es σ, sino σ_q, salvo que para los estados "sí" y "no", por los cuales el elemento es "sí" o "no" respectivamente.
- Al comienzo el cursor no está en $\triangleright$, sino en el primer símbolo de la entrada.
- El cursor nunca visita al símbolo $\triangleright$ extremo al la izquierda de la cinta; está se logra por acoplar dos movimientos si M visitaría ese $\triangleright$.
- El primer símbolo de cada fila es siempre $\triangleright$, nunca $\triangleright_q$.
- Si M para antes de llegar a $|x|^k$ pasos, o sea, $T_{i,j} \in$ "sí", "no" para algún $i < |x|^k - 1$ y j, todas las filas después serán idénticas.

Se dice que la tabla T es *aceptante* si y sólo si $T_{|x|^k-1,j} =$ "sí" para algún j. Además, M acepta a su entrada x si y sólo si la tabla de computación de M con x es aceptante.

Ahora se aplica el método para establecer que cualquier computación determinista de tiempo polinomial se puede capturar en el problema de determinación del valor de un circuito booleano:

Teorema 15. CIRCUITVALUE *es* **P***-completo.*

Ya se ha establecido que CIRCUITVALUE $\in \mathbf{P}$. Para establecer que sea completo, basta con demostrar que para todo lenguaje $L \in \mathbf{P}$, existe una reducción R de L a CIRCUITVALUE. Para la entrada x, el resultado $R(x)$ será un circuito libre de variables tal que $x \in L$ si y sólo si el valor de $R(x)$ es $\top$.

Considera una máquina Turing M que decide a L en tiempo n^k y su tabla de computación T. Cuando $i = 0$ o $j = 0$ o $j = |x|^k - 1$, conocemos el valor de $T_{i,j}$ a priori. Todos los otros elementos de T dependen *únicamente* de los valores de las posiciones $T_{i-1,j-i}$, $T_{i-1,j}$ y $T_{i-1,j+1}$. La idea es codificar este relación de las celdas de la tabla en un circuito booleano. Primero se codifica T en representación binaria. Después se utiliza el hecho que cada función booleana se puede representar como un circuito booleano. Se llega a un circuito C que depende de M y no de x. Este circuito se copia $(|x|^k-1)\times(|x|^k-2)$ veces, un circuito por cada entrada de la tabla T que no está en la primera fila o las columnas extremas. Habrá que establecer que sea correcta la reducción y que la construcción de

$R(x)$ es posible en espacio logarítmico. Los detalles de la demostración están en el libro de texto de Papadimitriou [15].

Como consecuencias del resultado, se puede establecer que el problema de valores de circuitos monótonos es también **P**-completo y también que HORNSAT es **P**-completo.

También resulta que computación *no determinista* en tiempo polinomial se puede capturar en el problema de satisfiabilidad de circuitos:

Teorema 16. CIRCUITSAT *es* **NP***-completo.*

Ya se ha establecido que CIRCUITSAT $\in$ **NP**. Sea L un lenguaje en la clase **NP**. Se procede a definir una reducción R que para cada cadena x construye un circuito booleano $R(x)$ tal que $x \in L$ si y sólo si $R(x)$ es satisfactible. Sea M una máquina Turing no determinista con una sola cinta que decide a L en tiempo n^k.

Se comienza por estandardizar las selecciones hechas por M: se supone que M tiene exactamente dos alternativas $\delta_1, \delta_2 \in \Delta$ en cada paso de computación. Si en realidad hubieran más, se puede añadir estados adicionales para dividir las opciones en un árbol binario. Si en realidad hubiera un paso determinista, se asigna $\delta_1 = \delta_2$.

Entonces, una secuencia $\mathbf{c}$ de elecciones no deterministas se puede representar en forma binaria donde $\delta_1 = 0$ y $\delta_2 = 1$:

$$\mathbf{c} = (c_0, c_1, \ldots, c_{|x|^k-2}) \in \{0,1\}^{|x|^k-1}. \tag{5.36}$$

Si fijamos la secuencia elegida $\mathbf{c}$, la computación hecha por M es efectivamente determinista. Definimos ahora la tabla de computación $T(M, x, \mathbf{c})$ que corresponde a la máquina Turing M, una entrada x y una secuencia fija de elecciones $\mathbf{c}$.

En la codificación binaria de $T(M, x, \mathbf{c})$, la primera fila y las columnas extremas resultan fijas como en el caso anterior. Los otros elementos $T_{i,j}$ dependen de los elementos $T_{i-1,j-1}$, $T_{i-1,j}$ y $T_{i-1,j+1}$, y adicionalmente de la elección c_{i-1} hecha en el paso anterior.

Existe un circuito booleano C con $3m + 1$ entradas y m salidas que compute la codificación binaria de $T_{i,j}$ dado como entradas los cuatro datos $T_{i-1,j-1}$, $T_{i-1,j}$, $T_{i-1,j+1}$ y c_{i-1} en representación binaria.

El circuito $R(x)$ será como en el caso determinista, salvo que la parte para $\mathbf{c}$ que será incorporada. Como C tiene tamaño constante que no depende de $|x|$, el circuito $R(x)$ puede ser computada en espacio logarítmico. Además, el circuito $R(x)$ es satisfactible si y sólo si existe una secuencia de elecciones $\mathbf{c}$ tal que la tabla de computación es aceptante si y sólo si $x \in L$.

Entonces hemos llegado a probar es resultado. Una consecuencia de mucha importancia es el teorema de Cook:

Teorema 17 (Teorema de Cook)**.** SAT *es* **NP***-completo.*

La demostración, ahora que conocemos el resultado de CIRCUITSAT, es simple: sea $L \in$ **NP**. Entonces, existe una reducción de L a CIRCUITSAT, como CIRCUITSAT es **NP**-completo. Además, como CIRCUITSAT tiene una reducción a SAT, se puede reducir también de L a SAT por la composición de reducciones. Como SAT $\in$ **NP**, tenemos que SAT es **NP**-completo.

5.3.1. Problemas NP-completos

La clase **NP** contiene numerosos problemas de importancia páctica. Típicamente el problema tiene que ver con la construcción o existencia de un objeto matemático que satisface ciertas especificaciones. La versión de la construcción misma es un problema de *optimización*, donde la mejor construcción posible es la solución deseada. En la versión de *decisión*, que es la versión que pertenece a la clase **NP**, la pregunta es si existe por lo menos una configuración que cumpla con los requisitos del problema. El libro de Garey y Johnson [7] es la referencia clásica de los problemas **NP**-completos.

Se puede decir que la mayoría de los problemas interesantes de computación en el mundo real pertenecen a la clase **NP**. La aplicación "cotidiana" de la teoría de complejidad computacional es el estudio sobre las clases a cuales un dado problema pertenece: solamente a **NP** o también a **P**, o quizás a ninguna.

La herramienta básica para esta tarea es construir una demostración que el problema de interés sea **NP**-completo, porque este implicaría que el problema es entre los menos probables de pertenecer en la clase **P**. Note que si algún problema **NP**-completo perteneciera a **NP**, aplicaría **NP** $=$ **P**.

Si se sube a un nivel suficientemente general, muchos problemas resultan **NP**-completos. En muchos casos es importante identificar cuáles requisitos exactamente causan que el problema resulte **NP**-completo versus polinomial. Una técnica básica es estudiar el conjunto de instancias producidas por una reducción utilizada en la demostración de ser **NP**-completo para capturar otro problema **NP**-completo.

El proceso de diseñar una demostración de **NP**-completitud para un problema Q es en general el siguiente:

(I) Jugar con instancias pequeñas de Q para desarrollar unos "gadgets" u otros componentes básicos para usar en la demostración.

(II) Buscar por problemas que ya tengan demostración de ser **NP**-completos que podrían servir para la reducción necesaria.

(III) Construir un reducción R *de* un problema P conocido y **NP**-completo *al* problema Q.

(IV) Probar que con la entrada $R(x)$, si la solución de Q es "sí" que este implica que la solución de P con x también es siempre "sí".

(V) Probar que con la entrada x, si la solución de P es "sí" que este implica que la solución de Q con $R(x)$ también es "sí".

En la construcción típicamente se busca por representar las elecciones hechas, la consistencia y restricciones. Lo difícil es cómo expresar la naturaleza del problema P en términos de Q.

Cuando ya está establecido que un problema de interés es **NP**-completo, normalmente se dirige el esfuerzo de investigación a

- estudiar casos especiales,
- algoritmos de aproximación,
- análisis asintótica del caso promedio,
- algoritmos aleatorizados,
- algoritmos exponenciales para instancias pequeñas o
- métodos heurísticos de búsqueda local.

5.3.2. Caracterización por relaciones: certificados

Una relación $\mathcal{R} \subseteq \Sigma^* \times \Sigma^*$ se puede decidir en tiempo polinomial si y sólo si existe una máquina Turing determinista que decide el lenguaje $\{x; y \mid (x, y) \in \mathcal{R}\}$ en tiempo polinomial. Una relación $\mathcal{R}$ está *polinomialmente balanceada* si

$$\big((x, y) \in \mathcal{R}\big) \rightarrow (\exists k \geq 1 \text{ tal que } |y| \leq |x|^k). \tag{5.37}$$

Sea $L \subseteq \Sigma^*$ un lenguaje.

Aplica que $L \in$ **NP** si y sólo si existe una polinomialmente balanceada

relación $\mathcal{R}$ que se puede decidir en tiempo polinomial tal que

$$L = \{x \in \Sigma^* \mid (x, y) \in \mathcal{R} \text{ para algún } y \in \Sigma^*\}. \tag{5.38}$$

Para entender porqué está definición es válida, suponga que existe tal relación $\mathcal{R}$. Entonces L está decidida por una máquina Turing no determinista que con la entrada x "adivina" un valor y con largo máximo $|x|^k$ y utiliza la máquina para $\mathcal{R}$ para decidir en tiempo polinomial si aplica $(x, y) \in \mathcal{R}$.

Habrá que razonar la equivalencia lógica también en la otra dirección: suponga que $L \in$ **NP**. Esto implica que existe una máquina Turing no determinista N que decide a L en tiempo $|x|^k$ para algún valor de k.

Definimos la relación $\mathcal{R}$ en la manera siguiente: $(x, y) \in \mathcal{R}$ si y sólo si y es una codificación de una computación de N que acepta la entrada x. Esta $\mathcal{R}$ est polinomialmente balanceada, como cada computación de N tiene cota polinomial, y además se puede decidir $\mathcal{R}$ en tiempo polinomial, como se puede verificar en tiempo polinomial si o no y es una codificación de una computación que acepta a x en N. Como N decide a L, tenemos

$$L = \{x \mid (x, y) \in R \text{ para algún } y\}, \tag{5.39}$$

exactamente como queríamos.

Un problema A pertenece a la clase **NP** si cada instancia con la respuesta "sí" del problema A tiene por lo menos un *certificado conciso* y, también llamado un *testigo polinomial*.

En los problemas típicos que tratan de la existencia de un objeto matemático que cumple con ciertos criterios, el objeto mismo sirve como un certificado — típicamente el objeto tiene un tamaño no muy grande en comparación con la entrada. Por ejemplo, si se trata se encontrar un cierto subgrafo, el subgrafo tiene tamaño no mayor al tamaño del grafo de entrada. Los requisitos también suelen suficientemente simples para poder verificar en tiempo polinomial que un dado objeto realmente corresponde a los criterios definidos.

5.4. Complejidad de algunas variaciones de SAT

5.4.1. Problemas kSAT

El lenguaje kSAT, donde $k \geq 1$ es un entero, es el conjunto de expresiones booleanas $\phi \in$ SAT (en CNF) en las cuales *cada cláusula contiene exactamente k literales*.

Teorema 18. 3SAT *es* **NP***-completo.*

Claramente 3SAT $\in$ **NP** por ser un caso especial de SAT y sabemos que SAT $\in$ **NP**. Ya hemos mostrado que CIRCUITSAT es **NP**-completo y además una reducción de CIRCUITSAT a SAT está construida. Ahora vamos a considerar las cláusulas producidas en esa reducción: ¡todas tienen tres literales o menos! Como las cláusulas son disyunciones, podemos "aumentar" cada cláusula de uno o dos literales a tener tres simplemente por copiar literales. Entonces, tenemos una reducción de CIRCUITSAT a 3SAT.

A veces es posible aplicar transformaciones que eliminen algunas propiedades de un lenguaje **NP**-completo tal que el hecho que está **NP**-completo no está afectado.

Teorema 19. 3SAT *es* **NP***-completo aún cuando solamente se permite que cada variable aparezca por máximo tres veces en la expresión* $\phi \in 3\text{SAT}$ *y además que cada literal aparezca por máximo dos veces en* ϕ*.*

La demostración es una reducción donde cada instancia $\phi \in 3\text{SAT}$está transformada a eliminar las propiedades no permitidas. Considera una variable x que aparece $k > 3$ veces en ϕ. Introducimos variables nuevas $x_1, \ldots, x_k$ y reemplazamos la primera ocurrencia de x en ϕ por x_1, la segunda por x_2, etcétera. Para asegurar que las variables nuevas tengan todas el mismo valor, añademos las cláusulas siguientes en ϕ:

$$(\neg x_1 \vee x_2), (\neg x_2 \vee x_3), \ldots, (\neg x_k \vee x_1). \tag{5.40}$$

Denota por ϕ' la expresión que resulta cuando cada ocurrencia extra de una variable en ϕ ha sido procesada de esta manera. Resulta que ϕ' ya cumple con las propiedades deseadas y que ϕ es satisfactible si y sólo si ϕ es satisfactible.

Para problemas $k\text{SAT}$, la frontera entre problemas polinomiales y problemas **NP**-completos está entre 2SATy 3SAT. Para cada instancia ϕ de 2SAT, existe un algoritmo polinomial basado en el problema REACHABILITY en un grafo dirigido $G(\phi)$. Las variables x de ϕ y sus negaciones $\bar{x}$ forman el conjunto de vértices de $G(\phi)$. Una arista conecta vértice x_i al x_j si y sólo si existe una cláusula $\bar{x_i} \vee x_j$ en ϕ ya que esto es lo mismo que $((x_i \to x_j) \wedge (\bar{x_j} \to \bar{x_i}))$.

La demostración resulta del hecho que ϕ no es satisfactible si y sólo si existe una variable x tal que existen caminos de x a $\bar{x}$ y viceversa en $G(\phi)$ (o sea, un ciclo de x a su negación). Entonces, sabemos que 2SAT es polinomial. Además, resulta que $2\text{SAT} \in \mathbf{NL}$ (habrá que recordar que $\mathbf{NL} \subseteq \mathbf{P}$).

Como $\mathbf{NL}$ está cerrada bajo el complemento, podemos mostrar que 2SAT Complement $\in \mathbf{NL}$. La existencia del camino de "no satisfiabilidad" del resultado anterior puede ser verificada en espacio logarítmico por computación no determinista por "adivinar" una variable x y el camino entre x y $\neg x$.

El problema MAX2SAT es una generalización de 2SAT:

Problema 17 (MAX2SAT)**.** *Dada:* una expresión booleana ϕ en CNF con no más que dos literales por cláusula y un entero k. *Pregunta:* ¿existe una asignaciones de valores T que satisfecha por lo menos k cláusulas de ϕ?

Se puede probar que MAX2SAT es **NP**-completo.

5.4.2. SAT de "no todos iguales" (NAESAT)

El lenguaje $\text{NAESAT} \subset 3\text{SAT}$ contiene las expresiones booleanas ϕ para las cuales existe una asignación de valores T tal que en ninguna cláusula de ϕ, todas las literales tengan el mismo valor $\top$ o $\bot$.

Teorema 20. NAESAT *es* **NP**-*completo.*

Ya se ha establecido que CIRCUITSAT es **NP**-completo y se cuenta con una reducción R de CIRCUITSAT a SAT: para todo circuito C, tenemos que $C \in$ CIRCUITSAT si y sólo si $R(C) \in$SAT. Primero aumentamos todas las clausulas de la reducción a contener exactamente tres literales por añadir uno o dos duplicados de un literal z donde hace falta. Vamos a mostrar que para la expresión booleana $R_z(C)$ en CNF de tipo 3SAT aplica que $R_z(C) \in$ NAESAT$\Leftrightarrow C \in$CIRCUITSAT.

En la dirección ($\Rightarrow$), si una asignación de valores T satisface a la expresión original $R(C)$ en el sentido de NAESAT, también la asignación *complementaria* $\bar{T}$ lo satisface por la condición NAESAT: en una de las dos asignaciones asigna $\bot$ al literal z, por lo cual todas las cláusulas originales están satisfechas en esta asignación. Entonces, por la reducción de CIRCUITSAT a SAT, existe una asignación que satisface el circuito.

En la dirección ($\Leftarrow$), si C es satisfactible, existe una asignación de valores T que satisface a $R_z(C)$. Entonces extendemos T para $R_z(C)$ por asignar $T(z) = \bot$. En ninguna cláusula de $R_z(C)$ es permitido que todos los literales tengan el valor $\top$ (y tampoco que todos tengan el valor $\bot$). Cada cláusula que corresponde a una puerta tipo $\top$, $\bot$, $\neg$ o variable tenía originalmente dos literales o menos, por lo cual contienen z y $T(z) = \bot$. Sin embargo, están satisfechas por T, por lo cual algún literal es necesariamente asignado el valor $\top$ en T para cada cláusula.

Los únicos casos de preocupación son puertas de tipo $\wedge$ y $\vee$. En el caso de $\wedge$, las cláusulas tienen la forma

$$(\neg g \vee h \vee z), (\neg g \vee h' \vee z), (g \vee \neg h \vee \neg h'), \tag{5.41}$$

donde las primeras dos tienen z con su valor $\bot$. En la tercera, no todas las tres pueden tener el valor $\top$ porque así no es posible satisfacer a las dos primeras. El caso de $\vee$ es parecido.

5.5. Complejidad de problemas de grafos

En esta sección, consideramos solamente grafos no dirigidos.

Teorema 21. INDEPENDENT SET *es* **NP**-*completo.*

Para demostrar el teorema, necesitamos un "gadget": el triángulo. Si el grafo de entrada contiene un triángulo, es decir, una camarilla de tres vértices, solamente uno de los tres participantes del triángulo puede ser considerado para formar un conjunto independiente, porque en un conjunto independiente, ningún par de vértices puede compartir una arista.

Vamos a *restringir* la clase de grafos que permitimos: solamente consideramos grafos los vértices de cuales se puede dividir en *triángulos disjuntos*. Es decir, cada vértice del grafo

solamente puede tomar parte en un triángulo (por lo de disjunto) y cada vértice tiene que ser parte de un triángulo (por lo de haber sido dividido). Denotamos el número de tales triángulos por t.

Por construcción, obviamente en ese tipo de grafo ningún conjunto independiente puede tener cardinalidad mayor a t. Además, un conjunto independiente de cardinalidad t existe solamente si las otras aristas del grafo permiten elegir un vértice de cada triángulo sin que tengan aristas en común los vértices elegidos. Nuestra reducción R será de 3SAT a INDEPENDENT SET: para cada cláusula de una instancia ϕ de 3SAT, generamos un triángulo en el grafo $G = R(\phi)$. Sean a_i, b_i y c_i las tres literales de una cláusula C_i . Entonces, habrán vértices v_{ai}, v_{bi} y v_{ci} en el grafo G y además las tres aristas $\{v_{ai}, v_{bi}\}$, $\{v_{ai}, v_{ci}\}$ y $\{v_{bi}, v_{ci}\}$ que forman el triángulo de los tres vértices de la cláusula C_i.

Además, hay que conectar las cláusulas que comparten variables. Un vértice v_i que corresponde a un literal de la cláusula C_i y otro vértice v_j que corresponde a un literal de la cláusula C_j, donde $i \neq j$, están conectadas por una arista si y sólo si los literales a cuales corresponden los vértices v_i y v_j son de la misma variable, pero el literal es positivo en C_i y negativo en C_j. Como un ejemplo, se puede construir un grafo que corresponde a

$$\phi = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neq x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3). \tag{5.42}$$

Con el grafo G así construido y la cota k que es la cardinalidad del conjunto independiente en el problema INDEPENDENT SET siendo $k = t$, tenemos definida nuestra reducción.

Habrá que mostrar que es correcta la reducción: existe un conjunto independiente $I \subseteq V$ en el grafo $G = R(\phi)$ tal que $|I| = k$ y ϕ tiene k cláusulas si y sólo si ϕ es satisfactible.

Suponga que tal conjunto I existe. Como ϕ tiene k cláusulas y $|i| = k$, por construcción I necesariamente contiene un vértice de cada uno de los k triángulos. El conjunto I no puede contener ningún par de vértices que corresponda una ocurrencia positiva de un variable x y una ocurrencia negativa $\neg x$ de la misma variable. Entonces, ¡I define una asignación de valores T! Si un vértice v pertenece a I y v corresponde a una ocurrencia positiva de la variable x, aplica que $T(x) = \top$. Si el vértice $v \in I$ corresponde a una ocurrencia negativa $\neg x$, aplica que $T(x) = \bot$. Cada par de literales contradictorios está conectado en G por una arista, y entonces la asignación T así definida es consistente. Por último, como I contiene un vértice de cada triángulo, y cada triángulo es una cláusula, cada cláusula tiene exactamente un literal con el valor $\top$, porque necesariamente $T(x) = \top$ o $T(\neg x) = \bot$ que implica que $T(x) = \top$ para la variable x el vértice de cual está incluido en el conjunto independiente I.

En la otra dirección, utilizamos el mismo truco: si ϕ es satisfactible, dada la asignación T que la satisface, identificamos cuales literales tienen el valor $\top$ en T. Elegimos de cada cláusula un literal con el valor $\top$. Los vértices que corresponden a estos literales forman el conjunto I. Son uno por cláusula, y entonces $|I| = k$ si ϕ tiene k cláusulas. Además, por construcción, los vértices así elegidos no están conectados por ninguna arista, porque

solamente elegimos uno por triángulo y las aristas fuera de los triángulos están entre literales contradictorios.

Por el teorema 19, podemos hacer esta construcción solamente suponiendo que cada literal ocurre por máximo dos veces. En consecuencia, el grafo que resulta tiene grado máximo cuatro:

Teorema 22. INDEPENDENT SET *es* **NP***-completo para grafos de grado máximo cuatro.*

También aplica que INDEPENDENT SET es **NP**-completo para grafos planos, pero la demostración se deja como ejercicio.

Por las conexiones conocidas entre INDEPENDENT SET, CLIQUE y VERTEX COVER, llegamos con reducciones triviales a lo siguiente:

Teorema 23. CLIQUE *y* VERTEX COVER *son* **NP***-completos.*

5.5.1. El problema de flujo máximo

Dado un grafo dirigido con capacidades en las aristas y un flujo no-óptimo, se puede aumentar el flujo que cruza un corte desde el lado de s al lado de t o alternativamente por disminuir el flujo desde el lado de t al lado de s. Este nos ofrece un algoritmo para descubrir el flujo máximo. Para empezar, podemos elegir el flujo cero, donde el flujo por cada arista es cero — no rompe con ninguna restricción, por lo cual es un flujo factible, aunque no óptimo.

Para aumentar el flujo, buscamos un *camino aumentante* que en este caso es un camino C de s a t en el cual de puede viajar por las aristas según su dirección o *en contra*. Las aristas $\langle v, w\rangle$ incluidas serán tales que si se viaja en la dirección original, aplica que $f(v, w) < c(v, w)$, pero si se viaja en contra, $f(v, w) > 0$. Definimos una función auxiliar

$$\delta(v, w) = \begin{cases} c(v, w) - f(v, w), & \text{si } \langle v, w\rangle \in E, \\ f(v, w), & \text{si } \langle w, v\rangle \in E, \end{cases} \tag{5.43}$$

y sea $\delta = \min_C \delta(v, w)$. El flujo se aumenta por añadir δ en todos los flujos que van según la dirección de las aristas en el camino C y restar δ de todos los flujos que van en contra en C. Por la manera en que elegimos a δ, no rompemos ninguna restricción, ni de capacidades ni de balance. Este procedimiento se itera hasta que ya no existan caminos aumentantes. Cuando ya no existe camino aumentante ninguno, el flujo es maximal.

La eficiencia del método presentado depende de cómo se construye los caminos aumentantes. La mayor eficiencia se logra por elegir siempre el camino aumentante de largo

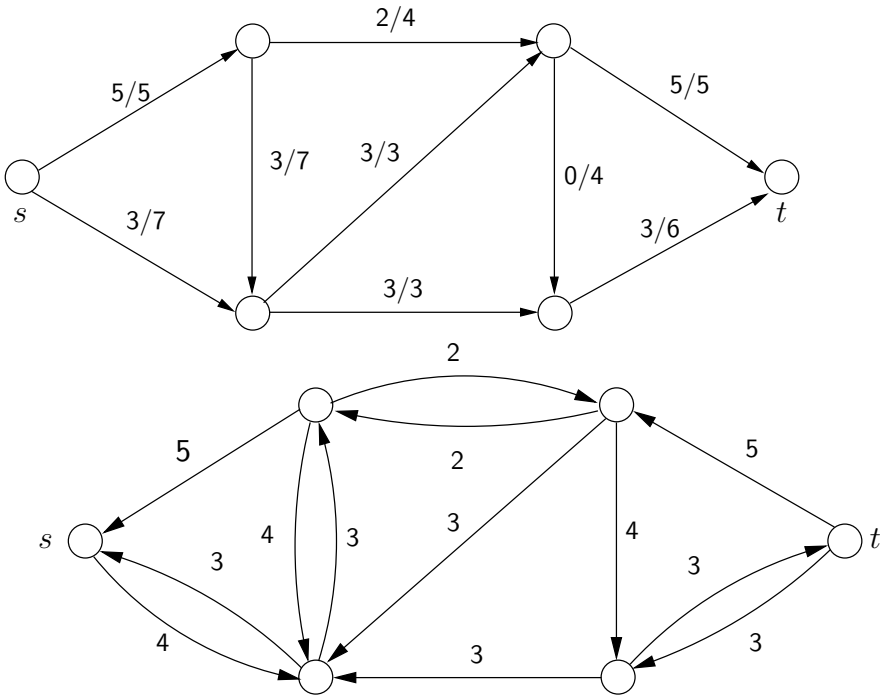

Figura 5.1: En ejemplo de un grafo con flujo (arriba) y su grafo residual resultante (abajo).

mínimo; el algoritmo que resulta es polinomial, $\mathcal{O}(nm^2) = \mathcal{O}(n^5)$. Es posible que hayan más de un camino de largo mínimo — aplicándolos todos al mismo paso resulta en un algoritmo de complejidad asintótica $\mathcal{O}(n^3)$.

Primero construyamos un *grafo residual* que captura las posibilidades de mejoramiento que tenemos. El grafo residual $G_f = (V, E_f)$ del grafo $G = (V, E)$ con respeto a un flujo f tiene

$$\left\{\{v, w\} \in E \mid \big((f(v, w) < c(v, w)) \vee (f(w, v) > 0)\big)\right\}. \tag{5.44}$$

La *capacidad de aumento* de la arista $\{v, w\} \in E_f$ se define como

$$c'(v, w) = \begin{cases} c(v, w) - f(v, w) & \text{si } \{v, w\} \in E, \\ f(w, v) & \text{si } \{w, v\} \in E. \end{cases} \tag{5.45}$$

Para un ejemplo de un grafo residual, ver la figura 5.1.

Cada camino simple entre s y t en el grafo residual G_f es un camino aumentante de G. El valor de δ es igual al capacidad de aumento mínimo del camino. Se podría resolver el problema de flujo máximo por empezar con el flujo cero, iterar la construcción de G_f y

la incorporación de un camino aumentante hasta que ya no haya caminos en G_f de s a t.

Sea A el conjunto de vértices que en el último grafo residual G_f se puede alcanzar desde s. Entonces, por construcción en el grafo original G, cada arista que cruza el corte $(A, V \setminus A)$ desde A al otro lado tiene flujo igual a su capacidad y cada arista que va en contra sobre el corte tiene flujo cero. Es decir, la capacidad del corte es igual al flujo actual.

Para elegir los caminos aumentantes más cortos en el grafo residual, utilizamos búsqueda en anchura desde s. En subgrafo formado por los caminos cortos en G_f se llama la *red de capas* (inglés: layered network) y la denotamos con G'_f. Su construcción es la siguiente: se asigna a cada vértice un valor de "capa" que es su distancia desde s. Solamente vértices con distancias finitas están incluidas. Una arista $\{v, w\}$ de G_f se incluye en G'_f solamente si el valor de capa de w es el valor de capa de v más uno. La complejidad asintótica de la construcción de G'_f es $\mathcal{O}(m)$.

En el grafo G'_f, cada camino de s a t tiene el mismo largo. El mejor aumento sería igual al flujo máximo en G'_f, pero en el peor caso es igual en complejidad al problema original. Entonces se construye una aproximación: se define el *flujo mayor* en G'_f como un flujo que ya no se puede aumentar con caminos que solamente utilizan aristas que "avanzan" hacía t.

Definimos como el *flujo posible* de un vértice es el mínimo de la suma de las capacidades de las aristas que entran y de la suma de las capacidades de las aristas que salen:

$$v_f = \text{mín} \left\{ \sum_{\{u,v\} \in G'_f} c'(u, v), \sum_{\{v,w\} \in G'_f} c'(v, w) \right\}. \tag{5.46}$$

Primero sacamos de G'_f todos los vértices con flujo posible cero y cada arista adyacente a estos vértices. Después identificamos el vértice v con flujo posible mínimo. "Empujamos" una cantidad de flujo igual al flujo posible de v desde v hacía t. Después "retiramos" flujo a v de sus aristas entrantes por construir caminos desde s a v hasta que se satisface la demanda de flujo que sale de v a t. Ahora actualizamos las capacidades de las aristas afectadas y memorizamos el flujo generado y el camino que toma.

Computamos de nuevo los flujos posibles y eliminamos de nuevo vértices con flujo posible cero juntos con sus aristas adyacentes. Si s y t quedaron fuera, el flujo construido es el flujo mayor en G'_f. Si todavía están, repetimos el proceso de elegir el mínimo y empujar y retirar flujo. Iteramos así, sumando los flujos generados hasta que s y t ya no estén conectados en G'_f.

La construcción de una red de capas G'_f toma tiempo $\mathcal{O}(n^2)$ La distancia entre s y t está en el peor caso $\mathcal{O}(n)$. Cada iteración de construcción de una red de capas G'_f utiliza caminos más largos que el anterior, por lo cual la construcción se repite $\mathcal{O}(n)$

veces. Las operaciones de empujar y retirar flujo son ambas de complejidad asintótica $\mathcal{O}(n)$ y de ejecutan en total $\mathcal{O}(n)$ veces. Entonces, el algoritmo del flujo mayor tiene complejidad asintótica $\mathcal{O}(n^3)$.

5.5.2. El problema de corte mínimo

El problema de corte mínimo es igual al problema del flujo máximo: se resuelve por fijar un vértice s cualquiera y después resolver el flujo máximo entre s y todos los otros vértices. El valor mínimo de los flujos máximos corresponde al corte mínimo del grafo entero. Entonces, como existen algoritmos polinomiales para el problema de flujo máximo, y solamente repetimos $n-1$ veces su ejecución, el problema del corte mínimo pertenece a la clase **P**.

Sin embargo, el problema de decisión siguiente es mucho más difícil:

Problema 18 (MaxCut)**.** *Dado:* un grafo $G=(V,E)$ no dirigido y no ponderado y un entero k. *Pregunta:* ¿existe un corte en G con capacidad igual o mayor a k?

Teorema 24. MaxCut *es* **NP***-completo.*

Se provee la demostración para *multigrafos* con una reducción desde Naesat. Note que un grafo simple es un caso especial de multigrafos.

Para una conjunción de cláusulas $\phi = C_1 \wedge \ldots \wedge C_r$, construimos un grafo $G=(V,E)$ tal que G tiene capacidad de corte $5r$ si y sólo si ϕ es satisfactible en el sentido de Naesat.

Los vértices del grafo serán los literales $x_1, \ldots, x_n, \neg x_1, \ldots, \neg x_n$ donde $x_1, \ldots, x_n$ son las variables de ϕ. Para cada cláusula $\alpha \vee \beta \vee \gamma$, incluimos las aristas del triángulo entre los vértices que representan a α, β y γ. Si la cláusula tiene solamente dos literales, ponemos dos aristas entre los vértices que corresponden; de hecho, mejor convertir cada cláusula a uno con tres literales por repetir un literal según necesidad. Entonces tenemos en total $3r$ ocurrencias de variables.

Además, incluyemos n_i copias de la arista $\{x_i, \neg x_i\}$ donde n_i es el número total de ocurrencias de los literales x_i y $\neg x_i$ en ϕ.

Para comprobar que sea correcta la construcción, suponga que existe un corte $(S, V \setminus S)$ con capacidad $5m$ o mayor.

Si un literal que corresponde al vértice v_i participa en n_i cláusulas, v_i tiene al máximo $2n_i$ aristas a otros vértices además de las n_i aristas a su negación. Se supone que una variable y su negación no aparecen en la misma clausula porque así la cláusula sería una tautología. Entonces, los dos vértices representando a x_i y $\neg x_i$ tienen en total al máximo $2n_i$ aristas a vértices que representan a otros literales.

Si estuvieran cada variable y su negación las dos en el mismo lado, su contribución a la capacidad del corte sera por máximo $2n_i$. Si cambiamos al otro lado el vértice con menos aristas "externas", la contribución al tamaño de corte del par no puede disminuir (las aristas n_i entre los dos vértices cruzarían el corte después del cambio). Entonces podemos asumir que caen en *lados distintos* del corte.

Sea S el conjunto de literales asignadas $\top$, por lo cual $V \setminus S$ contiene los literales que tienen asignado el valor $\bot$. Como ahora cada variable y su negación están el lados diferentes, contribuyen una arista por ocurrencia en ϕ, es decir, en total $3r$ aristas. Para lograr que sea mayor o igual a $5m$ la capacidad del corte, habrá que ser $2r$ aristas que son aristas de los triángulos representando a las cláusulas cruzando el corte. Entonces, cada triángulo tiene que estar separado: dos vértices en un lado y uno al otro. Así cada uno de los r triángulos necesariamente contribuye dos aristas. Además, como por lo menos un vértice de cada cláusula pertenece en el conjunto de literales asignados a $\top$ y por lo menos un vértice de cada cláusula pertenece al conjunto de literales asignados a $\bot$, la asignación satisface a ϕ en el sentido de NAESAT.

En la otra dirección es fácil: dada una asignación T a ϕ en el sentido de NAESAT, podemos agrupar los vértices a darnos un corte con capacidad mayor o igual a $5r$.

Antes definimos un problema de bisección máxima (MAX BISECTION). Una pregunta interestante es si MAX BISECTION es más fácil que MAXCUT?

Teorema 25. MAX BISECTION *es* **NP**-*completo.*

La reducción es de MAXCUT a MAX BISECTION por modificar la entrada: añadimos n vértices no conectados a G. Ahora cada corte de G se puede balancear a ser una bisección por organizar los vértices no conectados apropiadamente a los dos lados. Entonces, el grafo original tiene un corte $(S, V \setminus S)$ de tamaño k o mayor si y sólo si el grafo modificado tiene un corte de tamaño k o mayor con $|S| = |V \setminus S|$.

El problema de minimización correspondiente, o sea, MINCUT con el requisito de ser una bisección, también es **NP**-completo, aunque ya sabemos que MINCUT$\in \mathbf{P}$:

Problema 19 (MIN BISECTION)**.** *Dado:* un grafo no dirigido y un entero k. *Pregunta:* ¿existe una bisección con cardinalidad menor o igual a k?

Teorema 26. MIN BISECTION *es* **NP**-*completo.*

La demostración es una reducción de MAX BISECTION: la instancia es un grafo $G = (V, E)$ con un número par de vértices $n = 2c$. Ese grafo tiene una bisección de tamaño k o más si y sólo si el grafo complemento $\bar{G}$ tiene una bisección de tamaño $c^2 - k$ o menos.

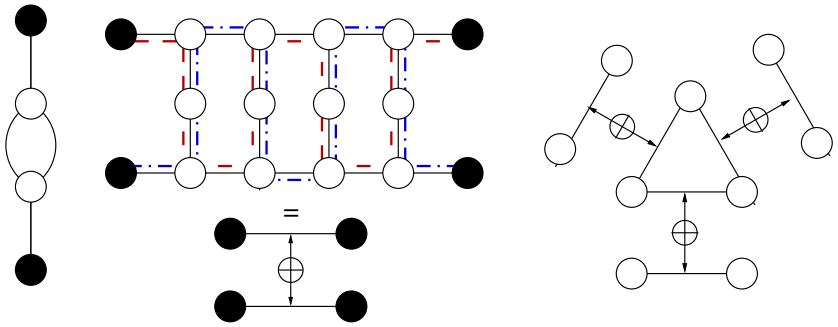

Figura 5.2: Los gadgets de la reducción de 3SAT a HAMILTON PATH: a la izquierda, el gadget de elección, en el centro, el gadget de consistencia y a la derecha, el gadget de restricción.

5.5.3. Caminos y ciclos de Hamilton

Teorema 27. HAMILTON PATH *es* **NP**-*completo.*

La reducción es de 3SAT a HAMILTON PATH: dada una expresión ϕ en CNF con las variables $x_1, \ldots, x_n$ y cláusulas $C_1, \ldots, C_m$ tal que cada cláusula contiene tres literales, un grafo $G(\phi)$ está construida tal que $G(\phi)$ contiene un camino de Hamilton si y sólo si ϕ es satisfactible. Se necesita tres tipos de gadgets (ver figura 5.2:

- gadgets de *elección* que eligen la asignación a las variables x_i,
- gadgets de *consistencia* que verifican que todas las ocurrencias de x_i tengan el mismo valor asignado y que todas las ocurrencias de $\neg x_i$ tengan el valor opuesto,
- gadgets de *restricción* que garantizan que cada cláusula sea satisfecha.

Los gadgets de elección de las variables se conecta en serie. Cada cláusula tiene un gadget de restricción. Las conexiones entre los gadgets de restricción y elección llevan un gadget de consistencia conectando los gadgets de restricción a la arista de "verdad" de x_i si el literal es positivo y a la arista de "falso" de x_i si el literal es negativo.

Adicionalmente se añade aristas para conectar todos los triángulos, el último vértice de la cadena de los gadgets de elección y un vértice adicional v tal que formen una camarilla estos $3n+2$ vértices. Otro vértice auxiliar w está añadido y conectado con una arista a v. La idea de la construcción es que un lado de un gadget de restricción está recorrida por el camino de Hamilton si y sólo si el literal a cual corresponde es falso. En consecuencia, por lo menos un literal de cada cláusula es verdad porque en el otro caso todo el triángulo será recorrido. El camino empezará en el primer vértice de la cadena de los gadgets de elección y termina en el vértice w. Se omite las detalles de verificar que esté funcional la reducción, dejándolas como un ejercicio.

Como una consecuencia, obtenemos un resultado sobre el problema del viajante:

Teorema 28. TSPD *es* **NP**-*completo.*

La reducción será de HAMILTON PATH a TSPD: dado un grafo $G = (V, E)$ con n vértices, hay que construir una matriz de distancias d_{ij} y decidir un presupuesto B tal que existe un ciclo de largo menor o igual a B si y sólo si G contiene un camino de Hamilton (nota: no un ciclo, sino un camino). Etiquetamos los vértices $V = \{1, 2, \ldots, n\}$ y asignamos simplemente

$$d_{ij} = \begin{cases} 1, & \text{si } \{i, j\} \in E, \\ 2, & \text{en otro caso.} \end{cases} \tag{5.47}$$

El presupuesto será $B = n + 1$. Note que así el grafo ponderado que es la instancia de TSPD es completo.

Ahora, si existe un ciclo de largo menor o igual a $n + 1$, este camino utiliza al máximo un par de vértices con $d_{ij} = 2$, por lo cual utiliza por máximo un elemento que no corresponde a una arista del grafo original G. Entonces, si quitamos esta conexión del ciclo, nos queda un camino que también existe en G y visita a cada vértice. Este es exactamente un camino de Hamilton.

En la otra dirección: si G contiene un camino de Hamilton, basamos el ciclo de TSPD es ese camino y añadimos una conexión entre los puntos finales. Esta conexión es costo máximo dos, por lo cual cumplimos con el presupuesto $n+1$, porque el costo del camino es $n - 1$ (todos los pares son aristas en G) y la conexión adicional tiene costo menor o igual a dos.

5.5.4. Coloreo

El coloreo de grafos con dos colores es fácil (pertenece a $\mathbf{P}$), pero con tres colores ya es difícil:

Teorema 29. 3COLORING *es* **NP**-*completo.*

La reducción será de NAESAT a 3COLORING. De una conjunción de cláusulas $\phi = C_1 \wedge \ldots \wedge C_m$ con variables $x_1, \ldots, x_n$, se construye un grafo $G(\phi)$ tal que se puede colorear $G(\phi)$ con tres colores, denotados $\{0, 1, 2\}$ si y sólo si existe una asignación de valores que satisface a ϕ en el sentido NAESAT.

Los gadgets para las variables son unos de elección en forma de triángulo tal que los vértices del triángulo son v, x_i y $\neg x_i$ — el vértice v participa en todos los gadgets de

elección. Para cada cláusula, ponemos también un triángulo con los vértices $[C_{i1}, C_{i2}, C_{i3}]$ donde además cada C_{ij} está conectado al vértice del literal número j de la cláusula C_i.

Verificando la reducción en la dirección "$\Rightarrow$": supone que el vértice v tenga color 2. Entonces ninguno de los vértices de las variables puede tener el color dos, sino que uno o cero. Interpretamos sus colores como una asignación de valores: uno es verdad y cero es falso. Si todos los literales de alguna cláusula son verdad o todos falso, será imposible colorear a $[C_{i1}, C_{i2}, C_{i3}]$. Entonces, los colores satisfacen a ϕ en el sentido de NAESAT.

En la dirección "$\Leftarrow$": supone que ϕ es satisfactible por una asignación T en el sentido NAESAT. Entonces podemos "extraer" el coloreo de $G(\phi)$ de T de la manera siguiente: v tiene color dos. Si $T(x_i) = \top$, el color de x_i es uno y de $\neg x_i$ es cero. En el caso contrario, los colores se asigna viceversa. De cada $[C_{i1}, C_{i2}, C_{i3}]$, dos literales con valores opuestos serán coloreados con uno y cero y el tercer vértice con el color dos.

5.5.5. Conjuntos y números

El problema de 3MATCHING es el siguiente:

Problema 20. 3MATCH 3MATCHING *Dado:* tres conjuntos A, B y C, cada uno con n elementos y una relación ternaria $T \subseteq A \times B \times C$. *Pregunta:* ¿existe un conjunto de n triples (a, b, c) en T que no comparten ningún componente entre cualesquiera dos triples?

Teorema 30. 3MATCHING *es* **NP**-*completo.*

La reducción es de 3SATy está presentado en el libro de texto de Papadimitriou [15]. La cantidad de problemas **NP**-completos de conjuntos es muy grande y uno de ellos sirve para dar una reducción que muestra que *programación entera* es **NP**-completo, mientras *programación lineal* pertenece a $\mathbf{P}$.

5.6. Algoritmos pseudo-polinomiales

También el famoso *problema de la mochila* (KNAPSACK, inglés: knapsack) es **NP**-completo, lo que se muestra por un problema de conjuntos (cubierto exacto, inglés: exact cover). Es un problema clásico de optimización combinatoria donde la instancia es una lista de N diferentes artículos $\varphi_i \in \Phi$ y cada objeto tiene una *utilidad* $\nu(\varphi_i)$ y un peso $\omega(\varphi_i)$. La pregunta es qué conjunto $M \subseteq \Phi$ de artículo debería uno elegir para tener un valor total por lo menos k si tiene una mochila que solamente soporta peso hasta un cierto límite superior Ψ. Entonces, con la restricción

$$\Psi \geq \sum_{\varphi \in M} \omega(\varphi) \tag{5.48}$$

se aspira maximizar la utilidad total

$$\sum_{\varphi \in M} \nu(\varphi) \geq k. \tag{5.49}$$

Cada instancia del problema de la mochila se puede resolver en tiempo $\mathcal{O}(N \cdot \Psi)$, aunque el problema es **NP**-completo. Definimos variables auxiliares $V(w, i)$ que es el valor total máximo posible seleccionando algunos entre los primeros i artículos tal que su peso total es exactamente w. Cada uno de los $V(w, i)$ con $w = 1, \ldots, \Psi$ y $i = 1, \ldots, N$ se puede calcular a través de la ecuación recursiva siguiente:

$$V(w, i+1) = \text{máx}\{V(w, i), v_{i+1} + V(w - w_{i+1}, i)\} \tag{5.50}$$

donde $V(w, 0) = 0$ para todo w y $V(w, i) = -\infty$ si $w \leq 0$. Entonces, podemos calcular en tiempo constante un valor de $V(w, i)$ conociendo algunos otros y en total son $N\Psi$ elementos. Este algoritmo tiene tiempo de ejecución $\mathcal{O}(N \cdot \Psi)$. La respuesta de la problema de decisión es "sí" únicamente en el caso que algún valor $V(w, i)$ sea mayor o igual a k.

Entonces, ¿cómo puede ser esto un problema **NP**-completo? Para pertenecer a $\mathbf{P}$, necesitaría tener un algoritmo polinomial *en el tamaño de la instancia*, que es más como $N \cdot \log \Psi$ y así menor que el parámetro obtenido $N \cdot \Psi$ (tomando en cuenta que $\Psi = 2^{\log \Psi}$). Tal algoritmos donde la cota de tiempo de ejecución es polinomial en los enteros de la entrada y no sus logaritmos se llama un algoritmo *pseudo-polinomial*.

5.7. Problemas fuertemente NP-completos

Un problema es *fuertemente* **NP**-*completo* si permanece **NP**-completo incluso en el caso que toda instancia de tamaño n está restringida a contener enteros de tamaño máximo $p(n)$ para algún polinomial p. Un problema fuertemente **NP**-completo no puede tener algoritmos pseudo-polinomiales salvo que si aplica que **P** sea igual a **NP**. Los problemas SAT, MAXCUT, TSPD y HAMILTON PATH, por ejemplo, son fuertemente **NP**-completos, pero KNAPSACK no lo es.

Capítulo 6

Estructuras de datos

Un paso típico en el diseño de un algoritmo es la elección de una estructura de datos apropiada para el problema. En esta sección se introducen algunas de las estructuras básicas que se utilizan para construir algoritmos eficientes para problemas de decisión y problemas de optimización.

6.1. Arreglos

Un *arreglo* es una estructura capaz de guardar en un orden fijo n elementos. Los índices de las posiciones pueden empezar de cero

$$a[] = [a_0, a_1, a_2, \ldots, a_{n-1}] \tag{6.1}$$

o alternativamente de uno

$$b[] = [b_1, b_2, \ldots, b_{n-1}, b_n]. \tag{6.2}$$

Hay que cuidar que se use las índices consistentemente. Se refiere comúnmente al elemento con índice k como $a[k]$ en vez de a_k. El tiempo de acceso del elemento en posición k en un arreglo es $\mathcal{O}(1)$.

Por defecto, se supone que los elementos guardados en un arreglo no estén ordenados por ningún criterio. La complejidad de identificar si un arreglo no ordenado $a[]$ contiene un cierto elemento x es $\mathcal{O}(n)$, como habrá que comprar cada elemento $a[k]$ con x y el algoritmo termina al encontrar igualdad. Si el elemento no está en el arreglo, tenemos el peor caso de exactamente n comparaciones.

Arreglos sirven bien para situaciones donde el número de elementos que se necesita es conocido y no muy variable. Si el tamaño no está fijo ni conocido, comúnmente hay que ajustar el tamaño por reservar en la memoria *otro* arreglo del tamaño deseado y copiar los contenidos del arreglo actual al nuevo.

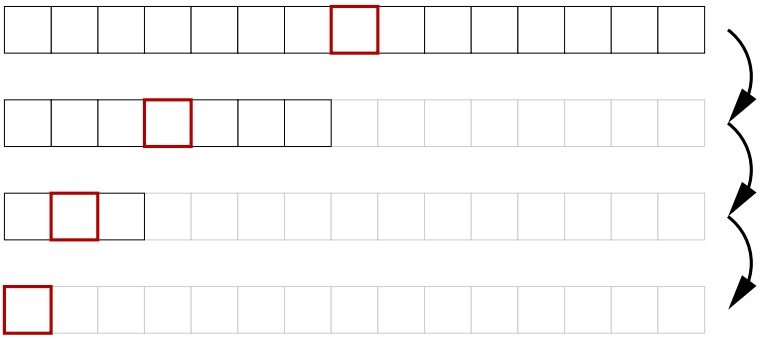

Figura 6.1: El peor caso de la búsqueda binaria: el elemento pivote es $a[k]$ tal que $k = \lfloor \frac{n}{2} \rfloor$. Estamos buscando para el elemento pequeño x en un arreglo que no lo contiene, $x < a[i]$ para todo i. La parte dibujada en gris está rechazada en cada iteración.

6.1.1. Búsqueda binaria

Si el arreglo está ordenado en un orden conocido, un algoritmo mejor para encontrar un elemento igual a x es *la búsqueda binaria*: comparar x con el elemento $a[k]$ donde $k = \lfloor \frac{n}{2} \rfloor$ (o alternativamente $k = \lceil \frac{n}{2} \rceil$, depende de si los índices comienzan de cero o uno). El elemento $a[k]$ se llama el elemento *pivote*. Si $a[k] = x$, tuvimos suerte y la búsqueda ya terminó. Las otras opciones son:

(I) Si $a[k] < x$ y el arreglo está en orden creciente, habrá que buscar entre los elementos $a[k+1]$ y el último elemento.

(II) Si $a[k] > x$ y el arreglo está en orden creciente, habrá que buscar entre el primer elemento y el elemento $a[k-1]$.

(III) Si $a[k] < x$ y el arreglo está en orden decreciente, habrá que buscar entre el primer elemento y el elemento $a[k-1]$.

(IV) Si $a[k] > x$ y el arreglo está en orden decreciente, habrá que buscar entre los elementos $a[k+1]$ y el último elemento.

Entonces, si i es el primer índice del área donde buscar y j es el último índice del área, *repetimos* el mismo procedimiento de elección de k y comparación con un arreglo $a^{(1)} = [a_i, a_{i+1}, \ldots, a_{j-1}, a_j]$. El resto del arreglo nunca será procesado, que nos ofrece un ahorro. De esta manera, si el arreglo siempre se divide en dos partes de aproximadamente el mismo tamaño, la iteración termina cuando la parte consiste de un sólo elemento. El peor caso es que el elemento esté en la primera o la última posición del arreglo. La figura 6.1 muestra un ejemplo del peor caso.

Para contar cuántas divisiones tiene el peor caso, tomamos en cuenta que el tamaño de la parte que queda para buscar tiene al máximo $\lceil \frac{n}{2} \rceil$ elementos. Sabemos que al último nivel el tamaño de la parte es uno. Considerando estas observaciones, llegamos ya con pura intuición el hecho que en el peor caso habrá $\log_2(n)$ divisiones. Cada división contiene una comparación de x con un $a[k]$ y la asignación del nuevo índice inferior y el nuevo índice superior. Entonces son $3\log_2(n)$ operaciones y la complejidad asintótica es $\mathcal{O}(\log(n))$.

6.1.2. Ordenación de arreglos

Algoritmo de burbuja (bubble sort)

Hay varios algoritmos para ordenar los elementos de un arreglo. Uno de los más básicos — y menos eficientes — es el algoritmo de *burbuja* (inglés: bubble sort):

(I) Inicia una variable contadora a cero: $c := 0$.

(II) Comenzando desde el primer elemento, compáralo con el siguiente.

(III) Si su orden está correcto con respeto a la ordenación deseada, déjalos así.

(IV) Si no están en orden, con la ayuda de una variable auxiliar t, intercambia sus valores y incrementa a la contadora, $c := c + 1$.

(V) Avanza a comparar el segundo con el tercero, repitiendo el mismo procesamiento, hasta llegar al final del arreglo.

(VI) Si al final, $c \neq 0$, asigna $c := 0$ y comienza de nuevo.

(VII) Si al final $c = 0$, el arreglo está en la orden deseada.

En cada paso un elemento encuentra su lugar correcto, por lo cuál el número máximo de iteraciones de hacer es n.

El peor caso es que el orden de los elementos es exactamente lo contrario a lo deseado: por ejemplo, para ordenar n elementos en orden creciente a orden decreciente, necesitamos repetir el acceso al arreglo entero n veces (ve ejemplo en figura 6.2. En cada iteración, se hace $n - 1$ comparaciones y en el peor caso cada uno llega a un intercambio de posiciones. La complejidad asintótica del algoritmo es $\mathcal{O}(n^2)$.

Figura 6.2: El peor caso de ordenación con el algoritmo burbuja: los elementos están al comienzo en orden creciente y el orden deseado es decreciente. La posición del procesamiento está marcada con letra negrita y por cada cambio, hay una línea nueva. Las iteraciones están separadas por líneas horizontales.

Ordenación por selección (selection sort)

Dado un arreglo de n elementos, podemos ordenar sus elementos en el orden creciente con el siguiente procedimiento:

(I) Asigna $i :=$ primer índice del arreglo $a[]$.

(II) Busca entre i y el fin del arreglo el elemento menor.

(III) Denote el índice del mejor elemento por k y guarda su valor en una variable auxiliar $t := a[k]$.

(IV) Intercambia los valores de $a[i]$ y $a[k]$: $a[k] := a[i]$, $a[i] := t$.

(V) Incrementa el índice de posición actual: $i := i + 1$.

(VI) Itera hasta que i esté en el fin del arreglo.

Para ordenar en orden decreciente, habrá que siempre buscar el elemento mayor entre i y el fin del arreglo. Este algoritmo se llama *ordenación por selección* (inglés: selection sort).

En el peor caso, cuando los valores del arreglo están en orden reverso a lo deseado, la primera iteración tiene $n-1$ comparaciones, la segunda tiene $n-2$, etcétera, hasta que el último solamente tiene una. En total, hay

$$1 + 2 + \ldots + n - 2 + n - 1 = \sum_{j=1}^{n-1} j = \sum_{j=0}^{n-1} j = \frac{n(n-1)}{2} \tag{6.3}$$

por ecuación 1.35. Entonces, la complejidad asintótica es $\mathcal{O}(n^2)$, pero en práctica, ahorramos varias operaciones en comparación con la ordenación por burbuja.

Ordenación por fusión (mergesort)

Es una operación $\mathcal{O}(n)$ combinar dos listas ordenadas en una sola lista ordenada bajo el mismo criterio. Por ejemplo, si la entrada son dos listas A y B de números enteros ordenaros del menor a mayor, la lista combinada se crea por leer el primer elemento de A y el primer elemento de B, añadir el mínimo de estos dos en la lista nueva C y re-emplazar en la variable auxiliar por leer el siguiente elemento de su lista de origen. En este idea se basa el algoritmo siguiente de ordenación.

La ordenación *por fusión* (inglés: mergesort) funciona por divisiones parecidas a las de la búsqueda binaria. El arreglo está dividido a dos partes del mismo tamaño (más o menos un elemento): la primera parte tiene largo $\lfloor \frac{n}{2} \rfloor$ y la segunda tiene largo $n - \lfloor \frac{n}{2} \rfloor$. Ambos

subarreglos están divididos de nuevo hasta que contengan k elementos, $k \geq 1 \ll n$. Al llegar al nivel donde la lista tiene k elementos, se utiliza otro algoritmo de sortearlo, por ejemplo uno de los dos ya presentados. Opcionalmente se podría fijar $k = 1$.

Las dos subarreglos $b_\ell[]$ y $b_r[]$ así ordenados están *combinados* con uso de memoria auxiliar: se forma un arreglo $b[]$ por elegir elementos del comienzo de los dos arreglos ordenados:

(I) Asigna $i_\ell := 0$, $i_r := 0$ y $i := 0$

(II) Si $b_\ell[i_\ell] < b_r[i_r]$, asigna $b[i] := b_\ell[i_\ell]$ y después actualiza los índices relacionados: $i_\ell := i_\ell + 1$ y $i := i + 1$.

(III) Si $b_\ell[i_\ell] \geq b_r[i_r]$, asigna $b[i] := b_r[i_r]$ y después actualiza los índices relacionados: $i_r := i_r + 1$ y $i := i + 1$.

(IV) Cuando i_ℓ o i_r pasa afuera de su subarreglo, copia todo lo que queda del otro arreglo al final de $b[]$.

(V) Mientras todavía quedan elementos en los dos subarreglos, repite la elección del elemento menor a guardar en $b[]$.

Ordenación rápida (quicksort)

La idea de la ordenación rápida es también dividir el arreglo, pero no necesariamente en partes de tamaño igual. Se elige un elemento pivote $a[k]$ según algún criterio (existen varias opciones como elegirlo), y divide el arreglo de entrada $a[]$ en dos partes: una parte donde todos los elementos son menores a $a[k]$ y otra parte donde son mayores o iguales a $a[k]$ por escanear todo el arreglo una vez. Esto se puede implementar con dos índices moviendo el el arreglo, uno del comienzo y otro del final. El índice del comienzo busca por el primer elemento con un valor mayor o igual al pivote, mientras el índice de atrás mueve en contra buscando por el primer elemento menor al pivote. Al encontrar elementos tales antes de cruzar un índice contra el otro, se intercambia los dos valores y continua avanzando de las mismas posiciones de los dos índices. Al cruzar, se ha llegado a la posición donde cortar el arreglo a las dos partes.

Se repite el mismo procedimiento con cada uno de las dos partes. Así nivel por nivel resultan ordenadas los subarreglos, y por el procesamiento hecha ya en los niveles anteriores, todo el arreglo resulta ordenado. El análisis de quicksort está presentado como un ejemplo del análisis amortizada en sección 7.3.1.

6.2. Listas

Listas son estructuras un poco más avanzadas que puros arreglos, como típicamente permiten ajustes de su capacidad.

Una lista *enlazada* (inglés: linked list) consiste de elementos que todos contengan además de su dato, un *puntero* al elemento siguiente. Si el orden de los elementos no está activamente mantenido, es fácil agregar un elemento en la lista: crear el elemento nuevo, inicializar su puntero del siguiente elemento a nulo, y hacer que el puntero del siguiente del último elemento actualmente en la lista apunte al elemento nuevo. Para acceder la lista, hay que mantener un puntero al primer elemento. Si también se mantiene un puntero al último elemento, añadir elementos cuesta $\mathcal{O}(1)$ unidades de tiempo, mientras solamente utilizando un puntero al comienzo, se necesita tiempo $\mathcal{O}(n)$, donde n es el número de elementos en la lista.

Si uno quiere mantener el orden mientras realizando inserciones y eliminaciones, hay que primero ubicar el elemento *anterior* al punto de operación en la lista:

- para *insertar* un elemento nuevo v inmediatamente *después* del elemento u actualmente en la lista, hay que ajustar los punteros tal que el puntero del siguiente v.sig de v tenga el valor de u.sig, después de que se cambia el valor de u.sig a puntar a v;
- para *eliminar* un elemento v, el elemento anterior siendo u, primero hay que asignar u.sig $:= v$.sig y después simplemente eliminar v, a que ya no hay referencia de la lista.

Una lista *doblemente enlazada* tiene además en cada elemento un enlace al elemento anterior. Su mantenimiento es un poco más laborioso por tener que actualizar más punteros por operación, pero hay aplicaciones en las cuales su eficacia es mejor.

Con listas, ganamos tamaño dinámico, pero búsquedas y consultas de elementos ahora tienen costo $\mathcal{O}(n)$ mientras con arreglos tenemos acceso en tiempo $\mathcal{O}(1)$ y tamaño "rígido".

6.2.1. Pilas

Una *pila* (inglés: stack) es una lista especial donde todas las operaciones están con el primer elemento de la lista: se añade al frente y remueve del frente. Su función es fácil de entender pensando en una pila de papeles en un escritorio donde siempre se toma el papel de encima para procesar y cada documento nuevo se coloca encima de los anteriores. Una pila es bastante fácilmente implementada como una lista enlazada manteniendo un puntero p al primer elemento. Al añadir un elemento nuevo v, primero se asigna v.sig $:= p$ y después $p := v$.

6.2.2. Colas

Una *cola* (inglés: queue) es una estructura donde los elementos nuevos llegan al final, pero el procesamiento se hace desde el primer elemento. También colas están fácilmente implementadas como listas enlazadas, manteniendo un puntero al comienzo de la cola y otro al final.

6.2.3. Ordenación de listas

Ordenación por inserción

Para ordenar una lista $L = [\ell_1, \ell_2, \ldots, \ell_n]$ en orden creciente, necesitamos una subrutina `insertar(L, i, x)` que busca desde el comienzo la posición i en la lista L por un elemento $\ell_j \leq x$ hacía la primera posición, tal que $j \leq i$. Al encontrar tal elemento, el elemento x estará insertada en la posición justo después del elemento ℓ_j. Si no se encuentra un elemento así, se inserta x al comienzo de la lista.

El procedimiento de la ordenación empieza con el primer elemento de la lista y progresa con una variable indicadora de posición i hasta el último elemento. Para cada elemento, quitamos ℓ_i de la lista y llamamos la subrutina `insertar`(L, i, x) para volver a guardarlo.

6.3. Árboles

Un árbol de n vértices etiquetados $1, 2, \ldots, n$ se puede guardar en un arreglo $a[]$ de n posiciones, donde el valor de $a[i]$ es la etiqueta del vértice padre del vértice i. Otra opción es guardar en cada elemento un puntero al vértice padre (y posiblemente del padre una estructura de punteros a sus hijos).

Árboles son estructuras muy útiles para servir como *índices* de bases de datos, o más simplemente, grupos de datos que habrá que ordenar. Árboles permiten realizar eficientemente operaciones como *inserciones*, *eliminaciones* y *búsquedas* de los elementos guardados. El único requisito es que exista un *orden* sobre el espacio de las *claves* de los elementos que se maneja, tal como el orden alfabético o orden creciente numérico, etcétera. Cada elemento consiste de una clave (no necesariamente único) y un dato — el árbol de índice se ordena por las claves y se puede buscar por el dato asociado a una cierta clave.

6.3.1. Árboles binarios

Árboles binarios son una clase de árbol en uso muy común. En un árbol binario, cada

vértice que no es una hoja tiene al máximo dos vértices hijos: su hijo izquierdo y su hijo derecho. Para un ejemplo, ve la figura 6.3. Si ningún vértice tiene solamente un hijo, se dice que el árbol está *lleno*.

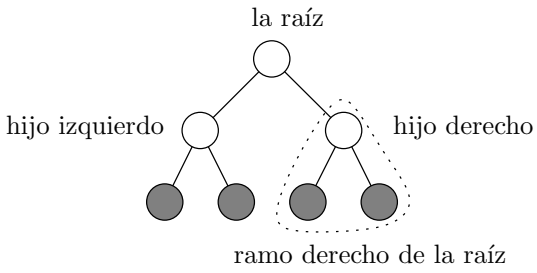

Figura 6.3: Un ejemplo de un árbol binario.

Su uso como índices es relativamente fácil también para bases de datos muy grandes, como diferentes ramos y partes del árbol se puede guardar en diferentes *páginas* de la memoria física de la computadora. La figura 6.4 contiene un ejemplo, adaptado de Knuth [11].

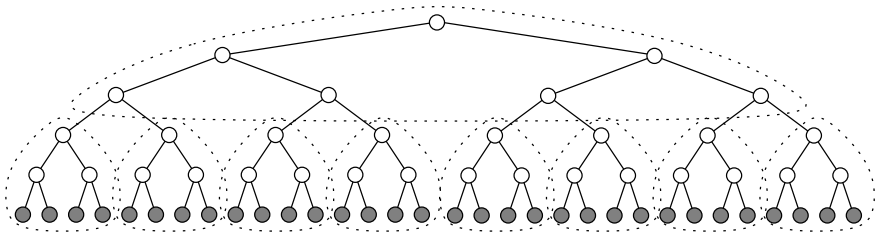

Figura 6.4: Un ejemplo de cómo dividir un árbol de índice en varias páginas de memoria: las líneas agrupan juntos subárboles del mismo tamaño.

El problema con árboles binarios es que su forma depende del orden de inserción de los elementos y en el peor caso puede reducir a casi una lista (ver la figura 6.5).

6.3.2. Árboles AVL

Árboles AVL (de Adel'son-Vel'skii y Landis [2]) son árboles binarios que aseguran complejidad asintótica $\mathcal{O}(\log n)$ para las operaciones básicas de índices.

La variación de árboles AVL que estudiamos acá guarda toda la información en sus hojas y utiliza los vértices "internos" para información utilizado al realizar las operaciones del índice. Los vértices que no son hojas ahora se llaman vértices de *ruteo* . El orden del árbol es tal que todas las hojas en el ramo del hijo izquierdo contienen claves *menores*

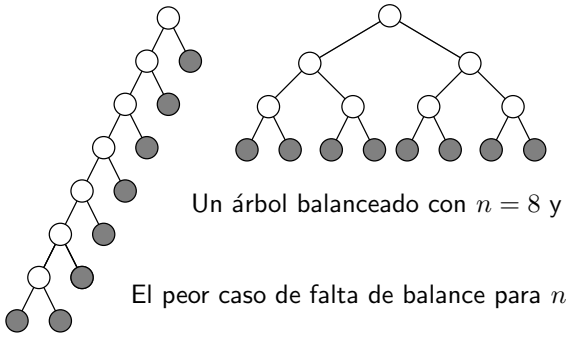

Figura 6.5: Un árbol binario de peor caso versus un árbol binario de forma óptima, ambos con ocho vértices hojas.

que el valor del vértice de ruteo y todas las hojas en el ramo del hijo derecho contienen claves *mayores o iguales* que el valor del vértice de ruteo mismo.

Por el uso de vértices de ruteo, los árboles que resultan son árboles llenos. Utilizamos en los ejemplos enteros positivos como las claves. Para buscar la hoja con clave i, se empieza del raíz del árbol y progresa recursivamente al hijo izquierdo si el valor del raíz es *mayor* a i y al hijo derecho si el valor es *menor o igual* a i. Cuando la búsqueda llega a una hoja, se evalúa si el valor de la hoja es i o no. Si no es i, el árbol no contiene la clave i en ninguna parte. El pseudocódigo de la búsqueda está en el cuadro 6.1.

El árbol está *balanceado* si el largo máximo es k, el largo mínimo tiene que ser mayor o igual a $k - 1$. En este caso, el número de hojas del árbol n es

$$2^k \leq n < 2^{k+1}. \tag{6.4}$$

La condición que utilizamos para decidir si o no un dado árbol esta balanceado se llama la *condición de balance AVL* [2]; también existen otras condiciones. Para formular la condición, hay que definir la *altura* de un vértice:

$$\mathcal{A}(v) = \begin{cases} 1, \text{ si } v \text{ es una hoja} \\ \max\{\mathcal{A}(\mathit{izq}(t)), \mathcal{A}(\mathit{der}(t))\} + 1, \text{ si } v \text{ es de ruteo.} \end{cases} \tag{6.5}$$

La altura de un ramo de un vértice v, es decir, un subárbol la raíz de cual es v es la altura de v. La altura del árbol entero es la altura de su raíz. Un árbol balanceado se puede caracterizar como un árbol con raíz v_r con $\mathcal{A}(v_r) = \mathcal{O}(\log n)$. La condición de balance AVL es que $\forall v \in V$

$$|\mathcal{A}(\mathit{izq}(v)) - \mathcal{A}(\mathit{der}(v))| \leq 1. \tag{6.6}$$

Cuadro 6.1: Pseudocódigo del procedimiento de búsqueda en un árbol binario ordenado con los datos en las hojas.

```
procedimiento ubicar(int clave, nodo actual) : hoja {
  si (actual es una hoja) {
    devuelve hoja ;
  } en otro caso {
    si (actual.clave < clave) {
      ubicar(clave, actual.derecho);
    } en otro caso {
      ubicar(clave, actual.izquierdo);
    }
  }
}
```

Para derivar unas cotas sobre la forma del árbol, definimos además la *profundidad* de cada vértice del árbol:

$$\mathcal{D}(v) = \begin{cases} 0, \text{ si } v \text{ es la raíz,} \\ \mathcal{D}(v.\mathcal{P}) + 1, \text{ en otro caso.} \end{cases} \tag{6.7}$$

La profundidad del árbol entero es simplemente $\max_v \mathcal{D}(v)$. Aplica que $\mathcal{D} = \mathcal{A} - 1$.

Denotamos por n el numero de vértices en total y por $\mathcal{H}$ el numero de hojas del árbol. Para todo $n = 2k$, tenemos $\mathcal{H} = 2n - 1$ y $\mathcal{D} = \log_2 n$. Para ubicar a una clave a profundidad d toma exactamente d pasos en el árbol, es decir, tiempo $\mathcal{O}(d)$. Entonces, para cada árbol perfectamente balanceado de $\mathcal{H}$ hojas, se puede localizar una clave en tiempo

$$\mathcal{O}(\mathcal{D}) = \mathcal{O}(\log_2 \mathcal{H}) = \mathcal{O}(\log_2 n). \tag{6.8}$$

Para un árbol balanceado, la diferencia en el largo de los caminos es una constante, de hecho uno, por lo cual también para esos árboles aplica que su tiempo de acceso a cualquier clave es $\mathcal{O}(\log_2 n)$.

La condición de balance AVL implica que para un vértice de ruteo v_r con $\mathcal{A}(v_r) = a$ es necesario que

- o ambos de sus hijos tengan altura $a - 1$
- o un hijo tiene altura $a - 1$ y el otro altura $a - 2$.

El número de vértices de ruteo en el ramo de una hoja es cero. El *tamaño del ramo* con raíz en v se puede expresar en términos de los tamaños de los ramos de sus hijos: el número de vértices de ruteo $\mathcal{R}_v$ en el ramo de v es la suma del número de vértices de ruteo en el ramo de su hijo izquierdo $\mathcal{R}_w$ con el número de vértices de ruteo en el ramo de su hijo derecho $\mathcal{R}_u$ más uno por el vértice de ruteo v mismo.

Utilizamos esta relación para escribir una ecuación recursiva para llegar a una cota superior de la altura de un árbol: sea la altura del hijo izquierdo w exactamente $a-1$ y la altura del otro hijo vu la otra opción $a-2$. Denotamos por $\mathcal{R}(a)$ el número de vértices de ruteo en un ramo con raíz un v en altura a. Nota que suponemos que el árbol está en balance no perfecto en esta parte: w tiene altura $a-1$ mientras u tiene altura $a-2$. La ecuación recursiva es

$$\begin{aligned}
\mathcal{R}_v &= \mathcal{R}_w + \mathcal{R}_u + 1 \\
\mathcal{R}(a) &= \mathcal{R}(a-1) + \mathcal{R}(a-2) + 1 \\
1 + \mathcal{R}(a) &= 1 + \mathcal{R}(a-1) + \mathcal{R}(a-2) + 1 \\
\big(1 + \mathcal{R}(a)\big) &= \big(1 + \mathcal{R}(a-1)\big) + \big(1 + \mathcal{R}(a-2)\big) \\
\mathcal{F}(a) &= \mathcal{F}(a-1) + \mathcal{F}(a-2).
\end{aligned} \tag{6.9}$$

La sustitución que hicimos era $1 + \mathcal{R}(k) = \mathcal{F}(k)$. Hemos llegado a la definición de la *sucesión de Fibonacci*, donde el elemento en posición k se llama el k-écimo *número de Fibonacci*:

$$\mathcal{F}_k = \begin{cases} 0, & \text{si } k = 0 \\ 1, & \text{si } k = 1 \\ \mathcal{F}_{k-1} + \mathcal{F}_{k-2}, & \text{para } k > 1. \end{cases} \tag{6.10}$$

A los números de Fibonacci aplica que

$$\mathcal{F}(a) > \frac{\phi^a}{\sqrt{5}} - 1 \tag{6.11}$$

donde $\phi = \frac{1+\sqrt{5}}{2}$ es la *tasa dorada*. Las condiciones iniciales son los siguientes: $\mathcal{R}(0) = 0$ porque no hay vértices de ruteo en ramos de las hojas y $\mathcal{R}(1) = 1$ porque el vértice de ruteo mismo es el único en su ramo si sus ambos hijos son hojas.

Entonces la ecuación recursiva aplica para $\mathcal{R}(k)$ donde $k \geq 2$. La sustitución $1 + \mathcal{R}(k) = \mathcal{F}(k)$ tampoco aplica desde el comienzo, como

$$\begin{aligned}
\mathcal{F}(0) = 0 &< \mathcal{R}(0) + 1 = 0 + 1 = 1 \\
\mathcal{F}(1) = 1 &< \mathcal{R}(1) + 1 = 1 + 1 = 2.
\end{aligned} \tag{6.12}$$

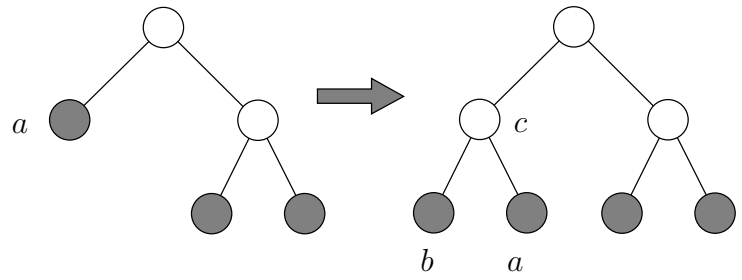

Figura 6.6: La inserción de un elemento nuevo con la clave b tal que $b < a$ resulta en la creación de un vértice de ruteo nuevo c, hijos del cual serán los vértices hojas de a y b.

Sin embargo, los valores 1 y 2 también aparecen en la sucesión de Fibonacci:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, \ldots \tag{6.13}$$

— ¡son $\mathcal{F}(2)$ y $\mathcal{F}(3)$! Entonces, podemos escribir para cada k que $\mathcal{R}(k) = \mathcal{F}(k+2)+1$. Esto nos da

$$\begin{aligned} \mathcal{R}(a) &= \mathcal{F}(a+2) - 1 > \tfrac{\phi^{a+2}}{\sqrt{5}} - 2 \\ \mathcal{R}(a) - 2 &= \tfrac{\phi^{a+2}}{\sqrt{5}} \\ \log_\phi(\mathcal{R}(a)+2) &= \log_\phi\left(\tfrac{\phi^{a+2}}{\sqrt{5}}\right) \\ \log_\phi(\mathcal{R}(a)+2) &= a + 2 - \log_\phi(\sqrt{5}) \\ a &\approx 1{,}440\, \log(\mathcal{R}(a)+2) - 0{,}328. \end{aligned} \tag{6.14}$$

Ya sabemos que $n > \mathcal{R}(\mathcal{A})$ porque $\mathcal{H} > 0$ — siempre hay hojas si el árbol no está completamente vacío. Entonces aplica que siguiente:

Teorema 31. *Para cada árbol que cumple con la condición de balance AVL,* $\mathcal{A} \lessapprox 1{,}440 \log(n+2) - 0{,}328$.

Para *insertar* un elemento nuevo al árbol de índice, primero hay que buscar la ubicación de la clave del elemento. Llegando a la hoja v_h donde debería estar la clave, hay que crear un vértice de ruteo v_r nuevo. La hoja v_h va a ser uno de los hijos del vértice de ruteo y el otro hijo será un vértice nuevo v_n creado para el elemento que está insertado. El elemento menor de v_h y v_n será el hijo izquierdo y el mayor el hijo derecho. El valor del vértice de ruteo v_r así creado será igual al valor de su hijo derecho. La figura 6.6 muestra un ejemplo.

Para *eliminar* un elemento del árbol, hay que primero ubicar su posición y después eliminar además de la hoja su vértice de ruteo v_r y mover el otro vértice hijo del vértice de ruteo v_h a la posición que ocupó v_r. La figura 6.7 muestra un ejemplo.

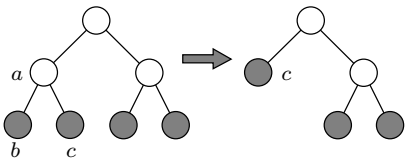

Figura 6.7: Al eliminar una hoja con clave b, también su padre, el vértice de ruteo con el valor a está eliminado.

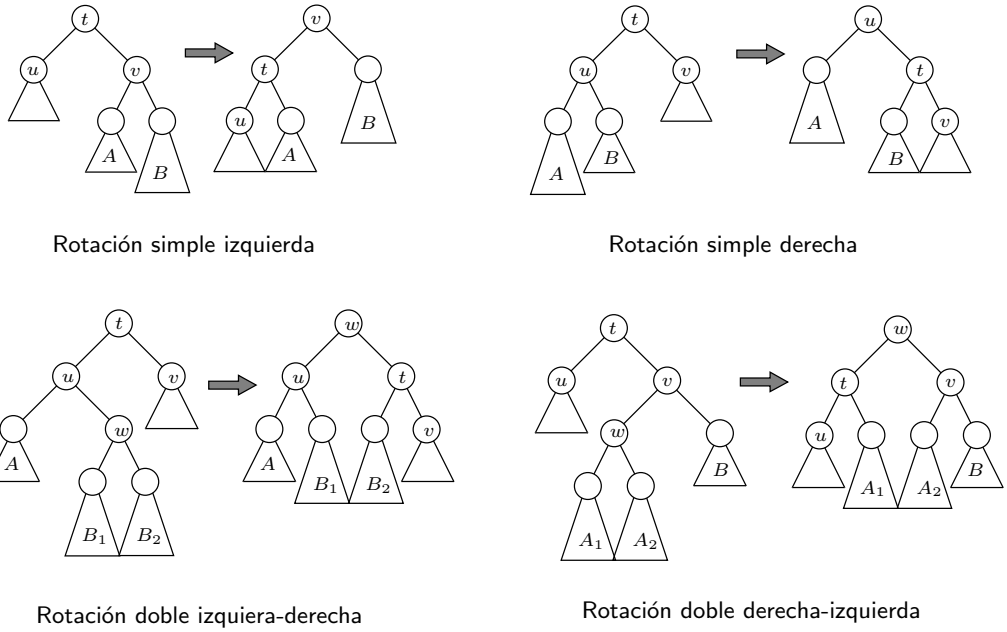

Figura 6.8: Cuatro rotaciones básicas de árboles binarios para restablecer balance de las alturas de los ramos.

Las operaciones de insertar y eliminar claves modifican la forma del árbol. La garantía de tiempo de acceso $\mathcal{O}(\log n)$ está solamente válida a árboles *balanceados*. Un árbol está *perfectamente balanceado* si su estructura es óptima con respeto al largo del camino de la raíz a cada hoja: todas las hojas están en el mismo nivel, es decir, el largo máximo de tal camino es igual al largo mínimo de tal camino sobre todas las hojas. Esto es solamente posible cuando el número de hojas es 2^k para $k \in \mathbb{Z}^+$, en que caso el largo de todos los caminos desde la raíz hasta las hojas es exactamente k.

Necesitamos operaciones para "recuperar" la forma balanceada después de inserciones y eliminaciones de elementos, aunque no cada operación causa una falta de balance en el árbol. Estas operaciones se llaman *rotaciones* . Las cuatro variaciones básicas de rotaciones están presentadas en la figura 6.8.

La rotación adecuada está elegida según las alturas de los ramos que están fuera de balance, es decir, tienen diferencia de altura mayor o igual a dos. Si se balancea después de *cada inserción y eliminación* siempre y cuando es necesario, la diferencia será siempre exactamente dos: denotamos los hijos que están fuera de balance del vértice t por u y v, con la notación de la figura 6.8 para los ramos: la "dirección" de la diferencia juntos con propiedades de los ramos determinan cuál rotación será implementada en v

$$\begin{cases} \mathcal{A}(u) \geq \mathcal{A}(v) + 2 \quad : \quad \begin{cases} \mathcal{A}(A) \geq \mathcal{A}(B) & \Rightarrow \\ \text{rotación simple a la derecha,} \\ \mathcal{A}(A) < \mathcal{A}(w) & \Rightarrow \\ \text{rotación doble izquierda-derecha,} \end{cases} \\ \mathcal{A}(u) \leq \mathcal{A}(v) - 2 \quad : \quad \begin{cases} \mathcal{A}(A) \geq \mathcal{A}(B) & \Rightarrow \\ \text{rotación simple a la izquierda,} \\ \mathcal{A}(B) < \mathcal{A}(w) & \Rightarrow \\ \text{rotación doble derecha-izquierda.} \end{cases} \end{cases} \tag{6.15}$$

Con esas rotaciones, ninguna operación va a aumentar la altura de un ramo, pero la puede reducir por una unidad. La manera típica de encontrar el punto de rotación t es regresar hacía la raíz después de haber operado con una hoja para verificar si todos los vértices en camino todavía cumplan con la condición de balance. La complejidad asintótica de buscar una hoja toma $\mathcal{O}(\log n)$ tiempo, la operación en la hoja tiempo constante $\mathcal{O}(1)$, la "vuelta" hacía la raíz otros $\mathcal{O}(\log n)$ pasos, y cada rotación un tiempo constante $\mathcal{O}(1)$. Si al ejecutar una rotación, la altura de t cambia, habrá que continuar hacía la raíz porque otras faltas de balance pueden técnicamente haber resultado. Si no hay cambio en la altura de t, no hay necesidad de continuar más arriba en el árbol.

6.3.3. Árboles rojo-negro

Otro tipo de árboles binarias son los *árboles rojo-negro* (inglés: red-black tree) . Las reglas para mantener orden de las claves es las mismas. También ofrecen tiempo de acceso y actualización $\mathcal{O}(\log n)$ y se balancea por rotaciones. En vez de la condición AVL, se identifica vértices fuera de balance por asignar colores a los vértices. Un árbol rojo-negro cumple las siguientes propiedades:

(I) Cada vértice tiene exactamente uno de los dos colores: rojo y negro.

(II) La raíz es negro.

(III) Cada hoja es negro.

(IV) Si un vértice es rojo, sus ambos hijos son negros.

(V) Para cada vértice v, todos los caminos de v a sus descendientes contienen el mismo número de nodos negros.

Entonces, la estructura de un vértice contiene además de su clave y los punteros al padre y los dos hijos, un color. Aplica para los árboles rojo-negro que su *altura* es $\mathcal{O}(\log n)$ y que la expresión exacta tiene cota superior $2\log_2(n+1)$.

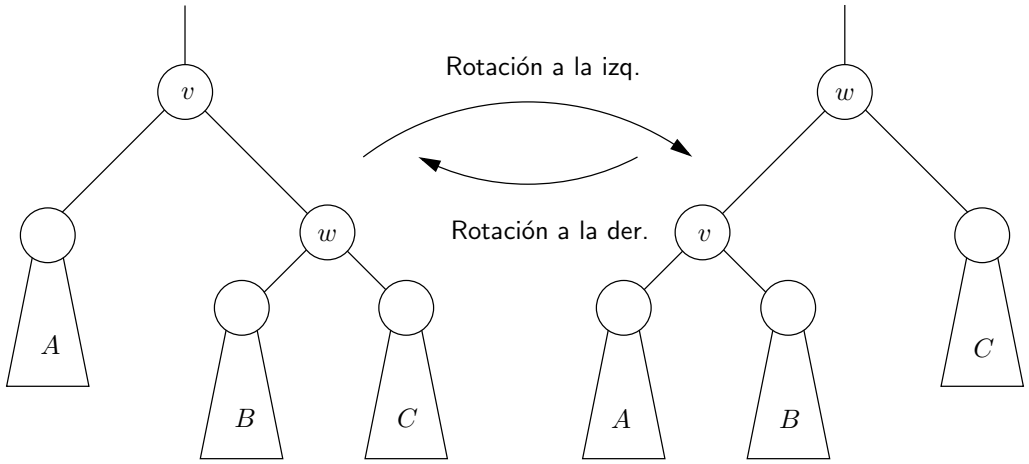

Figura 6.9: Las dos rotaciones de los árboles rojo-negro son operaciones inversas una para la otra.

Los árboles rojo-negro se balancea con rotaciones simples (ver figura 6.9. Al insertar hojas nuevas, además de las rotaciones para restaurar balance, puede hacer falta recolorear algunos vértices en el camino desde la hoja nueva hasta la raíz. Las posibles violaciones son de dos tipos: vértices de ruteo rojos que llegan a tener un hijo rojo por las rotaciones y la raíz siendo un vértice rojo por rotaciones. Es una idea buena implementar las rotaciones y los cambios de color en una sola subrutina que se invoca después de cada inserción para manejar los cambios eficientemente. También las operaciones de eliminación necesitan su propia subrutina para las rotaciones y recoloreo, aunque es un poco más complicado que el caso de inserción.

6.3.4. Árboles B

Los árboles B (inglés: B-tree) son aboles balanceados *no* binarios. Todos los vértices contienen datos y el número por datos por vértice puede ser mayor a uno. Si un vértice

internal contiene k claves $a_1, a_2, \ldots, a_k$, tiene necesariamente $k+1$ hijos que contienen las claves en los intervalos $[a_1, a_2]$, $[a_2, a_3], \ldots, [a_{k-1}, a_k]$.

Cada vértice contiene la información siguiente:

1. su número de claves k
2. las k claves en orden no decreciente
3. un valor binario que indica si o no el vértice es una hoja
4. si no es una hoja, $k+1$ punteros a sus hijos $c_1, c_2, \ldots, c_{k+1}$

Aplica para las d claves $b_1, \ldots, b_d$ en el ramo del hijo c_i que $a_i \leq b_j \leq a_{i+1}$ para cada $j \in [1, d]$. Todas las hojas del árbol tienen la misma profundidad y la profundidad es exactamente la altura del árbol. Además, se impone cotas superiores e inferiores al número de claves que un vértice puede contener:

(I) Cada vértice salvo que la raíz debe contener por lo menos $t-1$ claves.

(II) Cada vértice puede contener al máximo $2t-1$ claves.

En consecuencia, cada vértice que no es hoja tiene por lo menos t hijos y al máximo $2t$ hijos. Un vértice es *lleno* si contiene el número máximo permitido de claves.

En los árboles B aplica para la altura a del árbol que (omitimos la demostración) para $t \geq 2$,

$$a \leq \log_t \frac{n+1}{2}. \tag{6.16}$$

Búsqueda de una clave en un árbol B no diferencia mucho de la operación de búsqueda en árboles binarios, el único cambio siendo que habrá que elegir entre varias alternativas en cada vértice intermedio.

Para *insertar*, primero buscamos la posición en dónde insertar la clave nueva. Si el vértice donde deberíamos realizar la inserción todavía no está lleno, insertamos la clave y estamos listos. Si el vértice es lleno, habrá que identificar su clave mediana y dividir el vértice en dos partes. La mediana misma moverá al vértice padre para marcar la división, pero esto puede causar que el padre también tendrá que dividirse. Las divisiones pueden continuar recursivamente hasta la raíz.

Como también impusimos una cota inferior al número de claves, al eliminar una clave podemos causar que un vértice sea "demasiado vacío". Entonces, al eliminar claves, los vértices "chupan" claves de reemplazo de sus hojas o de su padre. La operación que resulta se divide en varios casos posibles.

Los árboles *multicaminos* (inglés: B+ trees) tienen además punteros extras entre vértices que son "hermanos" para ofrecer más posibilidades simples para mover claves al buscar, insertar y eliminar claves.

6.3.5. Árboles biselados

Los *árboles biselados* (inglés: splay tree) son árboles binarios que ofrecen en tiempo $\mathcal{O}(\log n)$ cualquiera de las operaciones siguientes: búsqueda, inserción y eliminación de una clave, igual como la unión y división de árboles. En un árbol biselado *cada vértice* contiene a una clave y las claves en el ramo izquierdo son menores que la clave del vértice mismo, mientras a la derecha están las claves mayores. Las claves son únicas. Los árboles biselados no están balanceados, si no por suerte — las operaciones no cuidan ni restauran balance.

En este caso, no hay ningún dato asociado a una clave, por lo cual una búsqueda por la clave ℓ tiene simplemente salidas "sí" (en el caso que la clave está presente en el árbol) y "no" (en el caso que la clave no está incluido). Para aplicar una unión, todas las claves de uno de los dos árboles tienen que ser menores a la clave mínima del otro. La división divide a un árbol a dos árboles tal que todas las claves de uno de los árboles que resultan son menores o iguales a una clave dada como parámetro y las mayores están en el otro árbol.

La operación básica utilizada para implementar todas estas operaciones es `splay`(ℓ, A): lo que hace `splay` es convertir el árbol A a tal forma que el vértice con clave ℓ es la raíz, si presente, y en la ausencia de ℓ en A, la raíz será

$$\max\{k \in A \mid \ell > k\} \tag{6.17}$$

si A contiene claves menores a ℓ y $\min\{k \in A\}$ en otro caso. El orden de las claves después de la operación cumple el mismo requisito de orden de las claves.

Las operaciones están implementadas de la manera siguiente utilizando `splay`:

- **búsqueda de ℓ en A**: ejecuta `splay`(ℓ, A). Si la raíz en el resultado es ℓ, la salida es "sí", en otro caso "no".
- **unión de A_1 con A_2**: se supone que las claves de A_1 son todos menores que la clave mínima de A_2. Ejecutamos `splay`(∞, A_1) para lograr que la raíz de A_1 modificado es su clave máxima y todos los otros vértices están en el ramo izquierdo. Hacemos que A_2 sea el ramo derecho. (Ver figura 6.10.)
- **división de A con la clave ℓ**: ejecutamos `splay`(ℓ, A). Los dos árboles resultantes serán tal que A_1 contiene solamente el ramo derecho de la raíz y A_2 el resto del A modificado.
- **eliminación de ℓ de A**: divide A con la clave ℓ. Si resulta que ℓ es la raíz, quítalo y ejecuta la unión de los dos ramos sin ℓ. Si ℓ no es la raíz, solamente vuelve a juntar A_1 como el ramo derecho de A_2.

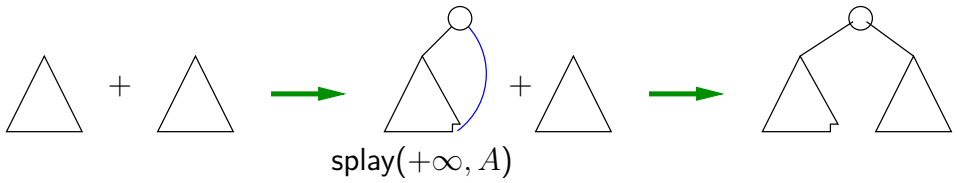

Figura 6.10: La unión de dos árboles biselados.

- **inserción de ℓ en A**: ejecuta la división de A con ℓ. Si ℓ ya es la raíz, solamente vuelve a juntar A_1 como el ramo derecho de A_2. Si no lo es, crea un vértice raíz nuevo con la clave ℓ y haz A_2 su ramo derecho y A_1 su ramo izquierdo.

Lo que queda entender es cómo implementar el método `splay` mismo. Comenzamos como cualquier búsqueda en un árbol binario ordenado desde la raíz utilizando el hecho que las claves pequeñas están a la izquierda y las grandes a la derecha — en comparación con la clave del vértice actual. Al terminar la búsqueda en un vértice v, empezamos rotaciones simples para mover v hacía la raíz sin mezclar el orden de las claves (ver las rotaciones simples en la figura 6.8). Las rotaciones se elige según las relaciones entre un vértice v y su padre y "abuelo".

Si v tiene padre pero *no tiene abuelo*, elegimos una rotación simple derecha de la figura 6.8). Este es el caso de la última operación con cual v llega al ser la raíz.

Si v tiene un padre u y un abuelo w los dos, hay varios casos. En el caso que v y u los dos son hijos derechos, elegimos una rotación doble derecha-derecha (ilustrada en la figura 6.11), y si son ambos hijos izquierdos, una rotación doble izquierda-izquierda que es la misma pero "por espejo". En el caso que otro de v y u es un hijo izquierdo y el otro derecho, elegimos una de las rotaciones dobles de la 6.8.

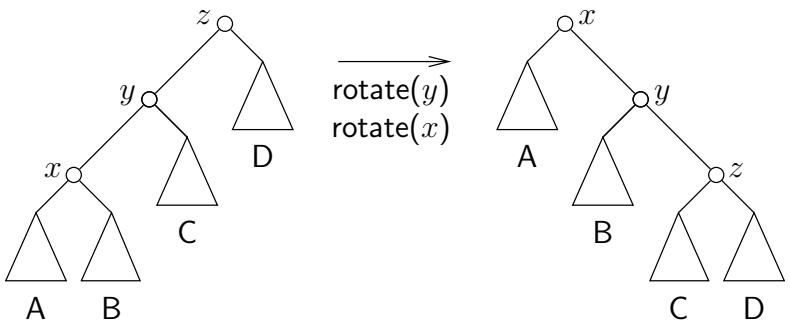

Figura 6.11: Una rotación doble derecha-derecha.

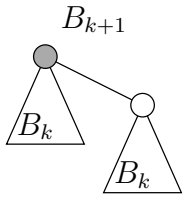

Figura 6.12: La definición recursiva de un árbol binómico B_{k+1} en términos de dos copias de B_k. La raíz de B_{k+1} es el vértice gris.

6.4. Montículos

Un *montículo* (inglés: heap) es una estructura compuesta por árboles. Existen muchas variaciones de montículos. La implementación típica de un montículo se basa de árboles, mientras árboles se puede guardar en arreglos. Entonces, las implementaciones se basan en arreglos o el uso de elementos enlazados.

6.4.1. Montículos binómicos

Un *montículo binómico* se define de una manera recursiva. Un *árbol binómico* de un vértice es B_0 y el único vértice es la raíz. El árbol B_{k+1} contiene dos copias de B_k tales que la raíz de una copia es la raíz de B_{k+1} y la raíz de la otra copia es un hijo directo de esta raíz (ver figura 6.12). Algunas consecuencias de esta definición son que

1. El árbol B_k contiene 2^k vértices.
2. La altura de B_k es k.
3. La raíz de B_k tiene k hijos directos $B_{k-1}, B_{k-2}, \ldots, B_1, B_0$.

Las claves están guardadas en un árbol binómico según el *orden de montículo*: la clave del vértice padre es menor que la clave de su hijo. Cada vértice contiene una clave. Un *montículo binómico* es un conjunto de árboles binómicos tal que no hay duplicados de los B_k. Si el número de claves para guardar es n, tomamos la representación binaria de n,

$$n = b_k b_{k-1} \ldots b_2 b_1 b_0, \tag{6.18}$$

donde los b_i son los bits y k es el número mínimo de bits requeridos para representar n. Si $b_i = 1$, en el montículo está presente un árbol binómico B_i, y si $b_i = 0$, ese tamaño de árbol no forma parte del montículo. Por definición, el largo del número binario es $\log_2 n$, o sea $\mathcal{O}(\log n)$. Por seguir el orden de montículo en cada árbol, se puede encontrar la

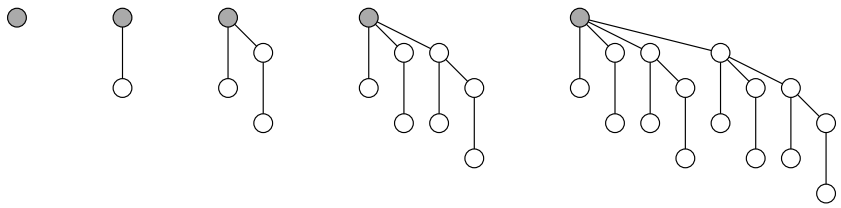

Figura 6.13: Un ejemplo de un montículo binómico, compuesto por cinco árboles binómicos.

clave mínima del montículo en tiempo $\mathcal{O}(\log|n)$. La figura 6.13 muestra un ejemplo de un montículo binómico compuesto por cinco árboles.

Cada vértice del montículo tiene guardado además de su propia clave, su grado (en este contexto: el número de hijos que tiene) y tres *punteros*: a su padre, a su hermano y a su hijo directo. La operación de *encadenación* forma de dos árboles B_n un árbol B_{n+1} tal que el árbol B_n con clave mayor a su raíz será un ramo del otro árbol B_n. Esta operación se realiza en tiempo $\mathcal{O}(1)$.

Para *unir* dos montículos binómicos, hay que recorrer las listas de raíces de los dos montículos simultaneamente. Al encontrar dos árboles del mismo tamaño B_i, se los junta a un árbol B_{i+1}. Si uno de los montículos ya cuenta con un B_{i+1} se los junta recursivamente. Si hay dos, uno queda en el montículo final como un árbol independiente mientras en otro se une con el recien creado. Esta operación necesita $\mathcal{O}(\log n)$ tiempo.

Entonces, para *insertar* un elemento, creamos un B_0 en tiempo $\mathcal{O}(1)$ y lo juntamos en el montículo en tiempo $\mathcal{O}(\log n)$, dando una complejidad total de $\mathcal{O}(\log n)$ para la inserción.

Para *eliminar el elemento mínimo*, lo buscamos en tiempo $\mathcal{O}(\log n)$, le quitamos y creamos otro montículo de sus hijos en tiempo $\mathcal{O}(\log n)$. Unimos los dos montículos en tiempo $\mathcal{O}(\log n)$. Entonces el tiempo total para esta operación es también $\mathcal{O}(\log n)$.

Para *disminuir* el valor de una clave, levantamos el vértice correspondiente más cerca de la raíz para no violar el orden del montículo en tiempo $\mathcal{O}(\log n)$.

Para eliminar un elemento cualquiera, primero disminuimos su valor a $-\infty$ en tiempo $\mathcal{O}(\log n)$ y después quitamos el mínimo en tiempo $\mathcal{O}(\log n)$.

6.4.2. Montículos de Fibonacci

Un *montículo de Fibonacci* es una colección de árboles (no binómicos) que respetan el orden de montículo con las claves. Cada vértice contiene, además de su clave, punteros a su padre, sus dos hermanos (a la izquierda y a la derecha), el grado y un *marcador*. El marcador del vértice v tiene el valor uno si el vértice v ha perdido un hijo después de la

última vez que volvió a ser un hijo de otro vértice el vértice v mismo. Las raíces están en una cadena a través de los punteros a los hermanos, formando una lista doblemente enlazada. La lista se completa por hacer que el último vértice sea el hermano izquierdo del primero.

De nuevo, por el orden de montículo y el manejo del número de árboles, el elemento mínimo se encuentra en tiempo $\mathcal{O}(1)$. Para insertar un elemento, se crea un árbol con un sólo vértice con marcador en cero (en tiempo $\mathcal{O}(1)$). La unión de dos listas de raíces se logra simplemente por modificar los punteros de hermanos para lograr que sean una lista después de la otra (en tiempo $\mathcal{O}(1)$ — son cuatro punteros que serán modificados).

Para eliminar el elemento mínimo, simplemente se lo quita de la lista de raíces y adjuntando la lista de los hijos del vértice eliminado en el montículo como si estuviera otro montículo. Después de la eliminación, se repite una operación de *compresión* hasta que las raíces tengan grados únicos. Aquí la dificultad viene de encontrar eficientemente las raíces con grados iguales para juntarlos. Se logra por construir un arreglo auxiliar mientras recorriendo la lista de raíces: el arreglo tiene el tamaño de grados posibles y se guarda en el elemento i una raíz encontrado con grado i. Si el elemento ya está ocupado, se junta los dos árboles y libera el elemento del arreglo. Esto claramente hay que aplicar recursivamente. Es importante notar que después de haber compresionado la lista de raíces, el número de raíces es $\mathcal{O}(\log n)$ porque todas las raíces tienen una cantidad diferente de hijos y ninguna raíz puede tener más que $\mathcal{O}(\log n)$ hijos.

Para disminuir el valor de una clave, hay dos opciones: si está en una raíz, es trivial (solamente modificando su valor). En el otro caso, habrá que quitar el vértice v con la clave de la lista de hermanos y convertirlo a una raíz nueva. Si el padre w de v llevaba el valor uno en su marcador, también se quita w y lo convierte en una raíz nueva. En el otro caso, se marca el vértice w (por poner el valor uno en el marcador). Lo de convertir vértices en raíces habrá que hacer recursivamente hasta llegar a un padre con el valor de marcador en cero.

En un *montículo binómico*, el número de hijos por vértice es al máximo $\mathcal{O}(\log n)$ donde n es la cantidad total de vértices en el montículo. En el peor caso, los vértices forman un sólo árbol, donde la raíz tiene muchos hijos y no hay muchos otros vértices.

Denotamos por B_k el árbol binómico mínimo donde la raíz tenga k hijos. Los hijos son $B_0, \ldots, B_{k-1}$ y B_i tiene i hijos. Aplica que

$$|B_k| = 1 + \sum_{i=0}^{k-1} |B_i| = 2^k. \tag{6.19}$$

Entonces, $\log_2 |B_k| = \log_2 2^k = k \log_2 2 = k$, que nos da el resultado deseado de $\mathcal{O}(\log n)$.

Mostramos de la misma manera que en un montículo Fibonacci un vértice tiene al máximo $\mathcal{O}(\log n)$ hijos. En el peor caso, todos los vértices están en un sólo árbol y

una cantidad máxima de los vértices son hijos directos de la raíz. Lo único que hay que establecer es que los ramos de los hijos de la raíz tienen que ser grandes. Así establecemos que solamente unos pocos vértices (en comparación con el número total n) pueden ser hijos de la raíz.

Sea $h \geq 0$ y F_h un árbol de un montículo Fibonacci donde la raíz v tiene h hijos, pero donde la cantidad de otros vértices es mínima. Al momento de juntar el hijo H_i, ambos vértices v y H_i tuvieron exactamente $i-1$ hijos. Después de esto, para que tenga la cantidad mínima de hijos, H_i ha necesariamente perdido un hijo; si hubiera perdido más que uno, H_i habría sido movido a la lista raíz. Entonces, H_i tiene $i-2$ hijos y cada uno de ellos tiene la cantidad mínima posible de hijos. Entonces, H_i es la raíz de un F_{i-2}. Entonces, los hijos de un F_r son las raíces de $F_0, F_1, \ldots, F_{r-2}$ y además un hijo tipo hoja que no tiene hijos. En total, la cantidad total de vértices es

$$|F_r| = \begin{cases} 1, & \text{si } r = 0, \quad \text{(solamente la raíz)} \\ 2 + \sum_{i=0}^{r-2} |F_i|, & \text{si } r > 0, \quad \text{la raíz, la hoja y los ramos.} \end{cases} \tag{6.20}$$

Abrimos la recursión:

$$\begin{aligned} |F_0| &= 1 \\ |F_1| &= 2 \\ |F_2| &= 2 + |F_0| = |F_1| + |F_0| \\ |F_3| &= 2 + |F_0| + |F_1| = |F_2| + |F_1| \\ |F_4| &= 2 + |F_0| + |F_1| + |F_2| = |F_3| + |F_2| \\ &\vdots \\ |F_r| &= |F_{r-1}| + |F_{r-2}|\,. \end{aligned} \tag{6.21}$$

Resulta que el número de vértices en F_r es el número de Fibonacci $\mathcal{F}(r+2)$ (definido en la ecuación 6.10 en página 98). Utilizamos de nuevo la cota inferior de ecuación 6.11:

$$|F_r| = \mathcal{F}(r+2) \geq \left(\frac{1+\sqrt{5}}{2}\right)^{r+1}. \tag{6.22}$$

Con las propiedades de logaritmo, llegamos a $\log n = \log F_r \geq (r+1)\log C$ donde C es alguna constante, por lo cual tenemos $r = \mathcal{O}(\log n)$.

6.5. Grafos

Para guardar un grafo de n vértices etiquetados $1, 2, 3, \ldots, n$, se puede guardar la matriz de adyacencia solamente es eficiente si el grafo es muy denso, porque la matriz ocupa n^2 elementos y si m es mucho menor que n^2, la mayoría del espacio reservado tiene el valor cero. Para ahorrar espacio, se puede utilizar *listas de adyacencia*. Se necesita un arreglo $a[]$, cada elemento de cuál es una lista de largo dinámico. La lista de $a[i]$ contiene las etiquetas de cada uno de los vecinos del vértice i. El tamaño de la estructura de listas de adyacencia es $\mathcal{O}(n+m) \leq \mathcal{O}(m) = \mathcal{O}(n^2)$. Esto es muy parecido al implementar un grafo como una estructura enlazada, con punteros a todos los vecinos de cada vértice.

6.5.1. Búsqueda y recorrido en grafos

Los algoritmos de procesamiento de grafos comúnmente utilizan colas. Los dos ejemplos más fundamentales son los algoritmos de *búsqueda* y *recorrido*. El recorrido de un grafo (o un árbol) es el proceso de aplicación de un método sistemático para visitar cada vértice del grafo (o árbol). Un algoritmo de búsqueda es un método sistemático para recorrer un grafo de entrada $G = (V, E)$ con el propósito de encontrar un vértice del G que tenga una cierta propiedad.

Algorítmicamente cada algoritmo de búsqueda realiza un recorrido en el caso que visita todos los vértices sin encontrar solución. Entonces, el mismo pseudocódigo sirve para los dos usos con modificaciones muy pequeñas. Los usos típicos de algoritmos de recorrido incluyen

- la construcción de caminos
- la computación distancias
- la detección de ciclos
- la identificación de los componentes conexos

Búsqueda en profundidad (DFS)

Dado G y un vértice inicial $v \in V$, el procedimiento general es el siguiente ($\mathcal{L}$ es una pila):

1. crea una pila vacía $\mathcal{L}$
2. asigna $u := v$
3. marca u visitado

4. añade *cada* vértice *no marcado* en $\Gamma(v)$ al **comienzo** de $\mathcal{L}$
5. quita del comienzo de $\mathcal{L}$ todos los vértices marcados
6. si $\mathcal{L}$ está vacía, termina
7. asigna $u :=$ el *primer* vértice en $\mathcal{L}$
8. quita el primer vértice de $\mathcal{L}$
9. continua de paso (3)

El algoritmo $\textsc{Dfs}$ puede progresar en varias maneras; el orden de visitas depende de cómo se elige a cuál vecino se va. Algunas aristas encontradas apuntan a vértices ya visitados.

También se puede formular $\textsc{Dfs}$ como un algoritmo recursivo: dado un vértice de inicio, el grafo $G = (V, E)$ y un conjunto $\mathcal{L}$ de vértices ya visitados (inicialmente $\mathcal{L} := \emptyset$), basta con definir

$$\begin{array}{l} \mathsf{dfs}(v, G, \mathcal{L})\,\{ \\ \qquad \mathcal{L} := \mathcal{L} \cup \{v\} \\ \qquad \text{para todo } w \in \Gamma(v) \setminus \mathcal{L}: \\ \qquad\qquad \mathsf{dfs}(w, G, \mathcal{L}). \\ \} \end{array} \tag{6.23}$$

Si el algoritmo $\textsc{Dfs}$ se utiliza para realizar algún procesamiento en los vértices, con la implementación recursiva se puede fácilmente variar en qué momento "realizar la visita": las opciones son "visitar" antes o después de llamar la subrutina dfs para los vecinos. El orden de visitas a los vértices si la visita se realiza antes de la llamada recursiva se llama el *preorden* y el orden de visitas en el caso de visitar después se llama *postorden*.

En las dos implementaciones, la complejidad asintótica de $\textsc{Dfs}$ es $\mathcal{O}(n + m)$: cada arista está procesada por máximo una vez "de ida" y otra "de vuelta". Cada vértice solamente se procesa una vez, porque los vértices "ya marcados" no serán revisitados.

$\textsc{Dfs}$ produce una clasificación en las aristas $\{v, w\}$ del grafo: las *aristas de árbol* son las aristas por las cuales progresa el procedimiento, es decir, en la formulación recursiva, w fue visitado por una llamada de v o vice versa. Estas aristas forman un árbol cubriente del componente conexo del vértice de inicio. Depende de la manera en que se ordena los vecinos que habrá que visitar cuáles aristan serán aristas de árbol. Un vértice v es el *padre* (directo o inmediato) de otro vértice w si v lanzó la llamada recursiva para visitar a w. Si v es el padre de w, w es *hijo* de v. Cada vértice, salvo que el vértice de inicio,

que se llama la *raíz* del árbol, tiene un vértice padre único. El número de hijos que tiene un vértice puede variar.

Un vértice v es un *antepasado* de otro vértice w si existe una sucesión de vértices $v = u_1, u_2, \ldots, u_k = w$ tal que u_i es el padre de u_{i+1}. En ese caso, w es un *descendiente* de v. Si $k = 2$, o sea, v es el padre directo de w, v es el antepasado inmediato de w y w es un descendiente inmediato de v. La raíz es un antepasado de todos los otros vértices. Los vértices sin descendientes son *hojas*.

Las aristas que no son aristas de ábrol se clasifica en tres clases:

(I) una *arista procedente* conectan un antepasado a un descendiente *no inmediato*,

(II) una *arista retrocedente* conecta un descendiente a un antepasado *no inmediato*,

(III) una *arista transversa* conecta un vértice a otro tal que no son ni antepasados ni descendientes uno al otro — están de diferentes *ramos* del árbol.

Nota que aristas procedentes y aristas retrocedentes son la misma clase en un grafo no dirigido.

El *nivel* de un vértice v tiene la siguiente definición recursiva:

$$\text{nivel}(v) = \begin{cases} 0, & \text{si } v \text{ es la raíz,} \\ \text{nivel}(u) + 1, & \text{si } u \text{ es el padre de } v. \end{cases} \tag{6.24}$$

La definición del *altura* es similar, pero en la otra dirección:

$$\text{altura}(v) = \begin{cases} 0, & \text{si } v \text{ es una hoja,} \\ \max\{\text{altura}(u) + 1\}, & \text{si } u \text{ es un hijo de } v. \end{cases} \tag{6.25}$$

El subárbol de v es el árbol que es un subgrafo del árbol cubriente donde v es la raíz; es decir, solamente vértices que son descendientes de v están incluidos además de v mismo.

Búsqueda en anchura

La *búsqueda en anchura* utiliza una cola $\mathcal{L}$. Dado G y un vértice inicial $v \in V$, el procedimiento general es el siguiente:

1. crea una cola vacía $\mathcal{L}$
2. asigna $u := v$

3. marca u visitado
4. añade *cada* vértice *no marcado* en $\Gamma(v)$ al **fin** de $\mathcal{L}$
5. si $\mathcal{L}$ está vacía, termina
6. asigna $u :=$ el *primer* vértice en $\mathcal{L}$
7. quita el primer vértice de $\mathcal{L}$
8. continua de paso (3)

Algunas aristas encontradas apuntan a vértices ya visitados.

6.5.2. Componentes conexos

Se puede utilizar la búsqueda en profundidad para determinar los componentes conexos de un grafo. Como ya definido en sección 1.8, un grafo es conexo si cada vértice está conectado a todos los vértices por un camino. Por iniciar DFS en el vértice v, el conjunto de vértices visitados por el recorrido corresponde al componente conexo de v, porque el algoritmo efectivamente explora todos los caminos que pasan por v. Si el grafo tiene vértices que no pertenecen al componente de v, elegimos uno de esos vértices u y corremos DFS desde u. Así encontramos el componente conexo que contiene a u. Iterando así hasta que todos los vértices han sido clasificados a un componente, se logra determinar todos los componentes conexos.

El número de vértices siendo n, repetimos DFS por máximo $\mathcal{O}(n)$ veces y cada recorrido toma tiempo $\mathcal{O}(n+m)$. Sin embargo, cada recorrido DFS ve valores menores de n y m porque no vuelve a procesar nada ya marcado, y por definiciones, desde un componente conexo no hay ninguna arista a otro componente conexo. Entonces, efectivamente procesamos cada vértice y cada arista exactamente una vez. Considerando que

$$n \in \mathcal{O}(m) \in \mathcal{O}(n+m) \in \mathcal{O}\left(n^2\right), \tag{6.26}$$

tenemos un algoritmo para identificar los componentes conexos en tiempo $\mathcal{O}(n^2)$.

En un grafo conexo, podemos clasificar las aristas según un recorrido DFS: asignamos al vértice inicial la etiqueta "uno". Siempre al visitar a un vértice por la primera vez, le asignamos una etiqueta numérica uno mayor que la última etiqueta asignada. Así todos los vértices llegan a tener etiquetas únicas en $[1, n]$. Llamamos la etiqueta así obtenida "el número de inicio" del vértice y lo denotamos por $I(v)$.

Asignamos otra etiqueta a cada vértice tal que la asignación ocurre cuando todos los vecinos han sido recurridos, empezando de 1. Así el vértice de inicio tendrá la etiqueta n. Estas etiquetas se llaman "números de final" de los vértices y son denotados por $F(v)$.

Las $I(v)$ definen el orden previo (inglés: preorder) del recorrido y las $F(v)$ el orden posterior (inglés: postorder). Una arista $\{v, u\}$ es

- una arista de árbol si y sólo si el recorrido llegó a u directamente desde v,
- una arista retrocedente si y sólo si $(I(u) > I(v)) \wedge (F(u) < F(v))$,
- una arista transversa si y sólo si $(I(u) > I(v)) \wedge (F(u) > F(v))$, y
- una arista procedente si y sólo si en el recorrido v es un antepasado de u.

De nuevo hay que tomar en cuanta que en grafos no dirigidos, las aristas retrocedentes son también procedentes y vice versa.

Componentes doblemente conexos

Un grafo no dirigido es k-conexo si de cada vértice hay por lo menos k caminos *distintos* a cada otro vértice. El requisito de ser distinto puede ser de parte de los vértices tal que no pueden pasar por los mismos vértices ningunos de los k caminos (inglés: vertex connectivity) o de las aristas tal que no pueden compartir ninguna arista los caminos (inglés: edge connectivity).

En el sentido de vértices distintos, aplica que la intersección de dos componentes distintos que son ambos 2-conexos consiste por máximo un vértice. Tal vértice se llama un *vértice de articulación*. Utilizamos esta definición para llegar a un algoritmo para encontrar los componentes 2-conexos de un grafo no dirigido.

Sea v un vértice de articulación y tengamos un bosque de extensión del grafo (o sea, un árbol cubriente de cada componente conexo — el mismo $\textsc{Dfs}$ que identifica los componentes conexos simples genera tal árbol). Si v es la raíz de un árbol cubriente, tiene necesariamente por lo menos dos hijos, porque por definición de los vértices de articulación, un recorrido no puede pasar de algún componente a otro sin pasar por v.

Si v no es la raíz, por lo menos un ramo de v contiene un componente doblemente conexo (o sea, 2-conexo). De tal ramo no es posible tener una arista retrocedente a ningún antepasado.

Hacemos un recorrido y marcamos para cada vértice que tan cerca de la raíz se puede llegar solamente por las aristas de árbol y las aristas retrocedentes:

$$R(v) = \min\left\{\{I(v)\} \cup \left\{I(u) \mid \wedge \begin{array}{l} u \text{ es un antepasado de } v \\ \\ v \text{ o un descendiente suyo tiene arista con } u \end{array}\right\}\right\} \tag{6.27}$$

Cuadro 6.2: Procedimiento de inicialización.

inicio $:= 0$;
$\mathcal{P} := \emptyset$;
para todo $v \in V$;
  $I(v) := 0$;
para todo $v \in V$;
  si $I(v) = 0$
  doblemente-conexo(v)

Un vértice v que *no* es la raíz es un vértice de articulación si y sólo si tiene un hijo u tal que $R(u) \geq I(v)$. Una definición recursiva de $R(v)$ es

$$R(v) = \text{mín} \left\{ \cup \begin{array}{l} \{I(v)\} \cup \{R(u) \mid u \text{ es un hijo de } v\} \\ \{I(u) \mid \{v, u\} \text{ es una arista retrocedente}\} \end{array} \right\}. \tag{6.28}$$

De hecho, podemos incorporar en el DFS original la calculación de los $R(v)$ de todos los vértices.

Para facilitar la implementación, se puede guardad aristas en una pila $\mathcal{P}$ al procesarlos. Cuando el algoritmo está "de vuelta" y encuentra un componente doblemente conexo, las aristas del componente están encima de la pila y se puede quitarlas con facilidad. El procedimiento de inicialización será la del cuadro 6.2 y el procedimiento recursivo que realiza el algoritmo la del cuadro 6.3.

El algoritmo visita cada vértice y recorre cada arista. El tiempo de procesamiento de un vértice o una arista es constante, $\mathcal{O}(1)$. Entonces la complejidad del algoritmo completo es $\mathcal{O}(n + m)$.

Componentes fuertemente conexos

Un grafo dirigido está *fuertemente conexo* si de cada uno de sus vértices existe un camino dirigido a cada otro vértice. Sus *componentes fuertemente conexos* son los subgrafos maximales fuertemente conexos. La figura 6.14 muestra un ejemplo.

Los componentes fuertemente conexos de $G = (V, E)$ determinan una partición de los vértices de G a las clases de equivalencia según la relación de clausura reflexiva y transitiva de la relación de aristas E. Las aristas entre los componentes fuertemente conexos determinan una orden parcial en el conjunto de componentes. Ese orden parcial se puede aumentar a un orden lineal por un algoritmo de *ordenación topológica* .

Cuadro 6.3: Procedimiento recursivo.

```
procedimiento doblemente-conexo(v)
  inicio := inicio +1;
  I(v) := inicio;
  R(v) := I(v)
  para cada {v, u} ∈ E
  si I(u) = 0
    añade {v, u} en P;
    padre(u) := v;
    doblemente-conexo(u);
    si R(u) ≥ I(v)
      elimina aristas de P hasta e inclusa {v, v'};
    R(v) := mín{R(v), R(u)};
  en otro caso si u ≠ padre(v);
    R(v) := mín{R(v), I(u)}
```

Cuando uno realiza un recorrido en profundidad, los vértices de un componente conexo se quedan en el mismo ramo del árbol cubriente. El vértice que queda como la raíz del ramo se dice la raíz del componente. La meta de ordenación topológica es encontrar las raíces de los componentes según el orden de sus números $F(v)$. Al llegar a una raíz v_i, su componente está formado por los vértices que fueron visitados en el ramo de v_i pero no fueron clasificados a ninguna raíz anterior $v_1, \ldots, v_{i-1}$. Esto se puede implementar fácilmente con una pila auxiliar $\mathcal{P}$, empujando los vértices en la pila en el orden del DFS y al llegar a una raíz, quitándolos del encima de la pila hasta llegar a la raíz misma. Así nada más el componente está eliminado de la pila.

Lo que queda es saber identificar las raíces. Si el grafo contiene solamente aristas de árbol,

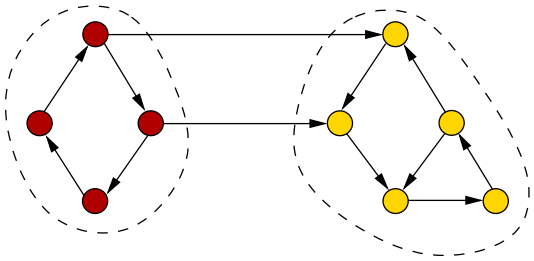

Figura 6.14: Un grafo dirigido pequeño con dos componentes fuertemente conexos.

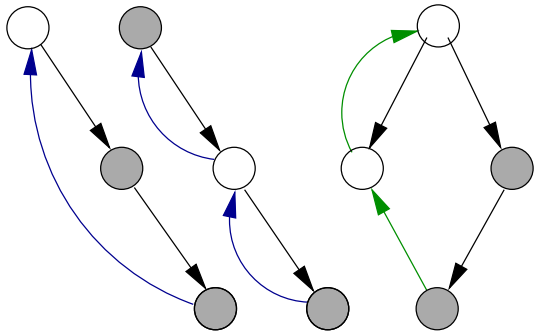

Figura 6.15: Situaciones en las cuales dos vértices (en gris) pueden pertenecer en el mismo componente fuertemente conexo; las aristas azules son retrocedentes y las aristas verdes transversas. Las aristas de árbol están dibujados en negro y otros vértices (del mismo componente) en blanco.

cada vértice forma su propio componente. Las aristas procedentes no tienen ningún efecto en los componentes. Para que una vértice pueda pertenecer en el mismo componente con otro vértice, tienen que ser conectados por un camino que contiene aristas retrocedentes o transversas (ver figura 6.15).

Para buscar las raíces, utilizamos un arreglo auxiliar $\mathcal{A}(v)$ para guadrar un número para cáda vértice encontrado (ver figura 6.16 para una ilustración):

$$\mathcal{A}(v) = \text{mín} \left\{ \{I(v)\} \cup \left\{ I(v') \mid \begin{array}{l} \exists w \text{ que es descendente de } v \text{ tal que} \\ \wedge \quad \{w,u\} \text{ es retrocedente o transversa} \\ \qquad \text{la raíz del componente de } u \text{ es un antepasado de } v \end{array} \right\} \right\} \tag{6.29}$$

Entonces, si $\mathcal{A}(v) = I(v)$, sabemos que v es una raíz de un componente. Para todo otro vértice aplica que $\mathcal{A}(v) < I(v)$, porque $\mathcal{A}(v)$ contiene el valor del $I(v)$ de un vértice anteriormente recorrido por una arista retrocedente o transversa o alternativamente un valor de $\mathcal{A}(v)$ está pasado a v por un descendiente. También podemos formular $\mathcal{A}(v)$ en forma recursiva:

$$\begin{array}{rll} \mathcal{A}(v) = & \text{mín}\{ & \{I(v)\} \\ & \cup & \{\mathcal{A}(u) \mid u \text{ es hijo de } v\} \\ & \cup & \{I(u) \mid \{v,u\} \text{ es retrocedente o transversa} \\ & & \quad \wedge \text{ la raíz del componente de } u \text{ es un antepasado de } v\}\}\,. \end{array} \tag{6.30}$$

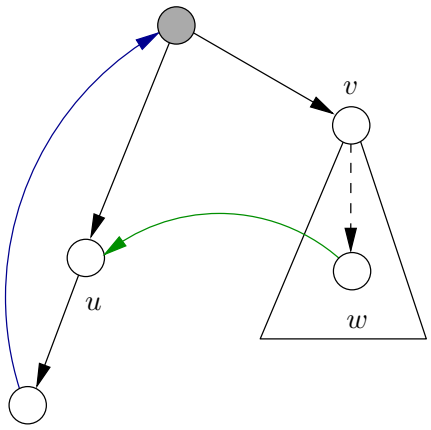

Figura 6.16: La situación de la ecuación 6.29. La raíz del componente está dibujado en gris y la flecha no continua es un camino, no necesariamente una arista directa.

Durante la ejecución el algoritmo, en la pila auxiliar $\mathcal{P}$ habrá vértices los componentes de los cuales no han sido determinados todavía. Para facilitar el procesamiento, mantenemos un arreglo de indicadores: $\texttt{guardado}(v) =$ **verdadero** si v está en la pila auxiliar y **falso** en otro caso. Así podemos determinar en el momento de procesar una arista retrocedente o transversa $\{v, u\}$ si la raíz de u es un antepasado de v. Si lo es, u está en el mismo componente con v y los dos vértices v y la raíz están todavía en la pila. El algoritmo en pseudo-código está en el cuadro 6.4

En el algoritmo del cuadro 6.4, cada vértice entra la pila $\mathcal{P}$ una vez y sale de la pila una vez. Adicionamente hay que procesar cada arista. La computación por vértice o arista es de tiempo constante. Entonces tenemos un algoritmo con tiempo de ejecución $\mathcal{O}(n + m)$.

6.6. Tablas de dispersión dinámicas

Una *tabla de dispersión* (inglés: hash table) son estructuras de datos que asocien *claves* con *valores*. Por ejemplo, se podría implementar una guía telefónica por guardar los números (los valores) bajo los nombres (las claves). La idea es reservar primero espacio en la memoria de la computadora y después alocarlo a la información insertada, de tal manera que siempre será rápido obtener la información que corresponde a una clave dada.

Una *función de dispersión* (también: una función hash) es una función del conjunto de claves posibles a las direcciones de memoria. Su implementación típica es por arreglos unidimensionales, aunque existen también variantes multidimensionales.

El tiempo de acceso promedio es constante por diseño: hay que computar la función hash y buscar en la posición indicada por la función. En el caso que la posición indicada ya está ocupada, se puede por ejemplo asignar el elemento al espacio libre siguiente, en cual caso el proceso de búsqueda cambia un poco: para ver si está o no una clave, hay que ir a la posición dada por la función hash y avancar desde allá hasta la clave o en su ausencia hasta llegar a un espacio no usado.

Cuando toda la memoria reservada ya está siendo utilizada (o alternativamente cuando el porcentaje utilizado supera una cota pre-establecida), hay que reservar más. Típicamente se reserva una área *de tamaño doble* de lo anterior. Primero se copia todas las claves existentes con sus valores en la área nueva, después de que se puede empezar a añadir claves nuevas. Entonces, "casi todas" las operaciones de inserción son fáciles (de tiempo constante), pero el caso peor es $\Omega(n)$ para una inserción y $\Omega(n^2)$ para un total de n inserciones.

6.7. Colas de prioridad

Una *cola de prioridad* es una estructura para guardar elementos con claves asociadas tal que el valor de la clave representa la "prioridad" del elemento. El menor valor corresponde a la prioridad más urgente.

La operaciones de *colas de prioridad de adjunto* están enfocadas a lograr fácilmente averiguar el elemento mínimo guardado (en el sentido de los valores de las claves), o sea, el elemento más importante. Este implica que es necesario que tengan un orden los elementos procesados por la cola de prioridad. Las operaciones son

1. insertar un elemento,
2. consultar el elemento mínimo,
3. retirar el elemento mínimo,
4. reducir el valor de un elemento,
5. juntar dos colas.

También es posible utilizar el mismo dato como la clave en las aplicaciones donde solamente es de interés tener acceso al elemento mínimo pero el concepto de prioridad no aplica. Las colas de prioridad se puede implementar con montículos.

6.8. Conjuntos

6.8.1. Estructuras unir-encontrar

Una estructura *unir-encontrar* (inglés: union-find) sirve para manejar un grupo C de conjuntos distintos de elementos tales que cada elemento tiene un nombre único. Sus operaciones básicas son

- `form`(i, S) que forma un conjunto $S = \{i\}$ y lo añade en C: $C := C \cup \{S\}$; no está permitido que el elemento i pertenezca a ningún otro conjunto de la estructura,
- `find`(i) que devuelva el conjunto $S \in C$ de la estructura donde $i \in S$ y reporta un error si i no está incluido en ningún conjunto guardado,
- `union`(S, T, U) que junta los dos conjuntos $S \in C$ y $T \in C$ en un sólo conjunto $U = S \cup T$ y actualiza $C := (C \setminus \{S, T\}) \cup \{U\}$; recuerda que por definición $S \cap T = \emptyset$.

No es realmente necesario asignar nombres a los conjuntos, porque cada elemento pertenece a un sólo conjunto en C. Entonces, se puede definir las operaciones también en la manera siguiente:

- `form`(i): $C := C \cup \{i\}$,
- `find`(i): devuelva la lista de elementos que estén en el mismo conjunto con i,
- `union`(i, j): une el conjunto en el cual pertenece i con el conjunto en el cual pertenece j.

6.8.2. Implementación de un conjunto en forma de un árbol

Otra opción de manejar conjuntos es a través de árboles: cada elemento está representada por un vértice hoja del árbol. El vértice padre de unas hojas representa el conjunto de los elementos representadas por sus vértices hijos. Un subconjunto es un vértice intermedio, el padre de cual es su superconjunto. El conjunto de todos los elementos es su propio padre. La estructura de los punteros padre está mostrada en el ejemplo de la figura 6.17

La operación de crear un árbol nuevo es fácil: crear el primer vértice v. Marcamos su clave (única) con c. Hay que asignar que sea su propio padre: $\mathcal{P}(c) := c$. Esto toma tiempo constante $\mathcal{O}(1)$.

La operación de búsqueda del conjunto a cual pertenece una clave c, habrá que recorrir el árbol (o sea, el arreglo) desde le vértice con la clave deseada:

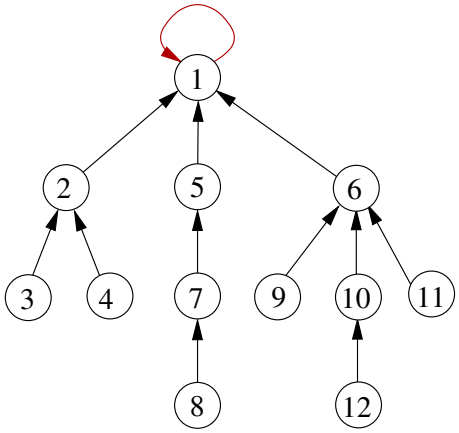

Figura 6.17: Un ejemplo de los punteros padre de una árbol con n claves posibles para poder guardar su estructura en un arreglo de tamaño n.

$p := c;$
mientras $\mathcal{P}(p) \neq p$
  $p := \mathcal{P}(p);$
devuelve $p;$

En el peor caso, el arrego se ha degenerado a una lista, por lo cual tenemos complejidad asintótica $\mathcal{O}(n)$.

Para juntar dos conjuntos, basta con hacer que la raíz de una (con clave c_1) sea un hijo de la raíz de la otra (con clave c_2): $\mathcal{P}(c_1) := c_2$. Esta operación necesita tiempo $\mathcal{O}(1)$.

Podemos considerar la complejidad de una *sucesión de operaciones*. Por ejemplo, si creamos n conjuntos, entre cuales ejecutamos por máximo $n-1$ operaciones de unir dos conjuntos y no más de dos búsquedas de conjuntos por cada unión, el tiempo total es $\Theta(n + n - 1 + 2(n-1)n) = \Theta(n^2)$.

Si aseguramos que el juntar dos árboles, el árbol de menor tamaño será hecho hijo del otro, se puede demostrar que la altura del árbol que resulta es $\mathcal{O}(\log n)$, por lo cual podemos asegurar que la operación de búsqueda del conjunto toma tiempo $\mathcal{O}(\log n)$ y la secuencia analizada sería de complejidad asintótica $\mathcal{O}(n \log n)$, que ya es mejor. Lo único que hay que hacer es guardar en un arreglo auxiliar el tamaño de cada conjunto y actualizarlo al unir conjuntos y al generar un nuevo conjunto.

Aún mejor sería guardar información sobre la altura de cada árbol, la altura siendo un número menor o igual al tamaño. La ventaja de la complejidad asintótica queda igual, pero solamente necesitamos $\mathcal{O}(\log \log n)$ bits para guardar la altura.

Hay diferentes opciones para modificar los caminos cuando uno realiza una búsqueda de conjunto en la estructura.

- Condensación del camino: traer los vértices intermedios a la raíz

$p := c$;
mientras $\mathcal{P}(p) \neq p$
  $p := \mathcal{P}(p)$;
$q := i$;
mientras $\mathcal{P}(q) \neq q$
  $r := \mathcal{P}(q)$; $\mathcal{P}(q) := p$;
  $q := r$
devuelve p;

- División del camino: mover vértices intermediados a otra parte

$p := c$;
mientras $\mathcal{P}(\mathcal{P}(p)) \neq \mathcal{P}(p)$
  $q := \mathcal{P}(p)$;
  $\mathcal{P}(p) := \mathcal{P}(\mathcal{P}(p))$;
  $p := q$
devuelve y;

- Cortar el camino a su mitad: saltando a abuelos

$p := c$;
mientras $\mathcal{P}(\mathcal{P}(p)) \neq \mathcal{P}(p)$
  $\mathcal{P}(p) := \mathcal{P}(\mathcal{P}(p))$;
  $y := \mathcal{P}(p)$
devuelve $\mathcal{P}(p)$;

Cuadro 6.4: Un algoritmo basado en DFS para determinar los componentes fuertemente conexos de un grafo dirigido. Se supone contar con acceso global al grafo y las estructuras auxiliares.

```
procedimiento fuerteconexo(v);
  a := a + 1;
  I(v) := a;
  A(v) := I(v);
  empuja v en P;
  guardado[v] := verdadero ;
  para todo {v, u} ∈ E haz
  si I(u) = 0
    fuerteconexo(u);
    A(v) := mín{A(v), A(u)};
  en otro caso
  si (I(u) < I(v) ∧ guardado(u) = verdadero )
    A(v) := mín{A(v), I(u)};
  si A(v) = I(v)
    quita de P los elementos hasta e incluso v,
      guardado(w) := falso ;
      imprime w como parte del componente de v;


procedimiento main(V, E)
  a := 0;
  I(v) := 0;
  P := ∅;
  para todo v ∈ V haz
    I(v) := 0;
    guardado(v) := falso
  para todo v ∈ V
  si I(v) = 0
    fuerteconexo(v)
```

Capítulo 7

Análisis de algoritmos

7.1. Algoritmos simples

Para analizar desde un pseudocódigo la complejidad, típicamente se aplica las reglas siguientes:

- Asignación de variables simples toman tiempo $\mathcal{O}(1)$.
- Escribir una salida simple toma tiempo $\mathcal{O}(1)$.
- Leer una entrada simple toma tiempo $\mathcal{O}(1)$.
- Si las complejidades de una sucesión de instrucciones $I_1, I_2, \ldots, I_k$ donde k *no depende* del tamaño de la instancia, son respectivamente $f_1, f_2, \ldots, f_k$, la complejidad total de la sucesión es

$$\mathcal{O}(f_1 + f_2 + \ldots + f_k) = \mathcal{O}(\max\{f_1, \ldots, f_k\}). \tag{7.1}$$

- La complejidad de una cláusula de condición (**si**) es la suma del tiempo de evaluar la condición y la complejidad de la alternativa ejecutada.
- La complejidad de una repetición (**mientras** , **para** , ...) es $\mathcal{O}(k(f_t + f_o))$, donde k es el número de veces que se repite, f_t es la complejidad de evaluar la condición de terminar y f_o la complejidad de la sucesión de operaciones de las cuales consiste una repetición.
- La complejidad de tiempo de una *llamada de subrutina* es la suma del tiempo de calcular sus parámetros, el tiempo de asignación de los parámetros y el tiempo de ejecución de las instrucciónes.
- Operaciones aritméticas y asignaciones que procesan arreglos o conjuntos tienen complejidad lineal en el tamao del arreglo o conjunto.

7.2. Complejidad de algoritmos recursivos

La complejidad de programas recursivos típicamente involucra la solución de una ecuación diferencial. El método más simple es adivinar una solución y verificar si está bien la adivinanza. Como un ejemplo, tomamos la ecuación siguiente

$$T(n) \leq \begin{cases} c, & \text{si } n = 1 \\ g\Big(T(n/2), n\Big), & \text{si } n > 1 \end{cases} \tag{7.2}$$

y adivinamos que la solución sea, en forma general, $T(n) \leq f(a_1, \ldots, a_j, n)$, donde $a_1, \ldots, a_j$ son parámetros de la función f. Para mostrar que para algunos valores de los parámetros $a_1, \ldots, a_j$ aplica para todo n que la solución sea la adivinada, tenemos que demostrar que

$$c \leq f(a_1, \ldots, a_j, 1) \tag{7.3}$$

y también que

$$g\left(f\left(a_1, \ldots, a_j, \frac{n}{2}\right), n\right) \leq f(a_1, \ldots, a_j, n), \text{ si } n > 1. \tag{7.4}$$

Esto se logra por inducción. Hay que mostrar que $T(k) \leq f(a_1, \ldots, a_j, k)$ para $1 \leq k < n$. Cuando está establecido, resulta que

$$\begin{aligned} T(n) &\leq g\left(T\left(\tfrac{n}{2}\right), n\right) \\ &\leq g\left(f\left(a_1, \ldots, a_j, \tfrac{n}{2}\right), n\right) \\ &\leq f(a_1, \ldots, a_j, n), \text{ si } n > 1. \end{aligned} \tag{7.5}$$

Otra opcion es aplicar la ecuación recursiva de una manera iterativa. Por ejemplo, con la ecuación diferencial

$$\begin{cases} T(1) &= c_1, \\ T(n) &\leq 2T\left(\tfrac{n}{2}\right) + c_2 n. \end{cases} \tag{7.6}$$

obtenemos por aplicación repetida

$$\begin{aligned} T(n) &\leq 2T\left(\tfrac{n}{2}\right) + c_2 n \leq 2\left(2T(n/4) + \tfrac{c_2 n}{2}\right) + c_2 n \\ &= 4T(n/4) + 2c_2 n \leq 4\left(2T(n/8) + c_2 n/4\right) + 2c_2 n \\ &= 8T(n/8) + 3c_2 n. \end{aligned} \tag{7.7}$$

Observando la forma en que abre la ecuación, podemos adivinar que por lo general aplica

$$T(n) \leq 2^i T\left(\frac{n}{2^i}\right) + ic_2 n \text{ para todo } i. \tag{7.8}$$

Si asumimos que $n = 2^k$, la recursión acaba cuando $i = k$. En ese momento tenemos $T\left(\frac{n}{2^k}\right) = T(1)$, o sea

$$T(n) \leq 2^k T(1) + kc_2 n. \tag{7.9}$$

De la condición $2^k = n$ sabemos que $k = \log n$. Entonces, con $T(1) \leq c_1$ obtenemos

$$T(n) \leq c_1 n + c_2 n \log n \tag{7.10}$$

o sea $T(n) \in \mathcal{O}(n \log n)$.

7.2.1. Solución general de una clase común

En forma más general, tomamos la ecuación siguiente

$$\begin{cases} T(1) &= 1 \\ T(n) &= a(n/b) + d(n), \end{cases} \tag{7.11}$$

donde a y b son constantes y $d : \mathbb{Z}^+ \to \mathbb{R}^+$. Para simplificar la situación, se supone que $n = b^k$ por algún $k \geq 0$ (o sea $k = \log b$). La solución necesariamente tiene la representación siguiente:

$$T(n) = \underbrace{a^k}_{\text{parte homogénica}} + \underbrace{\sum_{j=0}^{k-1} a^j d(b^{k-j})}_{\text{parte heterogénica}}. \tag{7.12}$$

En el caso que $d(n) \equiv 0$, no habrá parte heterogénica. La demostración es por descomposición:

$$\begin{aligned} T(n) &= aT\left(\frac{n}{b}\right) + d(n) \\ &= a\left(aT\left(\frac{n}{b^2}\right) + d\left(\tfrac{n}{b}\right)\right) + d(n) \\ &= a^2T\left(\frac{n}{b^2}\right) + ad\left(\tfrac{n}{b}\right) + d(n) \\ &\vdots \\ &= a^kT\left(\frac{n}{b^k}\right) + a^{k-1}d\left(\frac{n}{b^{k-1}}\right) + \ldots + d(n) \\ &= a^kT(1) + \sum_{j=0}^{k-1} a^j d(b^{k-j}) \\ &= a^k + \sum_{j=0}^{k-1} a^j d(b^{k-j}). \end{aligned} \tag{7.13}$$

Para la parte homogénica a^k aplica que $a^k = a^{\log_b n} = n^{\log_b a}$. En el análisis de ordenación por fusión tenemos $a = b = 2$, en cual caso $a^k = n$ (dejamos el análisis mismo de ordenación por fusión como ejercicio).

En algunos casos la solución es más simple: si d es *multiplicativa*, o sea $d(xy) = d(x)d(y)$, aplica que $d(b^{k-j}) = (d(b))^{k-j}$ que nos permite reformular la parte heterogénica:

$$\sum_{j=0}^{k-1} a^j d(b)^{k-j} = d(b)^k \sum_{j=0}^{k-1} \left(\frac{a}{d(b)}\right)^j = d(b)^k \frac{\left(\frac{a}{d(b)}\right)^k - 1}{\frac{a}{d(b)} - 1} = \frac{a^k - d(b)^k}{\frac{a}{d(b)} - 1}. \tag{7.14}$$

Entonces, cuando d es multiplicativo, tenemos

$$T(n) = \begin{cases} \mathcal{O}\left(n^{\log_b a}\right), & \text{si} \quad a > d(b) \\ \mathcal{O}\left(n^{\log_b d(b)}\right), & \text{si} \quad a < d(b) \\ \mathcal{O}\left(n^{\log_b d(b)} \log_b n\right), & \text{si} \quad a = d(b). \end{cases} \tag{7.15}$$

En especial, si $a < d(b)$ y $d(n) = n^\alpha$, tenemos $T(n) \in \mathcal{O}\left(n^\alpha\right)$. Si en vez $a = d(b)$ y $d(n) = n^\alpha$, tenemos $T(n) \in \mathcal{O}\left(n^\alpha \log_b n\right)$. Esto se demuestra por el siguiente análisis.

Sea $a > d(b)$. La parte heterogénica es entonces

$$\frac{a^k - d(b)^k}{\frac{a}{d(b)} - 1} \in \mathcal{O}\left(a^k\right), \tag{7.16}$$

por lo cual $T(n) \in \mathcal{O}\left(a^k\right) = \mathcal{O}\left(n^{\log_b a}\right)$, porque $a^k = a^{\log_b n} = n^{\log_b a}$. En este caso la parte homogénica y la parte heterogénica son practicamente iguales.

En el caso que $a < d(b)$, la parte heterogénica es $\mathcal{O}\left(d(b)^k\right)$ y $T(n) \in \mathcal{O}\left(d(b)^k\right) = \mathcal{O}\left(n^{\log_b d(b)}\right)$, porque $d(b)^k = d(b)^{\log_b n} = n^{\log_b d(b)}$.

Si $a = d(b)$, tenemos para la parte heterogénica que

$$\sum_{j=0}^{k-1} a^j d(b)^{k-j} = d(b)^k \sum_{j=0}^{k-1} \left(\frac{a}{d(b)}\right)^j = d(b)^k \sum_{j=0}^{k-1} 1^j = d(b)^k k, \tag{7.17}$$

por lo cual $T(n) \in \mathcal{O}\left(d(b)^k\, k\right) = \mathcal{O}\left(n^{\log_b d(b)}\, \log_b n\right)$.

Por ejemplo, si $T(1) = 1$ y $T(n) = 4T\left(\frac{n}{2}\right) + n$, tenemos $T(n) \in \mathcal{O}\left(n^2\right)$. Si tenemos $T(1) = 1$ con $T(n) = 4T\left(\frac{n}{2}\right) + n^2$, llegamos a $T(n) \in \mathcal{O}\left(n^2 \log n\right)$. Mientras con $T(1) = 1$ y $T(n) = 4T\left(\frac{n}{2}\right) + n^3$, resulta que $T(n) \in \mathcal{O}\left(n^3\right)$.

En todos estos ejemplos $a = 4$ y $b = 2$, por lo cual la parte homogénica es para todos $a^{\log_b n} = n^{\log_b a} = n^2$.

Incluso se puede estimar la parte heterogénica en algunos casos donde d no es multiplicativa. Por ejemplo, en

$$\begin{cases} T(1) &= 1 \\ T(n) &= 3T\left(\frac{n}{2}\right) + 2n^{1,5} \end{cases} \tag{7.18}$$

$d(n) = 2n^{1,5}$ no es multiplicativa, mientras $n^{1,5}$ sóla lo es. Usamos la notación $U(n) = \frac{T(n)}{2}$ para todo n. Entonces

$$\begin{aligned} U(1) &= \tfrac{1}{2} \\ U(n) &= \tfrac{T(n)}{2} = \frac{3T\left(\frac{n}{2}\right)}{2} + n^{1,5} = 3U\left(\frac{n}{2}\right) + n^{1,5}. \end{aligned} \tag{7.19}$$

Si tuvieramos que $U(1) = 1$, tendríamos una parte homogénica $3^{\log n} = n^{\log 3}$. En el caso $U(1) = \frac{1}{2}$ tendríamos $\frac{1}{2} n^{\log 3}$.

En la parte heterogénica, el valor de $U(1)$ no tiene ningún efecto, por lo cual en el caso $a = 3$ y $b = 2$ con $d(b) = b^{1,5} \approx 2,82$ aplica que $d(b) < a$. Entonces la parte heterogénica es $\mathcal{O}\left(n^{\log 3}\right)$.

En consecuencia, tenemos que $U(n) \in \mathcal{O}\left(n^{\log 3}\right)$, y porque $T(n) = 2\,U(n)$, también $T(n) \in \mathcal{O}\left(n^{\log 3}\right)$.

Como otro ejemplo, analizamos la ecuación

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 2T\left(\tfrac{n}{2}\right) + n \log n \end{aligned} \tag{7.20}$$

donde la parte homogénica es $a^k = 2^{\log n} = n^{\log 2} = n$. $d(n) = n \log n$ no es multiplicativa, por lo cual habrá que estimar directamente la parte heterogénica:

$$\begin{aligned} \sum_{j=0}^{k-1} a^j d(b^{k-j}) &= \sum_{i=0}^{k-1} 2^j 2^{k-j} \log 2^{k-j} = 2^k \sum_{j=0}^{k-1} k - j \\ &= 2^k \left(k + (k-1) + (k-2) + \ldots + 1\right) \\ &= 2^k \tfrac{k(k+1)}{2} = 2^{k-1} k(k+1). \end{aligned} \tag{7.21}$$

Se asigna $k = \log n$:

$$\begin{aligned} 2^{k-1} k(k+1) &= 2^{\log n - 1} \log n (\log n + 1) \\ &= 2^{\log\left(\frac{n}{2}\right)} (\log^2 n + \log n) \\ &= \tfrac{n}{2} (\log^2 n + \log n), \end{aligned} \tag{7.22}$$

o sea, la parte heterogénica es $\mathcal{O}\left(n \log^2 n\right)$.

7.2.2. Método de expansión

El *método de expansión* facilita la computación involucrada en abrir una ecuación recursiva. Como un ejemplo, veremos la ecuacion siguiente:

$$\begin{cases} R(1) &= 1 \\ R(n) &= 2\,R(n-1) + n, \text{ donde } n \geq 2. \end{cases} \tag{7.23}$$

Por descomponer la ecuación se obtiene

$$\begin{aligned} R(n) &= 2R(n-1) + n \\ &= 2(2R(n-2) + n - 1) + n \\ &= 4R(n-2) + 3\,n - 2 \\ &= 4(2R(n-3) + n - 2) + 3\,n - 2 \\ &= 8R(n-3) + 7\,n - 10 \\ &= \ldots \end{aligned} \tag{7.24}$$

La expansión se realiza de la manera siguiente:

$$\begin{array}{lcll} R(n) & = & 2R(n-1)+n & | \times \ 1 \\ R(n-1) & = & 2R(n-2)+(n-1) & | \times \ 2 \\ R(n-2) & = & 2R(n-3)+(n-2) & | \times \ 4 \\ & \vdots & & \\ R(n-i) & = & 2R(n-i-1)+(n-i) & | \times \ 2^i \\ & \vdots & & \\ R(n-(n-2)) & = & 2R(1)+2 & | \times \ 2^{n-2} \end{array} \tag{7.25}$$

Ahora multiplicamos por los coeficientes del lado izquiera y sumamos para obtener el resultado:

$$\begin{array}{rcl} R(n) & = & 2^{n-1}R(1)+n+2(n-1)+4(n-2)+\ldots+2^{n-2}2 \\ & = & n+2(n-1)+4(n-2)+\ldots+2^{n-2}2+2^{n-1} \\ & = & \displaystyle\sum_{i=0}^{n-1} 2^i(n-i) \\ & = & \underbrace{2^0+\ldots+2^0}_{n \text{ veces}}+\underbrace{2^1+\ldots+2^1}_{n-1 \text{ veces}}+\underbrace{2^2+\ldots+2^2}_{n-2 \text{ veces}}+\ldots+\underbrace{2^{n-1}}_{1 \text{ vez}} \\ & = & \displaystyle\sum_{i=0}^{n-1}\sum_{j=0}^{i} 2^j = \sum_{i=0}^{n-1}(2^{i+1}-1) = \sum_{i=0}^{n-1} 2^{i+1} - \sum_{i=0}^{n-1} 1 \\ & = & 2^{n+1}-2-n. \end{array} \tag{7.26}$$

7.2.3. Transformaciones

Para simplificar la solución, podemos hacer una asignación cambiando el dominio o el rango del mapeo T. Por ejemplo, con

$$\begin{array}{rcl} T(0) & = & 1 \\ T(1) & = & 2 \\ & \ldots & \\ T(n) & = & T(n-1)T(n-2), \text{ si } n \geq 2 \end{array} \tag{7.27}$$

podemos asignar $U(n) = \log T(n)$. Esto nos deja con

$$\begin{aligned} U(0) &= 0 \\ U(1) &= 1 \\ &\dots \\ U(n) &= U(n-1) + U(n-2), \text{ si } n \geq 2. \end{aligned} \tag{7.28}$$

Hemos llegado a tener $U(n) = F_n$, o sea, el $n$ésimo número de Fibonacci . Entonces $T(n) = 2^{F_n}$.

Otro ejemplo es

$$\begin{cases} T(1) &= 1 \\ T(n) &= 3T(n/2) + n, \text{ si } n = 2^k > 1. \end{cases} \tag{7.29}$$

donde con la asignación $U(n) = T(2^n)$ llegamos a

$$\begin{cases} U(0) &= 1 \\ U(n) &= T(2^n) = 3T(2^{n-1}) + 2^n \\ &= 3U(n-1) + 2^n, \text{ si } n \geq 1. \end{cases} \tag{7.30}$$

Abriendo esta definición, se obtiene

$$\begin{aligned} U(n) &= 3U(n-1) + 2^n &&| \times \ 1 \\ U(n-1) &= 3U(n-2) + 2^{n-1} &&| \times \ 3 \\ U(n-2) &= 3U(n-3) + 2^{n-2} &&| \times \ 9 \\ &\vdots \\ U(1) &= 3U(0) + 2 &&| \times \ 3^{n-1} \end{aligned} \tag{7.31}$$

y entonces

$$\begin{aligned} U(n) &= 3^n + \sum_{i=0}^{n-1} 3^i \, 2^{n-i} = 3^n + 2^n \sum_{i=0}^{n-1} \left(\frac{3}{2}\right)^i \\ &= 3^n + 2^n \frac{\left(\frac{3}{2}\right)^n - 1}{\frac{1}{2}} = 3^n + 2^{n+1} \cdot \left(\frac{3}{2}\right)^n - 2^{n+1} \\ &= 3^n + 2 \cdot 3^n - 2^{n+1} = 3^{n+1} - 2^{n+1} = 3 \cdot 2^{n \log 3} - 2^n \cdot 2. \end{aligned} \tag{7.32}$$

De la condición $U(n) = T(2^n)$ derivamos que $T(n) \in U(\log n)$, o sea,

$$T(n) = U(\log n) = 3 \cdot 2^{\log n \log 3} - 2^{\log n} 2 = 3n^{\log 3} - 2n. \tag{7.33}$$

7.3. Análisis de complejidad promedio

En muchos casos, la cota superior de la análisis asintótica del caso peor da una idea bastante pesimista de la situación — puede ser que son muy escasas las instancias de peor caso, mientras una gran mayoría de las instancias tiene tiempo de ejecución mucho mejor. Si uno conoce la *distribución de probabilidad* de las instancias (de un caso práctico), se puede analizar la complejidad *promedio* de un algoritmo. En los casos donde no hay información *a priori* de las probabilidades, se asume que cada instancia es equiprobable (es decir, la instancia está seleccionada del espacio de todas las instancias posibles uniformemente al azar).

Típicamente el análisis de complejidad promedio es más desafiante que el análisis asintótica del por caso. En la sección siguiente verémos el análisis *amortizada* que resulta más fácil para muchos casos de procesamiento de estructuras de datos.

7.3.1. Ordenación rápida

En esta sección, como un ejemplo de análisis de complejidad promedio, analizamos el algoritmo de ordenación rápida (inglés: quicksort; presentado en la sección 6.1.2). Su peor caso es $\mathcal{O}(n^2)$ para n elementos, pero resulta que en la práctica suele ser el método más rápido. El análisis de complejidad promedia da a ordenación rápida la complejidad $\mathcal{O}(n \log n)$, que implica que el caso peor no es muy común.

Cada vez que dividimos una cola o un arreglo, usamos tiempo $\Theta(()\, n)$ y un espacio auxiliar de tamao $\mathcal{O}(1)$. Para unir dos colas (de tamaos n_1 y n_2) en uno, se necesita por máximo el tiempo $\mathcal{O}(n_1 + n_2)$; en la implementación que usa un arreglo, unir las partes explicitamente no será necesario.

Suponemos que la subrutina de elección del pivote toma $\mathcal{O}(n)$ tiempo para n elementos, el tiempo total de ejecución del algoritmo de ordenación rápida está capturado en la ecuación recursiva siguiente:

$$T(n) = \begin{cases} \mathcal{O}(1), & \text{si } n \leq 1, \\ T(p) + T(n-p) + \Theta(n), & \text{si } n > 1, \end{cases} \tag{7.34}$$

donde p es el tamao de la una de las dos partes de la división (y $n - p$ el tamao de la otra parte). El peor caso corresponde a la situación donde $p = 1$ en cada división del

algoritmo.

$$T_{\text{peor}}(n) = \begin{cases} \Theta(1), & \text{si } n \leq 1, \\ T_{\text{peor}}(n-1) + \Theta(n), & \text{en otro caso,} \end{cases} \tag{7.35}$$

La solución de la ecuación del peor caso es $T_{\text{peor}}(n) = \Theta(n^2)$. Para analizar el caso promedio, hacemos las siguientes suposiciones:

1. Los n elementos son $\{1, 2, \ldots, n\}$ (o en términos más generales, todos los elementos son distintos).
2. La permutación en la cual aparecen los elementos está elegida entre los $n!$ posibles permutaciones uniformemente al azar.
3. El último elemento está siempre elegido como el pivote; esto de logra en en tiempo $\mathcal{O}(1) \in \mathcal{O}(n)$.
4. Las operaciones de dividir y unir las partes usan al máximo tiempo cn, donde c es una constante.
5. La complejidad del caso donde $n \leq 1$ usa d pasos de computación.

La complejidad del algoritmo ahora depende de la elección del pivote, que en turno determina exactamente los tamaos de las dos partes:

$$T(n) \leq \begin{cases} T(0) + T(n) + cn, & \text{si el pivote es } 1 \\ T(1) + T(n-1) + cn, & \text{si el pivote es } 2 \\ T(2) + T(n-2) + cn, & \text{si el pivote es } 3 \\ \vdots & \vdots \\ T(n-2) + T(2) + cn, & \text{si el pivote es } n-1 \\ T(n-1) + T(1) + cn, & \text{si el pivote es } n. \end{cases} \tag{7.36}$$

Considerando el conjunto de las $n!$ permutaciones de los elementos, cada caso ocurre $(n-1)!$ veces (se fija el último elemento y considera las permutaciones de los otros elementos). Entonces, la complejidad del caso promedio se puede caracterizar con la ecuación diferencial:

$$
\begin{aligned}
T(n) &= \begin{cases} d, & n = 1, \\ \frac{(n-1)!}{n!}\left(T(0) + T(n) + cn + \sum_{i=1}^{n-1}(T(i) + T(n-i) + cn)\right), & n > 1, \end{cases} \\
&= \tfrac{1}{n}\Big(T(0 + T(n) + cn + \sum_{i=1}^{n-1}\Big(T(i) + T(n-i) + cn\Big)\Big) \\
&= \tfrac{1}{n}\Big(T(0 + T(n + cn + \sum_{i=1}^{n-1}\Big(T(i) + T(n-i) + cn\Big)\Big) \\
&= \tfrac{1}{n}\Big(T(0 + T(n) + cn + (n-1)cn + 2\sum_{i=1}^{n-1} T(i)\Big) \\
&\leq \tfrac{d}{n} + Cn + cn + \tfrac{2}{n}\sum_{i=1}^{n-1} T(i) \\
&\leq (d + C + c)n + \tfrac{2}{n}\sum_{i=1}^{n-1} T(i).
\end{aligned}
\tag{7.37}
$$

La sumación está simplifacada por la observación que tiene además de cn dos ocurrencias de cada $T(i)$. El penúltimo paso con $\leq$ viene del hecho que ya sabemos del análisis de peor caso que $T(n) \leq Cn^2$ para alguna constante C. Además sabemos que $T(1) = d$. Para simplificar, reemplazamos la constante $d + C + c = D$ para obtener la ecuación

$$T(n) \leq Dn + \frac{2}{n}\sum_{i=1}^{n-1} T(i) \tag{7.38}$$

y intentamos solucionarla con la adivinanza $T(n) \leq \alpha n \log n$, donde α es una constante suficientemente grande. La demostracíón es por inducción: para $n = 2$, aplica la ecuación siempre y cuando $\alpha \geq D + \frac{d}{2}$. Para el caso $n > 2$, asumimos que para todo $i < n$ aplica

que $T(i) \leq \alpha i \log i$. Para el caso de valores pares de n, se obtiene

$$\begin{aligned}
T(n) &\leq Dn + \tfrac{2}{n}\sum_{i=1}^{n-1} \alpha i \log i = Dn + \tfrac{2\alpha}{n}\left(\sum_{i=1}^{\frac{n}{2}}(i\log i) + \sum_{i=\frac{n}{2}+1}^{n-1}(i\log i)\right) \\
&= Dn + \tfrac{2\alpha}{n}\left(\sum_{i=1}^{\frac{n}{2}}(i\log i) + \sum_{i=1}^{\frac{n}{2}-1}(\tfrac{n}{2}+i)\log\left(\tfrac{n}{2}+i\right)\right) \\
&= Dn + \tfrac{2\alpha}{n} + \left(\sum_{i=1}^{\frac{n}{2}} i(\log n - 1) + \sum_{i=1}^{\frac{n}{2}-1}(\tfrac{n}{2}+i)\log n\right) \\
&= Dn + \tfrac{2\alpha}{n}\left((\log n - 1)\sum_{i=1}^{\frac{n}{2}} i + \log n\left(\tfrac{n}{2}\left(\tfrac{n}{2}-1\right) + \sum_{i=1}^{\frac{n}{2}-1} i\right)\right) \\
&\leq Dn + \tfrac{2\alpha}{n}\left((\log n - 1)\left(\tfrac{n}{2}\cdot\frac{1+\frac{n}{2}}{2}\right) + \log n\left(\tfrac{n^2}{4} - \tfrac{n}{2} + (\tfrac{n}{2}-1)\cdot\frac{1+\frac{n}{2}-1}{2}\right)\right) \\
&= Dn + \tfrac{2\alpha}{n}\left(\log n\left(\tfrac{n^2}{8}+\tfrac{n}{4}\right) - \tfrac{n^2}{8} - \tfrac{n}{4} + \log n\left(\tfrac{n^2}{4} - \tfrac{n}{2} + \tfrac{n^2}{8} - \tfrac{n}{4}\right)\right) \\
&= Dn + \alpha\left(\log n(n-1) - \tfrac{n}{4} - \tfrac{1}{2}\right) \\
&\leq Dn + \alpha n \log n - \alpha\cdot\tfrac{n}{4} \\
&\leq \alpha n \log n, \text{ si } \alpha \geq 4D,
\end{aligned} \tag{7.39}$$

porque para todo i aplican

$$\begin{aligned}
\log i &\leq \log \tfrac{n}{2} = \log n - 1 \\
\log\left(\tfrac{n}{2}+i\right) &\leq \log(n-1) \leq \log n
\end{aligned} \tag{7.40}$$

con logaritmos de base dos y por aplicar la suma de sucesión aritmética. Para valores impares de n, el análisis es muy parecido. Estas calculaciones verifican la adivinanza $T(n) = \alpha n \log n$.

7.4. Análisis de complejidad amortizada

La idea en el análisis de complejidad *amortizada* es definir una *sucesión* de n operaciones y estimar una cota superior para su tiempo de ejecución. Esta cota superior está dividido por el número de operaciones n para obtener la complejidad amortizada (inglés: amortized complexity) de una operación. No es necesario que las operaciones de la sucesión sean iguales ni del mismo tipo. La motivación de complejidad amortizada es poder

"pronósticar" con mayor exactitud el tiempo de ejecución del uso del algoritmo en el "mundo real": si uno típicamente quiere realizar ciertos tipos de cosas, ¿cuánto tiempo toma?

Para formalizar el concepto, sea D_0 el estado inicial de una estructura de datos. En cada momento, se aplica una operación O en la estructura. Después de i operaciones, el estado de la estructura será D_i. Denotamos la sucesión de operaciones como $O_1, \ldots, O_n$. El "costo" computacional de operación O_i puede ser muy distinto del costo de otra operación O_j. La idea es sumar los costos y pensar en términos de "costo promedio". Para que tenga sentido ese tipo de análisis, la sucesión de operaciones estudiada tiene que ser la *peor* posible — si no lo es, no hay guarantía ninguna que se pueda generalizar el análisis para sucesiones más largas.

7.4.1. Arreglos dinámicos

Como un ejemplo, se evalúa la complejidad de operar un *arreglo dinámico*: se duplica el tamao siempre cuando hace falta aadir un elemento (y se corta el tamao a mitad cuando se ha liberado tres cuartas partes del espacio). Primero se evalúa la sucesión de 2^k inserciones en tal arreglo. Al realizar la primera inserción, se crea un arreglo de un elemento. Con la segunda, se duplica el tamao para lograr a guardar el segundo elemento. En práctica, esto significa que se reserva un arreglo nuevo del tamao doble en otra parte y mueve cada clave que ya estaba guadrara en el arreglo al espacio nuevo. Con la tercera inserción, se crea espacio para dos elementos más, etcétera.

Entonces, con la sucesión de operaciones definida, vamos a tener que multiplicar el tamao del arreglo k veces y las otras $2^k - k$ inserciones van a ser "baratas". La complejidad de una inserción barata es $\mathcal{O}(1)$. Cuando el tamao del arreglo es n, el costo de duplicar su tamao es $\mathcal{O}(n)$. Entonces, la complejidad de una inserción "cara" es $\mathcal{O}(1) + \mathcal{O}(n) \in \mathcal{O}(n)$ y para la inserción difícil número i, el tamao va a ser 2^i. Entonces la suma de la complejidad de las k inserciones difíciles es

$$\mathcal{O}\left(\sum_{i=1}^{k} 2^i\right) = \mathcal{O}\left(2^{k+1}\right). \tag{7.41}$$

La complejidad total de los $2^k - k$ operaciones fáciles es $\mathcal{O}\left(2^k - k\right) \in \mathcal{O}\left(2^k\right)$ porque cada una es de tiempo constante. Entonces, la complejidad de la sucesión completa de 2^k inserciones es $\mathcal{O}\left(2^{k+1} + 2^k\right) \in \mathcal{O}\left(3 \cdot 2^k\right)$. Dividimos este por el número de operaciones 2^k para obtener la complejidad amortizada por operación:

$$\mathcal{O}\left(\frac{3 \cdot 2^k}{2^k}\right) = \mathcal{O}(3) \in \mathcal{O}(1). \tag{7.42}$$

Entonces, si hacemos n operaciones y cada uno tiene complejidad amortizada constante, cada sucesión de puras inserciones tiene complejidad amortizada lineal, $\mathcal{O}(n)$.

Para poder incluir las eliminaciones en el análisis, hay que cambiar la técnica de análisis. Utilizamos el *método de potenciales*, también conocido como el método de *cuentas bancarias*), donde se divide el costo de operaciones caras entre operaciones más baratas.

Supone que al comienzo nos han asignado una cierta cantidad M de pesos. Cada operación básica de tiempo constante $\mathcal{O}(1)$ cuesta un peso. Se realizan n operaciones. El *costo planeado* de una operación O_i es $p_i = \frac{M}{n}$ pesos. Si una operación no necesita todos sus $\frac{M}{n}$ pesos asignados, los *ahorramos* en una cuenta bancaria común. Si una operación necesita más que $\frac{M}{n}$ pesos, la operación puede retirar fondos adicionales de la cuenta bancaria. Si en la cuenta no hay balance, la operación los toma prestados del banco. Todo el préstamo habrá que pagar con lo que ahorran operaciones que siguen. Denotamos el balance de la cuenta después de la operación número i, o sea, en el estado D_i, con Φ_i — este balance también se llama la *potencial* de la estructura. La meta es poder mostrar que la estructura siempre contiene "la potencial suficiente" para poder pagar la operaciones que vienen.

Sea r_i el *costo real* de la operación O_i. El *costo amortizado* de la operación O_i es

$$a_i = t_i + \underbrace{\Phi_i - \Phi_{i-1}}_{\text{pesos ahorrados}} \tag{7.43}$$

El costo amortizado total de las n operaciones es

$$\begin{aligned} A = \sum_{i=1}^{n} a^i &= \sum_{i=1}^{n} (r_i + \Phi_i - \Phi_{i-1}) \\ &= \sum_{i=1}^{n} r_i + \sum_{i=1}^{n} (\Phi_i - \Phi_{i-1}) \\ &= \sum_{i=1}^{n} r_i + \Phi_n - \Phi_0, \end{aligned} \tag{7.44}$$

por lo cual el costo real total es

$$R = \sum_{i=1}^{n} r_1 = \Phi_0 - \Phi_n + \sum_{i=1}^{n} a_i. \tag{7.45}$$

Para poder realizar el análisis, hay que asignar los costos planeados p_i, después de que se intenta definir una *función de balance* $\Phi_i : \mathbb{N} \to \mathbb{R}$ tal que el costo amortizado de cada operación es menor o igual al costo planeado:

$$a_i = r_i + \Phi_i - \Phi_{i-1} \leq p_i. \tag{7.46}$$

Así también aplicaría

$$\sum_{i=1}^{n} r_i \leq \Phi_0 - \Phi_n + \sum_{i=1}^{n} p_i. \tag{7.47}$$

La transacción con el banco (ahorro o préstamo) de la operación O_i es de $p_i - r_i$ pesos. Típicamente queremos que $\Phi_i \geq 0$ para cada estado D_i. Si esto aplica, se tiene que

$$\sum_{i=1}^{n} r_i \leq \Phi_0 + \sum_{i=1}^{n} p_i. \tag{7.48}$$

Comúnmente además tenemos que $\Phi_0 = 0$, en cual caso la suma de los costos planeados es una cota superior a la suma de los costos reales. Entonces, si uno elige costos planeados demasiado grandes, la cota será cruda y floja y la estimación de la complejidad no es exacta, mientras haber elegido un valor demasiado pequeo de los costos planeados, resulta imposible encontrar una función de balance apropiada.

Regresamos por ahora al caso de puras inserciones a un arreglo dinámico. Hay que asegurar que el precio de una inserción cara se pueda pagar con lo que han ahorrado las inserciones baratas anteriores. Suponemos que el tiempo de una inserción fácil sea t_1 y el tiempo de mover una clave al espacio nuevo (de tamao doble) sea t_2. Entonces, si antes de la inserción cara, la estructura contiene n claves, el precio real total de la inserción cara es $t_1 + nt_2$.

Se define una función de balance tal que su valor inicial es cero y también su valor después de hacer duplicado el tamao es cero. En cualquier otro momento, será $\Phi_i > 0$. La meta es llegar al mismo resultado del método anterior, o sea $\sum_{i=1}^{n} r_i = \mathcal{O}(n)$ para tener complejidad amortizada $\mathcal{O}(1)$ por operación.

Se prueba primero el análisis con $p_i = t_1 + 2t_2$: habrá que poder pagar el costo de aadir la clave nueva (t_1) y preparar para mover la clave misma después a un arreglo nuevo de tamao doble (t_2) además de preparar mover todas las claves anteriores (uno por elemento, el otro t_2). Queremos cumplir con ecuación 7.46: para todo i, necesitamos que

$$a_i = r_i + \Phi_i - \Phi_{i-1} \leq p_i = t_1 + 2t_2. \tag{7.49}$$

Denotamos por c_i el número de claves guardados en el arreglo en el estado D_i. Claramente $c_i = c_{i-1} + 1$. Intentamos con la función

$$\Phi_i = 2t_2 c_i - t_2 e_i, \tag{7.50}$$

donde e_i es el espacio total del arreglo después de la operación O_i. Para inserciones fáciles, aplica $e_i = e_{i-1}$, mientras con las difíciles aplica $e_i = 2e_{i-1} = 2c_{i-1}$.

El caso de una inserción fácil será entonces

$$\begin{aligned} a_i &= t_1 + (2t_2(c_{i-1} + 1) - t_2 e_{i-1}) - (2t_2 c_{i-1} - t_2 e_{i-1}) \\ &= t_1 + 2t_2 = p_i. \end{aligned} \tag{7.51}$$

Entonces el costo amortizado es igual al costo planeado. Para las inserciones difíciles, tenemos

$$\begin{aligned} a_i &= (t_1 + t_2 c_{i-1}) + (2t_2(c_{i-1}+1) - t_2 2c_{i-1}) \\ &- (2t_2 c_{i-1} - t_2 c_{i-1}) \\ &= t_1 + 2t_2 = p_i, \end{aligned} \tag{7.52}$$

que también cumple con lo que queríamos: tenemos que $a_i = t_1 + 2t_2$ para todo i. Entonces cada operación tiene costo amortizado $\mathcal{O}(1)$ y el costo real de una serie de n operaciones es $\mathcal{O}(n)$.

Ahora podemos considerar eliminaciones: si un arreglo de tamao e_i contiene solamente $c_i \leq \frac{1}{4}e_i$ claves, su tamao será cortado a la mitad: $e_{i+1} = \frac{1}{2}e_i$. El costo de la reducción del tamao es $t_2 c_i$, porque habrá que mover cada clave. Marcamos el precio de una eliminación simple (que no causa un ajuste de tamao) con t_3.

Se cubre el gasto del reducción por cobrar por eliminaciones que ocurren cuando el arreglo cuenta con menos que $\frac{e_i}{2}$ claves guardadas en el estado D_i. Entonces, por cada eliminación de ese tipo, ahorramos t_2 pesos. Al momento de reducir el tamao, hemos ahorrado $(\frac{e_i}{4} - 1)$ pesos (nota que e_i solamente cambia cuando duplicamos o reducimos, no en operaciones baratas). Cuando guardamos claves a las posiciones de la última cuarta parte de la estructura, usamos los pesos ahorrados y no ahorramos náda más. Elegimos ahora una función de balance definido por partes:

$$\Phi_i = \begin{cases} 2t_2 c_i - t_2 e_i, & \text{si } c_i \geq \frac{e_i}{2}, \\ \frac{e_1}{2} t_2 - t_2 c_i, & \text{si } c_i < \frac{e_i}{2}. \end{cases} \tag{7.53}$$

Analizamos primero el caso de inserciones: Para el caso $c_i > \frac{e_i}{2}$, no cambia nada a la situación anterior: $a_i = t_1 + 2t_2$. Para el otro caso $c_i < \frac{e_i}{2}$, se tiene

$$\begin{aligned} a_i &= r_i + \Phi_i - \Phi_{i-1} \\ &= t_1 + \left(\frac{t_2 e_i}{2} - t_2 c_i\right) - \left(\frac{t_2 e_{i-1}}{2} - t_2 c_{i-1}\right) \\ &= t_1 - t_2, \end{aligned} \tag{7.54}$$

porque para estas inserciones, siempre se tiene $e_i = e_{i-1}$ (nunca causan que se duplique el espacio). Para las inserciones "especiales" que ocurren justo cuando cambia la definición

de la función de balance (o sea, el caso donde $c_i = \frac{e_i}{2}$), se obtiene

$$\begin{aligned} a_i &= r_i + \Phi_i - \Phi_{i-1} \\ &= t_1 + 2t_2c_i - t_2e_i - \frac{t_2e_{i-1}}{2} + t_2c_{i-1} \\ &= t_1 + 2t_2c_i - 2t_2c_i - t_2c_i + t_2(c_i - 1) \\ &= t_1 - t_2. \end{aligned} \tag{7.55}$$

Para las eliminaciones, empezamos con el caso de eliminaciones fáciles sin ahorro, o sea $c_i \geq \frac{e_i}{2}$:

$$\begin{aligned} a_i &= r_i + \Phi_i - \Phi_{i-1} \\ &= t_3 + (2t_2c_i - t_2e_i) - (2t_2c_{i-1} - t_2e_{i-1}) \\ &= t_3 + 2t_2(c_{i-1} - 1) - 2t_2c_{i-1} \\ &= t_3 - 2t_2. \end{aligned} \tag{7.56}$$

Para el caso de $c_i = \frac{e_i}{2} - 1$ al cambio de definición de Φ se obtiene

$$\begin{aligned} a_i &= r_i + \Phi_i - \Phi_{i-1} \\ &= t_3 + \left(\frac{t_2e_i}{2} - t_2e_i\right) - (2t_2e_{i-1} - t_2e_i) \\ &= t_3 + \frac{3t_2e_i}{2} - t_2(e_{i-1} - 1) - 2t_2c_{i-1} \\ &= t_3 + 3t_2c_{i-1} - t_2e_{i-1} + t_2 - 2t_2c_{i-1} \\ &= t_3 + t_2. \end{aligned} \tag{7.57}$$

Para las eliminaciones con ahorro, o sea cuando $c_i < \frac{e_1}{2} - 1$ cuando $c_{i-1} \geq \frac{e_i}{4} + 1$, se tiene

$$\begin{aligned} a_i &= r_i + \Phi_i - \Phi_{i-1} \\ &= t_3 + \left(\frac{t_2e_i1}{2} - t_2c_i\right) - \left(\frac{t_2e_{i-1}}{2} - t_2c_{i-1}\right) \\ &= t_3 - t_2(c_{i-1} - 1) + t_2c_{i-1} \\ &= t_3 + t_2. \end{aligned} \tag{7.58}$$

Para el último caso, el eliminación, $c_{i-1} = \frac{e_{i-1}}{4}$, se obtiene

$$\begin{aligned} a_i &= r_i + \Phi_i - \Phi_{i-1} \\ &= t_3 + \left(t_2 c_i - \frac{t_2 e_i}{2}\right) - \left(\frac{t_2 e_{i-1}}{2} - t_2 c_{i-1}\right) \\ &= t_3 + \frac{t_2 e_{i-1}}{4} - \frac{t_2 e_{i-1}}{2} + t_2 c_{i-1} \\ &= t_3 - \frac{t_2 e_{i-1}}{4} + \frac{t_2 e_{i-1}}{4} \\ &= t_3. \end{aligned} \tag{7.59}$$

Entonces en todos los casos llegamos a tener una complejidad amortizada constante $\mathcal{O}(1)$ por operación, por lo cual la complejidad de cualquier combinación de n inserciones y/o eliminaciones, la complejidad amortizada total es $\mathcal{O}(n)$.

La regla general de poder lograr la meta de "siempre poder pagar las operaciones que quedan de hacer" es asegurar que se mantenga un *invariante de costo* (inglés: credit invariant) durante toda la sucesión. El invariante del ejemplo de arreglos dinámicos consiste de las siguientes reglas:

1. Si $c_i = \frac{e_i}{2}$, el balance Φ_i es cero.
2. Si $(\frac{e_i}{4} \leq c_i < \frac{e_i}{2})$, el balance Φ_i es igual a la cantidad de celdas libres en las posiciones de la segunda cuarta parte del arreglo.
3. Si $c_i > \frac{e_i}{2}$, el balance Φ_i es igual a dos veces la cantidad de claves guardados en las posiciones de la segunda mitad del arreglo.

7.4.2. Árboles biselados

El siguiente ejemplo del análisis amortizado son los árboles biselados. Denotamos la cantidad de claves guardadas en el árbol por n y el número de operaciones realizados al árbol con m. Queremos mostrar que la complejidad amortizada de cualquier sucesión de m operaciones es $\mathcal{O}(m \log n)$. Marcamos con $R(v)$ el ramo la raíz de cual es el vértice v y con $|R(v)|$ el número de vértices en el ramo (incluyendo a v). Definimos

$$\mu(v) = \lfloor \log |R(v)| \rfloor. \tag{7.60}$$

Sea r la raíz del árbol completo. El costo planeado de las operaciones será

$$p_i = \mathcal{O}(\log n) = \mathcal{O}(\log |R(r)|) = \mathcal{O}(\mu(r)). \tag{7.61}$$

Cada operación de un árbol adaptivo (ver sección 6.3.5) constituye de un número constante de operaciones `splay` y un número constante de operaciones simples, basta con establecer que la complejidad amortizada de una operación `splay` es $\mathcal{O}(\mu(r))$.

Pensamos en este caso que cada vértice del árbol tiene una cuenta propia, pero el balance está en uso de todos. Mantenemos el siguiente invariante de costo: cada vértice v siempre tiene por lo menos $\mu(v)$ pesos en su propia cuenta.

Teorema 32. *Cada operación* `splay`*(v, A) requiere al máximo $(3(\mu(A) - \mu(v)) + 1)$ unidades de costo para su aplicación y la actualización del invariante de costo.*

La demostración del teorema es la siguiente: la operación `splay` consiste de una sucesión de rotaciones (ver figura 6.8).

En el caso que el vértice u que contiene la llave de interés tenga un padre t, pero no un abuelo, hace falta una rotación simple derecha. Aplica que $\mu_{\text{después}}(u) = \mu_{\text{antes}}(t)$ y que $\mu_{\text{después}}(t) \leq \mu_{\text{después}}(u)$. Para mantener el invariante, necesitamos pesos:

$$\begin{aligned} \mu_{\text{después}}(u) + \mu_{\text{después}}(t) - (\mu_{\text{antes}}(u) + \mu_{\text{antes}}t)) &= \mu_{\text{después}}(t) - \mu_{\text{antes}}(u)) \\ &\leq \mu_{\text{después}}(u) - \mu_{\text{antes}}(u). \end{aligned} \tag{7.62}$$

El peso que sobra se gasta en las operaciones de tiempo constante y cantidad constante por operación `splay` (comparaciones, actualizaciones de punteros, etcétera).

En el segundo caso, u tiene además del padre t un abuelo t'. Hay que hacer dos rotaciones derechas — primero para mover t al lugar del abuelo, empujando el abuelo abajo a la derecha y después para mover u mismo al lugar nuevo del padre. El costo total de mantener el invariante es

$$T = \mu_{\text{después}}(u) + \mu_{\text{después}}(t) + \mu_{\text{después}}(t') - \mu_{\text{antes}}(u) - \mu_{\text{antes}}(t) - \mu_{\text{antes}}(t'). \tag{7.63}$$

Por la estructura de la rotación aplica que

$$\mu_{\text{después}}(u) = \mu_{\text{antes}}(t'), \tag{7.64}$$

por lo cual

$$\begin{aligned} T &= \mu_{\text{después}}(t) + \mu_{\text{después}}(t') - \mu_{\text{antes}}(u) - \mu_{\text{antes}}(t) \\ &= (\mu_{\text{después}}(t) - \mu_{\text{antes}}(u)) + (\mu_{\text{después}}(t') - \mu_{\text{antes}}(t)) \\ &\leq (\mu_{\text{después}}(u) - \mu_{\text{antes}}(u)) + (\mu_{\text{después}}(u) - \mu_{\text{antes}}(u)) \\ &= 2(\mu_{\text{después}}(u) - \mu_{\text{antes}}(u)). \end{aligned} \tag{7.65}$$

Si logramos tener $\mu_{\text{después}}(u) > \mu_{\text{antes}}(u)$, nos queda por lo menos un peso para ejecutar la operación. Hay que analizar el caso que $\mu_{\text{después}}(x) = \mu_{\text{antex}}(x)$.

Necesitamos asegurarnos que $T \leq 0$, o sea, no nos cuesta nada mantener el invariante válido. Mostramos que

$$\mu_{\text{después}}(u) = \mu_{\text{antes}}(u) \tag{7.66}$$

es una suposición que llega a una contradicción con

$$\mu_{\text{después}}(u) + \mu_{\text{después}}(t) + \mu_{\text{después}}(t') \geq \mu_{\text{antes}}(u) + \mu_{\text{antes}}(t) + \mu_{\text{antes}}(t'). \tag{7.67}$$

Las ecuaciones 7.66 y 7.64 dan que

$$\mu_{\text{antes}}(t') = \mu_{\text{después}}(u) = \mu_{\text{antes}}(u) \tag{7.68}$$

lo que implica

$$\mu_{\text{antes}}(u) = \mu_{\text{antes}}(t) = \mu_{\text{antes}}(t') \tag{7.69}$$

porque t está en el camino desde u a t' antes de la operación, por definición de μ. Entonces

$$\mu_{\text{después}}(u) + \mu_{\text{después}}(t) + \mu_{\text{después}}(t') \geq 3\,\mu_{\text{antes}}(t') = 3\mu_{\text{después}}(u) \tag{7.70}$$

y además

$$\mu_{\text{después}}(t) + \mu_{\text{después}}(t') \geq 2\mu_{\text{después}}(u). \tag{7.71}$$

Por la estructura de la rotación, tenemos

$$\mu_{\text{después}}(t) \leq \mu_{\text{después}}(u) \text{ y } \mu_{\text{después}}(t') \leq \mu_{\text{después}}(u), \tag{7.72}$$

que en combinación con ecuación 7.71 da

$$\mu_{\text{después}}(u) = \mu_{\text{después}}(t) = \mu_{\text{después}}(t'). \tag{7.73}$$

Ahora, por ecuación 7.64 llegamos a

$$\mu_{\text{antes}}(u) = \mu_{\text{antes}}(t) = \mu_{\text{antes}}(t') = \mu_{\text{después}}(u) = \mu_{\text{después}}(t) = \mu_{\text{después}}(u). \tag{7.74}$$

Sea q_1 el tamao de $R(u)$ antes de la operación y q_2 el tamao de $R(t')$ después de la operación. Por definición de μ tenemos

$$\begin{aligned} \mu_{\text{antes}}(u) &= \lfloor \log q_1 \rfloor \\ \mu_{\text{después}}(t') &= \lfloor \log q_2 \rfloor \\ \mu_{\text{después}}(u) &= \lfloor \log(q_1 + q_2 + 1) \rfloor \end{aligned} \tag{7.75}$$

que juntos con la condición combinada de

$$\mu_{\text{antes}}(u) = \mu_{\text{después}}(u) = \mu_{\text{después}}(t') \tag{7.76}$$

llegamos a tener

$$\lfloor \log q_1 \rfloor = \lfloor \log(q_1 + q_2 + 1) \rfloor = \lfloor \log q_1 \rfloor. \tag{7.77}$$

Si $q_1 \leq q_2$, esto significa que

$$\lfloor \log(q_1 + q_2 + 1) \rfloor \geq \lfloor \log 2q_1 \rfloor = \lfloor \log q_1 \rfloor + 1 > \lfloor \log q_1 \rfloor, \tag{7.78}$$

que es una contradicción con $\mu_{\text{antes}}(u) = \mu_{\text{después}}(u)$.

Habrá que analizar el caso $q_1 > q_2$ igualmente — también llega a una contradicción. Entonces, $\mu_{\text{después}}(u) = \mu_{\text{antes}}(u)$, por lo cual $T < 0$.

El tercer caso de `splay` es una rotación doble izquierda-derecha y omitimos sus detalles.

En resumen: al insertar un elemento el el árbol, hay que asignarle $\mathcal{O}(\log n)$ pesos. Al unir dos árboles, se asigna a la raíz nueva $\mathcal{O}(\log n)$ pesos. Cada operación puede gastar $\mathcal{O}(\log n)$ pesos en ejecutar las operaciones `splay` y el mantenimiento del invariante de costo y las operaciones adicionales que se logra en tiempo $\mathcal{O}(1)$. Entonces, cada operación tiene costo amortizado $\mathcal{O}(\log n)$, por lo cual la complejidad amortizada de cualquier sucesión de m operaciones en un árbol biselado es $\mathcal{O}(m \log n)$.

7.4.3. Montículos de Fibonacci

Para los montículos de Fibonacci (de sección 6.4.2 utilizamos el siguiente invariante de costo: cada raíz tiene un peso y cada vértice marcado tiene dos pesos. Los costos planeados de inserción, decrementación del valor de clase y unir dos montículos son un peso por operación, mientras la eliminación del mínimo tiene costo planeado de $\mathcal{O}(\log n)$ pesos. Demostramos que con estos costos uno puede realizar las operaciones mientras manteniendo el invariante de costo propuesto.

Al insertar una clave, lo único que se hace es crear una raíz nueva con la clave nueva y un peso. Esto toma tiempo constante $\mathcal{O}(1)$. Para disminuir el valor de una clave existente, movemos el vértice con la clave a la lista de raíces en tiempo $\mathcal{O}(1)$ junto con un peso. Si el padre está marcado, su movimiento a la lista de raízes se paga por el peso extra que tiene el vértice marcado — también se le quita la marca por la operación, por lo cual no sufre el invariante. El otro peso el vértice padre lleva consigo. Si hay que marcar el abuelo, hay que aadirle dos pesos al abuelo. El tiempo total es constante, $\mathcal{O}(1)$.

Para unir dos montículos, solamente unimos las listas de raíces. Por la contracción de las listas que se realiza al eliminar el mínimo, el costo de esta operación es $\mathcal{O}(1)$. Esta operación no causa ningún cambio en el invariante. Hay que tomar en cuenta que estamos

pensando que las listas de raíces están implementadas como listas enlazadas, por lo cual no hay que copiar nada, solamente actualizar unos punteros.

Para eliminar el mínimo, la operación en sí toma tiempo $\mathcal{O}(1)$. Además depositamos un peso en cada hijo directo del vértice eliminado. Para recorrer y contraer la lista de raíces, gastamos los pesos de las raíces mismas. Después depositamos de nuevo un peso en cada raíz. Son $\mathcal{O}(\log n)$ raíces, por lo cual la complejidad amortizada de la operación es $\mathcal{O}(\log n)$.

Hay que recordar que la manera de eliminar un elemento cualquiera es por primero disminuir su valor a $-\infty$ y después quitando el mínimo. Esto toma tiempo $\mathcal{O}(1) + \mathcal{O}(\log n) \in \mathcal{O}(\log n)$. Con estos valores, completamos en cuadro 7.1 los costos de operaciones en algunas estructuras de datos.

Cuadro 7.1: Complejidad (asintótica o amortizada) de algunas operaciones básicas en diferentes estructuras de datos fundamentales: listas, árboles *balanceados*, montículos, montículos binomiales y montículos de Fibonacci. Las complejidades amortizadas se indican un asterisco (*).

Operación	Listas	Árboles	Montículos	Mont. bin.	Mont. Fib.
Inserción	$\Theta(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
Ubicar mínimo	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
Eliminar mínimo	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)^*$
Eliminación	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)^*$
Disminuir clave	$\Theta(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)^*$
Unir dos	$\Theta(n)$	$\Theta(n)$	$\Omega(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$

Capítulo 8

Técnicas de diseño de algoritmos

La meta al diseñar un algoritmo es encontrar una manera eficiente a llegar a la solución deseada. En este capítulo presentamos algunas técnicas comunes del diseño de algoritmos que aplican a muchos tipos de problemas. El diseño empieza por buscar un punto de vista adecuado al problema: muchos problemas tienen transformaciones (como las reducciones del análisis de clases complejidad en la sección 5.2) que permiten pensar en el problema en términos de otro problema, mientras en optimización, los problemas tienen *problemas duales* que tienen la misma solución pero pueden resultar más fáciles de resolver.

Muchos problemas se puede dividir en subproblemas tales que la solución del problema entero estará compuesta por las soluciones de sus partes y las partes pueden ser solucionados (completamente o relativamente) independientemente. La composición de tal algoritmo puede ser iterativo (por ejemplo los algoritmos de *línea de barrer* de la sección 8.1) o recursivo (por ejemplo los algoritmos de *dividir y conquistar* de la sección 8.2). Hay casos donde los mismos subproblemas ocurren varias veces y es importante evitar tener que resolverlos varias veces; los algoritmos que logran esto cuentan con *programación dinámica* (de la sección 8.4).

La idea de los *algoritmos de aumentación* es la formación de una solución óptima por mejoramiento de una solución factible. En algunos casos es mejor hacer cualquier aumento, aunque no sea el mejor ni localmente, en vez de considerar todas las alternativas para poder después elegir vorazmente o con otra heurística una de ellas. En general, algoritmos heurísticos pueden llegar al óptimo (global) en algunos casos, mientras en otros casos terminan en una solución factible que no es la óptima, pero ninguna de las operaciones de aumento utilizados logra mejorarla. Este tipo de solución se llama un *óptimo local*.

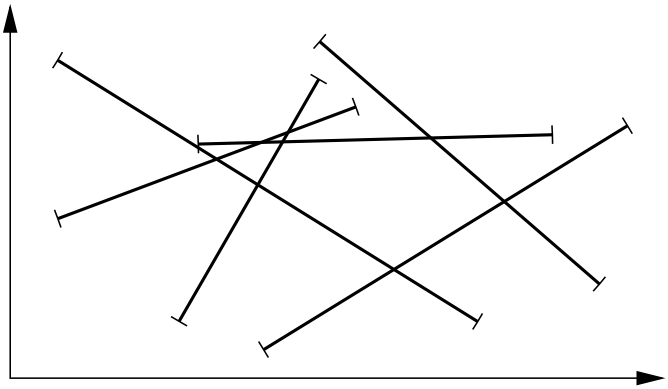

Figura 8.1: Una instancia del problema de intersecciones de segmentos.

8.1. Algoritmos de línea de barrer

Los algoritmos de *línea de barrer* (inglés: sweep line o scan line) dividen el problema en partes que se puede procesar en secuencia. Son particularmente comúnes para los problemas de geometría computacional, donde se procesa objetos geométricos como puntos, líneas, polígonos, etcétera. La instancia del problema está compuesta por la información de la *ubicación* de los objetos en un espacio definido.

Para el caso que es espacio sea un plano, se puede imaginar que una línea mueva en el plano, cruzando la parte relevante del espacio donde se ubican los objetos. El movimiento de la línea sigue uno de los ejes u otra línea fija y la línea se *para* en puntos relevantes al problema. Los puntos de parada se guarda en un montículo. La inicialización del montículo y su actualización son asuntos importantes en el diseño del algoritmo. También se puede aprovechar de otras estructuras de datos auxiliares, por ejemplo para guardar información sobre cuáles objetos están actualmente bajo de la línea. Entre dos paradas de la línea, los papeles relativos de los objetos pueden cambiar, pero los factores esenciales de la solución del problema no deben cambiar de una parada a otra. Si el espacio tiene una sola dimensión, en vez de una línea basta con usar un punto. Para dimensiones mayores n, se "barre" con un hiperplano de dimensión $n-1$.

Problema 21 (Intersecciones de segmentos)**.** *Dado:* n segmentos de líneas en plano de dos dimensiones. *Pregunta:* ¿cuáles son los puntos de intersección entre todos los segmentos?

El algoritmo ingenuo procesa cada uno de los $n(n-1) = \Theta(n^2)$ pares de segmentos $s_i \in S$ y $s_j \in S$ tal que $i \neq j$ y computa su intersección. En el peor caso, esto es el comportamiento óptimo: si todos los segmentos se intersectan, habrá que calcular todas las intersecciones en cualquier caso y solamente imprimir la lista de las instrucciones ya toma $\Theta(n^2)$ tiempo. Para un ejemplo, vea la figura 8.1.

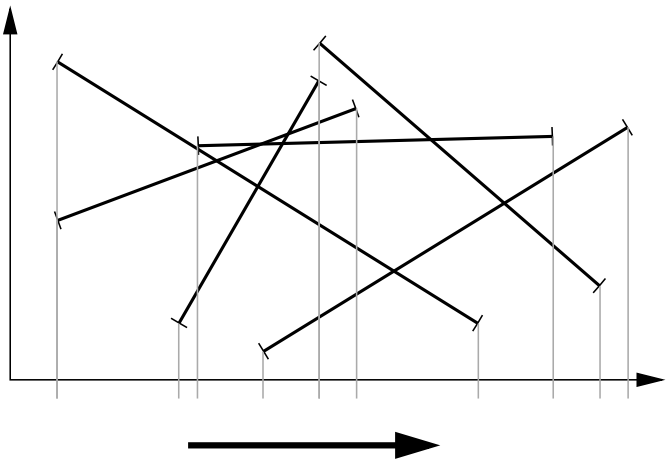

Figura 8.2: Los puntos de inicio y fin de la instancia de la figura 8.1 y la dirección de procedimiento del algoritmo de línea de barrer.

Sin embargo, en una instancia promedia o típica, el número de puntos de intersección k es mucho menor que $n(n-1)$. El algoritmo mejor imaginable tuviera complejidad asintótica $\mathcal{O}(n+k)$, porque se necesita tiempo $\mathcal{O}(n)$ para leer la entrada y tiempo $\mathcal{O}(k)$ para imprimir la salida. Aquí presentamos un algoritmo de línea de barrer que corre en tiempo $\mathcal{O}((n+k)\log n)$.

Los puntos de parada serán todos los puntos donde empieza o termina un segmento y además los puntos de intersección. Son n puntos de comienzo, n puntos finales y k intersecciones. La línea de barrer moverá en perpendicular al eje x, o sea, en paralelo al eje y. Se guardará los puntos de parada en un montículo. Las actualizaciones necesarias del montículo tomarán $\mathcal{O}(\log n)$ tiempo cada una.

Entre dos puntos de parada, por definición no hay ninguna intersección, y además, el subconjunto de segmentos cuales quedan "bajo" de la línea de barrer, o sea, los segmentos que intersectan con la línea de barrer mientras mueva de un punto de parada al siguiente *no cambia*. Además, el orden de los puntos de intersección en comparación a cualquier de los dos ejes está fijo.

Entonces, guardamos en un árbol binario balanceado los segmentos que quedan actualmente bajo de la línea de barrer. El árbol nos da acceso en tiempo $\mathcal{O}(\log n)$. Se insertarán según el orden de la coordenada y del punto de intersección del segmento con la línea de barrer y se actualizará el orden en las siguientes situaciones:

(I) cuando comienza un segmento, se inserta el segmento en el lugar adecuado en el árbol binario,

(II) cuando termina un segmento, se saca el segmento del árbol, y

(III) cuando se intersectan dos segmentos, el segmento que antes estaba arriba se cambia por abajo y viceversa — en este último caso también se imprime el punto de intersección encontrada a la salida.

Por el hecho que el orden está según la coordenada y, al momento de llegar a la intersección de dos segmentos s_i y s_j, necesariamente son "vecinos" en el árbol por lo menos justo antes de llegar al punto de intersección. Ser vecinos quiere decir que son vértices hoja del árbol que están uno al lado del otro en el nivel bajo. Entonces, al haber actualizado la estructura, lo único que tenemos que hacer es calcular las puntos de intersección de los vecinos actuales y añadirles en la cola de puntos de parada. El número máximo de puntos de parada nuevos encontrados al haber hecho una parada es dos. El cuadro 8.1 muestra el agoritmo en pseudocódigo.

La construcción del montículo al comienzo toma $\mathcal{O}(n)$ tiempo. En cada punto preprocesado se realiza una inserción o un retiro de un elemento de un árbol binario balanceado. Son en total $2n$ de tales operaciones. Juntas necesitan $\mathcal{O}(n \log n)$ tiempo. Se añade al máximo dos elementos al montículo por cada inserción y al máximo un elemento por cada retiro hecho. Cada operación necesita $\mathcal{O}(\log n)$ tiempo y son $\mathcal{O}(n)$ operaciones. Hasta ahora todo necesita $\mathcal{O}(n \log n)$ tiempo.

En los puntos de intersección se intercambian posiciones dos segmentos. Esto se puede implementar con dos retiros seguidos por dos inserciones al árbol, de costo $\mathcal{O}(\log n)$ cada uno. Además se inserta al máximo dos puntos al montículo, de costo $\mathcal{O}(\log n)$ por inserción. En total son k intersecciones, por lo cual se necesita $\mathcal{O}(k \log n)$ tiempo. El tiempo total es entonces

$$\mathcal{O}(n \log n) + \mathcal{O}(k \log n) = \mathcal{O}((n + k) \log n). \tag{8.1}$$

8.2. Dividir y conquistar

El método *dividir y conquistar* divide un problema grande en varios subproblemas tal que cada subproblema tiene la misma pregunta que el problema original, solamente con una instancia de entrada más simple. Después se soluciona todos los subproblemas de una manera recursiva. Las soluciones a los subproblemas están combinadas a formar una solución del problema entero. Para las instancias del tamaño mínimo, es decir, las que ya no se divide recursivamente, se utiliza algún procedimiento de solución simple. La idea es que el tamaño mínimo sea constante y su solución lineal en su tamaño por lo cual la solución de tal instancia también es posible en tiempo constante desde el punto de vista del método de solución del problema entero.

Para que sea eficiente el método para el problema entero, además de tener un algoritmo de tiempo constante para las instancias pequeñas básicas, es importante que el costo computacional de dividir un problema a subproblemas sea bajo y también que la

Cuadro 8.1: Un algoritmo de línea de barrer para encontrar los puntos de intersección de un conjunto S de n segmentos en $\mathbb{R}^2$.

$M :=$ un montículo vacío
para todo $s_i \in S$,
    $s_i = \big((x_1, y_1), (x_2, y_2)\big)$ tal que $x_1 \leq x_2$
    insertar en M el dato $[(x_1, y_1), C, i, -]$
    insertar en M el dato $[(x_2, y_2), F, i, -]$
$B :=$ un árbol binario balanceado vacío
mientras M no está vacío
    $(x, y) :=$ el elemento mínimo de M
    remueve (x, y) de M
    si (x, y) es de tipo C como "comienzo"
        insertar s_i a B ordenado según la clave y
        si el vecino izquiero de s_i en B está definido y es s_ℓ
            computar la intersección (x', y') de s_i y s_ℓ
            si $x' > x$
                insertar en M el dato $[(x', y'), I, \ell, i]$
        si el vecino derecho de s_i en B está definido y es s_r
            computar la intersección (x'', y'') de s_i y s_r
            si $x'' > x$
                insertar en M el dato $[(x'', y''), I, i, r]$
    si (x, y) es de tipo F como "final"
        $s_\ell :=$ el segmento a la izquierda de s_i en B
        $s_r :=$ el segmento a la derecha de s_i en B
        remueve s_i de B
        si están definidos ambos s_ℓ y s_r,
            computar la intersección (x', y') de s_ℓ y s_r
            si $x'' > x$
                insertar en M el dato $[(x', y'), I, \ell, r]$
    si (x, y) es de tipo "intersección"
        $i :=$ el primer índice definido en el dato
        $j :=$ el segundo índice definido en el dato
        intercambia las posiciones de s_i y s_j en B
        si el nuevo vecino izquiero de s_j en B está definido y está s_ℓ
            computar la intersección (x', y') de s_j y s_ℓ
            si $x' > x$
                insertar en M el dato $[(x', y'), I, j, \ell]$
        si el nuevo vecino derecho de s_i en B está definido y está s_r
            computar la intersección (x'', y'') de s_i y s_r
            si $x'' > x$
                insertar en M el dato $[(x'', y''), I, i, r]$
        imprimir (x, y)

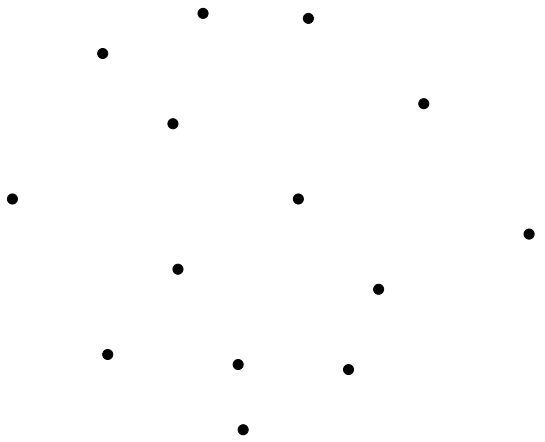

Figura 8.3: Una instancia del problema de cubierta convexa de un conjunto de puntos en el plano.

computación de juntar las soluciones de los subproblemas sea eficiente. Las divisiones a subproblemas genera un árbol abstracto, la altura de cual determina el número de niveles de división. Para lograr un árbol abstracto balanceado, normalmente es deseable dividir un problema a subproblemas de más o menos el mismo tamaño en vez de dividir a unos muy grandes y otros muy pequeños. Un buen ejemplo del método dividir y conquistar es ordenamiento por fusión que tiene complejidad asintótica $\mathcal{O}(n \log n)$.

8.2.1. Cubierta convexa

Como otro ejemplo del método dividir y conquistar, verémos la computación de la *cubierta convexa* de un conjunto de puntos en $\mathbb{R}^2$. La cubierta convexa es la *región convexa* mínima que contiene todos los puntos del conjunto. Una región es convexa si todos los puntos de un segmento de línea que conecta dos puntos incluidos en la región, también están incluidos en la misma región. Una instancia de ejemplo se muestra en la figura 8.3.

La idea del algoritmo es dividir iterativamente el conjunto de puntos en dos subconjuntos de aproximadamente el mismo tamaño, calcular la cubierta de cada parte y después juntar las soluciones de los subproblemas iterativamente a una cubierta convexa de todo el conjunto. Es bastante simple dividir el conjunto en dos partes tales que uno esté completamente a la izquierda del otro por ordenarlos según su coordenada x (en tiempo $\mathcal{O}(n \log n)$ para n puntos).

La cubierta de un sólo punto es el punto mismo. Para juntar dos cubiertas, el caso donde una está completamente a la izquierda de la otra es fácil: basta con buscar dos segmentos de "puente" que tocan en cada cubierta pero no corten ninguna (ver figura 8.4). Tales puentes se encuentra por examinar en orden los puntos de las dos cubiertas,

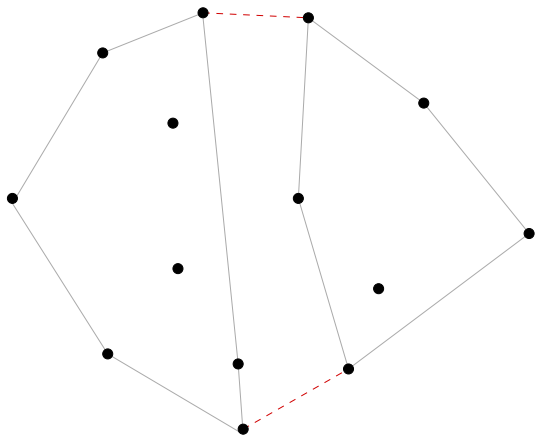

Figura 8.4: Dos "puentes" (en rojo) que juntan las soluciones de dos subproblemas de cubierta convexa a la solución del problema completo.

asegurando que la línea infinita definida por el segmento entre los dos puntos elegidos no corta ninguna de las dos cubiertas. También hay que asegurar que los dos puentes no corten uno al otro.

Un orden posible para evaluar pares de puntos como candidatos de puentes es empezar del par donde uno de los puntos maximiza la coordinada y en otro maximiza (o minimiza) la coordinada x. Hay que diseñar cómo avanzar en elegir nuevos pares utilizando alguna heurística que observa cuáles de las dos cubiertas están cortadas por el candidato actual. Lo importante es que cada punto está visitado por máximo una vez al buscar uno de los dos puentes.

La complejidad de tal algoritmo se caracteriza por la siguiente ecuación:

$$S(n) = \begin{cases} \mathcal{O}(1), & \text{si } n = 1, \\ 2S(\frac{n}{2}) + \mathcal{O}(n), & \text{en otro caso} \end{cases} \tag{8.2}$$

y su solución es $\mathcal{O}(n \log n)$.

Multiplicación de matrices

Otro ejemplo del método de dividir y conquistar es el *algoritmo de Strassen* para multiplicar dos matrices. Sean $A = (a_{ij})$ y $B = (b_{ij})$ matrices de dimensión $n \times n$. El algoritmo ingenuo para su multiplicación está basada en la formula $AB = C = (c_{ij})$ donde

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}. \tag{8.3}$$

La computación de cada c_{ij} toma tiempo Θn (n multiplicaciones y $n-1$ sumaciones) y son exactamente n^2 elementos, por lo cual la complejidad asintótica del algoritmo ingenuo es $\Theta(n^3)$. El otro algoritmo tiene la siguiente idea; sea $n = k^2$ para algún entero positivo k. Si $k = 1$, utilizamos el método ingenuo. En otro caso, divimos las matrices de entrada en cuatro partes:

$$\left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) \quad \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right) \tag{8.4}$$

tal que cada parte tiene dimensión $\frac{n}{2} \times \frac{n}{2}$. Para estas partes, calculamos sus multiplicaciones

$$\begin{aligned} &A_{11} \cdot B_{11}, \quad A_{12} \cdot B_{21}, \quad A_{11} \cdot B_{12}, \quad A_{12} \cdot B_{22}, \\ &A_{21} \cdot B_{11}, \quad A_{22} \cdot B_{21}, \quad A_{21} \cdot B_{12}, \quad A_{22} \cdot B_{22} \end{aligned} \tag{8.5}$$

y sumamos para obtener partes de la matriz C:

$$\begin{aligned} C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \\ C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21} \\ C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{aligned} \tag{8.6}$$

tal que

$$C = \left(\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right) \tag{8.7}$$

que es exactamente eñ producto AB.

Para analizar el algoritmo, marcamos con a el número de multiplicaciones hechas y con b el número de sumaciones. La complejidad de tiempo total es

$$T(n) \leq \begin{cases} a + b, & \text{si } k = 1 \\ aT(\frac{n}{2}) + b\,(\frac{n}{2})^2, & \text{para } k > 1. \end{cases} \tag{8.8}$$

La solucionamos por abrirla:

$$\begin{aligned}
T(2) &= a+b \\
xT(4) &= a^2+ab+b\left(\frac{4}{2}\right)^2 \\
T(8) &= a^3+a^2b+ab\left(\frac{4}{2}\right)^2+b\left(\frac{8}{2}\right)^2 \\
&\vdots \\
T(2^\ell) &= a^\ell+b\sum_{i=0}^{\ell-1}\left(\frac{2^{\ell-i}}{2}\right)^2 a^i \\
&= a^\ell+b\sum_{i=0}^{\ell-1}\frac{2^{2\ell-2i}}{4}\cdot a^i \\
&= a^\ell+b\frac{2^{2\ell}}{4}\sum_{i=0}^{\ell-1}\left(\frac{a}{4}\right)^i \\
&= a^\ell+b\frac{2^{2\ell}}{4}\frac{\left(\frac{a}{4}\right)^\ell-1}{\underbrace{\frac{a}{4}-1}_{>1,\text{si } a>4}} \\
&\leq a^\ell+ba^\ell \\
&= \mathcal{O}\left(a^\ell\right) \\
&= \mathcal{O}\left(a^{\log n}\right)=\mathcal{O}\left(2^{\log a\cdot\log n}\right)=\mathcal{O}\left(n^{\log a}\right).
\end{aligned}$$

En el algoritmo anterior tenemos $a=8$, por lo cual $T(n)=\mathcal{O}\left(n^3\right)$ y no hay ahorro asegurado. El truco del algoritmo de Strassen es lograr a tener $a=7$ y así lograr complejidad asintótica $\mathcal{O}\left(n^{\log 7}\right)\approx\mathcal{O}\left(n^{2,81}\right)$:

$$\begin{aligned}
S_1 &= (A_{12}-A_{22})\cdot(B_{21}+B_{22}) \\
S_2 &= (A_{11}+A_{22})\cdot(B_{11}+B_{22}) \\
S_3 &= (A_{11}-A_{21})\cdot(B_{11}+B_{12}) \\
S_4 &= (A_{11}+A_{12})\cdot B_{22} \\
S_5 &= A_{11}\cdot(B_{12}-B_{22}) \\
S_6 &= A_{22}\cdot(B_{21}-B_{11}) \\
S_7 &= (A_{21}+A_{22})\cdot B_{11}
\end{aligned}$$

tal que

$$\begin{aligned} C_{11} &= S_1 + S_2 - S_4 + S_6 \\ C_{12} &= S_4 + S_5 \\ C_{21} &= S_6 + S_7 \\ C_{22} &= S_2 - S_3 + S_5 - S_7, \end{aligned}$$

8.3. Podar-buscar

El método *podar-buscar* (inglés: prune and search) es parecido a dividir-conquistar, con la diferencia que después de dividir, el algoritmo ignora la otra mitad por saber que la solución completa se encuentra por solamente procesar en la otra parte. Una consecuencia buena es que tampoco hay que unir soluciones de subproblemas.

Como un ejemplo, analizamos la búsqueda entre n claves de la clave en posición i en orden decreciente de los datos. Para $i = \frac{n}{2}$, lo que se busca es el *mediana* del conjunto. Un algoritmo ingenuo cuadrático sería buscar el mínimo y eliminarlo i veces, llegando a la complejidad asintótica $\Theta(i \cdot n)$. Por aplicar un algoritmo de ordenación de un arreglo, llegamos a la complejidad asíntótica $\Theta(n \log n)$.

Un método podar-buscar parecido al ordenación rápida llega a complejidad mejor: elegimos un elemento pivote p y divimos los elementos en dos conjuntos: A donde las claves son menores a p y B donde son mayores o iguales. Continuamos de una manera recursiva solamente con uno de los dos conjuntos A y B: lo que contiene al elemento $i$ésimo en orden decreciente. Para saber dónde continuar, solamente hay que comprar i con $|A|$ y $|B|$.

El truco en este algoritmo es en la elección del elemento pivote así que la división sea buena: queremos asegurar que exista una constante q tal que $\frac{1}{2} \leq q < 1$ tal que el conjunto mayor de A y B contiene nq elementos. Así podríamos llegar a la complejidad

$$\begin{aligned} T(n) &= T(qn) + \Theta(n) \\ &\leq cn \sum_{i=0}^{\infty} q^i = \frac{cn}{1-q} \\ &= \mathcal{O}(n). \end{aligned}$$

El método de elección del pivote puede ser por ejemplo el siguiente:

1. Divide los elementos en grupos de cinco (o menos en el último grupo).
2. Denota el número de grupos por $k = \lceil \frac{n}{5} \rceil$.
3. Ordena en tiempo $\mathcal{O}(5) \in \mathcal{O}(1)$ cada grupo.

Cuadro 8.2: Un algoritmo para elegir un pivote que divide un conjunto tal que ninguna de las dos partes es tiene menos que una cuarta parte de los elementos.

procedimiento pivote($i \in \mathbb{Z}$, $D \subseteq \mathbb{R}$)
  si $|D| < 20$
    ordenar(D);
    devuelve $D[i]$
  en otro caso
    Dividir D en $\lceil |D|/5 \rceil$ grupos de cinco elementos
    Ordena cada grupo de cinco elementos;
    $M :=$ las medianes de los grupos de cinco;
    $p \leftarrow$ pivote($\lceil |M|/2 \rceil, M$);
    Dividir D a A y B tal que
      $a \in A$ si $a < p$;
      $b \in B$ si $b \geq p$;
    si $|A| \geq i$ **devuelve** pivote(i, A)
      en otro caso devuelve pivote($i - |A|, |B|$)

4. Elige la mediana de cada grupo.

5. Entre las k medianas, elige su mediana p utilizando este mismo algoritmo recursivamente.

6. El elemento p será el pivote.

De esta manera podemos asegurar que por lo menos $\lfloor \frac{n}{10} \rfloor$ medianas son mayores a p y para cada mediana mayor a p hay dos elementos mayores más en su grupo, por lo cual el número máximo de elementos menores a p son

$$3\lfloor \frac{n}{10} \rfloor < \frac{3}{4}n \text{ para } n \geq 20. \tag{8.9}$$

También sabemos que el número de elementos mayores a p es por máximo $\frac{3}{4}n$ para $n \geq 20$, por lo cual aplica que

$$\frac{n}{4} \leq p \geq \frac{3n}{4} \text{ para } n \geq 20. \tag{8.10}$$

El pseudocódigo del algoritmo está en el cuadro 8.2.

Para analizar la complejidad asintótica $T(n)$ del algoritmo desarrollado, hacemos las siguientes observaciones sobre el cuadro 8.2:

$$|M| = \lceil \frac{n}{5} \rceil \tag{8.11}$$

por lo cual la llamada recursiva `pivote`$(\lceil \frac{|M|}{2} \rceil, M)$ toma tiempo $T(\lceil \frac{n}{5} \rceil)$ por máximo. También sabemos que

$$\max\{|A|, |B|\} \leq \frac{3n}{4}, \tag{8.12}$$

por lo cual la llamada recursiva del último paso toma al máximo tiempo $T(\frac{3n}{4})$. Entonces, para alguna constante c tenemos que

$$T(n) = \begin{cases} c, & \text{si } n < 20 \\ T(\frac{n}{5} + T(\frac{3n}{4}) + cn, & \text{si } n \geq 20. \end{cases} \tag{8.13}$$

Con inducción llegamos bastante fácilmente a $T(n) \leq 20cn$.

8.4. Programación dinámica

En *programación dinámica*, uno empieza a construir la solución desde las soluciones de los subproblemas más pequeños, guardando las soluciones en una forma sistemática para construir soluciones a problemas mayores. Típicamente las soluciones parciales están guardadas en un arreglo para evitar a tener que solucionar un subprobelma igual más tarde el la ejecución del algoritmo. Su utilidad está en problemas donde la solución del problema completo contiene las soluciones de los subproblemas — una situación que ocurre en algunos problemas de *optimización*.

Un ejemplo básico es el cómputo de los coeficientes binómicos con ecuación 1.9 (en página 2). Si uno lo aplica de la manera dividir y conquistar, es necesario volver a calcular varias veces algunos coeficientes pequeños, llegando a la complejidad asintótica

$$\Omega\left(\binom{n}{k}\right) = \Omega\left(\frac{2^n}{\sqrt{n}}\right), \tag{8.14}$$

aunque guardando las soluciones parciales (o sea, cada coeficiente ya calculado), llegamos a la complejidad $\mathcal{O}(nk)$ (de tiempo — la de espacio crece). El arreglo del algoritmo de programación dinámica para los coeficientes binómicos resulta ser el *triángulo de Pascal* (cf. figura 8.5).

8.4.1. Triangulación óptima de un polígono convexo

Nuestro segundo ejemplo de programación dinámica es el siguiente problema:

Problema 22 (Polígono convexo)**.** *Dado:* un polígono convexo de n lados en formato de la lista de los $n + 1$ puntos finales de sus lados en la dirección del reloj. *Pregunta:* ¿cuál división del polígono a triángulos da la suma mínima de los largos de los lados de los triángulos?

n \ k	0	1	2	3	4	5	6	7	8	9	10	...
0	1											
1	1	1										
2	1	2	1									
3	1	3	3	1								
4	1	4	6	4	1							
5	1	5	10	10	5	1						
6	1	6	15	20	15	6	1					
7	1	7	21	35	35	21	7	1				
8	1	8	28	56	70	56	28	8	1			
9	1	9	36	84	126	126	84	36	9	1		
10	1	10	45	120	210	252	210	120	45	10	1	
⋮										⋱		

Figura 8.5: El inicio del triángulo de Pascal.

Si el número de los puntos es $n+1$, cada uno de los n lados del polígono tiene n opciones de asignación a triángulos, incluyendo un triángulo degenerado que consiste en el lado sólo (ver figura 8.7). Alguno de estos triángulos necesariamente pertenece a la triangulación óptima. El resto de la tarea es triangular el área o las áreas que quedan afuera del triángulo elegida.

Entonces, uno podría resolver el problema por examinar cada triangulación de cada lado, empezando del lado entre el primero y el segundo punto, y continuando recursivamente para los otros lados del polígono. Definimos como el "precio" de cada triángulo degenerado cero y suponemos que estamos triangulizando el polígono parcial definido por los puntos $i-1, i, \ldots, j-1, j$. El precio de la triangulación óptima es

$$T[i,j] = \begin{cases} 0, & \text{si } i = j \\ \min\limits_{i \le k \le j-1} \{T[i,k] + T[k+1,j] + w(i-1,k,j)\}, & i \neq j \end{cases} \tag{8.15}$$

donde $w(i-1,k,j)$ es el costo del triángulo definido por los tres puntos $i-1$, k y j.

Lo primero del algoritmo es guardar en la tabla de los $T[i,j]$ el valor cero para todo $T[i,i]$, $i = 1, \ldots, n$. La tabla se completa diagonal por diagonal hasta llegar al elemento $T[1,n]$ que será el costo de la triangulación óptima de todo el polígono. Habrá que recordar cuáles $w(i-1,k,j)$ fueron utilizados para saber de cuáles triángulos consiste la triangulación. El tiempo de cómputo por elemento de la tabla es $\Theta(n)$ y son $\Theta(n^2)$ elementos en total, por lo cual el algoritmo corre en tiempo $\Theta(n^3)$.

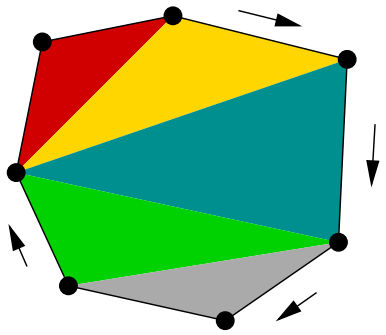

Figura 8.6: Una instancia de triangulación de un polígono convexo con una solución factible (aunque su optimalidad depende de cuál función objetivo está utilizada — la de la formulación presentada u otra. Los puntos están dibujados como círculos negros, mientras el polígono está dibujado en línea negra y los triángulos en colores variados.

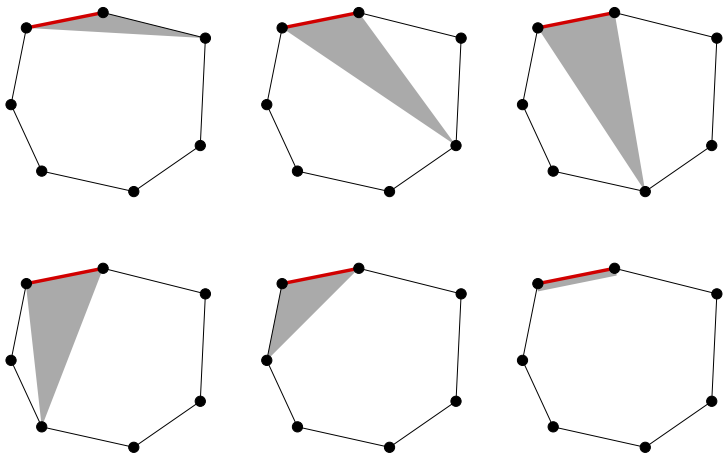

Figura 8.7: Las opciones de n triángulos los cuales puede pertenecer un lado del polígono en una triangulación óptima.

Capítulo 9

Optimización combinatoria

En *optimización combinatoria*, la manera ingenua de solucionar problemas es hacer una lista completa de todas las soluciones factibles y evaluar la función objetivo para cada una, eligiendo al final la solución cual dio el mejor valor. La complejidad de ese tipo de solución es por lo menos $\Omega(|F|)$ donde F es el conjunto de soluciones factibles. Desafortunadamente el número de soluciones factibles suele ser algo como $\Omega(2^n)$, por lo cual el algoritmo ingenuo tiene complejidad asintótica exponencial. Si uno tiene un método eficiente para generar en una manera ordenada soluciones factibles y rápidamente decidir sí o no procesarlos (en el sentido de podar-buscar), no es imposible utilizar un algoritmo exponencial.

Otra opción es buscar por soluciones *aproximadas*, o sea, soluciones cerca de ser óptima sin necesariamente serlo (ver capítulo 10). Una manera de aproximación es utilizar *métodos heurísticos*, donde uno aplica una regla simple para elegir candidatos. Si uno siempre elije el candidatos que desde el punto de vista de evaluación local se ve el mejor, la heurística es *voraz* (también se dice *glotona*).

Muchos problemas de optimización combinatoria consisten de una parte de construcción de cualquier solución factible, que tiene la misma complejidad que el problema de decisión de la *existencia* de una solución factible. No es posible que sea más fácil solucionar el problema de optimización que el problema de decisión a cual está basado.

Formalmente, para mostrar que un problema de optimización es difícil, lo transformamos en un problema de decisión y demostramos que el problema de decisión es **NP**-completo (o maás difícil). El forma "normal" de la transformación es la siguiente (para problemas de minimización — la extensión a problemas de maximización es fácil):

Problema 23 (Opt Dec)**.** *Dada:* una instancia x de un problema de optimización y un costo c. *Pregunta:* ¿existe una solución $i \in \mathcal{S}_x$ con costo menor o igual a c?

En un sentido, si el problema de decisión es **NP**-completo, el problema de optimización es **NP**-duro, porque si tenemos un algoritmo para el problema de optimización,

podemos con uan reducción trivial decidir el problema de decisión, y como problemas de optimización por definición no pertenecen a **NP** (**NP** siendo una clase de problemas de decisión), en vez de ser **NP**-completos, son **NP**-duros.

De los libros de texto sobre optimización combinatoria cabe mencionar el libro de Papadimitriou y Steiglitz [16].

9.1. Árbol cubriente mínimo

Estudiamos algunos algoritmos de encontrar un árbol cubriente mínimo en grafos ponderados no dirigidos. Para construir un árbol cubriente cualquiera – o sea, una solución factible — podemos empezar de cualquier vértice, elegir una arista, y continuar al vecino indicado, asegurando al añadir aristas que nunca regresamos a un vértice ya visitado con anterioridad. En este problema, logramos a encontrar la solución óptima con una heurística voraz: siempre elige la arista con menor peso para añadir en el árbol que está bajo construcción.

En el *algoritmo de Prim* , empezamos por incluir en el árbol la arista de peso mínimo y marcando los puntos finales de esta arista. En cada paso, se elige entre los vecinos de los vértices marcados el vértice que se puede añadir con el menor peso entre los candidatos. En una implementación simple, se guarda el "costo" de añadir un vértice en un arreglo auxiliar $c[v]$ y asignamos $c[v] = \infty$ para los que no son vecinos de vértices ya marcados. Para saber cuales vértices fueron visitados, se necesita una estructura de datos. Con montículos normales se obtiene complejidad de $\mathcal{O}(m \log n)$ y con montículos de Fibonacci complejidad de $\mathcal{O}(m + n \log n)$.

Otra posibilidad es empezar a añadir aristas, de la menos pesada a la más pesada, cuidando a no formar ciclos por marcar vértices al haberlos tocado con una arista. El algoritmo termina cuando todos los vértices están en el mismo árbol, por lo cual una estructura tipo unir-encontrar resulta muy útil. Este método se conoce como el *algoritmo de Kruskal* y su complejidad es $\mathcal{O}(m \log m) + \mathcal{O}(m \cdot (\text{ unir-encontrar })) = \mathcal{O}(m \log m) = \mathcal{O}(m \log n)$.

9.2. Ramificar-acotar

En los algoritmos de optimización combinatoria que evalúan propiedades de varios y posiblemente todos los candidatos de solución, es esencial saber "guiar" la búsqueda de la solución y evitar evaluar "candidatos malos". Un ejemplo de ese tipo de técnica es el método *podar-buscar* (de la sección 8.3). Algoritmos que avancen siempre en el candidato *localmente óptimo* se llaman *voraces* (por ejemplo los algoritmos de construcción de árboles cubriente de peso mínimo de la sección 9.1).

Un método de este tipo es el algoritmo "vuelta atrás" (inglés: backtracking). Su idea es aumentar una solución parcial utilizando candidatos de aumento. En cuanto una solución está encontrada, el algoritmo vuelve a examinar un ramo de aumento donde no todos los candidatos han sido examinados todavía. Cuando uno utiliza *cotas* para decidir cuáles ramos dejar sin explorar, la técnica se llama *ramificar-acotar* (inglés: branch and bound).

Es recomendable utilizar métodos tipo ramificar-acotar solamente en casos donde uno no conoce un algoritmo eficiente y no basta con una solución aproximada. Los ramos de la computación consisten de soluciones factibles distintas y la subrutina para encontrar una cota (superior para maximización y inferior para minimización) debería ser rápida. Normalmente el recorrido del árbol de soluciones factibles se hace en profundidad — cada hoja del árbol corresponde a una solución factible, mientras los vértices internos son las operaciones de aumento que construyen las soluciones factibles. El algoritmo tiene que recordar el mejor resultado visto para poder eliminar ramos que por el valor de su cota no pueden contener soluciones mejores a la ya conocida.

9.2.1. Problema de viajante

En la versión de optimización del problema de viajante (TSP), uno busca por el ciclo de menor costo/peso en un grafo ponderado. En el caso general, podemos pensar que el grafo sea no dirigido y completo. Utilizamos un método tipo ramificar-acotar para buscar la solución óptima. El árbol de soluciones consiste en decicir para cada uno de los $\binom{n}{2}$ aristas del grafo sí o no está incluida en el ciclo. Para el largo $\mathcal{L}(R)$ de cualquier ruta R aplica que

$$\mathcal{L}(R) = \frac{1}{2}\sum_{i=1}^{n} \left(\mathcal{L}(v_{i-1}, v_i) + \mathcal{L}(v_i, v_{i+1})\right), \tag{9.1}$$

donde los vértices de la ruta han sido numerados según su orden de visita en la ruta tal que el primer vértice tiene dos números v_1 y v_{n+1} y el último se conoce como v_n y v_0 para dar continuidad a la ecuación. Además sabemos que para cualquier ruta R el costo de la arista incidente a cada vértice es por lo menos el costo de la arista más barata incidente a ese vértice, por lo cual para la ruta más corta $R_{\text{mín}}$ aplica que

$$\mathcal{L}(R_{\text{mín}}) \geq \tfrac{1}{2}\sum_{v\in V} \text{ suma de los largos de las dos aristas más baratas incidentes a } v. \tag{9.2}$$

Suponemos que el orden de procesamiento de las aristas es fijo. Al procesar la arista $\{v, w\}$, el paso "ramificar" es el siguiente:

1. Si al *excluir* $\{v, w\}$ resultaría que uno de los vértices v o w tenga menos que dos aristas incidentes para la ruta, ignoramos el ramo de excluirla.

2. Si al *incluir* $\{v, w\}$ resultaría que uno de los vértices v o w tenga más que dos aristas incidentes para la ruta, ignoramos el ramo de inclusión.

3. Si al *incluir* $\{v, w\}$ se generaría un ciclo en la ruta actua sin haber incluido todos los vértices todavía, ignoramos el ramo de inclusión.

Después de haber eliminado o incluído aristas así, computamos un nuevo valor de $R_{\text{mín}}$ para las elecciones hechas y lo utilizamos como la cota inferior. Si ya conocemos una solución mejor a la cota así obtenida para el ramo, ignoramos el ramo.

Al cerrar un ramo, regresamos por el árbol (de la manera $\textsc{Dfs}$) al nivel anterior que todavía tiene ramos sin considerar. Cuando ya no queda ninguno, el algoritmo termina.

Capítulo 10

Algoritmos de aproximación

En situaciones donde todos los algoritmos conocidos son lentos, vale la pena considerar la posilibidad de usar una solución *aproximada*, o sea, una solución que tiene un valor de la función objetivo *cerca* del valor óptimo, pero no necesariamente el óptimo mismo. Depende del área de aplicación sí o no se puede hacer esto eficientemente. En muchos casos es posible llegar a una solución aproximada muy rápidamente mientras encontrar la solución óptima puede ser imposiblemente lento. Un algoritmo de aproximación puede ser determinista o no determinista — en segundo caso será atentido el el capítulo 11. Si el algoritmo de aproximación no es determinista y ejecuta muy rápidamente, es común ejecutarlo varias veces y elegir el mejor de las soluciones aproximadas así producidas.

Un algoritmo de aproximación bien diseñado cuenta con un análisis formal que muestra que la diferencia entre su solución y la solución óptima es de un factor constante. Este factor constante se llama el *factor de aproximación* y es menor a uno para problemas de maximización y mayor a uno para problemas de minimización. Depende de la aplicación qué tan cerca debería ser la solución aproximada a la solución óptima. El valor extremo de ese factor sobre el conjunto de todas las instancias del problema (el mínimo para problemas de maximización y el máximo para los de minimización) se llama la *tasa* o *índice de aproximación* (inglés: approximation ratio). Un algoritmo de aproximación tiene *tasa constante* si el valor de la solución encontrada es por máximo un múltiple constante del valor óptimo.

También habrá que mostrar formalmente que el algoritmo de aproximación tiene complejidad polinomial (o polinomial con alta probabilidad en el caso no determinista). Si existe un método sistemático para aproximar la solución a factores arbitrarios, ese método se llama una *esquema de aproximación (de tiempo polinomial)* (inglés: (polynomial-time) approximation scheme). En el caso de tiempo polinomial se utiliza la abreviación PTAS. Un libro de texto recomendable sobre algoritmos de aproximación es lo de Vazirani [17].

10.1. Ejemplos

En *problema de la mochila* (inglés: knapsack) es un problema clásico de optimización combinatoria:

Problema 24 (Knapsack)**.** *Dada:* una lista de N diferentes *artículos* $\varphi_i \in \Phi$ donde cada objecto tiene una *utilidad* $\nu(\varphi_i)$ y un *peso* $\psi(\varphi_i)$ y una *mochila* que soporta peso hasta un cierto límite Ψ. *Pregunta:* ¿cómo elegir un conjunto de artículos $M \subseteq \Phi$ con la restricción

$$\Psi \geq \sum_{\varphi \in M} \psi(\varphi)$$

tal que

$$\max_{M \subseteq \Phi} \left\{ \sum_{\varphi \in M} \nu(\varphi) \right\},$$

o sea, la *utilidad total* es máxima?

Para solucionar este problema através de *programación entera* asignamos para cada artículo φ_i una variable binaria x_i tal que

$$x_i = \begin{cases} 1, & \text{si} \quad \varphi \in M, \\ 0, & \text{si} \quad \varphi \notin M. \end{cases} \tag{10.1}$$

La función objetivo es la suma de utilidades y su coeficiente es $\nu(\varphi_i)$:

$$f(\mathbf{x}) = \max_{\mathbf{x}} \left\{ \sum_{i=1}^{N} \nu(\varphi_i) \cdot x_i \right\}. \tag{10.2}$$

Una restricción limita la suma de pesos tal que $b_1 = \Psi$ y $a_{N+1,i} = \psi(\varphi_i)$):

$$\sum_{i=1}^{N} a_{N+1,i} \cdot x_i. \tag{10.3}$$

En una versión generalizada del problema de la mochila, tenemos una cierta cantidad c_i del artículo φ_i disponible. Entonces las variables ya no son binaria, sino enteras en general, y aplican restricciones para cada variable x_i que representa la inclusión de un cierto tipo de artículo en M:

$$(x_i \in \mathbb{Z}) \wedge (0 \leq x_i \leq c_i). \tag{10.4}$$

La función objetivo y la restricción de peso total se formula igual como en el caso de 0–1 del problema original. Una versión aun más general no tiene límite superior:

$$(x_i \in \mathbb{Z}) \wedge (x_i \geq 0). \tag{10.5}$$

En el *problema de empaquetear a cajas* (inglés: bin packing) tenemos un conjunto finito de objetos $\Phi = \{\varphi_1, \varphi_2 \ldots, \varphi_N\}$, cada uno con un *tamaño* definido $t(\varphi_i) \in \mathbb{R}$. Los objetos habrá que empaquetear en cajas de tamaño fijo T tal que

$$T \geq \max\{t(\varphi_i) \mid \varphi_i \in \Phi\} \tag{10.6}$$

y queremos encontrar un orden de empaquetear que minimiza el número de cajas utilizadas. Este problema es **NP**-completo.

Un algoritmo de aproximación simple y rápido empieza por ordenar las cajas en una fila. Procesamos los objetos en orden. Primero intentamos poner el objecto actualmente procesado en la primera caja de la fila. Si cabe, lo ponemos allí, y si no, intentamos en la siguiente caja. Iterando así obtenemos alguna asignación de objetos a cajas. Denotamos con $\mathrm{OPT}(\Phi)$ el número de cajas que contienen por lo menos un objeto en la asignación óptima. Se puede mostrar que el algoritmo de aproximación simple utiliza al máximo $\frac{17}{10}\,\mathrm{OPT}(\Phi) + 2$ cajas. Esto significa que nunca alejamos a más de 70 por ciento de la solución óptima. Podemos mejorar aún por ordenar los objetos tal que intentamos primero el más grande y después el segundo más grande, en cual caso se puede mostrar que llegamos a utilizar al máximo $\frac{11}{9}\,\mathrm{OPT}(\Phi)+4$ cajas, que nos da una distancia máxima de unos 22 por ciento del óptimo.

Si tomamos una versión del problema del viajante donde los pesos un grafo completo ponderado son *distancias* entre los vértices $d(v, w)$ que cumplen con la *desigualdad de triángulo*

$$d(v, u) \leq d(v, w) + d(w, u), \tag{10.7}$$

que también es un problema **NP**-completo. El algoritmo siguiente encuentra en tiempo polinomial una solución aproximada, o sea, imprime una lista de vértices que define un orden de visitas que efectivamente fija la ruta R:

1. Construye un árbol cubriente mínimo en tiempo $\mathcal{O}\,(m \log n)$.
2. Elige un vértice de inicio cualquiera v.
3. Recorre el árbol con $\textsc{Dfs}$ en tiempo $\mathcal{O}\,(m + n)$ e imprime cada vértice a la primera visita (o sea, en preorden).
4. Imprime v en tiempo $\mathcal{O}\,(1)$.

El $\textsc{Dfs}$ recorre cada arista del árbol dos veces; podemos pensar en el recorrido como una ruta larga R' que visita cada vértice por lo menos una vez, pero varias vértices más de una vez. Por "cortar" de la ruta larga R' cualquier visita a un vértice que ya ha sido visitado, logramos el efecto de imprimir los vértices en preorder. Por la desigualdad de triángulo, sabemos que la ruta R no puede ser más cara que la ruta larga R'. El costo total de R' es dos veces el costo del árbol cubriente mínimo.

Para lograr a comparar el resultado con el óptimo, hay que analizar el óptimo en términos de árboles cubrientes: si eliminamos cualquier arista de la ruta óptima R_{OPT}, obtenemos un árbol cubriente. El peso de este árbol es por lo menos el mismo que el peso de un árbol cubriente mínimo C. Entonces, si marcamos el costo de la ruta R con $c(R)$, hemos mostrado que necesariamente

$$c(R) \leq c(R') = 2C \leq 2c(R_{\mathrm{OPT}}). \tag{10.8}$$

10.2. Búsqueda local

Cuando hemos obtenido una solución heurística y aproximada de manera cualquiera a un problema de optimización, podemos intentar mejorarla por *búsqueda local* donde uno aplica operaciones pequeñas y rápidamente realizadas para causar cambios pequeños en la solución así que la solución mantiene factible y puede ser que mejora. Libros buenos de búsqueda local incluyen el libro de de Aarts y Lenstra [1] y el libro de Hoos y Stützle [9]. Las tres variaciones típicas para guiar una búsqueda local son

1. búsqueda voraz (inglés: greedy local search; hill-climbing),
2. búsqueda tabu (inglés: tabu search), y
3. recocido simulado (inglés: simulated annealing).

10.2.1. Definiciones básicas

En esta sección, resumimos las definiciones utilizadas en el libro de texto editado por Aarts y Lenstra [1]. Un *problema* combinatorial se define por un conjunto específico de *instancias* que son objetos combinatoriales y de la tarea de *minimizar* o *maximizar*.

Una instancia *de* un problema combinatorial es un par $(\mathcal{S}, f)$ donde $\mathcal{S}$ es un conjunto de soluciones *factibles* y $f : \mathcal{S} \mapsto \mathbb{R}$ es una función objetivo. Sin pérdida de generalidad, consideramos solamente problemas de minimización, ya que uno puede convertir cualquier problema de maximización a una problema de minimización por minimizar $-f$ en vez de maximizar f.

La tarea relacionada a la instancia del problema es encontrar una solución *globalmente óptima* $i^* \in \mathcal{S}$ tal que $f(i^* \leq f(i)$ para todo $i \in \mathcal{S}$. Denotamos por $f^* = f(i^*)$ el costo de la solución óptima y por $\mathcal{S}^*$ el conjunto de soluciones globalmente óptimas.

Típicamente, uno cuenta con una representación compacta del conjunto de las soluciones factibles y un algoritmo polinomial para verificar si $i \in \mathcal{S}$ y (posiblemente otro algoritmo polinomial) para calcular el valor $f(i)$ para cualquier $i \in \mathcal{S}$.

Una función de *vecindario* es un mapeo $\Gamma : \mathcal{S} \mapsto 2^{\mathcal{S}}$ que define para cada $i \in \mathcal{S}$ un conjunto de *vecinos* $\Gamma(i) \subseteq \mathcal{S}$ tal que los vecinos de una solución i son en algún sentido "cercanos" a i.

Una búsqueda local voraz empieza con una solución inicial arbitraria y mueve iterativamente siempre al vecino con menor costo hasta que ya no exista tal vecino. Cada solución que no tiene ningún vecino con menor costo es un *óptimo local*. Se dice que la función Γ es *exacta* para $(\mathcal{S}, f)$ si el conjunto de soluciones localmente óptimas es igual al conjunto de soluciones globalmente óptimas. Por ejemplo, el vecindario del algoritmo Simplex es exacto y Simplex es una búsqueda local voraz.

Una opción que evita tener que evaluar el costo de todos los vecinos en cada paso es agarrar el primer vecino que ofrece una mejora. Igualmente, si no existe tal vecino, la solución actual es una óptimo local.

Vecindarios típicos están construidos por intercambiar elementos de las instancias combinatoriales (por ejemplo la presencia y ausencia de un elemento o las posiciones de dos elementos).

10.2.2. Ejemplo: 2-opt

En esta sección veremos un ejemplo de las operaciones de modificación: la *la elección de pares*, abreviada comúnmente como *2-opt*, aplicada en el problema del viajante en un grafo ponderado no dirigido $G = (V, E)$; denotamos el costo de una arista $\{v, w\}$ por $c(v, w) > 0$. Elegimos (al azar) dos aristas de la ruta R, sean $\{s, t\}$ y $\{u, v\}$. Marcamos el segmento de la ruta entre t y u por A y el otro segmento entre v y s por B tal que

$$\begin{aligned} A &= [tr_1r_2 \dots r_k u] \\ B &= [vw_1w_2 \dots w_\ell s] \\ R &= [tr_1r_2 \dots r_k uvw_1w_2 \dots w_\ell st] \end{aligned} \tag{10.9}$$

Si el grafo G *también* contiene las aristas $\{v, t\}$ y $\{s, u\}$, se evalúa si

$$c(s, t) + c(u, v) > c(s, u) + c(v, t). \tag{10.10}$$

En el caso que esto es verdad, podemos llegar a un costo total menor por reemplazar las aristas originales $\{s, t\}$ y $\{u, v\}$ en la ruta R por las aristas más baratas $\{v, t\}$ y $\{s, u\}$, creando una ruta nueva R' con costo total menor tal que

$$R' = [tr_1r_2 \dots r_k usw_\ell w_{\ell-1} \dots w_2w_1vt]. \tag{10.11}$$

La figura 10.1 ilustra el intercambio realizado.

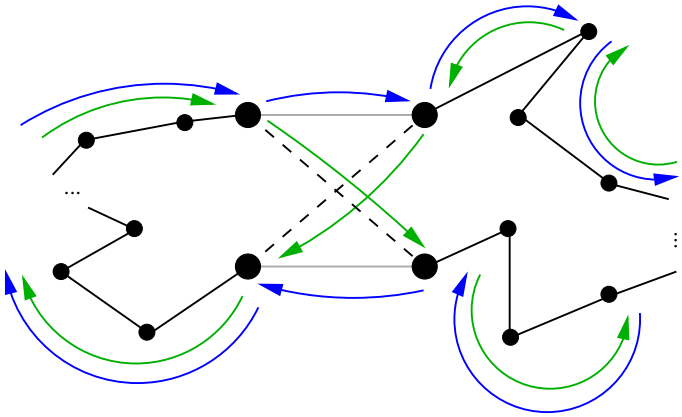

Figura 10.1: Una modificación 2-opt para el problema del viajante: la ruta original R está marcada por las flechas azules y la ruta más económica R' por las flechas verdes; las aristas eliminadas están en gris y las añadidas en línea negro descontínua. Los cuatro vertices que son puntos finales de las aristas involucradas están dibujados en mayor tamaño que los otros vértices.

10.2.3. Complejidad de búsqueda local

Consideramos ahora el problema siguiente por contraejemplo:

Problema 25 (LOCAL OPT)**.** *Dada:* una instancia $(\mathcal{S}, f)$ y una solución factible $i \in \mathcal{S}$. *Pregunta:* ¿es i localmente óptima?

Si i no es localmente óptima, habrá que presentar un vecino suyo con menor costo como un contraejemplo. La clase **PLS** fue diseñado para capturar la complejidad de búsqueda local. Un problema de búsqueda local L pertenece a **PLS** si existen tres algoritmos de tiempo polinomial A_L, B_L y C_L tales que.

(I) Dada una palabra x del lenguaje $\{0,1\}^*$, A_L determina si x es una representación binaria de una instancia $(\mathcal{S}_x, f_x)$ de L, y si lo es, produce alguna solución factible $i_0 \in \mathcal{S}_x$.

(II) Dada una instancia x (en su representación binaria) y un i (en representación binaria), el algoritmo B_L determina si $i \in \mathcal{S}_x$, y si lo es, calcula el costo $f_x(i)$.

(III) Dada una instancia $x = (\mathcal{S}_x, f_x, \Gamma_x)$ y una solución i, el algoritmo C_L determina si i es un óptimo local, y si *no* lo es, produce un vecino $i \in \Gamma_x$ con costo (estrictamente) mejor al costo de i.

Podemos definir versiones de búsqueda de las clases **P** y **NP**. La clase $\mathbf{NP}_S$ consiste de relaciones $\mathcal{R} \subseteq \{0,1\}^* \times \{0,1\}^*$ tal que para una relación $\mathcal{R}$,

1. si $(x, y) \in \mathcal{R}$, $|y| = \mathcal{O}\left(\left(\right)\right. g(|x|)$ donde g es un polinomio (polinomialmente *acotada*), y

2. existe un algoritmo polinomial que determina si un dado par (x, y) pertenece a $\mathcal{R}$ (polinomialmente *reconocida*).

Si además existe un algoritmo polinomial que *decide* a $\mathcal{R}$ (o sea, dado x, produce un $(x, y) \in \mathcal{R}$ si tal par existe y reporta "no" en otro caso), la relación pertenece a la clase $\mathbf{P}_S$. Aplica que $\mathbf{P}_S = \mathbf{NP}_S$ si y sólo si $\mathbf{P} = \mathbf{NP}$. Además, se puede demostrar que

$$\mathbf{P}_S \subseteq \mathbf{PLS} \subseteq \mathbf{NP}_S, \tag{10.12}$$

aunque no se sabe si son subconjuntos propios. Parece improbable que sean iguales $\mathbf{P}_S$ y **PLS** y hay fuertes indicaciones (aunque no demostración formal) que **PLS** no coindica con $\mathbf{NP}_S$.

También han definido reducciones para la clase **PLS** y encontrado problemas completos, como por ejemplo TSP y MAXCUT.

Capítulo 11

Algoritmos aleatorizados

Algoritmos aleatorizados son algoritmos que incorporan además de instrucciones deterministas algunas elecciones al azar. Además del diseño de algoritmos aleatorizados, probabilidades también juegan un papel importante en análisis del caso promedio (sección 7.3).

Un ejemplo simple de un algoritmo aleatorio sería una versión de la ordenación rápida donde el elemento pivote está elegido uniformemente al azar entre los elementos para ordenar.

En esta sección esperamos que el lector tenga conocimientos básicos de probabilidad — buenos libros de texto sobre probabilidad incluyen el libro de Milton y Arnold [12]. Según nuestra notación, X y Y son variables aleatorias, x y y son algunas valores que pueden tomar estas variables, $\Pr[X = x]$ es la probabilidad que X tenga el valor x, $\mathrm{E}[X]$ es la *esperanza* (matemática) (tambin: valor esperado)

$$\mathrm{E}[X] = \sum_x x \Pr[X = x], \tag{11.1}$$

donde la sumación extiende por todo x en el rango de X, y $\mathrm{Var}[X]$ la varianza

$$\mathrm{Var}[X] = \mathrm{E}\left[(X - \mathrm{E}[X])^2\right] = \mathrm{E}\left[X^2\right] - (\mathrm{E}[X])^2. \tag{11.2}$$

Al diseñar un algoritmo que haga elecciones al azar, el análisis del comportamiento del algoritmo necesariamente involucra calculaciones de probabilidades para determinar propiedades tales como

- ¿Con qué probabilidad el algoritmo da el resultado correcto?
- ¿Qué es la esperanza del número de los pasos de computación necesarios para su ejecución?

- (En el caso de algoritmos de aproximación aleatorizadas:) ¿Qué es la deviación esperada de la solución óptima?

Una observación interesante aquí es que estamos utilizando una definición más "flexible" de algoritmo: un algoritmo aleatorizado tiene "permiso" para dar una salida incorrecta, mientras al definir algoritmos (deterministas) exigimos que siempre termine su ejecución y que el resultado sea el deseado. Sin embargo, no *todos* los algoritmos aleatorizados dan resultados incorrectos — el caso ideal sería que un resultado incorrecto sea imposible o que ocurra con probabilidad muy pequeña.

Algoritmos con una probabilidad no cero de fallar (y probabilidad no cero de dar el resultado correcto) se llaman *algoritmos Monte Carlo*. Un algoritmo de decisión tipo Monte Carlo tiene *error bilateral* si la probabilidad de error es no cero para los ámbos casos: con la respuesta "si" y con la respuesta "no". Tal algoritmo tiene *error unilateral* si siempre contesta correctamente en uno de los dos casos pero tiene probabilidad no cero de equivocarse en el otro.

Un algoritmo que siempre da el resultado correcto se dice un *algoritmo Las Vegas*. En tal algoritmo, la parte aleatoria está limitada al tiempo de ejecución del algoritmo, mientras los algoritmos Monte Carlo tienen hasta sus salidas "aleatorias".

Si tenemos un algoritmo Monte Carlo con probabilidad de error $0 < p < 1$, lo podemos convertir a un algoritmo Monte Carlo con probabilidad de error arbitrariamente pequeña $0 < q \leq p$ simplemente por *repetir* ejecuciones independientes del algoritmo original k veces tal que $p^k \leq q$.

Cada algoritmo Monte Carlo que sabe distinguir entre haberse equivocado se puede convertir a un algoritmo Las Vegas por repertilo hasta obtener éxito — este tiene obviamente efectos en su tiempo de ejecución. También si tenemos un algoritmo rápido para *verificar* si una dada "solución" es correcta (cf. los certificados concisos de los problemas $\mathbf{NP}$), podemos convertir un algoritmo Monte Carlo a un algoritmo Las Vegas por repetirlo.

Resumimos ahora algunos resultados de la teoría de probabilidad que pueden resultar útiles al contestar las preguntas mencionadas. El libro de texto de Mitzenmacher y Upfal [13] contiene varios ejemplos concretos de su uso, igual como el libro de Motwani y Raghavan [14].

Desigualdad de Jensen Si f es una función convexa, $\mathrm{E}[f(X)] \geq f(\mathrm{E}[X])$.

Desigualdad de Markov Sea X una variable aleatoria no negativa. Para todo $\alpha > 0$, $\Pr[X \geq \alpha] \leq \alpha^{-1} \mathrm{E}[X]$.

Desigualdad de Chebyshev Para todo $\alpha > 0$, $\Pr[|X - \mathrm{E}[X]| \geq \alpha] \leq \alpha^{-2} \mathrm{Var}[X]$.

Cotas de Chernoff Para $\alpha > 0$, para todo $t > 0$ aplica que $\Pr[X \geq \alpha] \leq e^{-t\alpha} \mathrm{E}\left[e^{tX}\right]$ y para todo $t < 0$ aplica que $\Pr[X \leq \alpha] \leq e^{-t\alpha} \mathrm{E}\left[e^{tX}\right]$.

El método probabilista Sea X una variable aleatoria definida en el espacio de probabilidad $\mathcal{S}$ tal que $\mathrm{E}[X] = \mu$. Entonces $\Pr[X \geq \mu] > 0$ y $\Pr[X \leq \mu] > 0$.

El método del momentum segundo Sea X una variable aleatoria entera. Entonces $\Pr[X = 0] \leq (\mathrm{E}[X])^{-2} \operatorname{Var}[X]$.

11.1. Complejidad computacional

La clase **RP** (tiempo polinomial aleatorizado, inglés: randomized polynomial time) es la clase de todos los lenguajes L que cuenten con un algoritmo aleatorizado A con tiempo de ejecución polinomial del peor caso tal que para cada entrada $x \in \Sigma^*$

$$\begin{aligned} x \in L &\impliedby \Pr[A(x) = \text{“sí”}] \geq p, \\ x \notin L &\impliedby \Pr[A(x) = \text{“sí”}] = 0, \end{aligned} \tag{11.3}$$

donde $p > 0$ (comúnmente se define $p = 0{,}5$, pero la elección del valor de p es de verdad arbitraria). El algoritmo A es entonces un algoritmo Monte Carlo con error unilateral. La clase $\mathbf{coRP}$ es la clase con error unilateral en el caso $x \notin L$ pero sin error con las entradas $x \in L$. La existencia de un algoritmo Las Vegas polinomial muestra que un lenguaje pertenece a las clases $\mathbf{RP}$ y $\mathbf{coRP}$ las dos. De hecho, esta clase de lenguajes con algoritmos Las Vegas polinomiales se denota por **ZPP** (tiempo polinomial aleatorizado cero-error, inglés: zero-error probabilistic polynomial time):

$$\mathbf{ZPP} = \mathbf{RP} \cap \mathbf{coRP}. \tag{11.4}$$

Una clase más débil es la **PP** (tiempo polinomial aleatorizada, inglés: probabilistic polynomial time) que consiste de los lenguajes L para las cuales existe un algoritmo aleatorizado A con tiempo de ejecución de peor caso polinomial y para todo $x \in \Sigma^*$ aplica que

$$\begin{aligned} x \in L &\impliedby \Pr[A(x) = \text{“sí”}] > \tfrac{1}{2}, \\ x \notin L &\impliedby \Pr[A(x) = \text{“sí”}] < \tfrac{1}{2}. \end{aligned} \tag{11.5}$$

Por ejecutar A varias veces, podemos reducir la probabilidad de error, pero no podemos garantizar que un número pequeño de repeticiones basta para mejorar significativamente la situación.

Una versión más estricta de **PP** se llama **BPP** (tiempo polinomial aleatorizado, inglés: bounded-error probabilistic polynomial time) que es la clase de los lenguajes L para las cuales existe un algoritmo aleatorizado A con tiempo de ejecución de peor caso

polinomial y para todo $x \in \Sigma^*$ aplica que

$$\begin{aligned} x \in L &\impliedby \Pr[A(x) = \text{“sí”}] \geq \tfrac{3}{4}, \\ x \notin L &\impliedby \Pr[A(x) = \text{“sí”}] \leq \tfrac{1}{4}. \end{aligned} \tag{11.6}$$

Para esta clase se puede demostrar que la probabilidad de error se puede bajar a 2^{-n} con $p(n)$ iteraciones donde $p()$ es un polinomio.

Problemas abiertos interesantes incluyen por ejemplo si **BPP** es un subclase de **NP**. Para más información sobre las clases de complejidad aleatorizada, recomendamos el libro de Papadimitriou [15] y el libro de Motwani y Raghavan [14].

11.2. Problema de corte mínimo

Como un ejemplo, veremos un algoritmo aleatorizado para el problema MinCut. Ahora vamos a considerar *multigrafos*, o sea, permitimos que entre un par de vértices exista más que una arista en el grafo de entrada G. Considerando que un grafo simple es un caso especial de un multigrafo, el resultado del algoritmo que presentamos aplica igual a grafos simples.

Estamos buscando un corte (ver sección 4.5.2) $C \subseteq V$ de G, o sea, una partición $(C, V \setminus C)$. La capacidad del corte es el número de aristas que lo crucen. Todo lo que mostramos aplicaría también para grafos (simples o multigrafos) ponderados con pesos no negativos.

Para que sea interesante el problema, habrá que suponer que la entrada G sea conexo (verificado por ejemplo por Dfs en tiempo polinomial) — con varios componentes, el corte mínimo con capacidad cero se obtiene por agrupar algunos componentes en C y los demás en $V \setminus C$.

El problema MinCut se puede resolver por un algoritmo que compute el flujo máximo del grafo de entrada. Eso, aunque es polinomial, suele ser algo lento en la práctica. El mejor algoritmo determinista conocido para el flujo máximo tiene complejidad asintótica $\mathcal{O}(nm \log(n^2/m))$ y habría que repetirlo $n(n-1)$ veces para considerar todos los pares de fuente-sumidero en la forma más simple.

Sin embargo, se puede demostrar que basta con $(n-1)$ repeticiones, pero en cualquier caso, todo esto significa que para resolver el problema de corte mínimo, tenemos un algoritmo determinista de tiempo $\Omega(n^2 m)$. Con unos trucos más se puede mejorar la situación así que la complejidad de MinCut queda también en $\mathcal{O}(nm \log(n^2/m))$. Para grafos densos, $m \in \mathcal{O}(n^2)$, que hace esa complejidad todavía bastante lento.

En esta sección desarrollamos un algoritmo aleatorizado que resuelve a MinCut más rápidamente, en tiempo $\mathcal{O}\left(n^2 (\log n)^{\mathcal{O}(1)}\right)$.

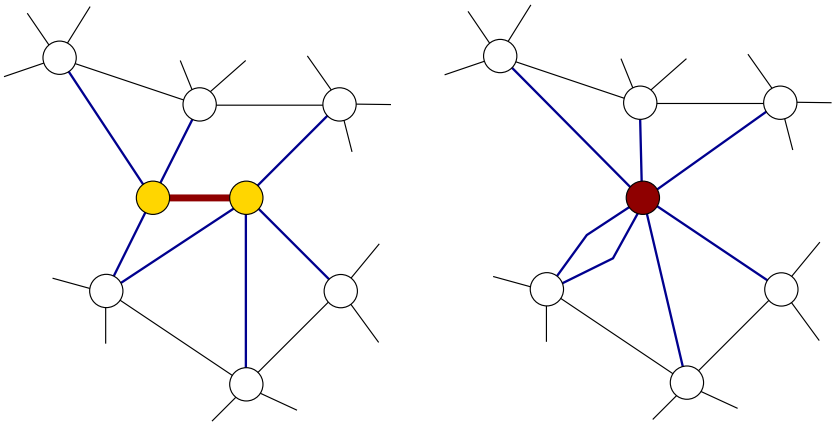

Figura 11.1: La arista gruesa roja en el grafo original (un fragmento mostrado a la izquierda) está contraida así que sus puntos finales (los vértices amarillos) están reemplazados por un sólo vértice (rojo en el frangmento modificado a la derecha) y las aristas entrantes de los vértices eliminados están "redirigidas" al vértice de reemplazo.

La operación "básica" del algoritmo será la *contracción* de una arista. Al contraer la arista $\{u, v\}$ reemplazamos los dos vértices u u v por un vértice nuevo w. La arista contraída desaparece, y para toda arista $\{s, u\}$ tal que $s \notin \{u, v\}$, "movemos" la arista a apuntar a w por reemplazarla por $\{s, w\}$. Igualmente reemplazamos aristas $\{s, u\}$ por aristas $\{s, w\}$ para todo $s \notin \{u, v\}$. La figura 11.1 muestra un ejemplo.

Si contraemos un conjunto $F \subseteq E$, el resultado no depende del orden de contracción. Después de las contracciones, los vértices que quedan representan subgrafos conexos del grafo original. Empezando con el grafo de entraga G, si elegimos iterativamente al azar entre las aristas presentes una para contracción hasta que quedan sólo dos vértices, el número de aristas en el multigrafo final entre esos dos vértices corresponde a un corte de G.

Con una estructura unir-encontrar, podemos fácilmente mantener información sobre los subgrafos representados. Al contraer la arista $\{u, v\}$, el nombre del conjunto combinado en la estructura será w y sus miembros son u u w; originalmente cada vértice tiene su propio conjunto. Entonces, podemos imprimir los conjuntos C y $V \setminus C$ que corresponden a los dos vértices que quedan en la última iteración.

La elección uniforme de una arista para contraer se puede lograr en tiempo lineal $\mathcal{O}(n)$. En cada iteración eliminamos un vértice, por lo cual el algoritmo de contracción tiene complejidad cuadrática en n.

Lo que queda mostrar es que el corte así producido sea el mínimo con una probabilidad no cero. Así por repetir el algoritmo, podríamos aumentar la probabilidad de haber encontrado el corte mínimo. Las siguientes observaciones nos ayudan:

- Si la capacidad del corte mínimo es k, ningún vértice puede tener grado menor a k.
- El número de aristas satisface $m \geq \frac{1}{2}nk$ si la capacidad del corte mínimo es k.
- La capacidad del corte mínimo en G después de la contracción de una arista es mayor o igual a la capacidad del corte mínimo en G.

Fijamos un corte mínimo de $G = (V, E)$ con las aristas $F \subseteq E$ siendo las aristas que cruzan de C a $V \setminus C$. Denotamos $|F| = k$.

Suponemos que estamos en la iteración i del algoritmo de contracción y que ninguna arista en F ha sido contraida todavía. Quedan $n_i = n - i + 1$ aristas en el grafo actual G_i. Por la tercera observación, el conjunto de aristas F todavía define un corte mínimo en el grafo actual G_i y los vértices de los dos lados del corte definido por las aristas en F corresponden a los conjuntos C y $V \setminus C$, aunque (posiblemente) en forma contraida.

El grafo G_i tiene, por la segunda observación, por lo menos $\frac{1}{2}n_i k$ aristas, por lo cual la probabilidad que la siguiente iteración contraiga una de las aristas de F es menor o igual a ${2n_i}^{-1}$. Así podemos acotar la probabilidad p que ninguna arista de F sea contraida durante la ejecución del algoritmo:

$$\begin{aligned} p &\geq \prod_{i=1}^{n-2}(1 - 2(n-i+1)^{-1}) \\ &= \binom{n}{2}^{-1} = \tfrac{2}{n(n-1)}. \end{aligned} \tag{11.7}$$

Hemos establecido que un corte mínimo específido de G corresponde al resultado del algoritmo de contracción con probabilidad $p \in \Omega\left(n^{-2}\right)$. Como cada grafo tiene por lo menos un corte mínimo, la probabilidad de éxito del algoritmo es por lo menos $\Omega\left(n^{-2}\right)$. Si repetimos el algoritmo, digamos $\mathcal{O}\left(n^2 \log n\right)$ veces ya da razón de esperar que el corte de capacidad mínima encontrado sea el corte mínimo del grafo de entreda.

Para hacer que el algoritmo Monte Carlo resultante sea más rápida, hay que aumentar la probabilidad de que un corte mínimo pase por el algoritmo sin ser contraido. Una posibilidad de mejora está basada en la observación que la probabilidad de contraer una arista del corte mínimo crece hacía el final del algoritmo. Si en vez de contraer hasta que queden dos vértices, contraemos primero hasta aproximadamente $n/\sqrt{2}$ vértices y después hacemos dos llamadas recursivas independientes del mismo algoritmo con el grafo G_i que nos queda, un análisis detallado muestra que llegamos a la complejidad $\mathcal{O}\left(n^2(\log n)^{\mathcal{O}(1)}\right)$. La mayoría de los detalles de la demostración están explicados en el libro de Motwani y Raghavan [14].

Capítulo 12

Transiciones de fase

En este capítulo, estudiamos un fenómeno interesante de complejidad en algunos problemas. Al generar instancias del problema aleatoriamente según algún conjunto de parámetros, en algunas ocaciones se ha observado que ciertas combinaciones de los parámetros de generación causan que instancias del mismo tamaño tienen un tiempo de ejecución promedio mucho más largo de lo que uno obtiene con otras combinaciones. Ese "pico" en el tiempo de ejecución con una cierta combinación de parámetros se conoce como una *transición de fase*. Este capítulo está basada en el libro de Hartmann y Weigt [8].

12.1. Modelo de Ising

Un *vidrio de espín* (inglés: spin glass) es un material magnético. Los átomos del material pueden acoplarse aleatoriamente de la manera *ferromagnética* o altenativamente de la manera *antiferromagnética*.

En ferromagnetismo, todos los momentos magnéticos de los partículos de la materia se alignan en la misma dirección y sentido. En falta de presencia de un campo magnético intenso, las alineaciones pueden estár aleatorias, pero al someter el material a tal campo magnético, los átomos gradualmente se alinean. Tal organización permanece vigente por algo de tiempo aún después de haber eliminado el campo magnético que lo causó.

En antiferromagnetismo, los átomos se alinean en la misma dirección pero *sentido inverso* así que un átomo tiene el sentido opuesto al sentido de sus vecinos. En la presencia de un campo magnético muy intenso, se puede causar que se piera algo del antiferromagnetismo y que los átomos se alinean según el campo. Los dos tipos de magnetismo se pierde en temperaturas altas.

Dos átomos con interacción ferromagnática llegan a su estado de energía mínima cuando tienen el mismo sentido, mientras dos átomos con interacción antiferromagnética llegan

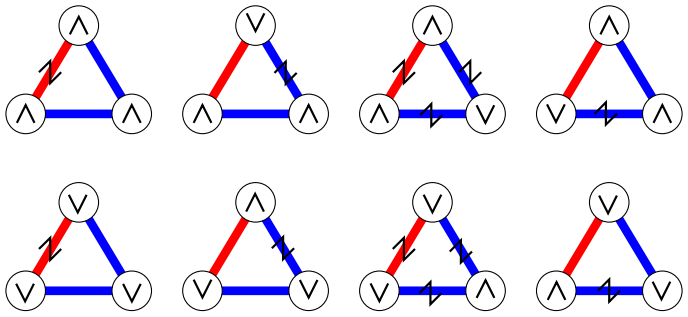

Figura 12.1: Un sistema de tres átomos (representados por los vértices) con dos interacciones ferromagnéticas (aristas azules) y una interacción antiferromagnética (aristas rojas) que no puede alcanzar estado de energía mínima. En cada una de las ocho posibles configuraciones de los sentidos (marcados con $\wedge$ y $\vee$), por lo menos una interacción está en conflicto (marcado con una zeta sobre la arista).

a su estado de energía mínima al tener sentidos opuestos.

El modelo matemático de Ising representa las interacciones como aristas y los átomos como vértices σ_i. Cada vértice puede tener uno de dos valores,

$$\sigma_i = \{-1, 1\} \tag{12.1}$$

que representan los dos posibles sentidos de alineación. Denotamos el sistema por $G = (V, E)$.

Una función J_{ij} da la fuerza de la interacción entre los vértices i y j (cero si no hay interacción) y un valor no cero implica la presencia de una arista en E. En una interacción ferromagnética entre i y j, $J_{ij} > 0$, mientras interacciones antiferromagnéticas tienen $J_{ij} < 0$.

Se usa el término *frustración* a referir a situaciones donde las combinaciones de interacciones que no permiten a todo los partúculos llegar a su estado de energía mínima (ver figura 12.1 para un ejemplo).

La función *hamiltoniana* se define como

$$H(G) = -\sum_{\{i,j\}\in E} J_{ij}\sigma_i\sigma_j. \tag{12.2}$$

Es una medida de energía total del sistema G. Se considera J_{ij} como algo fijo, mientras los valores de σ_i pueden cambiar según la temperatura ambiental. En temperatura $T = 0$, el sistema alcanza su energía mínima. Las configuraciones que tienen energía mínima se llaman *estados fundamentales* (inglés: ground state).

En temperaturas bajas el sistema queda organizado para minimizar "conflictos" de interacciones. Al subir la temperatura del sistema, en un cierto valor crítico T_c de repente se pierde la organización global.

Cuando ambos tipos de interacciones están presentes, hasta determinar el valor mínimo alcanzable para la función hamiltoniana se complica. Típicamente, en los casos estudiados, las aristas forman una reja regular — para dos dimensiones, el problema de encontrar un estado fundamental es polinomial, pero en tres dimensiones, es exponencial. También has estudiado sistemas tipo Ising en grafos aleatorios uniformes.

12.2. Problema del viajante (TSPD)

Generamos instancias para el TSPD, o sea, la variación de decisión del problema de viajante, por colocar n vértices (o sea, ciudades) aleatoriamente en un plano cuadrático de superficie A; denotamos las coordenadas de vértice i por x_i y y_i. Los costos de las aristas serán las distancias euclidianas de los puntos de los vértices:

$$\mathrm{dist}\,(i,j) = \sqrt{(x_1 - x_j)^2 \text{“} (y_i - y_j)^2}. \tag{12.3}$$

La pregunta es si la instancia contiene un ciclo hamiltoniano de costo total menor o igual a un presupuesto D. Entonces, para crear una instancia del problema, hay que elegir los valores n, A y D. La probabilidad de encontrar un ciclo hamiltoniano de largo D depende obviamente de una manera inversa del superficie elegido A y del número de vértices.

Se define un "largo escalado"

$$\ell = \frac{D}{\sqrt{An}} \tag{12.4}$$

y se estudia la probabilidad p que exista en una instancia así generada un ciclo hamiltoniano de largo ℓ con diferentes valores de n y un valor fijo A. Con diferentes valores de n, uno observa que con pequeños valores de ℓ (menores a $0{,}5$), $p \approx 0$, mientras para valores grandes de ℓ (mayores a uno), $p \approx 1$. Las curvas de p versus ℓ suben rápidamente desde cero hasta uno cerca del valor $\ell = 0{,}78$ [18], independientemente del número de vértices.

Al estudiar el tiempo de ejecución de un cierto algoritmo tipo ramificar-acotar para TSPD, los tiempos mayores ocurren con instancias que tienen valores de ℓ cerca de $0{,}78$.

12.3. Grafos aleatorios uniformes

En esta sección estudiamos comportamiento de grafos aleatorios, un ejemplo clásico de transiciones de fase de cierto tipo. Seguimos por mayor parte el libro de texto de Bollobas [3], aunque no lo alcanzamos en el nivel formalidad.

12.3.1. Familias y propiedades

Una *familia* $\mathcal{G}$ de grafos incluye todos los grafos que cumplen con la definición de la familia. La definición está típicamente compuesta por parámetros que controlen el tamaño de la familia. Por ejemplo, todos los grafos con k vértices forman una familia y todos los grafos k-regulares otra familia (y su intersección no es vacía).

Una *propiedad* de grafos de la familia $\mathcal{G}$ es un subconjunto *cerrado* $\mathcal{P} \subseteq \mathcal{G}$. Cuando $grafo \in \mathcal{P}$, se dice que "el grafo G tiene la propiedad $\mathcal{P}$". Lo que significa ser cerrado es que al tener dos grafos $G \in \mathcal{P}$ y $G' \in \mathcal{G}$, si aplica que G y G' sea isomorfos, necesariamente $G' \in \mathcal{P}$ también.

Una propiedad $\mathcal{P}$ es *monotona* si de $G \in \mathcal{P}$ y G es subgrafo de G', implica que $G' \in \mathcal{P}$. Un ejemplo de tal propiedad sería "el grafo contiene un triángulo" o "el grafo contiene un camino de largo k" — por añadir aristas o vértices al grafo, no es posible hacer desaparecer estas propiedades.

Una propiedad es *convexa* si el hecho que G' es subgrafo de G y G'' es subgrafo de G' tales que $G'', G \in \mathcal{P}$, implica que también $G' \in \mathcal{P}$.

Decimos que "casi cada grafo" $G = (V, E)$ de la familia $\mathcal{G}_n$ la definición de cual depende de $n = |V|$ tiene $\mathcal{P}$ si para $G \in \mathcal{G}$

$$\lim_{n\to\infty} \Pr\left[G \in \mathcal{P}\right] = 1. \tag{12.5}$$

12.3.2. Modelos de generación

Para generar un grafo aleatorio uniforme (simple, no dirigido), hay dos modelos básicos: en el modelo de Gilbert, se fija n y una probabilidad p. Para generar un grafo $G = (V, E)$, asignamos $V = \{1, 2, \ldots, n\}$ y generamos E de la manera siguiente: considera a la vez cada uno de los $\binom{n}{2}$ pares de vértices distintos u y v e incluye la arista $\{u, v\}$ independientemente al azar con probabilidad p. Con este modelo, conodido como el modelo $G_{n,p}$, tenemos

$$\mathrm{E}\left[m\right] = p\binom{n}{2} \tag{12.6}$$

$$\mathrm{E}\left[\deg\left(u\right)\right] = p(n-1). \tag{12.7}$$

La otra opción fue estudiada en detalle por Erdős y Rényi [5, 6]: fijar los dos n y m y elegir uniformemente al azar uno de los posibles conjuntos de m aristas. El número de posibles maneras de elegir las aristas entre vértices distintos es

$$\binom{\binom{n}{2}}{m}, \tag{12.8}$$

aunque en realidad los automorfismos reducen el número de grafos distintos que se pueda generar. El segundo modelo se conoce como $G_{n,m}$.

Una tercera opción para generar grafos uniformes dado un conjunto de n vértices $V = \{1, 2, \ldots, n\}$, es definir un *proceso* (estocástico) como una sucesión $(G_t)_0^N$ donde t representa el *tiempo* discreto que toma valores $t = 0, 1, \ldots, N$ tal que

(I) cada G_t es un grafo en el conjunto V, es decir $G_t = (V, E_t)$ — lo que cambia durante el proceso es el conjunto de las aristas;

(II) G_t tiene t aristas para cada valor de t, o sea, $|E_t| = t$;

(III) G_{t-1} es un subgrafo de G_t, empezando de G_0 y terminando en G_N: $E_0 \subset E_1 \subset \ldots E_N$; y

(IV) $N = \binom{n}{2}$.

Podemos definir un mapeo entre el espacio de todos los procesos posibles en el momento $t = m$ y los grafos $G_{n,m}$, por lo cual los grafos $G_{n,m}$ son todos posibles resultados intermedios de todos los procesos posibles que cumplan con la definición.

12.3.3. Comportamiento repentino

Suponemos que $\mathcal{P}$ sea una propiedad monotona de grafos. Observamos el proceso de generación $(G_t)_0^N$ y checamos en cada instante t si o no G_t tiene la propiedad $\mathcal{P}$. Llamamos el momento τ cuando $G_t \in \mathcal{P}$ el *tiempo de llegada* (inglés: hitting time) del proceso a la propiedad,

$$\tau = \tau(n) = \min_{t \geq 0} \{G_t \in \mathcal{P}\}. \tag{12.9}$$

Nota que por ser monotona $\mathcal{P}$, $G_t \in \mathcal{P}$ para todo $t \geq \tau$ y que τ puede variar entre diferentes instancias del proceso con un valor n fijo.

Podemos definir una *función umbral* para la propiedad $\mathcal{P}$ en términos del valor de n utilizado al definir el proceso: $U(n)$ es la función umbral de $\mathcal{P}$ si aplica que

$$\Pr\left[U(n)f(n)^{-1} < \tau(n) < U(n)f(n)\right] \to 1 \tag{12.10}$$

con cualquier función $f(n) \to \infty$. En términos vagos, la función umbral es el "tiempo crítico" antes de que es poco probable que G_t tenga $\mathcal{P}$ y después de que es muy probable que la tenga. Se ha mostrado, ya desde el trabajo de Erős y Rényi, que varias propiedades monotonas aparecen muy de repente en tales procesos: casi ningún G_t los tiene antes de un cierto momento y casi todos G_t lo tienen después de ese momento, sobre el conjunto de todos los $N!$ procesos posibles.

12.3.4. Aparición de un componente gigante

Continuamos con los procesos $(G_t)_0^N$ y estudiamos el tamaño de los componentes conexos de G_t. Por definición, en G_0 cada vértice está aislado y G_N es el grafo completo de n vertices, K_n. Definimos $\mathcal{P}$ como la propiedad que el grafo sea conexo. En algún momento,

$$\tau \in \left[n-1, \binom{n-1}{2} + 1 \right], \tag{12.11}$$

el grafo llega a tener la propiedad $\mathcal{P}$ por la primera vez. La cota inferior viene del caso "mejor": las primeras $n-1$ aristas forman un árbol de expansión. El cota superior es el peor caso: uno de los n vértices permanece aislado así que todas las aristas añadidas caen entre los demás hasta que ellos ya formen una camarilla de $n-1$ vértices, después de que ya necesariamente se conecta el grafo al añadir la arista siguiente.

En vez de estudiar exactamente el momento cuando todo el grafo ya sea conexo, resumimos algunos resultados sobre la cantidad de aristas necesarias para que el grafo tenga un componente conexo grande. Fijamos un valor m con la ayuda de un parámetro constante c tal que

$$m = \lfloor cn \rfloor, \tag{12.12}$$

donde $c \in [0, \frac{(n-1)]}{2}$. El resultado interesante sobre los tamaños de los componentes es que con $c < \frac{1}{2}$, el tamaño del mayor componente conexo en un G_t cuando $t = m$ es de orden logarítmico en n, pero para $c > \frac{1}{2}$ — que es todavía un valor muy pequeño — un G_t cuando $t = m$ tiene un componente conexo de aproximadamente ϵn vértices donde $\epsilon > 0$ únicamente depende del valor de c y el resto de los vértices pertecen a componentes mucho menores.

Interpretando el valor de c llegamos a notar que está relacionado con el grado promedio de los vértices:

$$\frac{\sum_{v \in V} \deg(v)}{n} = \frac{2m}{n} = \frac{2\lfloor cn \rfloor}{n} \approx 2c. \tag{12.13}$$

Entonces, cuando el grado promedio del grafo llega a aproximadamente uno (o sea $c \approx \frac{1}{2}$), aparece un componente gigante en el G_t.

12.4. Cubierta de vértices (Vertex Cover)

También en el problema de cubierta de vértices se encuentra una transición de fase con un algoritmo ramificar-acotar, usando como el modelo de generación el $G_{n,p}$ tal que $p = c/n$. De la ecuación 12.7, tenemos que el grado promedio k de un vértice en tal grafo es

$$k = p(n-1) = c\frac{n-1}{n} \approx c, \tag{12.14}$$

y podemos hablar del parámetro c como en grado promedio (para grandes valores de n). Estudiando la probabilidad q de la existencia de una cubierta con una cierta candidad x de vértices cuando se fija c, de nuevo se encuentra que para valores bajos de x, q es bajo, mientras para valores grandes de x, q es alto, y la subida ocurre de una manera repentina en un cierto valor de x sin importar el valor de n.

Bibliografía

[1] Emile Aarts and Jan Karel Lenstra. *Local Search in Combinatorial Optimization*. John Wiley & Sons, Inc., Chichester, UK, 1997.

[2] G. M. Adel'son-Vel'skii and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1263, 1962.

[3] Béla Bollobás. *Random Graphs*. Number 73 in Cambridge Studies in Advanced Mathematics. Cambridge University Press, Cambridge, UK, second edition, 2001.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. McGraw-Hill Book Co., Boston, MA, USA, second edition, 2001.

[5] Pál Erdős and Alfréd Rényi. On random graphs i. In *Selected Papers of Alfréd Rényi*, volume 2, pages 308–315. Akadémiai Kiadó, Budapest, Hungary, 1976. First publication in Publ. Math. Debrecen 1959.

[6] Pál Erdős and Alfréd Rényi. On the evolution of random graphs. In *Selected Papers of Alfréd Rényi*, volume 2, pages 482–525. Akadémiai Kiadó, Budapest, Hungary, 1976. First publication in MTA Mat. Kut. Int. Közl. 1960.

[7] Michael R. Garey and David S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman, San Francisco, CA, USA, 1979.

[8] Alexander K. Hartmann and Martin Weigt. *Phase Transitions in Combinatorial Optimization Problems : Basics, Algorithms and Statistical Mechanics*. Wiley-VCH, Weinheim, Germany, 2005.

[9] Holger H. Hoos and Thomas Stützle. *Stochastic Local Search: Foundations and Applications*. Morgan Kaufmann, San Francisco, CA, USA, 2005.

[10] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Addison-Wesley / Pearson Education, Inc., Boston, MA, USA, 2006.

[11] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, USA, second edition, 1998.

[12] Janet Susan Milton and Jesse C. Arnold. *Introduction to Probability and Statistics : Principles and Applications for Engineering and the Computing Sciences*. McGraw-Hill, Singapore, third edition, 1995.

[13] Michael Mitzenmacher and Eli Upfal. *Probability and Computing : Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, New York, NY, USA, 2005.

[14] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, Cambridge, England, UK, 1995.

[15] Christos H. Papadimitriou. *Computational Complexity*. Addison Wesley, Reading, MA, USA, 1994.

[16] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization : Algoritms and Complexity*. Dover Publications, Inc., Mineola, NY, USA, 1998.

[17] Vijay V. Vazirani. *Approximation Algorithms*. Springer-Verlag GmbH, Berlin, Germany, 2001.

[18] René V. V. Vidal, editor. *Applied Simulated Annealing*. Lecture Notes in Economics & Mathematical Systems. Springer-Verlag GmbH, Berlin / Heidelberg, Germany, 1993.

Listas de smbolos

Elementos y ordenamiento

$a, b, \ldots$	elementos
$a = b$	el elemento a es igual al elemento b
$a \neq b$	el elemento a no es igual al elemento b
$a \approx b$	el elemento a es *approximadamente* igual al elemento b
$a < b$	el elemento a es menor al elemento b
$a > b$	el elemento a es mayor al elemento b
$a \leq b$	el elemento a es menor o igual al elemento b
$a \geq b$	el elemento a es mayor o igual al elemento b
$a \ll b$	el elemento a es *mucho* menor al elemento b
$a \gg b$	el elemento a es *mucho* menor al elemento b

Conjuntos y conteo

$A, B, \ldots$	conjuntos
$\mathbb{R}$	el conjunto de los números reales
$\mathbb{Z}$	el conjunto de los números enteros
$\emptyset$	el conjunto vació
$\{a_1, a_2, \ldots, a_n\}$	un conjunto de n elementos
$a \in A$	el elemento a pertecene al conjunto A
$a \notin A$	el elemento a no pertenece al conjunto A
$\lvert A \rvert$	la cardinalidad del conjunto A
$\{a \mid a \in A\}$	el conjunto de los elementos a que cumplen con la propiedad que $a \in A$
$A = B$	los conjuntos A y B contienen los mismos elementos
$A \neq B$	los conjuntos A y B no contienen todos los mismos elementos
$A \subseteq B$	el conjunto A es un subconjunto de B o igual a B
$A \subset B$	el conjunto A es un subconjunto propio de B
$A \nsubseteq B$	el conjunto A no es un subconjunto de B
$A \cup B$	la unión de los conjuntos A y B
$A \cap B$	la intersecćon de los conjuntos A y B
$\bigcup_{i=1}^{n} A_i$	la unión de los n conjuntos $A_1, A_2, \ldots, A_n$
$\bigcap_{i=1}^{n} A_i$	la intersección de los n conjuntos $A_1, A_2, \ldots, A_n$
$\bar{A}$	el conjunto que es complemento del conjunto A
$A \setminus B$	la diferencia de los conjuntos A y B
(a, b)	un par ordenado de elementos
$A \times B$	el producto cartesiano de los conjuntos A y B
$k!$	el factorial de k
$\binom{n}{k}$	el coeficiente binomial

Mapeos y funciones

$g : A \to B$	un mapeo del conjunto A al conjunto B, donde A es el dominio y B el rango
$g(a)$	la imagen del elemento $a \in A$ en el rango bajo el mapeo g
$g^{-1}(b)$	la imagen inversa del elemento $b \in B$ en el dominio bajo el mapeo g
$f(a)$	el valor asignado al elemento a por la funcción f
$\|x\|$	el valor absoluto de $x \in \mathbb{R}$
$\log_b(x)$	el logaritmo a base b de $x \in \mathbb{R}$, $x > 0$
$f(n) = \mathcal{O}(g(n))$	$\|f(n)\| \leq c\|g(n)\|$ para algún constante c y valores suficientemente grandes de n
$f(n) = \Omega(g(n))$	$\|f(n)\| \geq \|g(n)\|$ para algún constante c y valores suficientemente grandes de n
$f(n) = \Theta(g(n))$	$c'\|g(n)\| \leq \|f(n)\| \leq c\|g(n)\|$ para algunos constantes c, c' y valores suficientemente grandes de n

Lógica matemática

a	una proposición lógica
x_i	una variable booleana
$\top$	verdadero
$\bot$	falso
T	una asignación de valores
ϕ	una expresión booleana
$T \models \phi$	T satisface ϕ
$\models \phi$	ϕ es una tautologa
$\neg a$	la negación: no a
$a \vee b$	la disyunción: a o b
$a \wedge b$	la conjunción: a y b
$a \rightarrow b$	la implicación: si a, entonces b
$a \leftrightarrow b$	la equivalencia: a si y sólo si b
$\forall a \in A$	cuentificación universal: para todo $a \in A$
$\exists a \in A$	cuantificación existencial: existe un $a \in A$

Grafos

V	un conjunto de vértices
E	un conjunto de aristas
n	el orden de un grafo
m	el número de aristas de un grafo
G	un grafo $G = (V, E)$
$v, w, u, \ldots$	vértices
$\{v, w\}$	una arista no dirigida entre los vértices v y w
$\langle v, w \rangle$	una arista dirigida del vértice v al vértice w
$\omega(v)\, w$	el peso de la arista $\{v, w\}$ o $\langle v, w \rangle$
$\deg(v)$	el grado del vértice v
$\overleftarrow{\deg}(v)$	el grado de entrada del vértice v
$\overrightarrow{\deg}(v)$	el grado de salida del vértice v
$\mathrm{dist}(v, w)$	la distancia del vértice v al vértice w
$\mathrm{diam}(G)$	el diámetro del grafo G
$\delta(G)$	la densidad del grafo G
K_k	el grafo completo de orden k
$K_{k,\ell}$	el grafo bipartitio completo de conjuntos de vértices de cardinalidades k y ℓ

Probabilidad

$X, Y, Z, \ldots$	variables aleatorias
$x, y, z, \ldots$	valores posibles de variables aleatorias
$\Pr[X = x]$	la probabilidad que X asume el valor x
$\Pr[X = x \mid Y = y]$	la probabilidad condicional que $X = x$ dado que $Y = y$
$\mathrm{E}[X]$	el valor esperado de X
$\mathrm{Var}[X]$	la varianza de X

Lista de abreviaciones

	Inglés	Español
BFS	breadth-first search	búsqueda en anchura
CNF	conjuntive normal form	forma normal conjuntiva
DFS	depth-first search	búsqueda en profundidad
DNF	disjuntive normal form	forma normal disyunctiva
NTM	nondeterministic Turing machine	máquina Turing no determinista
PTAS	polynomial-time approximation scheme	esquema de aproximación de tiempo polinomial
RAM	random access machine	máquina de acceso aleatorio
TSP	travelling salesman problem	problema del viajante

Diccionario terminológico

Español-inglés

acomplamiento	matching
aleatorio	random
algoritmo	algorithm
análisis asintótica	asymptotic analysis
árbol	tree
árbol binario	binary tree
árbol cubriente	spanning tree
árbol de expansión	spanning tree
arista	edge
arreglo	array
barrer	sweep
biselado	splay
bósque	forest
bucle	loop
burbuja	bubble
búsqueda	search
cadena	string

capa	layer
camarilla	clique
camino	path
ciclo	cycle
clausura	closure
clave	key
cola	queue
cota	bound
complejidad	complexity
conexo	connected
conjunto	set
cubierta	cover
dirigido	directed
estructura de datos	data structure
enlazada	linked
flujo	flow
fuerza bruta	brute-force
grafo	graph
hashing	dispersión
hoja	leaf
hijo	child
intratable	intractable
lista	list
montículo	heap

no dirigido	undirected
orden posterior	postorder
orden previo	preorder
ordenamiento	sorting
padre	parent
patrón	pattern
peor caso	worst case
pila	stack
podar	prune
ponderado	weighted
succesión	series
raíz	root
ramificar-acotar	branch and bound
ramo	branch
recorrido	traversal
red	network
rojo-negro	red-black
vértice	vertex
voraz	greedy
vuelta atrás	backtracking
unir-encontrar	union-find

Inglés-español

algorithm	algoritmo
array	arreglo
asymptotic analysis	análisis asintótica
backtracking	vuelta atrás
binary tree	árbol binario
bound	cota
branch	ramo
branch and bound	ramificar-acotar
brute-force	fuerza bruta
bubble	burbuja
child	hijo
clique	camarilla
closure	clausura
complexity	complejidad
connected	conexo
cover	cubierta
cycle	ciclo
data structure	estructura de datos
directed	dirigido
dispersión	hashing
edge	arista
flow	flujo
forest	bósque

graph	grafo
greedy	voraz
heap	montculo
intractable	intratable
key	clave
layer	capa
leaf	hoja
linked	enlazada
list	lista
loop	bucle
matching	acomplamiento
network	red
parent	padre
path	camino
pattern	patrón
postorder	orden posterior
preorder	orden previo
prune	podar
queue	cola
random	aleatorio
red-black	rojo-negro
root	raz
search	búsqueda
series	succesión

set	conjunto
sorting	ordenamiento
spanning tree	árbol cubriente
spanning tree	árbol de expansión
splay	biselado
stack	pila
string	cadena
sweep	barrer
traversal	recorrido
tree	árbol
undirected	no dirigido
union-find	unir-encontrar
vertex	vértice
weighted	ponderado
worst case	peor caso

Índice de figuras

Índice de cuadros