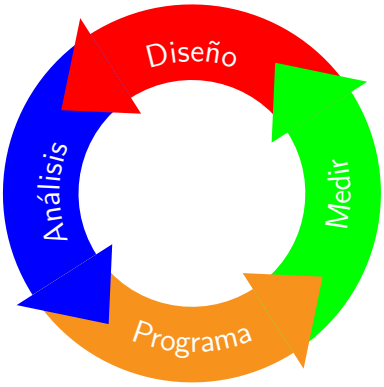
Algoritmos y Complejidad

Algoritmos y Complejidad
INF-221

Horst H. von Brand



7 de marzo de 2017

Departamento de Informática
Universidad Técnica Federico Santa María

Compuesto por el autor en LaTeX 2$_\varepsilon$ con Utopia para textos y Fourier-GUTenberg para matemáticas.

Versión 0.1-75-g74c0a0c del 2017-03-01

Agradezco a mi familia, a quienes he descuidado demasiado durante el desarrollo del presente texto.
Agradezco a mis estudiantes, quienes sufrieron versiones preliminares del presente texto.

El Departamento de Informática de la Universidad Técnica Federico Santa María provee el ambiente ideal de trabajo.

Este documento presenta la materia del curso "Algoritmos y Complejidad" (INF-221) como dictado en la Casa Central de Universidad Técnica Federico Santa María por el autor. Son más que nada resultados del esfuerzo de Aldo Berríos, quien se dio el trabajo de transcribir las (a veces muy desorganizadas) clases a LaTeX, ligeramente editadas por el autor, incorporando comentarios y correcciones de estudiantes, y llevadas al formato actual.

Índice general

Índice de figuras

Índice de cuadros

Índice de listados

Parte I

Análisis Numérico

Clase 1

Análisis numérico

El *análisis numérico* trata de métodos computacionales para obtener valores numéricos precisos para una variedad de objetos, como funciones, integrales, soluciones de ecuaciones y sistemas de éstas. Una triste verdad es que la inmensa mayoría de las situaciones de interés práctico llevan a ecuaciones que no tienen solución "exacta", hay que recurrir a técnicas aproximadas. Incluso puede darse la situación que es más eficiente calcular una solución aproximada que usar una engorrosa fórmula exacta. Esta es un área enorme, en este curso cubriremos solo algunas áreas específicas de interés más bien general. Un texto general es el de Gautschi [2], y hay numerosas colecciones de notas de clase disponibles, como la de Olver [6], de Cowley [1] o de Philip [7, 8].

Para obtener valores numéricos de interés en una situación práctica se debe construir un modelo de la situación, extraer de él las relaciones relevantes, y resolver las ecuaciones resultantes para obtener el resultado. Hay varias fuentes de error en esto:

(I) El modelo físico es una simplificación de la realidad (se omite la fricción del aire, suponer que la aceleración de la gravedad es constante, ...)

(II) Los parámetros que entran al modelo se conocen con precisión finita

(III) Aproximaciones usadas para resolver el modelo matemático

(IV) Errores de redondeo en los cálculos

El punto (I) es tema de modelamiento matemático, no nos concierne acá. De (II) se preocupa quien monta el experimento. El punto (IV) tiene que ver con la representación de infinitos números reales en espacio finito, cosa que discuten en detalle Goldberg [3] y Haberman [4, 5]. El punto (III) el tema central de interés acá.

Hay dos formas principales de cuantificar el error:

Definición 1.1. Sea x un valor real, y x^* una aproximación. El *error absoluto* de la aproximación $x^* \approx x$ es $|x^* - x|$, el *error relativo* (siempre que $x \neq 0$) es $|x^* - x|/|x|$.

Por ejemplo, 1000 es una aproximación de 1024 con error absoluto de 24 y error relativo de 0,023. Diremos que la aproximación *tiene k dígitos decimales significativos* si el error relativo es menor que 10^{-k-1}, o sea, después del primer dígito decimal no cero hay k dígitos correctos.

1.1. Propagación de errores

Considere calcular el valor $y = f(x)$. Solo podemos calcular una aproximación y^*, hay dos maneras de cuantificar el error asociado a esta aproximación.

1.1.1. Hacia adelante

En inglés, se habla de *forward error*. Es una medida de la diferencia entre la aproximación y^* y el valor correcto y, ya sea absoluto o relativo. Como no conocemos y, solo podemos obtener una cota superior. Suele ser difícil obtener cotas ajustadas.

Una técnica es aproximar la función por la serie de Taylor centrada en x, o sea:

$$|y^* - y| \approx |f'(x)| \cdot |\Delta x|$$

En caso que hayan más datos de entrada, se usa la serie de Taylor en múltiples variables, conservando los términos lineales únicamente.

1.1.2. Hacia atrás

En inglés, *backward error*. Acá la pregunta es, conozco y^*, que es respuesta a algún problema $f(x^*)$. Específicamente, nos interesa el mínimo Δx tal que:

$$y^* = f(x + \Delta x)$$

A tal $|\Delta x|$ (o $|\Delta x|/|x|$) se le llama el error hacia atrás. Suele ser más fácil de calcular, y obtener cotas ajustadas.

1.2. Estabilidad y condicionamiento

Hay que tener presente que los cálculos en punto flotante *no* cumplen las conocidas leyes. Por ejemplo, exactamente:

$$\begin{aligned}\frac{301}{4000} - \frac{300}{4001} &= \frac{301 \cdot 4001 - 300 \cdot 4000}{4000 \cdot 4001} \\ &= \frac{4301}{16004000} \\ &\approx 0{,}0002687453\end{aligned}$$

Si hacemos los cálculos con 3 cifras, obtenemos:

$$\begin{aligned}\frac{301}{4000} &= 0{,}0753 \\ \frac{300}{4001} &= 0{,}0750 \\ \frac{301}{4000} - \frac{300}{4001} &= 0{,}0003\end{aligned}$$

El resultado no tiene ni una sola cifra correcta. Escribiendo:

$$\begin{aligned}\frac{301}{4000} - \frac{300}{4001} &= \frac{301 \cdot 4001 - 300 \cdot 4000}{4000 \cdot 4001} \\ &= \frac{1{,}20 \cdot 10^6 - 1{,}20 \cdot 10^6}{1{,}60 \cdot 10^7} \\ &= 0\end{aligned}$$

Esto claramente es incorrecto.

Consideremos el problema de valor inicial:

$$u' - 2u = -e^{-t} \quad u(0) = \frac{1}{3}$$

La solución es un simple ejercicio:

$$u(t) = \frac{1}{3}e^{-t}$$

Esto decae exponencialmente cuando $t \to \infty$.

En un computador, con precisión finita, no podemos representar 1/3 en forma exacta, en el mejor caso estamos resolviendo algo como:

$$v' - 2v = -e^{-t} \quad v(0) = \frac{1}{3} + \epsilon$$

donde ϵ es el error en la representación de 1/3. Su solución es:

$$v(t) = \frac{1}{3}e^{-t} + \epsilon e^{2t}$$

Esta solución crece exponencialmente, el error cometido solo en el valor inicial pronto abruma la verdadera solución.

Wilkinson [9, 10] al probar un conjunto de rutinas de punto flotante para un computador temprano por casualidad se tropezó con el fenómeno que discutiremos. Da un resumen divertido en [11].

Consideremos el polinomio

$$p(x) = (x-1)(x-2)\cdots(x-10)$$

Conocemos sus ceros en forma exacta. Cabe esperar que los ceros del polinomio:

$$q(x) = p(x) + x^5$$

sean cercanos, la diferencia está en los términos $-902055x^5$ en p y $-902054x^5$ en q, una diferencia de 0,0001 % en un coeficiente. Pero los ceros de $q(x)$ son:

$$1{,}0000027558, \quad 1{,}99921, \quad 3{,}02591, \quad 3{,}82275,$$
$$5{,}24676 \pm 0{,}751485\text{i}, \quad 7{,}57271 \pm 1{,}11728\text{i}, \quad 9{,}75659 \pm 0{,}368389\text{i}$$

El menor cero es cercano, los demás se alejan crecientemente y los últimos seis mutan a complejos conjugados.

Vale decir, hay problemas cuyas soluciones son muy sensibles a los datos de entrada, se les llama *mal condicionados* (en inglés *ill-conditioned*). Podemos considerar un problema como una función $f(x)$ de un dato de entrada, suele cuantificarse la condición del problema $y = f(x)$ mediante el *número de condición*, el error relativo de y dividido por el de x:

$$\frac{\frac{|\Delta y|}{|y|}}{\frac{|\Delta x|}{|x|}} = \frac{|\Delta y|}{|\Delta x|} \cdot \left|\frac{x}{y}\right|$$
$$\approx \left|\frac{x f'(x)}{f(x)}\right|$$

Si el número de condición es alto, el problema es mal condicionado. Definiciones similares se aplican cuando hay más datos de entrada.

Otro fenómeno se produce cuando errores intermedios del algoritmo se amplifican, posiblemente abrumando el resultado. Esta situación se llama *inestabilidad*. Un ejemplo es calcular la integral:

$$I_n = \int_0^1 x^n \mathrm{e}^{x-1}\,\mathrm{d}x$$

Por integración por partes obtenemos la recurrencia:

$$I_n = 1 - nI_{n-1} \quad I_0 = 1 - \mathrm{e}^{-1}$$

Esta es una forma exacta y eficiente de calcular I_n, sin embargo si se efectúa con 6 cifras el valor calculado de I_9 es negativo.

Nótese que estas dos situaciones son diferentes, el condicionamiento es inherente al problema, la estabilidad es una característica del algoritmo. Obtener una solución a un problema mal condicionado será difícil, incluso con un algoritmo estable.

Ejercicios

1. En la fórmula tradicional para los ceros de la cuadrática:

   $$ax^2 + bx + c$$

   $$x_1, x_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

   si b^2 es mucho mayor que $4ac$, en uno de los casos se restan términos muy parecidos, y el resultado es muy poco preciso. Proponga una técnica para obtener valores precisos de ambos ceros usando la fórmula de Vieta, $x_1 x_2 = c/a$. Considere además los diferentes casos especiales que se producen, si $a \approx 0$ o $c \approx 0$. Prográmela en Python.

2. Calcule el valor de $\mathrm{e}^{-5,3}$, usando 4 cifras significativas en los valores intermedios:

   a) Usando directamente la serie de Maclaurin

   b) Mediante la identidad:

   $$\mathrm{e}^{-5,3} = \frac{1}{\mathrm{e}^{5,3}}$$

   y calculando la exponencial mediante la serie

   Compare con el valor exacto 0,00499159390691021621.

3. Complete el ejemplo de algoritmo inestable efectuando los cálculos indicados. Analice la estabilidad de la iteración.

4. Analice la iteración:

   $$x_{n+2} = ax_{n+1} + bx_n$$

   desde el punto de vista de estabilidad.

Bibliografía

[1] Stephen J. Cowley: *Mathematical Tripos: IB Numerical Analysis*. http://www.damtp.cam.ac.uk/user/sjc1/teaching/NAIB/notes.pdf, Lent 2014.

[2] Walter Gautschi: *Numerical Analysis*. Birkhäuser, second edition, 2012.

[3] David Goldberg: *What every computer scientist should know about floating-point arithmetic*. ACM Computing Surveys, 23(1):5–48, March 1991.

[4] Josh Haberman: *Floating point demystified, part 1*. http://blog.reverberate.org/2014/09/what-every-computer-progr September 2014.

[5] Josh Haberman: *Floating point demystified, part 2*. http://blog.reverberate.org/2016/02/06/floating-point-demysti February 2016.

[6] Peter J. Olver: *Lecture notes on numerical analysis*. http://math.umn.edu/~olver/num.html, May 2008.

[7] Peter Philip: *Numerical analysis I*. http://www.mathematik.uni-muenchen.de/~philip/publications/lectureNotes/nu December 2016.

[8] Peter Philip: *Numerical analysis II*. http://www.mathematik.uni-muenchen.de/~philip/publications/lectureNotes/n December 2016.

[9] James H. Wilkinson: *The evaluation of the zeros of ill-conditioned polynomials. part I*. Numerische Mathematik, 1(1):150–166, December 1959.

[10] James H. Wilkinson: *The evaluation of the zeros of ill-conditioned polynomials. part II*. Numerische Mathematik, 1(1):167–180, December 1959.

[11] James H. Wilkinson: *The perfidious polynomial*. In Gene H. Golub (editor): *Studies in Numerical Analysis*, pages 3–28. Mathematical Association of America, 1984.

Clase 2

Encontrar ceros de funciones

Idea: Dada $f(x)$, hallar x^* tal que $f(x^*) = 0$ (f debe ser continua). Por ejemplo: ¿para qué valor de x se cumple $x = e^{-x}$? Una idea para resolver este problema es simplemente graficar la función.

$$f(x) = x - e^{-x} \tag{2.1}$$

y ver cuáles son los valores de x^* tal que $f(x^*) = 0$.

Otra forma de solucionar el problema puede ser graficar x, e^{-x}, para buscar las intersecciones (figura 2.1).

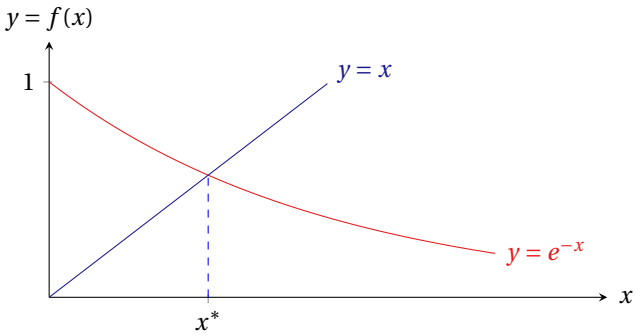


Figura 2.1 – Apreciamos que $0 < x^* < 1$.

2.1. Métodos Bracketing

Corresponden a métodos para encontrar ceros de funciones los cuales usan el *teorema del valor intermedio* y que básicamente van encerrando la solución hasta encontrar un punto de convergencia. A continuación, se explican dos métodos de horquillado (en inglés, *bracketing*): el método de la bisección y *regula falsi*.

2.1.1. Método de la Bisección

Si tenemos x_0, x_1 tales que $f(x_0) \cdot f(x_1) < 0$, hay un cero de f en $[x_0, x_1]$, elegimos

$$x_2 = \frac{x_0 + x_1}{2} \tag{2.2}$$

con $[x_0, x_2]$ o $[x_2, x_1]$ según el cual tenga valores de f de signo distinto. Por ejemplo, consideremos

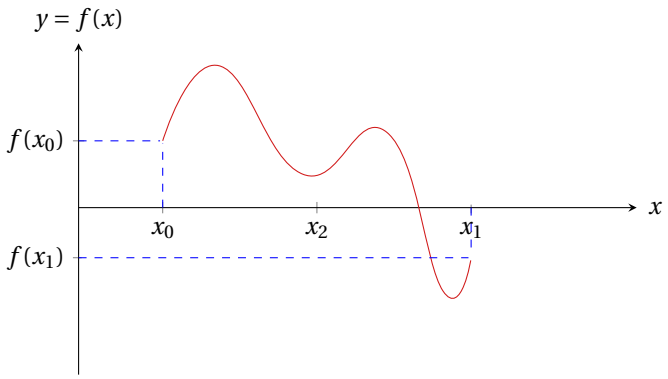


Figura 2.2 – Si $f(x_0) \cdot f(x_1) < 0$, hay $x^* \in [x_0, x_1]$ donde $f(x^*) = 0$.

la figura 2.2. En ella, podemos apreciar que el intervalo $[x_2, x_1]$ cumple con $f(x_2) \cdot f(x_1) < 0$. En consecuencia, sabemos que $x^* \in [x_2, x_1]$. Luego, para aproximar x^* obtenemos el valor medio del intervalo $[x_2, x_1]$ y repetimos el proceso.

Nótese que la gracia de todo esto es encontrar sólo *un* cero, ¡no todos! Esto quiere decir lo ideal es escoger intervalo $[a, b]$ tal que sólo tenga *un* cero. Si nuestra función sólo toca el eje X (vale decir, tiene un cero de multiplicidad par) esto claramente no sirve. Por lo tanto, el método de la bisección

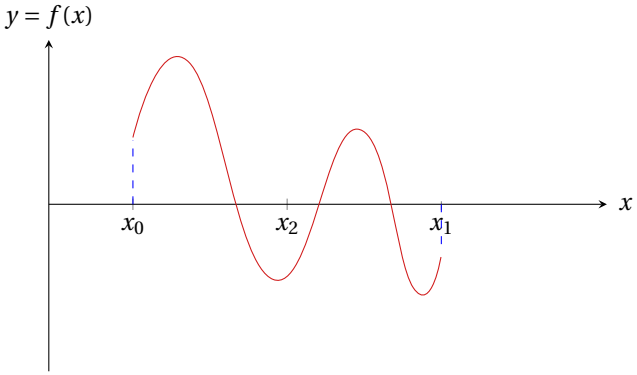


Figura 2.3 – La presente función tiene 3 ceros.

no abarca todos los casos posibles.

2.1.2. Método *Regula Falsi*

Cuando tenemos la sospecha o simplemente nos dicen que nuestra curva f tiene una especie de "guatita" o simplemente tiene tendencia lineal (figura 2.4), podemos usar:

$$x_2 = x_1 - f(x_1) \cdot \frac{x_1 - x_0}{f(x_1) - f(x_0)} \tag{2.3}$$

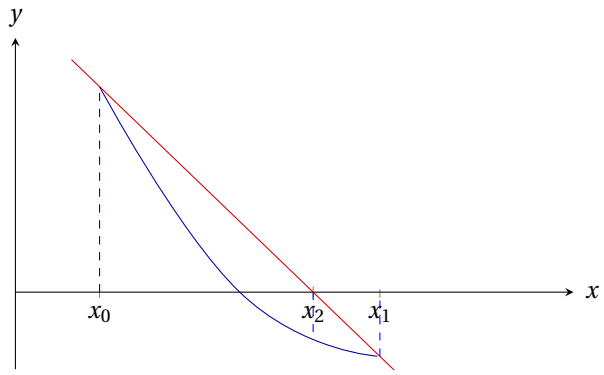


Figura 2.4 – *Regula falsi*

2.2. Iteración de Punto Fijo (FPI)

Definición 2.1. Sea $g(x)$ una función. Un *punto fijo* de g es x^* tal que $x^* = g(x^*)$.

Encontrar un cero de una función por algún método de punto fijo consiste en reescribir:

$$f(x) = 0 \tag{2.4}$$

luego, despejamos x de (2.4) y lo que quede al lado opuesto de la ecuación será una función $g(x)$ tal que:

$$g(x) = x \tag{2.5}$$

Nótese que siempre podemos escribir:

$$g(x) = x - \alpha f(x)$$

Queda elegir un α adecuado...

Ejemplo 2.1. Supongamos que queremos encontrar una solución a la ecuación:

$$\cos(x) - 2x = 0$$

Es fácil ver que $f(x) = \cos(x) - 2x$. Entonces, se tiene una posible función g sería:

$$\underbrace{\cos(x) - x}_{g(x)} = x$$

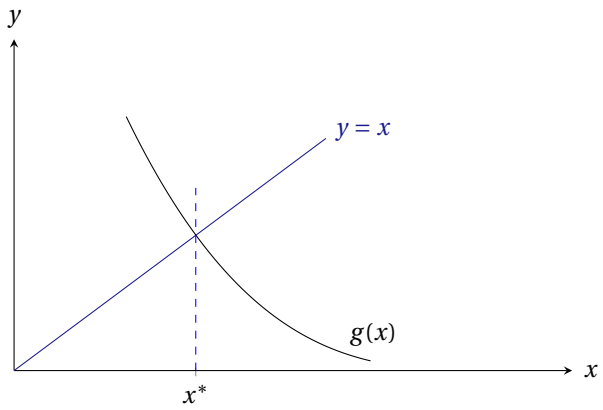


Figura 2.5 – La intersección entre $y = x$ y $g(x)$ es un punto fijo

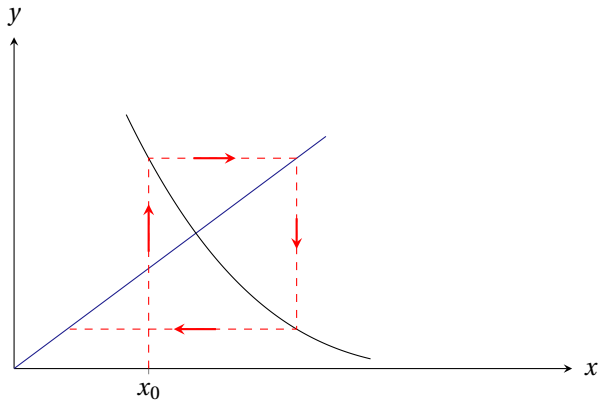


Figura 2.6 – En este caso, la espiral diverge (mire las flechas).

Usando iteraciones de punto fijo, se tiene que el cero de la función corresponde a la intersección entre $y = x$ y $g(x)$ (figura 2.5). Para efectos de convergencia, se puede ver como una espiral (figura 2.6 y figura 2.7). Nótese que para construir estas espirales debe partir por el eje x, en algún x_0 a su gusto. Luego, tirar una linea vertical hasta chocar con g. En seguida, continúe con una línea horizontal hasta llegar a $y = x$ y vuelva a trazar otra vertical hasta $g(x)\ldots$ y así sucesivamente hasta encontrar una aproximación suficiente al punto de convergencia.

2.2.1. Método de la Secante

La idea, al igual que para *regula falsi*, es calcular el intercepto de la recta que pasa por dos puntos, ver figura 2.8. Pero a diferencia de ese método, no se mantienen puntos fijos. Partiendo con x_0, x_1 se calculan valores sucesivos usando:

$$x_{n+2} = x_{n+1} - f(x_{n+1}) \cdot \frac{x_{n+1} - x_n}{f(x_{n+1} - f(x_n)}$$

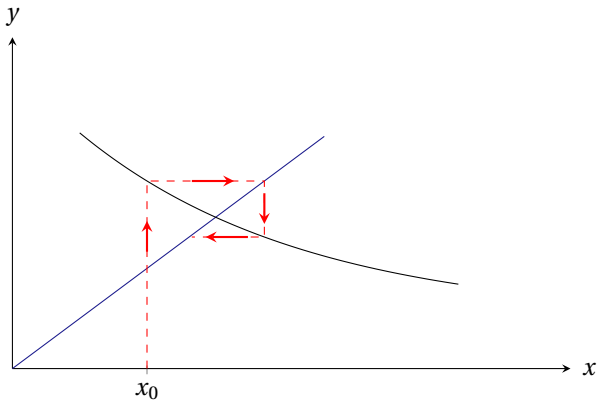


Figura 2.7 – En este caso, la espiral converge (mire las flechas).

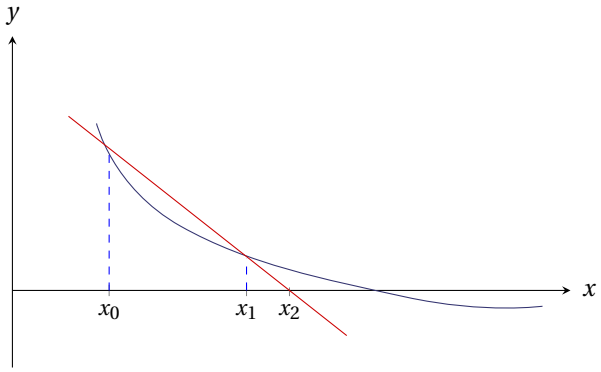


Figura 2.8 – Una iteración del método de la secante.

2.2.2. Método de la tangente (Newton)

La idea consiste en elegir un valor arbitrario x_0 que esté razonablemente cerca de x^* (cero de la función). Luego, sucesivamente encontramos el intercepto con el eje x de la recta tangente de la curva en x_n usando la ecuación:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{2.6}$$

resultando un valor x_{n+1} que se encuentra más cerca de x^* (figura 2.9). Finalmente, iteramos el proceso hasta obtener un valor de convergencia.

2.3. Orden de Convergencia

Si definimos $e_n = |x_n - x^*|$, se dice que un método es de *orden* p, si hay constantes $C > 0$, p tales que

$$|e_{n+1}| \leq C|e_n|^p \tag{2.7}$$

Donde e_n corresponde al error con n iteraciones.

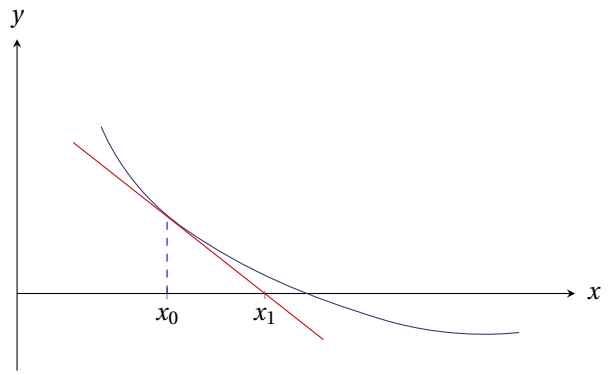


Figura 2.9 – El valor x_1 de (2.6) es más cercano a x^* que x_0.

- Si $p = 1$ hablamos de convergencia lineal. En este caso debe ser $0 < C < 1$ para convergencia. Bisección tiene $C = \frac{1}{2}$.
- Si $p = 2$ hablamos de convergencia cuadrática (en cada paso, el error se eleva a $p = 2$).
- Si $p = 3$ hablamos de convergencia cúbica.
- Si $1 < p < 2$ decimos que es superlineal.

Nótese que mientras mayor sea el orden de convergencia p, más rápido (en menos iteraciones) encontraremos una aproximación adecuada al valor buscado.

Clase 3

Análisis de Convergencia

Sea $f(x)$ la función que buscamos el cero x^*:

$$f(x^*) = 0$$

Suponiendo $f(x)$ continua, que puede derivarse tres veces en un entorno de x^*, por *teorema de Taylor*:

$$f(x) = f(x^*) + f'(x^*)(x - x^*) + \frac{1}{2} f''(x^*)(x - x^*)^2 + O((x - x^*)^3)$$

Definiendo $e = x - x^*$:

$$= f'(x^*)e(1 + Me + O(e^2) \tag{3.1}$$

con

$$M = \frac{f''(x^*)}{2f'(x^*)}$$

La forma de Lagrange del residuo es:

$$\frac{1}{3!} f'''(\zeta) e^3 \tag{3.2}$$

donde $x \leq \zeta \leq x^*$. Si llamamos:

$$e_n = x_n - x^* \tag{3.3}$$

donde n es el número de la iteración correspondiente. El análisis de cómo evoluciona el error al ir iterando debe efectuarse por separado para cada uno de los métodos.

3.1. Regula Falsi

Para *regula falsi*, recordemos que:

$$x_{n+1} = x_n - f(x_n) \cdot \frac{x_n - x_0}{f(x_n) - f(x_0)} \tag{3.4}$$

entonces, el error es:

$$\begin{aligned} e_{n+1} &= e_n - f(x_n)\cdot\frac{x_0 - x_n}{f(x_0) - f(x_n)} \\ &\approx e_n - f'(x^*)e_n\cdot\frac{x_0 - x^*}{f(x_0)} \\ &= e_n(1 - f'(x^*)\frac{e_0}{f(x_0)} \end{aligned}$$

Como $e_{n+1} \approx Ce_n$, convergencia lineal. Ojo, esto es sólo para *regula falsi*.

3.2. Método de Newton

Considere la figura 3.1. donde:

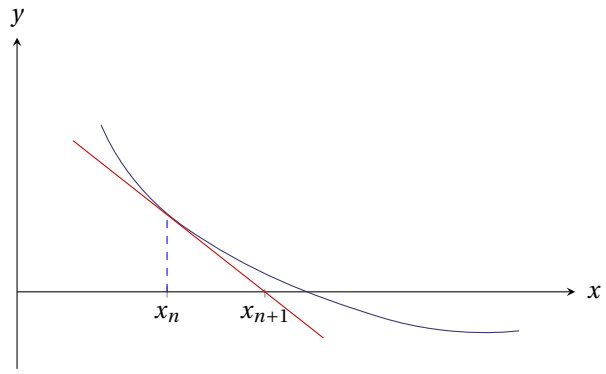


Figura 3.1 – Una iteración del método de Newton.

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{3.5}$$

Considere nuevamente la aproximación (3.1), entonces:

$$\begin{aligned} e_{n+1} &= e_n - \frac{f(x_n)}{f'(x_n)} \\ &\approx e_n - \frac{f'(x^*)e_n(1 + Me_n)}{f'(x^*)} \\ &= -Me_n^2 \\ &= -\frac{f''(x^*)}{2f'(x^*)}\cdot e_n^2 \end{aligned}$$

O sea, el método de Newton es cuadrático. En el fondo, el *método de Newton* duplica el número de cifras correctas en cada iteración. Supongamos que el error cumple $e_k/x^* = a\cdot 10^{-n}$, con $0 < a \leq 1$, o sea, conocemos x^* con n cifras a la iteración k. Por la forma aproximada del error:

$$\begin{aligned} \frac{e_{k+1}}{x^*} &\approx M\frac{e_k^2}{x^*} \\ &= Mx^*a^2\cdot 10^{-2n} \end{aligned}$$

Esto corresponde a conocer x^* con aproximadamente $2n$ cifras (siempre que el factor sea pequeño, claro).

3.3. Método de la Secante

Considere la figura 3.2, donde:

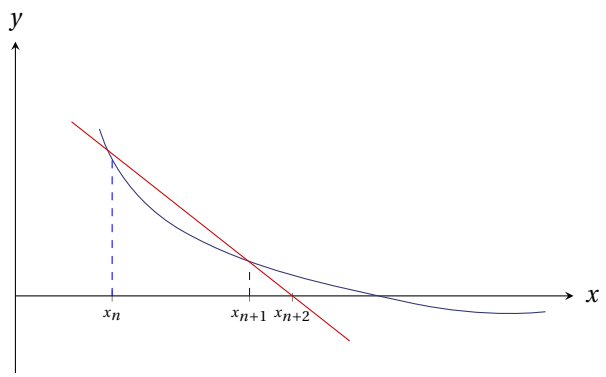


Figura 3.2 – Una iteración del método de la secante.

$$x_{n+2} = x_{n+1} - f(x_{n+1)} \cdot \frac{x_{n+1} - x_n}{f(x_{n+1)} - f(x_n)} \tag{3.6}$$

Considere nuevamente la aproximación (3.1), entonces:

$$\begin{aligned}
x_{n+2} &= x_{n+1} - f(x_{n+1}) \frac{x_{n+1} - x_n}{f(x_{n+1}) - f(x_n)} \\
e_{n+2} &= e_{n+1} - f(x_{n+1)} \frac{e_{n+1} - e_n}{f(x_{n+1}) - f(x_n)} \\
&= \frac{e_n f(x_{n+1}) - e_{n+1} f(x_n)}{f(x_{n+1}) - f(x_n)} \\
&\approx \frac{e_n f'(x^*) e_{n+1}(1 + M e_{n+1}) - e_{n+1} f'(x^*) e_n (1 + M e_n)}{f'(x^*)(e_{n+1} - e_n)} \\
&= \frac{e_n e_{n+1}(1 + M e_{n+1} - 1 - M e_n)}{e_{n+1} - e_n} \\
&= M e_{n+1} e_n
\end{aligned}$$

Supongamos:

$$\begin{aligned}
|e_{n+1}| &= C|e_n|^p \\
|e_{n+2}| &= C|e_{n+1}|^p \\
&= C(C|e_n)^p|^p \\
&= C^{p+1}|e_n|^{p^2}
\end{aligned}$$

Por lo tanto, para el método de la secante:

$$C^{p+1}|e_n|^{p^2} = MC|e_n|^{p+1} \tag{3.7}$$

Entonces:

$$p^2 = p + 1 \tag{3.8}$$

De acá:

$$p = \tau \approx 1{,}618$$

Por lo tanto, el método de la secante es *superlineal*.

Una discusión accesible de los problemas a resolver para construir una rutina "para uso general" es la de Kahan [1]. Kahan [2] revisa la teoría que sustenta búsqueda de ceros sin suponer que estamos "muy cerca".

Bibliografía

[1] William M. Kahan: *Personal calculator has key to solve any equation $f(x) = 0$.* Hewlett-Packard Journal, 30(12):20–26, December 1979.

[2] William M. Kahan: *Lecture notes on real root-finding.* https://people.eecs.berkeley.edu/~wkahan/Math128/RealRoots.p March 2016.

Clase 4

Iteración de Punto Fijo

Definición 4.1. Sea $g(x)$ una función. Un *punto fijo* de g es x^* tal que $x^* = g(x^*)$.

Teorema 4.1 (Punto fijo de Brouwer, una dimensión)**.** *Sea* $g\colon [a,b] \to [a,b]$ *una función continua. Entonces* g *tiene* al menos *un punto fijo en* $[a,b]$.

Demostración. Por definición de g, sabemos:

$$a \leq g(x) \leq b$$

En particular, $a \leq g(a)$ y $g(b) \leq b$. Si alguna vez se cumple con igualdad estamos listos.

Supongamos entonces $a < g(a)$ y $g(b) < b$. Consideremos $f(x) = g(x) - x$. Vemos que f es continua, y $f(a) = g(a) - a > 0$, $f(b) = g(b) - b < 0$. Por el *teorema del valor medio*, hay $x^* \in [a,b]$ tal que $f(x^*) = 0$, o sea, $x^* = g(x^*)$. □

Definición 4.2. Sea $g\colon [a,b] \to [a,b]$. Se dice que g es una *contracción* si existe L, $0 < L < 1$, tal que para todo $x, y \in [a,b]$ es:

$$|g(x) - g(y)| \leq L|x - y| \tag{4.1}$$

(condición de Lipschitz, L es la constante de Lipschitz).

Teorema 4.2 (Contraction Mapping)**.** *Suponga que* $g\colon [a,b] \to [a,b]$ *es continua y cumple la condición de Lipschitz. Entonces tiene un único punto fijo en* $[a,b]$.

Demostración. Por el teorema de Brouwer, g tiene al menos un punto fijo. Para demostrar que es único, supongamos puntos fijos c_1, c_2:

$$|c_1 - c_2| = |g(c_1) - g(c_2)| \leq L|c_1 - c_2| \tag{4.2}$$

Como $0 < L < 1$, esto es posible solo si $c_1 = c_2$. □

Esto es algo bastante obvio, ya que en el fondo tomamos un área más grande y en cada iteración la vamos reduciendo hasta tal punto que $c_1 = c_2$ (Figura 4.1) Definamos la secuencia:

$$x_{n+1} = g(x_n) \tag{4.3}$$

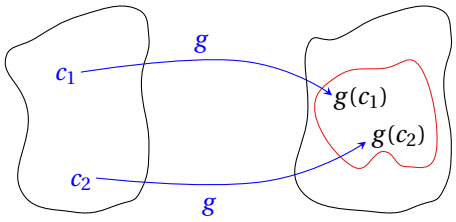


Figura 4.1 – Acotamos el área hasta converger en un punto.

Si g es una contracción en $[a,b]$, la secuencia converge al punto fijo x^* de g en $[a,b]$.

De partida, si

$$\lim_{n\to\infty} x_n = x^*$$

existe, es un punto fijo de g. Si $x_0 \in [a,b]$, consideremos:

$$\begin{aligned} |x_{n+1} - x^*| &= |g(x_n) - g(x^*)| && (4.4)\\ &\leq L|x_n - x^*| && (4.5)\\ &\leq \dots && (4.6)\\ &\leq L^{n+1}|x_0 - x^*| && (4.7) \end{aligned}$$

Como $|L| < 1$, $L^n \to 0$, y el lado izquierdo también tiende a 0 (para llegar a (4.7) sólo tenemos que desarrollar paso a paso (4.5)).

Supongamos que queremos llegar a $|x_n - x^*| \leq \epsilon$. Sabemos que $|x_n - x^*| \leq L^n|x_0 - x^*|$. Queremos deshacernos del x^* desconocido al lado derecho:

$$\begin{aligned} |x_0 - x^*| &= |(x_0 - x_1) + (x_1 - x^*)|\\ &\leq |x_0 - x_1| + |x_1 - x^*|\\ &\leq |x_0 - x_1| + L|x_0 - x^*| \quad \Big/ - L|x_0 - x^*|\\ (1-L)|x_0 - x^*| &\leq |x_1 - x_0|\\ |x_0 - x^*| &\leq \frac{|x_1 - x_0|}{1-L} \end{aligned}$$

O sea:

$$|x_n - x^*| \leq \frac{L^n}{1-L}|x_1 - x_0| \tag{4.8}$$

Queremos $|x_n - x^*| \leq \epsilon$:

$$\begin{aligned} \epsilon &\leq \frac{L^n}{1-L}|x_1 - x_0|\\ L^n &\geq \frac{\epsilon(1-L)}{|x_1 - x_0|}\\ n &\geq \frac{1}{\ln L} \cdot \ln \frac{\epsilon(1-L)}{|x_1 - x_0|} \end{aligned}$$

No hemos supuesto g diferenciable, pero en casos de interés lo es.

La condición de Lipschitz es:

$$\frac{|g(x)-g(y)|}{|x-y|} \leq L$$
$$\left|\frac{g(x)-g(y)}{x-y}\right| \leq L$$

Por el teorema del valor intermedio (ver por ejemplo las notas de Chen [1]):

$$\frac{g(x)-g(y)}{x-y} = g'(\zeta), \qquad x \leq \zeta \leq y \tag{4.9}$$

por lo tanto, $|g'(\zeta)| \leq L$ para $\zeta \in [a,b]$ es condición suficiente para Lipschitz, se aplica el teorema de contraction mapping y hay un único punto fijo en $[a,b]$ y $x_{n+1} = g(x_n)$ converge.

Importante: No buscamos encontrar ζ, sólo demostrar que existe. Y por favor, ζ se lee "zeta".

Bibliografía

[1] William W. L. Chen: *First year calculus.*
http://rutherglen.science.mq.edu.au/wchen/lnfycfolder/lnfyc.html, 2008.

Clase 5

Interpolación

Idea: Nos dan los valores exactos de una función desconocida en $n+1$ puntos $f(x_0),\ldots,f(x_n)$, queremos hallar una función que tome esos valores (para calcular valores intermedios). El caso más común es utilizar *polinomios*. Para esta ocasión sabemos que hay exactamente *un* polinomio de grado a lo más n que pasa por $n+1$ puntos.

Supongamos que hay dos polinomios distintos, $p(x)$ y $q(x)$, de grado a lo más n que pasan los $n+1$ puntos (véase la figura 5.1). Entonces $p(x)-q(x)$ es un polinomio de grado a lo más n que tiene

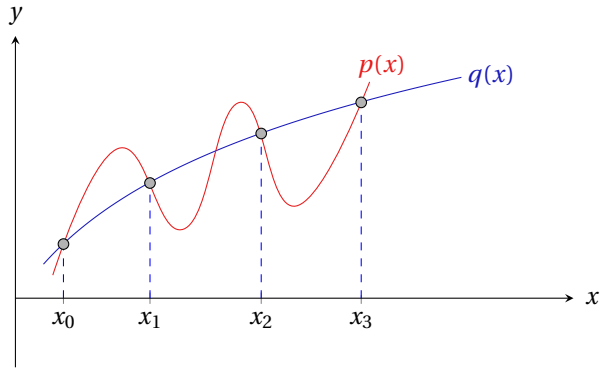


Figura 5.1 – Sabemos que $p(x)$ y $q(x)$ se interceptan en x_0, x_1, x_2 y x_3.

$n+1$ ceros $x_0,\ldots,x_n$, implicando que $p(x)-q(x)$ es exactamente igual a cero.

Entre las cosas que podemos hacer para hallar la interpolación de $f(x)$ tenemos:

- De forma implícita.
- Con un sistema de ecuaciones.

5.1. De forma implícita

Para encontrar la interpolación de f de manera implícita simplemente inventamos un polinomio que pase por los puntos. Para ello podemos usar varios métodos:

- Lagrange
- Newton

5.1.1. Forma de Lagrange

Consiste en usar el polinomio:

$$\begin{aligned} p(x) &= f(x_0)\frac{(x-x_1)(x-x_2)\cdots(x-x_n)}{(x_0-x_1)(x_0-x_2)\cdots(x_0-x_n)} + f(x_1)\frac{(x-x_0)(x-x_2)\cdots(x-x_n)}{(x_1-x_0)(x_1-x_2)\cdots(x_1-x_n)} + \cdots \\ &\quad + f(x_n)\frac{(x-x_0)\cdots(x-x_{n-1})}{(x_n-x_0)\cdots(x_n-x_{n-1})} \\ &= \sum_{0\le k\le n} f(x_k) \prod_{\substack{0\le j\le n \\ j\neq k}} \frac{x-x_j}{x_k-x_j} \end{aligned} \tag{5.1}$$

como interpolación de f. Es claro que si evalúa p en alguno de los x_k con $k \in \{1,2,\ldots,n\}$ que nos entregan, se tiene que $p(x_k) = f(x_k)$, y el polinomio es de grado n.

En la práctica, la fórmula (5.1) es poco deseable, requiere $O(n^2)$ computación para calcular $p(x)$. Una forma alternativa es la *fórmula baricéntrica* que recomienda Winrich [4] al comparar varias alternativas en términos de números de operaciones:

$$p(x) = \frac{\sum_{0\le j\le n} \frac{w_j f(x_j)}{x-x_j}}{\sum_{0\le j\le n} \frac{w_j}{x-x_j}} \tag{5.2}$$

donde:

$$w_j = \frac{1}{\prod_{\substack{0\le k\le n \\ k\neq j}} (x_j - x_k)} \tag{5.3}$$

Berrut y Trefthen [1] discuten esta fórmula en detalle. Higham [2] analiza esta técnica, concluyendo que es numéricamente estable y recomendable para uso general.

5.1.2. Forma de Newton

Podemos escribir:

$$\begin{aligned} Q_0(x) &= f(x_0) = a_0 \\ Q_1(x) &= f(x_0) + (x_1 - x_0)\underbrace{\frac{f(x_1)-f(x_0)}{x_1-x_0}}_{Q_0(x_1)} \\ Q_2(x) &= f(x_0) + (x-x_0)\frac{f(x_1)-f(x_0)}{x_1-x_0} + (x-x_0)(x-x_1)\frac{f(x_2)-Q_1(x_2)}{(x_2-x_0)(x_2-x_1)} \\ &\vdots \\ a_k &= \frac{f(x_k)-Q_{k-1}(x_k)}{\prod_{0\le i\le k-1}(x_k-x_i)}, \qquad Q_k(x) = Q_{k-1}(x) + (x-x_0)\cdots(x-x_{k-1})a_k \end{aligned} \tag{5.4}$$

donde Q_k corresponde a la interpolación de grado k.

Como comentario al margen, la fórmula prohibida que vimos en la primera clase (ecuación (5.5)) que asegura VTR 2 hace que la precisión se vaya a las pailas.

$$x_2 = \frac{f(x_1)\cdot x_0 - f(x_0)\cdot x_1}{f(x_1) - f(x_0)} \tag{5.5}$$

5.2. Por sistema de ecuaciones

Suponiendo $p(x) = a_0 + a_1 x + \cdots + a_n x^n$, creamos un sistema de ecuaciones de la forma:

$$\begin{aligned}
p(x_0) = f(x_0) &= a_0 + a_1 x_0 + a_2 x_0^2 + \cdots a_n x_0^n \\
p(x_1) = f(x_1) &= a_0 + a_1 x_1 + a_2 x_1^2 + \cdots a_n x_1^n \\
&\vdots \\
p(x_n) = f(x_n) &= a_0 + a_1 x_n + a_2 x_n^2 + \cdots a_n x_n^n
\end{aligned}$$

que matricialmente se escribe como:

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} f(x_0) \\ f(x_1) \\ \vdots \\ f(x_n) \end{pmatrix} \tag{5.6}$$

Con igual cantidad de ecuaciones e incógnitas, el sistema de ecuaciones (5.6) tiene solución única si y solo si el determinante es distinto de cero (ver por ejemplo Treil [3]):

$$\begin{vmatrix} 1 & \cdots & x_0^n \\ \vdots & \ddots & \vdots \\ 1 & \cdots & x_n^n \end{vmatrix} \neq 0 \tag{5.7}$$

El determinante de la ecuación (5.7) resulta ser el *determinante de Vandermonde*.

Teorema 5.1. *El determinante de Vandermonde vale:*

$$\begin{vmatrix} 1 & a_1 & a_1^2 & \cdots & a_1^{n-1} \\ 1 & a_2 & a_2^2 & \cdots & a_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & a_n & a_n^2 & \cdots & a_n^{n-1} \end{vmatrix} = \prod_{1 \leq i < j \leq n} (a_j - a_i)$$

Demostración. Por inducción sobre n. El caso $n = 1$ es trivial, es simplemente:

$$\left|1\right| = 1$$

Usamos $n = 2$ como base. Para abreviar, llamaremos:

$$V_n = \begin{vmatrix} 1 & a_1 & a_1^2 & \cdots & a_1^{n-1} \\ 1 & a_2 & a_2^2 & \cdots & a_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & a_n & a_n^2 & \cdots & a_n^{n-1} \end{vmatrix}$$

Base: Cuando $n = 2$ tenemos:

$$\begin{vmatrix} 1 & a_1 \\ 1 & a_2 \end{vmatrix} = a_2 - a_1$$

que ciertamente se cumple.

Inducción: Supongamos que vale para k, y consideremos el determinante:

$$\begin{vmatrix} 1 & a_1 & a_1^2 & \cdots & a_1^k \\ 1 & a_2 & a_2^2 & \cdots & a_2^k \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & a_k & a_k^2 & \cdots & a_k^k \\ 1 & x & x^2 & \cdots & x^k \end{vmatrix}$$

Expandiendo por la última fila, vemos que es un polinomio en x de grado a lo más k, llamémosle $f(x)$. Como determinantes con filas iguales son cero, sabemos que $f(a_i) = 0$ para $1 \le i \le k$, o sea:

$$\begin{aligned} f(x) &= c(x-a_1)(x-a_2)\cdots(x-a_k) \\ &= c \prod_{1 \le i \le k} (x - a_i) \end{aligned}$$

Como el grado de f es a lo más k, y tenemos k factores lineales en x, c es independiente de x. La expansión por la última fila, nuevamente, muestra que el coeficiente de x^k es V_k, con lo que $c = V_k$. Reemplazando $x \mapsto a_{k+1}$ con nuestra hipótesis de inducción entrega:

$$\begin{aligned} V_{k+1} &= V_k \cdot \prod_{1 \le i \le k} (a_{k+1} - a_i) \\ &= \prod_{1 \le i < j \le k} (a_j - a_i) \cdot \prod_{1 \le i \le k} (a_{k+1} - a_i) \\ &= \prod_{1 \le i < j \le k+1} (a_j - a_i) \end{aligned}$$

Por inducción, vale para todo $n \in \mathbb{N}$. □

Ejercicios

1. Demuestre la fórmula baricéntrica (5.2). Defina:

   $$\ell(x) = \prod_{0 \le j \le n} (x - x_j)$$

   escriba (5.1) como:

   $$p(x) = \ell(x) \sum_{0 \le j \le n} \frac{w_j f(x_j)}{x - x_j}$$

   Use (5.1) para interpolar la función 1, divida y simplifique.

2. Plantee algoritmos eficientes para evaluar el polinomio interpolador en x dados los puntos $x_0, \ldots, x_n$ usando las fórmulas (5.1), (5.2) y (5.4). Separe inicialización que depende de los x_j de cálculos que dependen de x (o sea, maneje el caso en que dados los x_j interesa evaluar para varios x). Calcule el número de operaciones de punto flotante (*flops*) para evaluar el polinomio interpolador en x en cada caso.

Bibliografía

[1] Jean Paul Berrut and Lloyd N. Trefthen: *Barycentric Lagrange interpolation.* SIAM Review, 46(3):501–517, 2004.

[2] Nicholas J. Higham: *The numerical stability of barycentric Lagrange interpolation.* IMA Journal of Numerical Analysis, 24(4):547–556, October 2004.

[3] Sergei Treil: *Linear algebra done wrong.* https://www.math.brown.edu/~treil/papers/LADW/LADW.html, September 2014.

[4] Lonny B. Winrich: *Note on a comparison of evaluation schemes for the interpolating polynomial.* Computer Journal, 12(2):154–155, 1969.

Clase 6

Error de Interpolación

La clase pasada vimos cómo obtener la interpolación dado los pares de puntos $(x_k, f(x_j)$ con $k \in \{0, \dots, n\}$ que nos daban. Nuestro tema de interés ahora es obtener el error de esas interpolaciones (figura 6.1).

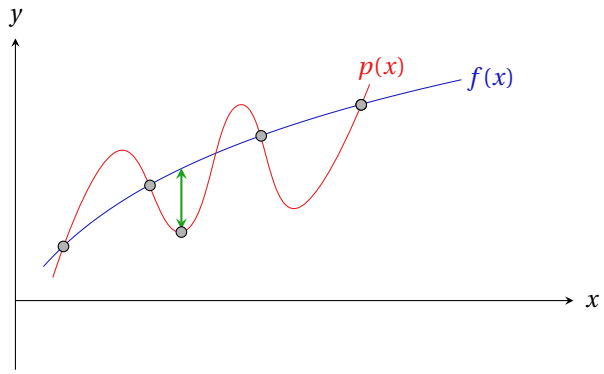


Figura 6.1 – El error es la diferencia entre $p(x)$ y $f(x)$ en cada punto (flecha verde).

Teorema 6.1 (Rolle)**.** *Si f es continua en $[a,b]$ y tiene derivada continua en $[a,b]$, si $f(a) = f(b) = 0$, hay $x^* \in (a,b)$ tal que $f'(x^*) = 0$ (pariente del teorema del valor medio de la derivada).*

Teorema 6.2. *Sea $f \in C^{n+1}[a,b]$. Sea $Q_n(x)$ el polinomio de grado n que interpola f en los puntos distintos $x_0, \dots, x_n \in [a,b]$. Entonces para todo $x \in [a,b]$ hay $\zeta \in [a,b]$ tal que*

$$f(x) - Q_n(x) = \underbrace{\frac{1}{(n+1)!} f^{(n+1)}(\zeta) \prod_{0 \le j \le n} (x - x_j)}_{Error} \tag{6.1}$$

Demostración. Fijemos $x \in [a,b]$. Si $x = x_j$ para algún j, el lado izquierdo y derecho de (6.1) se anulan y estamos listos.

En caso contrario, sea

$$\omega(x) = \prod_{0 \le j \le n} (x - x_j) \tag{6.2}$$

Notamos para referencia futura que $\omega(x)$ es mónico,[1] con lo que $\omega^{(n+1)}(x) = (n+1)!$.

Definamos:

$$F(y) = f(y) - Q_n(y) - \lambda\omega(y) \tag{6.3}$$

donde elegimos λ tal que $F(x) = 0$.

La función $F(y)$ está en $C^{n+1}[a,b]$, y tiene $n+2$ ceros en $[a,b]$ $(x, x_0, \ldots, x_n)$. Por el teorema de Rolle, $F'(y)$ tiene $n+1$ ceros en $[a,b]$, $\ldots$, $F^{(n+1)}(y)$ tiene un cero en $[a,b]$, llamémosle ζ. O sea:

$$F^{(n+1)}(\zeta) = f^{(n+1)}(\zeta) - \cancelto{0}{Q_n^{(n+1)}(\zeta)} - \lambda \cancelto{(n+1)!}{\omega^{(n+1)}(\zeta)} = 0$$

Vale decir:

$$f^{(n+1)}(\zeta) = \lambda(n+1)!$$

Con esto:

$$\lambda = \frac{f^{(n+1)}(\zeta)}{(n+1)!}$$

$$F(y) = f(y) - Q_n(y) - \frac{f^{(n+1)}(\zeta)}{(n+1)!}\omega(x)$$

El error en x es:

$$\frac{f^{(n+1)}(\zeta)}{(n+1)!}\omega(x)$$

□

Acá n lo elegimos nosotros y corresponde al grado de la interpolación. Es claro que mientras mayor sea el grado de nuestra interpolación, menor será el error (vea el $(n+1)!$ como denominador).

6.1. Cuadratura

Queremos evaluar:

$$\int_a^b f(x)\mathrm{d}x \tag{6.4}$$

Supongamos f dado en $x_0, \ldots, x_n$ (aka *puntos de cuadratura*). Para encontrar el valor de (6.4) simplemente interpolamos f, e integramos el polinomio interpolante.

6.1.1. Caso más simple: Polinomio de grado 0

Corresponde a una aproximación con rectángulos (figura 6.2) Para ello, usábamos la fórmula:

[1] Para un polinomio de grado n, el coeficiente que acompaña a x^n es 1.

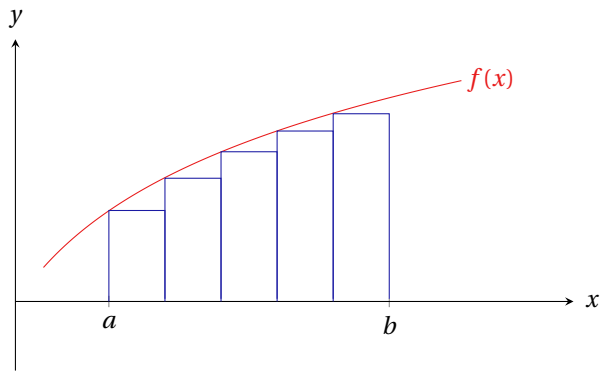


Figura 6.2 – Aproximar el área bajo una curva usando rectangulitos (integral de Riemann).

$$\int_a^b f(x)\mathrm{d}x \approx \sum_{0\le j\le n} f(x_j)(x_{j+1}-x_j) \qquad a=x_0, b=x_n$$

Si son igualmente espaciados, se tiene que $x_{j+1}-x_j = h$ para $0 \le j < n$

$$\int_a^b f(x)\mathrm{d}x \approx h \sum_{0\le j<n} f(x_j)$$

Consideremos la antiderivada[2]:

$$F(x) = \int_a^x f(t)\mathrm{d}t$$

Expandiendo en serie de Taylor:

$$\begin{aligned} F(x) &= F(a) + F'(a)(x-a) + \frac{1}{2!}F''(a)(x-a)^2 + O((x-z)^3) \\ &= f(a)(x-a) + \frac{1}{2!}f'(\xi)(x-a)^2 \end{aligned}$$

donde $a \le \xi \le x$.

Escribiendo $h = b-a$, esto es:

$$\begin{aligned} \int_a^b f(x)\mathrm{d}x &= f(a)h + \frac{f'(\xi)}{2}h^2 \\ E &= \int_a^b f(x)\mathrm{d}x - f(a)h \\ &= \frac{f'(\xi)}{2}h^2 \end{aligned}$$

El error es cuadrático en h.

¿Podemos obtener un error más pequeño que el caso de la figura 6.2?

6.1.2. Variante del caso simple: punto medio

En lugar de considerar una cota como se hizo en el caso de la figura 6.2, el punto de evaluación de $f(x)$ será el punto medio de la base del rectángulo (figura 6.3). Suponemos que f es integrable, y

[2] Es claro suponer que la antiderivada existe, de lo contrario este cuento no tiene chiste.

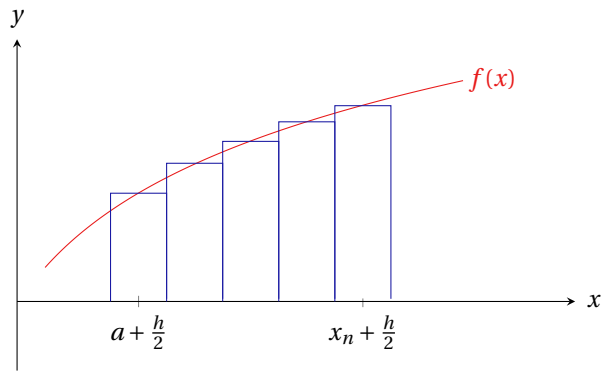


Figura 6.3 – El excedente de "triangulitos" por sobre f compensan la falta de éstos que están bajo f.

que además tiene "suficientes" derivadas continuas.

Para este caso se tiene que:

$$\int_a^b f(x)\mathrm{d}x \approx (b-a) f\left(\frac{a+b}{2}\right)$$

Expandimos usando series de Taylor:

$$\begin{aligned} F(a+h) &= F(a) + F'(a)h + \frac{1}{2}F''(a)h^2 + \frac{1}{6}F'''(a)h^3 + O(h^4) \\ &= f(a)h + \frac{1}{2}f'(a)h^2 + \frac{1}{6}f''(a)h^3 + O(h^4) \end{aligned}$$

Si $b = a+h$, tenemos (expandiendo $f(a+h/2)$) para el error:

$$\begin{aligned} E &= \int_a^{a+h} f(x)\mathrm{d}x - hf\left(a+\frac{h}{2}\right) \\ &= hf(a) + \frac{h^2}{2}f'(a) + \frac{h^3}{6}f''(a) + O(h^4) - h\left(f(a) + \frac{h}{2}f'(a) + \frac{h^2}{8}f''(a) + O(h^3)\right) \\ &= hf(a) + \frac{h^2}{2}f'(a) + \frac{h^3}{6}f''(a) + O(h^4) - \left(hf(a) + \frac{h^2}{2}f'(a) + \frac{h^3}{8}f''(a) + O(h^4)\right) \\ &= \frac{1}{24}f''(a)h^3 + O(h^4) \end{aligned} \tag{6.5}$$

Sorprendentemente, el error es de carácter cúbico. Si nos dan la segunda derivada de f en a, estamos listos.

6.1.2.1. Teorema del Valor Intermedio Ad Hoc

Considerando el error obtenido en la ecuación (6.5):

$$E = \frac{1}{24}f''(\zeta)h^3 \qquad \text{con } a \le \zeta \le a+h \tag{6.6}$$

Suponiendo intervalitos $a = x_0, x_1, \ldots, x_n = b$ con $x_{j+1} - x_j = h$, tenemos para cada uno:

$$E_j = \frac{1}{24}f''(\zeta_j)h^3, \qquad x_j \le \zeta_j \le x_{j+1}$$

Si $m \le f''(x) \le M$ en $[a,b]$:

$$E = \sum_j E_j = \frac{h^3}{24} \sum_j f''(\zeta_j) = \frac{nh^3}{24} f''(\zeta)$$

porque:

$$nm \le \sum_j f''(\zeta_j) \le nM$$
$$m \le \frac{1}{n} \sum_j f''(\zeta_j) \le M$$

Lo que nos dice que hay $\zeta \in [a,b]$ tal que:

$$\frac{1}{n} \sum_j f''(\zeta_j) = f''(\zeta)$$

6.1.3. Polinomio de grado 1 (trapezoides)

La siguiente idea más simple es aproximar $f(x)$ mediante la recta que pasa por $(a, f(a))$ y $(b, f(b))$, lo que lleva a la aproximación:

$$\int_a^b f(x)\mathrm{d}x = \frac{1}{2}(f(a) + f(b))(a-b)$$

Si tenemos múltiples intervalos, digamos n con $x_0, \ldots, x_n$ donde $x_{i+1} - x_i = h$, donde $a = x_0$ y $b = x_n$, la regla se traduce en:

$$\int_a^b f(x)\mathrm{d}x = \left(\frac{1}{2}(f(x_0) + f(x_n)) + \sum_{0<i<n} f(x_i) \right) \frac{b-a}{n}$$

El análisis tradicional es relativamente complejo, pero Cruz-Uribe y Neugebauer [1] proponen la técnica que usaremos. Consideremos un intervalo, tenemos para el error:

$$E = \frac{b-a}{2}(f(a) + f(b)) - \int_a^b f(x)\mathrm{d}x \tag{6.7}$$

Sea c el centro del intervalo:

$$c = \frac{a+b}{2}$$

con lo que:

$$b - c = c - a = \frac{b-a}{2}$$

De (6.7) vemos que corresponde a usar integración por partes “en reversa”:

$$E = \int_a^b (x-c) f'(x)\mathrm{d}x$$

Nuevamente integrando por partes:

$$E = \frac{1}{2} \int_a^b \left(\left(\frac{b-a}{2} \right)^2 - (x-c)^2 \right) f''(x)\mathrm{d}x$$

Si para $a \le x \le b$ tenemos la cota:

$$|f''(x)| \le M$$

vemos que:

$$\begin{aligned} E &\le \frac{1}{2}\int_a^b \left|\left(\frac{b-a}{2}\right)^2 - (x-c)^2\right| |f''(x)| \mathrm{d}x \\ &\le \frac{M}{2}\int_a^b \left(\left(\frac{b-a}{2}\right)^2 - (x-c)^2\right) \mathrm{d}x \\ &= \frac{M}{2}\cdot\frac{1}{6}(b-a)^3 \\ &= \frac{M}{12}(b-a)^3 \end{aligned}$$

Sorprendentemente, el error es cúbico.

6.1.4. Regla de Simpson

El siguiente paso es interpolar con una parábola (polinomio cuadrático). Hay varias maneras, más o menos complicadas, para deducir la fórmula del caso. Es recomendable el artículo de Talvila y Wiersma [3], que resume derivaciones sencillas de los métodos más comunes. Acá tomaremos un camino distinto.

Integrar el polinomio interpolador cuadrático significa 3 puntos, que para simplificar consideramos igualmente espaciados, en a, $(a+b)/2$ y b. Una transformación lineal transforma esto en los puntos $-1, 0, 1$, aún más cómodos. Lo que buscamos es una fórmula de la forma:

$$\int_{-1}^{1} f(x)\mathrm{d}x = w_{-1}f(-1) + w_0 f(0) + w_1 f(1)$$

Si esto es exacto para polinomios hasta de grado 2, quiere decir que:

$$\begin{aligned} \int_{-1}^{1} \mathrm{d}x &= 2 = w_{-1} + w_0 + w_1 \\ \int_{-1}^{1} x\mathrm{d}x &= 0 = -w_{-1} + w_1 \\ \int_{-1}^{1} x^2\mathrm{d}x &= \frac{2}{3} = w_{-1} + w_1 \end{aligned}$$

Notamos que también:

$$\int_{-1}^{1} x^3\mathrm{d}x = 0 = -w_{-1} + w_1$$

que simplemente repite la ecuación que tenemos para x, inesperadamente el método es exacto hasta grado 3. La solución a nuestro sistema de ecuaciones es $w_{-1} = w_1 = 1/3$, $w_0 = 4/3$.

Tenemos el siguiente resultado:

Teorema 6.3. *Sean $f \in C^4([a,b])$, $h = (b-a)/2$, y llamemos $x_0 = a$, $x_1 = x_0 + h$, $x_2 = b$. Entonces hay $\xi \in [a,b]$ tal que:*

$$E = \int_a^b f(x)\mathrm{d}x - \frac{h}{3}(f(x_0) + 4f(x_1) + f(x_2)) = -\frac{h^5}{90}f^{(4)}(\xi)$$

Demostración. Esta demostración viene de Süli y Mayers [2, capítulo 7]. Considere el cambio de variable:

$$x(t) = x_1 + ht \quad t \in [-1, 1]$$

Defina $F(t) = f(x(t))$, con lo que $\mathrm{d}x = h\mathrm{d}t$, y nuestra integral es:

$$\int_{x_0}^{x_1} f(x)\mathrm{d}x = h\int_{-1}^{1} F(\tau)\mathrm{d}\tau$$

y el error es:

$$E = \int_a^b f(x)\mathrm{d}x - \frac{h}{3}(f(x_0) + 4f(x_1) + f(x_2)) = h\left(\int_{-1}^{1} F(\tau)\mathrm{d}\tau - \frac{1}{3}(F(-1) + 4F(0) + F(1))\right)$$

Para $t \in [-1, 1]$ definamos la función:

$$G(t) = \int_{-t}^{t} F(\tau)\mathrm{d}\tau - \frac{t}{3}(F(-t) + 4F(0) + F(t))$$

En particular, el error de integración que nos interesa estimar es $E = hG(1)$. Consideremos ahora:

$$H(t) = G(t) - t^5 G(1)$$

Vemos que $H(0) = H(1) = 0$, con lo que por el teorema de Rolle hay $\xi_1 \in (0, 1)$ tal que $H'(\xi_1) = 0$. Como a su vez $H'(0) = 0$, por el teorema de Rolle hay $\xi_2 \in (0, \xi_1) \subset (0, 1)$ tal que $H''(\xi_2) = 0$. El mismo argumento muestra que hay $\xi_3 \in (0, 1)$ con $H'''(\xi_3) = 0$. Nótese que la tercera derivada de G es:

$$G'''(t) = -\frac{1}{3}(F'''(t) - F'''(-t))$$

de donde:

$$H'''(\xi_3) = -\frac{\xi_3}{3}(F'''(\xi_3) - F'''(-\xi_3)) - 60\xi_3^2 G(1) = 0$$

O sea, dividiendo la última igualdad por $2\xi_3 = \xi_3 - (-\xi_3)$, lo que es válido ya que $\xi_3 \neq 0$, y reorganizando:

$$-\frac{F'''(\xi_3) - F'''(-\xi_3)}{\xi_3 - (-\xi_3)} = 90G(1)$$

Por el teorema del valor medio de la derivada, sabemos que hay $\xi_4 \in (-\xi_3, \xi_3) \subset (-1, 1)$ tal que:

$$90G(1) = -F^{(4)}(\xi_4)$$

de donde tenemos para el error:

$$\begin{aligned} E &= -\frac{h}{90}F^{(4)}(\xi_4) \\ &= -\frac{h^5}{90}f^{(4)}(x_1 + h\xi_4) \\ &= -\frac{h^5}{90}f^{(4)}(\xi) \end{aligned}$$

donde $\xi = x_1 + h\xi_4 \in (a, b)$. □

Ejercicios

1. Derive las fórmulas de cuadratura integrando los polinomios interpolantes. Compare con nuestras derivaciones.

2. Hay una segunda regla de Simpson, que parte de interpolación cúbica. Derive esa fórmula mediante nuestra técnica, junto con la estimación del error (conviene elegir 4 puntos igualmente espaciados, centrados en 0).

Bibliografía

[1] David Cruz-Uribe and C. J. Neugebauer: *An elementary proof of error estimates for the trapezoidal rule*. Mathematics Magazine, 76(4):303–306, October 2003.

[2] Endre Süli and David F. Mayers: *An Introduction to Numerical Analysis*. Cambridge University Press, 2003.

[3] Erik Talvila and Matthew Wiersma: *Simple derivation of basic quadrature formulas*. Atlantic Electronic Journal of Mathematics, 5(1):47–59, 2012.

Clase 7

Cuadratura Gaussiana

Estamos interesados en investigar la posibilidad de escribir cuadraturas (cálculo de la integral definida) más precisas sin incrementar el número de *puntos de cuadratura* (aka $x_0, x_1, \ldots, x_n$). Esto puede ser posible si nos tomamos la libertad de escoger estos puntos. Por lo tanto, el problema de cuadratura se transforma en un problema de escoger los puntos de cuadratura en adición a determinar los respectivos coeficientes tal que la cuadratura es exacta para los polinomios de grado máximo. Las cuadraturas que son obtenidas con este método se conocen como *cuadratura gaussiana*.

Ejemplo 7.1 (Cuadratura gaussiana con $n = 2$)**.** Supongamos que queremos encontrar dos puntos de cuadratura de la ecuación:

$$\int_{-1}^{1} f(x)\mathrm{d}x \approx a_0 f(x_0) + a_1 f(x_1) \tag{7.1}$$

Como queremos encontrar los valores de a_0, a_1, x_0 y x_1, esperamos que la ecuación (7.1) sea exacta para polinomios hasta de grado $2 \cdot 2 - 1 = 3$. O sea, es exacta para $1, x, x^2, x^3$ (puede reemplazar $f(x)$ por cualquier otra cosa, claro, si es que busca complicarse la existencia...). Entonces, reemplazamos $f(x) = x^k$ con $k \in \{0,1,2,3\}$ para la k-ésima ecuación, y con ello, formamos el siguiente sistema de ecuaciones:

$$\int_{-1}^{1} x^k \mathrm{d}x = a_0 x_0^k + a_1 x_1^k \qquad k \in \{0,1,2,3\} \tag{7.2}$$

Resolvemos la integral:

$$\int_{-1}^{1} x^k \mathrm{d}x = \frac{x^{k+1}}{k+1}\bigg|_{-1}^{1} = \begin{cases} 0, & \text{si } k \text{ es impar} \\ \frac{2}{k+1}, & \text{si } k \text{ es par} \end{cases}$$

y reemplazamos en el sistema de ecuaciones (7.2):

$$2 = a_0 + a_1 \tag{7.3}$$
$$0 = a_0 x_2 + a_1 x_1 \tag{7.4}$$
$$\frac{2}{3} = a_0 x_2^2 + a_1 x_1^2 \tag{7.5}$$
$$0 = a_0 x_0^3 + a_1 x_1^3 \tag{7.6}$$

Finalmente, al resolver el sistema de ecuaciones de (7.3), (7.4), (7.5) y (7.6) se obtiene:

$$a_0 = 1, \qquad a_1 = 1, \qquad x_0 = \frac{1}{\sqrt{3}}, \qquad x_1 = -\frac{1}{\sqrt{3}}$$

Una observación es que los coeficientes de los valores extremos a_0 y a_1 son iguales:

$$a_0 = a_1$$

y que los puntos de cuadratura extremos x_0 y x_1 son opuestos, vale decir:

$$x_0 = -x_1$$

Lo anterior es extendible para n puntos de cuadratura, donde para $0 \leq i < n$:

$$a_i = a_{n-i-1}$$
$$x_i = -x_{n-i-1}$$

Importante: No lo demostraremos...

Ejemplo 7.2 (Cuadratura Gaussiana con $n = 3$)**.** Supongamos que queremos encontrar tres puntos de cuadratura, entonces usamos la ecuación:

$$\int_{-1}^{1} f(x)\mathrm{d}x \approx a_0 f(x_0) + a_1 f(x_1) + a_2 f(x_2)$$

Sospechamos que:

$$x_0 = -x_2$$
$$x_1 = 0$$
$$a_0 = a_2$$

Siguiendo los pasos del ejemplo 7.1, podemos resumir el sistema de ecuaciones generado a través del cuadro 7.1. Comenzamos con la ecuación que tiene $k = 2$:

k	$\int_{-1}^{1} x^k \mathrm{d}x$	$=$	$a_0 x_0^k + a_1 x_1^k + a_2 x_2^k$
0	2	$=$	$a_0 + a_1 + a_2$
1	0	$=$	$a_0 x_0 + a_1 x_1 + a_2 x_2$
2	$\frac{2}{3}$	$=$	$a_0 x_0^2 + a_1 x_1^2 + a_2 x_2^2$
3	0	$=$	$a_0 x_0^3 + a_1 x_1^3 + a_2 x_2^3$
4	$\frac{2}{5}$	$=$	$a_0 x_0^4 + a_1 x_1^4 + a_2 x_2^4$
5	0	$=$	$a_0 x_0^5 + a_1 x_1^5 + a_2 x_2^5$

Cuadro 7.1 – Comprobamos nuestras sospechas.

$$\frac{2}{3} = 2a_0 x_2^2$$
$$x_2^2 = \frac{1}{3a_0} \tag{7.7}$$

Continuamos con $k = 4$:

$$\frac{2}{5} = 2a_0 x_2^4$$
$$a_0 = \frac{5}{9}$$

Luego, reemplazamos a_0 en (7.7):

$$x_2^2 = \frac{1}{3 \cdot \frac{5}{9}}$$
$$x_2 = \sqrt{\frac{3}{5}}$$

Además:

$$a_1 = 2 - 2a_0$$
$$= \frac{13}{9}$$

Luego, sólo basta con reemplazar en el sistema de ecuaciones para comprobar que esto se cumpla.

7.1. Teoría de cuadraturas gaussianas

Muy bonito todo lo anterior... pero no es una técnica particularmente elegante, y no da luces sobre el comportamiento de las reglas de cuadratura resultantes. Para tratar ese tema, se requiere un desvío. Lo siguiente es básicamente de Levy [1, capítulos 4 y 6], sazonado con un poco de Treil [2, capítulo 5].

7.1.1. Polinomios ortogonales

Para simplificar notación, llamaremos Π_n al conjunto de polinomios con coeficientes reales de grado menor o igual a n. Notamos que Π_n es un espacio vectorial de dimensión $n+1$, y que Π_m es un subespacio de Π_n si $m < n$.

Definición 7.1. Consideremos un intervalo $[a,b]$, y una función peso $w\colon [a,b] \to \mathbb{R}$, continua y positiva. Dadas dos funciones f y g, continuas sobre el intervalo, definimos su *producto interno* (sobre $[a,b]$ respecto de w) por:

$$\langle f, g \rangle_w = \int_a^b w(x) f(x) g(x) \mathrm{d}x$$

Definimos la *norma* (sobre $[a,b]$ respecto de w) por:

$$\|f\|_w = \sqrt{\langle f, f \rangle_w}$$

Decimos que f y g son *ortogonales* (sobre $[a,b]$ con peso w) si $\langle f, g \rangle_w = 0$. Decimos que f está *normalizado* (sobre $[a,b]$ con peso w) si $\|f\|_w = 1$

Comúnmente omitimos el subíndice, la función peso se subentiende.
Algunas propiedades simples son las que definen productos internos en espacios vectoriales:

Teorema 7.1. *El producto interno cumple con:*

Simetría: $\langle f, g\rangle_w = \langle g, f\rangle_w$

Linearidad en el primer argumento: *Para todo* $\alpha, \beta \in \mathbb{R}$ *es* $\langle \alpha f + \beta g, h\rangle_w = \alpha\langle f, h\rangle_w + \beta\langle g, h\rangle_w$

No negatividad: $\|f\|_w \geq 0$,

No degenerado: $\|f\|_w = 0$, *si y solo si* $f = 0$

La demostración es simple, queda de ejercicio. En el caso de vectores sobre los complejos la propiedad de simetría se expresa como:

$$\langle f, g\rangle_w = \overline{\langle g, f\rangle_w}$$

O sea, es el complejo conjugado.

El teorema siguiente relaciona normas con productos internos.

Teorema 7.2 (Desigualdad de Cauchy-Schwartz)**.**

$$|\langle \mathbf{x}, \mathbf{y}\rangle| \leq \|\mathbf{x}\| \cdot \|\mathbf{y}\|$$

Demostración. La demostración en sí es simple, el razonamiento previo es para justificar el t misterioso empleado en ella.

Si $\mathbf{x}$ o $\mathbf{y}$ son cero, el resultado es trivial. Supongamos entonces $\mathbf{y} \neq 0$. Por las propiedades del producto interno, para todo escalar t:

$$\begin{aligned} 0 &\leq \|\mathbf{x} - t\mathbf{y}\|^2 \\ &= \langle \mathbf{x} - t\mathbf{y}, \mathbf{x} - t\mathbf{y}\rangle \\ &= \langle \mathbf{x}, \mathbf{x} - t\mathbf{y}\rangle - t\langle \mathbf{y}, \mathbf{x} - t\mathbf{y}\rangle \\ &= \|\mathbf{x}\|^2 - 2t\langle \mathbf{x}, \mathbf{y}\rangle + t^2\|\mathbf{y}\|^2 \end{aligned}$$

Se obtiene lo prometido al substituir el valor de t que minimiza la expresión indicada:

$$t = \frac{\langle \mathbf{x}, \mathbf{y}\rangle}{\|\mathbf{y}\|^2}$$

□

Tenemos el importante corolario:

Corolario 7.3 (Desigualdad triangular)**.** $\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$

Demostración.

$$\begin{aligned} \|\mathbf{x} + \mathbf{y}\|^2 &= \langle \mathbf{x} + \mathbf{y}, \mathbf{x} + \mathbf{y}\rangle \\ &= \|\mathbf{x}\|^2 + 2|\langle \mathbf{x}, \mathbf{y}\rangle| + \|\mathbf{y}\|^2 \\ &\leq \|\mathbf{x}\|^2 + 2\|\mathbf{x}\| \cdot \|\mathbf{y}\| + \|\mathbf{x}\|^2 \\ &= (\|\mathbf{x}\| + \|\mathbf{y}\|)^2 \end{aligned}$$

□

Definición 7.2. Sea w un peso sobre $[a, b]$. Diremos que la secuencia de polinomios $p_n(x)$ con $\deg(p_n) = n$ son *w-ortogonales* (o simplemente *ortogonales*, si w se subentiende) si $\langle p_i, p_j\rangle_w = 0$ para todo $i \neq j$. Los llamaremos *w-ortonormales* (o simplemente *ortonormales*) si además $\|p_i\|_w = 1$.

Dado un conjunto de vectores linealmente independientes (en nuestro caso, $\{1, x, x^2, \ldots\}$), el proceso de Gram-Schmidt (ver cualquier texto de álgebra lineal, recomendamos el de Treil [2]) permite construir un conjunto ortogonal.

Teorema 7.4. *Sea $p_n(x)$ un polinomio ortogonal de grado n sobre $[a,b]$ con peso w. Entonces p_n tiene n ceros reales simples en $[a,b]$.*

Demostración. Sean $x_1, \ldots, x_r$ los ceros de p_n en $[a,b]$, y consideremos:

$$q(x) = (x - x_1)(x - x_2)\cdots(x - x_r)$$

Claramente $\deg(q) = r \leq n$. En $[a,b]$, $p_n(x)q(x)$ tiene un único signo, por lo que:

$$\int_a^b w(x)p_n(x)q(x)\mathrm{d}x \neq 0$$

Pero p_n es ortogonal a todo Π_{n-1}, por lo que $\deg(q) = n$.

Supongamos que x_1 es un cero múltiple de p_n, y analicemos:

$$p_n(x) = (x - x_1)^2 g(x)$$

O sea:

$$p_n(x)g(x) = (x - x_1)^2 g^2(x) = \left(\frac{p_n(x)}{(x - x_1)}\right)^2 \geq 0$$

Por tanto:

$$\int_a^b w(x)p_n(x)g(x)\mathrm{d}x > 0$$

Esto se contradice con $g \in \Pi_{n-2}$, con los que p_n es ortogonal. □

7.1.2. Cuadratura de Gauß

Buscamos reglas de cuadratura de la forma:

$$\int_a^b w(x)f(x)\mathrm{d}x = \sum_{0\leq i\leq n} A_i f(x_i) \tag{7.8}$$

La ecuación (7.8) es exacta para $f \in \Pi_n$ si y solo si (esto es básicamente por la forma de Lagrange del polinomio interpolador):

$$A_i = \int_a^b w(x) \prod_{\substack{0\leq j\leq n \\ j\neq i}} \frac{x - x_j}{x_i - x_j}\mathrm{d}x \tag{7.9}$$

En (7.8) tenemos $2n+2$ grados de libertad, ($x_0, \ldots, x_n$ y $A_0, \ldots, A_n$), aspiramos a una regla que sea exacta en Π_{2n+1}.

Teorema 7.5. *Sea q un polinomio de grado $n+1$, ortogonal a Π_n, o sea para todo $p \in \Pi_n$:*

$$\int_a^b w(x)p(x)q(x)\mathrm{d}x = 0$$

Si x_i para $0 \leq i \leq n$ son los ceros de q, entonces la regla de cuadratura (7.8) con los coeficientes (7.9) es exacta para $f \in \Pi_{2n+1}$.

Demostración. Sea $f \in \Pi_{2n+1}$, escribamos por el algoritmo de división:

$$f(x) = q(x)p(x) + r(x)$$

Acá $\deg(r) \leq n$, con lo que notamos que $p, r \in \Pi_n$. En los ceros de q tenemos:

$$f(x_i) = r(x_i)$$

Por lo tanto:

$$\begin{aligned}\int_a^b w(x)f(x)\mathrm{d}x &= \int_a^b w(x)(q(x)p(x) + r(x))\mathrm{d}x \\ &= \int_a^b w(x)r(x)\mathrm{d}x \\ &= \sum_{0\leq i\leq n} A_i r(x_i) \\ &= \sum_{0\leq i\leq n} A_i f(x_i)\end{aligned}$$

y la regla es exacta para f. □

Otro punto interesante es el siguiente:

Lema 7.6. *En una regla de cuadratura gaussiana, los coeficientes son positivos y su suma es:*

$$\sum_{0\leq i\leq n} A_i = \int_a^b w(x)\mathrm{d}x$$

Demostración. Fijemos n, y sea $q \in \Pi_{n+1}$ w-ortogonal a Π_n, donde $q(x_i) = 0$ en los puntos de cuadratura $\{x_i\}_{0\leq i\leq n}$:

$$\int_a^b w(x)f(x) \approx \sum_{0\leq i\leq n} A_i f(x_i)$$

Fijemos $0 \leq j \leq n$, y sea $p \in \Pi_n$ dado por:

$$p(x) = \frac{q(x)}{x - x_j}$$

Siendo x_j un cero de q, $\deg(p) < n$, y $\deg(p^2) < 2n$, así que la siguiente es exacta:

$$0 < \int_a^b w(x)p^2(x)\mathrm{d}x = \sum_{0\leq i\leq n} A_i p^2(x_i) = A_j p^2(x_j)$$

Concluimos que $A_j > 0$.

Por el otro lado, la regla de cuadratura es exacta para $1 \in \Pi_{2n+1}$:

$$\int_a^b w(x)\mathrm{d}x = \sum_{0\leq i\leq n} A_i$$

□

Ejercicios

1. Demostrar el teorema 7.1.
2. Demostrar que los coeficientes A_i están dados por la ecuación (7.9).

Bibliografía

[1] Doron Levy: *Introduction to numerical analysis*. http://www.math.umd.edu/~dlevy/books/na.pdf, September 2010.

[2] Sergei Treil: *Linear algebra done wrong*. https://www.math.brown.edu/~treil/papers/LADW/book.pdf, September 2015.

Parte II

Algoritmos Combinatorios

Clase 8

Matrimonios Estables

Nuestro interés principal es algoritmos que operan con objetos discretos, de los que estudia la combinatoria. Un primer ejemplo muestra que situaciones a primera vista simples pueden tener profundidades insospechadas.

8.1. El problema

El problema a resolver es el de matrimonios estables (*stable marriage problem*): Dados un conjunto de mujeres $\mathscr{X}$ y otro de hombres $\mathscr{Y}$, donde $|\mathscr{X}| = |\mathscr{Y}|$, cada mujer define un orden de deseabilidad para los hombres, y similarmente los hombres con las mujeres, donde suponemos que no hay empates.

Definición 8.1. Un conjunto de matrimonios se dice *inestable* si incluye parejas uv y xy, tales que u prefiere a y frente a v y x prefiere a v frente a y.

Si el conjunto es inestable, u y x pueden mejorar sus elecciones divorciándose y volviéndose a casar. Buscamos matrimonios estables. Este problema y variantes aparece en un amplio rango de situaciones, resumidas por Iwama y Miyazaki [2].

Hallar un conjunto estable de matrimonios parece ser simplemente "cumpla las preferencias", pero reflexión más profunda muestra que ni siquiera es obvio que tal conjunto existe.

Una posibilidad es asignar parejas al azar, y en caso de que cambiando parejas se pueda mejorar, permitir divorcios y nuevos matrimonios. Sin embargo, el siguiente ejemplo (adaptado de Knuth [3]) muestra que esto no siempre termina. Considere las preferencias del cuadro 8.1, donde simplemente damos las mujeres en orden de preferencia para cada hombre y viceversa. Una secuencia de divor-

1: 2 1 3	1: 1 3 2
2: lista arbitraria	2: 3 1 2
3: 1 2 3	3: lista arbitraria
(a) Hombres	(b) Mujeres

Cuadro 8.1 – Contraejemplo para divorcios y matrimonios

cios y matrimonios entra en un ciclo; pero hay soluciones estables, como $((x_1, y_2), (x_2, y_3), (x_3, y_1))$ o $((x_1, y_1), (x_2, y_3), (x_3, y_2))$.

Una solución puede hallarse mediante un algoritmo bastante sencillo, y este algoritmo muestra incidentalmente que los matrimonios estables siempre existen. El algoritmo fue discutido formalmente por primera vez por Gale y Shapley [1].

Teorema 8.1. *Siempre hay un conjunto de matrimonios estable.*

Demostración. Consideremos el modelo tradicional, en el cual los hombres proponen matrimonio a las mujeres, y éstas aceptan o no. Efectuamos varias rondas, en cada ronda los hombres sin pareja proponen matrimonio a la mujer más alta en su preferencia, cada mujer elige entre las propuestas que recibe al hombre más alto en sus preferencias y se compromete provisoriamente. Si una mujer provisoriamente comprometida recibe una mejor oferta, disuelve el compromiso y se compromete con el nuevo pretendiente.

Note que una vez que una mujer recibe una propuesta, nunca más queda libre (puede cambiar de novio, claro). En cada ronda un hombre propone a mujeres hasta hallar una que lo acepte, el número de mujeres no comprometidas disminuye. Los hombres pueden proponer siempre en orden de preferencia decreciente (las mujeres que ya lo rechazaron solo pueden mejorar su pretendiente, no lo aceptarán después). El número total de propuestas es a lo más n^2-2n+2 propuestas, si $|\mathscr{X}| = |\mathscr{Y}| = n$. Una vez que todas las mujeres han recibido propuestas el "pololeo" se declara terminado y los compromisos se formalizan.

El resultado es estable, cosa que demostramos por contradicción. Supongamos que x tiene a y de pareja, pero prefiere a y'. Entonces x propuso matrimonio a y' antes que a y, y fue rechazado por alguien a quien y' prefiere a x. Si y' cambió su compromiso en el intertanto, fue por alguien a quien prefiere aún más que a x. O sea, y' prefiere a su marido, no hay inestabilidad. □

Podemos extender trivialmente al caso $|\mathscr{X}| \neq |\mathscr{Y}|$, simplemente sobrarán hombres o mujeres que no encuentran pareja, y por el mismo razonamiento los matrimonios acordados son estables.

Una pregunta obvia es si la solución es única, y la relación entre ésta y la que da el algoritmo simétrico en que proponen las mujeres. Resulta que pueden haber muchas soluciones, como demuestra Knuth [3] con el ejemplo del cuadro 8.2. Vemos que el hombre 1 puede formar parejas estables con

1:	1	2
2:	2	1
⋮	⋮	⋮
$n-1$:	$n-1$	n
n:	n	$n-1$

(a) Hombres

1:	⋯	1
2:	⋯	2
⋮		⋮
$n-1$:	⋯	$n-1$
n:	⋯	n

(b) Mujeres

Cuadro 8.2 – Preferencias para muchas soluciones, n par

las mujeres 1 o 2, mientras el hombre 2 queda con 2 o 1; el hombre 3 con 3 o 4, dejando la otra para 4; y así sucesivamente. Ambas posibilidades son estables, si dos hombres se conforman con sus segundas opciones, son la última preferencia para su primera opción y ella nunca lo preferirá. Esto da $2^{n/2}$ soluciones posibles.

Incidentalmente, si se permiten preferencias incompletas ("prefiero muerto que casado con…"), puede no haber solución estable. Nuevamente Knuth [3] plantea un ejemplo. En el cuadro 8.3 la única posibilidad es $((x_1, y_1), (x_2, y_2), (x_3, y_3))$, pero ésta es inestable por x_2 e y_3.

En realidad, se da:

1: 1	
2: 3	2 1
3: 3	1

(a) Hombres

1: 3	1 2
2: 2	1 3
3: 1	2 3

(b) Mujeres

Cuadro 8.3 – Preferencias incompletas

Teorema 8.2. *La solución dada por el algoritmo de Gale-Shapley es la mejor posible para los hombres.*

Demostración. Llame a una mujer *posible* para x si hay una solución estable que la da como pareja para x. Demostraremos el resultado por inducción. Supongamos que en un cierto punto del algoritmo ningún hombre ha sido rechazado por una mujer posible. Al inicio esto se cumple vacuamente. Suponga que en este punto y, habiendo recibido una propuesta mejor, rechaza a x. Debemos demostrar que y es imposible para x. Si y elige a x', es porque prefiere a x' sobre x; y x' se propuso a y porque todas sus mejores opciones lo rechazaron. Por inducción, x' es imposible para esas otras opciones. Si se casara x con y, x' deberá conformarse con y', que considera menos deseable. Pero esta configuración es inestable, x' e y estarían dispuestos a intercambiar parejas. En resumen, x es imposible para y. □

Solo si la solución es única los resultados de propuestas de hombres y propuestas de mujeres coinciden.

8.2. Postulación a carreras

Una extensión es considerar estudiantes $\mathscr{A}$ que postulan a universidades $\mathscr{U}$, donde la universidad $u \in \mathscr{U}$ ofrece q_u vacantes. Nuevamente, cada estudiante tiene una lista de prioridades de las universidades y cada universidad una lista de preferencias de postulantes. Todos postulan a su universidad preferida entre las que aún no lo han rechazado, y la universidad con q cupos elige los q mejores entre los que tiene en la lista actualmente y los nuevos llegados (posiblemente rechazando a quienes no cumplen requisitos mínimos, con lo que la lista podría tener menos de q postulantes). El proceso termina cuando todos los estudiantes o están en una lista de espera o han sido rechazados por todas las universidades a las que pueden postular. En forma similar al teorema 8.1 se demuestra que el resultado es estable (ningún estudiante se cambiaría de universidad con otro), como en el teorema 8.2 los postulantes obtienen sus mejores cupos posibles.

Ejercicios

1. Encuentre las soluciones que entrega el algoritmo orientado a hombres y a mujeres para las preferencias del cuadro 8.4.

2. Acote el número de propuestas de matrimonio, como menciona la demostración del teorema 8.2.

3. Demuestre que a lo más un hombre recibe su última elección con el algoritmo dado. En consecuencia, si hay una asignación estable en la cual varios hombres se deben conformar con sus últimas preferencias, hay varias soluciones.

4. El algoritmo esbozado en el teorema 8.1 no especifica el orden en que los hombres se proponen. Demuestre que cualquiera sea este orden, la asignación resultante es la misma.

1:	5	7	1	2	6	8	4	3
2:	2	3	7	5	4	1	8	6
3:	8	5	1	4	6	2	3	7
4:	3	2	7	4	1	6	8	4
5:	7	2	5	1	3	6	8	4
6:	1	6	7	5	8	4	2	3
7:	2	5	7	6	3	4	8	1
8:	3	8	4	5	7	2	6	1

(a) Hombres

1:	5	3	7	6	1	2	8	4
2:	8	6	3	5	7	2	1	4
3:	1	5	6	2	4	8	7	3
4:	8	7	3	2	4	1	5	6
5:	6	4	7	3	8	1	2	5
6:	2	8	5	3	4	6	7	1
7:	7	5	2	1	8	6	4	3
8:	7	4	1	5	2	3	6	8

(b) Mujeres

Cuadro 8.4 – Preferencias para ejercicio

5. Demuestre que el algoritmo como esbozado en teorema 8.1 a cada mujer le asigna la peor de sus preferencias entre todas las soluciones estables.

6. Demuestre que el algoritmo de llenado de cupos en universidades cumple lo enunciado.

Bibliografía

[1] David Gale and Lloyd S. Shapley: *College admissions and the stability of marriage.* American Mathematical Monthly, 69(1):9–15, January 1962.

[2] Kazuo Iwama and Shuichi Miyazaki: *The stable marriage problem and its variants.* In *International Conference on Informatics Education and Research for Knowledge-Circulating Society*, pages 131–136, Kyoto, Japan, January 2008.

[3] Donald E. Knuth: *Stable Marriage and Its Relation to Other Combinatorial Problems*, volume 10 of *CRM Proceedings and Lecture Notes.* American Mathematical Society, annotated edition, 1996.

Clase 9

Optimización Combinatoria

Un área de interés es optimización combinatoria. Buscamos una configuración óptima de algún objeto discreto. Un ejemplo es un problema de programación de tareas (*scheduling* en inglés)

Ejemplo 9.1. Supongamos que usted está a cargo de programar observaciones en ALMA. Para justificar el gasto de este enorme recurso, su misión es programar el máximo número de observaciones. Las observaciones tienen instante de inicio y duración, y no pueden traslaparse.

Formalmente, el proyecto i tiene duración el intervalo abierto $[s_i, s_i + \ell_i)$. Se pide elegir el subconjunto $\Pi \subseteq P$ tales que los elementos de Π sean disjuntos y el número de elementos de Π sea máximo (figura 9.1). ¿Cómo hacerlo? (Básicamente, estamos suponiendo que el observatorio se

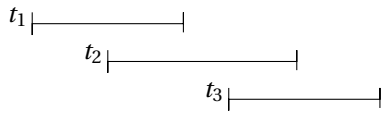


Figura 9.1 – Intervalos de tiempo que dura cada proyecto/tarea.

arrienda por proyecto y no por tiempo). Proponemos algunas sugerencias:

Sugerencia 1: Repetidamente elegir la tarea más corta que no entra en conflicto. La figura 9.2 mues-

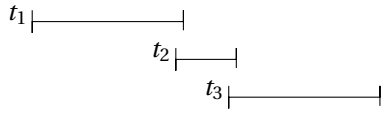


Figura 9.2 – Empezando con t_2, efectuamos 1 tarea. Con t_1 completamos 2.

tra un contraejemplo, por lo tanto, esta sugerencia no siempre da un óptimo.

Sugerencia 2: Elegir la tarea con inicio más temprano que no crea conflicto. La figura 9.3 muestra un contraejemplo, por lo tanto, esta sugerencia no siempre da un óptimo.

Sugerencia 3: Marcar cada proyecto con el número de proyectos con que entra en conflicto, programar en orden creciente de conflictos. La figura 9.4 muestra un contraejemplo, por lo tanto,

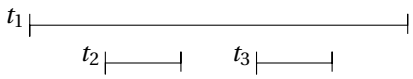


Figura 9.3 – Empezando con t_1, completa 1 tarea. Con t_2 y t_3 hacemos 2.

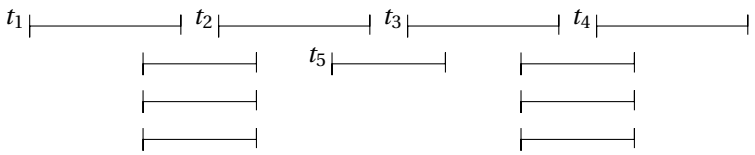


Figura 9.4 – Elije t_5, t_1 y t_4, un total de 3 tareas; pero t_1, t_2, t_3, t_4 son 4. Me echaron a perder el día.

esta sugerencia no siempre da un óptimo.

¿Estructura común de las propuestas?

- Eliga elementos sucesivamente hasta que no queden opciones viables.
- Entre las opciones visibles en cada paso, elija la que minimiza (maximiza) alguna propiedad.

De eso es lo que precisamente tratan los *Greedy Algorithms* (algoritmos voraces, aunque mejor traducción sería "ávido"). Es importante destacar que estos no siempre encuentran una solución óptima del problema, pero sí pueden ofrecer una aproximación bastante buena.

9.1. Comprobar que un algoritmo da un óptimo

Volvamos al ejemplo 9.1. En ese caso, un criterio que a usar para encontrar una solución óptima consta en elegir la tarea que *finaliza* más temprano y no entra en conflictos con las ya elegidas. Para el proyecto i, el instante de *fin* es

$$f_i = s_i + \ell_i$$

¿Es óptima la programación que esto construye?

Una forma de comprobar que un algoritmo voraz retorna un óptimo es demostrar lo siguiente:

Elección Voraz (*greedy choice*): Para toda instancia P, hay una solución óptima que incluye el primer elemento $\hat{p}$ elegido.

Estructura Inductiva: Dada la elección voraz $\hat{p}$, queda un subproblema menor P' tal que si Π' es solución viable de P', $\{\hat{p}\} \cup \Pi'$ es solución viable de P (P' no tiene "restricciones externas").

Subestructura Óptima: Si P' queda de P al sacar $\hat{p}$, y Π' es óptima para P', $\Pi' \cup \{\hat{p}\}$ es óptima para P.

Con estos tres podemos demostrar que la secuencia de elecciones de $\hat{p}$ da una solución óptima por inducción sobre los pasos. El esqueleto de la demostración es el siguiente:

Teorema 9.1. *Si un algoritmo cumple con Elección Voraz, Estructura Inductiva y Subestructura óptima, entrega una solución óptima al problema.*

Demostración. Por inducción sobre el tamaño del problema.

Base: Si $|P|$, por Elección Voraz se elige el único p posible, que claramente es óptimo.

Inducción: Supongamos que el algoritmo voraz da una solución óptima para todos los problemas hasta tamaño k, y consideremos la instancia P de tamaño $k+1$. Elegimos $\widehat{p}$ por el criterio voraz, sabemos que hay una solución óptima que incluye $\widehat{p}$ por Elección Voraz. Sea P' el problema que resulta al eliminar $\widehat{p}$ de P, junto con sus dependencias. Es claro que $|P'| \leq k$, sea Π' la solución dada por el algoritmo voraz a P'. Por inducción, Π' es óptima para P'. Por Estructura Inductiva, $\Pi' \cup \{\widehat{p}\}$ es viable para P, por Subestructura Óptima, $\Pi' \cup \{\widehat{p}\}$ es óptima para P.

□

9.1.1. Demostrando un algoritmo voraz

Volvamos al ejemplo 9.1. Si quisiéramos demostrar que la sugerencia entrega un óptimo solo tenemos que seguir los pasos anteriores:

Elección Voraz: Sea $\widehat{p}$ la primera tarea elegida y Π^* una solución óptima para P. Si $\widehat{p} \in \Pi^*$, estamos listos. En caso contrario, sea Π' la solución obtenida reemplazando el proyecto más temprano de Π^* por $\widehat{p}$. Esto no produce nuevos conflictos, ya que la primera tarea de P^* no termina antes de $\widehat{p}$ por cómo fue elegida ésta, y $|\Pi^*| = |\Pi'|$, ambos son óptimos.

Estructura Inductiva: El elegir $\widehat{p}$ nos deja un problema P' sin restricciones externas (elegir $\widehat{p}$ elimina de consideración las tareas que entran en conflicto con $\widehat{p}$, pero nada más).

Subestructura Óptima: Si P' queda después de elegir $\widehat{p}$, y Π' es óptima para P', entonces $\Pi' \cup \{\widehat{p}\}$ es óptima para P.

Demostración. Sea Π' como dado. Entonces $\Pi' \cup \{\widehat{p}\}$ es viable para P, y $|\Pi' \cup \{\widehat{p}\}| = |\Pi'| + 1$. Sea Π^* una solución óptima para P que contiene $\widehat{p}$. Entonces $\Pi^* \setminus \{\widehat{p}\}$ es una solución óptima para P'. Pero entonces:

$$\begin{aligned} |\Pi'| &= |\Pi^* \setminus \{\widehat{p}\}| \\ &= |\Pi^*| - 1 \end{aligned}$$

O sea:

$$|\Pi' \cup \{\widehat{p}\}| = |\Pi^*|$$

y $\Pi' \cup \{\widehat{p}\}$ es óptima. □

Clase 10

Problema de Optimización

10.1. Problema de Tareas

Teorema 10.1. *Para el problema de programación de tareas, la estrategia de elegir en cada paso la tarea sin conflicto con fin más temprano entrega una solución óptima.*

Demostración. Por inducción sobre $|P|$, el número de tareas.

Base: Si hay una única tarea, la estrategia la programa. Esto es óptimo.

Inducción: Supongamos que obtiene una solución óptima para a lo más k tareas. Sea P una instancia con $|P| = k+1$. Elegimos $\hat{p}$ según criterio voraz. Por Elección Voraz hay solución óptima que lo incluye, queda P', $|P'| \leq k$.

Por inducción, obtengo una solución óptima Π' de P'. Combinando $\Pi' \cup \{\hat{p}\}$ por Estructura Inductiva una solución viable para P' es viables con $\hat{p}$. Esta es una solución óptima para P por Subestructura Óptima.

□

10.2. Knapsack (mochila)

Hay una mochila de capacidad M, y un conjunto de n tipos de ítem, del ítem tipo i hay disponible p_i en total, de valor v_i. Se pueden incluir fracciones de ítem (es café, azúcar, arroz, ...)

Estrategia:

- Ordenar los ítem por

  $$\frac{v_i}{p_i}$$

  decreciente.

- Echar en la mochila sucesivamente todo lo que se pueda del ítem i, en el orden anterior.

Mochila Discreta: El ítem i se agrega completo o no (no fracciones). En este caso la estrategia voraz *no* da óptimo.

EVQA: Contraejemplo.

10.3. Minimal Spanning Tree

Dado un grafo $G = (V, E)$, con arcos rotulados $c\colon E \to \mathbb{R}^+$, se busca el árbol recubridor (o sea, el que une todos los vértices) de costo mínimo (suma de los c sobre sus arcos). En inglés se le conoce como *minimal spanning tree*, y se abrevia MST. Para el grafo $G = (V, E)$ usamos la notación $V = G_V$, $E = G_E$. Dos algoritmos alternativos para resolver este problema son el algoritmo de Prim (algoritmo 10.1, en realidad de Jarník [2], redescubierto por Prim [4] y Dijkstra [1]) y el algoritmo de

Algoritmo 10.1: Algoritmo de Prim

```
procedure prim(G)
    Ordenar G_E en orden de c(uv) creciente

    Elija un vértice u ∈ G_V
    T ← ({u}, ∅)
    for uv ∈ G_E tal que u ∈ T_V, v ∉ T_V do
        T ← (T_V ∪ {v}, T_E ∪ {uv})
    end
    return T
```

Kruskal [3] (algoritmo 10.2). Ambos son algoritmos voraces, como puede apreciarse, aunque usan

Algoritmo 10.2: Algoritmo de Kruskal

```
procedure prim(G)
    Ordenar G_E en orden de c(e) creciente

    T ← (∅, ∅)
    for uv ∈ G_E do
        if uv no forma ciclo en T then
            T ← (T_V ∪ {u, v}, T_E ∪ {uv})
        end
    end
    return T
```

criterios diferentes.

La demostración de ambos se basa en:

Proposición 10.1. *Sea $G = V, E)$ un grafo como indicado, y sea V_1, V_2 una partición de V. Entonces el árbol recubridor mínimo de G se divide en árboles $T_1 = (V_1, E_1$ y $T_2 = (V_2, E_2$, y si el arco $v_1 v_2$ tiene costo mínimo entre los arcos entre V_1 y V_2, hay un árbol recubridor mínimo de G que incluye $v_1 v_2$.*

10.4. Otras técnicas para demostrar correctitud

La técnica expuesta para demostrar que el algoritmo voraz entrega un óptimo (basada en las tres propiedades) es bastante general, pero no siempre es aplicable ni la manera más natural de enfrentar el problema.

10.4.1. Demostración por contradicción

Notar las diferencias entre este caso y la demostración por contradicción para demostrar que se cumple la propiedad de elección voraz.

Consideremos el problema de ordenar archivos en forma óptima en una cinta, donde el largo del archivo i es l_i. Los usuarios solicitan el archivo i con probabilidad p_i, y el costo de extraer un archivo (que llamaremos L_i) es proporcional a la suma de los largos de los archivos que lo preceden y de ese mismo. Interesa minimizar el valor esperado del tiempo para extraer archivos, determinando el orden de los archivos en la cinta:

$$T = \sum_i p_i L_i$$

Usamos el algoritmo voraz de ordenar los archivos en la cinta en orden creciente de l_i / p_i. Este orden se puede determinar en tiempo $O(n \log n)$, el costo de ordenar domina.

Demostramos que esto es óptimo por contradicción. Supongamos que un orden diferente da una solución mejor. Eso quiere decir que hay archivos vecinos (a, b) tales que:

$$\frac{l_a}{p_a} > \frac{l_b}{p_b}$$

pero a se almacena antes de b. Demostraremos que intercambiándolos mejora T, este orden no puede ser óptimo.

Como a y b son vecinos, intercambiarlos no afecta el tiempo de extracción de ningún otro archivo, por lo que el cambio en T es:

$$\begin{aligned} p_a l_a + p_b(l_a + l_b) - (p_b l_b + p_a(l_a + l_b)) &= p_b l_a - p_a l_b \\ &= p_a p_b \left(\frac{l_a}{p_a} - \frac{l_b}{p_b} \right) \\ &> 0 \end{aligned}$$

Al intercambiarlos, disminuye el tiempo promedio. Esto contradice el que haya sido óptimo antes del cambio.

10.4.2. Usando un invariante

Un caso claro donde el esquema no funciona es el problema de hallar los caminos más cortos a todos los vértices de un grafo desde un vértice dado. Nuestro problema es un grafo dirigido $G = (V, E)$ con arcos rotulados $w(u, v) \colon V \times V \to \mathbb{R}^+$, y el costo del camino p es:

$$w(p) = \sum_{(u,v) \in p} w(u, v)$$

Dado un vértice origen s, nos interesan los costos mínimos de los caminos a cada vértice $v \in V \setminus \{s\}$. Llamaremos $\delta(v)$ al costo de tal camino.

El algoritmo de Dijkstra es un algoritmo voraz que resuelve el problema si no hay arcos de peso negativo. Funciona como sigue: Mantiene una partición de los vértices, S y $V \setminus S$. En cada instante,

S es el conjunto de vértices a los cuales ya se conocen los caminos más cortos. Inicialmente $S = \varnothing$. Para cada vértice $v \in V$ tenemos una variable $d[v]$ que es el largo del mejor camino desde s a v que se ha hallado. Los vértices se ubican en una cola de prioridad Q, con prioridad $d[v]$. El algoritmo 10.3 describe esto informalmente. Demostramos que el algoritmo de Dijkstra es correcto demostrando

Algoritmo 10.3: Algoritmo de Dijkstra

```
for v ∈ V do
    d[v] ← ∞
end
d[s] ← 0
for v in V do
    Inserte v en Q con prioridad d[v]
end
while Q no vacía do
    u ← DeleteMin(Q)
    S ← S ∪ {u}
    for v vecino de u do
        if d[v] > d[u] + w(u, v) then
            d[v] ← d[u] + w(u, v)
            Actualizar la clave de v en Q
        end
    end
end
```

por inducción sobre S que se cumple el invariante:

$$\forall v \in S, d[v] = \delta[v] \tag{10.1}$$

Base: Inicialmente $S = \varnothing$, y el invariante vale trivialmente.

Inducción: Supongamos que cuando $|S| = k$ el invariante se cumple. Sea v el siguiente vértice extraído de Q, (y colocado en S), y sea p un camino de s a v de costo $d[v]$ (claramente existe, y el algoritmo en un uso real registrará tal camino con el vértice). Sea u el vértice inmediatamente predecesor de v en p. Entonces $u \in S$, y $d[u] = \delta[u]$ por inducción.

Demostraremos por contradicción que p es un camino de costo mínimo de s a v. Supongamos que hay un camino p^* de s a v tal que $w(p^*) = \delta(v) < w(p)$. Como p^* conecta al vértice $s \in S$ con el vértice $v \in V \setminus S$, debe haber un primer arco $ab \in p^*$ con $a \in S$ y $b \in V \setminus S$. Podemos dividir el camino en p_1, p_2, con p_1 de s a a y p_2 de b a v. Por inducción, $d[a] = \delta[a]$. Como p^* es un camino más corto, p_1, b es un camino más corto de s a b (si hubiera uno más corto, p^* no sería óptimo). Después de agregar a a S se consideró el arco ab, con lo que después de actualizarlo $d[b] = \delta[b]$. Como v se agregó a S mientras b estaba en Q, es $\delta[v] \leq d[b]$. Como los pesos son no negativos, $\delta[v] = w(p^*) \geq d[b]$. En conjunto con $d[v] \leq d[b]$ resulta $w(p^*) \geq d[v] = w(p)$, contradiciendo que $w(p^*) < w(p)$.

Ejercicios

1. Demuestre en detalle que el algoritmo voraz da una solución óptima al problema de la mochila, siguiendo el esquema planteado.

2. Demuestre la proposición 10.1.
3. Demuestre que el algoritmo de Prim da un árbol recubridor mínimo.
4. Demuestre que el algoritmo de Kruskal da un árbol recubridor mínimo.

Bibliografía

[1] Edsger W. Dijkstra: *A note on two problems in connexion with graphs.* Numerishe Mathematik, 1(1):269–271, December 1959.

[2] Vojtěch Jarník: *O jistém problému minimálním.* Práce Moravské Přírodovědecké Společnosti, 6(4):57–63, 1930.

[3] Joseph B. Kruskal: *On the shortest spanning subtree of a graph and the travelling salesman problem.* Proceedings of the American Mathematical Society, 7(1):48–50, February 1956.

[4] Robert C. Prim: *Shortest connection networks and some generalizations.* Bell System Technical Journal, 36(6):1389–1401, November 1957.

Clase 11

Código Huffman

El código Huffman es una aplicación muy importante de algoritmo voraz.
Ámbito: Compresión de datos.
Dado: Un texto, formado por caracteres. Buscamos codificarlo eficientemente. Si cada caracter se codifica en k bits (total 2^k caracteres posibles), de texto de largo n usa nk bits.

En texto, las frecuencias son *muy* desiguales. Moby Dick: 117 194 veces 'e', 640 veces 'z'. Nuestro principal objetivo es asignarle codificaciones más largas a los caracteres que menos se repiten y codificaciones más cortas a aquellos que se repiten más.
Cuidado:

$$a \mapsto 0$$
$$b \mapsto 1$$
$$c \mapsto 01$$

Bajo esta codificación podemos escribir:

$$ababc \rightsquigarrow 010101 \tag{11.1}$$

Pero también:

$$ccc \rightsquigarrow 010101 \tag{11.2}$$

¡Se produce ambigüedad entre (11.1) y (11.2)!

Condición suficiente para evitarlo: ningún código es prefijo de otro. Esto es importante porque hace eficiente el decodificar ("*prefix-free code*" o "*prefix code*"). Puede demostrarse (ver por ejemplo la discusión de compresión de Blelloch [1]) que si un código puede decodificarse en forma única, hay un código prefijo con códigos del mismo largo para cada símbolo.

11.0.0.1. Descripción del problema

Dada una secuencia T sobre $\Sigma = \{x_1, \ldots, x_n\}$, donde x_i aparece con frecuencia f_i, construir una función de codificación $C\colon \Sigma \to$ cadenas de bits, tal que C es un código prefijo y el número total de bits para representar T se minimiza.

Idea: Representar el código como árbol binario: cada arco se rotula con un bit 0 o 1 (digamos si va la izquierda lo rotulamos con 0, y si va a la derecha lo marcamos con 1)

Cada caracter rotula una de las hojas, el camino desde la raíz es el código de ese carácter (figura 11.1).

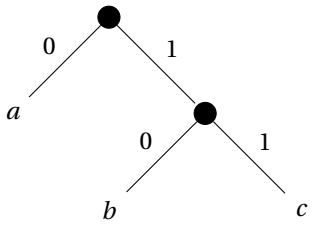


Figura 11.1 – Nodos en nivel más alto son los de mayor frecuencia

- Es un código prefijo (llegó a la hoja x_i, ya no sigue, camino desde la raíz a la hoja es único).
- En el código óptimo, todo nodo interno tiene dos hijos (podemos "saltarnos" un nodo interno con un único hijo para crear un código más compacto).

Definición 11.1. La *profundidad* de la hoja ℓ_i, anotada $d(\ell_i)$, es el largo del camino de la raíz a esa hoja. El caracter x_i queda codificado por $d(x_i)$ bits, el texto completo por:

$$\sum_i f_i d(x_i) \quad \texttt{bits}$$

Intuitivamente, buscamos letras poco frecuentes a altas profundidades, frecuentes a profundidades bajas. Para ello, armamos el árbol desde las hojas. Vamos uniendo subárboles hasta tener uno solo.

Sea $L = (\ell_1, \ldots, \ell_n)$ el conjunto de hojas para todos los caracteres, y sea f_i la frecuencia de la letra x_i. Hallar las dos letras de frecuencia mínima, digamos x_a y x_b con frecuencias f_a y f_b. Unir sus hojas en la hoja ℓ_{ab} con frecuencia $f_a + f_b$ dando un árbol R_{ab} (figura 11.2): Recursivamente resolver

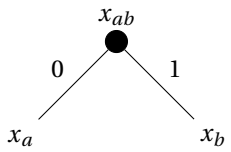


Figura 11.2 – El nodo x_{ab} es la unión entre x_a y x_b.

el problema con:

$$L = \{\ell_1, \ldots, \ell_n\} \setminus \{\ell_a, \ell_b\} \cup \{\ell_{ab}\} \tag{11.3}$$

y frecuencias ajustadas ($\ell_{ab} \rightsquigarrow f_a + f_b$)

Ejemplo 11.1. Considere el cuadro 11.1. El algoritmo de Huffman nos pide encontrar los símbolos de menor frecuencia y crear un sub-árbol con ellos. En el cuadro 11.1, se tiene que los símbolos *c* y *e* son los de menor frecuencia. Luego, formamos un sub-árbol con ellos (figura 11.3). De inmediato, agregamos este "nodo conjunto" al cuadro 11.1, cuya frecuencia es equivalente al peso del árbol de la figura 11.3 (suma de las frecuencias de *c* y *e*). El resultado se logra apreciar en el cuadro 11.2. Repetimos el proceso, es decir, escogemos dos símbolos del cuadro 11.2 que tienen menor frecuencia

Símbolo	Frecuencia
a	9
b	4
c	2
d	15
e	3
f	17

Cuadro 11.1 – Frecuencias de los símbolos a, b, c, d, e, f.

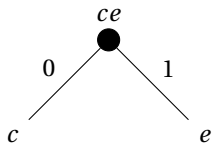


Figura 11.3 – Hojas son los símbolos que menos se repiten.

Símbolo	Frecuencia
a	9
b	4
d	15
f	17
ce	5

Cuadro 11.2 – Nodo conjunto ce con frecuencia la suma de las de c y e.

y creamos un nuevo sub-árbol. Estos símbolos son ce y b. Entonces, la figura 11.4 muestra el árbol resultante. Inmediatamente, reemplazamos los símbolos b y ce del cuadro 11.2 y lo reemplazamos con bce de frecuencia $f_{bce} = 9$. El resultado queda en el cuadro 11.3. Iteramos nuevamente. En el cuadro 11.3 se tiene que los dos símbolos con menor frecuencia son bce y a. Entonces, el árbol resultante queda representado por la figura 11.5. Luego, quitamos estos símbolos del cuadro 11.3 y los reemplazamos por $abce$. El resultado se puede apreciar mirando el cuadro 11.4. Continuamos iterando. Al observar el cuadro 11.4 vemos que los símbolos con menor frecuencia son d y f. Por lo tanto, tomamos estos dos símbolos y creamos un nuevo árbol que los tenga como hojas (figura 11.6). En seguida, sacamos esos símbolos y lo reemplazamos por df. El resultado de hacer esto se puede observar en el cuadro 11.5. Hacemos la última iteración, ya que sólo nos quedan dos símbolos. Tomamos los dos últimos símbolos y creamos el árbol final que se logra apreciar en la figura 11.7.

Demostración (algoritmo voraz). Como siempre, para demostrar que este algoritmo da un óptimo, lo hacemos vía demostrar que:

Elección voraz: Sea L la instancia original (o sea, el texto completo, con la frecuencia de cada símbolo respectivo), sean ℓ_a y ℓ_b las hojas menos frecuentes. Entonces hay un árbol óptimo que incluye R_{ab}.

Demostración. Sea R un árbol óptimo para L. Si R_{ab} es parte de R, salimos a carretear. Si el árbol R_{ab} *no* es parte de R, sean ℓ_x, ℓ_y dos hojas en R con padre común (hermanos), con $\delta = d(\ell_x) = d(\ell_y)$ máximo.

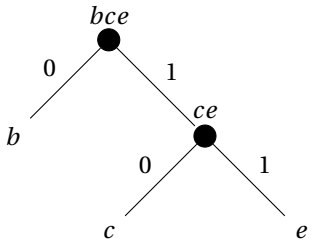


Figura 11.4 – Hojas son los símbolos que menos se repiten.

Símbolo	Frecuencia
a	9
d	15
f	17
bce	9

Cuadro 11.3 – Reemplazando los símbolos b y ce del cuadro 11.2.

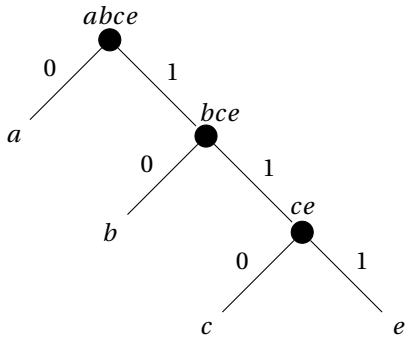


Figura 11.5 – Árbol con peso de $f_{bce} + f_a = 18$.

Claramente, a o b pueden coincidir con x o y. Consideraremos el caso en que son diferentes, la situación en que alguno coincide es similar.

Obtenga R^* intercambiando $x \leftrightarrow a$, $y \leftrightarrow b$, R^* contiene R_{ab}. Sea $B(R)$ el número de bits usados por el árbol R (importante, que en la demostración nos referiremos a d como el árbol original R).

En el árbol R^*; con $d()$ referenciando al árbol original R:

$$\begin{aligned} B(R^*) &= B(R) - (f_x + f_y)\delta - f_a d(\ell_a) - f_b d(\ell_b) + (f_a + f_b)\delta + f_x d(\ell_a) + f_y d(\ell_b) \\ &= B(R) - \underbrace{(f_x - f_a)}_{\geq 0}\underbrace{(\delta + d(\ell_a))}_{\geq 0} - \underbrace{(f_y - f_b)}_{\geq 0}\underbrace{(\delta + d(\ell_b))}_{\geq 0} \end{aligned}$$

Pero R es óptimo. Por lo tanto, una contradicción. □

□

Símbolo	Frecuencia
d	15
f	17
$abce$	18

Cuadro 11.4 – Reemplazando los símbolos bce y a del cuadro 11.3.

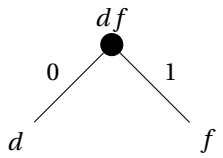


Figura 11.6 – Hojas son los símbolos de menor frecuencia del cuadro 11.4.

Símbolo	Frecuencia
d f	32
a b c e	18

Cuadro 11.5 – Reemplazando d y f del cuadro 11.4.

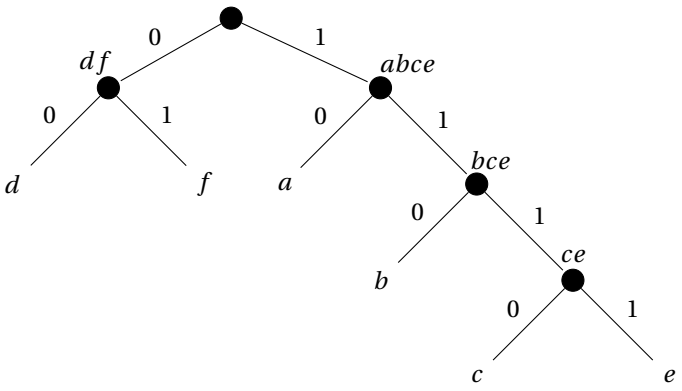


Figura 11.7 – Este árbol tiene un peso de $f_{bce} + f_a = 18$.

Bibliografía

[1] Guy E. Blelloch: *Introduction to data compression.* http://www.cs.cmu.edu/afs/cs/project/pscico-guyb/realworld/www January 2013.

Clase 12

Código Huffman (continuación)

Tenemos un texto T formado por símbolos de $\Sigma = \{x_1, \ldots, x_n\}$ tales que x_i aparece f_i veces. Queremos minimizar

$$B(R) = \sum_{1 \le i \le n} f_i d(x_i)$$

donde $d(x_i)$ es el largo del código binario para x_i obtenido en el árbol binario de R que corresponde al código prefijo del caso. Observaciones:

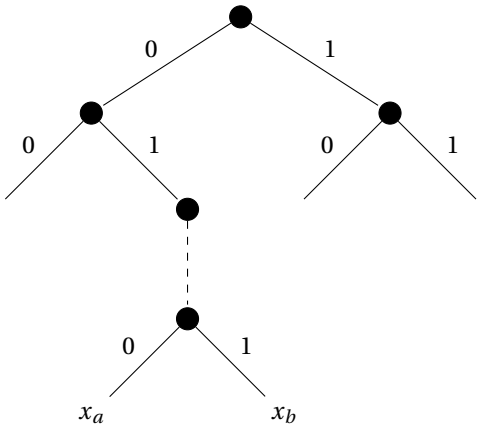


Figura 12.1 – Hojas a mayor profundidad representan símbolos con menor frecuencia.

- Si R es óptimo, todo nodo interno tiene dos hijos.
- Si $d(x_i)$ es la profundidad de x_i hay dos hojas x_a, x_b a la profundidad máxima que son hermanos.

12.1. Algoritmo

Sucesivamente:

1. Tome los dos símbolos con menos frecuencia de su tabla y reemplácelos por un nuevo símbolo que representa a ambos. Supongamos que estos símbolos son x_a y x_b, entonces el nuevo símbolo es x_{ab}. La frecuencia de este símbolo conjunto será la suma de la frecuencia de x_a y x_b.
2. Cree un árbol que tenga como raíz al símbolo conjunto x_{ab} con x_a y x_b como hijos.
3. Volver a 1 hasta que nuestra tabla esté formada por sólo un símbolo conjunto, que representará a todos los símbolos de Σ.

Ejemplo 12.1. Consideremos el cuadro 12.1. Nuestro algoritmo nos dice que debemos tomar dos

Símbolo	Frecuencia
a	2
b	6
c	3
d	3
e	21
f	5
g	15

Cuadro 12.1 – Frecuencias para los símbolos a, b, c, d, e, f, g.

símbolos con menor frecuencia en el cuadro 12.1, que en este caso serían a y c o a y d, y lo reemplazamos con uno nuevo ac o ad (para este ejemplo escogeremos ac). La frecuencia de este símbolo será la suma de las frecuencias de a y c. Es decir:

$$f_{ac} = f_a + f_c = 2 + 3 = 5$$

Luego, eliminamos los símbolos a y c del cuadro 12.1 y agregamos al símbolo conjunto ac dando como origen al cuadro 12.2. En seguida, representamos este símbolo conjunto a través de un árbol

Símbolo	Frecuencia
b	6
d	3
e	21
f	5
g	15
ac	5

Cuadro 12.2 – Eliminamos a y c de 12.1, reemplazamos por ac, con frecuencia $f_{ac} = 5$.

binario cuya raíz sería ac y sus hijos a y c, tal como se muestra en la figura 12.2. Nuevamente,

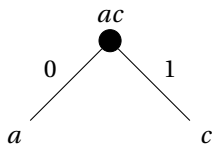


Figura 12.2 – Hojas son los símbolos que menos se repiten.

escogemos dos símbolos con menor frecuencia en el cuadro 12.2. Entre ellos, podemos escoger

- d y f
- d y ac

En esta ocasión, escogeremos los símbolos d y f (queda como tarea averiguar qué es lo que ocurre si escogemos d y ac). Entonces, eliminamos los símbolos d y f del cuadro 12.2 y lo reemplazamos por df, que tiene una frecuencia de $f_{fd} = f_d + f_f = 8$. El resultado de esto queda representado en el cuadro 12.3. En seguida, representamos este símbolo conjunto a través de un árbol binario con

Símbolo	**Frecuencia**
b	6
e	21
g	15
ac	5
df	8

Cuadro 12.3 – Eliminamos d y f de 12.2 agregando df con frecuencia $f_{df} = 8$.

raíz df e hijos d y f. Esto está representado en la figura 12.3. Pasamos a la siguiente iteración y

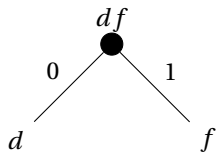


Figura 12.3 – Árbol de la segunda iteración. Tiene un peso de $f_{df} = 8$

buscamos en el cuadro 12.3 los dos símbolos de menor frecuencia. Estos son ac y b. Por lo tanto, removemos estos dos y agregamos un nuevo símbolo llamado bac (véase el cuadro 12.4). Luego,

Símbolo	**Frecuencia**
e	21
g	15
bac	11
df	8

Cuadro 12.4 – Eliminamos b y ac de 12.3, reemplazamos por bac, frecuencia $f_{bac} = f_b + f_{ac} = 11$.

como es costumbre, creamos un árbol con raíz el nodo bac e hijos b y ac (figura 12.4). Continuamos

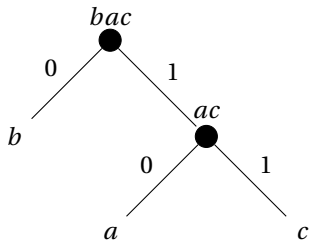


Figura 12.4 – Árbol en la tercera iteración.

en la cuarta iteración. En ella, buscamos los dos símbolos que tengan la menor frecuencia, los eliminamos y en su lugar agregamos otro símbolo que represente a ambos y cuya frecuencia será la suma de los dos eliminados. Mirando el cuadro 12.4, vemos que estos son bac y df con frecuencias $f_{bac} = 11$ y $f_{df} = 8$, respectivamente. Eliminamos estos dos y agregamos $dfbac$ con una frecuencia $f_{dfbac} = f_{df} + f_{bac} = 19$. El resultado se encuentra en el cuadro 12.5. En seguida, creamos un nuevo

Símbolo	Frecuencia
e	21
g	15
$dfbac$	19

Cuadro 12.5 – Eliminamos bac y df de 12.4, agregamos $dfbac$ con frecuencia $f_{dfbac} = 19$.

árbol con raíz $dfbac$ e hijos df y bac (figura 12.5). Vamos por la quinta iteración. Buscamos en el

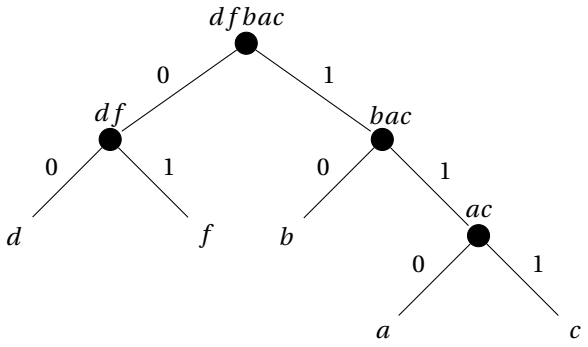


Figura 12.5 – Árbol a la cuarta iteración.

cuadro 12.5 los dos símbolos con menor frecuencia, y estos son g y $dfbac$ cuya suma de frecuencias es de 34. Entonces, quitamos estos símbolos y agregamos $gdfbac$ en su lugar (resultados en el cuadro 12.6). Luego, creamos un árbol con raíz $gdfbac$ con hijos g y $dfabc$, tal cual como se muestra

Símbolo	Frecuencia
e	21
$gdfbac$	34

Cuadro 12.6 – De 12.5 eliminamos g y $dfbac$ agregando $gdfbac$ con frecuencia 34.

en la figura 12.6. En la última iteración, vemos que el cuadro 12.6 sólo tiene dos símbolos, por lo tanto, es claro que el árbol generado por el Algoritmo de Huffman para la determinada secuencia de palabras del cuadro 12.1 es aquel que está representado en la figura 12.7. Finalmente, hacemos una tabla para representar la codificación de cada símbolo siguiendo el árbol de la figura 12.7. El resultado se encuentra en el cuadro 12.7. Esto da una codificación de 2,40 bits por caracter en promedio. Códigos de largo fijo requerirían $\lceil \log_2 7 \rceil = 3$ bits por caracter, esto da un ahorro de 20 %.

Llegamos a la parte entretenida: tenemos que demostrar que el algoritmo de Huffman halla un árbol óptimo.

Demostración. Para demostrar que es óptimo:

Elección Voraz: Lo demostramos la clase pasada.

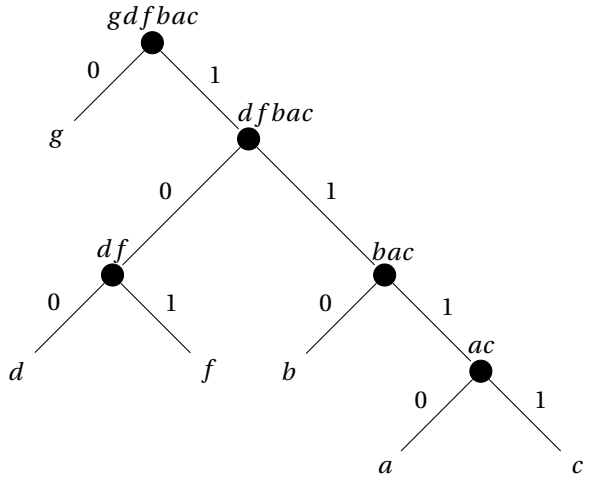


Figura 12.6 – Árbol generado en la quinta iteración.

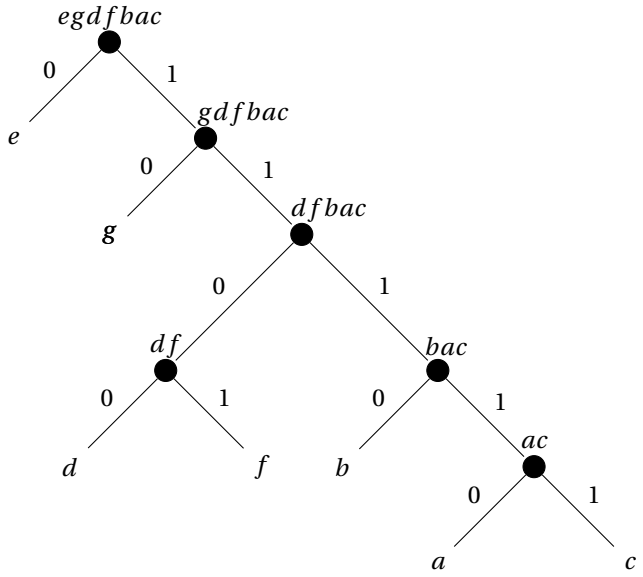


Figura 12.7 – Árbol generado por el Algoritmo de Huffman de los símbolos del cuadro 12.1.

Estructura inductiva: Elegir un (sub)árbol no interfiere con los demás.

Subestructura óptima: Recordemos que:

- L: instancia original. En otras palabras, el texto T que nos entregan o la tabla de símbolos con frecuencias respectivas.
- x_a, x_b símbolos de frecuencia mínima; f_a, f_b las frecuencias respectivas.
- R': árbol óptimo para L'
- L': instancia $L \setminus \{x_a, x_b\} \cup \{x_{ab}\}$, frecuencias, ...
- R: óptimo para L no es peor que $R' \setminus \{x_{ab}\} \cup x_a \cup x_b$

Símbolo	Código
a	11110
b	1110
c	11111
d	1100
e	0
f	1101
g	10

Cuadro 12.7 – Codificación final.

□

EVQQTEAR (Ejercicio voluntario para quienes quieran tener la esperanza de aprobar el ramo): Demostrar que los algoritmos de Kruskal, Prim y Huffman hallan los respectivos óptimos.

Clase 13

Programación Dinámica

13.1. Proyectos de plantas

Una corporación tiene US$ 5 millones a invertir este año, y planea expandir 3 de sus plantas. Cada planta ha entregado a lo más tres propuestas, con sus costos y retornos estimados. Las diferentes propuestas de las plantas son excluyentes, vale decir, de las tres se puede ejecutar solo una. El cuadro 13.1 resume los costos de las propuestas y sus retornos. Algunas plantas no completaron

	Planta 1		**Planta 2**		**Planta 2**	
Propuesta	c_1	r_1	c_2	r_2	c_3	r_3
1	0	0	0	0	0	0
2	1	5	2	8	1	4
3	2	6	3	9		
4			4	12		

Cuadro 13.1 – Propuestas, sus costos y retornos

las tres propuestas, y en todos los casos se agrega la propuesta de "no hacer nada". El objetivo es maximizar los retornos asignando los 5 millones. Se asume que si no se invierten todos, el resto se "pierde" (no genera retornos). Un ejercicio interesante es considerar opciones más realistas.

Una forma directa de resolver esto es considerar las $3 \cdot 4 \cdot 2 = 24$ posibilidades, y elegir la mejor. Claro que con más plantas y más proyectos, esto rápidamente se hace inmanejable.

Una manera de obtener la solución es la siguiente: dividamos el problema en tres *etapas* (cada etapa representa la asignación a una planta). Imponemos un orden artifical a las etapas, considerando las plantas en orden de número. Cada etapa la dividimos en *estados*, que recogen la información para ir a la etapa siguiente. En nuestro caso, los estados de la etapa 1 son $\{0,1,2,3,4,5\}$, correspondientes a invertir esas cantidades en la planta 1 Cada estado tiene un retorno asociado. Nótese que para decidir cuánto conviene asignar a la planta 3 (cual de los proyectos financiar) basta saber cuánto queda por asignar luego de financiar los proyectos de las plantas 1 y 2. Los proyectos aprobados no interesan. Note que nos interesa que $x = 5$ (queremos invertir todo, o la mayor parte posible).

Calculemos los retornos asociados a cada estado. Esto es simple en la etapa 1. El cuadro 13.2 resume los resultados. Estamos en condiciones de atacar la etapa 2, la mejor combinación para las

Capital x	Propuesta óptima	Retorno 1
0	1	0
1	2	5
2	3	6
3	3	6
4	3	6
5	3	6

Cuadro 13.2 – Cómputo de la etapa 1

plantas 1 y 2. Dada cierta cantidad total x a invertir, consideramos cada propuesta para la planta 2 en turno, y sumamos su retorno con lo que rendiría lo que reste al invertir de la mejor manera en la planta 1 (como da el cuadro 13.2). Por ejemplo, si en la planta 2 elegimos la propuesta 3, tenemos un retorno de 9 y nos queda 1 para la planta 1, que da retorno 5, para un total de 14. El cuadro resume esto para las distintas opciones. Vamos por la etapa 3, con la misma idea tenemos el cuadro 13.4.

Capital x	Propuesta óptima	Retorno 1 y 2
0	1	0
1	1	5
2	2	8
3	2	13
4	2	14
5	4	17

Cuadro 13.3 – Cómputo de la etapa 2

La entrada para $x = 5$ dice que el mejor retorno posible es 18. Nos indica que la mejor opción para

Capital x	Propuesta óptima	Retorno 1, 2 y 3
0	0	0
1	1	5
2	1	8
3	1	13
4	2	17
5	2	18

Cuadro 13.4 – Cómputo de la etapa 3

la planta 3 es su proyecto 2, lo que deja $5-1=4$ para las otras; del cuadro 13.3 vemos que la mejor opción para la planta 2 es la 2; queda $4-2=2$, con lo que del cuadro 13.2 vemos que la mejor opción para la planta 1 es la 3.

Podemos generalizar lo anterior. Sea r_{jk} el retorno para la propuesta k en la etapa j, y sea c_{jk} el costo de esa propuesta. Sea $f_j(x)$ la ganancia total en la etapa j si el capital disponible en ella es x.

Entonces, incluyendo siempre la opción de "no haga nada y no gaste en esta planta":

$$f_1(x) = \max_{k:\, c_{1k} \le x} \{r_{1k}\}$$
$$f_j(x) = \max_{k:\, c_{jk} \le x} \{r_{jk} + f_{j-1}(x - c_{jk})\}$$

y si son n etapas y tenemos x capital disponible nos interesa $f_n(x)$. Lo que hicimos arriba es evaluar esta recursión.

Como está desarrollado, trabajamos "hacia adelante". En muchos casos, resulta más natural plantearse estar en la última etapa, y ver cómo llegamos a ella. Esto da una recursión "hacia atrás". Ambas son equivalentes, claro está.

13.2. Producto de matrices

Queremos calcular el producto de n matrices, $A_1 \cdot A_2 \cdots A_n$, donde A_i es $n_i \times n_{i+1}$ (si no es $A_i \cdot A_{i+1}$ imposible).

La técnica tradicional de multiplicar una matriz de $r \times s$ por otra $s \times t$ toma rst multiplicaciones, y usaremos esto como medida de costo. Nuestro resultado no depende realmente del detalle de ésto.

Sabemos que la multiplicación de matrices es asociativa:

$$A \cdot (B \cdot C) = (A \cdot B) \cdot C$$

El trabajo total depende del orden:

A: 2×12
B: 12×3
C: 3×4

Entonces, el costo de calcular $(A \cdot B) \cdot C$ es:

$$(A \cdot B) \cdot C \rightsquigarrow 2 \cdot 12 \cdot 3 + 2 \cdot 3 \cdot 4 = 96$$

Explicamos de donde vienen los números anteriores:

- Comenzamos multiplicando $A \cdot B$. Como ambas matrices son de 2×12 y 12×3 respectivamente. Es claro que la cantidad de multiplicaciones que tenemos que hacer para obtener la matriz resultante es $2 \cdot 12 \cdot 3 = 72$. Dado lo anterior, se tiene que la dimensión de $A \cdot B$ es 2×3.
- Llegados a este punto, consideramos la matriz $A \cdot B$ como una sola, la que llamaremos X. Entonces, siguiendo lo anterior, vemos que $X \cdot C$ nos toma una cantidad de $2 \cdot 3 \cdot 4 = 24$ multiplicaciones.
- Finalmente, sumamos la cantidad de multiplicaciones que nos llevó a obtener $A \cdot B = X$ y $X \cdot C$, lo que se traduce a:

$$(A \cdot B) \cdot C \rightsquigarrow 2 \cdot 12 \cdot 3 + 2 \cdot 3 \cdot 4 = 96 \tag{13.1}$$

Continuamos con el cálculo del costo de obtener $A \cdot (B \cdot C)$:

$$A \cdot (B \cdot C) \rightsquigarrow 2 \cdot 12 + 4 + 12 \cdot 3 \cdot 4 = 240$$

Para obtener el cálculo óptimo, consideremos "de afuera adentro)". Si el último producto es:

$$\underbrace{(A_1 \cdot A_2 \cdots A_i)}_{\text{óptimo}} \underbrace{(A_{i+1} \cdots A_n)}_{\text{óptimo}} \tag{13.2}$$

Nótese que $(A_1 \cdot A_2 \cdots A_i)$ y $(A_{i+1} \cdots A_n)$ también deben calcularse de forma óptima, de lo contrario el cuento no sirve. En consecuencia, tendremos que dividir nuevamente lo que está entre paréntesis sucesivamente hasta obtener un producto de dos matrices. Es importante destacar que no conocemos $i\ldots$ así que tendremos que probar todas las opciones. Ojo, muchos subproblemas se repiten.

Idea de programación no recursiva:

- $T[i,j]$: costo de calcular el producto $A_i \cdot \ldots \cdot A_j$.

Inicialmente:

$$T[i,i] = 0$$

Sabemos que:

$$T[i,j] = \min_k \{T[i,k] + T[k+1,j] + \underbrace{n_i \cdot n_{k+1} \cdot n_{j+1}}_{\text{costo del producto}}\} \tag{13.3}$$

Donde k corresponde a la k-ésima matriz que hicimos el corte (Donde dividimos con los paréntesis. Por ejemplo, en (13.2) se tiene que $k = i$) y que dio el valor mínimo de (13.3). Además, n corresponde al valor izquierdo de la dimensión de la matriz. Por ejemplo, si nuestra A_i tiene dimensión $a \times b$, $A_{k+1} : c \times d$ y $A_{j+1} : e \times f$, se tiene que $n_i = a$, $n_{k+1} = c$ y $n_{j+1} = e$ respectivamente.

Nos interesa: $T[1,n]$. Calculamos:

$$T[i,i]$$
$$T[i,i+1]$$
$$\vdots$$

Esta da sólo el costo… hay que registrar con cada $T[i,j]$ cuál fue el k que dio el mínimo. Siguiendo esos desde $T[1,n]$ da el orden óptimo.

Ejemplo 13.1. Supongamos que queremos calcular el producto de matrices:

$$A \cdot B \cdot C \cdot D \cdot E \cdot F$$

donde cada uno de los tamaños son:

- $A = A_1 : 1 \times 2$
- $B = A_2 : 2 \times 7$
- $C = A_3 : 7 \times 4$
- $D = A_4 : 4 \times 2$
- $E = A_5 : 2 \times 9$
- $F = A_6 : 9 \times 1$

	1	2	3	4	5	6
1	0 1					
2		0 2				
3			0 3			
4				0 4		
5					0 5	
6						0 6

Cuadro 13.5 – Para la primera iteración no necesitamos realizar multiplicaciones.

Para la primera iteración, es decir, $T[i,i]$ es claro que intentamos calcular la cantidad de productos que son necesarios para hacer la multiplicación A_i. Cómo no lo estamos multiplicando con nada más, se tiene que la cantidad de multiplicaciones necesarias es $T[i,i] = 0$ para cualquier i. Agregamos esta información al cuadro 13.5. Nótese que cada casilla del cuadro 13.5 está configurado de la forma que se logra apreciar en el cuadro 13.6 independientemente de la iteración en la que vayamos. Para la segunda iteración, tenemos que calcular todos los $T[i,i+1]$, es decir, la mínima cantidad de

	1	2	3	4	5	6
1	$T[1,1]$ k	$T[1,2]$ k	$T[1,3]$ k	$T[1,4]$ k	$T[1,5]$ k	$T[1,6]$ k
2		$T[2,2]$ k	$T[2,3]$ k	$T[2,4]$ k	$T[2,5]$ k	$T[2,6]$ k
3			$T[3,3]$ k	$T[3,4]$ k	$T[3,5]$ k	$T[3,6]$ k
4				$T[4,4]$ k	$T[4,5]$ k	$T[4,6]$ k
5					$T[5,5]$ k	$T[5,6]$ k
6						$T[6,6]$ k

Cuadro 13.6 – Los $T[i,j]$ con $j > i$ de (13.3) y k al corte que da el mínimo de $T[i,j]$.

multiplicaciones para obtener $A_i \cdot A_{i+1}$. Como solo tenemos dos matrices involucradas, es bastante fácil realizar este cálculo:

- $T[1,2] = 1 \cdot 2 \cdot 7 = 14$
- $T[2,3] = 2 \cdot 7 \cdot 4 = 56$
- $T[3,4] = 7 \cdot 4 \cdot 2 = 56$
- $T[4,5] = 4 \cdot 2 \cdot 9 = 72$
- $T[5,6] = 2 \cdot 9 \cdot 1 = 18$

Luego, agregamos estos valores al cuadro 13.5. Los cambios se pueden apreciar en el cuadro 13.7. Seguimos con la tercera iteración. En esta ocasión, tendremos que calcular los $T[i, i+2]$, es decir, la

	1	2	3	4	5	6
1	0 1	14 1				
2		0 2	56 2			
3			0 3	56 3		
4				0 4	72 4	
5					0 5	18 5
6						0 6

Cuadro 13.7 – Segunda iteración.

cantidad de multiplicaciones mínima para obtener $A_i \cdot A_{i+1} \cdot A_{i+2}$. Para ello, comenzamos calculando $T[1,3]$, es decir:

$$T[1,3] = \min_k \{T[1,k] + T[k+1,3] + n_1 \cdot n_{k+1} \cdot n_4\} \tag{13.4}$$

Vamos por partes:

- Para $k = 1$:

$$\begin{aligned} T[1,3](k=1) &= T[1,1] + T[2,3] + n_1 \cdot n_2 \cdot n_4 \\ &= 0 + 56 + 1 \cdot 2 \cdot 4 = 64 \end{aligned}$$

- Para $k = 2$:

$$\begin{aligned} T[1,3](k=2) &= T[1,2] + T[3,3] + n_1 \cdot n_3 \cdot n_4 \\ &= 14 + 0 + 1 \cdot 7 \cdot 4 = 42 \end{aligned}$$

Por lo tanto, de acuerdo a lo anterior la ecuación (13.4) obtiene el mínimo cuando hacemos el corte en $k = 2$. Es decir, obtenemos el mínimo de productos para $A_1 \cdot A_2 \cdot A_3$ si hacemos un corte

$$(A_1 \cdot A_2) \cdot A_3 \tag{13.5}$$

En otras palabras, si resolvemos primero $A_1 \cdot A_2$ y luego, la matriz resultante se la multiplicamos a A_3 obtendremos el mínimo para $T[1,3]$. Más adelante agregaremos esto a la tabla.

Para dejar más en claro cómo calcular (13.3) repetiremos los pasos anteriores, pero para calcular $T[2,4]$, el que se puede obtener a través de:

$$T[2,4] = \min_k \{T[2,k] + T[k+1,4] + n_2 \cdot n_{k+1} \cdot n_5\} \tag{13.6}$$

Vamos por partes:

- Para $k = 2$:

$$\begin{aligned} T[2,4](k=2) &= T[2,2] + T[3,4] + n_2 \cdot n_3 \cdot n_5 \\ &= 0 + 56 + 2 \cdot 7 \cdot 2 = 84 \end{aligned}$$

- Para $k = 3$:

$$\begin{aligned} T[2,4](k=3) &= T[2,3] + T[4,4] + n_2 \cdot n_4 \cdot n_5 \\ &= 56 + 0 + 2 \cdot 4 \cdot 2 = 72 \end{aligned}$$

Nuevamente, de acuerdo a la ecuación (13.6) el mínimo para $T[2,4]$ se obtiene para $k = 3$, lo que implica que la cantidad mínima de productos para $A_2 \cdot A_3 \cdot A_4$ se obtiene al hacer un corte en $k = 3$, es decir,

$$(A_2 \cdot A_3) \cdot A_4 \tag{13.7}$$

Esto nos dice que si resolvemos primero el producto $A_2 \cdot A_3$ y luego la matriz resultante se la multiplicamos a A_4, vamos a hacerlo en la menor cantidad de operaciones posibles.

El objetivo de este ejemplo es mostrar cómo funciona el algoritmo de Programación Dinámica, por lo que no es necesario explicar paso a paso cómo obtener el resto de las casillas de la tabla. Así que agregamos los valores obtenidos al cuadro 13.7, cuyo resultado se refleja en el cuadro 13.8. Nuestro principal interés en esto es obtener el último producto representado en (13.2). Para efectos

	1	2	3	4	5	6
1	0 1	14 1	42 2			
2		0 2	56 2	72 3		
3			0 3	56 3		
4				0 4	72 4	
5					0 5	18 5
6						0 6

Cuadro 13.8 – Algunos valores correspondientes de la tercera iteración.

de nuestra tablita, este último producto está ubicado en la casilla pintada marcada con un × del cuadro 13.9.

Demostración. Como siempre, demostramos que cumple con:

Estructura inductiva: Dada la selección k (última) se subdivide en problemas, cuyas soluciones viables $+k$ dan una solución viable para todo.

Subestructura óptima: Con soluciones óptimas para $1 \cdots k$ y $k+1 \cdots n$ obtenemos la solución óptima para $1 \cdots n$, *suponiendo* k.

Elección completa: Elegimos aquel k que da el mejor "último paso".

	1	2	3	4	5	6
1	0 1	14 1	42 2			× k
2		0 2	56 2	72 3		
3			0 3	56 3		
4				0 4	72 4	
5					0 5	18 5
6						0 6

Cuadro 13.9 – Nos interesa obtener el "último producto".

□

Por el momento solo nos interesa obtener el valor del costo mínimo, más adelante veremos cómo construir la solución de los k registrados para los mejores pasos.

Ejercicios

1. Considere los valores $\langle a_n \rangle$ y $\langle b_n \rangle$ definidos mediante:

$$a_0 = a_1 = 1$$
$$b_0 = b_1 = 2$$
$$a_n = a_{n-2} + b_{n-1}$$
$$b_n = a_{n-1} + b_{n-2}$$

Podemos calcularlos mediante las funciones recursivas dadas en el algoritmo 13.1.

Algoritmo 13.1: Cálculo obvio de a_n y b_n

```
function CalculeA(n)
    if n < 2 then
        return 1
    else
        return CalculeA(n − 2) + CalculeB(n − 1)
    end
function CalculeB(n)
    if n < 2 then
        return 2
    else
        return CalculeA(n − 1) + CalculeB(n − 2)
    end
```

a) Demuestre que el tiempo que demanda 13.1 para calcular a_n es exponencial.

b) Describa un algoritmo más eficiente para calcular a_n, y derive su complejidad.

2. Escriba un programa que resuelva el caso general de asignación de proyectos a plantas, para un número arbitrario de plantas y proyectos. Entregue no solo el mejor retorno, sino también los proyectos a ser realizados. ¿Qué modificaciones deben hacerse a las recurrencias para acomodar el caso en que la opción de no hacer nada no se muestre explícitamente?

3. Un *palíndromo* es una palabra que se lee igual de adelante o de atrás, como *arenera* o *reconocer*. Cualquier palabra puede verse como una secuencia de palíndromos, considerando una única letra como un palíndromo de largo 1. Interesa obtener la división de una palabra en el mínimo número de palíndromos, MinPal(σ).

   a) Describa la recurrencia para MinPal(σ) en términos de subpalabras de σ.

   b) Describa un algoritmo que toma tiempo $O(|\sigma|^3)$ para hallar MinPal(σ).

4. Considere una hoja rectangular de papel cuadriculado, algunos de cuyos cuadritos están marcados con X. Nos interesa determinar para cada cuadradito el largo de la secuencia de X de largo máximo que pasa por él, en vertical, horizontal o diagonal.

5. Nos encargan escribir frases usando un conjunto de azulejos predefinidos $\{A_0, \ldots, A_{m-1}\}$ con secuencias de letras (y espacios). Debemos determinar si se puede escribir la frase S con los azulejos dados, o sea si:

   $$S = A_{i_1} A_{i_2} \ldots A_{i_n}$$

   para alguna secuencia $\langle i_1, i_2, \ldots, i_n \rangle$.

6. Sean $x = x_1 x_2 \ldots x_m$, $y = y_1 y_2 \ldots y_n$, $z = z_1 z_2 \ldots z_{m+n}$ dos palabras (los x_i, y_i y z_i son símbolos individuales). Un *barajamiento* de x e y es una palabra de largo $|x| + |y|$ en la cual aparecen x e y como subsecuencias no traslapantes. La pregunta es si z es un barajamiento de x e y, y dar una posible división de z en subsecuencias (note que pueden haber varias posibilidades).

Clase 14

Subsecuencia común más larga

En inglés conocido como *Longest Common Subsequence*, LCS. La *idea* consiste en que tenemos dos archivos: X e Y, y queremos hacer `diff`[1] sobre ellos, es decir ejecutar el comando `diff X Y`. Para ello:

- Hallar la subsecuencia común más larga (LCS) entre ellos.
- Marcar líneas agregadas/borradas

Ejemplo 14.1. Supongamos que tenemos dos archivos: X e Y, cuyo contenido se muestra en el cuadro 14.1. En consecuencia, si hacemos un `diff X Y` obtenemos como resultado, la columna

X	Y
foo	bar
bar	xyzzy
baz	plugh
quux	baz
windows	foo
	quux
	linux

Cuadro 14.1 – Los archivos X e Y. Cada fila de la tabla es una línea del archivo.

"Resultado" del cuadro 14.2. Explicamos con más detalle cómo funciona el algoritmo acorde a lo arrojado por el cuadro 14.2:

- Comenzamos leyendo ambos archivos. Como podemos observar en el cuadro 14.1, la primera línea de Y es `bar`. Por lo tanto, buscamos en X la primera línea que tenga `bar`. En X, la línea `bar` se encuentra en la línea 2, así que tendremos que eliminar el contenido de la línea 1 de X. Por lo tanto, en la columna "Resultado" agregamos `-bar`.
- En este momento, ambos archivos comienzan con `bar`, así que no realizamos ningún cambio y agregamos esto a "Resultado". El cuadro 14.3 muestra nuestra dónde estamos ubicados actualmente.

[1] pruebe `man diff` para ver qué es lo que hacer

Línea	X	Y	Resultado
1	foo	bar	- foo
2	bar	xyzzy	bar
3	baz	plugh	+ xyzzy
4	quux	baz	+ plugh
5	windows	foo	baz
6		quux	+ foo
7		linux	quux
8			- windows
9			+ linux

Cuadro 14.2 – La columna "Resultado" resume las operaciones.

Línea	X	Y	Resultado
1	foo	bar	- foo
2	bar	xyzzy	bar
3	baz	plugh	
4	quux	baz	
5	windows	foo	
6		quux	
7		linux	

Cuadro 14.3 – Celdas azules representan dónde estamos, celdas rojas son la siguiente coincidencia.

- Iteramos hasta encontrar la siguiente coincidencia (en este punto nos encontramos en la línea 2 de X y 1 de Y). La siguiente coincidencia es la línea `baz`. Como podemos observar en el cuadro 14.3, para poder construir Y a partir de X agregamos las líneas `xyzzy` y `plugh`. Luego, nos ubicamos en las líneas donde se produjo dicha coincidencia para ambos archivos y aprovechamos a de agregarlos (ver cuadro 14.4).

Línea	X	Y	Resultado
1	foo	bar	- foo
2	bar	xyzzy	bar
3	baz	plugh	+ xyzzy
4	quux	baz	+ plugh
5	windows	foo	baz
6		quux	
7		linux	

Cuadro 14.4 – Agregamos `xyzzy` y `plugh` a "Resultado".

- La siguiente coincidencia es `quux`.
  - En X, `quux` está inmediatamente después de la línea que tiene `baz`.
  - En Y, para llegar a la línea que tiene `quux` desde `baz` tendremos que pasar `foo`.

  Dado lo anterior, en X tendremos que agregar la línea `foo` para poder construir Y. Los resultados se pueden apreciar en el cuadro 14.5.

Línea	X	Y	Resultado
1	`foo`	`bar`	`- foo`
2	`bar`	`xyzzy`	`  bar`
3	`baz`	`plugh`	`+ xyzzy`
4	`quux`	`baz`	`+ plugh`
5	`windows`	`foo`	`  baz`
6		`quux`	`+ foo`
7		`linux`	`  quux`

Cuadro 14.5 – Agregamos `foo` a "Resultado".

- Como podemos apreciar en el cuadro 14.5, ya no tenemos más coincidencias. Por lo tanto, para terminar simplemente eliminamos `windows` y agregamos `linux`.[2] La tabla resultante puede verse a través del cuadro 14.2.

14.1. Aspectos formales

Arreglos $X[1,\ldots,n]$, $Y[1,\ldots,m]$, palabras sobre un alfabeto ("símbolo" es una línea). Hallar la secuencia de pares de índices (números de línea) (x_1,y_1), (x_2,y_2), $\ldots$, (x_q,y_q) tales que:

$$\begin{aligned} x_1 < x_2 < \cdots < x_q \quad & ;\forall x_i \in \mathbb{N} \\ y_1 < y_2 < \cdots < y_q \quad & ;\forall x_i \in \mathbb{N} \\ X[x_i] = Y[y_i] \quad & \text{para } 1 \leq i \leq q \end{aligned}$$

Interesa la secuencia más larga (máximo q, correspondiente a la cantidad coincidencias). Para ello, consideremos $X[n]$, $Y[m]$. Tres opciones:

1. $X[n]$ queda fuera de la subsecuencia. En consecuencia, lo marcamos con `(-)`
2. $Y[m]$ queda fuera de la subsecuencia. Para efectos de `diff` marcamos con `(+)`

\+ O ambos

3. Si $X[n] = Y[m]$, hacer $x_q = n$, $y_q = m$.

Notar que si $X[n] \neq Y[m]$ no pueden pertenecer ambos a la LCS.

Tres opciones. Subproblemas:

1. $X[1,\ldots,n-1]$, Y
2. X, $Y[1,\ldots,m-1]$
3. Solo si $X[n] = Y[m]$, $X[1,\ldots,n-1]$, $Y[1,\ldots,m-1]$

Sea $LCS(A,B)$ la subsecuencia común más larga entre A y B. Entonces (hasta el momento sólo nos interesa el largo de la subsecuencia óptima, después veremos como encontrar la secuencia):

$$\begin{aligned} |LCS(X,Y)| = \text{máx}\{&|LCS(X[1,\ldots,n-1),Y|]+0, |LCS(X,Y[1,\ldots,m-1)|]+0, \\ &|LCS(X[1,\ldots,n-1),Y[1,\ldots,m-1]|+1\} \end{aligned}$$

[2] No nos arrepentimos de nada.

Esto sugiere un arreglo $L[i, j]$:

$$L[i, j] = |LCS(X[1, \ldots, i), Y[1, \ldots, j|] \tag{14.1}$$

Sabemos $L[0, j] = L[i, 0] = 0$. Para calcular $L[i, j]$ necesitamos $L[i-1, j]$, $L[i, j-1]$, $L[i-1, j-1]$ (posiblemente). Llenar el arreglo, calculando $L[i, k]$ para i de 1 a n, llenando los j de 1 a m. Vemos que el costo total es $O(n \cdot m)$.

Clase 15

Más de programación dinámica

Hasta el momento hemos tratado casos en los cuales la aplicación de las recurrencias es directo. Mostraremos un par de ejemplos en los cuales se requiere trabajo previo. Vienen de las notas de Ignjatović [1].

15.1. Torre de Tortugas

Nos dan n tortugas, para cada una se da su peso y su resistencia. La resistencia de una tortuga es el peso máximo que es capaz de soportar sin romper su caparazón. Se busca el máximo número de tortugas que se pueden apilar sin romper sus caparazones.

Llamemos $T_1, \ldots, T_n$ a las tortugas (en orden arbitrario), donde el peso de T_i es $W(T_i)$ y su fuerza $S(T_i)$. Diremos que una torre de tortugas es *legítima* si la fuerza de cada tortuga es mayor o igual al peso de las tortugas sobre ella. Ordenamos las torres desde la punta a la base.

La programación dinámica consiste en construir recursivamente una solución al problema de soluciones a subproblemas. Podemos plantear por ejemplo para $1 \leq j \leq n$ el subproblema $P(j)$ de hallar el máximo número de tortugas del conjunto $\{T_1, \ldots, T_j\}$ que pueden apilarse. Lamentablemente, este planteo no permite recursión. Nos interesaría hallar una solución a $P(j)$ vía soluciones a todos los problemas $P(i)$ para $1 \leq i < j$. Pero la cadena más larga construida con tortugas de $\{T_1, \ldots, T_j\}$ puede que incluya a T_j, pero no en la última posición. Por tanto la solución óptima a $P(j)$ no siempre es una simple extensión de una solución óptima a algún $P(i)$ anterior.

Debemos hallar un ordenamiento adecuado junto con un subconjunto de subproblemas que permitan recurrencia. Hallar tal ordenamiento no es simple.

Proposición 15.1. *Si hay una torre legítima de altura k, hay una torre legítima de altura k en orden no-decreciente de peso más fuerza.*

Demostración. Demostraremos que cualquier torre legítima puede reordenarse para dar otra torre legítima en el orden indicado. Sea $\langle t_1, \ldots, t_m \rangle$ una torre legítima, basta demostrar que si dos tortugas consecutivas cumplen:

$$W(t_{i+1}) + S(t_{i+1}) < W(t_i) + S(t_i)$$

podemos intercambiar esas tortugas obteniendo otra torre legítima. Con esto, podemos usar la idea del método de burbuja para ordenar las tortugas en orden creciente de $W+S$, manteniendo siempre la legitimidad.

Sea τ la torre original y τ^* la obtenida al intercambiar. O sea:

$$\tau = \langle t_1, \ldots, t_{i-1}, t_i, t_{i+1}, \ldots, t_m \}$$
$$\tau^* = \langle t_1, \ldots, t_{i-1}, t_{i+1}, t_i, \ldots, t_m \}$$

La única tortuga con más carga en su espalda en τ^* es t_i, debemos demostrar que no sobrepasa su fuerza, o sea que:

$$\sum_{1 \le r \le i-1} W(t_r) + W(t_{i+1}) < S(t_i)$$

Como la torre original era legítima:

$$\sum_{1 \le r \le i-1} W(t_r) + W(t_i) \le S(t_{i+1})$$

Sumando $W(t_{i+1})$ a esta desigualdad tenemos:

$$\sum_{1 \le r \le i-1} W(t_r) + W(t_i) + W(t_{i+1}) \le W(t_{i+1}) + S(t_{i+1})$$

Pero supusimos $W(t_{i+1}) + W(t_{i+1}) < W(t_i) + S(t_i)$, con lo que:

$$\sum_{1 \le r \le i-1} W(t_r) + W(t_i) + W(t_{i+1}) < W(t_i) + S(t_i)$$
$$\sum_{1 \le r \le i-1} W(t_r) + W(t_{i+1}) < S(t_i)$$

Esto era lo que había que demostrar. □

La proposición 15.1 permite restringirnos a torres no-decrecientes en $W+S$ y aún así obtener una solución óptima. Supondremos entonces que las tortugas están numeradas en este orden.

Pero hay un problema: consideremos la torre legítima más alta que termina en T_i:

$$\langle t_1, \ldots, t_m, T_i \rangle$$

donde $\{t_1, \ldots, t_{m-1}, t_m\} \subseteq \{T_1, \ldots, T_{i-1}\}$. Desafortunadamente, la torre $\langle t_1, \ldots, t_{m-1}, t_m \rangle$ podría no ser la torre más alta con t_m en la base; puede haber una torre legítima con al menos $m+1$ tortugas $\langle t_1^*, \ldots, t_m^*, t_m \rangle$ pero demasiado pesada para la tortuga T_i. Esta formulación no cumple con subestructura óptima, debemos generalizar nuestro problema con mayor cuidado.

Construiremos una secuencia de las torres más livianas de cada altura. O sea, resolvemos los siguientes subproblemas para $j \le n$: $P'(j)$ para cada $r < j$ para el que hay una torre de tortugas de largo r de tortugas del conjunto $\{T_1, \ldots, T_j\}$ (no necesariamente incluyendo a T_j) encuentre la más liviana. Con esto la recursión funciona: resuelto el problema $P'(i-1)$, buscamos la torre más liviana θ_k^i de largo k incluyendo solo tortugas $\{T_1, \ldots, T_i\}$. Para ello consideramos las torres más livianas θ_k^{i-1} y θ_{k-1}^{i-1}, y vemos si podemos extender legítimamente la última con T_i. Esto da el óptimo, si sobre T_i podemos poner una torre de largo m, ciertamente podemos poner la torre más liviana de largo m sobre ella. Si la torre más alta construida con $\{T_1, \ldots, T_{i-1}\}$ tiene altura m y T_i puede extenderla, obtenemos la primera torre de altura $m+1$ compuesta con $\{T_1, \ldots, T_i\}$. Nótese que nuestro problema se hizo bidimensional en el proceso.

15.2. Variación mínima

Definimos la *variación total* de una secuencia $s = \langle x_1, \ldots, x_n \rangle$ como:

$$V(s) = \sum_{1 \le i \le n-1} |x_{i+1} - x_i|$$

Dan una secuencia de números $a_1, \ldots, a_n$. Divídala en dos subsecuencias (manteniendo el orden original) de manera que la suma de las variaciones totales de las subsecuencias se la menor posible, o sea, halle:

$$s_1 = \langle a_{i_1}, \ldots, a_{i_k} \rangle \qquad i_1 < i_2 < \cdots < i_k$$
$$s_2 = \langle a_{j_1}, \ldots, a_{j_k} \rangle \qquad j_1 < j_2 < \cdots < j_{n-k}$$

y $\{i_1, i_2, \ldots, i_k\} \cup \{j_1, j_2, \ldots, j_{n-k}\} = \{1, 2, \ldots, n\}$ tal que $V(s_1) + V(s_2)$ es mínimo.

Esta también tiene su truco. Uno se ve tentado a resolver subproblemas $P(j)$ para todo $m \le n$, donde $P(j)$ es dividir $\langle a_1, \ldots, a_m \rangle$ en subsecuencias con mínima variación. Extendemos las subsecuencias $\langle x_1, \ldots, x_r \rangle$ donde $r \le m$ y $\langle y_1, \ldots, y_s \rangle$ donde $s \le m$ considerando el menor de $|x_r - a_{m+1}|$ y $|y_s - a_{m+1}|$ para agregar a_{m+1} a una o la otra. Desafortunadamente, puede haber una división no óptima de $\langle a_1, \ldots, a_m \rangle$ en $\langle u_1, \ldots, u_{r'} \rangle$ y $\langle v_1, \ldots, v_{s'} \rangle$ tal que:

$$\sum_i |u_{i+1} - u_i| + \sum_j |v_{j+1} - v_j| > \sum_i |x_{i+1} - x_i| + \sum_j |y_{j+1} - y_j|$$

pero tal que $|v_{s'} - a_{m+1}|$ es mucho menor que $|x_r - a_{m+1}|$ y $|y_s - a_{m+1}|$, de manera que:

$$\sum_i |u_{i+1} - u_i| + \sum_j |v_{j+1} - v_j| + |v_s - a_{m+1}| < \sum_i |x_{i+1} - x_i| + \sum_j |y_{j+1} - y_j| + \min\{|x_r - a_{m+1}|, |y_s - a_{m+}|\}$$

Para resolver esto, planteamos el siguiente problema bidimensional: $P(r, s)$ es dividir la secuencia en secuencias que terminan en a_r y a_s de forma que la suma de sus variaciones totales se minimice. Para la solución del subproblema $P(r, s)$ consideramos varios casos:

1. Si $r < s - 1$, extendemos la solución óptima para $P(r, s-1)$ agregando a_s a la secuencia que termina en a_{s-1}, ya que la otra termina en a_r.

2. Si $r = s - 1$, consideramos soluciones para todos los subproblemas $P(t, s-1)$ con $t < s - 1$, extendiendo la subsecuencia que termina en a_t y eligiendo aquella con la mínima variación total. Esto lo comparamos con las subsecuencias $|a_1, \ldots, a_{s-1}|$ y $|a_s|$, reteniendo la menor.

Ejercicios

1. Complete la discusión sobre la torre de tortugas, desarrollando un programa que resuelva el problema. ¿Cuál es su complejidad?

2. Use un razonamiento similar a la torre de tortugas para hallar la subsecuencia creciente más larga de una secuencia de n números en tiempo $O(n \log n)$.

3. Reduzca el problema de variación mínima a una única dimensión considerando los subproblemas $P(s-1, s)$ únicamente.

4. Escriba un programa que resuelva el problema de variación mínima. ¿Cuál es su complejidad?

5. Halle, módulo 10^{16}, el número de subconjuntos no vacíos de $\{1^1, 2^2, 3^3, \ldots, 250250^{250250}\}$ cuya suma es divisible por 250.

6. Una *partición* de $n \in \mathbb{N}$ es un conjunto $\{p_1, \ldots, p_k\}$ (las *partes*, $p_i \in \mathbb{N}$) tal que $p_1 + \cdots + p_k = n$. Dé un algoritmo para obtener el número de particiones de n, y dé su complejidad.

7. El *problema del vendedor viajero* es un famoso problema NP-completo. Plantea un grafo $G = (V, E)$ con costos de arcos $w(e)$ para $e \in E$. Muestre cómo resolverlo, eligiendo $u \in V$ arbitrario para comenzar la gira, sean $u \neq v \in S \subseteq V$, y sea $d[v][S]$ el costo mínimo de un viaje por todos los vértices de S, comenzando en u y terminando en v. Plantee una recurrencia para d considerando el último arco del viaje. ¿Cuál es la complejidad de su algoritmo?

Bibliografía

[1] Aleksandar Ignjatović: *COMP 3121/9101/3821/9801 lecture notes: More on dynamic programming (DP)*. http://www.cse.unsw.com/~cs3121/Lectures/COMP3121_Lecture_Notes_DP.pdf, April 2016.

Clase 16

Recursión

Nuestra herramienta principal es *reducir* problemas a problemas más "simples". Por ejemplo: de una expresión regular obtener un programa eficiente para reconocer un patrón. Para resolver este problema usamos:

- Expresión Regular a NFA (por ejemplo Thompson)
- NFA a DFA (algoritmo de subconjuntos)
- DFA a DFA mínimo (hay varias opciones)
- Interpretar el DFA o traducirlo a código.

Al resolver un problema, lo dividimos en subproblemas y combinamos resultados. Notar que al hacer esto (por ejemplo, invocar `printf(3)` en C) confiamos en que la solución al subproblema hace su trabajo correctamente. *Confiamos* en terceros. De la misma manera, al invocar una función que nosotros escribimos, *confiamos* en que hace su trabajo correctamente. Usar recursión es lo mismo, solo que la función se invoca a sí misma (directa o indirectamente). Recursión es inducción en forma de programa. Igual que en inducción requerimos casos base y paso de inducción.

La belleza de la recursión es que nos debemos preocupar solo del problema entre manos, subproblemas se resuelven automáticamente.

16.1. Torres de Hanoi

Descripción del problema: Hay 64 placas redondas ubicadas de mayor a menor (figura 16.1). Solo se puede mover una placa a la vez y nunca una placa mayor sobre una placa menor.

16.1.1. Solución (recursiva)

Una solución (recursiva) es como se muestra en la figura 16.2. Esto es solución porque traduce el problema de mover n piezas a mover $n-1$ recursivamente, luego mover 1 (trivial, figura 16.2), luego mover $n-1$ recursivamente.

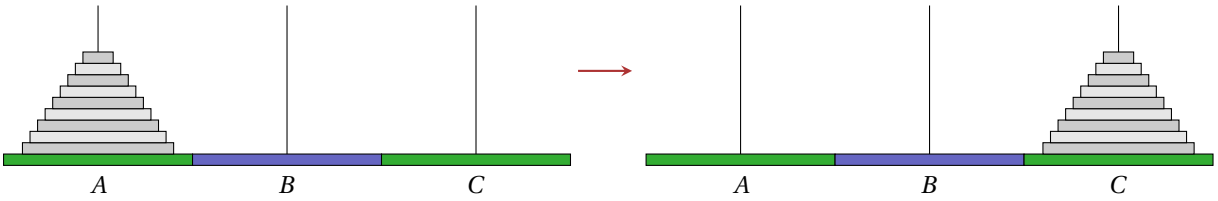


Figura 16.1 – Mover las placas desde la plataforma A a la C.

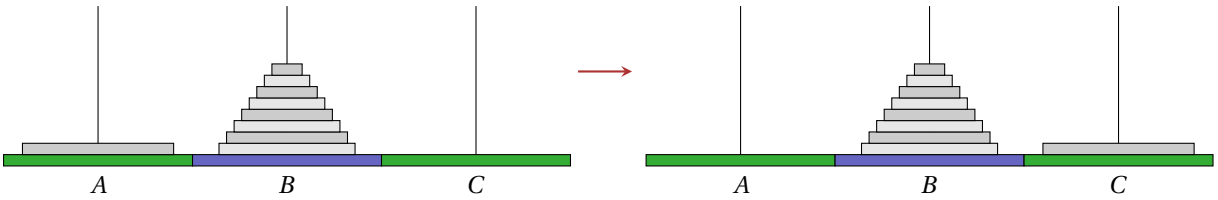


Figura 16.2 – Solo nos queda mover el último disco (más grande) de A a C directamente.

Problema: Mover n piezas de A a C.

Demostración. **Base:** Si $n = 0$, no hay que hacer nada.

Inducción: Supongamos que sabemos mover k piezas de i a j (con $i \neq j$). Para mover $k+1$ piezas de A a B (con C de "apoyo"):

- Movemos las k piezas superiores de A a C.
- Movemos la pieza inferior de A a B.
- Movemos las k piezas de C a B.

□

Pseudocódigo es el del algoritmo 16.1. Por la estructura de la solución, siempre cumplimos con

Algoritmo 16.1: Solución recursiva a las torres de Hanoi

```
procedure hanoi(n, src, dst, tmp)
    if n > 0 then
        hanoi(n − 1, src, tmp, dst)
        Mover de src a dst
        hanoi(n − 1, tmp, dst, src)
    end
```

las restricciones.

¿Cuántas movidas se requieren? Sea $T(n)$ el número de movidas para transferir n platos usando el algoritmo 16.1.

$$\begin{aligned} T(0) &= 0 \\ T(n+1) &= 2T(n)+1, \qquad n \geq 0 \end{aligned}$$

Sea:

$$h(z) = \sum_{n\geq 0} T(n) z^n \tag{16.1}$$

Por propiedades:

$$\frac{h(z) - T(0)}{z} = 2h(z) + \frac{1}{1-z}$$

Entonces, despejando $h(z)$ de lo anterior se obtiene:

$$h(z) = \frac{z}{(1-z)(1-2z)} \tag{16.2}$$

$$h(z) = \frac{A}{1-z} + \frac{B}{1-2z}$$

$$h(z)(1-z) = A + \frac{(1-z)B}{1-2z} \tag{16.3}$$

En seguida, aplicamos límite cuando $z \to 1$:

$$h(z)(1-z) = A + \frac{(1-z)B}{1-2z}$$

$$\lim_{z\to 1} h(z)(1-z) = \lim_{z\to 1}(A + \frac{\cancelto{0}{(1-z)B}}{1-2z}$$

$$\lim_{z\to 1} h(z)(1-z) = A \tag{16.4}$$

Luego, reemplazamos $h(z)$ de (16.4) por lo que está en (16.2):

$$A = \lim_{z\to 1} \frac{z}{(1-z)(1-2z)}(1-z)$$

$$= \lim_{z\to 1} \frac{z}{1-2z}$$

Y resulta:

$$A = -1$$

Análogamente:

$$B = 1$$

Entonces:

$$h(z) = \frac{1}{1-2z} - \frac{1}{1-z} \tag{16.5}$$

Usando la fórmula de series geométricas, es sabido que:

$$\frac{1}{1-\alpha z} = \sum_{n\geq 0} \alpha^n z^n \tag{16.6}$$

Luego, si reemplazamos $h(z)$ por (16.1) en (16.5) y las fracciones de (16.5) por las obtenidas en (16.6) se obtiene:

$$\sum_{n\geq 0} T(n) z^n = \sum_{n\geq 0} 2^n z^n - \sum_{n\geq 0} z^n$$

$$T(n) = 2^n - 1$$

Para demostrar que esta solución es óptima, procedemos por inducción sobre n, el número de discos.

Base: Si hay un solo disco, hacemos $2^1 - 1 = 1$ movidas. Esto claramente es óptimo.

Inducción: Supongamos que para mover k discos se requieren $2^k - 1$ movidas, y consideremos mover $k+1$ discos. El disco mayor no puede ayudar en el proceso, siempre está abajo. Para poder moverlo, deberemos mover k discos de A a B, luego mover el disco mayor de A a C, finalmente mover los que están en B a C. Esto suma:

$$(2^k - 1) + 1 + (2^k - 1) = 2^{k+1} - 1$$

Por inducción, se requieren $2^n - 1$ movidas para todo $n \in \mathbb{N}$.

Como la cota inferior es exactamente lo que da nuestro algoritmo, éste es óptimo.

16.2. Mergesort

Queremos ordenar $A[1,\ldots,n]$. Nuestro pseudocódigo para mergesort es el del algoritmo 16.2. De la misma forma que lo hicimos con las torres de Hanoi: ¿Cuántas operaciones se requieren? Sea

Algoritmo 16.2: Mergesort

```
procedure mergesort(A[1,...,n])
    if n > 1 then
        mergesort(A[1,...,⌊n/2⌋])
        mergesort(A[⌊n/2⌋ + 1, n])
        merge(A[1,...,⌊n/2⌋], A[⌊n/2⌋ + 1, n])
    end
```

$M(n)$ el número de operaciones para completar el ordenamiento. Es claro que:

$$\begin{aligned} M(0) &= M(1) \\ M(n) &= M(\left\lfloor \frac{n}{2} \right\rfloor) + M(\left\lceil \frac{n}{2} \right\rceil) + n \end{aligned} \tag{16.7}$$

Supongamos $n = 2^k$. Reemplazando sobre (16.7) vamos obteniendo:

$$\begin{aligned} M(2^k) &= M(\left\lfloor \frac{2^k}{2} \right\rfloor) + M(\left\lceil \frac{2^k}{2} \right\rceil) + 2^k \\ &= M(\left\lfloor 2^{k-1} \right\rfloor) + M(\left\lceil 2^{k-1} \right\rceil) + 2^k \end{aligned} \tag{16.8}$$

Es claro que $2^{k-1} \in \mathbb{N}$, por lo tanto:

$$M(\left\lfloor 2^{k-1} \right\rfloor) = M(\left\lceil 2^{k-1} \right\rceil) = M(2^{k-1}) \tag{16.9}$$

Luego, reemplazando (16.9) en (16.8) obtenemos:

$$M(2^k) = 2M(2^{k-1}) + 2^k \tag{16.10}$$

Sea $m(k) = M(2^k)$. Entonces, reemplazando sobre (16.10) y ajustando índices se tiene:

$$m(k+1) = 2m(k) + 2^{k+1} \qquad m(0) = 0 \tag{16.11}$$

Definiendo:

$$g(z) = \sum_{k \geq 0} m(k) z^k$$

por las propiedades de funciones generatrices ordinarias tenemos:

$$\frac{g(z) - m(0)}{z} = 2g(z) + 2\frac{1}{1-2z}$$

En forma de fracciones parciales, esto es:

$$\begin{aligned} g(z) &= \frac{2z}{(1-2z)^2} \\ &= \frac{1}{(1-2z)^n} - \frac{1}{1-2z} \end{aligned}$$

De acá obtenemos:

$$\begin{aligned} m(k) &= (-1)^k \binom{-2}{k} 2^k - 2^k \\ &= (k+1)2^k - 2^k \\ &= k \cdot 2^k \end{aligned}$$

Volvemos a nuestras variables originales:

$$M(n) = n \log_2 n$$

Dentro de un factor constante, mergesort es óptimo.

Ejercicios

1. Una solución no recursiva al problema de torres de Hanoi es como sigue. Nombre las posiciones como A, B y C, con los discos originalmente en A y deben quedar en C. Si el número de discos es par, repita lo siguiente hasta completar la tarea:

   - Haga la movida legal entre A y B (en cualquier dirección)
   - Haga la movida legal entre A y C (en cualquier dirección)
   - Haga la movida legal entre B y C (en cualquier dirección)

   Si el número de discos es impar, repita lo siguiente hasta completar la tarea:

   - Haga la movida legal entre A y C (en cualquier dirección)
   - Haga la movida legal entre A y B (en cualquier dirección)
   - Haga la movida legal entre B y C (en cualquier dirección)

   a) Determine el número de movidas para mover n discos.

   b) Demuestre que esta estrategia resuelve el problema.

Clase 17

Backtracking

Otra estrategia recursiva... La idea es ir construyendo la solución incrementalmente, explorando distintas ramas y volviendo atrás (*backtrack*) si resulta que un camino es sin salida.

Ejemplo 17.1 (Un clásico)**.** En el ajedrez la reina es la pieza más poderosa. Amenaza los casilleros en su fila y columna, y los ubicados en diagonal. La figura 17.1 muestra los casilleros que amenaza una reina en el ajedrez. Un problema clásico es determinar si se pueden ubicar ocho reinas en el tablero

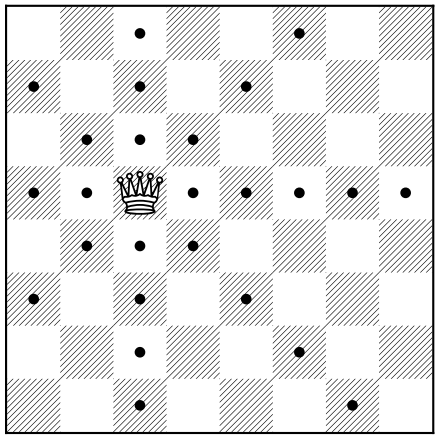


Figura 17.1 – Los casilleros amenazados por una reina

de manera que ninguna pueda amenzar a otra. Claramente no pueden ser más de ocho, puede haber a lo más una reina por columna.

Pasos:

- Reducir espacio de búsqueda. En este caso, si son 8 reinas, hay una por columna. Por lo tanto, llenar por columnas, solución (parcial) indica las filas de las reinas ya ubicadas.

- Ordenar avance.
- Subproblemas similares.

En este caso:

- Ubicar reina en columna 1, 2, ...
- Registrar filas libres (para omitir ocupadas al ubicar la siguiente reina)
- Registrar diagonales libres.

  Reina en r, c:

  $y - c = 1 \cdot (x - r)$
  $r - c = x - y$ es constante
  $y - c = -1 \cdot (x - r)$
  $c - r = x + y$ es constante

Por ejemplo, luego de ubicadas las primeras tres reinas en las filas 1, 3 y 5 la configuración resultante es la de la figura 17.2. Vemos que las filas y diagonales amenazadas por estas tres restringen

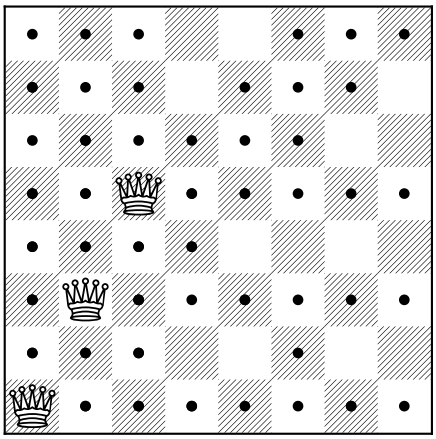


Figura 17.2 – Configuración con tres reinas

muchísimo las posiciones viables para la cuarta y siguientes. Con estas tres reinas, para la cuarta reina quedan solo 3 posibilidades.

Elegimos Python (!), en Python los arreglos tienen índices desde 0, rango de r, c es $0, \ldots, 7$. Interesan los rangos de:

$\boldsymbol{r - c}$**:** Rango es $-7, \ldots, 7$, sumar 7 para llevar al rango $0, \ldots, 14$.

$\boldsymbol{r + c}$**:** Rango es $0, \ldots, 14$

El programa final es el del listado 17.1.

```
#!/usr/bin/python3

queen = [0 for c in range(8)]          # Row of queen in column c
rfree = [True for r in range(8)]       # Row r free
du = [True for i in range(15)]         # Diagonal i = r + 7 - c
dd = [True for i in range(15)]         # Diagonal i = r + c free

def solve(n):
    global solutions

    if n == 8:
        solutions += 1
        print(solutions, end = ": ")
        for r in range(8):
            print(queen[r] + 1, end = " " if r < 7 else "\n")
    else:
        for r in range(8):
            if rfree[r] and dd[n + r] and du[n + 7 - r]:
                queen[n] = r
                rfree[r] = dd[n + r] = du[n + 7 - r] = False
                solve(n + 1)
                rfree[r] = dd[n + r] = du[n + 7 - r] = True

solutions = 0
solve(0)

print()
print(solutions, "solutions")
```

Listado 17.1 – Ocho reinas en Python

La figura 17.3 muestra una de las 92 soluciones.

Ejercicios

1. Una *rotulación graciosa* de un grafo $G = (V, E)$ con n vértices asigna los rótulos 1 a n a los vértices, tal que al rotular los arcos con el valor absoluto de la diferencia entre los rótulos de sus vértices los arcos están rotulados con los números de 1 a $n-1$, cada uno exactamente una vez. Es claro que esto solo se puede hacer en árboles.

   Escriba un programa para determinar cuántas rotulaciones graciosas tiene el grafo P_n.

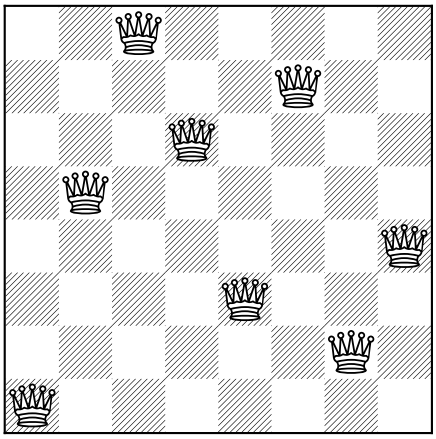


Figura 17.3 – Una solución para el problema de 8 reinas.

Clase 18

Backtracking (continuación)

Clase basada en la presentación sobre backtracking de Steven Skiena [2]. El marco general es el de una secuencia de decisiones, que registramos en un vector $\mathbf{a}_k$. La idea es que una vez decididos $a_1, \ldots, a_{k-1}$, tenemos un conjunto de posibles continuaciones S_k (que generalmente dependerá de las decisiones anteriores). Exploramos las distintas decisiones (y sus posibles continuaciones) por turno. El algoritmo genérico es el 18.1. La idea de backtracking es usarlo cuando no tengamos sub-

Algoritmo 18.1: Esquema de backtracking

```
procedure backtrack(a, k)
    if a es solución then
        Imprima a
    end
    k ← k + 1
    Calcule S_k
    while S_k ≠ ∅ do
        a_k ← elija un elemento de S_k
        S_k ← S_k ∖ {a_k}
        backtrack(a, k)
    end
```

problemas similares si cambiamos la iteración. Por ejemplo, en programación dinámica se originan muchos subproblemas similares entre distintas ramas del árbol que se genera... backtracking no es lo ideal para estos casos (en el caso de las 8 reinas, no se originan subproblemas similares cuyas soluciones puedan reusarse).

18.1. Sodoku

En Sudoku se plantea una grilla de 9×9 casillas a ser llenadas con los dígitos 1 a 9 de forma que cada fila, columna y subcuadrado de 3×3 contenga cada dígito exactamente una vez. Un ejemplo de Sudoku es el dado en la figura 18.1. Estrategias para elegir siguiente casillero a llenar:

- Al azar/primero libre.

		1 2
	3 5	
6		7
7		3
	4	8
1		
	1 2	
8		4
5		6

(a) Problema

6	7	3	8	9	4	5	1	2
9	1	2	7	3	5	4	8	6
8	4	5	6	1	2	9	7	3
7	9	8	2	6	1	3	5	4
5	2	6	4	7	3	8	9	1
1	3	4	5	8	9	2	6	7
4	6	9	1	2	8	7	3	5
2	8	7	3	5	6	1	4	9
3	5	1	9	4	7	6	2	8

(b) Solución

Figura 18.1 – Sudoku muy difícil

- Más restringido.

Estrategias para podar:

- Cuenta local (revisar si quedan opciones fila/columna/cuadrante)
- *Look ahead* (revisar si quedan casilleros sin opciones)

Skiena [3] reporta resultados del cuadro 18.1 para distintos problemas. El simple es de los que típicamente se dan para principiantes, el mediano se planteó en un campeonato (y ninguno de los participantes pudo resolverlo), el difícil es el de la figura 18.1 (es el con menos posiciones llenas que se sabe tiene una única solución). El programa que obtiene estos valores es [1].

Combinaciones	**Criterio de Poda**	**Simple**	**Mediano**	**Difícil**
Azar	Local	1 904 832	863 305	No ...
Azar	Look ahead	127	142	12 507 212
Restringida	Local	48	84	1 243 838
Restringida	Look ahead	48	65	10 374

Cuadro 18.1 – Rendimiento de variantes de backtracking en Sudoku

Bibliografía

[1] Steven S. Skiena: *A backtracking program to solve Sudoku.* http://www3.cs.stonybrook.edu/~skiena/algorist/book/programs/sudoku.c, August 2006.

[2] Steven S. Skiena: *Backtracking.* http://www3.cs.stonybrook.edu/~algorith/video-lectures/2007/lecture15.pdf, 2007.

[3] Steven S. Skiena: *The Algorithm Design Manual.* Springer, second edition, 2008.

Clase 19

Búsqueda en Grafos

Hay una variedad de situaciones en las que se debe explorar algún grafo en busca de un vértice (nodo) meta. Las técnicas son variantes de *backtracking*. Exploraremos algunas de ellas.

En los casos de interés, el grafo a recorrer no se conoce explícitamente, dado un vértice en él se generan descendientes (vecinos) bajo demanda. La estructura general es como describen Dechter y Pearl [1].

19.1. Branch and Bound

Bajo este rótulo se agrupan muchas técnicas diferentes para buscar un nodo óptimo en el grafo, basadas en la idea de *generar* nuevas opciones (esto es *branch*) que se evalúan para *podar* ramas que nacen de opciones que se sabe no llevan a la meta (la evaluación provee el *bound*).

Formalmente, buscamos el nodo en el grafo $G = (V, E)$ que minimiza la función $f\colon V \to \mathbb{R}$. Suponemos que hay una función $\Gamma\colon V \to 2^V$ que, dado un nodo entrega los vecinos relevantes. Contamos con una función cota $g\colon V \to \mathbb{R}$ tal que $g(x) < \min_{y \text{ alcanzable de } x} f(y)$.

Si se revisa el algoritmo 19.1, es una versión modificada de las rutinas no recursivas de recorrido de grafos. Considera una cola Q (puede ser una cola de prioridad, donde puede tener sentido ordenar por el valor de g u otra función que refleja la probabilidad de hallar la solución desde ese nodo) y una cota B, el valor de la mejor solución hallada hasta el momento. En la práctica se agregará la restricción que Γ no genere nodos anteriores para que no entre en un ciclo (por ejemplo, el grafo es un árbol o al menos un DAG).

19.2. El algoritmo A^*

Un algoritmo genérico de búsqueda es A^*, desarrollado inicialmente para planificar rutas de robots moviéndose en un ambiente con obstáculos. Sus proponentes, Hart, Nilsson y Raphael [2] demuestran que es óptimo en el sentido que discutiremos. Dechter y Pearl [1] discuten esquemas de búsqueda en situaciones generalizadas, que incluyen el nuestro como caso muy particular, y discuten optimalidad de A^*.

Suponemos un grafo dirigido $G = (V, E)$, con una función de costo de los arcos $c\colon E \to \mathbb{R}$, donde suponemos que siempre $c(e) \geq \delta$, donde nos dan un conjunto de *fuentes* $S \subset V$, un conjunto de *metas* $T \subset V$ y un operador *sucesor* $\Gamma\colon V \to 2^V$ (vale decir, el grafo está dado en forma implícita solamente;

Algoritmo 19.1: Esquema de *Branch and Bound*

```
function BB
    B ← ∞

    x ← initial guess
    enqueue(Q, x)

    while Q not empty do
        x ← dequeue(Q)
        if f(x) < B then
            B ← f(x)
        end
        for v ∈ Γ(x) do
            if g(v) < B then
                enqueue(Q, v)
            end
        end
    end
    return B
```

suponemos además que cuando Γ nos entrega los descendientes de v simultáneamente entrega los costos desde el nodo actual a cada uno de los vecinos). Nótese que *no* estamos suponiendo que G es finito, suponemos eso sí que el número de nodos vecinos (descendientes) es siempre finito (a un grafo con esta propiedad le llaman *localmente finito*). El subgrafo G_v es el nodo v junto con todos sus descendientes. Dado un nodo fuente $s \in S$ nos interesa hallar en G_s el nodo $t \in T$ que minimiza el costo del camino (la suma de los costos de los arcos) de s a t. Al costo mínimo de un camino de u a v lo anotaremos $h(u,v)$, para abreviar escribiremos $h(v)$ para $h(s,v)$.

Diremos que un algoritmo es *admisible* si garantiza hallar un camino óptimo de s a una meta para todo grafo como descrito. Algoritmos admisibles podrán expandir diferentes nodos, o hacerlo en distinto orden. Interesa que el algoritmo expanda el mínimo número de nodos. Expandir nodos que se sabe que no pueden estar en un camino óptimo es desperdiciar esfuerzo, mientras ignorar nodos que están en un camino óptimo puede hacer que no lo encuentre y no ser admisible. Supondremos una *función de evaluación* $\widehat{f}\colon V \to \mathbb{R}$, de manera de expandir a continuación el nodo de mínimo valor de $\widehat{f}$. Esto sugiere el algoritmo 19.2, que contempla una cola de prioridad Q. Diremos que nodos en Q están *abiertos*, y marcaremos ciertos nodos como *cerrados* para no considerarlos nuevamente. En realidad nos interesa el camino de s a T, la modificación obvia es registrar el nodo padre de v cuando lo marcamos cerrado (y corregirlo al volverlo a cerrar), finalmente seguimos la lista desde el nodo meta hacia atrás para reconstruir el camino buscado.

19.2.1. La función de evaluación

Para el subgrafo G_s sea $f(v)$ el costo óptimo de un camino de s a T, con la restricción que el camino pase por v. Note que $f(s) = h(s)$, que $f(v) = f(s)$ para todo v en un camino óptimo, y que $f(v) > f(s)$ si v no está en un camino óptimo. No conocemos f (determinar su valor es precisamente el objetivo del ejercicio), pero es razonable usar una estimación de f como función de evaluación $\widehat{f}$ en el algoritmo 19.2.

Algoritmo 19.2: El algoritmo A^*

```
function A*
    enqueue(Q, s, f̂(s))
    while Q not empty do
        v ← dequeue(Q)
        Mark v closed
        if v ∈ T then
            return v
        end
        for u ∈ Γ(v) do
            if u is not closed then
                enqueue(Q, u, f̂(u))
            else if new f̂(u) less than old value then
                Remove closed mark from u
                enqueue(Q, u, f̂(u))
            end
        end
    end
```

Podemos escribir f como una suma:

$$f(v) = g(v) + h(v) \tag{19.1}$$

donde $g(v)$ es el costo óptimo de un camino de s a v mientras $h(v)$ es el costo óptimo de un camino de v a T. Dadas estimaciones de g y h podemos calcular una aproximación a f. Sea $\widehat{g}$ una estimación de g, un valor obvio es el costo del camino más corto hallado entre s y v hasta el momento, lo que implica $\widehat{g}(v) \geq g(v)$. El siguiente punto es una estimación de h, que llamaremos $\widehat{h}$. Dependiendo del problema, podremos definir funciones $\widehat{h}$ apropiadas, por el momento demostramos que si $\widehat{h}(v) \leq h(v)$, el algoritmo 19.2 es admisible.

Lema 19.1. *Para un nodo no cerrado v y un camino óptimo P de s a v, hay un nodo abierto v' en P con $\widehat{g}(v') = g(v')$.*

Demostración. Sea $P = \langle s = v_0, \ldots, v_k = v \rangle$. Si s está abierto (no se ha completado ninguna iteración), sea $s = v'$, con lo que $\widehat{g}(s) = g(s) = 0$, y el lema se cumple trivialmente. Supongamos ahora que s está cerrado, sea Δ el conjunto de nodos cerrados v_i en P para los que $\widehat{g}(v_i) = g(v_i)$. Sabemos que $\Delta \neq \varnothing$, ya que $s \in \Delta$. Sea v^* el elemento de Δ con máximo índice, donde $v^* \neq v$ porque v está abierto. Sea v' el sucesor de v^* en P. Entonces:

$\widehat{g}(v') \leq \widehat{g}(v^*) + c(v^* v')$	por definición de $\widehat{g}$
$\widehat{g}(v^*) = g(v^*)$	porque $v^* \in \Delta$
$g(v') = g(v^*) + c(v^* v')$	dado que P es óptimo

Concluimos que $\widehat{g}(v') \leq g(v')$, como $\widehat{g}(v') \geq g(v')$ resulta $\widehat{g}(v') = g(v')$, y por la definición de Δ, v' está abierto. □

Corolario 19.2. *Suponga que para todo v es $\widehat{h}(v) < h(v)$, y que A^* no ha terminado. Entonces para todo camino óptimo P de s a T hay un nodo abierto $v' \in P$ con $\widehat{f}(v') \leq f(s)$.*

Demostración. Por el lema, hay un nodo abierto $v' \in P$ con $\widehat{g}(v') = g(v')$, con lo que por la definición de $\widehat{f}$:

$$\begin{aligned}\widehat{f}(v') &= \widehat{g}(v') + \widehat{h}(v') \\ &= g(v') + \widehat{h}(v') \\ &\leq g(v') + h(v') \\ &= f(v')\end{aligned}$$

Como P es óptimo, $f(v') = f(s)$ para todo $v' \in P$. □

Estamos en condiciones de demostrar:

Teorema 19.3. *Si para todo $v \in V$ es $\widehat{h}(v) \leq h(v)$, A^* es admisible.*

Demostración. La demostración es por contradicción. Hay dos casos a considerar:

No termina: Sea $t \in T$, alcanzable desde s en un número finito de pasos con costo mínimo $f(s)$. Como el costo de cada arco es a lo menos δ, se alcanza t en a lo más $M = f(s)/\delta$ pasos, y para todos los vértices v más lejos de s que M es:

$$\widehat{f}(v) \geq \widehat{g}(v) \geq g(v) \geq M\delta$$

O sea, ningún nodo a distancia mayor a M de s se expande, ya que por el corolario 19.2 habrá un nodo abierto v'en un camino óptimo tal que $\widehat{f}(v') \leq f(s) < f(v)$. El algoritmo eligirá v' en vez de v. Hay un número finito de nodos a distancia a lo más M, cada uno de ellos puede ser reabierto solo un número finito de veces, ya que hay un número finito de caminos que pasan por él, y se reabre solo si calculamos un $\widehat{g}(v)$ menor.

Entrega un camino no óptimo: Supongamos que A^* termina en el nodo t con $\widehat{f}(t) = \widehat{g}(t) > f(s)$. Pero por el corolario 19.2 había un nodo abierto v' en un camino óptimo con $\widehat{f}(v') \leq f(s) < \widehat{f}(t)$. Se habría elegido v' para ser expandido en vez de t, con lo que A^* no habría terminado.

□

19.2.2. Optimalidad de A^*

Hemos demostrado que si $\widehat{h}(v) \leq h(v)$, A^* es admisible. Una cota inferior obvia es $\widehat{h}(v) = 0$, con lo que A^* es ciego (el resultado es esencialmente el algoritmo de Dijkstra). Muchos problemas ofrecen cotas inferiores mejores, que restringen los nodos a ser considerados. Por ejemplo, en el problema original de movimiento de un robot en un área con obstáculos, una cota inferior a la distancia a recorrer es la distancia entre dos puntos, obviando los obstáculos. En general, si omitimos algunas de las restricciones del problema, obtendremos un costo no mayor, o sea un valor admisible de $\widehat{h}(v)$.

Resulta que A^* es óptimo, en el sentido que expande el mínimo número de nodos entre todos los algoritmos que usan la misma información (la misma cota $\widehat{h}$). Esto porque un algoritmo que *no* expanda todos los nodos con $\widehat{f}(v) < f(s)$ para la meta s puede omitir el camino óptimo.

Diremos que la estimación $\widehat{h}$ cumple la condición de *monotonía*:

$$h(u,v) + \widehat{h}(u) \geq \widehat{h}(v) \tag{19.2}$$

La condición (19.2) expresa que la estimación $\widehat{h}(v)$ no puede mejorarse usando datos correspondientes de otros nodos.

Resulta que si $\widehat{h}$ cumple monotonía, nunca se reconsideran nodos.

Lema 19.4. *Suponga que se cumple la condición de monotonía* (19.2)*, y que A^* cerró el nodo v. Entonces $\widehat{g}(v) = g(v)$.*

Demostración. Por contradicción. Considere el subgrafo G_s justo antes de cerrar v, y suponga que $\widehat{g}(v) > g(v)$. Sea P un camino óptimo de s a v, como $\widehat{g}(v) > g(v)$ el algoritmo no lo encontró. Por el lema 19.1, hay un nodo abierto $v' \in P$ con $\widehat{g}(v') = g(v')$. Por suposición, $v \neq v'$, con lo que:

$$\begin{aligned} g(v) &= g(v') + h(v'v) \\ &= \widehat{g}(v') + h(v'v) \end{aligned}$$

Vale decir:

$$\widehat{g}(v) > \widehat{g}(v') + h(v'v)$$

Sumando $\widehat{h}$ a ambos lados:

$$\widehat{g}(v) + \widehat{h}(v) > \widehat{g}(v') + h(v'v) + \widehat{h}(v')$$

Aplicando (19.2) al lado derecho:

$$\widehat{g}(v) + \widehat{h}(v) > \widehat{g}(v') + \widehat{h}(v')$$

Por la definición de $\widehat{f}$:

$$\widehat{f}(v) > \widehat{f}(v')$$

Pero en tal caso A^* hubiese expandido v', que estaba disponible, en vez de v. □

19.3. Juegos

Consideremos un juego en que compiten dos jugadores, A y B, que juegan alternadamente. A cada posición (o estado) del juego se le asigna un valor, que indica qué tan buena es para el jugador. Claramente, cada jugador hará la movida que maximice el valor mínimo resultando de las posibles movidas siguientes del oponente. Una posibilidad es asignar el valor +1 si es una posición en que A gana inmediatamente, −1 si gana B, y 0 si es empate. En este sentido, A busca maximizar, B busca minimizar.

19.3.1. Min-Max

Generalmente no es posible explorar el árbol completo, y evaluamos posiciones mediante alguna función heurística al llegar a una profundidad máxima. Un posible algoritmo es 19.3, que se invoca como minmax(*inicio*, depth, A) si A es quien abre el juego y queremos explorar hasta depth. Es simple registrar además la movida que da lugar al mejor valor (es la jugada a hacer).

19.3.2. Alpha-Beta

Supongamos que es el turno de A (busca maximizar), analizando posibles jugadas de B (busca minimizar). Si ya conocemos una cota α (hemos visto una movida que garantiza ese valor para A) no tiene sentido continuar explorando un camino si lo mejor que podemos lograr en él es peor, con consideraciones simétricas para B.

Nuestro algoritmo 19.4 mantiene valores α (el mínimo que ya tiene garantizado A que puede obtener) y β (el máximo que ya tiene garantizado B que puede obtener), y los usa para cortar la

Algoritmo 19.3: Algoritmo MinMax

```
function minmax(node, depth, turn)
    if depth = 0 ∨ node is terminal then
        return heuristic value of node
    end
    if turn = A then
        best ← −1
        for child of node do
            v ← minmax(child, depth −1, B)
            best ← max(best, v)
        end
    else
        best ← +1
        for child of node do
            v ← minmax(child, depth −1, A)
            best ← min(best, v)
        end
    end
    return best
```

Algoritmo 19.4: Algoritmo Alpha-Beta

```
function alphabeta(node, depth, α, β, turn)
    if depth = 0 ∨ node is terminal then
        return heuristic value of node
    end
    if turn = A then
        best ← −1
        for child of node do
            v ← alphabeta(child, depth −1, α, β, B)
            best ← max(best, v)
            α ← max(α, best)
            if β ≤ α then
                break
            end
        end
    else
        best ← +1
        for child of node do
            v ← alphabeta(child, depth −1, α, β, B)
            best ← min(best, v)
            β ← min(β, best)
            if β ≤ α then
                break
            end
        end
    end
    return best
```

exploración tempranamente. Se invoca inicialmente como como alphabeta($inicio, -1, +1,$ depth, A) (la única garantía que tiene A es que puede perder, simétricamente B gana). Es obvio registrar con α (respectivamente β) la movida que da lugar a ese valor. Note que el algoritmo 19.4 no especifica el orden en que se exploran los hijos de un nodo, claramente conviene explorar de forma que α aumente rápidamente (β disminuya), porque eso limita las búsquedas. O sea, conviene explorar primero las mejores movidas. En la práctica se usa alguna evaluación heurística para ordenarlos adecuadamente. Knuth y Moore [4] discuten la historia del algoritmo, arguyendo que muchas de las variantes tempranas que se discuten como tal en realidad son algoritmos similares, bastante más limitados; dan una de las primeras descripciones precisas y un análisis de su rendimiento. Pearl [5] demuestra que es óptimo.

Bibliografía

[1] Rina Dechter and Judea Pearl: *Generalized best-first search strategies and the optimality of A**. Journal of the ACM, 32(3):505–536, July 1985.

[2] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael: *A formal basis for the heuristic determination of minimum cost paths*. IEEE Transactions on Systems Science and Cybernetics, 4(2):100–107, July 1968. Correction [3].

[3] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael: *Correction to "A formal basis for the heuristic determination of minimum cost paths"*. ACM SIGART Bulletin, (37):28–29, December 1972.

[4] Donald E. Knuth and Ronald W. Moore: *An analysis of alpha-beta pruning*. Artificial Intelligence, 6(4):293–326, Winter 1975.

[5] Judea Pearl: *The solution for the branching factor of the alpha-beta pruning algorithm and it's optimality*. Communications of the ACM, 25(8):559–564, August 1982.

Clase 20

Dividir y Conquistar

Una de las mejores estrategias para diseñar algoritmos.

Idea: Dado un problema grande, reducirlo a varios problemas menores del mismo tipo, y combinar resultados.

Ejemplo 20.1 (Merge Sort)**.** Para ordenar N elementos:

- Dividir en "mitades" de $\lfloor \frac{N}{2} \rfloor$ y $\lceil \frac{N}{2} \rceil$
- Ordenarlas recursivamente.
- Intercalar resultados.

Ejemplo 20.2 (Búsqueda binaria)**.** Un arreglo ordenado de N elementos, y una clave a buscar. Obtener el elemento en la posición $\lfloor \frac{N}{2} \rfloor$, buscar en la mitad que tiene que contener la clave

Ejemplo 20.3 (Multiplicación de Karatsuba)**.** Para multiplicar números de $2n$ dígitos, dividimos ambos en mitades [4]:

$$A = 10^n a + b$$
$$B = 10^n c + d$$

con $a, b, c, d < 10^n$.

Además:

$$A \cdot B = 10^{2n} ac + 10^n (ad + bc) + bd \tag{20.1}$$

Notando que:

$$(a+b)\cdot(c+d) = ac + ad + bc + bd - (ad + bc) + ac + bd$$

Podemos calcular los coeficientes de (20.1) con 3 (no 4) multiplicaciones:

$$ac$$
$$bd$$
$$(a+b)(c+d) - ac - bd$$

Ejemplo 20.4. Otro ejemplo de esta estrategia es el algoritmo de Strassen [8] para multiplicar matrices. Consideremos primeramente el producto de dos matrices de 2×2:

$$\begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

Sabemos que:

$$\begin{aligned} c_{11} &= a_{11}b_{11} + a_{12}b_{21} & c_{12} &= a_{11}b_{12} + a_{12}b_{22} \\ c_{21} &= a_{21}b_{11} + a_{22}b_{21} & c_{22} &= a_{21}b_{12} + a_{22}b_{22} \end{aligned}$$

Esto corresponde a 8 multiplicaciones. Definamos los siguientes productos:

$$\begin{aligned} m_1 &= (a_{11} + a_{22})(b_{11} + b_{22}) & m_2 &= (a_{21} + a_{22})\, b_{11} \\ m_3 &= a_{11}(b_{12} - b_{22}) & m_4 &= a_{22}(b_{21} - b_{11}) \\ m_5 &= (a_{11} + a_{12})\, b_{22} & m_6 &= (a_{21} - a_{11})(b_{11} + b_{12}) \\ m_7 &= (a_{12} - a_{22})(b_{21} + b_{22}) \end{aligned}$$

Entonces podemos expresar:

$$\begin{aligned} c_{11} &= m_1 + m_4 - m_5 + m_7 & c_{12} &= m_3 + m_5 \\ c_{21} &= m_2 + m_4 & c_{22} &= m_1 - m_2 + m_3 + m_6 \end{aligned}$$

Con estas fórmulas se usan 7 multiplicaciones para evaluar el producto de dos matrices. Cabe hacer notar que estas fórmulas no hacen uso de conmutatividad, por lo que son aplicables también para multiplicar matrices de 2×2 cuyos elementos son a su vez matrices. Podemos usar esta fórmula recursivamente para multiplicar matrices de $2^n \times 2^n$.

20.1. Estructura común

Un problema de tamaño n se reduce a a problemas de tamaño n/b, que se resuelven recursivamente y las soluciones se combinan. El siguiente desarrollo toma de Bentley, Haken y Saxe [2] y CLRS [3].

Si el trabajo para resolver una instancia de tamaño n la llamamos $t(n)$, y el trabajo para reducir y combinar soluciones lo llamamos $f(n)$:

$$t(n) = at(n/b) + f(n) \qquad t(1) = t_1 \tag{20.2}$$

La situación que estamos analizando indica que $a \geq 1$, $b > 1$, $f(n) > 0$.

Buscamos resolver esta recurrencia. Suponiendo que n es una potencia de b podemos hacer los cambios de variable indicados y ajustar índices:

$$\begin{aligned} n = b^k \qquad t(b^k) = T(k) \qquad k = \log_b n \\ T(k+1) = aT(k) + f(b^{k+1}) \end{aligned}$$

Definimos la función generatriz ordinaria:

$$g(z) = \sum_{r \geq 0} T(k) z^r$$

Por propiedades:

$$\begin{aligned}\frac{g(z)-t_1}{z} &= ag(z)+\sum_{r\geq 0} f(b^{r+1})z^r \\ g(z) &= \frac{t_1}{1-az}+\frac{z}{1-az}\sum_{r\geq 0} f(b^{r+1})z^r \\ [z^k]g(z) &= t_1 a^k + \sum_{0\leq r\leq k} a^{k-1-r} f(b^{r+1}) \\ &= t_1 a^k + a^k \sum_{1\leq r\leq k+1} a^{-r} f(b^r)\end{aligned}$$

En términos de las variables originales, como:

$$\begin{aligned}a^k &= a^{\log_b n} \\ &= \left(b^{\log_b a}\right)^{\log_b n} \\ &= n^{\log_b a}\end{aligned}$$

resulta:

$$t(n) = t_1 n^{\log_b a} + n^{\log_b a} \sum_{1\leq r\leq 1+log_b n} a^{-r} f(b^r)$$

Si la segunda suma converge cuando n tiende a infinito, vale decir, si $f(n) = O(n^c)$ para algún $c < \log_b a$, vemos que domina el primer término:

$$t(n) = \Theta(n^{\log_b a})$$

En caso que la suma no converja, resulta dominante el segundo término. Analicemos primeramente el caso en que $f(n) = \Omega(n^c)$, donde $c > \log_b a$. Es central la suma:

$$\sum_{1\leq r\leq 1+log_b n} a^{-r} f(b^r)$$

Los términos son positivos y crecen, siendo el último el mayor. O sea, $t(n) = \Omega(f(n)$. Si suponemos además que $af(n/b) \leq kf(n)$ para alguna constante $k < 1$ resulta:

$$\begin{aligned}\sum_{1\leq r\leq 1+log_b n} a^{-r} f(b^r) &= \frac{1}{a^{1+\log_b n}} \sum_{0\leq r\leq log_b n} a^r f(n/b^r) \\ &\leq \frac{1}{an^{\log_b a}} \sum_{0\leq r\leq log_b n} k^r f(n) \\ &\leq \frac{1}{an^{\log_b a}} \frac{f(n)}{1-k}\end{aligned}$$

Esta es una suma geométrica convergente, lo que con la cota inferior anterior se resume en:

$$t(n) = \Theta(f(n))$$

El caso intermedio de mayor interés es $f(n) = \Theta(n^{\log_b a} \log^\alpha n)$:

$$\begin{aligned}\sum_{1\leq r\leq 1+log_b n} a^{-r} f(b^r) &= \sum_{1\leq r\leq 1+log_b n} a^{-r}\Theta(a^r r^\alpha) \\ &= \Theta\left(\sum_{1\leq r\leq 1+log_b n} r^\alpha\right)\end{aligned}$$

Esta suma a su vez converge siempre que $\alpha < -1$, si $\alpha = -1$ es una suma harmónica (sabemos que $H_n \sim \ln n + \gamma$) y si $\alpha > -1$ podemos aproximar por una integral. Esto resulta en:

$$t(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{si } \alpha < -1 \\ \Theta(n^{\log_b a} \log\log n) & \text{si } \alpha = -1 \\ \Theta(n^{\log_b a} \log^{\alpha+1} n) & \text{si } \alpha > -1 \end{cases}$$

Uniendo todas las piezas, tenemos:

Teorema 20.1 (Teorema Maestro)**.** *La recurrencia:*

$$t(b) = a\,t(n/b) + f(n) \qquad t(1) = t_1 \tag{20.3}$$

con constantes $a \geq 1$, $b > 1$, $f(n) > 0$ tiene solución:

$$\begin{aligned} t(n) &= \Theta(n^{\log_b a}) && \text{si } f(n) = O(n^c) \text{ para } c < \log_b a \\ &= \Theta(n^{\log_b a}) && \text{si } f(n) = \Theta(n^{\log_b a} \log^{\alpha} n) \text{ con } \alpha < -1 \\ &= \Theta(n^{\log_b a} \log\log n) && \text{si } f(n) = \Theta(n^{\log_b a} \log^{\alpha} n) \text{ con } \alpha = -1 \\ &= \Theta(n^{\log_b a} \log^{\alpha+1} n) && \text{si } f(n) = \Theta(n^{\log_b a} \log^{\alpha} n) \text{ con } \alpha > -1 \\ &= \Theta(f(n)) && \text{si } f(n) = \Omega(n^c) \text{ con } c > \log_b a \text{ y } a f(n/b) < k f(n) \text{ para } n \text{ grande y } k < 1 \end{aligned}$$

Nuestros ejemplos se resumen en el cuadro 20.1. En los casos límite (mergesort y búsqueda binaria) tenemos $\alpha = 0 > -1$, que es lejos lo más común en la práctica. Una variante de esto es el

Algoritmo	a	b	$f(n)$	$t(n)$
Mergesort	2	2	n	$\Theta(n \log n)$
Búsqueda binaria	1	2	1	$\Theta(\log n)$
Karatsuba	3	2	n	$\Theta(n^{\log_2 3})$
Strassen	7	2	n^2	$\Theta(n^{\log_2 7})$

Cuadro 20.1 – Complejidad de nuestros ejemplos

teorema de Akra-Bazzi, del que reportamos la versión de Leighton [5].

Teorema 20.2 (Akra-Bazzi)**.** *Sea una recurrencia de la forma:*

$$T(z) = g(z) + \sum_{1 \leq k \leq n} a_k T(b_k z + h_k(z)) \quad \text{para } z \geq z_0$$

donde z_0, a_k y b_k son constantes, sujeta a las siguientes condiciones:

- *Hay suficientes casos base.*
- *Para todo k se cumplen $a_k > 0$ y $0 < b_k < 1$.*
- *Hay una constante c tal que $|g(z)| = O(z^c)$.*
- *Para todo k se cumple $|h_k(z)| = O(z/(\log z)^2)$.*

Entonces, si p es tal que:

$$\sum_{1 \leq k \leq n} a_k b_k^p = 1$$

la solución a la recurrencia cumple:

$$T(z) = \Theta\left(z^p \left(1 + \int_1^z \frac{g(u)}{u^{p+1}}\,\mathrm{d}u\right)\right)$$

Frente a nuestro tratamiento tiene la ventaja de manejar divisiones desiguales (b_k diferentes), y explícitamente considera pequeñas perturbaciones en los términos, como lo son aplicar pisos o techos, a través de los $h_k(z)$. Diferencias con pisos y techos están acotados por una constante, mientras la cota del teorema permite que crezcan. Por ejemplo, la recurrencia correcta para el número de comparaciones en ordenamiento por intercalación es:

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n - 1$$

El teorema de Akra-Bazzi es aplicable. La recurrencia es:

$$T(n) = T(n/2 + h_+(n)) + T(n/2 + h_-(n)) + n - 1$$

Acá $|h_\pm(n)| \leq 1/2$, además $a_\pm = 1$ y $b_\pm = 1/2$. Tenemos $|g(z)| = z - 1 = O(z)$. Estos cumplen las condiciones del teorema, de:

$$\sum_{1 \leq k \leq 2} a_k b_k^p = 1$$

resulta $p = 1$, y tenemos la cota:

$$T(z) = \Theta\left(z\left(1 + \int_1^z \frac{u-1}{u^2}\,\mathrm{d}u\right)\right) = \Theta(z \ln z + 1) = \Theta(z \log z)$$

Otro ejemplo son los árboles de búsqueda aleatorizados (*Randomized Search Trees*, ver por ejemplo Aragon y Seidel [1], Martínez y Roura [6] y Seidel y Aragon [7]) en uno de ellos de tamaño n una búsqueda toma tiempo aproximado:

$$T(n) = \frac{1}{4} T(n/4) + \frac{3}{4} T(3n/4) + 1$$

Nuevamente es aplicable el teorema 20.2, de:

$$\frac{1}{4}\left(\frac{1}{4}\right)^p + \frac{3}{4}\left(\frac{3}{4}\right)^p = 1$$

obtenemos $p = 0$, y por tanto la cota

$$T(z) = \Theta\left(z^0\left(1 + \int_1^z \frac{\mathrm{d}u}{u}\right)\right) = \Theta(\log z)$$

Una variante útil del teorema maestro, pero de demostración bastante engorrosa, es la que presenta Yap [9].

Ejercicios

1. Para cierto problema cuenta con tres algoritmos alternativos:

   Algoritmo A: Resuelve el problema dividiéndolo en cinco problemas de la mitad del tamaño, los resuelve recursivamente y combina las soluciones en tiempo lineal.

   Algoritmo B: Resuelve un problema de tamaño n resolviendo recursivamente dos problemas de tamaño $n-1$ y combina las soluciones en tiempo constante.

   Algoritmo C: Divide el problema de tamaño n en nueve problemas de tamaño $n/3$, resuelve los problemas recursivamente y combina las soluciones en tiempo $O(n^2)$.

¿Cuál elije si n es grande, y porqué?

2. En un arreglo a se dice que la posición i es un *mínimo local* si $a[i]$ es menor a sus vecinos, o sea $a[i-1] > a[i]$ y $a[i] < a[i+1]$. Decimos además que 0 es un mínimo local si $a[0] < a[1]$, y que lo es $n-1$ si $a[n-2] > a[n-1]$ (los extremos tienen un único vecino). Dado un arreglo a de n números distintos, diseñe un algoritmo eficiente basado en dividir y conquistar para hallar un mínimo local (pueden haber varios). Justifique su algoritmo, y derive su complejidad aproximada.

3. Se dan k listas ordenadas de n elementos, y se pide intercalarlas para obtener una única lista ordenada. Considere que el costo de intercalar es proporcional al largo del resultado. Analice las siguientes alternativas:

   a) Intercalar las primeras dos listas, intercalar el resultado con la tercera, y así sucesivamente hasta terminar el trabajo.

   b) Usar un esquema de dividir y conquistar.

Bibliografía

[1] Cecilia R. Aragon and Raimund G. Seidel: *Randomized search trees.* In *Thirtieth Annual Symposium on Foundations of Computer Science*, pages 540–545, October – November 1989.

[2] Jon Louis Bentley, Dorothea Haken, and James B. Saxe: *A general method for solving divide-and-conquer recurrences.* ACM SIGACT News, 12(3):36–44, Fall 1980.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald R. Rivest, and Clifford Stein: *Introduction to Algorithms.* MIT Press, third edition, 2009.

[4] A. Karatsuba and Yu. Ofman: *Multiplication of many-digital numbers by automatic computers.* Proceedings of the USSR Academy of Sciences, 145:293–294, 1962.

[5] Tom Leighton: *Notes on better master theorems for divide-and-conquer recurrences.* http://citeseer.ist.psu.edu/252350.html, 1996.

[6] Conrado Martínez and Salvador Roura: *Randomized binary search trees.* Journal of the ACM, 45(2):288–323, March 1998.

[7] Raimund G. Seidel and Cecilia A. Aragon: *Randomized search trees.* Algorithmica, 16(4/5):464–497, October 1996.

[8] Volker Strassen: *Gaussian elimination is not optimal.* Numerische Mathematik, 13(4):354–356, August 1969.

[9] Chee K. Yap: *A real elementary approach to the master recurrence and generalizations.* In Mitsunori Ogihara and Jun Tarui (editors): *Theory and Applications of Models of Computation: 8th Annual Conference*, volume 6648 of *Lecture Notes in Computer Science*, pages 14–26, Tokyo, Japan, May 2011. Springer.

Clase 21

Diseño de Algoritmos

Discutiremos un problema planteado por Bentley [1]. Dado el arreglo $a[n]$, hallar la máxima suma de un rango:

$$\max_{i,j}\left\{\sum_{i\le k\le j} a[k]\right\} \tag{21.1}$$

Si todos los valores son positivos, la respuesta es obvia: la suma de todos los elementos del arreglo. El punto está si hay elementos negativos: ¿incluimos uno de ellos en la esperanza que los elementos positivos que lo rodean más que lo compensen? Finalmente, acordamos que la suma de un rango vacío es cero, y que en un arreglo de elementos negativos la suma máxima es cero.

21.1. Algoritmo ingenuo

La solución obvia, traducción directa de la especificación dada por la ecuación (21.1), es la mostrada en el listado 21.1.

```
double MaxSum(double a[], int n)
{
        double MaxSoFar;

        MaxSoFar = 0.0;
        for(int i = 0; i < n; i++) {
                for(int j = i; j <= n; j++) {
                        double Sum = 0.0;
                        for(int k = i; k < j; k++)
                                Sum += a[k];
                        MaxSoFar = max(MaxSoFar, Sum);
                }
        }
        return MaxSoFar;
}
```

Listado 21.1 – Algoritmo 1: Versión ingenua

La complejidad del algoritmo 1 es $O(n^3)$. Lo que buscamos es mejorarlo.

21.2. No recalcular sumas

Hay dos ideas sencillas para evitar recalcular sumas.

21.2.1. Extender sumas

En vez de calcular la suma del rango cada vez, extendemos la suma anterior. Esto da el programa del listado 21.2. La complejidad del algoritmo 2 es $O(n^2)$.

```
double MaxSum(double a[], int n)
{
        double MaxSoFar;

        MaxSoFar = 0.0;
        for(int i = 0; i < n; i++) {
                double Sum = 0.0;
                for(int j = i; j < n; j++) {
                        Sum += a[j];
                        MaxSoFar = max(MaxSoFar, Sum);
                }
        }
        return MaxSoFar;
}
```

Listado 21.2 – Algoritmo 2: Evitar recalcular sumas

21.2.2. Sumas cumulativas

Una manera de manejar rangos es usar sumas cumulativas, y obtener el valor para el rango restando. Esta idea da el listado 21.3. La complejidad del algoritmo 3 es $O(n^2)$. En función de nuestro algoritmo original resulta una mejora, pero no respecto a la segunda variante.

21.3. Dividir y Conquistar

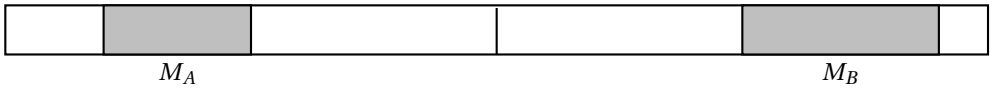
Aplicar la estragia vista la clase pasada lleva a la figura 21.1a. Pero debemos también considerar que el rango con máxima suma esté a hojarcadas, cruzando el punto central, como en la figura 21.1b. El algoritmo es el dado en el listado 21.4. Usando el "teorema maestro", las constantes del algoritmo 4 son $a = 2$, $b = 2$, $d = 1$, por lo tanto la complejidad es $O(n \log n)$.

21.4. Un algoritmo lineal

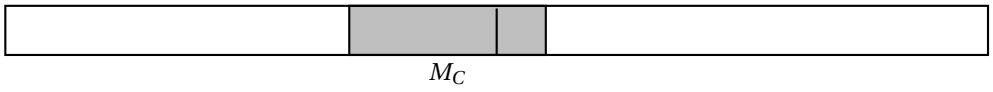
Otro algoritmo resulta de la idea, común al procesar arreglos, de tener una solución parcial hasta $a[i]$, y analizar cómo extenderla para cubrir hasta $a[i+1]$. En nuestro caso, esto significa considerar la máxima suma que llega hasta $a[i]$, y recordar la máxima suma vista hasta ahora, ver la figura 21.2. Esto da el algoritmo 5, del listado 21.5

```
double MaxSum(double a[], int n)
{
        double CumArray[n + 1]; /* Sum of a[0] to a[i - 1] */
        double MaxSoFar;

        CumArray[0] = 0.0;
        for(int k = 0; k < n; k++)
                CumArray[k + 1] = CumArray[k] + a[k];

        MaxSoFar = 0.0;
        for(int i = 0; i < n; i++)
                for(int j = i; j < n; j++)
                        MaxSoFar = max(MaxSoFar,
                                        CumArray[j + 1] - CumArray[i]);

        return MaxSoFar;
}
```

Listado 21.3 – Algoritmo 3: Usar arreglo cumulativo



(a) Máximos en las mitades



(b) Máximo al medio

Figura 21.1 – Dividir y conquistar

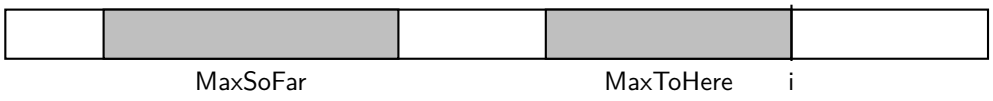


Figura 21.2 – Extender la solución

La complejidad del algoritmo 5 es $O(n)$. Sin embargo, es imposible tener una complejidad menor que n, dado que es necesario revisar cada elemento del arreglo.

Reportar la complejidad de un algoritmo en términos de $O(\cdot)$ es incompleto, pero el cuadro 21.1 muestra su relevancia. La ventaja es que la complejidad en estos términos es sencilla de obtener, en nuestros casos simples (algoritmos 1, 2, 3 y 5) por inspección, el teorema maestro da la complejidad para el algoritmo 4 directamente.

```
static double *aa;

static double msi(int l, int u)
{
        double Sum, MaxToRight, MaxToLeft, MaxCrossing,
               MaxInA, MaxInB;

        if(l >= u) /* Zero-element vector */
                return 0.0;
        if(l == u - 1) /* One-element vector */
                return max(aa[l], 0.0);

        int m = (l + u) / 2;
        /* Find max crossing to left */
        Sum = MaxToLeft = 0.0;
        for(int i = m - 1; i >= l; i--) {
                Sum += aa[i];
                MaxToLeft = max(MaxToLeft, Sum);
        }
        /* Find max crossing to right */
        Sum = MaxToRight = 0.0;
        for(int i = m; i < u; i++) {
                Sum += aa[i];
                MaxToRight = max(MaxToRight, Sum);
        }
        MaxCrossing = MaxToLeft + MaxToRight;

        MaxInA = msi(l, m);
        MaxInB = msi(m, u);

        return max(MaxCrossing, max(MaxInA, MaxInB));
}

double MaxSum(double a[], int n)
{
        aa = a;

        return msi(0, n);
}
```

Listado 21.4 – Algoritmo 4: Usar sumas cumulativas

```
double MaxSum(double a[], int n)
{
        double MaxSoFar, MaxEndingHere;

        MaxSoFar = MaxEndingHere = 0.0;
        for(int i = 0; i < n; i++) {
                MaxEndingHere = max(MaxEndingHere + a[i], 0.0);
                MaxSoFar = max(MaxSoFar, MaxEndingHere);
        }
        return MaxSoFar;
}
```

Listado 21.5 – Algoritmo 5: Ir extendiendo resultado parcial

Algoritmo		**1**	**2**	**4**	**5**
Líneas de C		8	7	14	7
Tiempo en [μs]		$3{,}4n^3$	$13n^2$	$46n\log n$	$33n$
Tiempo para $n =$	10^2	3,4[s]	130[ms]	30[ms]	3,3[ms]
	10^3	0,94[h]	14[s]	0,45[s]	33[ms]
	10^4	39 días	22[min]	6,1[s]	0,33[s]
	10^5	108 años	1,5 días	1,3[min]	3,3[s]
	10^6	108 millones de años	5 meses	15[min]	33[s]

Cuadro 21.1 – Comparativa de Bentley [1] entre las variantes

Bibliografía

[1] Jon Louis Bentley: *Algorithm design techniques*. Communications of the ACM, 27(9):865–871, September 1984.

Clase 22

Complejidad

22.1. Métodos simples de ordenamiento

Tenemos los métodos de la burbuja (listado 22.1), selección, (listado 22.2), e inserción (listado 22.3). Es fácil ver que todos ellos tienen tiempo de ejecución $O(n^2)$. El método de selección tiene mejor caso $O(n^2)$, los otros dos tienen mejor caso $O(n)$.

```
void
sort(double a[], int n)
{
        int k;

        for (int i = n - 1; i; i = k) {
                k = 0;
                for (int j = 0; j < i; j++) {
                        if (a[j + 1] < a [j]) {
                                double tmp;
                                tmp = a[j]; a[j] = a[j + 1]; a[j + 1] = tmp;

                                k = j;
                        }
                }
                i = k;
        }
}
```

Listado 22.1 – Método de la burbuja

```
void
sort(double a[], int n)

{
        for(int i = 0; i < n; i++) {
```

```
                int imin = i; double min = a[i];
                for(int j = i + 1; j < n; j++)
                        if(a[j] < min) {
                                imin = j; min = a[j];
                        }
                double tmp = a[i]; a[i] = a[imin]; a[imin] = tmp;
        }
}
```

Listado 22.2 – Método de selección

```
void
sort(double a[], int n)
{
        for (int i = 1; i < n; i++) {
                double tmp = a[i];
                int j;
                for (j = i - 1; j >= 0 && tmp < a[j]; j--)
                        a[j + 1] = a[j];
                a[j + 1] = tmp;
        }
}
```

Listado 22.3 – Método de inserción

22.1.1. Rendimiento de métodos simples de ordenamiento

Nos interesa obtener información más detallada que ésta sobre el rendimiento de estos algoritmos. En particular, interesa el tiempo promedio de ejecución. Para ello debemos considerar una distribución de los datos de entrada (valores repetidos, orden original del arreglo). Para simplificar, supondremos que no hay datos repetidos, y como los algoritmos únicamente comparan elementos podemos asumir que la entrada es una permutación de $1,\ldots,n$. O sea, requerimos la distribución de las permutaciones dadas al algoritmo. Esto en general es imposible de conseguir (y engorroso de tratar), así que supondremos que todas las permutaciones son igualmente probables. Esto reduce el análisis detallado a derivar propiedades promedio de las permutaciones.

Definición 22.1. Una *inversión* de la permutación π de $1,\ldots,n$ es un par de índices i, j tales que $i < j$ y $\pi(i) > \pi(j)$.

El número mínimo de inversiones es 0 (el arreglo ordenado no tiene inversiones), el máximo es $n(n-1)/2$ (en el arreglo ordenado de mayor a menor el elemento i participa en $i-1$ inversiones con elementos mayores, sumando para $1 \le i \le n$ se tiene el valor citado).

Es claro que en el método de la burbuja cada intercambio elimina exactamente una inversión. Vale decir, el número de asignaciones de elementos es tres veces el número de inversiones.

El método de inserción funciona esencialmente como el de la burbuja, solo que en vez de intercambiar en cada paso deja un espacio libre en la posición original, copia cada elemento una posición hacia arriba si es mayor que el elemento bajo consideración, moviendo la posición libre hacia abajo; finalmente ubica el elemento en su posición (la que queda libre después de los malabares anteriores). Compare con la figura 22.1. Nuevamente, si suponemos que el espacio vacío eventualmente será ocupado por el valor temporal, cada asignación elimina una inversión. Esto se resume en dos asignaciones para cada elemento, y una asignación adicional para cada inversión.

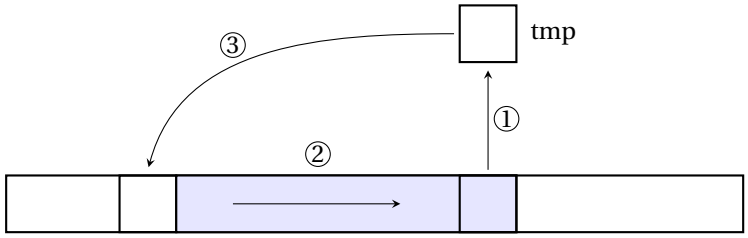


Figura 22.1 – Operación del método de inserción

22.1.2. Funciones generatrices cumulativas

Comparar los métodos es entonces esencialmente obtener el número medio de inversiones en las permutaciones de $1,\ldots,n$. Para ello recurrimos a nuestra técnica preferida, funciones generatrices. De manera muy similar a como contabilizamos las estructuras de un tamaño dado mediante funciones generatrices podemos representar el total de alguna característica. Dividiendo por el número de estructuras del tamaño respectivo tenemos el promedio del valor de interés.

Para precisar, consideremos una clase de objetos $\mathscr{A}$. Como siempre el número de objetos de tamaño n lo anotaremos a_n, con función generatriz:

$$A(z) = \sum_{\alpha\in\mathscr{A}} z^{|\alpha|} \tag{22.1}$$

$$= \sum_{n\geq 0} a_n z^n \tag{22.2}$$

Consideremos no sólo el número de objetos, sino alguna característica, cuyo valor para el objeto α anotaremos $\chi(\alpha)$. Es natural definir la *función generatriz cumulativa*:

$$C(z) = \sum_{\alpha\in\mathscr{A}} \chi(\alpha) z^{|\alpha|} \tag{22.3}$$

Vale decir, los coeficientes son la suma de la medida χ para un tamaño dado:

$$[z^n]C(z) = \sum_{|\alpha|=n} \chi(\alpha) \tag{22.4}$$

Así tenemos el valor promedio para objetos de tamaño n:

$$\mathsf{E}_n[\chi] = \frac{[z^n]C(z)}{[z^n]A(z)} \tag{22.5}$$

La discusión precedente es aplicable si tenemos objetos no rotulados entre manos. Si corresponden objetos rotulados, podemos definir las respectivas funciones generatrices exponenciales:

$$\widehat{A}(z) = \sum_{\alpha\in\mathscr{A}} \frac{z^{|\alpha|}}{|\alpha|!} \tag{22.6}$$

$$= \sum_{n\geq 0} a_n \frac{z^n}{n!} \tag{22.7}$$

$$\widehat{C}(z) = \sum_{\alpha\in\mathscr{A}} \chi(\alpha) \frac{z^{|\alpha|}}{|\alpha|!} \tag{22.8}$$

Nuevamente, como los factoriales en los coeficientes se cancelan:

$$\mathsf{E}_n[\chi] = \frac{[z^n]\widehat{C}(z)}{[z^n]\widehat{A}(z)} \tag{22.9}$$

22.1.3. Análisis de burbuja e inserción

Nuestros objetos de interés son permutaciones, objetos rotulados. Corresponde usar funciones generatrices exponenciales.

Anotemos $\iota(\pi)$ para el número de inversiones de la permutación π, y definamos la función generatriz cumulativa:

$$I(z) = \sum_{\pi \in \mathscr{P}} \iota(\pi) \frac{z^{|\pi|}}{|\pi|!} \tag{22.10}$$

En particular, nos interesa el número promedio de inversiones para permutaciones de tamaño n.

Podemos describir permutaciones mediante la expresión simbólica:

$$\mathscr{P} = \mathscr{E} + \mathscr{P} \star \mathscr{I} \tag{22.11}$$

Vale decir, una permutación es vacía o es una permutación combinada con un elemento adicional. Dada la permutación π construimos permutaciones de largo $|\pi|+1$ añadiendo un nuevo elemento vía la operación $\star$. Si elegimos j como nuevo último elemento, habrán $j-1$ elementos menores que él antes, o sea serán $j-1$ inversiones adicionales. Estamos creando $|\pi|+1$ nuevas permutaciones, cada una de las cuales conserva las inversiones que tiene, y agrega entre 0 y $|\pi|$ nuevas inversiones dependiendo del valor elegido como último. El total de inversiones en el conjunto de permutaciones así creado a partir de π es:

$$(|\pi|+1)\iota(\pi) + \sum_{0 \le k \le |\pi|} k = (|\pi|+1)\iota(\pi) + \frac{|\pi|(|\pi|+1)}{2} \tag{22.12}$$

Con esto tenemos la descomposición para la función generatriz cumulativa (la permutación de cero elementos no tiene inversiones):

$$\begin{aligned} I(z) &= \iota(\epsilon) + \sum_{\pi \in \mathscr{P}} \left((|\pi|+1)\iota(\pi) + \frac{|\pi|(|\pi|+1)}{2} \right) \frac{z^{|\pi|+1}}{(|\pi|+1)!} \\ &= \sum_{\pi \in \mathscr{P}} \iota(\pi) \frac{z^{|\pi|+1}}{|\pi|!} + \frac{1}{2} \sum_{\pi \in \mathscr{P}} \frac{z^{|\pi|+1}}{|\pi|!} |\pi| \end{aligned} \tag{22.13}$$

Como hay $k!$ permutaciones de tamaño k, sumando sobre tamaños resulta:

$$\begin{aligned} &= zI(z) + \frac{1}{2} z \sum_{k \ge 0} k z^k \\ &= zI(z) + \frac{z^2}{2(1-z)^2} \end{aligned} \tag{22.14}$$

Despejando:

$$I(z) = \frac{1}{2} \frac{z^2}{(1-z)^3} \tag{22.15}$$

Obtenemos el número promedio de inversiones directamente, ya que hay $n!$ permutaciones de

tamaño n, y el promedio casualmente es el coeficiente de z^n en la función generatriz exponencial:

$$\begin{aligned}
\mathsf{E}_n[\iota] &= [z^n]I(z) && (22.16)\\
&= \frac{1}{2}[z^n]\frac{z^2}{(1-z)^3}\\
&= \frac{1}{2}[z^{n-2}](1-z)^{-3}\\
&= \frac{1}{2}\binom{-3}{n-2}\\
&= \frac{1}{2}\binom{n-2+(3-1)}{3-1}\\
&= \frac{1}{2}\binom{n}{2}\\
&= \frac{n(n-1)}{4} && (22.17)
\end{aligned}$$

Podemos resumir las anteriores como número asintótico de asignaciones al ordenar n elementos en el cuadro 22.1. Queda claro (salvo para optimistas incurables) que el método de inserción es mejor.

Algoritmo	**Min**	**Prom**	**Máx**
Burbuja	0	$3n^2/4$	$3n^2/2$
Inserción	$2n$	$n^2/4$	$n^2/2$

Cuadro 22.1 – Comparación entre métodos de burbuja e inserción

22.1.4. Análisis de selección

El método de selección tiene sentido si queremos minimizar el número de copias (podemos simplemente copiar y comparar claves, y copiar elementos solo para ubicarlos en su lugar). En tal caso interesa fundamentalmente el número de asignaciones.

Para el método de selección el número de asignaciones está dado por el número de veces que hallamos un elemento mayor, o sea, el número de máximos de izquierda a derecha en la permutación. Si llamamos $\chi(\pi)$ al número de máximos de izquierda a derecha en la permutación π, como el último elemento de la permutación es un máximo de izquierda a derecha si es el máximo de todos ellos, usando la convención de Iverson podemos expresar el número de máximos de izquierda a derecha en la permutación resultante de $\pi \star (1)$ si se asigna el rótulo j al elemento nuevo como:

$$\chi(\pi) + [j = |\pi| + 1] \tag{22.18}$$

con lo que para las permutaciones que se crean de π vía $\pi \star (1)$ serán:

$$(|\pi|+1)\,\chi(\pi) + \sum_{0 \le j \le |\pi|+1} [j = |\pi| + 1] = (|\pi|+1)\,\chi(\pi) + 1 \tag{22.19}$$

Esto da para la fución generatriz cumulativa:

$$M(z) = \sum_{\pi\in\mathscr{P}} \chi(\pi)\frac{z^{|\pi|}}{|\pi|!} \tag{22.20}$$

$$= \chi(\epsilon) + \sum_{\pi\in\mathscr{P}} \left((|\pi|+1)\,\chi(\pi)+1\right)\frac{z^{|\pi|+1}}{(|\pi|+1)!}$$

$$= \sum_{\pi\in\mathscr{P}} \chi(\pi)\frac{z^{|\pi|+1}}{|\pi|!} + \sum_{\pi\in\mathscr{P}} \frac{z^{|\pi|+1}}{(|\pi|+1)!}$$

$$= zM(z) + \sum_{\pi\in\mathscr{P}} \frac{z^{|\pi|+1}}{(|\pi|+1)!} \tag{22.21}$$

Hay $k!$ permutaciones de tamaño k, sumando sobre tamaños resulta:

$$= zM(z) + \sum_{k\geq 0} k!\frac{z^{k+1}}{(k+1)!}$$

$$= zM(z) + \sum_{k\geq 0} \frac{z^{k+1}}{k+1}$$

Despejando:

$$M(z) = \frac{1}{1-z}\ln\frac{1}{1-z} \tag{22.22}$$

Reconocemos en (22.22) la función generatriz de los números harmónicos:

$$M(z) = \sum_{n\geq 1} H_n z^n$$

$$H_n = \sum_{1\leq k\leq n} \frac{1}{k} \tag{22.23}$$

$$= \ln n + \gamma + O(1/n) \tag{22.24}$$

donde $\gamma = 0{,}5772156649$ es la constante de Euler.

O sea, asintóticamente el número de asignaciones en el método de selección es mínimo 0, máximo $3n$, promedio $3\ln n$.

Clase 23

Quicksort

Quicksort, debido a Hoare [4], es otro algoritmo basado en dividir y conquistar, pero en este caso la división no es fija. Dado un rango de elementos de un arreglo a ser ordenado, se elige un elemento *pivote* de entre ellos y se reorganizan los elementos en el rango de forma que todos los elementos menores que el pivote queden antes de éste, y todos los elementos mayores queden después. Con

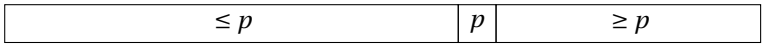


Figura 23.1 – Idea de Quicksort

esto el pivote ocupa su posición final en el arreglo, y bastará ordenar recursivamente cada uno de los dos nuevos rangos generados para completar el trabajo. La figura 23.2 indica una manera popular de

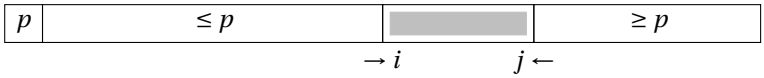


Figura 23.2 – Particionamiento en Quicksort

efectuar esta *partición*: se elige un pivote de forma aleatoria y el pivote elegido se intercambia con el primer elemento del rango (para sacarlo de en medio), luego se busca un elemento mayor que el pivote desde la izquierda y uno menor desde la derecha. Estos están fuera de orden, se intercambian y se continúa de la misma forma hasta agotar el rango. Después se repone el pivote en su lugar, intercambiándolo con el último elemento menor que él. El rango finalmente queda como indica la figura 23.1.

```
static double *a;

static int partition(const int lo, const int hi)
{
        int i = lo; j = hi + 1;

        for(;;) {
                while(a[++i] < a[lo])
```

```
                if(i == hi) break;
            while(a[lo] < a[--j])
                if(i == lo) break;
            if(i >= j) break;
            tmp = a[i]; a[i] = a[j]; a[j] = tmp;
        }
        return j;
}

static void qsr(const int lo, const int hi)
{
        int j;

        if(hi <= lo) return;
        j = partition(lo, hi);
        qsr(lo, j - 1);
        qsr(j + 1, hi);
}

void quicksort(double aa[], const int n)
{
        a = aa;
        qsr(0, n - 1);
}
```

Listado 23.1 – Versión simple de Quicksort

El listado 23.1 muestra una versión simple del programa, que elige siempre el primer elemento del rango como pivote.

23.1. Análisis del promedio

Evaluaremos el tiempo promedio de ejecución del algoritmo. Supondremos n elementos todos diferentes, que las $n!$ permutaciones de los n elementos son igualmente probables, y que el pivote se elige al azar en cada etapa. En este caso está claro que el método de particionamiento planteado no altera el orden de los elementos en las particiones respecto del orden que tenían originalmente. Luego, los elementos en cada partición también son una permutación al azar.

Para efectos del análisis del algoritmo tomaremos como medida de costo el número promedio de comparaciones que efectúa Quicksort al ordenar un arreglo de n elementos. El trabajo adicional que se hace en cada partición será aproximadamente proporcional a ésto, por lo que esta es una buena vara de medida. Al particionar, cada uno de los $n-1$ elementos fuera del pivote se comparan con este exactamente una vez en el método planteado, y además es obvio que este es el mínimo número de comparaciones necesario para hacer este trabajo. Si llamamos k a la posición final del pivote, el costo de las llamadas recursivas que completan el ordenamiento será $C(k-1)+C(n-k)$. Si elegimos el pivote al azar la probabilidad de que k tenga un valor cualquiera entre 1 y n es la misma. Cuando el rango es vacío no se efectúan comparaciones. Estas consideraciones llevan a la recurrencia:

$$C(n) = n-1+\frac{1}{n}\sum_{1\le k\le n-1}(C(k-1)+C(n-k)) \quad C(0)=0$$

Por simetría podemos simplificar la suma, dado que estamos sumando los mismos términos en orden creciente y decreciente. Extendiendo el rango de la suma y multiplicando por n queda:

$$nC(n) = n(n-1) + 2 \sum_{0 \le k \le n-1} C(k)$$

Ajustando los índices:

$$(n+1)C(n+1) = n(n+1) + 2 \sum_{0 \le k \le n} C(k) \quad C(0) = 0$$

Definimos la función generatriz ordinaria:

$$c(z) = \sum_{n \ge 0} C(n) z^n$$

Aplicando las propiedades de funciones generatrices ordinarias a la recurrencia queda la ecuación diferencial:

$$(z\mathrm{D} + 1)\frac{c(z)}{z} = \left((z\mathrm{D})^2 + zD\right)\frac{1}{1-z} + \frac{2c(z)}{1-z} \qquad c(0) = 0$$
$$c'(z) = \frac{2c(z)}{1-z} + \frac{2z}{(1-z)^3}$$

La solución a esta ecuación es:

$$c(z) = -2\frac{\ln(1-z)}{(1-z)^2} - \frac{2z}{(1-z)^2}$$

El primer término corresponde a la suma parcial de números harmónicos (se deriva su función generatriz en [2, capítulo 19]), el segundo término da un coeficiente binomial:

$$\begin{aligned} C(n) &= 2 \sum_{0 \le k \le n} H_k - 2\binom{n}{1} \\ &= 2 \sum_{0 \le k \le n} H_k - 2n \end{aligned}$$

Los coeficientes del primer término son conocidos:

$$\sum_{0 \le k \le n} H_k = (n+1)H_n - n \tag{23.1}$$

Esto da finalmente:

$$C(n) = 2(n+1)H_n - 4n$$

Sabemos que (ver [2, capítulo 18]) que $H_n = \ln n + \gamma + O(1/n)$, donde $\gamma = 0{,}57721\,56649\,01532\,65120$ con lo que $C(n) = 2n\ln n + O(n)$.

23.2. Análisis del peor y mejor caso

Pero podemos hacer más. En el peor caso, al particionar en cada paso elegimos uno de los elementos extremos, con lo que las particiones son de largo 0 y $n-1$, lo que da lugar a la recurrencia:

$$C_{\text{peor}}(n) = n - 1 + C_{\text{peor}}(n-1) \quad C_{\text{peor}}(0) = 0$$

Las técnicas estándar dan como solución:

$$\begin{aligned} C_{\text{peor}}(n) &= \frac{n(n-1)}{2} \\ &= \frac{1}{2}n^2 + O(n) \end{aligned}$$

El mejor caso es cuando en cada paso la división es equitativa, lo que lleva casi a la situación de dividir y conquistar analizada antes con $a = 2$, $b = 2$ y $d = 1$, cuya solución sabemos es $C_{\text{mejor}}(n) = O(n \log n)$. Un análisis más detallado, restringido al caso en que $n = 2^k - 1$ de manera que los dos rangos siempre resulten del mismo largo, es como sigue. La recurrencia original se reduce a:

$$C_{\text{mejor}}(n) = n - 1 + 2C_{\text{mejor}}((n-1)/2) \quad C_{\text{mejor}}(0) = 0$$

Con el cambio de variables:

$$n = 2^k - 1 \quad F(k) = C_{\text{mejor}}(2^k - 1)$$

esto se transforma en:

$$F(k) = 2^k - 2 + 2F(k-1) \quad F(0) = 0$$

cuya solución es:

$$\begin{aligned} F(k) &= k2^k + 2^{k+1} + 2 \\ C_{\text{mejor}}(n) &= (n+1)\log_2(n+1) + 2(n+1) + 2 \\ &= \frac{1}{\ln 2}\, n \ln n + O(n) \end{aligned}$$

La constante en este caso es aproximadamente 1,443, el mejor caso no es demasiado mejor que el promedio; pero el peor caso es mucho peor.

23.3. Consideraciones prácticas

Una variante común es usar un método de ordenamiento simple para rangos chicos, dado que Quicksort es más costoso que métodos simples para rangos pequeños. Una opción es cortar la recursión no cuando el rango se reduce a un único elemento sino cuando cae bajo un cierto margen; y luego se ordena todo mediante inserción, que funciona muy bien si los datos vienen "casi ordenados", como resulta de lo anterior. Para analizar esto se requieren medidas más ajustadas del costo de los métodos, y se cambian las condiciones de forma que para valores de n menor que el límite se usa el costo del método alternativo. Esto puede hacerse, pero es bastante engorroso y no lo veremos acá.

Para evitar el peor caso (que se da cuando el pivote es uno de los elementos extremos) una opción es tomar una muestra de elementos y usar la mediana (el elemento del medio de la muestra) como pivote. La forma más simple de hacer esto es tomar tres elementos. Como además es frecuente que se invoque el procedimiento con un arreglo "casi ordenado" (o incluso ya ordenado), conviene tomar como muestra el primero, el último y un elemento del centro, de forma de elegir un buen pivote incluso en ese caso patológico. A esta idea se le conoce como *mediana de tres*. Esta estrategia disminuye un tanto la constante por efecto de una división más equitativa. Tiene la ventaja adicional que tener elementos menor que el pivote al comienzo del rango y mayor al final no es necesario comparar índices para determinar si se llegó al borde del rango. El análisis detallado se encuentra por ejemplo en Sedgewick y Flajolet [7].

Por el otro lado, McIllroy [5] muestra cómo lograr que siempre tome el máximo tiempo posible. Quicksort (haciendo honor a su nombre) es muy rápido ya que las operaciones en sus ciclos internos implican únicamente una comparación y un incremento o decremento de un índice. Es ampliamente usado, y como su peor caso es muy malo, vale la pena hacer un estudio detallado de la ingeniería del algoritmo, como hacen Bentley y McIllroy [1]. Debe tenerse cuidado con Quicksort por su peor caso, si un atacante puede determinar los datos puede hacer que el algoritmo consuma muchísimos recursos. Para evitar el peor caso se ha propuesto cambiar a Heapsort, debido a Williams [8] (garantizadamente $O(n \log n)$, pero mucho más lento que Quicksort) si se detecta un caso malo, como propone Musser [6].

Lo anterior supone que todos los valores son diferentes. Si hay muchos elementos repetidos (en el extremo, son todos iguales), resulta que si al particionar paramos en elementos iguales al pivote caemos en el peor caso (el pivote termina en un extremo). Conviene parar si hallamos un elemento igual al pivote, como se hace en el listado 23.1.

Pero lo que realmente nos conviene es particionar es *tres* tramos, como muestra la figura 23.3. Esto es una variante del *problema de la bandera holandesa* (*Dutch national flag problem*, ver la

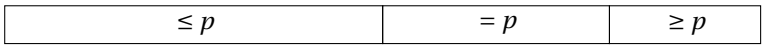


Figura 23.3 – Particionamiento archo

figura 23.4) propuesto por Dijkstra [3]: dada una secuencia de canicas de colores rojo, blanco y azul, ordenarlas de manera de tener juntas las rojas, las blancas y las azules. Una manera de efectuar esto

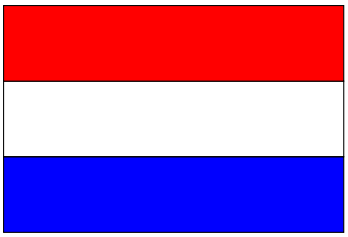


Figura 23.4 – La bandera holandesa

es usar el invariante de la figura 23.5. Código es como indica el listado 23.2.

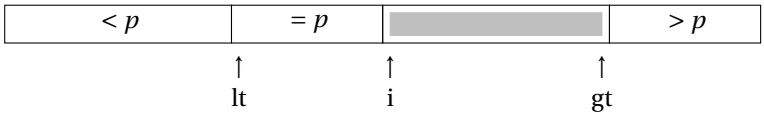


Figura 23.5 – Invariante para particionamiento ancho

```
static void partition(const int lo, const int hi, int *lt, int *gt)
{
        int i = lo;
        double p = a[lo];

        *lt = lo; *gt = hi - 1;

        while(i <= *gt) {
```

```
                if(p < a[i])
                        swap(&a[*lt++], &a[i++]);
                else if(p > a[i])
                        swap(&a[i], &a[*gt--]);
                else
                        i++;
        }

}
```

Listado 23.2 – Partición ancha

Bibliografía

[1] Jon Louis Bentley and M. Douglas McIllroy: *Engineering a sort function.* Software: Practice and Experience, 23(11):1249–1265, November 1993.

[2] Horst H. von Brand: *Fundamentos de Informática.* git://csrg.inf.utfsm.cl/vonbrand/Ramos/trainwreck, Agosto 2016. Versión 0.83.

[3] Edsger W. Dijkstra: *A Discipline of Programming.* Prentice Hall, 1976.

[4] C. A. R. Hoare: *Quicksort.* Computer Journal, 5(1):10–15, April 1962.

[5] M. Douglas McIllroy: *A killer adversary for Quicksort.* Software: Practice and Experience, 29(4):341–344, April 1999.

[6] David Musser: *Introspective sorting and selection algorithms.* Software: Practice and Experience, 27(8):983–993, August 1997.

[7] Robert Sedgewick and Philipe Flajolet: *An Introduction to the Analysis of Algorithms.* Addison-Wesley, second edition, 2013.

[8] J. W. J. Williams: *Algorithm 232 – Heapsort.* Communications of the ACM, 7(6):347–348, June 1964.

Clase 24

Algoritmo de Kruskal, Union-Find

Un problema recurrente es hallar el árbol recubridor mínimo (*Minimal Spanning Tree* en inglés, abreviado MST) de un grafo rotulado conexo. En detalle, dado un grafo $G = (V, E)$ conexo, con arcos rotulados por $w\colon E \to \mathbb{R}^+$ (el rótulo representa costo del arco), hallar un árbol recubridor de costo total mínimo. Una solución a este problema da el algoritmo de Kruskal [3], (algoritmo 24.1). La idea es ir construyendo un bosque (conjunto de árboles), uniendo sucesivamente árboles mediante arcos de costo mínimo. Inicialmente el bosque es simplemente cada vértice por separado, al final es un árbol recubridor mínimo. Este es un ejemplo clásico de algoritmo voraz. Nos interesa derivar

Algoritmo 24.1: Algoritmo de Kruskal

```
Ordenar E en orden de costo creciente
S ← ∅
for v ∈ V do
    Agregar {v} a S
end
T ← ∅
for uv ∈ E do
    if u y v no pertenecen al mismo conjunto de S then
        Agregar uv a T
        Unir los conjuntos en S a los que pertenecen u y v
    end
end
return (V, T)
```

un programa eficiente (y deducir su complejidad). Es claro que la manipulación del conjunto S es crucial. Parte de la discusión que sigue viene de Erickson [1, clase 17].

24.1. Una estructura de datos para *union-find*

Abstrayendo las operaciones empleadas, vemos que requerimos una estructura de datos que representa una partición de un conjunto universo V, con operaciones de inicializar con elementos

solitarios, hallar (*find*) la partición a la que pertenece un elemento, y unir particiones disjuntas (*union*). Por ellas se le llama problema de *union-find*. La manera de identificar a una subconjunto

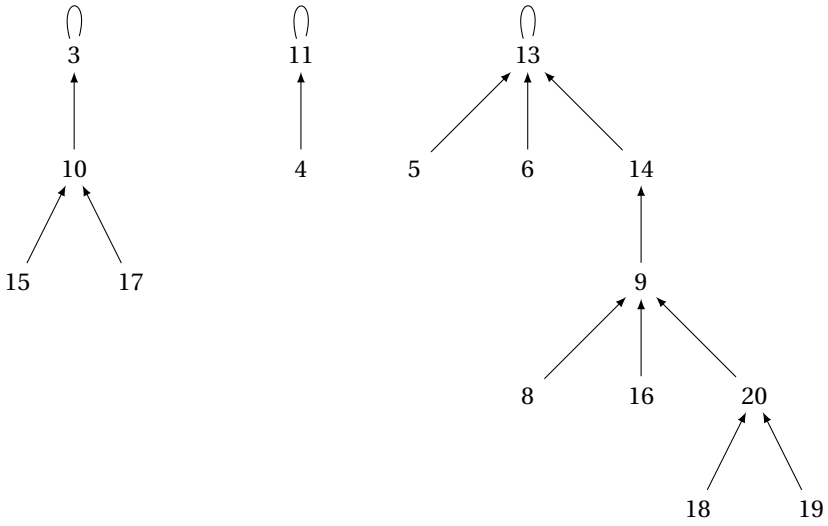


Figura 24.1 – Esquema de la estructura para *union-find*.

de V es irrelevante, podemos elegir un elemento cualquiera como representante. Para hallar el representante del conjunto al que pertenece v podemos hacer que cada elemento apunte a un elemento padre, siguiendo la lista hasta el final hallamos el representante. Unir dos conjuntos es hacer que el representante de uno quede como padre del representante del otro, ver la figura 24.1. Para simplificar, de ahora en adelante omitiremos las flechas y los bucles en las raíces.

La operación find depende de la altura de los árboles que se construyan, interesa construir árboles bajos. Nos conviene hacer que el representante con el árbol menor dependa del representante con el árbol más alto, ya que esto no aumenta la altura. Si mantenemos un arreglo rank con la altura del árbol de cada representante, basta poner de hijo al representante de altura menor. Solo en caso de empate la altura aumenta, elegimos uno de los dos como nuevo representante con rank uno mayor. Inicialmente rank es cero para todos los vértices. Nótese que solo cuando se unen dos árboles de la misma altura se ajusta rank del nuevo representante. Es un simple ejercicio de inducción demostrar que de esta manera si v es representante de una clase, ésta contiene al menos $2^{\text{rank}[v]}$ elementos. En consecuencia, el máximo camino posible de un vértice a su representante en la clase C es de largo $\log_2 C$. El costo para cada operación es $O(\log n)$. Esta estructura y su manipulación fueron propuestas por Galler y Fisher [2], aunque su análisis tomó años. La idea se basa en arreglos globales rank (la altura del árbol con raíz en el vértice) y parent (el padre del vértice, para la raíz es el mismo para simplificar el código). Vea los algoritmos para MakeSets (crea la estructura inicial), union (une las clases de u y v) y find (halla el representante para la clase de v), respectivamente 24.2, 24.3, y 24.4,.

Algoritmo 24.2: Algoritmo para crear conjuntos

```
procedure MakeSets(V)
    for v ∈ V do
        rank[v] ← 0
        parent[v] ← v
    end
```

Algoritmo 24.3: Algoritmo para unir conjuntos

```
procedure union(u, v)
    u ← find(u)
    v ← find(u)
    if rank[u] > rank[v] then
        parent[v] ← u
    else
        parent[u] ← v
        if rank[u] = rank[v] then
            rank[u] ← rank[u] + 1
        end
    end
```

Algoritmo 24.4: Algoritmo para encontrar representante

```
function find(v)
    while v ≠ parent[v] do
        v ← parent[v]
    end
    return v
```

Bibliografía

[1] Jeff Erickson: *Algorithms, etc.* http://jeffe.cs.illinois.edu/teaching/algorithms, January 2015.

[2] Bernhard A. Galler and Michael J. Fisher: *An improved equivalence algorithm.* Communications of the ACM, 7(5):301–303, May 1964.

[3] Joseph B. Kruskal: *On the shortest spanning subtree of a graph and the traveling salesman problem.* Proceedings of the American Mathematical Society, 7(1):48–50, February 1956.

Clase 25

Análisis de Union-Find

Vimos que la estructura union-find es central en varios algoritmos, corresponde evaluar su rendimiento. Las técnicas empleadas son instructivas, las expandiremos en clases sucesivas.

25.1. Análisis de la versión simple

Veamos algunas propiedades cruciales de nuestra estructura inicial bajo las operaciones mencionadas:

Propiedad 1: *Para todo v,* $\text{rank}[v] < \text{rank}[\text{parent}[v]]$.

Un nodo de rango k se crea al unir dos árboles de rango $k-1$, una vez que un nodo deja de ser raíz su rango no se modifica más.

Propiedad 2: *Una raíz de rango k tiene al menos 2^k nodos en su árbol.*

Una simple inducción. Esto se aplica a nodos internos (no raíz) también: un nodo de rango k tiene al menos 2^k descendientes, ya que alguna vez fue raíz y una vez que deja de serlo su rango y sus descendientes no cambian más.

Propiedad 3: *Si hay un total de n nodos, hay a lo más $n/2^k$ nodos de rango k.*

Demostración. Considere su valor favorito k, y cada vez que rank de un nodo x cambia de $k-1$ a k marque todos sus descendientes. Cada nodo puede ser marcado a lo más una vez, ya que el rango de la raíz solo aumenta. Un nodo con rango k representa al menos a 2^k nodos, habiendo un total de n nodos, hay a lo más $n/2^k$ nodos de rango k. □

Esta observación indica, crucialmente, que el rango máximo es $\log_2 n$, con lo que todos los árboles tienen altura a lo más $\log_2 n$, y esta es una cota superior al tiempo de ejecución de find y de union.

Resulta que la cota de altura $\log_2 n$ es ajustada, el árbol binomial B_k (construido uniendo dos árboles binomiales B_{k-1}, ver figura 25.1) tiene 2^k nodos y altura k.

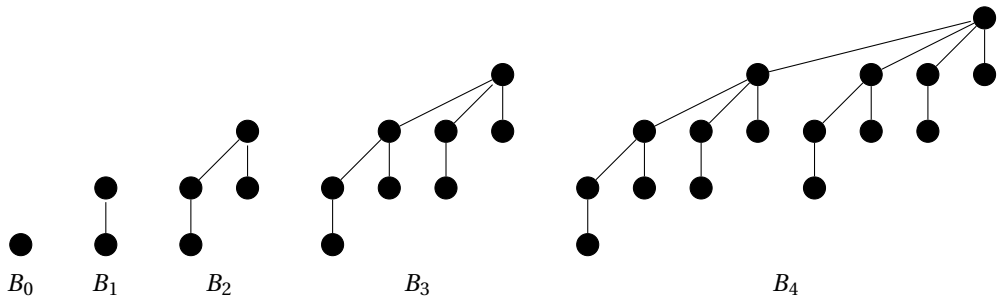


Figura 25.1 – Árboles binomiales

25.2. Compresión de caminos

Una mejora se obtiene de la observación que luego de una búsqueda podemos acortar los caminos al representante a un solo paso para todos los nodos que encontramos en el camino, la figura 25.2 ilustra el efecto de buscar 20 y comprimir caminos. El algoritmo resultante 25.1 es la

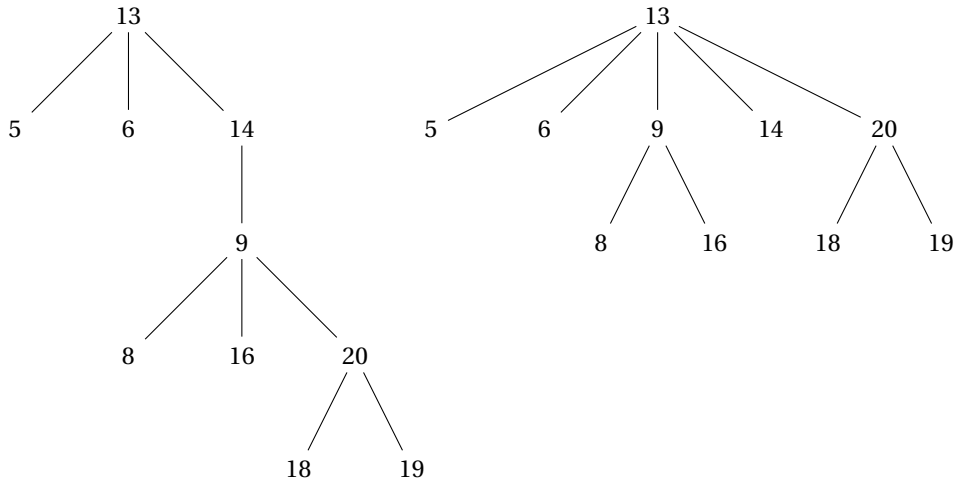


Figura 25.2 – Acortar caminos (*path compression*).

versión modificada de find. La idea es pagar un costo extra en las operaciones find en la esperanza de ahorrar en operaciones futuras. Una variante más simple es cambiar abuelos por padres, ver el algoritmo 25.2, que difiere del original en una única línea. La figura 25.3 ilustra el efecto al buscar 19.

25.3. Análisis de compresión de caminos

Como estamos pagando un costo extra en ciertas operaciones en la esperanza de que produzca ahorros futuros, debemos analizar secuencias de operaciones, no operaciones individuales.

Para $g\colon \mathbb{R} \to \mathbb{R}$ tal que para $x > 1$ siempre es $g(x) < x$ definimos:

$$g^*(x) = \begin{cases} 0 & x \leq 1 \\ 1 + g^*(g(x)) & x > 1 \end{cases} \tag{25.1}$$

Algoritmo 25.1: Algoritmo modificado para encontrar representante

```
function find(v)
    u ← v
    while v ≠ parent[v] do
        v ← parent[v]
    end
    while u ≠ v do
        p ← parent[u]
        parent[u] ← v
        u ← p
    end
    return v
```

Algoritmo 25.2: Algoritmo para encontrar representante con compresión de abuelos

```
function find(v)
    while v ≠ parent[v] do
        parent[v] ← parent[parent[v]]
        v ← parent[v]
    end
    return v
```

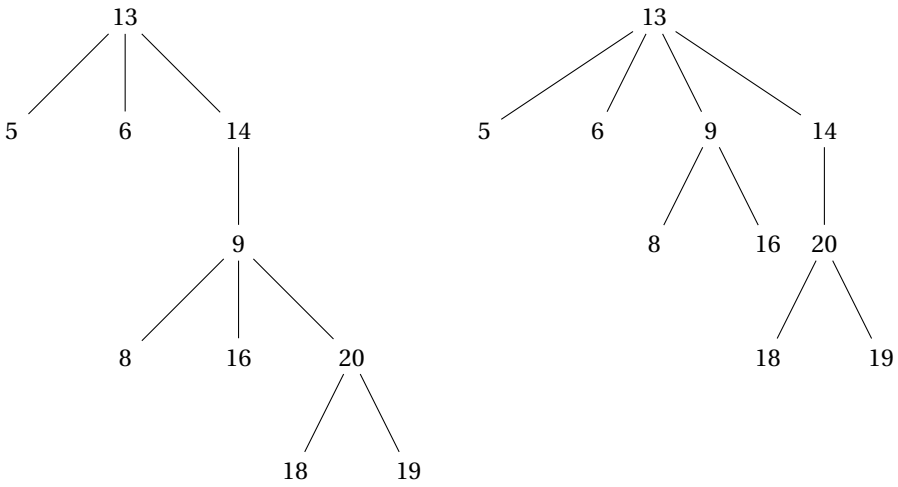


Figura 25.3 – Acortar caminos (*path compression*) con abuelos.

En el fondo, $g^*(x)$ es el número de veces que hay que aplicar g a x hasta obtener un valor 1 o menor. De acá definimos $\log^* x$, donde el logaritmo es en base 2 (¡somos computines!). Es claro que $\log^* n$

crece extremadamente lento:

$$\log^* n = \begin{cases} 0 & n \leq 1 \\ 1 & 1 < n \leq 2 \\ 2 & 2 < n \leq 2^2 \\ 3 & 2^2 < n \leq 2^4 \\ 4 & 2^4 < n \leq 2^{16} \\ 5 & 2^{16} < n \leq 2^{65536} \end{cases}$$

El análisis clásico de esta estructura (Hopcroft y Ullman [2], Tarjan [4]) es complejo. Acá seguimos la idea de Seidel y Sharir [3], que da un análisis sencillo (todo depende del cristal con que se mira...). Primeramente, las tres propiedades enunciadas antes se siguen cumpliendo aún si se comprimen caminos. Basan su análisis en dos nuevas operaciones, compress(u, v) que comprime un camino cualquiera en el bosque (no necesariamente llegando a una raíz) entre nodos u y v, donde v es un ancestro de u, y la operación shatter(u, v), que hace una raíz de todo nodo en el camino. Cabe hacer notar que estas operaciones no son para uso en el programa, sirven para reordenar las

Algoritmo 25.3: Operación compress

```
procedure compress(u, v)
    v must be ancestor of u
    if u ≠ v then
        compress(parent[u], v)
        parent[u] ← parent[v]
    end
```

Algoritmo 25.4: Operación shatter

```
procedure shatter(u, v)
    v must be ancestor of u
    if parent[u] ≠ v then
        shatter(parent[u], v)
        parent[u] ← u
    end
```

acciones simplificando las demostraciones. En particular, si union no reorganiza los árboles, solo manipula las raíces, una secuencia cualquiera de union y find puede efectuarse haciendo las union, seguidas por compress sin cambiar el número de manipulaciones de punteros. El costo de union es constante ($O(1)$), find es básicamente compress, que es $O(1)$ más un término proporcional al número de punteros manipulados. Fijaremos entonces el número de punteros manipulados como medida de costo. Sea $T(m, n, r)$ el número de asignaciones de punteros en el peor caso en cualquier secuencia de m operaciones compress sobre un bosque de a lo más n nodos, con rank a lo más r.

La siguiente cota trivial sirve de base a nuestro argumento.

Teorema 25.1. $T(m, n, r) \leq nr$

Demostración. Cada nodo puede cambiar padre a lo más r veces, ya que rank siempre aumenta. □

Sea $\mathcal{F}$ un bosque de n nodos con rank máximo r y una secuencia C de m operaciones compress sobre $\mathcal{F}$, y sea $T(\mathcal{F}, C)$ el número total de asignaciones de punteros ejecutados por esta secuencia. Divida el bosque en dos sub-bosques, un bosque "bajo" $\mathcal{F}_-$ con los nodos de rank$[v] \leq s$ y el bosque "alto" $\mathcal{F}_+$ con los nodos de rank$[v] > s$. Como rank aumenta al seguir punteros a padres, el padre de un nodo alto es otro nodo alto. Sean n_- y n_+ el número de nodos bajos y altos, respectivamente. Ver la figura 25.4 que muestra un bosque (un único árbol en el ejemplo) dividido en sub-bosques.

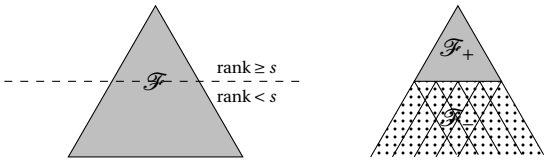


Figura 25.4 – Dividiendo el bosque según rank

Cualquier secuencia de operaciones compress sobre $\mathcal{F}$ puede descomponerse en una secuencia de operaciones compress sobre $\mathcal{F}_+$ y una secuencia de operaciones compress y shatter sobre $\mathcal{F}_-$ con el mismo costo. La modificación es prohibir a un nodo bajo tener un padre alto, ver la figura 25.5. El punto de hacer esto es descomponer la secuencia en secuencias menores. Al dividir el bosque

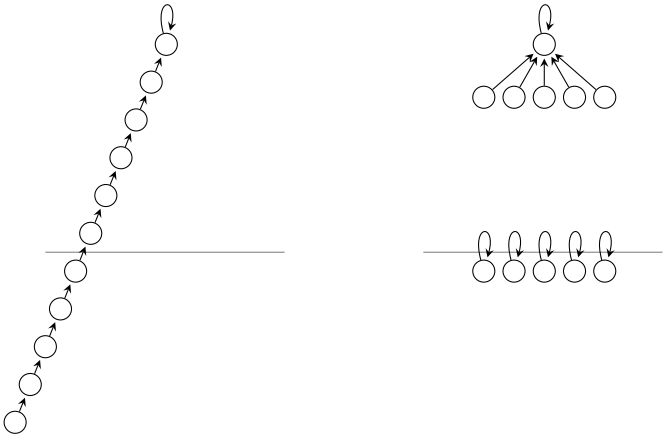


Figura 25.5 – División de una operación compress

en nodos "altos" y "bajos", queda un bosque alto muy pequeño (son pocos los nodos con rank alto), con lo que bastarán cotas bastante burdas para el costo de operaciones en él. El resultado es una recurrencia para el costo de la secuencia.

La operación compress adaptada a bosques divididos considera el caso en que el camino u a v es enteramente "alto" (caso rank$[u] > s$), enteramente "bajo" (cuando rank$[v] \leq s$), o cruza el rango y hay que subdividir la operación.

La última asignación del algoritmo 25.5 parece superflua, pero es necesaria en el análisis para simular una operación parent$[z] \leftarrow w$, con z un nodo bajo, w un nodo alto y el padre de z era un nodo alto también. Estas asignaciones "redundantes" se ejecutan inmediatamente después de una operación compress en el bosque superior, por lo que hay a lo más m_+ de estas operaciones.

Durante la secuencia de operaciones C cada nodo es tocado por a lo más una operación shatter, por lo que el número total de operaciones con punteros en ellas es a lo más n.

Al dividir el bosque hemos dividido la secuencia de operaciones compress en subsecuencias C_- y C_+ de operaciones compress, de largos respectivos m_- y m_+ (es $m = m_+ + m_-$), además de

Algoritmo 25.5: Operación equivalente

```
procedure compress-rank(u, v)
    if rank[u] > s then
        compress(u, v)
    else if rank[v] ≤ s then
        compress(u, v)
    else
        z ← u
        while rank[parent[z]] ≤ s do
            z ← parent[z]
        end
        compress(parent[z], v)
        shatter(u, z)
        parent[z] ← z
    end
```

operaciones shatter. En vista de las consideraciones anteriores se cumple la desigualdad:

$$T(\mathscr{F}, C) \leq T(\mathscr{F}_-, C_-) + T(\mathscr{F}_+, C_+) + m_+ + n \tag{25.2}$$

Como hay a lo más $n/2^i$ nodos de rango i, tenemos que:

$$n_+ \leq \sum_{i>s} \frac{n}{2^i} = \frac{n}{2^s}$$

Con esto la cota del teorema 25.1 implica:

$$T(\mathscr{F}_+, C_+) \leq \frac{r\,n}{2^s}$$

Fijemos $s = \log_2 r$, de manera que $T(\mathscr{F}_+, C_+) \leq n$. El bosque $\mathscr{F}_-$ tiene rank máximo $s = \lfloor \log_2 r \rfloor$, además es claro que $|C_-| \leq |C| = m$. Podemos simplificar nuestra recurrencia a:

$$T(\mathscr{F}, C) \leq T(\mathscr{F}_-, C_-) + m_+ + 2n$$

lo que con las observaciones previas es lo mismo que:

$$T(\mathscr{F}, C) - m \leq T(\mathscr{F}_-, C_-) - m_- + 2n$$

Como esto se aplica a *cualquier* bosque $\mathscr{F}$ y secuencia C, hemos demostrado que para $T'(m,n,r) = T(m,n,r) - m$:

$$T'(m,n,r) \leq T'(m,n,\lfloor \log_2 r \rfloor) + 2n$$

Como condición inicial, para $r = 1$ todos los nodos son raíces o hijos de una raíz, no hay manipulación de parent, y $T'(m,n,1) \leq 0$. La solución a esta recurrencia es:

$$T'(m,n,r) \leq 2n \log_2^* r$$

Hemos demostrado:

Teorema 25.2. $T(m,n,r) \leq m + 2n \log_2^* r$

El teorema 25.2 puede mejorarse. En la demostración usamos la cota del teorema 25.1, que nuestro teorema 25.2 mejora, y puede usarse recursivamente. Erickson [1, clase 17] completa el desarrollo.

Volvamos al algoritmo de Kruskal, llamemos simplemente V y E al número de vértices y arcos, respectivamente. Se hacen a lo más $2E$ operaciones find, y exactamente $V-1$ operaciones union. El paso inicial, ordenar los arcos, puede hacerse en tiempo $O(E \log E)$. El número de arcos está acotado por $V(V-1)/2$, con lo que $\log E = O(\log V)$. Sabemos que $r \leq \lfloor \log_2 V \rfloor$, como $2^r \leq V$, resulta $\log^* r = \log^* V - 1$. Las operaciones con clases de equivalencia aportan $O(V)$ para los union, los find aportan $O(2E + 2V \log^* V)$ para un costo total de:

$$O(E \log E) + O(V + 2E + 2V \log^* V) = O(E \log E)$$

Esto es dominado por el costo de ordenar los arcos.

Bibliografía

[1] Jeff Erickson: *Algorithms, etc.* http://jeffe.cs.illinois.edu/teaching/algorithms, January 2015.

[2] John E. Hopcroft and Jeffrey D. Ullman: *Set merging algorithms.* SIAM Journal of Computing, 2(4):294–303, 1973.

[3] Raimund Seidel and Micha Sharir: *Top-down analysis of path compression.* SIAM Journal of Computing, 34(3):515–525, 2005.

[4] Robert E. Tarjan: *Efficiency of a good but not linear set union algorithm.* Journal of the ACM, 22(2):215–225, April 1975.

Clase 26

Análisis Amortizado

Discutiremos una forma útil de análisis, llamada *análisis amortizado*. Al usar una estructura de datos (o sus algoritmos asociados) ciertamente interesa el costo de cada operación individual, pero en una perspectiva más amplia interesa acotar el costo total de la *secuencia* de las operaciones efectuadas. Acotar el peor caso puede dar una cota exageradamente pesimista si los costos de las operaciones varían fuertemente, en particular si una operación "cara" solo es posible luego de una seguidilla de operaciones "baratas". La definición es simple:

Definición 26.1. El *costo amortizado por operación* en una secuencia de n operaciones es el costo total de la secuencia dividido por n.

La definición es simple, la aplicación muchas veces requiere cuidado y creatividad.

Nótese que esto difiere de análisis de caso promedio. Por ejemplo, para Quicksort derivamos un costo promedio de $O(n \log n)$ para ordenar un arreglo de n elementos, pero el peor caso es $O(n^2)$. Nada garantiza que en una secuencia de m ordenamientos el costo total esté acotado por $O(mn \log n)$, perfectamente podemos caer casi siempre en el peor caso, dando $O(mn^2)$. Acá buscamos acotar el peor caso de la *secuencia* de operaciones, considerando las interacciones entre ellas. No entran en el análisis distribuciones de probabilidad de los datos de entrada (muchas veces impracticables de obtener, o al menos imposibles de tratar analíticamente, terminando en modelos bastante alejados de la realidad).

Hay tres métodos principales (comparar con CLRS [1, capítulo 17]): análisis agregado, método de contabilidad y funciones potenciales. El primero suele ser más simple; los dos últimos son equivalentes, en el sentido que los tres pueden aplicarse a los mismos problemas obteniendo los mismos resultados. La ventaja de los últimos dos métodos es que permiten análisis más detallado, asignando costos diferentes a operaciones distintas. Claro que dependiendo del problema uno puede resultar más natural que los otros.

26.1. Arreglo dinámico

Podemos representar un *stack* ("pila", para los puristas del castellano) mediante un arreglo, extendiendo el arreglo si llega a llenarse. Las operaciones son bastante simples, ver el listado 26.1. Hemos omitido el verificar si el *stack* contiene elementos o llenó el arreglo.

```
typedef item ...;
item A[...];
static int top = 0;

void push(item A[], item x)
{
        A[top++] = x;
}

item pop(item A[])
{
        return A[--top];
}
```

Listado 26.1 – Operaciones sobre un *stack*

La pregunta es qué hacer si el arreglo se llena. La biblioteca de C ofrece la opción de solicitar memoria, copiar el contenido de un área al comienzo de ésta y liberar el original (ver `realloc(3)` en su Unix preferido o refiérase a Wikipedia [2]). lo que permite completar las anteriores. Supongamos que decidimos expandir el arreglo cuando se llene, la pregunta es cuándo hacerlo para mantener rendimiento aceptable. Considerando el costo de un push o pop simplemente el costo de copiar un ítem, y similarmente que el costo de expandir el arreglo cuando tiene tamaño n es n (en el fondo, el costo es solo el copiar los ítem). Si extendemos el arreglo en un elemento cada vez, para llegar a tamaño n hay que hacer n operaciones push partiendo del *stack* vacío, cuando el *stack* tiene tamaño k el costo es $k+1$, para un costo total de n operaciones:

$$\sum_{0\le k\le n-1}(k+1)=\frac{n(n+1)}{2}$$

dando un costo amortizado de $(n+1)/2$. Para operaciones tan simples esto es inaceptable.

Si extendemos el arreglo duplicando su tamaño cada vez que se llena, lo que tenemos es que el costo de las duplicaciones en n operaciones está acotado por una suma de la forma:

$$1+(1+1)+(2+1)+1+(4+1)+1+1+1+\cdots+(1+2^k)+1+\cdots$$

Acá estamos copiando un elemento cada vez que se efectúa un push, y cada vez que pasamos de una potencia de dos además debemos copiar el arreglo actual. O sea, en total tenemos n copias por push y copiamos 2^k elementos cada vez que pasamos de esa potencia. El número total de copias es:

$$\begin{aligned} n+\sum_{0\le k\le \lfloor\log_2 n-1\rfloor-1} 2^k &\le n+2^{\lfloor\log_2 n\rfloor+1}-1\\ &< n+2n\\ &= 3n \end{aligned}$$

El costo amortizado es menor a 3. Bastante más aceptable.

Este es un ejemplo claro de *análisis agregado*, consideramos una secuencia de operaciones, calculamos el costo de ella y dividimos por el número de operaciones. Es exactamente la estrategia que usamos al analizar Union-Find.

26.2. Contador binario

Imagine que debemos almacenar un contador binario grande en un arreglo a, todas cuyas entradas se inician en 0 y en cada paso el contador se incrementa en 1. Supongamos el modelo de costo

que carga una unidad cada vez que un bit cambia. En una secuencia de n operaciones, el peor costo es $\lfloor \log_2 n \rfloor$, el número máximo de bits que cambian de 1 a 0. Pero el costo amortizado es menor a 2, cosa que demostraremos usando los distintos métodos.

26.2.1. Método contable

La idea es mantener un saldo, cobramos por operaciones y ahorramos los sobrepagos, pagando por operaciones caras con el saldo. Hay que tener cuidado que el saldo nunca pueda hacerse negativo.

Proposición 26.1. *El costo amortizado de la operación de incremento es a lo más* 2 *cambios de bit.*

Demostración. Cargue 2 unidades a la operación de incremento. Si cambia $0 \to 1$, gasta 1 en la operación y ahorra 1; si cambia $1 \to 0$ usa lo ahorrado para pagar por los cambios adicionales. Una manera de ver que nunca terminamos con saldo negativo es considerar que cada bit tiene su propio saldo, cuando cambia de 0 a 1 se le abona 1, y ese se gasta al cambiar de 1 a 0. Si tenemos una secuencia de 1 al final, e incrementamos, pagamos el 1 ahorrado en cada bit para cambiarlo a 0, pagamos 1 para cambiar el primer 0 a 1 y le dejamos 1 de ahorro. Hemos gastado 2.

Como de esta forma el saldo nunca es negativo, el costo nunca sobrepasa 2 cambios de bit por incremento. □

26.2.2. Método agregado

Para contraste, una demostración usando el método agregado sería:

Demostración. Consideremos una secuencia de incrementos, y consideremos cuántas veces cambia cada bit. Claramente, `a[0]` cambia cada vez, `a[1]` cambia cada dos incrementos, ..., `a[k]` cambia cada 2^k incrementos, y así sucesivamente. El costo total de los cambios a `a[0]` es n, los cambios a `a[1]` tienen costo total el piso de $n/2$, ..., cambios de `a[k]` cuestan el piso de $n/2^k$, y así sigue.

En total, el costo es:

$$\begin{aligned}
\lfloor n \rfloor + \left\lfloor \frac{n}{2} \right\rfloor + \cdots + \left\lfloor \frac{n}{2^{\lfloor \log_2 n \rfloor}} \right\rfloor &= \sum_{0 \le k \le \log_2 n} \left\lfloor \frac{n}{2^k} \right\rfloor \\
&\le n \sum_{0 \le k \le \log_2 n} 2^{-k} \\
&< n \sum_{k \ge 0} 2^{-k} \\
&= 2n
\end{aligned}$$

De aquí el costo amortizado es menor a 2. □

26.2.3. Función potencial

En las anteriores contabilizábamos costos por operaciones individuales. Una mirada alternativa es considerar que la estructura de datos tiene un *potencial*, una función $\Phi \colon \mathcal{S} \to \mathbb{R}$ de estados $\mathcal{S}$ de la estructura a los reales. Supongamos una secuencia de operaciones $\sigma_1, \sigma_2, \ldots, \sigma_n$, que llevan a la estructura del estado inicial s_0 sucesivamente a $s_1, \ldots, s_n$. Sea c_i el costo real de la operación σ_i, y defina el costo amortizado a_i de σ_i mediante:

$$a_i = c_i + \Phi(s_i) - \Phi(s_{i-1})$$

O sea:

(costo amortizado) = (costo real) + (cambio de potencial)

Sumando sobre la secuencia de operaciones:

$$\begin{aligned}\sum_i a_i &= \sum_i (c_i + \Phi(s_i) - \Phi(s_{i-1})) \\ &= \sum_i c_i + \Phi(s_n) - \Phi(s_0)\end{aligned}$$

Reorganizando:

$$\sum_i c_i = \sum_i a_i - (\Phi(s_n) - \Phi(s_0))$$

Si $\Phi(s_n) \geq \Phi(s_0)$ (caso común), tenemos:

$$\sum_i c_i \leq \sum_i a_i$$

Acotando los costos amortizados a_i, acotamos el costo de cualquier secuencia de operaciones.

Es claro que esta visión es equivalente a las anteriores.

Apliquemos este método a nuestro problema ahora.

Demostración. Sea el potencial Φ el número de bits 1 en el arreglo, y representemos el estado simplemente por el número representado por el contador. Vemos que $\Phi(0) = 0$ y que $\Phi(n) > 0$. Interesa acotar el costo amortizado de incrementos.

Considere el k-ésimo incremento, que cambia de $k-1$ a k. Sea c el número de *carries* en este incremento, con lo que el costo de esta operación es $c+1$. El cambio de potencial que produce es $-c+1$ (hay c unos que cambian a ceros, y un cero que cambia a uno). El costo amortizado de la operación es:

$$\begin{aligned}a_k &= c + 1 + (-c + 1) \\ &= 2\end{aligned}$$

Como el potencial final es mayor al inicial, tenemos que:

$$\begin{aligned}\sum_i c_i &< \sum_i a_i \\ &= 2n\end{aligned}$$

□

En este análisis, la selección de la función potencial es crítica.

Ejercicios

1. Suponga una secuencia de operaciones numeradas $1, 2, 3, \ldots$ tal que la operación i tiene costo 1 si i no es una potencia de 2, mientras el costo es i si i es una potencia de 2. ¿Cuál es una cota ajustada del costo amortizado de las operaciones?

2. Consideremos una cola de prioridad con operaciones dadas en el algoritmo 26.1.

   a) Analice el tiempo de ejecución de cada procedimiento.

Algoritmo 26.1: Operaciones sobre la cola de prioridad

```
procedure Init(n)
    for i ← 1 to n do
        a[i] ← true
    end

procedure Delete(i)
    a[i] ← false

function DeleteMin()
    i ← 1
    while ¬a[i] do
        i ← i + 1
    end
    if i ≤ n then
        Delete(i)
        return i
    else
        return 0
    end
```

b) Modifique DeleteMin de manera que su tiempo de ejecución amortizado es $O(1)$, manteniendo los órdenes de magnitud de los tiempos de las otras operaciones. Especifique la función potencial que emplea en su análisis.

c) Dé una implementación diferente con costos $O(1)$ en el peor caso para Delete y DeleteMin.

3. Un *stack* no se usa solo para operaciones push, puede también encogerse. Sea k el número actual de elementos, y L el largo del arreglo. Demuestre que si se recorta el arreglo a la mitad cuando se efectúa pop con $k = L/4$, el costo amortizado de las operaciones es a lo más 3. ¿Porqué no usar la cota más natural de ajustar el arreglo si $k = L/2$?

4. Suponiendo la estructura de un contador binario como en el texto, describa una secuencia de n operaciones sobre un contador de k bits, inicialmente 0, de costo $O(nk)$.

   Para manejar decrementos en forma eficiente, usamos "bits" que pueden tomar los valores $-1, 0, 1$ (no solo $0, 1$). Almacenamos el contador en un arreglo $a[k]$, y m es el último "bit" no cero (si todos son cero, definimos $m = -1$). El valor del contador es:

   $$\text{val}(a, m) = \sum_{0 \le i \le m} a[i] \cdot 2^i$$

   Note que $\text{val}(a, n) = 0$ si y solo si $m = -1$. Dé un ejemplo de dos representaciones diferentes de un número.

   Usando los procedimientos de los algoritmos 26.2 y 26.3 para incrementar y decrementar (suponemos largo infinito, $k = \infty$, para simplificar), demuestre que el costo amortizado de cada operación en una secuencia de n incrementos y decrementos sobre un contador inicialmente cero es $O(1)$.

Algoritmo 26.2: Incrementar el contador

```
procedure inc(a, m)
    if m = −1 then
        a[0] ← 1
        m ← 0
    else
        i ← 1
        while a[i] = 1 do
            a[i] ← 0
            i ← i + 1
        end
        a[i] ← a[i] + 1
        if a[i] = 0 ∧ m = i then
            m ← −1
        else
            m ← max(m, i)
        end
    end
```

Algoritmo 26.3: Decrementar el contador

```
procedure dec(a, m)
    if m = −1 then
        a[0] ← −1
        m ← 0
    else
        i ← 0
        while a[i] = −1 do
            a[i] ← 0
            i ← i + 1
        end
        a[i] ← a[i] − 1
        if a[i] = 0 ∧ m = i then
            m ← −1
        else
            m ← max(m, i)
        end
    end
```

Bibliografía

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald R. Rivest, and Clifford Stein: *Introduction to Algorithms*. MIT Press, third edition, 2009.

[2] Wikipedia: *C Standard Library*. https://en.wikipedia.org/wiki/C_standard_library, November 2016. Accessed 2016-11-07.

Clase 27

Pairing Heaps

En algunas aplicaciones de colas de prioridad es importante la operación de modificar la prioridad de un elemento (particularmente en algoritmos que trabajan sobre grafos, por ejemplo, el algoritmo de Dijkstra). Esta operación es provista en forma eficiente en el Fibonacci heap, de Fredman y Tarjan [2]. Pero estas estructuras son pesadas y complejas de programar, además de lentas en la práctica; Fredman, Sedgewick, Sleator y Tarjan [1] proponen una versión simplificada que llaman *pairing heaps*, que discutiremos acá. Hay varias variantes de la misma idea general, analizadas experimentalmente por Stasko y Vitter [5], una estructura afín nueva es el *rank-pairing heap* de Haeupler, Sen y Tarjan [3], quienes revisan las diversas estructuras propuestas; el resumen y análisis más reciente de estas y otras variantes es el de Iacono y Özkan [4]. El consenso parece ser que las diferencias son menores, con *pairing heaps* la más simple y algo más eficiente en la práctica.

27.1. Operaciones a soportar

Una estructura *heap* (una ruma, en castellano; para computines es una cola de prioridad) es una estructura que contiene un número finito de ítem, cada uno con una *clave*. Las operaciones a soportar por una cola de prioridad son:

make_heap: Retorna un nuevo *heap* vacío.

insert(H, x): Inserte el ítem x en el *heap* H, que no contiene x previamente. La clave se supone que es parte de x.

find_min(H): Retorna un ítem con clave mínima en H, sin cambiar éste. En caso que H esté vacío, retorna algún ítem especial nulo.

delete_min(H): Retorna un ítem con clave mínima en H, eliminándolo del *heap*. En caso que H esté vacío, retorna algún ítem especial nulo.

Es claro que find_min corresponde a efectuar delete_min seguido por insert, pero puede ofrecerse una versión más eficiente de esta operación común.

Operaciones menos comunes, pero importantes en algunas aplicaciones concretas, son las siguientes:

meld(H_1, H_2): Retorna un nuevo *heap*, conteniendo los elementos de H_1 y H_2, que se destruyen. Es prerequisito que H_1 y H_2 no tengan ítem en común.

decrease_key(H, x, Δ): Disminuye la clave del elemento x, miembro de H, en Δ.

delete(H, x): Elimina el elemento x miembro de H.

Sabemos que ordenar requiere $\Omega(n \log n)$ comparaciones para ordenar n elementos si solo se permiten comparaciones entre elementos, con lo que el costo amortizado de n operaciones insert y delete_min es $\Omega(\log n)$.

27.2. La estructura *pairing heap*

La idea es representar el *heap* en forma de árbol, con cada nodo conteniendo un ítem con clave menor que sus descendientes, como muestra la figura 27.1. Por ahora supondremos esta representa-

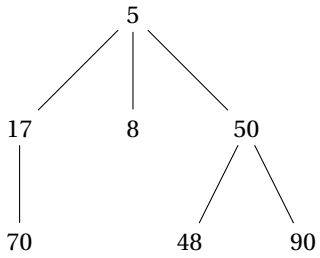


Figura 27.1 – Un ejemplo de *pairing heap*

ción abstracta, más adelante discutiremos una estructura concreta.

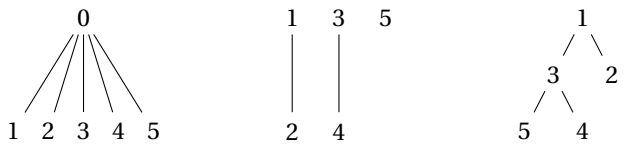
Obtener el mínimo (find_min(H)) es acceder a la raíz, la operación meld(H_1, H_2) es poner el árbol con raíz mayor debajo de la raíz del otro (elegidos arbitrariamente en caso de empate). La operación insert(H, x) es crear un nuevo árbol con x de raíz, y luego unirlo con el existente. Todas tienen costo constante.

Efectuar decrease_key(H, x, Δ) podría hacerse ajustando la clave de x siempre que no resulte menor que la de su padre, o tomando el *heap* con x de raíz, eliminando x de él y uniéndolo con el resto. Por simplicidad, usaremos siempre la segunda opción (acceder al padre es costoso en la estructura que plantearemos más adelante).

La operación central, delete_min(H) es más delicada. Al eliminar la raíz, quedan varios árboles huérfanos, que deben unirse. En el peor caso, todos los demás ítem son raíces, con lo que el costo de hacer esto en forma natural es $O(n)$. La propuesta más simple, que llaman *two pass pairing heaps*, es como sigue: Al eliminar la raíz, quedan cero o más árboles. Estos los unimos a pares, partiendo desde el más nuevo (suponemos que se agregan árboles a la izquierda), y los pares se acumulan luego del más viejo al más nuevo (de derecha a izquierda, en nuestro orden). La figura 27.2 muestra la operación. Debe considerarse que los nodos hijos a su vez son raíces de árboles, que se mantienen intactos. Nótese que el efecto es crear árboles con menos hijos, en el ejemplo la primera operación delete_min es cara, pero las siguientes serán baratas.

27.3. Estructura concreta

Dadas las operaciones que estamos efectuando con los árboles, es natural la representación enlazada mediante punteros al primer hijo y una lista doblemente enlazada de los hijos, con los



(a) Original (b) Parear (c) Acumular

Figura 27.2 – Operación delete_min

punteros extremos de la lista apuntando al padre, junto con indicación si el nodo es primero o último en la lista. Esto permite agregar o eliminar elementos en tiempo $O(1)$, y tener acceso al padre por ejemplo en caso que se quede sin hijos. En esta representación todas las operaciones toman tiempo $O(1)$, salvo delete_min, delete y decrease.

27.4. Análisis amortizado

Definimos el *tamaño* de x en un un árbol, anotado $s(x)$, como el número de nodos en el subárbol con x de raíz (incluyendo a x); y su *rango* como $r(x) = \log_2 s(x)$. Usamos el método potencial, con función potencial de un conjunto de árboles:

$$\Phi(H) = \sum_{x \in H} r(x) \tag{27.1}$$

El potencial de un conjunto vacío de árboles es 0, y el potencial nunca es negativo.

Observamos que el rango de un nodo en un árbol de n nodos está entre 0 y $\log_2 n$. Las operaciones make_heap y find_min no afectan a Φ, ya que no cambian el rango de ningún nodo; su costo amortizado es $O(1)$. Si el número total de nodos es n, las operaciones insert, meld y decrease_key tienen costo amortizado $O(\log n)$, cada una de ellas causa un aumento de potencial acotado por $\log_2 n + 1$. Esto porque la raíz menor aumenta de rango, y en menos de $\log_2 n$ (adquiere cuando más $n-1$ nuevos descendientes); y en caso de insert estamos agregando un nuevo nodo, que aporta 1.

La operación delete_min(H) es más difícil de analizar. Sea H un *heap* de n nodos de raíz x, y llamemos x_i el i-ésimo hijo de x. Buscamos acotar el número de operaciones al reconstruir un *heap* de los árboles con raíces $x_1, \ldots, x_k$. El tiempo de ejecución de esta operación es uno más de los enlaces de nodos efectuados. Los enlaces en el primer paso (parear) son al menos tantos como en el segundo paso (acumular). Cargaremos 2 por enlace en el primer paso.

Ejercicios

1. Escriba las operaciones de *pairing heap* como una clase por ejemplo en C++ o Java.

Bibliografía

[1] Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan: *The pairing heap: A new form of self-adjusting heap.* Algorithmica, 1(1):111–129, November 1986.

[2] Michael L. Fredman and Robert E. Tarjan: *Fibonacci heaps and their uses in improved network optimization algorithms.* Journal of the ACM, 34(3):596–615, July 1987.

[3] Bernhard Haeupler, Siddharta Sen, and Robert E. Tarjan: *Rank-pairing heaps.* SIAM Journal of Computing, 40(6):1463–1485, 2011.

[4] John Iacono and Özgür Özkan: *Why some heaps support constant-amortized-time decrease-key operations, and others do not.* In J. Esparza, P. Fraigniaud, T. Husfeldt, and E. Koutsouias (editors): *International Colloquium on Automata, Languages and Programming*, volume 8572 of *Lecture Notes in Computer Science*, pages 637–649. Springer, 2014.

[5] John T. Stasko and Jeffrey Scott Vitter: *Pairing heaps: Experiments and analysis.* Communications of the ACM, 30(3):234–249, March 1987.

Clase 28

Algoritmos Aleatorizados

Un área activa de investigación reciente son los *algoritmos aleatorizados* [12, 18] (fea traducción de *randomized algorithms* a la que nos obliga la RAE). En la visión tradicional de algoritmos deterministas nos interesan algoritmos que resuelven el problema *correctamente* (siempre) y *rápidamente* (típicamente, esperamos una solución en plazo polinomial en el tamaño de la entrada). Un algoritmo aleatorizado toma, además de la entrada, una secuencia de números aleatorios que usa para tomar decisiones durante la ejecución. Nótese que el comportamiento puede ser diferente incluso con la misma entrada. El interés es que en muchos casos un algoritmo aleatorizado es más simple y usa menos recursos (en promedio) que alternativas deterministas. Textos en el área son el clásico de Motwani y Raghavan [17] y el más accesible de Hromkovič [11]

28.1. Clasificación de algoritmos aleatorizados

Los algoritmos aleatorizados se clasifican en dos grandes ramas, algoritmos de *Monte Carlo* y *Las Vegas* (por las famosas ciudades de juegos, nombres propuestos por Lázló Babai [3], en analogía a los métodos de Monte Carlo usados en análisis numérico, física estadística y simulación, aplicados ya en el proyecto Manhattan).

28.1.1. Algoritmos de Monte Carlo

Un algoritmo de Monte Carlo tiene un tiempo de ejecución fijo, puede dar una respuesta incorrecta (típicamente con baja probabilidad). Es importante que las probabilidades y valores esperados involucrados son sobre las elecciones aleatorias del algoritmo, independientes de la entrada. Repitiendo el algoritmo suficientes veces la probabilidad de error disminuye exponencialmente con el número de corridas.

Términos relevantes son *errores unilaterales* (*one-sided error*) y bilaterales (*two-sided error*). Un algoritmo para resolver un problema de decisión tiene error unilateral si siempre que se equivoca es en el mismo sentido: *preferencia falso* (*false biased*) si siempre está en lo correcto si retorna falso, puede equivocarse si retorna verdadero; *preferencia verdadero* (*true biased*) si siempre está en lo correcto si retorna verdadero, puede equivocarse si retorna falso. En el caso de error bilateral, puede equivocarse en ambas direcciones.

Un algoritmo de Monte Carlo con preferencia falso, por ejemplo, puede usarse para determinar con alta probabilidad que la respuesta es "verdadero": podemos correr el algoritmo suficientes veces. Si alguna vez responde "falso", sabemos que esa es la respuesta; si en n corridas independientes nunca responde "falso" sabemos con alta probabilidad que la respuesta es "verdadero". En caso de un algoritmo con error bilateral, lo ejecutamos múltiples veces y quedamos con la respuesta mayoritaria.

28.1.2. Algoritmos de Las Vegas

Un algoritmo de Las Vegas siempre da el resultado correcto, pero no hay plazo definido. Usualmente se pide además que el valor esperado del tiempo de ejecución (dependiente de las elecciones aleatorias dentro del algoritmo) sea finito.

Al analizar el tiempo de ejecución de estos algoritmos, éste es una variable aleatoria (dependiente de los valores elegidos). Usaremos las siguientes definiciones para discutirlos.

Definición 28.1. Llamaremos $\mathrm{RT}(u)$ al tiempo de ejecución del algoritmo con entrada u.

Definición 28.2. El tiempo esperado de ejecución del algoritmo para entradas de tamaño n es:

$$T(n) = \max_{|u|=n} \mathbb{E}[\mathrm{RT}(u)]$$

Cuidado, un algoritmo aleatorio puede tener tiempo máximo de ejecución exponencial en n, o incluso ilimitado, aún si tiene buen tiempo de ejecución promedio. Por ejemplo, el tiempo de ejecución del algoritmo 28.1 es una variable con distribución geométrica de probabilidad $1/2$, con lo que, medido en número de invocaciones de RandBit():

$$\mathbb{E}[\mathrm{RT}] = 2$$

Sin embargo, en el peor caso nunca termina (si RandBit() siempre retorna 0).

Algoritmo 28.1: Perder el tiempo

```
while RandBit() = 1 do
    Nada
end
```

Otra definición relevante es:

Definición 28.3. Decimos que el tiempo de ejecución del algoritmo A es $O(f(n))$ *con alta probabilidad* si hay constantes $c > 0$ y $d \geq 1$ tales que:

$$\Pr[\mathrm{RT}(A) \geq c \cdot f(n)] \leq \frac{1}{n^d}$$

Requeriremos también que:

$$\mathbb{E}[\mathrm{RT}(A)] = O(f(n))$$

Nótese que hay varias definiciones alternativas, que difieren en ciertos detalles. Adoptaremos ésta, mientras no haya consenso.

28.2. Ámbitos de aplicación

Algoritmos aleatorizados son usados en el ámbito de teoría de números (el método de Miller-Rabin [16, 21], un algoritmo de Monte Carlo para verificar primalidad, es el más usado actualmente).

Una variante de Quicksort elige el pivote al azar, para hacer poco probable el peor caso (e incidentalmente complicarle la vida a un adversario que quiera forzar el peor caso). Estructuras de datos interesantes son *skip lists* [20] y *treaps* [2, 23] (una mezcla de *tree* con *heap*), usan elecciones aleatorias para obtener buen rendimiento en promedio. Estos son todos ejemplos de algoritmos Las Vegas.

Se usan valores aleatorios en muchos algoritmos distribuidos, para evitar *deadlock*, para obtener consensos o para romper empates.

Aplicaciones teóricas incluyen demostraciones probabilísticas de existencia: demostrar que un objeto con alguna característica especial debe aparecer con probabilidad no nula al elegir al azar entre una población adecuada demuestra que el objeto debe existir. Esta es la base del método probabilístico, que fue popularizado y aplicado ampliamente por Paul Erdős. Texto clásico es el de Alon y Spencer [1].

Una técnica para diseñar algoritmos deterministas es tomar un algoritmo aleatorizado y "desaleatorizarlo". Esto es relevante en la práctica, pero su importancia principal está en desentrañar la relación entre las clases de complejidad correspondientes.

28.3. Paradigmas de aplicación

Algunas de las razones que hacen útil un algoritmo aleatorizado son las siguientes.

Frustrar a un adversario Si un adversario puede aprovechar el comportamiento de un algoritmo determinista, haciendo que el algoritmo tome decisiones al azar dificulta ataques. Es una posible defensa frente a *ataques algorítmicos* (ver por ejemplo Crosby y Wallach [6] y McIllroy y Douglas [15]).

Muestreo En muchos casos, extraemos una pequeña muestra de una gran población para inferir propiedades de la población. Computación con pequeñas muestras es barato, sus propiedades pueden guiar el cálculo de propiedades de la población.

Abundancia de testigos Muchos problemas computacionales de traducen en hallar un *testigo* (o un certificado) que permita verificar una hipótesis eficientemente. Por ejemplo, para demostrar que un número es compuesto, basta exhibir un factor no trivial. Para muchos problemas, los testigos son parte de una población demasiado grande para ser revisada sistemáticamente. Si este espacio contiene un número relativamente grande de testigos, un elemento elegido al azar es probable que sea un testigo. Aún más, muestreando repetidas veces y no hallar un testigo disminuye exponencialmente la probabilidad de que haya un testigo, o sea que la hipótesis se cumple.

Huellas digitales Una *huella digital* (en inglés, *fingerprint*) es la imagen de un elemento de un gran universo en uno mucho menor. Huellas digitales obtenidas mediante mapas al azar tienen muchas propiedades útiles. Veremos un par de ejemplos.

Reordenar al azar Muchos problemas tienen la propiedad que un algoritmo bastante ingenuo se comporta extremadamente bien bajo el supuesto de datos entregados ordenados al azar. Aún si el algoritmo tiene mal peor caso, reordenar hace que el peor caso sea extremadamente improbable.

Balance de carga Cuando hay que elegir entre diversos recursos, elegir al azar puede usarse para "repartir" la carga en forma pareja. Esto resulta particularmente interesante cuando las decisiones deben tomarse en sistemas distribuidos, en forma local sin conocimiento global.

Aislar y quebrar simetría En computación distribuida, es común necesitar romper un *deadlock* o simetría, o elegir un valor común (acordar un ordenamiento al azar de las soluciones, y luego buscar la primera solución por separado).

28.4. Balance de carga

Supongamos un nuevo sitio social, *MalaLeche*. Agrupa a gente dada a reclamar por todo, y pelearse por los temas más triviales. Como el tráfico es alto, se ha determinado que se requieren varios procesadores. Si alguna de las máquinas se ve sobrepasada, el rendimiento sufre (con los consiguientes reclamos de los usuarios). Un experimento fue asignar tareas por las primeras letras de los mensajes, pero peleas sobre "**pr**eferencia de editor, emacs o vi" y "**pr**oyecto hidroeléctrico" produjeron serios problemas. Si se conociera el detalle de las tareas de antemano, asignarlas de forma óptima entre máquinas es una variante del problema BIN PACKING, que se sabe NP-completo. Hay soluciones aproximadas, pero si no se conoce de antemano el detalle de las tareas, esto no tiene caso tampoco. Los desarrolladores abandonaron, y asignaron tareas al azar a las máquinas. Para su sorpresa, el sistema funciona sin problemas.

Resulta que asignación al azar no solo balancea la carga razonablemente bien, también permite dar garantías de rendimiento. En general, conviene considerar un esquema aleatorizado si un sistema determinista es demasiado difícil o requiere información que simplemente no está disponible.

Específicamente, MalaLeche recibe 24 000 peticiones en cada período de 10 minutos. Se ha determinado que las peticiones toman a lo más 1 [s] de procesamiento; aunque la mayoría son triviales (quejarse de la ortografía del mensaje precedente y similares), siendo el tiempo promedio de ejecución 0,25 [s]. Midiendo el trabajo en unidades de 1 [s] de procesamiento, si a alguno de los servidores se le asignan más de 600 unidades de trabajo en 10 minutos, se cae y produce problemas. La carga total de MalaLeche de $24\,000 \cdot 0{,}25 = 6\,000$ unidades de trabajo cada 10 minutos indica que se requieren 10 máquinas trabajando al 100 % con balance de carga perfecto. Necesitaremos más de 10 para acomodar fluctuaciones en la carga y balance imperfecto de carga, la pregunta es cuántos se requieren.

Específicamente, nos interesa el número m de servidores que hace muy poco probable que alguno se vea sobrecargado al asignarle más de 600 unidades de trabajo en un período de 10 minutos.

Primero, acotemos la probabilidad de que el primer servidor se sobrecargue en un período dado. Sea T las unidades de trabajo asignadas a ese servidor, buscamos una cota superior a $\Pr[T \geq 600]$. Sea t_i el tiempo que la primera máquina dedica a la tarea i, con lo que $t_i = 0$ si se asigna a otra máquina. Así, con $n = 24\,000$:

$$T = \sum_{1 \leq i \leq n} t_i$$

Podemos usar las cotas de Chernoff si las variables son independientes y en el rango $[0, 1]$. La primera condición se cumple si la asignación de tareas a los servidores no depende de su tiempo de ejecución, la segunda se da porque ninguna tarea toma más de una unidad.

Hay 24 000 tareas, cada una de tiempo de procesamiento esperado de 0,25 [s]. Asignado tareas al

azar a los servidores, la carga esperada para el primer servidor es:

$$\begin{aligned}\mathbb{E}[T] &= \frac{24\,000 \cdot 0{,}25}{m} \\ &= \frac{6000}{m}\end{aligned}$$

Como vimos, con $m < 10$, esperamos que se sobrecargue, con $m = 10$ está al 100 % de capacidad.

Nos interesa el límite:

$$600 = c\mathbb{E}[T]$$

con lo que $c = m/10$. La cota de Chernoff es:

$$\begin{aligned}\Pr[T \geq 600] &= \Pr[T \geq c\mathbb{E}[T]] \\ &\leq \mathrm{e}^{-\beta(c)\cdot 6000/m}\end{aligned}$$

Por la cota de la unión, la probabilidad de que *alguna* de las máquinas se sobrecargue es:

$$\begin{aligned}\Pr[\text{alguna máquina se sobrecarga}] &\leq \sum_{1\leq i\leq m} \Pr[\text{el servidor } i \text{ se sobrecarga}] \\ &= m\Pr[\text{el servidor 1 se sobrecarga}] \\ &\leq m\mathrm{e}^{-\beta(c)\cdot 6000/m}\end{aligned}$$

Algunos valores se tabulan a continuación:

$$\begin{aligned}m = 11&: \quad 0{,}784\ldots \\ m = 12&: \quad 0{,}000999\ldots \\ m = 13&: \quad 0{,}0000000760\ldots\end{aligned}$$

O sea, con 11 máquinas alguna puede caerse casi inmediatamente, 12 debieran durar unos días, y 13 dan para un siglo o dos.

28.5. Cotas inferiores a números de Ramsey

Una aplicación del método probabilístico es la demostración de Erdős [8] de una cota inferior para el número de Ramsey $R(r, r)$.

El teorema de Ramsey [22] en la forma que nos interesa dice que todo grafo de tamaño $R(r, s)$ contiene una *clique* de tamaño r (un K_r) o un conjunto independiente (vértices que no están unidos por arcos) de tamaño s. Esto generalmente se expresa en términos de un grafo K_n cuyos arcos se colorean de rojo y azul, la pregunta es el mínimo n para el cual todo coloreo de arcos de K_n con rojo y azul tiene K_r rojo o K_s azul. El punto del teorema de Ramsey es que $R(r, s)$ es finito. Obtener sus valores ha demostrado ser extraordinariamente difícil.

Por ejemplo, tenemos el siguiente resultado:

Teorema 28.1. $R(3,3) = 6$

Demostración. Consideremos K_6, con sus arcos coloreados de rojo o azul. Elija v entre sus vértices. Entre los 5 vértices restantes, hay al menos 3 unidos con arcos del mismo color a v, digamos que es rojo. Si un par de estos tres vértices están conectados por un arco rojo, con v forman un K_3 rojo. Si no, están unidos por arcos azules entre sí, y forman un K_3 azul. Esto demuestra que $R(3,3) \leq 6$.

Por otro lado, la figura 28.1 muestra K_5 con arcos coloreados de rojo y azul que no contiene K_3 del mismo color, mostrando que $R(6,6) > 5$. □

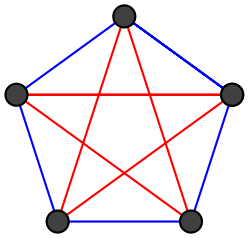


Figura 28.1 – K_5 sin K_3 monocromático

Supongamos que tenemos un grafo completo K_n, y coloreamos sus arcos de rojo y azul. Queremos demostrar que para valores suficientemente pequeños de n no hay r vértices los arcos entre los cuales son todos rojos o azules.

Coloreemos el grafo al azar, asignándole el color rojo o azul a cada arco independientemente con la misma probabilidad. Calculamos el número esperado de grafos monocromáticos de r vértices como sigue: Para un conjunto S de vértices, sea $X(S) = 1$ si todos los arcos entre vértices en S son del mismo color, $X(S) = 0$ en caso contrario. El número de K_r monocromáticos es simplemente la suma de $X(S)$ sobre todos los subconjuntos de r vértices.

Consideremos un conjunto cualquiera S de r vértices. La probabilidad de que todos los arcos entre ellos sean del mismo color es simplemente:

$$2 \cdot 2^{-\binom{r}{2}}$$

(el factor 2 es por los dos colores). La suma de los valores esperados de $X(S)$ sobre todos los S es:

$$\sum_S \mathbb{E}[X(S)] = \binom{n}{r} 2^{1-\binom{r}{2}}$$

Por la linearidad del valor esperado, esto es:

$$\mathbb{E}\left[\sum_S X(S)\right] = \binom{n}{r} 2^{1-\binom{r}{2}}$$

Pero esto es exactamente el número esperado de r-subgrafos monocromáticos.

Si este valor es menor a 1, como el número de K_r monocromáticos es un entero, debe haber al menos un coloreo en que es menor a 1. Pero el único entero menor a 1 es 0. O sea, si:

$$\binom{n}{r} < 2^{\binom{r}{2}-1}$$

hay un coloreo de los arcos de K_n tal que no contiene K_r rojos ni azules.

28.6. Verificar producto de matrices

Supongamos que debemos verificar el producto de matrices $\mathbf{A} \cdot \mathbf{B} = \mathbf{C}$. Usando el algoritmo tradicional con matrices de $n \times n$ esto toma $O(n^3)$ operaciones, el mejor algoritmo teórico da $O(n^{2,38})$. El algoritmo de Freivalds [9] reduce esto a $O(n^2)$.

Elegimos $\mathbf{a} \in \{0,1\}^n$ uniformemente al azar, y calculamos $\mathbf{A}(\mathbf{Ba})$, comparando con $\mathbf{Ca}$. Esto son tres multiplicaciones de una matriz de $n \times n$ por un vector de largo n, todas demandan $O(n^2)$ operaciones, y una comparación de dos vectores de largo n.

Si $\mathbf{A}\cdot\mathbf{B}=\mathbf{C}$, lo anterior resulta siempre en igualdad. Si $\mathbf{D}=\mathbf{A}\cdot\mathbf{B}-\mathbf{C}\neq\mathbf{0}$, tiene algún elemento no cero, digamos $d_{ij}\neq 0$. Este se multiplica por a_i para dar $\mathbf{Da}=\mathbf{A}(\mathbf{Ba})-\mathbf{Ca}$, por lo que a lo más una de las dos alternativas resultantes puede ser cero. O sea, a lo más la mitad de las elecciones de **a** dice "iguales" erróneamente. Repitiendo k veces, respondiendo "distinto" si alguna vez resultan diferentes y "probablemente iguales" si siempre resultan iguales la probabilidad de error es menor a $1/2^k$. El resultado tiene costo $O(kn^2)$; y multiplicar por **a** no requiere multiplicaciones, solo sumar o no los coeficientes.

28.7. Quicksort – análisis aleatorizado

Supongamos el siguiente modelo de Quicksort: estamos ordenando los valores $[1,n]$, antes de ordenar el arreglo los barajamos, y cada vez elegimos el primer elemento como pivote. Es un elemento al azar, no altera nuestro modelo. Definamos variables indicadoras X_{ij} para todos los pares $i<j$ como 1 si i se compara con j durante la ejecución del algoritmo y 0 en caso contrario. Sorprendentemente, podemos calcular la probabilidad de este evento (y en consecuencia $\mathbb{E}[X_{ij}]$): se comparan solo si uno de los dos valores es elegido como pivote antes de terminar en particiones diferentes. Terminan en particiones diferentes si algún valor k, con $i<k<j$, se elige como pivote, que es exactamente si k aparece antes de i y de j en el arreglo al comenzar el proceso. Otros elementos no afectan el proceso, basta concentrarse en analizar permutaciones del rango $[i,j]$. Tenemos éxito (i se compara con j) solo si la permutación de este rango comienza en i o j, lo que ocurre con probabilidad $2/(j-i+1)$, el valor esperado es:

$$\mathbb{E}[X_{ij}]=\frac{2}{j-i+1}$$

Sumando sobre los pares relevantes tenemos el número promedio de comparaciones:

$$\begin{aligned}
\mathbb{E}\left[\sum_{i<j}X_{ij}\right] &= \sum_{i<j}\mathbb{E}[X_{ij}]\\
&= \sum_{i<j}\frac{2}{j-i+1}\\
&= \sum_{1\le i\le n-1}\ \sum_{i\le j\le n}\frac{2}{j-i+1}\\
&= 2\sum_{1\le i\le n-1}\ \sum_{1\le k\le n-i}\frac{1}{k}\\
&= 2\sum_{1\le i\le n-1}H_{n-i}\\
&= 2\sum_{1\le i\le n-1}H_i\\
&= 2nH_n-2n
\end{aligned}$$

Igual que lo que obtuvimos antes (esta es la suma [5, ecuación 1.9] en el apunte de Fundamentos de Informática).

Pero podemos ir más allá. Supongamos que corremos Quicksort aleatorizado sobre un arreglo de n elementos y paramos la recursión a profundidad $c\ln n$ para una constante c. La pregunta es cuál es la probabilidad que haya una hoja del árbol de llamadas al que no le corresponde ordenar un rango de un único elemento. Llame una división (inducida por un pivote) *buena* si divide el conjunto S en pedazos S_1 y S_2 tales que:

$$\min\{|S_1|,|S_2|\}\ge\frac{1}{3}|S|$$

Si no es así, la llamamos *mala*. Con nuestra suposición que todos los elementos son diferentes, vemos que la probabilidad de una buena división es 1/3.

Cada buena división reduce el tamaño de la partición por un factor de al menos 2/3. Para llegar a un rango de un único elemento requerimos:

$$\begin{aligned} k &= \frac{\ln n}{\ln(3/2)} \\ &= a \ln n \end{aligned}$$

buenas divisiones. Interesa ahora qué tan grande debe ser c para que la probabilidad de menos de $a \ln n$ buenas divisiones sea pequeña.

Consideremos el camino de la raíz a una hoja del árbol de recursión. Divisiones sucesivas son buenas o malas independientemente, podemos usar la cota de Chernoff:

$$\begin{aligned} \Pr\left[\text{número de buenas divisiones} < \frac{1}{3} a \ln n\right] &\leq \mathrm{e}^{-\beta(1/c)\cdot\frac{1}{3}ca\ln n} \\ &= n^{-\beta(1/c)ca/3} \end{aligned}$$

Podemos elegir c de manera que $\beta(1/c)ca/3 \geq 2$ (con $c = 6$ tenemos de sobra).

Por lo anterior, la probabilidad que en *una* rama hayan muy pocas divisiones buenas es a lo más n^{-2}; como hay a lo más n ramas, por la cota de la unión esto significa que la probabilidad que en *alguna* rama hayan pocas divisiones buenas es a lo más n^{-1}. Pero esto es la probabilidad que tome más de $O(n \log n)$ comparaciones, y Quicksort aleatorizado ejecuta $O(n \log n)$ comparaciones con alta probabilidad.

28.8. Comparar por igualdad

Supongamos que tenemos un gran archivo, por ejemplo medios de instalación de su distribución favorita, y quiere verificar que no contiene errores, vale decir, coincide con la versión original. Obviamente, queremos hacer esto sin demasiado cómputo adicional (somos impacientes) ni usando demasiado tráfico de red (somos amarretes). Omitiendo consideraciones criptográficas, el problema es calcular alguna forma de *checksum*, tarea para la cual hay soluciones estándar, como CRC (*Cyclic Redundancy Check*, inventado por Peterson [19]; un rápido resumen da por ejemplo Williams [25]).

Concretamente, supongamos que el archivo de marras es de largo n bits, el original es $\langle a_1, a_2, \ldots, a_n \rangle$, la copia local es $\langle b_1, b_2, \ldots, b_n \rangle$. Algoritmos de *checksum* estándar garantizan que para la "mayoría" de los vectores **a** y **b** van a detectar si no son iguales. Acá la garantía es sobre una distribución de **a** y **b**. Para los algoritmos comunes hay técnicas para "falsificar" CRC (ver por ejemplo Stigge y otros [24]), con lo que esto no es suficiente. Nos interesa únicamente acotar el tráfico de datos entre el origen y nosotros, el costo de cómputo es secundario.

Nos interesa analizar el peor caso, interesa una garantía de la forma: para *todo* par de vectores **a** y **b**, elegiremos algunos valores al azar, y para la mayoría de los valores elegidos el algoritmo detectará si hay diferencias. Esta garantía no depende de "buenos" o "malos" vectores **a** y **b**, solo de posiblemente "malos" valores elegidos.

Nuestro algoritmo se basa en considerar los vectores como coeficientes de polinomios sobre un campo finito $\mathbb{F}_p$. Recordemos el siguiente teorema (ver el apunte de Fundamentos de Informática [5, sección 9.3]):

Teorema 28.2. *Sea $f(x)$ un polinomio no-cero de grado a lo más d sobre un campo. Entonces f tiene a lo más d ceros (hay a lo más d valores de x en el campo tales que $f(x) = 0$).*

El primer paso es elegir un primo $p \in [2n, 4n]$ (el postulado de Bertrand [7] asegura que entre m y $2m$ siempre hay un primo, podemos buscar en el rango hasta hallar uno; los primos son relativamente numerosos, esto no es demasiado costoso). Enseguida, construya los polinomios sobre $\mathbb{F}_p$:

$$a(x) = \sum_{1 \le k \le n} a_k x^k$$
$$b(x) = \sum_{1 \le k \le n} b_k x^k$$

Sea $g(x) = a(x) - b(x)$. Nótese que $g(x)$ es el polinomio cero si y solo si $\mathbf{a} = \mathbf{b}$; si $\mathbf{a} \neq \mathbf{b}$, $g(x)$ es un polinomio de grado a lo más n, que tiene a lo más n ceros. Si seleccionamos $x \in \mathbb{F}_p$ uniformemente al azar, la probabilidad que $g(x) = 0$ es a lo más:

$$\frac{n}{|\mathbb{F}_p|} = \frac{n}{p} \le \frac{1}{2}$$

Esto sugiere el algoritmo 28.2.

Algoritmo 28.2: Comparar archivos remotos

```
Acordar el primo p con el origen
k ← 0
for k ← 1 to m do
    El origen elige x ∈ F_p uniformemente al azar y calcula a(x)
    El origen envía x y a(x)
    Calculamos b(x)
    if a(x) ≠ b(x) then
        return Diferentes
    end
end
return Probablemente iguales
```

Vemos que este algoritmo nunca dice "diferentes" por equivocación, se equivoca cada vez a lo más $1/2$ de las veces, en m iteraciones la probabilidad de error es a lo más 2^{-m}. Intercambiamos m números en $[2n, 4n]$, el tráfico total es $O(m \log n)$.

Otra opción es elegir $p \in [rn, 2rn]$, lo que con una iteración da probabilidad de falla a lo más $1/r$, si esto es 2^{-m} es $r = 2^m$ y los bits intercambiados son $O(\log p) = O(\log n + \log r) = O(\log n + m)$, mejor que lo anterior.

Lo que discutimos acá es un ejemplo de lo que Karp [12] llama *fingerprinting*, representar una estructura grande y compleja por una huella digital pequeña. Si dos estructuras tienen la misma huella digital, es fuerte evidencia de que en realidad son iguales. Una huella elegida al azar dificulta ataques o coincidencias, y puede repetirse para aumentar la confianza.

28.9. Patrón en una palabra

Una tarea común es determinar si un patrón σ aparece en una palabra ω. El algoritmo obvio compara el patrón en cada posición de ω, dando un algoritmo determinista cuadrático ($O(|\sigma| \cdot |\omega|)$). Hay algoritmos deterministas lineales ($O(|\sigma| + |\omega|)$), como el de Knuth-Morris-Pratt [14] y el de Boyer-Moore [4] con la modificación de Galil [10], pero son muy complicados.

Discutiremos el algoritmo de Karp y Rabin [13]. Sigue la idea de fuerza bruta de ubicar el patrón en cada posición, pero en vez de comparar el patrón con la palabra compara una huella digital, fácil de calcular y de actualizar. Para simplificar lo que viene, sean $n = |\sigma|$ y $m = |\omega|$. Sea también $b \geq |\Sigma|$ una base conveniente. Usaremos por ejemplo ω_i para representar el i-ésimo símbolo de ω. Para abreviar, anotaremos además $\omega_{[i,j]}$ para referirnos al rango $\omega_i\omega_{i+1}\ldots\omega_j$. Sea p un primo, elegido al azar en el rango $[1, nm^2]$. Usamos la huella digital (nuestras operaciones son en $\mathbb{Z}_p$):

$$h(\sigma) = \sigma_1 \cdot b^{n-1} + \sigma_2 \cdot b^{n-2} + \cdots + \sigma_n$$

Lo crítico es que es fácil actualizar h al eliminar el primer símbolo y agregar uno nuevo:

$$h(\omega_{[i+1,i+n]}) = (h(\omega_{[i,i+n-1]}) - \omega_i b^{n-1}) \cdot b + \omega_{i+n}$$

Esto da lugar al algoritmo 28.3. Podemos verificar probables calces comparando símbolo a símbolo.

Algoritmo 28.3: El algoritmo de Karp-Rabin para calces de patrones

```
Elija p al azar como indicado
h ← 0
for k ← 1 to n do
    h ← h · b + σ_k
end
s ← 0
for k ← 1 to n do
    s ← s · b + ω_k
end
k ← n
while (s ≠ h) ∧ (k < m) do
    k ← k + 1
    s ← (s − ω_{k−n} · b^{n−1}) · b + ω_k
end
if s = h then
    return probable calce en k − n
else
    return no hay calce
end
```

Si $h(\alpha) \neq h(\beta)$, es claro que $\alpha \neq \beta$. Nos interesa el caso en que $\alpha \neq \beta$, pero $h(\alpha) = h(\beta)$. Un adversario que conoce el funcionamiento de nuestro algoritmo y p podría elegir β para forzar ésto en muchas posiciones, haciendo que debamos recurrir a comparaciones inútiles (llegando a tiempo $O(nm)$).

Bibliografía

[1] Noga Alon and Joel H. Spencer: *The Probabilistic Method*. Wiley Series in Discrete Mathematics and Optimization. John Wiley & Sons, fourth edition, 2015.

[2] Cecilia R. Aragon and Raimund G. Seidel: *Randomized search trees*. In *Thirtieth Annual Symposium on Foundations of Computer Science*, pages 540–545, October – November 1989.

[3] Lázló Babai: *Monte-Carlo algorithms in graph isomorphism testing*. Technical Report D.M.S 79-10, Université de Montreal, 1979.

[4] R. S. Boyer and J. Strother Moore: *A fast string searching algorithm*. Communications of the ACM, 20(10):762–772, October 1977.

[5] Horst H. von Brand: *Fundamentos de Informática*. `git://csrg.inf.utfsm.cl/vonbrand/Ramos/trainwreck`, Agosto 2016. Versión 0.83.

[6] Scott A. Crosby and Dan S. Wallach: *Denial of service via algorithmic complexity attacks*. In *Proceedings of the 12th USENIX Security Symposium*, pages 29–44, 2003.

[7] Paul Erdős: *Beweis eines Satzes von Tschebyschef*. Acta Scientiarum Mathematicarum Szegedensis, 5(3-4):194–198, 1930-1932.

[8] Paul Erdős: *Some remarks on the theory of graphs*. Bulletin of the American Mathematical Society, 53(4):292–294, April 1947.

[9] Rūsinš M. Freivalds: *Probabilistic machines can use less running time*. In *Proceedings of the IFIP Congress*, page 839–842, Toronto, Canada, August 1977. International Federation for Information Processing, North-Holland.

[10] Zvi Galil: *On improving the worst case running time of the Boyer-Moore string matching algorithm*. Communications of the ACM, 22(9):505–508, September 1979.

[11] Juraj Hromkovič: *Design and Analysis of Randomized Algorithms*. Texts in Theoretical Computer Science. Springer, 2005.

[12] Richard M. Karp: *An introduction to randomized algorithms*. Discrete Applied Mathematics, 34(1-3):165–201, November 1991.

[13] Richard M. Karp and Michael O. Rabin: *Efficient randomized pattern-matching algorithms*. IBM Journal of Research and Development, 31(2):249–260, March 1987.

[14] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt: *Fast pattern matching in strings.* SIAM Journal of Computing, 6(2):323–350, 1977.

[15] M. Douglas McIllroy: *A killer adversary for Quicksort.* Software: Practice and Experience, 29(4):341–344, April 1999.

[16] Gary L. Miller: *Riemann's hypothesis and tests for primality.* Journal of Computer and System Sciences, 13(3):300–317, December 1976.

[17] Rajeev Motwani and Prabhakar Raghavan: *Randomized Algorithms.* Cambridge University Press, 1995.

[18] Rajeev Motwani and Prabhakar Raghavan: *Randomized algorithms.* ACM Computing Surveys, 28(1):33–37, March 1996.

[19] W. W. Peterson and D. T. Brown: *Cyclic codes for error detection.* Proceedings of the IRE, 49(1):228–235, January 1961.

[20] William Pugh: *Skip Lists: A probabilistic alternative to balanced trees.* Communications of the ACM, 33(6):668–676, June 1990.

[21] Michael O. Rabin: *Probabilistic algorithm for testing primality.* Journal of Number Theory, 12(1):128–138, February 1980.

[22] Frank P. Ramsey: *On a problem in formal logic.* Proceedings of the London Mathematical Society, s2-30(1):264–268, 1930.

[23] Raimund G. Seidel and Cecilia A. Aragon: *Randomized search trees.* Algorithmica, 16(4/5):464–497, October 1996.

[24] Martin Stigge, Henryk Plötz, Wolf Müller, and Hans Peter Redlich: *Reversing CRC – theory and practice.* Technical Report SAR-PR-2006-05, Humboldt University Berlin, Computer Science Department, May 2006.

[25] Ross N. Williams: *A painless guide to CRC error detection algorithms index v3.00.* http://www.repairfaq.org/filipg/LINK/F_crc_v3.html, September 1996.

Apéndice A

Symbolic Method for Dummies

La idea básica es usar funciones generatrices en forma sistemática en combinatoria. Lo que sigue es un condensado del apunte de Fundamentos de Informática [1, capítulo 21]. Ver también Lumbroso y Morcrette [2].

La idea es tener una *clase* de objetos, que anotaremos mediante letras caligráficas, como $\mathcal{A}$. La clase $\mathcal{A}$ consta de *objetos*, $\alpha \in \mathcal{A}$. Para el objeto α hay una noción de *tamaño*, que anotamos $|\alpha|$ (un número natural, generalmente el número de *átomos* que componen α). Al número de objetos de tamaño n lo anotaremos a_n. Usaremos $\mathcal{A}_n$ para referirnos al conjunto de objetos de la clase $\mathcal{A}$ de tamaño n, con lo que $a_n = |\mathcal{A}_n|$. Condición adicional es que el número de objetos de cada tamaño sea finito.

A las funciones generatrices ordinaria y exponencial correspondientes les llamaremos $A(z)$ y $\widehat{A}(z)$, respectivamente:

$$A(z) = \sum_{\alpha \in \mathcal{A}} z^{|\alpha|} = \sum_{n \geq 0} a_n z^n$$

$$\widehat{A}(z) = \sum_{\alpha \in \mathcal{A}} \frac{z^{|\alpha|}}{|\alpha|!} = \sum_{n \geq 0} a_n \frac{z^n}{n!}$$

Nuestro siguiente objetivo es construir nuevas clases a partir de las que ya tenemos. Debe tenerse presente que como lo que nos interesa es contar el número de objetos de cada tamaño, basta construir objetos con distribución de tamaños adecuada (o sea, relacionados con lo que deseamos contar por una biyección). Comúnmente el tamaño de los objetos es el número de alguna clase de átomos que lo componen. Al combinar objetos para crear objetos mayores los tamaños simplemente se suman.

Las clases más elementales son $\varnothing$, la clase que no contiene objetos; $\mathcal{E} = \{\epsilon\}$, la clase que contiene únicamente el objeto vacío ϵ (de tamaño nulo); y la clase que comúnmente llamaremos $\mathcal{Z}$, conteniendo un único objeto de tamaño uno (que llamaremos ζ por consistencia). Luego definimos operaciones que combinan las clases $\mathcal{A}$ y $\mathcal{B}$ mediante *unión combinatoria* $\mathcal{A} + \mathcal{B}$, en que aparecen los α y los β con sus tamaños (los objetos individuales se "decoran" con su proveniencia, de forma que $\mathcal{A}$ y $\mathcal{B}$ no necesitan ser disjuntos; pero generalmente nos preocuparemos que $\mathcal{A}$ y $\mathcal{B}$ sean disjuntos, o podemos usar el principio de inclusión y exclusión para contar los conjuntos de interés). Ocasionalmente restaremos objetos de una clase, lo que debe interpretarse sin decoraciones (estamos dejando fuera ciertos elementos, simplemente). Usaremos *producto cartesiano* $\mathcal{A} \times \mathcal{B}$, cuyos

elementos son pares (α, β) y el tamaño del par es $|\alpha| + |\beta|$. Otras operaciones son formar *secuencias* de elementos de $\mathscr{A}$ (se anota $\textsc{Seq}(\mathscr{A})$), formar *conjuntos* $\textsc{Set}(\mathscr{A})$ y *multiconjuntos* $\textsc{MSet}(\mathscr{A})$ de elementos de $\mathscr{A}$.

De incluir objetos de tamaño cero en estas construcciones pueden crearse infinitos objetos de un tamaño dado, lo que no es una clase según nuestra definición. Por ello estas construcciones son aplicables sólo si $\mathscr{A}_0 = \varnothing$.

Es importante recalcar las relaciones y diferencias entre las estructuras. En una secuencia es central el orden de las piezas que la componen. Ejemplo son las palabras, interesa el orden exacto de las letras (y estas pueden repetirse). En un conjunto solo interesa si el elemento está presente o no, no hay orden. En un conjunto un elemento en particular está o no presente, a un multiconjunto puede pertenecer varias veces.

Hay dos grandes opciones: Objetos rotulados y no rotulados. Consideramos que el objeto α es *rotulado* si sus átomos componentes tienen identidad, cosa que se representa rotulándolos de 1 a $|\alpha|$. Si los átomos son libremente intercambiables, son objetos *no rotulados*. Un punto que produce particular confusión es que tiene perfecto sentido hablar de secuencias de elementos sin rotular. La secuencia impone un orden, pero elementos iguales se consideran indistinguibles (en una palabra interesa el orden de las letras, pero al intercambiar dos letras iguales la palabra sigue siendo la misma). Estos dos casos requieren tratamiento separado, y en algunos casos operaciones especializadas.

A.1. Objetos no rotulados

Nuestro primer teorema relaciona las funciones generatrices ordinarias respectivas para algunas de las operaciones entre clases definidas antes. Las funciones generatrices de las clases $\varnothing$, $\mathscr{E}$ y $\mathscr{Z}$ son, respectivamente, 0, 1 y z. En las derivaciones de las transferencias de ecuaciones simbólicas a ecuaciones para las funciones generatrices lo que nos interesa es contar los objetos entre manos, recurriremos a biyecciones para ello en algunos de los casos.

Teorema A.1 (Método simbólico, OGF)**.** *Sean $\mathscr{A}$ y $\mathscr{B}$ clases de objetos, con funciones generatrices ordinarias respectivamente $A(z)$ y $B(z)$. Entonces tenemos las siguientes funciones generatrices ordinarias:*

1. *Para enumerar $\mathscr{A} + \mathscr{B}$:*

$$A(z) + B(z)$$

2. *Para enumerar $\mathscr{A} \times \mathscr{B}$:*

$$A(z) \cdot B(z)$$

3. *Para enumerar $\textsc{Seq}(\mathscr{A})$:*

$$\frac{1}{1 - A(z)}$$

4. *Para enumerar $\textsc{Set}(\mathscr{A})$:*

$$\prod_{\alpha \in \mathscr{A}} \left(1 + z^{|\alpha|}\right) = \prod_{n \geq 1} (1 + z^n)^{a_n} = \exp\left(\sum_{k \geq 1} \frac{(-1)^{k+1}}{k} A(z^k)\right)$$

5. *Para enumerar* $\textsc{MSet}(\mathscr{A})$*:*

$$\prod_{\alpha\in\mathscr{A}} \frac{1}{1-z^{|\alpha|}} = \prod_{n\geq 1} \frac{1}{(1-z^n)^{a_n}} = \exp\left(\sum_{k\geq 1} \frac{A(z^k)}{k}\right)$$

6. *Para enumerar* $\textsc{Cyc}(\mathscr{A})$*:*

$$\sum_{n\geq 1} \frac{\phi(n)}{n} \ln \frac{1}{1-A(z^n)}$$

Demostración. Usamos libremente resultados sobre funciones generatrices, ver [1, capítulo 14], en las demostraciones de cada caso. Usaremos casos ya demostrados en las demostraciones sucesivas.

1. Si hay a_n elementos de $\mathscr{A}$ de tamaño n y b_n elementos de $\mathscr{B}$ de tamaño n, habrán $a_n + b_n$ elementos de $\mathscr{A}+\mathscr{B}$ de tamaño n.

   Alternativamente, usando la notación de Iverson (ver [1, sección 1.4]):

$$\sum_{\gamma\in\mathscr{A}+\mathscr{B}} z^{|\gamma|} = \sum_{\gamma\in\mathscr{A}+\mathscr{B}} \left([\gamma\in\mathscr{A}]z^{|\gamma|} + [\gamma\in\mathscr{B}]z^{|\gamma|}\right) = \sum_{\alpha\in\mathscr{A}} z^{|\alpha|} + \sum_{\beta\in\mathscr{B}} z^{|\beta|} = A(z) + B(z)$$

2. Hay:

$$\sum_{0\leq k\leq n} a_k b_{n-k}$$

   maneras de combinar elementos de $\mathscr{A}$ con elementos de $\mathscr{B}$ cuyos tamaños sumen n, y este es precisamente el coeficiente de z^n en $A(z)\cdot B(z)$.

   Alternativamente:

$$\sum_{\gamma\in\mathscr{A}\times\mathscr{B}} z^{|\gamma|} = \sum_{\substack{\alpha\in\mathscr{A}\\ \beta\in\mathscr{B}}} z^{|(\alpha,\beta)|} = \sum_{\substack{\alpha\in\mathscr{A}\\ \beta\in\mathscr{B}}} z^{|\alpha|+|\beta|} = \left(\sum_{\alpha\in\mathscr{A}} z^{|\alpha|}\right)\cdot\left(\sum_{\beta\in\mathscr{B}} z^{|\beta|}\right) = A(z)\cdot B(z)$$

3. Hay una manera de obtener la secuencia de largo 0 (aporta el objeto vacío ϵ), las secuencias de largo 1 son simplemente los elementos de $\mathscr{A}$, las secuencias de largo 2 son elementos de $\mathscr{A}\times\mathscr{A}$, y así sucesivamente. O sea, las secuencias se representan mediante:

$$\mathscr{E} + \mathscr{A} + \mathscr{A}\times\mathscr{A} + \mathscr{A}\times\mathscr{A}\times\mathscr{A} + \cdots$$

   Por la segunda parte y la serie geométrica, la función generatriz correspondiente es:

$$1 + A(z) + A^2(z) + A^3(z) + \cdots = \frac{1}{1-A(z)}$$

4. La clase de los subconjuntos finitos de $\mathscr{A}$ queda representada por el producto simbólico:

$$\prod_{\alpha\in\mathscr{A}} (\mathscr{E} + \{\alpha\})$$

   ya que al distribuir los productos de todas las formas posibles aparecen todos los subconjuntos de $\mathscr{A}$. Directamente obtenemos entonces:

$$\prod_{\alpha\in\mathscr{A}} \left(1 + z^{|\alpha|}\right) = \prod_{n\geq 0} (1+z^n)^{a_n}$$

Otra forma de verlo es que cada objeto de tamaño n aporta un factor $1+z^n$, si hay a_n de estos el aporte total es $(1+z^n)^{a_n}$. Esta es la primera parte de lo aseverado. Aplicando logaritmo:

$$\begin{aligned}\sum_{\alpha\in\mathscr{A}} \ln\left(1+z^{|\alpha|}\right) &= -\sum_{\alpha\in\mathscr{A}}\sum_{k\geq 1}\frac{(-1)^k z^{|\alpha|k}}{k}\\ &= -\sum_{k\geq 1}\frac{(-1)^k}{k}\sum_{\alpha\in\mathscr{A}} z^{|\alpha|k}\\ &= \sum_{k\geq 1}\frac{(-1)^{k+1}A(z^k)}{k}\end{aligned}$$

Exponenciando lo último resulta equivalente a la segunda parte.

5. Podemos considerar un multiconjunto finito como la combinación de una secuencia para cada tipo de elemento:

$$\prod_{\alpha\in\mathscr{A}} \textsc{Seq}(\{\alpha\})$$

La función generatriz buscada es:

$$\prod_{\alpha\in\mathscr{A}}\frac{1}{1-z^{|\alpha|}} = \prod_{n\geq 0}\frac{1}{(1-z^n)^{a_n}}$$

Esto provee la primera parte. Nuevamente aplicamos logaritmo para simplificar:

$$\begin{aligned}\ln\prod_{\alpha\in\mathscr{A}}\frac{1}{1-z^{|\alpha|}} &= -\sum_{\alpha\in\mathscr{A}}\ln\left(1-z^{|\alpha|}\right)\\ &= \sum_{\alpha\in\mathscr{A}}\sum_{k\geq 1}\frac{z^{k|\alpha|}}{k}\\ &= \sum_{k\geq 1}\frac{1}{k}\sum_{\alpha\in\mathscr{A}} z^{k|\alpha|}\\ &= \sum_{k\geq 1}\frac{A(z^k)}{k}\end{aligned}$$

6. Esta situación es más compleja de tratar, vea la discusión en [1, sección 21.2.3]. □

A.1.1. Algunas aplicaciones

La clase de los árboles binarios $\mathscr{B}$ es por definición es la unión disjunta del árbol vacío y la clase de tuplas de un nodo (la raíz) y dos árboles binarios. O sea:

$$\mathscr{B} = \mathscr{E} + \mathscr{Z}\times\mathscr{B}\times\mathscr{B}$$

de donde directamente obtenemos:

$$B(z) = 1 + zB^2(z)$$

Con el cambio de variable $u(z) = B(z) - 1$ queda:

$$u(z) = z(1+u(z))^2$$

Es aplicable la fórmula de inversión de Lagrange [1, teorema 17.8] con $\phi(u) = (u+1)^2$ y $f(u) = u$:

$$\begin{aligned}
[z^n]\, u(z) &= \frac{1}{n}\left[u^{n-1}\right]\phi(u)^n \\
&= \frac{1}{n}\left[u^{n-1}\right](u+1)^{2n} \\
&= \frac{1}{n}\left[u^{n-1}\right]\sum_{k\geq 0}\binom{2n}{k}u^k \\
&= \frac{1}{n}\binom{2n}{n-1} \\
&= \frac{1}{n}\,\frac{(2n)!}{(n+1)!(n-1)!} \\
&= \frac{1}{n+1}\,\frac{(2n)!}{(n!)^2} \\
&= \frac{1}{n+1}\binom{2n}{n}
\end{aligned}$$

Tenemos, como $u(z) = B(z) - 1$ y sabemos que $b_0 = 1$:

$$b_n = \begin{cases} \dfrac{1}{n+1}\dbinom{2n}{n} & \text{si } n \geq 1 \\ 1 & \text{si } n = 0 \end{cases}$$

Casualmente la expresión simplificada para $n \geq 1$ da el valor correcto $b_0 = 1$. Estos son los números de Catalan, es $b_n = C_n$.

Sea ahora $\mathscr{A}$ la clase de *árboles con raíz ordenados*, formados por un nodo raíz conectado a las raíces de una secuencia de árboles ordenados. La idea es que la raíz tiene hijos en un cierto orden. Simbólicamente:

$$\mathscr{A} = \mathscr{Z} \times \textsc{Seq}(\mathscr{A})$$

El método simbólico entrega directamente la ecuación:

$$A(z) = \frac{z}{1 - A(z)}$$

Nuevamente es aplicable la fórmula de inversión de Lagrange, con $\phi(A) = (1-A)^{-1}$ y $f(A) = A$:

$$\begin{aligned}
[z^n]\, A(z) &= \frac{1}{n}\left[A^{n-1}\right]\phi(A)^n \\
&= \frac{1}{n}\left[A^{n-1}\right](1-A)^{-n} \\
&= \frac{1}{n}\binom{2n-2}{n-1} \\
&= C_{n-1}
\end{aligned}$$

Otra vez números de Catalan.

La manera obvia de representar $\mathbb{N}_0$ es por secuencias de marcas, como |||| para 4; simbólicamente $\mathbb{N}_0 = \textsc{Seq}(\mathscr{Z})$. Para calcular el número de multiconjuntos de k elementos tomados entre n, un

multiconjunto queda representado por las cuentas de los n elementos de que se compone, y eso corresponde a:

$$\mathbb{N}_0 \times \cdots \times \mathbb{N}_0 = (\textsc{Seq}(\mathcal{Z}))^n$$

Para obtener el número que nos interesa:

$$\begin{aligned}\left(\!\binom{n}{k}\!\right) &= \left[z^k\right](1-z)^{-n} \\ &= (-1)^n\binom{-n}{k} \\ &= \binom{n+k-1}{n}\end{aligned}$$

Una *combinación* de n es expresarlo como una suma. Por ejemplo, hay 8 combinaciones de 4:

$$4 = 3+1 = 2+2 = 2+1+1 = 1+3 = 1+2+1 = 1+1+2 = 1+1+1+1$$

Llamemos $c(n)$ al número de combinaciones de n. Con la misma idea anterior, $\mathbb{N} = \mathcal{Z} \times \textsc{Seq}(\mathcal{Z})$, que da:

$$N(z) = \frac{z}{1-z}$$

A su vez, una combinación no es más que una secuencia de naturales (separados por +):

$$\mathcal{C} = \textsc{Seq}(\mathbb{N})$$

Directamente resulta:

$$\begin{aligned}C(z) &= \sum_{n\geq 0} c(n)z^n \\ &= \frac{1}{1-N(z)} \\ &= \frac{1}{2} + \frac{1}{2}\cdot\frac{1}{1-2z} \\ c(n) &= [z^n]C(z) \\ &= \frac{1}{2}[n=0] + \frac{1}{2}\cdot 2^n \\ &= \frac{1}{2}[n=0] + 2^{n-1}\end{aligned}$$

Esto es consistente con $c(4) = 8$ obtenido arriba.

A.2. Objetos rotulados

En la discusión previa solo interesaba el tamaño de los objetos, no su disposición particular. Consideraremos ahora objetos rotulados, donde importa cómo se compone el objeto de sus partes (los átomos tienen identidades, posiblemente porque se ubican en orden).

El objeto más simple con partes rotuladas son las permutaciones (biyecciones $\sigma\colon [n] \to [n]$, podemos considerarlas secuencias de átomos numerados). Para la función generatriz exponencial tenemos, ya que hay $n!$ permutaciones de n elementos:

$$\sum_{\sigma} \frac{z^{|\sigma|}}{|\sigma|!} = \sum_{n\geq 0} n!\,\frac{z^n}{n!} = \frac{1}{1-z}$$

Lo siguiente más simple de considerar es colecciones de ciclos rotulados. Por ejemplo, escribimos (1 3 2) para el objeto en que viene 3 luego de 1, 2 sigue a 3, y a su vez 1 sigue a 2. Así (2 1 3) es solo otra forma de anotar el ciclo anterior, que no es lo mismo que (3 1 2). Interesa definir formas consistentes de combinar objetos rotulados. Por ejemplo, al combinar el ciclo (1 2) con el ciclo (1 3 2) resultará un objeto con 5 rótulos, y debemos ver cómo los distribuimos entre las partes. El cuadro A.1 reseña las posibilidades al respetar el orden de los elementos asignados a cada parte. Es claro que

(1 2)(3 5 4)	(2 3)(1 5 4)	(3 4)(1 5 2)	(4 5)(1 3 2)
(1 3)(2 5 4)	(2 4)(1 5 3)	(3 5)(1 4 2)	
(1 4)(2 5 3)	(2 5)(1 4 3)		
(1 5)(2 4 3)			

Cuadro A.1 – Combinando los ciclos (1 2) y (1 3 2)

lo que estamos haciendo es elegir un subconjunto de 2 rótulos de entre los 5 para asignárselos al primer ciclo. El combinar dos clases de objetos $\mathscr{A}$ y $\mathscr{B}$ de esta forma lo anotaremos $\mathscr{A} \star \mathscr{B}$.

Otra operación común es la *composición*, anotada $\mathscr{A} \circ \mathscr{B}$. La idea es elegir un elemento $\alpha \in \mathscr{A}$, luego elegir $|\alpha|$ elementos de $\mathscr{B}$, y reemplazar los $\mathscr{B}$ por las partes de α, en el orden que están rotuladas; para finalmente asignar rótulos a los átomos que conforman la.estructura completa respetando el orden de los rótulos al interior de los $\mathscr{B}$ (igual como lo hicimos para $\star$).

Ocasionalmente es útil *marcar* uno de los componentes del objeto, operación que anotaremos $\mathscr{A}^\bullet$. Usaremos también la construcción $\textsc{MSet}(\mathscr{A})$, que podemos considerar como una secuencia de elementos numerados obviando el orden. Cuidado, muchos textos le llaman $\textsc{Set}()$ a esta operación.

Tenemos el siguiente teorema:

Teorema A.2 (Método simbólico, EGF)**.** *Sean $\mathscr{A}$ y $\mathscr{B}$ clases de objetos, con funciones generatrices exponenciales $\widehat{A}(z)$ y $\widehat{B}(z)$, respectivamente. Entonces tenemos las siguientes funciones generatrices exponenciales:*

1. *Para enumerar $\mathscr{A}^\bullet$:*

   $$z\mathrm{D}\widehat{A}(z)$$

2. *Para enumerar $\mathscr{A} + \mathscr{B}$:*

   $$\widehat{A}(z) + \widehat{B}(z)$$

3. *Para enumerar $\mathscr{A} \star \mathscr{B}$:*

   $$\widehat{A}(z) \cdot \widehat{B}(z)$$

4. *Para enumerar $\mathscr{A} \circ \mathscr{B}$:*

   $$\widehat{A}(\widehat{B}(z))$$

5. *Para enumerar $\textsc{Seq}(\mathscr{A})$:*

   $$\frac{1}{1 - \widehat{A}(z)}$$

6. *Para enumerar $\textsc{MSet}(\mathscr{A})$:*

   $$\mathrm{e}^{\widehat{A}(z)}$$

7. *Para enumerar* $\textsc{Cyc}(\mathscr{A})$*:*

$$-\ln(1-\widehat{A}(z))$$

Demostración. Usaremos casos ya demostrados en las demostraciones sucesivas.

1. El objeto $\alpha \in \mathscr{A}$ da lugar a $|\alpha|$ objetos al marcar cada uno de sus átomos, lo que da la función generatriz exponencial:

$$\sum_{\alpha\in\mathscr{A}} |\alpha| \frac{z^{|\alpha|}}{|\alpha|!}$$

   Esto es lo indicado.

2. Nuevamente trivial.

3. El número de objetos γ que se obtienen al combinar $\alpha \in \mathscr{A}$ con $\beta \in \mathscr{B}$ es:

$$\binom{|\alpha|+|\beta|}{|\alpha|}$$

   y tenemos la función generatriz exponencial:

$$\sum_{\gamma\in\mathscr{A}\star\mathscr{B}} \frac{z^{|\gamma|}}{|\gamma|!} = \sum_{\substack{\alpha\in\mathscr{A}\\ \beta\in\mathscr{B}}} \binom{|\alpha|+|\beta|}{|\alpha|} \frac{z^{|\alpha|+|\beta|}}{(|\alpha|+|\beta|)!} = \left(\sum_{\alpha\in\mathscr{A}} \frac{z^{|\alpha|}}{|\alpha|!}\right)\cdot\left(\sum_{\beta\in\mathscr{B}} \frac{z^{|\beta|}}{|\beta|!}\right) = \widehat{A}(z)\cdot\widehat{B}(z)$$

4. Tomemos $\alpha \in \mathscr{A}$, de tamaño $n = |\alpha|$, y n elementos de $\mathscr{B}$ en orden a ser reemplazados por las partes de α. Esa secuencia de $\mathscr{B}$ es representada por:

$$\mathscr{B}\star\mathscr{B}\star\cdots\star\mathscr{B}$$

   con función generatriz exponencial:

$$\widehat{B}^n(z)$$

   Sumando sobre las contribuciones:

$$\sum_{\alpha\in\mathscr{A}} \frac{\widehat{B}^{|\alpha|}(z)}{|\alpha|!}$$

   Esto es lo prometido.

5. Primeramente, podemos describir:

$$\textsc{Seq}(\mathscr{Z}) = \mathscr{E} + \mathscr{Z}\star\textsc{Seq}(\mathscr{Z})$$

   que lleva a ecuación:

$$\widehat{S}(z) = 1 + z\widehat{S}(z)$$

   de donde:

$$\widehat{S}(z) = \frac{1}{1-z}$$

   Aplicando composición se obtiene lo indicado.

6. Hay un único multiconjunto de n elementos rotulados (se rotulan simplemente de 1 a n), con lo que $\text{MSET}(\mathcal{Z})$ corresponde a:

$$\sum_{n\geq 0} \frac{z^n}{n!} = \exp(z)$$

Al aplicar composición resulta lo anunciado.

Otra demostración es considerar el multiconjunto de $\mathcal{A}$, descrito por $\mathcal{M} = \text{MSET}(\mathcal{A})$. Si marcamos uno de los átomos de $\mathcal{M}$ estamos marcando uno de los $\mathcal{A}$, el resto sigue formando un multiconjunto de $\mathcal{A}$:

$$\mathcal{M}^\bullet = \mathcal{A}^\bullet \star \mathcal{M}$$

Por lo anterior:

$$z\widehat{M}'(z) = z\widehat{A}'(z)\widehat{M}(z)$$

Hay un único multiconjunto de tamaño 0, o sea $\widehat{M}(0) = 1$; y hemos impuesto la condición que no hay objetos de tamaño 0 en $\mathcal{A}$, vale decir, $\widehat{A}(0) = 0$. Así la solución a la ecuación diferencial es:

$$\widehat{M}(z) = \exp(\widehat{A}(z))$$

7. Consideremos un ciclo de $\mathcal{A}$, o sea $\mathcal{C} = \text{CYC}(\mathcal{A})$. Si marcamos los $\mathcal{C}$, estamos marcando uno de los $\mathcal{A}$, y el resto es una secuencia:

$$\mathcal{C}^\bullet = \mathcal{A}^\bullet \star \text{SEQ}(\mathcal{A})$$

Esto se traduce en la ecuación diferencial:

$$z\widehat{C}'(z) = z\widehat{A}'(z)\frac{1}{1-\widehat{A}(z)}$$

Integrando bajo el entendido $\widehat{C}(0) = 0$ con $\widehat{A}(0) = 0$ se obtiene lo indicado.

Alternativamente, hay $(n-1)!$ ciclos de n elementos, con lo que para $\text{CYC}(\mathcal{Z})$ obtenemos:

$$\sum_{n\geq 1} (n-1)!\frac{z^n}{n!} = \sum_{n\geq 1} \frac{z^n}{n} = -\ln\frac{1}{1-z}$$

Aplicar composición completa la demostración.

□

A.2.1. Algunas aplicaciones

Un ejemplo simple es el caso de permutaciones, que son simplemente secuencias de elementos rotulados:

$$\begin{aligned}
\mathcal{P} &= \text{SEQ}(\mathcal{Z}) \\
\widehat{P}(z) &= \frac{1}{1-z} \\
P_n &= n![z^n]\widehat{P}(z) \\
&= n!
\end{aligned}$$

Consideremos colecciones de ciclos:

$$\textsc{MSet}(\textsc{Cyc}(\mathcal{Z}))$$

Vemos que esto corresponde a:

$$\exp\left(\ln\frac{1}{1-z}\right) = \frac{1}{1-z}$$

Hay tantas permutaciones de tamaño n como colecciones de ciclos. Una biyección se da ordenando los ciclos de manera que se inicien con su mayor elemento, y listar los ciclos en orden de máximo elemento creciente. Cualquier lista de elementos puede reinterpretarse como ciclos de una única manera de esta forma. En el fondo, podemos representar permutaciones como los ciclos que parten en cada elemento.

Podemos describir permutaciones como un conjunto de elementos que quedan fijos combinado con otros elementos que están fuera de orden (un *desarreglo*, clase $\mathcal{D}$). O sea:

$$\begin{aligned}\mathcal{P} &= \mathcal{D} \star \textsc{MSet}(\mathcal{Z})\\ \frac{1}{1-z} &= \widehat{D}(z)\mathrm{e}^z\\ \widehat{D}(z) &= \frac{\mathrm{e}^{-z}}{1-z}\end{aligned}$$

De acá, por propiedades de las funciones generatrices vemos que:

$$[z^n]\widehat{D}(z) = \sum_{0\le k\le n}\frac{(-1)^k}{k!}$$

y tenemos, usando la notación común para el número de desarreglos de tamaño n:

$$D_n = n!\sum_{0\le k\le n}\frac{(-1)^k}{k!}$$

Una *involución* es una permutación π tal que $\pi\circ\pi$ es la identidad. Es claro que una involución es una colección de ciclos de largos 1 y 2, o sea:

$$\mathcal{I} = \textsc{MSet}(\textsc{Cyc}_{\le 2}(\mathcal{Z}))$$

Revisando la derivación, vemos que $\textsc{Cyc}_{\le 2}(\mathcal{Z})$ corresponde a:

$$\frac{z}{1}+\frac{z^2}{2}$$

y tenemos para la función generatriz:

$$\widehat{I}(z) = \exp\left(\frac{z}{1}+\frac{z^2}{2}\right)$$

Un paquete de álgebra simbólica da:

$$\widehat{I}(z) = 1+1\cdot\frac{z}{1!}+2\cdot\frac{z^2}{2!}+4\cdot\frac{z^3}{3!}+10\cdot\frac{z^4}{4!}+26\cdot\frac{z^5}{5!}+76\cdot\frac{z^6}{6!}+232\cdot\frac{z^7}{7!}+764\cdot\frac{z^8}{8!}+2620\cdot\frac{z^9}{9!}+9496\cdot\frac{z^{10}}{10!}+\cdots$$

Un *desarreglo* es una permutación sin puntos fijos, vale decir, sin ciclos de largo 1. O sea:

$$\mathcal{D} = \textsc{MSet}(Cyc_{>1}(\mathcal{Z})$$

Revisando las derivaciones para las operaciones, esto corresponde a:

$$\begin{aligned}\widehat{D}(z) &= \exp\left(\sum_{k\geq 2}\frac{z^k}{k}\right)\\ &= \exp\left(\sum_{k\geq 1}\frac{z^k}{k} - z\right)\\ &= \exp\left(\ln\frac{1}{1-z} - z\right)\\ &= \frac{1}{1-z}\cdot e^{-z}\end{aligned}$$

Igual que antes.

A.3. Funciones generatrices bivariadas

Consideremos una clase $\mathscr{A}$, con objetos $\alpha \in \mathscr{A}$ de tamaño $|\alpha|$; y a su vez un parámetro, cuyo valor para α es $\chi(\alpha)$. Si los átomos que componen α son indistinguibles, es natural definir la función generatriz bivariada ordinaria:

$$A(z,u) = \sum_{\alpha\in\mathscr{A}} z^{|\alpha|}u^{\chi(\alpha)} \tag{A.1}$$

De la misma forma, si los átomos son distinguibles es apropiada la función generatriz exponencial:

$$\widehat{A}(z,u) = \sum_{\alpha\in\mathscr{A}} \frac{z^{|\alpha|}}{|\alpha|!}u^{\chi(\alpha)} \tag{A.2}$$

Es común que nos interese el valor promedio de $\chi(\alpha)$ para objetos de tamaño dado. Nótese que:

$$\frac{\partial A}{\partial u} = \sum_{\alpha\in\mathscr{A}} \chi(\alpha)u^{\chi(\alpha)-1}z^{|\alpha|} \tag{A.3}$$

Así podemos calcular los valores promedios a partir de los coeficientes de las siguientes sumas:

$$\sum_{\alpha\in\mathscr{A}} z^{|\alpha|} = A(z,1) \tag{A.4}$$

$$\sum_{\alpha\in\mathscr{A}} \chi(\alpha)z^{|\alpha|} = \left.\frac{\partial A}{\partial u}\right|_{u=1} \tag{A.5}$$

Vemos que (A.4) no es más que la función generatriz del número de objetos, mientras (A.5) es la función generatriz cumulativa vista en clase.

Un ejemplo es el algoritmo obvio para hallar el máximo de un arreglo, ver el listado A.1. Todas las operaciones se efectúan n veces, salvo las actualizaciones a la variable m. Es evidente que el número de veces que se actualiza m es $O(n)$, pero interesa una respuesta más precisa.

```
1  double maximum(const double a[], const int n)
2  {
3          int i;
4          double m;
5
6          m = a[0];
7          for(i = 1; i < n; i++)
```

```
8                   if(a[i] > m)
9                       m = a[i];
10          return m;
11  }
```

Listado A.1 – Hallar el máximo

Necesitamos un modelo para responder a la pregunta. Si suponemos que todos los valores son diferentes, y que todas las maneras de ordenarlos son igualmente probables, estamos buscando el número promedio de máximos de izquierda a derecha de permutaciones. Podemos describir la clase de permutaciones simbólicamente como:

$$\mathscr{P} = \mathscr{E} + \mathscr{P} \star \mathscr{Z} \tag{A.6}$$

Si llamamos $\chi(\pi)$ al número de máximos de izquierda a derecha en la permutación π, la función generatriz de probabilidad de que una permutación de tamaño n tenga k máximos de izquierda a derecha es:

$$M(z,u) = \sum_{\pi\in\mathscr{P}} \frac{z^{|\pi|}}{|\pi|!} u^{\chi(\pi)} \tag{A.7}$$

Esto casualmente es la función generatriz exponencial bivariada correspondiente a la clase (A.6).

Como el último elemento de la permutación es un máximo de izquierda a derecha si es el máximo de todos ellos (y los demás máximos de izquierda a derecha se mantienen al rerotular), usando la convención de Iverson podemos expresar el número de máximos de izquierda a derecha en la permutación resultante de $\pi \star (1)$ si se asigna el rótulo j al elemento nuevo como:

$$\chi(\pi) + [j = |\pi| + 1] \tag{A.8}$$

con lo que:

$$\begin{aligned} M(z,u) &= \frac{z^{|\epsilon|}}{|\epsilon|!} u^{\chi(\epsilon)} + \sum_{\pi\in\mathscr{P}} \sum_{1\le j\le|\pi|+1} \frac{z^{|\pi|+1}}{(|\pi|+1)!} u^{\chi(\pi)+[j=|\pi|+1]} && \text{(A.9)} \\ &= \frac{z^0}{0!} u^0 + \sum_{\pi\in\mathscr{P}} \frac{z^{|\pi|+1}}{(|\pi|+1)!} u^{\chi(\pi)} \sum_{1\le j\le|\pi|+1} u^{[j=|\pi|+1]} \\ &= 1 + \sum_{\pi\in\mathscr{P}} \frac{z^{|\pi|+1}}{(|\pi|+1)!} u^{\chi(\pi)} (|\pi| + u) && \text{(A.10)} \end{aligned}$$

Derivando respecto de z (indicamos derivadas por subíndices para simplificar notación):

$$\begin{aligned} M_z(z,u) &= \sum_{\pi\in\mathscr{P}} \frac{z^{|\pi|}}{|\pi|!} u^{\chi(\pi)} (|\pi| + u) \\ &= zM_z(z,u) + uM(z,u) \end{aligned}$$

Vale decir:

$$(1-z)M_z(z,u) - uM(z,u) = 0 \tag{A.11}$$

En (A.11) la variable u interviene como parámetro, esta es una ecuación diferencial ordinaria. Como $M(0,u) = 1$, la solución es:

$$M(z,u) = \left(\frac{1}{1-z}\right)^u$$

Las derivadas de interés son:

$$M_u(z,u) = \frac{1}{(1-z)^u} \ln \frac{1}{1-z}$$
$$M_{uu}(z,u) = \frac{1}{(1-z)^u} \ln^2 \frac{1}{1-z}$$
$$[z^n] M_u(z,1) = \sum_{1 \le k \le n} H_k$$

Con M_{uu} podemos también obtener la varianza.

Bibliografía

[1] Horst H. von Brand: *Fundamentos de Informática*. git://csrg.inf.utfsm.cl/vonbrand/Ramos/trainwreck, Agosto 2016. Versión 0.83.

[2] Jérémie Lumbroso and Basile Morcrette: *A gentle introduction to analytic combinatorics*. http://www.sfu.ca/~jlumbros/Events/Oxford12/stuff/draft-v7.pdf, September 2012.

Apéndice B

Una pizca de probabilidades

Requeriremos una pizca de probabilidades discretas en lo que sigue, ofrecemos un rápido repaso de lo requerido con sus derivaciones. La discusión general se adapta de Ash [1].

B.1. Definiciones básicas

El origen de la teoría de probabilidades viene de juegos de azar. Por ejemplo, se vio que al lanzar una moneda muchas veces, muy aproximadamente la mitad de las veces sale cara. De la misma forma, si se toma un mazo de cartas inglés (sin comodines), se baraja y se extrae una carta, y se repite el ejercicio muchas veces, un cuarto de las veces la carta es trébol, y una en trece es un as.

En el experimento con cartas tenemos 52 posibles resultados, y el "principio de razón insuficiente" o "mínima sorpresa" nos dice que esperamos cualquiera de los resultados, sin preferencias. Los pioneros del área definieron probabilidad como el número de casos favorables dividido por el número total de casos, con la justificación de que eran todos igualmente probables. En el caso de la pinta de la carta, $13/52$ o $1/4$. Esta definición es restrictiva (supone un número finito de posibilidades), pero, mucho más grave, es circular: estamos definiendo "probabilidad" en términos de "igualmente probable". Necesitamos una base más sólida.

B.1.1. Formalizando probabilidades

El primer ingrediente en una teoría matemática de las probabilidades es el *espacio muestral*, que anotaremos Ω, el conjunto de posibles resultados de un experimento al azar. El requisito esencial es que una ejecución del experimento entrega exactamente un resultado de Ω. El segundo ingrediente es el *evento*, una pregunta sobre el resultado de un experimento que tiene respuesta "si" o "no" (por ejemplo, si la carta elegida es una figura). Un evento no es más que un subconjunto del espacio muestral. Por convención, los eventos se anotan con letras romanas mayúsculas del comienzo del alfabeto A, B, y así sucesivamente. Los resultados en que se cumple el evento A se llaman *favorables* (para A). El evento Ω se dice *seguro* (siempre ocurre), el evento $\varnothing$ se dice *imposible* (jamás ocurre). Siendo conjuntos los eventos, se aplican las operaciones conocidas de conjuntos.

Buscamos asignar probabilidades a eventos. Aparecen problemas técnicos, no siempre es posible asignar probabilidades en forma consistente a subconjuntos de Ω. Requeriremos que la clase de

eventos $\mathscr{F}$ forme lo que se conoce como un *campo sigma*, o sea, cumple los siguientes tres requisitos:

$$\Omega \in \mathscr{F} \tag{B.1}$$

$$A_1, A_2, \ldots \in \mathscr{F} \text{ implica } \bigcup_n A_n \in \mathscr{F} \tag{B.2}$$

$$A \in \mathscr{F} \text{ implica } \overline{A} \in \mathscr{F} \tag{B.3}$$

O sea, es cerrado respecto de unión finita o infinita numerable (B.2) y complemento (B.3). Por (B.1) y (B.3) concluimos que $\varnothing \in \mathscr{F}$. Por de Morgan, de (B.2) y (B.3) también es cerrado respecto de intersección finita o infinita numerable. O sea, si la pregunta "¿Ocurrió A_i?" tiene respuesta definitiva para $i = 1, 2, \ldots$, tiene respuesta definitiva "¿Ocurrió alguno de los A_i?" al igual que "¿Ocurrieron todos los A_i?"

En muchos casos podremos tomar $\mathscr{F}$ como el conjunto de todos los subconjuntos de Ω, situaciones en las cuales se requieren las sutilezas indicadas se dan cuando Ω son conjuntos no numerables, como $\mathbb{R}$.

Estamos en condiciones de definir probabilidades de eventos. Ponemos los siguientes requisitos:

$$0 \leq \Pr[A] \leq 1 \tag{B.4}$$

$$\Pr[\Omega] = 1 \tag{B.5}$$

$$\Pr[A \cup B] = \Pr[A] + \Pr[B] \qquad \text{si } A \cap B = \varnothing \tag{B.6}$$

$$\Pr\left[\bigcup_i A_i\right] = \sum_i \Pr[A_i] \qquad \text{para colecciones contables de } A_i \text{ disjuntos} \tag{B.7}$$

$$\tag{B.8}$$

Definición B.1. Un *espacio de probabilidades* es un trío $(\Omega, \mathscr{F}, \Pr)$, con Ω es un conjunto, $\mathscr{F}$ es un campo sigma de subconjuntos de Ω, y Pr es una medida de probabilidad en $\mathscr{F}$.

Algunas consecuencias simples son:

1. $\Pr[\varnothing] = 1$

   Como $A \cup \varnothing = A$ y $A \cap \varnothing = \varnothing$, tenemos $\Pr[A] = \Pr[A \cup \varnothing] = \Pr[A] + \Pr[\varnothing]$, y concluimos $\Pr[\varnothing] = 0$.

2. $\Pr[A \cup B] = \Pr[A] + \Pr[B] - \Pr[A \cap B]$

   Este es simplemente un caso especial del principio de inclusión y exclusión. En particular, como $\Pr[A \cap B] \geq 0$, tenemos la *cota de unión* $\Pr[A \cup B] \leq \Pr[A] + \Pr[B]$

3. Si $A \subseteq B$, entonces $\Pr[A] \leq \Pr[B]$, específicamente $\Pr[A \setminus B] = \Pr[A] - \Pr[B]$.

4. Para una colección numerable de eventos A_i se cumple $\Pr[\bigcup_i A_i] \leq \sum_i \Pr[A_i]$

Las demostraciones de estas aseveraciones quedan de ejercicios.

B.1.2. Variables aleatorias

Una *variable aleatoria* es el resultado de un experimento. Generalmente nos interesará el caso de variables aleatorias numéricas. Usaremos letras mayúsculas hacia el final del alfabeto $(X, Y, \ldots)$ para representar variables, y las correspondientes letras minúsculas para sus valores. Los eventos en tales casos pueden describirse como por ejemplo $X = a$ o $a \leq X \leq b$, para valores a, b dados. Para ellas podemos definir el *valor esperado* de la variable X como:

$$\mathbb{E}[X] = \sum_x x \Pr[X = x] \tag{B.9}$$

y una medida importante de la dispersión es la *varianza*:

$$\text{var}[X] = \mathbb{E}[(X - \mathbb{E}[X])^2] \tag{B.10}$$

Comúnmente se usan las notaciones $\mu = \mathbb{E}[X]$ y $\sigma^2 = \text{var}[X]$. En el caso que las variables no tomen valores discretos (como acá), en vez de sumas aparecen integrales.

Un resultado extremadamente importante es:

Teorema B.1 (Linearidad del valor esperado)**.** *Sean X_1, X_2 variables aleatorias, α y β constantes arbitrarias. Entonces:*

$$\mathbb{E}[\alpha X + \beta Y] = \alpha \mathbb{E}[X] + \beta \mathbb{E}[Y]$$

Demostración. En el caso de variables discretas tenemos:

$$\begin{aligned}\mathbb{E}[\alpha X_1 + \beta X_2] &= \sum_x (\alpha x \Pr[X_1 = x] + \beta x \Pr[X_2 = x]) \\ &= \alpha \sum_x x \Pr[X_1 = x] + \beta \sum_x x \Pr[X_2 = x] \\ &= \alpha \mathbb{E}[X_1] + \beta E[X_2]\end{aligned}$$

Las sumas se extienden sobre posibles valores de X_1 y X_2. Esencialmente la misma demostración vale para variables continuas, con integrales en vez de sumas. □

Nótese que no hemos supuesto nada sobre las variables involucradas. Por inducción, esto se extiende a un número finito de variables.

B.1.3. Independencia

Decimos que dos variables X e Y son *independientes* si para todo par de valores x e y:

$$\Pr[(X = x) \wedge (Y = y)] = \Pr[X = x] \cdot \Pr[Y = y] \tag{B.11}$$

Una colección de variables es *mutuamente independiente* si para todo subconjunto $S \subseteq N$ y toda colección de valores x_i:

$$\Pr[(X_{i_1} = x_{i_1}) \wedge (X_{i_2} = x_{i_2}) \wedge \cdots \wedge (X_{i_s} = x_{i_s})] = \Pr[X_{i_1} = x_{i_1}] \cdot \Pr[X_{i_2} = x_{i_2}] \cdots \Pr[X_{i_s} = x_{i_s}] \tag{B.12}$$

Un teorema importante sobre variables independientes es:

Teorema B.2. *Si X_1, X_2 son independientes:*

$$\begin{aligned}\mathbb{E}[X_1 X_2] &= \mathbb{E}[X_1] \cdot E[X_2] \\ \mathbb{E}[f(X_1) f(X_2)] &= \mathbb{E}[f(X_1)] \cdot E[f(X_2)]\end{aligned}$$

Demostración. Si las variables X_1 y X_2 son independientes, entonces:

$$\begin{aligned}\mathbb{E}[X_1 X_2] &= \sum_{x_1, x_2} x_1 x_2 \Pr[X_1 = x_1 \wedge X_2 = x_2] \\ &= \sum_{x_1, x_2} x_1 x_2 \Pr[X_1 = x_1] \Pr[X_2 = x_2] \\ &= \sum_{x_1} x_1 \Pr[X_1 = x_1] \cdot \sum_{x_2} x_2 \Pr[X_2 = x_2]\end{aligned}$$

La misma demostración, con integrales en vez de sumas, vale para variables continuas. □

B.2. Relaciones elementales

Suele ser útil considerar si las funciones entre manos son *convexas* (o *cóncavas*).

Definición B.2. La función f se dice *convexa* si para $0 \leq \alpha \leq 1$:

$$\alpha f(x) + (1-\alpha) f(y) \geq f(\alpha x + (1-\alpha) y)$$

Vale decir, la gráfica de f está por debajo de una cuerda que une dos puntos, ver la figura B.1. Por

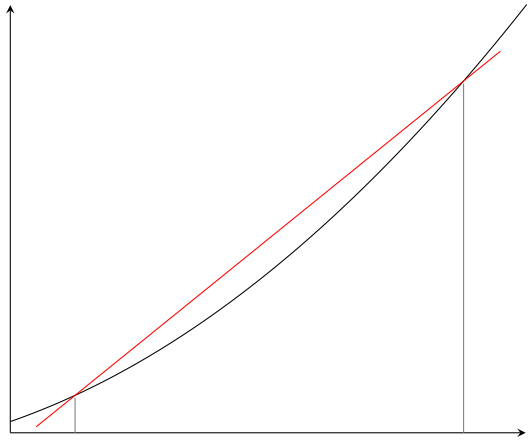


Figura B.1 – Una función convexa

inducción, si $\alpha_i \geq 0$ con $\sum_i \alpha_i = 1$, es:

$$\sum_i \alpha_i f(x_i) \geq f\left(\sum_i \alpha_i x_i\right)$$

Aplicando esto a una variable aleatoria finita, con los α_i las probabilidades de los valores de X, obtenemos:

Teorema B.3 (Desigualdad de Jensen)**.** *Si la función f es convexa:*

$$f(\mathbb{E}[X]) \leq \mathbb{E}[f(X)] \tag{B.13}$$

B.3. Desigualdad de Markov

Sea X una variable aleatoria discreta, no negativa, y sea $c > 0$ una constante. Interesa derivar la probabilidad de que X sea mayor a c. O sea, interesa $\Pr[X \geq c]$. Como X es discreta, podemos

escribir:

$$\begin{aligned}
\mu &= \mathbb{E}[X] \\
&= \sum_x x \Pr[X = x] \\
&= \sum_{0<x<c} x \Pr[X = x] + \sum_{x \geq c} x \Pr[X = x] \\
&\geq \sum_{x \geq c} x \Pr[X = x] \\
&\geq \sum_{x \geq c} c \Pr[X = x] \\
&\geq c \sum_{x \geq c} \Pr[X = x] \\
&= c \Pr[X \geq c]
\end{aligned}$$

de donde tenemos:

Teorema B.4 (Desigualdad de Markov)**.**

$$\Pr[X \geq c] \leq \frac{\mu}{c} \tag{B.14}$$

Es claro que exactamente lo mismo puede hacerse si X es una variable continua.

Esta desigualdad es útil por sí misma, pero aún más porque es instrumental en la derivación de desigualdades más ajustadas.

B.4. Desigualdad de Chebyshev

Para una variable aleatoria general X nos interesa acotar la probabilidad $\Pr[|X - \mu| > a]$, donde $\mu = \mathbb{E}[X]$. Notamos que este es el mismo evento $(X - \mu)^2 > a^2$, como $(X - \mu)^2$ es una variable no negativa, podemos aplicarle la desigualdad de Markov. Recordando que $\mathbb{E}[(X - \mu)^2] = \text{var}[X] = \sigma^2$:

$$\begin{aligned}
\Pr[|X - \mu| > a] = \Pr[(X - \mu)^2 > a^2] \\
&\leq \frac{\mathbb{E}[(X - \mu)^2]}{a^2} \\
&= \frac{\sigma^2}{a^2}
\end{aligned}$$

En particular, substituyendo $a = c\sigma$:

Teorema B.5 (Desigualdad de Chebyshev)**.**

$$\Pr[|X - \mu| \geq c\sigma] \leq \frac{1}{c^2} \tag{B.15}$$

B.5. Cotas de Chernoff

Para obtener la cota de Chebyshev elevamos al cuadrado, ahora exponenciamos. Esto da toda una familia de desigualdades, dependiendo de la distribución supuesta y el detalle de las cotas empleadas para simplificar el resultado. La importancia radica en que da cotas ajustadas usando información mínima sobre las variables. La idea básica es de Chernoff [2]. Desarrollaremos una versión general en detalle, basándonos en Lehman, Leighton y Meyer [3, sección 20.6.2]. Si se revisan los detalles de la demostración, es claro que el mismo desarrollo se aplica si alguna de las variables es continua.

Teorema B.6 (Cota superior de Chernoff)**.** *Sean $X_1, \ldots, X_n$ variables aleatorias discretas mutuamente independientes con $0 \le X_i \le 1$ para todo i. Sea $X = X_1 + \cdots + X_n$ y sea $\mu = \mathbb{E}[X]$. Entonces para todo $c \ge 1$:*

$$\Pr[X \ge c\mu] \le \mathrm{e}^{-\beta(c)\mu} \tag{B.16}$$

donde $\beta(c) = c \ln c - c + 1$.

En aras de la claridad, partiremos con la demostración central, lo que mostrará la necesidad de acotar feos productos, cotas que demostraremos inmediatamente a continuación como lemas.

Dejamos anotado para uso futuro que la función $\beta(c)$ es convexa para $c > 0$, con un mínimo en $c = 1$ donde $\beta(1) = 0$.

Demostración. El punto clave es exponenciar ambos lados de la desigualdad $X \ge c\mu$ y aplicar la desigualdad de Markov:

$$\begin{aligned}
\Pr[X \ge c\mu] &= \Pr[c^X \ge c^{c\mu}] \\
&\le \frac{\mathbb{E}[c^X]}{c^{c\mu}} && \text{(cota de Markov)} \\
&\le \frac{\mathrm{e}^{(c-1)\mu}}{\mathrm{e}^{\mu c \ln c}} && \text{(ver lema B.7)} \\
&= \mathrm{e}^{-\beta(c)\mu}
\end{aligned}$$

□

Hemos usado el siguiente resultado, expresado con las mismas variables anteriores:

Lema B.7.

$$\mathbb{E}[c^X] \le \mathrm{e}^{(c-1)\mu}$$

Demostración.

$$\begin{aligned}
\mathbb{E}\left[c^X\right] &= \mathbb{E}\left[c^{X_1+\cdots+X_n}\right] && \text{(definición de } X\text{)} \\
&= \mathbb{E}\left[c^{X_1} \cdots c^{X_n}\right] \\
&= \mathbb{E}\left[c^{X_1}\right] \cdots \mathbb{E}\left[c^{X_n}\right] && \text{(producto de valores independientes)} \\
&\le \mathrm{e}^{(c-1)\mathbb{E}[X_1]} \cdots \mathrm{e}^{(c-1)\mathbb{E}[X_n]} && \text{(ver lema B.8 más adelante)} \\
&= \mathrm{e}^{(c-1)(\mathbb{E}[X_1]+\cdots+\mathbb{E}[X_n])} \\
&= \mathrm{e}^{(c-1)(\mathbb{E}[X_1+\cdots+X_n])} && \text{(linearidad del valor esperado)} \\
&= \mathrm{e}^{(c-1)\mathbb{E}[X]}
\end{aligned}$$

□

Finalmente:

Lema B.8.

$$\mathbb{E}\left[c^{X_i}\right] \le \mathrm{e}^{(c-1)\mathbb{E}[X_i]}$$

Demostración. En lo que sigue, las sumas son sobre valores x tomados por la variable X_i. Por la definición de X_i, $x \in [0,1]$.

$$\begin{aligned}
\mathbb{E}\left[c^{X_i}\right] &= \sum_x c^x \Pr[X_i = x] && \text{(definición de valor esperado)}\\
&\le \sum_x (1 + (c-1)x) \Pr[X_i = x] && \text{(convexidad – ver abajo)}\\
&= \sum_x (\Pr[X_i = x] + (1-c)x \Pr[X_i = x])\\
&= 1 + (c-1)\,\mathbb{E}[X_i]\\
&\le \mathrm{e}^{(c-1)\,\mathbb{E}[X_i]} && \text{(porque } 1 + z \le \mathrm{e}^z\text{)}
\end{aligned}$$

El segundo paso usa la desigualdad:

$$c^x \le 1 + (c-1)x$$

que vale para $c \ge 1$ y $1 \le x \le 1$ ya que la función convexa c^x está por debajo de la cuerda entre los puntos $x = 0$ y $x = 1$. Esta es la razón de restringir las variables X_i a $[0,1]$, y el descomponer la demostración del lema B.7. □

Ocasionalmente interesa una cota superior, provista por el siguiente teorema.

Teorema B.9. *Con las mismas suposiciones del teorema B.6:*

$$\Pr[X < \mu/c] \le \mathrm{e}^{-\beta(1/c)\mu} \tag{B.17}$$

Demostración. La demostración es esencialmente igual a la del teorema B.6.

$$\begin{aligned}
\Pr[X < \mu/c] &= \Pr\left[c^{-X} > c^{-\mu/c}\right]\\
&\le \frac{\mathbb{E}[c^{-X}]}{c^{-\mu/c}} && \text{(cota de Markov)}\\
&\le \frac{\mathrm{e}^{-(1-1/c)\mu}}{\mathrm{e}^{-\mu \ln c/c}} && \text{(ver lema B.10)}
\end{aligned}$$

El resultado resulta reorganizando esto. □

Tenemos las variantes de los lemas B.7 y B.8:

Lema B.10.

$$\mathbb{E}[c^{-X}] \le \mathrm{e}^{-(1-1/c)\mu}$$

Demostración.

$$\begin{aligned}
\mathbb{E}\left[c^{-X}\right] &= \mathbb{E}\left[c^{-X_1-\cdots-X_n}\right] && \text{(definición de } X\text{)}\\
&= \mathbb{E}\left[c^{-X_1}\cdots c^{-X_n}\right]\\
&= \mathbb{E}\left[c^{-X_1}\right]\cdots\mathbb{E}\left[c^{-X_n}\right] && \text{(producto de valores independientes)}\\
&\le \mathrm{e}^{-(1-1/c)\,\mathbb{E}[X_1]}\cdots\mathrm{e}^{-(1-1/c)\,\mathbb{E}[X_n]} && \text{(ver lema B.11 más adelante)}\\
&= \mathrm{e}^{-(1-1/c)(\mathbb{E}[X_1]+\cdots+\mathbb{E}[X_n])}\\
&= \mathrm{e}^{-(1-1/c)(\mathbb{E}[X_1+\cdots+X_n])} && \text{(linearidad del valor esperado)}\\
&= \mathrm{e}^{-(1-1/c)\,\mathbb{E}[X]}
\end{aligned}$$

□

Lema B.11.

$$\mathbb{E}\left[c^{-X_i}\right] \leq \mathrm{e}^{-(1-1/c)\mathbb{E}[X_i]}$$

Demostración. En lo que sigue, las sumas son sobre valores x tomados por la variable X_i. Por la definición de X_i, $x \in [0,1]$.

$$\begin{aligned}
\mathbb{E}\left[c^{-X_i}\right] &= \sum c^{-x} \Pr[X_i = x] && \text{(definición de valor esperado)} \\
&\leq \sum (1-(1-1/c)x) \Pr[X_i = x] && \text{(convexidad – ver abajo)} \\
&= \sum (\Pr[X_i = x] - (1-1/c)x \Pr[X_i = x]) \\
&= 1-(1-1/c)\mathbb{E}[X_i] \\
&\leq \mathrm{e}^{-(1-1/c)\mathbb{E}[X_i]} && \text{(porque } 1+z \leq \mathrm{e}^z\text{)}
\end{aligned}$$

El segundo paso usa la desigualdad:

$$c^{-x} \leq 1-(1-1/c)x$$

que vale para $c \geq 1$ y $1 \leq x \leq 1$ ya que la función convexa c^{-x} está por debajo de la cuerda entre los puntos $x = 0$ y $x = 1$. Esta es la razón de restringir las variables X_i a $[0,1]$, y el descomponer la demostración del lema B.10. □

Ejercicios

1. Complete las demostraciones de las propiedades de probabilidades de eventos de la sección B.1.1.

Bibliografía

[1] Robert B. Ash: *Basic Probability Theory*. Dover Publications, Inc., 2008.

[2] Herman Chernoff: *A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations*. Annals of Mathematical Statistics, 23(4):493–507, December 1952.

[3] Eric Lehman, F. Thomson Leighton, and Albert R. Meyer: *Mathematics for Computer Science*. http://courses.csail.mit.edu/6.042/spring16/mcs.pdf, September 2016.