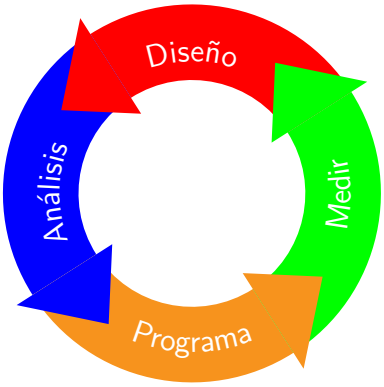
Algoritmos y Complejidad

Algoritmos y Complejidad
INF-221

Horst H. von Brand



4 de septiembre de 2018

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 0.4 (2018-2) del 2018-09-04

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

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. Se iniciaron con el 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. El resultado fue creciendo con extensiones posteriores y extensas reorganizaciones. Especial reconocimiento a Claudio Torres, quien leyó y comentó el apunte, especialmente la primera parte.

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 estas. Una de las mayores revoluciones se inició en el siglo XVII al desarrollarse el cálculo, que con la física llevó a modelos precisos de muchos fenómenos de interés, que luego se extendieron a otras ciencias. Estos modelos rara vez se pueden resolver en forma exacta, hay que recurrir a técnicas aproximadas. Incluso se da que es más eficiente calcular una solución aproximada que usar una engorrosa fórmula exacta. El mismo Newton inventó métodos numéricos, desarrollos de los cuales hoy llevan su nombre.

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 [3], otro buen compendio es el texto de Sauer [11], y hay numerosas colecciones de notas de clase disponibles, como la de Olver [8], de Cowley [2] o de Philip [9, 10]. Es recomendable el texto de Heinhold [7], que se centra en una visión intuitiva y desde el usuario de distintos métodos. El texto de Acton [1] es bastante antiguo (una reseña indica que muestra claramente su origen en épocas de cálculo manual y computadores rudimentarios), pero da una visión desde las trincheras en lenguaje coloquial.

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 [4] y Haberman [5, 6]. El punto (III) es 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. En nuestro caso, es $k = 2$ (porque $0{,}023 < 10^{-1}$), con lo que 1 000 es una aproximación de 2 cifras significativas a 1 024.

La misma idea puede aplicarse si estamos hablando de vectores, solo que en tal caso usaremos normas, $\|\mathbf{x}^* - \mathbf{x}\|$ es el error absoluto y $\|\mathbf{x}^* - \mathbf{x}\|/\|\mathbf{x}\|$ el relativo.

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. Para simplificar, supondremos que la función f es conocida, por lo discutido antes es común que solo se conozca en forma aproximada.

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 intermedios 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:

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

La solución es un simple ejercicio:

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

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

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

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

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

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

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

Wilkinson [12, 13] 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 [14].

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:

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

El menor cero es cercano, los demás se alejan crecientemente y los últimos seis mutan a complejos conjugados. Cerca de un cero de p, $q(x) \approx x^5$, que para x grande es muy grande.

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:

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

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:

$$I_n = \int_0^1 x^n \mathrm{e}^{x-1}\,\mathrm{d}x \tag{1.1}$$

Por integración por partes obtenemos la recurrencia:

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

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.

Note 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 función 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 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 de cálculo de la integral (1.1) 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, para distintos valores de a y b.

Bibliografía

[1] Forman S. Acton: *Numerical Methods that (Usually) Work*. The Mathematical Association of America, 1990. Updated abnd revised from the 1970 edition.

[2] Stephen J. Cowley: *Mathematical Tripos: IB Numerical Analysis*. http://www.damtp.cam.ac.uk/user/sjc1/teaching/NAIB/notes.pdf, Lent 2014. Department of Applied Mathematics and Theoretical Physics, University of Cambridge.

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

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

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

[6] Josh Haberman: *Floating point demystified, part 2*. http://blog.reverberate.org/2016/02/06/floating-point-demystified-part2.html, February 2016.

[7] Brian Heinhold: *An intuitive guide to numerical methods*. https://www.brianheinold.net/notes/An_Intuitive_Guide_to_Numerical_Methods_Heinold.pdf, 2013. Department of Mathematics and Computer Science, Mount St. Mary's University.

[8] Peter J. Olver: *Lecture notes on numerical analysis*. http://math.umn.edu/~olver/num.html, May 2008. School of Mathematics, University of Minnesota.

[9] Peter Philip: *Numerical analysis I*. http://www.mathematik.uni-muenchen.de/~philip/publications/lectureNotes/numericalAnalysis_new.pdf, July 2017. Mathematisches Institut, Universität München.

[10] Peter Philip: *Numerical analysis II*. http://www.mathematik.uni-muenchen.de/~philip/publications/lectureNotes/numericalAnalysis2.pdf, August 2017. Mathematisches Institut, Ludwig-Maximilians-Universität München.

[11] Timothy Sauer: *Numerical Analysis*. Pearson, second edition, 2011.

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

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

[14] 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

El problema de hallar ceros de funciones es muy común. Lamentablemente solo en casos muy especiales hay fórmulas exactas, la mayor parte de las veces debe recurrirse a hallar buenas aproximaciones numéricas.

El problema es 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 $f(x^*) = 0$. En la figura 2.1 se aprecia un cero cerca de 0,6.

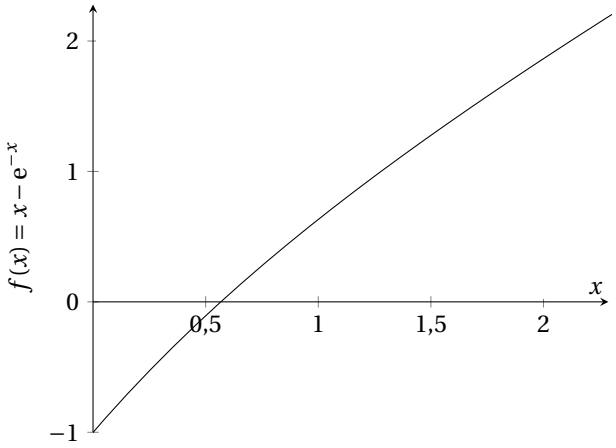


Figura 2.1 – Gráfica de la función $x - e^{-x}$

Otra forma de solucionar el problema puede ser graficar x, e^{-x}, para buscar las intersecciones (figura 2.2).

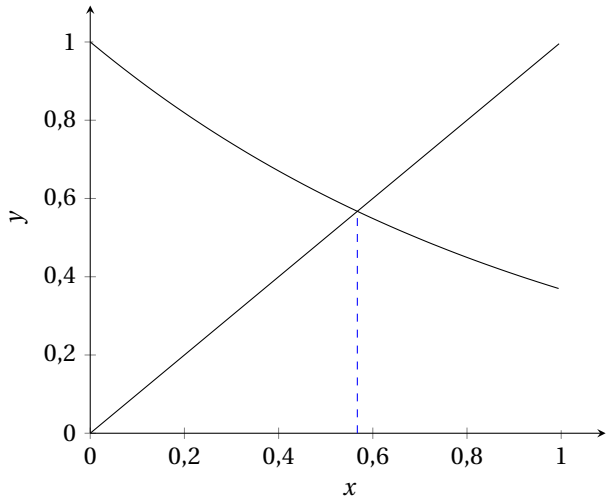


Figura 2.2 – Gráficas de x y e^{-x}

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 la

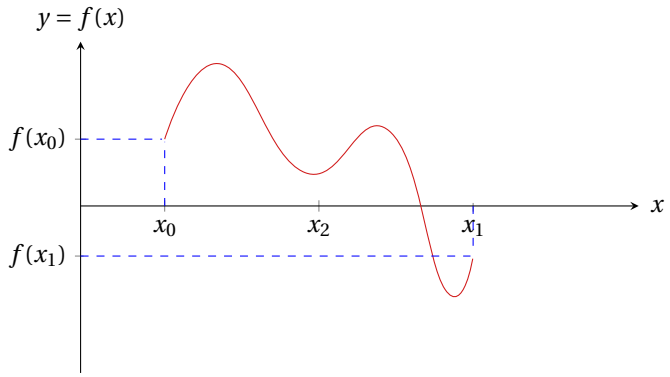


Figura 2.3 – Si $f(x_0) \cdot f(x_1) < 0$, hay $x^* \in [x_0, x_1]$ donde $f(x^*) = 0$.

figura 2.3. Podemos apreciar que el intervalo $[x_2, x_1]$ cumple con $f(x_2) \cdot f(x_1) < 0$. En consecuencia,

por el teorema del valor intermedio 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.

Note 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

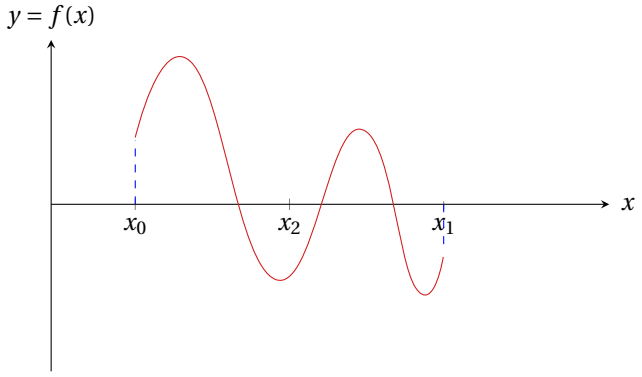


Figura 2.4 – Una función con 3 ceros.

no abarca todos los casos posibles.

2.1.2. Método *Regula Falsi*

Siempre podemos usar la idea de tomar dos puntos que horquillen un cero en la curva e interpolar para obtener una aproximación mejor, vea la figura 2.5. Esto da:

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

De allí procedemos iterando, calculando:

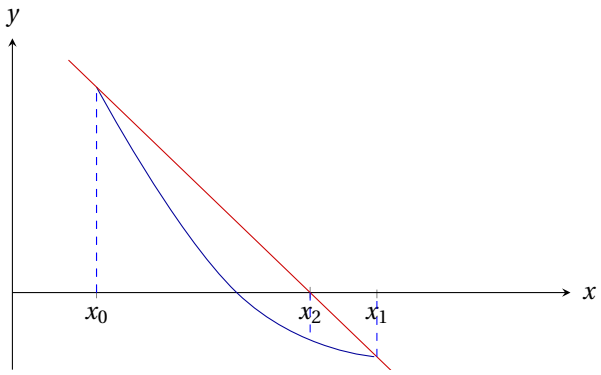


Figura 2.5 – *Regula falsi*

$$x_{n+1} = x_n - f(x_n) \cdot \frac{x_n - x_0}{f(x_n) - f(x_0)} \tag{2.4}$$

Mantenemos fijo el punto inicial x_0.

2.2. Iteración de Punto Fijo

En inglés, a esta técnica se le llama *fixed point iteration* y se abrevia 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:

$$f(x) = 0$$

en la forma:

$$g(x) = x$$

Note que siempre podemos escribir:

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

Queda elegir un $\alpha \neq 0$ 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:

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

Usando iteraciones de punto fijo, se tiene que el cero de la función corresponde a la intersección entre $y = x$ y $g(x)$ (figura 2.6). Para efectos de convergencia, se puede ver como una espiral (figura 2.7

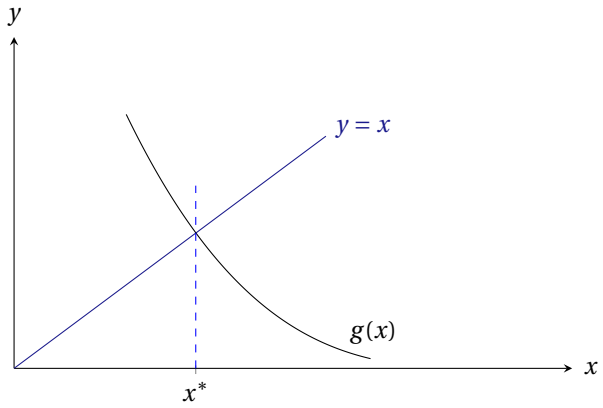


Figura 2.6 – La intersección entre $y = x$ e $y = g(x)$ es un punto fijo

y figura 2.8). A los diagramas de las figuras 2.7 y 2.8 les llaman *diagrama de telaraña* (en inglés, *cobweb diagram*) por razones fáciles de ver. 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 $g(x)$. Continúe así hasta encontrar una aproximación suficiente al punto fijo.

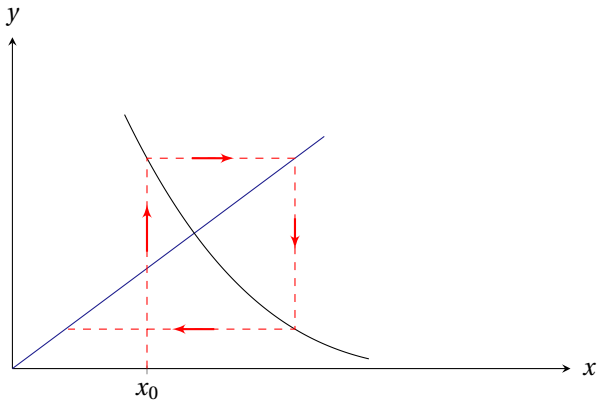


Figura 2.7 – En este caso, la espiral diverge (mire las flechas).

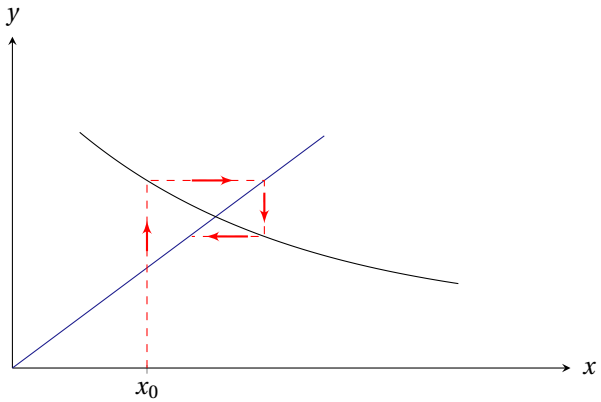


Figura 2.8 – En este caso, la espiral converge (mire las flechas).

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 2.9. Pero a diferencia de ese método, no se mantienen puntos fijos. Partiendo con x_0, x_1 se calculan valores sucesivos usando:

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

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:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{2.5}$$

resultando un valor x_{n+1} que se encuentra más cerca de x^* (figura 2.10). Finalmente, iteramos el proceso hasta obtener un valor suficientemente cercano al buscado.

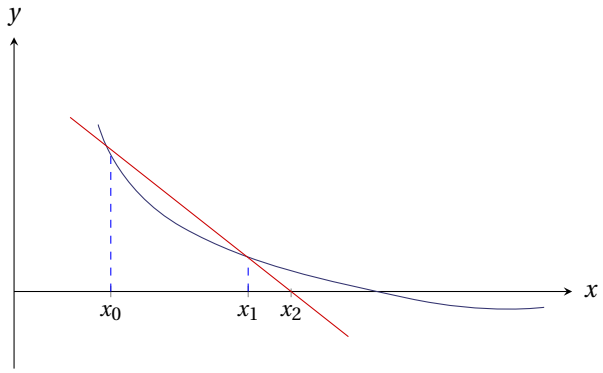


Figura 2.9 – Una iteración del método de la secante.

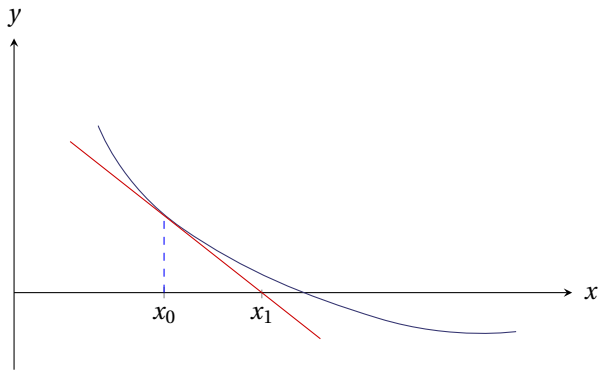


Figura 2.10 – El valor x_1 de (2.5) es más cercano a x^* que x_0.

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:

$$|e_{n+1}| \leq C|e_n|^p \tag{2.6}$$

Donde e_n corresponde al error con n iteraciones.

- 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.

Note que mientras mayor sea el orden de convergencia p, más rápido (en menos iteraciones) encontraremos una aproximación adecuada al valor buscado. Pero esto hay que balancearlo con otras consideraciones, como si conocemos derivadas (puede ser que la función quede definida por un procedimiento complejo, que no permita obtenerlas) y el trabajo extra por iteración que demande.

Clase 3

Análisis de Convergencia

Contamos con varios métodos iterativos para hallar un cero de una función, interesa compararlos según alguna medida de rendimiento. Una medida clara es cuánto disminuye el error en cada iteración, cosa que es relativamente sencilla de deducir. Claro que en la práctica más nos interesa cuánto tiempo demora, además de otras medidas como qué tan complicado es de programar y posible información adicional que requiere (por ejemplo, un método que necesita derivadas es impracticable cuando la función se conoce solo como el resultado de un complejo cálculo, no en forma explícita). Otras medidas importantes son el área desde la cual el método converge (si tenemos que partir muy cerca del cero de interés, no tiene demasiada gracia). Pero esto nos llevaría demasiado lejos, ver las notas de Kahan [4] para una discusión más detallada.

3.1. Notación asintótica

Necesitaremos estimaciones asintóticas de diversas cantidades, vale decir, obtener estimaciones de ciertas cantidades mediante «funciones simples» que muestran cómo estas cantidades se comportan «asintóticamente» (vale decir, cuando algún argumento tiende a infinito). Esta es un área extremadamente amplia y variada, resumida brillantemente por de Bruijn [1]. Acá nos basamos en el corto resumen de notación y propiedades de Hildebrand [2].

Primero, definimos:

$$f(s) = O(g(s)) \quad (s \to s_0) \tag{3.1}$$

si hay una constante $\delta > 0$ tal que hay $c > 0$ para la cual $|f(s)| \leq c|g(s)|$ siempre que $|s - s_0| \leq \delta$. La definición afín más común es:

$$f(s) = O(g(s)) \quad (s \to \infty) \tag{3.2}$$

si para todo $N_0 > 0$ hay un valor $c > 0$ para el cual $|f(s)| \leq c|g(s)|$ siempre que $s \geq N_0$. Comúnmente se usa en expresiones aritméticas, como:

$$\ln(1 + x) = x + O(x^2) \quad (x \to 0)$$

para expresar en forma más simple lo que debiéramos escribir:

$$\ln(1 + x) - x = O(x^2) \quad (x \to 0)$$

Hay que tener cuidado, la «igualdad» en expresiones con $O(\cdot)$ no cumple las reglas acostumbradas, debe considerarse que el lado derecho es una versión menos precisa del lado izquierdo. Así tiene sentido decir:

$$O(\sqrt{x}) = O(x)$$

porque cualquier función que cumple $f(x) \leq c\sqrt{x}$ para algún c también cumple $f(x) \leq c'x$. Pero:

$$O(x) = O(\sqrt{x})$$

claramente es absurdo. Por la misma razón, no pueden cancelarse términos, en realidad debemos considerar:

$$O(f(x)) \pm O(f(x)) = O(f(x))$$

ya que las distintas instancias de $O(f(x))$ representan funciones diferentes. Por ejemplo, si $f(x) = \log x + O(1/x)$ y $g(x) = \log x + O(1/x)$ solo podemos concluir $f(x) = g(x) + O(1/x)$.

Tenemos las siguientes reglas básicas, para simplificar omitiremos el rango.

Constantes: Si c es una constante positiva, $f(x) = O(cg(x))$ y $f(x) = O(g(x))$ son equivalentes. En particular, $f(x) = O(c$ es lo mismo que $f(x) = O(1)$.

Transitividad: Si $f(x) = O(g(x))$ y $g(x) = O(h(x))$, es $f(x) = O(h(x))$.

Multiplicación: Si $f_i(x) = O(g_i(x))$ para $i = 1,2$ entonces $f_1(x)f_2(x) = O(g_1(x)g_2(x))$.

Extraer factores: Si $f(x) = O(g(x)h(x))$, entonces $f(x) = g(x)O(h(x))$. Por ejemplo, si $f(x) = x + O(x/\log x)$ entonces $f(x) = x(1 + O(1/\log x))$.

Sumas: Si $f_i(x) = O(g_i(x))$ para $i = 1,2,\ldots,n$, siempre que las constantes implícitas en los $O(\cdot)$ no dependan de i podemos escribir:

$$\sum_{1\leq i\leq n} f_i(x) = O\left(\sum_{1\leq i\leq n} |g_i(x)|\right)$$

Esto también es válido para sumas infinitas.

La demostración de estas reglas es aplicación rutinaria de las definiciones.

Algunas reglas de transformación adicionales válidas si $\phi(x) \to 0$ son:

$$\frac{1}{1+O(\phi(x))} = 1 + O(\phi(x))$$
$$(1+O(\phi(x)))^p = 1 + O(\phi(x))$$
$$\ln(1+O(\phi(x))) = O(\phi(x))$$
$$\exp(O(\phi(x))) = 1 + O(\phi(x))$$

Estas resultan de las respectivas series de Taylor.

3.2. Análisis de las técnicas descritas

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 usando la forma de Lagrange del residuo, sabemos que hay $x^* < \xi < x$ (o $x < \xi x^*$ si $x < x^*$) tal que:

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

Definiendo $e = x - x^*$:

$$= f'(x^*)e(1+Me) + O(e^3) \tag{3.3}$$

donde:

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

Llamaremos:

$$e_n = x_n - x^* \tag{3.4}$$

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.

3.3. Regula Falsi

Para *regula falsi*, recordemos la ecuación (2.4):

$$x_{n+1} = x_n - f(x_n) \cdot \frac{x_n - x_0}{f(x_n) - f(x_0)} \tag{3.5}$$

Sea x^* el cero, en términos del error $e_n = x_n - x^*$ usando (3.3) podemos escribir:

$$\begin{aligned} e_{n+1} &= e_n - f(x_n) \cdot \frac{e_n - e_0}{f(x_n) - f(x_0)} \\ &= e_n - \left(f'(x^*)e_n(1+Me_n) + O(e_n^3)\right) \cdot \frac{e_n - e_0}{f(x^*) - f'(x^*)e_n(1+Me_n) + O(e_n^3)} \\ &= e_n \cdot \left(1 - \frac{f'(x^*)e_0}{f(x_0)}\right) + O(e_n^2) \end{aligned}$$

Como $e_{n+1} \approx Ce_n$, la convergencia es lineal.

3.4. Método de Newton

Considere la figura 3.1, donde:

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

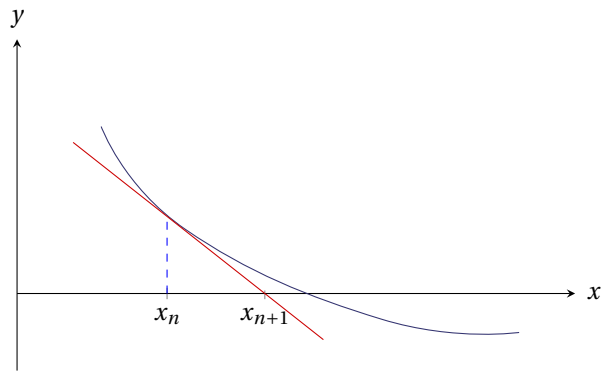


Figura 3.1 – Una iteración del método de Newton.

Considere nuevamente la aproximación (3.3), con $e_n = x_n - x^*$, usando la serie geométrica tenemos:

$$\begin{aligned}
e_{n+1} &= e_n - \frac{f(x_n)}{f'(x_n)} \\
&= e_n - \frac{f'(x^*)e_n(1+Me_n) + O(e_n^3)}{f'(x^*)(1+2Me_n) + O(e_n^2)} \\
&= e_n - e_n \frac{1+Me_n + O(e_n^2)}{1+2Me_n + O(e_n^2)} \\
&= e_n - e_n\left(1+Me_n + O(e_n^2)\right)\left(1-2Me_n + O(e_n^2)\right) \\
&= e_n - e_n\left(1-Me_n + O(e_n^2)\right) \\
&= Me_n^2 + O(e_n^3) \\
&= \frac{f''(x^*)}{2f'(x^*)} \cdot e_n^2 + O(e_n^3)
\end{aligned}$$

O sea, el método de Newton es cuadrático si $f'(x^*) \neq 0$. 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 $0 < a \leq 1/2$, o sea, conocemos x^* con n cifras a la iteración k. Por la forma aproximada del error:

$$\begin{aligned}
\left|\frac{e_{k+1}}{x^*}\right| &\approx |M| \cdot \left|\frac{e_k^2}{x^*}\right| \\
&= |Mx^* a^2| \cdot 10^{-2n} \\
&\leq \left|\frac{Mx^*}{4}\right| \cdot 10^{-2n}
\end{aligned}$$

Esto corresponde a conocer x^* con $2n$ cifras (siempre que $|Mx^*/4| \leq 1/2$, claro).

3.5. Método de la Secante

Considere la figura 3.2, donde:

$$x_{n+2} = x_{n+1} - f(x_{n+1}) \cdot \frac{x_{n+1} - x_n}{f(x_{n+1}) - f(x_n)} \tag{3.7}$$

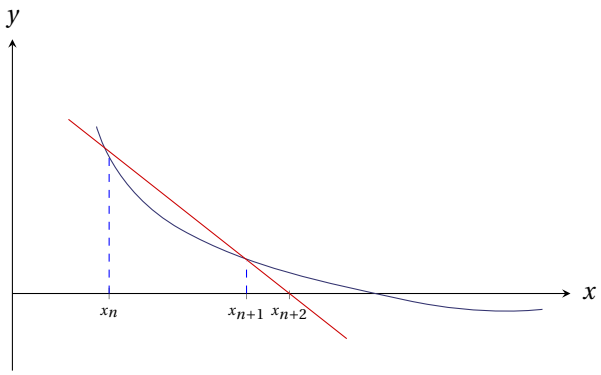


Figura 3.2 – Una iteración del método de la secante.

Considere nuevamente la aproximación (3.3), entonces:

$$
\begin{aligned}
x_{n+2} &= x_{n+1} - f(x_{n+1})\frac{x_{n+1}-x_n}{f(x_{n+1})-f(x_n)} \\
e_{n+2} &= e_{n+1} - f(x_{n+1})\frac{e_{n+1}-e_n}{f(x_{n+1})-f(x_n)} \\
&= \frac{e_n f(x_{n+1}) - e_{n+1} f(x_n)}{f(x_{n+1})-f(x_n)} \\
&= \frac{e_{n+1}\left(f'(x^*)e_n(1+Me_n+O(e_n^2))\right) - e_n\left(f'(x^*)e_{n+1}(1+Me_{n+1}+O(e_{n+1}^2))\right)}{f'(x^*)e_n(1+Me_n+O(e_n^2)) - f'(x^*)e_{n+1}(1+Me_{n+1}+O(e_{n+1}^2))} \\
&= \frac{Me_ne_{n+1}f'(x^*)(e_n-e_{n+1}+O(e_n^2+e_{n+1}^2))}{f'(x^*)(e_n-e_{n+1}+O(e_n^2)+O(e_{n+1}^2))} \\
&= Me_ne_{n+1}\frac{1+O(e_n+e_{n+1})}{1+O(e_n+e_{n+1})} \\
&= Me_ne_{n+1}(1+O(e_n+e_{n+1}))
\end{aligned}
$$

Si despreciamos el error y suponemos:

$$|e_{n+1}| = C|e_n|^p$$

Con lo que:

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

Por otro lado, de la relación para el error:

$$
\begin{aligned}
|e_{n+2}| &= |Me_{n+1}e_n| \\
&= |M|\cdot|C|\cdot|e_n|^p\cdot|e_n| \\
&= |M|C|e_n|^{p+1}
\end{aligned}
$$

Por lo tanto, para el método de la secante:

$$C^{p+1}|e_n|^{p^2} = |M|C|e_n|^{p+1} \tag{3.8}$$

Deben coincidir las potencias de la variable e_n:

$$p^2 = p + 1$$

De acá, dado que $p > 0$:

$$p = \tau = \frac{1+\sqrt{5}}{2} \approx 1{,}618$$

Por lo tanto, el método de la secante es *superlineal*. Tiene la ventaja de ser simple y rápido, y no requiere derivadas (en muchos casos la función no se conoce explícitamente, es el resultado de un proceso complejo).

Una discusión accesible de los problemas a resolver para construir una rutina «para uso general» es la de Kahan [3]. Kahan [4] revisa la teoría que sustenta búsqueda de ceros sin suponer que estamos «muy cerca».

Bibliografía

[1] N. G. de Bruijn: *Asymptotic Methods in Analysis*. North Holland, second edition, 1961.

[2] A. J. Hildebrand: *Math 595AMA: Asymptotic methods in analysis*. https://faculty.math.illinois.edu/~hildebr/595ama, September 2009. Department of Mathematics, University of Illinois at Urbana-Champaign.

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

[4] William M. Kahan: *Lecture notes on real root-finding*. https://people.eecs.berkeley.edu/~wkahan/Math128/RealRoots.pdf, March 2016. Electrical Engineering and Computer Science, University of California at Berkeley.

Clase 4

Otros métodos para encontrar ceros

Discutiremos algunos métodos adicionales para hallar ceros, junto con analizar su convergencia.

4.1. Método de Newton modificado

Si f tiene un cero múltiple en x^*, digamos $f(x) = (x-x^*)^m g(x)$ con $g(x^*) \neq 0$ (x^* es un cero de multiplicidad m), podemos escribir por el teorema de Taylor:

$$g(x) = g(x^*) + O(x-x^*)$$

Tenemos además:

$$\begin{aligned} f(x) &= (x-x^*)^m (g(x^*) + O(x-x^*)) \\ &= (x-x^*)^m g(x^*) + O((x-x^*)^{m+1})) \\ f'(x) &= m(x-x^*)^{m-1} g(x) + (x-x^*)^m g'(x) \\ &= m(x-x^*)^{m-1}(g(x^*) + O(x-x^*)) + (x-x^*)^m O(x-x^*) \\ &= m(x-x^*)^{m-1} g(x^*) + O((x-x^*)^m) \end{aligned}$$

Expandiendo nuevamente la fórmula para el método de Newton:

$$\begin{aligned} e_{n+1} &= e_n - \frac{e_n^m g(x^*)(1+O(e_n))}{m e_n^{m-1} g(x^*)(1+O(e_n))} \\ &= e_n - \frac{e_n}{m}(1+O(e_n)) \\ &= \frac{m-1}{m} e_n + O(e_n^2) \end{aligned}$$

Vale decir, la convergencia es lineal si $m > 1$.

Podemos corregir esto, notando que si f tiene un cero múltiple en x^*, $\mu(x) = f(x)/f'(x)$ tiene un cero simple, y se recupera la convergencia cuadrática:

$$\begin{aligned} x_{n+1} &= x_n - \frac{\mu(x_n)}{\mu'(x_n)} \\ &= x_n - \frac{f(x_n)}{f'(x_n)}\left(1 - \frac{f(x_n)f''(x_n)}{f'(x_n)^2}\right)^{-1} \end{aligned} \tag{4.1}$$

Analizamos la convergencia, extendiendo las series:

$$\begin{aligned} f(x) &= (x-x^*)^m\left(g(x^*) + g'(x^*)(x-x^*) + \frac{1}{2}g''(x^*)(x-x^*)^2\right) + O((x-x^*)^{m+3}) \\ f'(x) &= m(x-x^*)^{m-1}\left(g(x^*) + \frac{m+1}{m}g'(x^*)(x-x^*) + \frac{m+2}{2m}g''(x^*)(x-x^*)^2)\right) \\ &\quad + O((x-x^*)^{m+2}) \\ f''(x) &= m(m-1)(x-x^*)^{m-2}\left(g(x^*) + \frac{m+1}{m-1}g'(x^*)(x-x^*) + \frac{(m+2)(m+1)}{2m(m-1)}g''(x^*)(x-x^*)^2\right) \\ &\quad + O((x-x^*)^{m+1}) \end{aligned}$$

substituyendo en la ecuación (4.1) tenemos:

$$e_{n+1} = -\frac{g'(x^*)}{mg(x^*)}e_n^2 + O(e_n^3) \tag{4.2}$$

Como propusimos, el método de Newton modificado siempre tiene convergencia cuadrática.

Resulta de interés el límite:

$$\lim_{x\to x^*}\left(1 - \frac{f(x)f''(x)}{f'(x)^2}\right) = \frac{1}{m} \tag{4.3}$$

Por lo tanto, si f tiene un cero de multiplicidad m en x^*, la iteración:

$$x_{n+1} = x_n - m\frac{f(x_n)}{f'(x_n)} \tag{4.4}$$

converge cuadráticamente a x^*.

Claro que en general no sabemos cuánto es m. Una estrategia es estimar m usando la expresión del límite (4.3) al comienzo, y verificar el valor regularmente luego.

4.2. Métodos de Householder

Una familia de métodos de orden de convergencia arbitrario son los de Householder [4]. La idea es partir con una aproximación de Padé (la razón entre dos polinomios) a la función, en nuestro caso:

$$f(x+h) = \frac{a_0 + h}{b_0 + b_1 h + \cdots + b_{d-1}h^{d-1}} + O(h^d) \tag{4.5}$$

Puede demostrarse que la aproximación (4.5) es única.

La derivación del método de Householder de orden d parte con la aproximación (4.5), y considera la serie de Taylor de $1/f$:

$$\left(\frac{1}{f}\right)(x+h) = \left(\frac{1}{f}\right)(x) + \left(\frac{1}{f}\right)'(x)h + \frac{1}{2!}\left(\frac{1}{f}\right)''(x)h^2 + \cdots + \frac{1}{d!}\left(\frac{1}{f}\right)^{(d)}(x)h^d + O(h^{d+1})$$

Esto debe coincidir con la fracción (4.5), en particular, el coeficiente de h^d se anula; al multiplicar por $a_0 + h$ este es:

$$0 = a_0 \frac{\left(\frac{1}{f}\right)^{(d)}(x)}{d!} + \frac{\left(\frac{1}{f}\right)^{(d-1)}(x)}{(d-1)!}$$

de donde despejamos a_0, y sabemos que la aproximación (4.5) a f se anula para $h = -a_0$, o sea la siguiente aproximación a x se calcula:

$$x_{n+1} = x_n + d \frac{\left(\frac{1}{f}\right)^{(d-1)}(x_n)}{\left(\frac{1}{f}\right)^{(d)}(x_n)} \tag{4.6}$$

Por su derivación, es claro que el método de Householder de orden d converge de orden $d+1$. Para $d = 1$ resulta el método de Newton, $d = 2$ da el método de Halley:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \left(1 - \frac{f(x_n) f''(x_n)}{2 f'(x_n)^2}\right)^{-1} \tag{4.7}$$

Para el método de Halley nuestras técnicas para estimar el error dan:

$$e_{n+1} = -\frac{2f'(x^*)f'''(x^*) - 3f''(x^*)^2}{12 f'(x^*)^2} e_n^3 + O(e_n^4) \tag{4.8}$$

el método es cúbico.

4.3. Otro método cúbico

La instructiva derivación siguiente es de Sebah y Gourdon [6]. Podemos extender el método de Newton, aproximando la función mediante la serie de Taylor hasta el término cuadrático:

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

y buscar h más cercano a cero que anula el polinomio:

$$h = -\frac{f'(x)}{f''(x)} \left(1 - \sqrt{1 - \frac{2f(x)f''(x)}{f'(x)^2}}\right)$$

Podemos ahorrarnos la raíz cuadrada expandiendo para α pequeño (buscamos $f(x) = 0$) por el teorema del binomio:

$$\begin{aligned} 1 - \sqrt{1-\alpha} &= \frac{\alpha}{2} + \frac{\alpha^2}{8} + O(\alpha^3) \\ &= \frac{\alpha}{2}\left(1 + \frac{\alpha}{4}\right) + O(\alpha^3) \end{aligned}$$

al retener hasta el término cuadrático:

$$\begin{aligned} h &\approx -\frac{f'(x)}{f''(x)} \cdot \frac{f(x)f''(x)}{f'(x)^2} \left(1 + \frac{f(x)f''(x)}{2f'(x)^2}\right) \\ &= -\frac{f(x)}{f'(x)} \left(1 + \frac{f(x)f''(x)}{2f'(x)^2}\right) \end{aligned}$$

queda:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}\left(1 + \frac{f(x_n)f''(x_n)}{2f'(x_n)^2}\right) \tag{4.9}$$

Nuestras técnicas para estimar el error dan:

$$e_{n+1} = -\frac{f'(x^*)f'''(x^*) - 3f''(x^*)^2}{6f'(x^*)^2}e_n^3 + O(e_n^4) \tag{4.10}$$

Como esperábamos, el método es cúbico siempre que el cero sea simple.

4.4. Método de Muller

Una extensión del método de la secante es el método de Muller [5]. La idea es tomar los últimos tres puntos, interpolar mediante un polinomio de segundo grado y elegir el siguiente punto como una intersección con el eje. En detalle, dados puntos $(x_n, y_n), (x_{n+1}, y_{n+1}), (x_{n+2}, y_{n+2})$, buscamos $x_{n+3} = x_{n+2} + h$ que anula la función cuadrática que interpola entre esos puntos. La función cuadrática que interpola puede escribirse en la forma de Newton:

$$\begin{aligned} f(x) &= f[x_{n+2}] + f[x_{n+1}, x_{n+2}](x - x_{n+1}) + f[x_n, x_{n+1}, x_{n+2}](x - x_{n+2})(x - x_{n+1}) \\ &= f[x_{n+2}] + w(x - x_{n+2}) + f[x_n, x_{n+1}, x_{n+2}](x - x_{n+2})^2 \\ &= f[x_{n+2}] + wh + f[x_n, x_{n+1}, x_{n+2}]h^2 \end{aligned}$$

Acá usamos diferencias divididas:

$$\begin{aligned} f[x_0] &= f(x_0) \\ f[x_0, x_1] &= \frac{f[x_1] - f[x_0]}{x_1 - x_0} \\ f[x_0, x_1, x_2] &= \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0} \end{aligned}$$

junto con:

$$w = f[x_{n+1}, x_{n+2}] + f[x_n, x_{n+2}] - f[x_n, x_{n+1}]$$

Nos interesa el cero de menor valor absoluto de la cuadrática en h. Pero eso lleva a cancelación en la fórmula cerca del cero (valor pequeño de $f(x_{n+2})$). Dividiendo la cuadrática por h^2 obtenemos una cuadrática en $1/h$, que resolvemos para el cero de mayor magnitud obteniendo:

$$h = -\frac{2f(x_{n+2})}{w + \operatorname{sgn}(w)\sqrt{w^2 - 4f(x_{n+2})f[x_n, x_{n+1}, x_{n+2}]}} \tag{4.11}$$

donde usamos la función signo:

$$\operatorname{sgn}(x) = \begin{cases} -1 & x < 0 \\ 0 & x = 0 \\ 1 & x > 0 \end{cases}$$

O sea, la iteración es:

$$x_{n+3} = x_{n+2} - \frac{f(x_{n+2})}{w + \operatorname{sgn}(w)\sqrt{w^2 - 4f(x_{n+2})f[x_n, x_{n+1}, x_{n+2}]}} \tag{4.12}$$

Sospechando que el método es supralineal, pero no más que cuadrático, usamos:

$$f(x^* + h) = f'(x^*)h + \frac{1}{2}f''(x^*)h^2 + \frac{1}{6}f'''(x^*)h^3 + O(h^4)$$

y bajo el supuesto:

$$e_{n+1} = Ce_n^p$$

de la iteración (4.12) después de simplificar resulta $p^3 - p^2 - p - 1 = 0$, cuyo cero real es 1,8392867552. Tenemos:

$$|C| = \left|\frac{f'''(x^*)}{6f'(x^*)}\right|^{(p-1)/2} \tag{4.13}$$

Un problema del método de Muller es que aún si todos los valores son reales el resultado puede ser complejo.

4.5. Interpolación inversa

En vez de interpolar los $f(x)$ y buscar el punto donde la interpolación se anula, podemos interpolar la función inversa y evaluar para $y = 0$. Esto tiene la ventaja de no requerir raíces (y no lleva a complejos). Si usamos una función cuadrática, de la forma de Lagrange evaluada en $y = 0$ tenemos:

$$x_{n+3} = \frac{y_{n+1}y_{n+2}}{(y_n - y_{n+1})(y_n - y_{n+2})}x_n + \frac{y_n y_{n+2}}{(y_{n+1} - y_n)(y_{n+1} - y_{n+2})}x_{n+1} + \frac{y_n y_{n+1}}{(y_{n+2} - y_n)(y_{n+2} - y_{n+1})}x_{n+2} \tag{4.14}$$

Substituyendo $x = x^* + e$, evaluando (4.14) para x_0, x_1 y x_2 para obtener x_3, es claro de la fórmula que el término más significativo es proporcional a $e_0e_1e_2$, calculamos el límite:

$$\lim_{\substack{e_0\to 0\\ e_1\to 0\\ e_2\to 0}} \frac{e_3}{e_0e_1e_2} = -\frac{f'(x^*)f'''(x^*) - 3f''(x^*)^2}{6f'(x^*)}$$

Bajo el supuesto $|e_{n+1}| = C|e_n|^p$ obtenemos la ecuación:

$$C^{p^2+p+1}|e_n|^{p^3} = \frac{f'(x^*)f'''(x^*) - 3f''(x^*)^2}{6f'(x^*)}C^{p+2}|e_n|^{p^2+p+1}$$

El orden de convergencia es nuevamente el cero real de $p^3 - p^2 - p - 1$, el valor 1,8392867552.

Un problema de este método es que si resultan dos puntos iguales, falla.

4.6. El método de Brent

Un método híbrido muy popular es el de Brent [3, capítulo 4]. La idea es usar un método de interpolación (interpolación cuadrática inversa o secante) si se puede, recurriendo a bisección si lo anterior falla o se ve que converge lentamente. Exige iniciar con puntos que acotan el cero, elige el tercer punto por bisección, de allí usa interpolación cuadrática inversa si se puede (los tres puntos son diferentes) o secante, y cada cierto número de pasos fuerza usar al menos un paso de bisección para asegurar que el rango de búsqueda se encoja con rapidez. Garantiza convergencia razonable, incluso para funciones de muy mal comportamiento; y aprovecha la convergencia rápida de los métodos de interpolación cuando son aplicables.

La versión definitiva es el programa 4.1 en Algol 60 [1] como dado por Brent [2, sección 4.6], reindentado para gustos modernos. Brent lo discute en detalle, incluyendo consideraciones de error de redondeo en cálculo en punto flotante. El algoritmo usa el valor ϵ (argumento macheps) de precisión relativa de punto flotante, y retorna un cero con tolerancia $6\epsilon|b|+2t$, donde t es una tolerancia positiva. Supone que $f(a)$ y $f(b)$ tienen signo opuesto. En un paso típico, tenemos tres puntos a, b y c, tales que $f(b)f(c)<0$, $|f(b)|\le|f(c)|$, y a puede coincidir con c. b es la mejor aproximación hasta el momento al cero x^*, a es el valor previo de b y x^* está entre b y c. Inicialmente $a=c$.

Si $f(b)=0$, estamos listos. Esto puede ocurrir por calcular f en forma aproximada.

Si $f(b)\neq 0$, sea $m=\frac{1}{2}(c-b)$. No se retorna $\frac{1}{2}(b+c)$ en cuanto $|m|\le 2\delta$, ya que si tenemos convergencia superlineal probablemente b sea mucho mejor aproximación a x^*. Si $|m|\le\delta$, retornamos b (sabemos que el error es a lo más δ en tal caso).

Si no se dan las anteriores, interpolamos (o extrapolamos) linealmente entre a y b, dando un nuevo punto i. Más adelante discutimos la interpolación cuadrática. Para evitar problemas de rebalse o división por cero, buscamos números p y q tales que $i=b+p/q$, y omitimos la división si $2|p|\ge 3|mq|$ (no se requiere en tal caso). Como $0<|f(b)|\le|f(a)|$ (vea más adelante), podemos calcular $s=f(a)/f(b)$, $p\pm(a-b)s$ y $q\mp(1-s)$ sin meternos en problemas. Ahora definimos:

$$b''=\begin{cases} i & \text{si } i \text{ está entre } b \text{ y } b+m \text{ (interpolación)}\\ b+m & \text{caso contrario (bisección)}\end{cases}$$

$$b'=\begin{cases} b'' & |b-b''|>\delta\\ b+\delta\,\mathrm{sgn}(m) & \text{caso contrario (paso } \delta)\end{cases}$$

Aún evitando pasos δ la convergencia puede ser muy lenta, el algoritmo fuerza ocasionales biseccciones: sea e el valor de p/q el penúltimo paso, si $|e|<\delta$ o $|p/q|\ge\frac{1}{2}\delta$, haga una bisección. O sea, e se reduce al menos en un factor de 2 cada segundo paso, y si $|e|<\delta$ damos un paso de bisección. Luego de una bisección el avance es $e=m$. Experimentos mostraron que el criterio más simple de usar p/q del último paso frena la convergencia para funciones de buen comportamiento.

Si los puntos a, b y c son diferentes, podemos recurrir a interpolación cuadrática inversa, que debiera dar una mejor estimación de x^*. En este proceso hay que tener cuidado de no caer en rebalse o división por cero. Como b es la aproximación más reciente y a es el valor previo de b, si $|f(b)|\ge|f(a)|$ damos un paso de bisección. En caso contrario, es $|f(b)|<|f(a)|\le|f(c)|$, una forma segura de hallar i es calcular:

$$\begin{aligned} r_1&=\frac{f(a)}{f(c)}\\ r_2&=\frac{f(b)}{f(c)}\\ r_3&=\frac{f(b)}{f(a)}\\ p&=\pm r_3((c-b)r_1(r_1-r_2)-(b-a)(r_2-1))\\ q&=\mp(r_1-1)(r_2-1)(r_3-1)\end{aligned}$$

con lo que $i=b+p/q$, pero (igual que antes) no efectuamos la división a menos que resulte útil. Pero si entre $(b,f(b))$ y $(c,f(c))$ no estamos en la misma rama de la parábola, esta no es una buena aproximación a f, aceptamos i solo si está entre b y c, y hasta tres cuartas partes del camino de b a c (considere el caso extremo de $f(b)=-f(c)$, con la parábola con una tangente vertical en $(c,f(c))$). O sea, rechazamos i si $2|p|\ge 3|mq|$.

Ejercicios

1. Una forma alternativa de derivar la iteración de Halley (4.7) es aplicar el método de Newton a la función:

$$\mu(x) = \frac{f(x)}{\sqrt{|f'(x)|}}$$

   lo que es válido siempre que el cero sea simple ($f'(x^*) \neq 0$).

2. Analice la convergencia de los métodos cúbicos para ceros múltiples.

```
real procedure zero (a, b, macheps, t, f);
value a, b, macheps, t; real a, b, macheps, t;
real procedure f;

begin
  comment:
    Procedure zero returns a zero x of function f in the given
    interval [a, b], to within a tolerance 6 macheps |x| + 2 t,
    where macheps is the relative machine precision and t is a
    positive tolerance. The procedure assumes that f(a) and f(b)
    have different signs;

  real c, d, e, fa, fb, fc, tol, m, p, q, r, s;

  fa := f(a); fb := f(b);
  int: c := a; fc := fa; d := e := b - a;
  ext: if abs(fc) < abs(fb) then begin
          a := b; b := c; c := a;
          fa := fb; fb := fc; fc := fa
        end;
       tol := 2 × macheps × abs(b) + t; m := 0.5 × (c - b);
       if abs(m) > tol ∧ fb ≠ 0 then begin
         comment: See if a bisection is forced;
           if abs(e) < tol ∨ abs(fa) ≤ abs(fb) then
             d := e := m
           else begin
             s := fb / fa;
             if a = c then begin comment: Linear interpolation;
                 p := 2 × m × s; q := 1 - s
               end
             else begin comment: Inverse quadratic interpolation;
                 q := fa / fc; r := fb / fc;
                 p := s × (2 × m × q × (q - r) - (b - a) ×
(r - 1));
                 q := (q - 1) × (r - 1) × (s - 1)
               end;
           if p > 0 then
             q := -q
           else
             p := -p;
           s := e; e := d;
           if 2 × p < 3 × m × q - abs(tol × q)
               ∧ p < abs(0.5 × s × q) then
             d := p / q
           else
             d := e := m
         end;
       a := b; fa := fb;
       b := b + (if abs(d) > tol then d
                 else if m > 0 then tol else -tol);
       fb := f(b);
       go to if fb > 0 ≡ fc > 0 then int else ext
     end;
     zero := b
  end zero;
```

Listado 4.1 – El algoritmo de Brent

Bibliografía

[1] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A. J. Perlis, Rutishauser H., K. Samuelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, M. Woodger, R. M. De Morgan, I. D. Hill, and B. A. Wichmann: *Modified report on the algorithmic language Algol 60.* Computer Journal, 19(4):364–379, November 1976. IFIP Working Group 2.1.

[2] Richard P. Brent: *An algorithm with guaranteed convergence for finding a zero of a function.* Computer Journal, 14(4):422–425, January 1971.

[3] Richard P. Brent: *Algorithms for Minimization without Derivatives.* Prentice Hall, Inc., 1973.

[4] Alston Scott Householder: *The Numerical Treatment of a Single Nonlinear Equation.* McGraw-Hill, 1970.

[5] David E. Muller: *A method for solving algebraic equations using an automatic computer.* Mathematical Tables and Other Aids to Computation, 10(56):208–215, October 1956.

[6] Pascal Sebah and Xavier Gourdon: *Newton's method and higher order iterations.* http://numbers.computation.free.fr/Constants/Algorithms/newton.ps, October 2001.

Clase 5

Iteración de Punto Fijo

El método de iteración de punto fijo es simple, y adecuado en muchas situaciones. Más aún, todos los métodos iterativos vistos son de la forma:

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

con lo que estudiar sus propiedades en mayor detalle ilumina los métodos más complejos.

Definición 5.1. Sea $g(x)$ una función. Un *punto fijo* de g es x^* tal que $x^* = g(x^*)$.

Teorema 5.1 (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 intermedio, hay $x^* \in [a,b]$ tal que $f(x^*) = 0$, o sea, $x^* = g(x^*)$. □

Definición 5.2. 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:

$$|g(x) - g(y)| \leq L|x - y| \tag{5.1}$$

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

Teorema 5.2 (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:

$$|c_1 - c_2| = |g(c_1) - g(c_2)| \leq L|c_1 - c_2| \tag{5.2}$$

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 5.1). Definamos la secuencia:

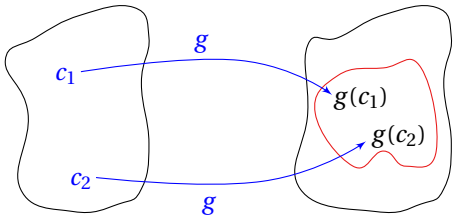


Figura 5.1 – Acotamos el área hasta converger en un punto.

$$x_{n+1} = g(x_n) \tag{5.3}$$

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:

$$\begin{aligned} |x_{n+1} - x^*| &= |g(x_n) - g(x^*)| && (5.4)\\ &\leq L|x_n - x^*| && (5.5)\\ &\leq \dots && (5.6)\\ &\leq L^{n+1}|x_0 - x^*| && (5.7) \end{aligned}$$

Como $|L| < 1$, $L^n \to 0$, y el lado izquierdo también tiende a 0 (para llegar a (5.7) solo tenemos que desarrollar paso a paso (5.5)).

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:

$$|x_n - x^*| \leq \frac{L^n}{1-L}|x_1 - x_0| \tag{5.8}$$

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]):

$$\frac{g(x)-g(y)}{x-y} = g'(\zeta), \qquad x \leq \zeta \leq y \tag{5.9}$$

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, solo demostrar que existe. Y por favor, ζ se lee «zeta».

5.1. Análisis de convergencia

Igual que antes, nos interesa analizar la convergencia de esta técnica. Usamos nuevamente series de Taylor:

$$\begin{aligned} g(x) &= g(x^*) + g'(x^*)(x-x^*) + O((x-x^*)^2) \\ &= x^* + g'(x^*)(x-x^*) + O((x-x^*)^2) \end{aligned} \tag{5.10}$$

Igual que antes, con $e_n = x_n - x^*$ obtenemos:

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

Vemos que la convergencia es lineal si $g'(x^*) \neq 0$, y en particular que converge solo si $|g'(x^*)| < 1$.

Bibliografía

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

Clase 6

Sistemas de ecuaciones no lineales

Hay situaciones en que estamos interesados en resolver sistemas de ecuaciones, tanto lineales como no lineales. La solución de sistemas lineales ya se vio en el colegio, y se profundizará bastante en ramos posteriores. Hay una muy linda teoría al respecto (el álgebra lineal, ver por ejemplo a Treil [9]), y un conjunto fascinante de algoritmos al efecto. Acá nos interesan sistemas no lineales, en los cuales al menos una variable aparece en una dependencia no lineal.

6.1. Iteración de punto fijo

Nuestro problema es el siguiente: hallar el vector $\mathbf{x}$ tal que se cumple:

$$\mathbf{f}(\mathbf{x}) = \mathbf{0} \tag{6.1}$$

Claramente, las dimensiones de $\mathbf{x}$ y de $\mathbf{f}$ deben ser iguales. En forma explícita nuestro problema es:

$$\begin{aligned} f_1(x_1, x_2, \ldots, x_n) &= 0 \\ f_2(x_1, x_2, \ldots, x_n) &= 0 \\ &\vdots \\ f_n(x_1, x_2, \ldots, x_n) &= 0 \end{aligned}$$

Formalmente la ecuación (6.1) es idéntica al problema tratado en la clase 5, nuevamente siempre podremos escribir:

$$\mathbf{x} = \mathbf{g}(\mathbf{x})$$

(siempre funciona la transformación $\mathbf{g}(\mathbf{x}) = \mathbf{x} + \mathbf{A}\mathbf{f}(\mathbf{x})$ para una matriz $\mathbf{A}$ no singular) y nos interesa un punto fijo $\mathbf{x}^*$ de $\mathbf{g}$.

6.2. Métodos de Newton y cuasi-Newton

Sistemas de ecuaciones pueden expresarse:

$$\mathbf{f}(\mathbf{x}) = \mathbf{0}$$

Muchas veces las funciones no están dadas explícitamente, pueden ser el resultado de un complicado cálculo o incluso ser el resultado de un experimento.

Usando la versión multivariable del teorema de Taylor, podemos aproximar:

$$\mathbf{f}(\mathbf{x}) = \mathbf{f}(\mathbf{x}^*) + \mathbf{f}'(\mathbf{x}^*)(\mathbf{x}-\mathbf{x}^*) + \cdots \tag{6.2}$$

Acá es $\mathbf{f}'(\mathbf{x})$ es la matriz jacobiana:

$$\mathbf{f}'(\mathbf{x}) = \left(\frac{\partial f_i}{\partial x_j}\right) \tag{6.3}$$

Como el método de Newton en una dimensión, esto sugiere la iteración:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - (\mathbf{f}'(\mathbf{x}_k))^{-1}\mathbf{f}(\mathbf{x}_k)$$

En forma muy similar al caso en una dimensión, puede demostrarse que el método converge en forma cuadrática:

$$\|\mathbf{x}_{k+1} - \mathbf{x}^*\| \leq \alpha \|\mathbf{x}_k - \mathbf{x}^*\|^2$$

Lo malo es que (6.3) en cada iteración requiere calcular n^2 derivadas si $\mathbf{f}$ y $\mathbf{x}$ son de dimensión n, y resolver el sistema de ecuaciones implícito en esto requiere $O(n^3)$ operaciones. Nos interesa deducir un método similar al de la secante, menos costoso. Hay una variedad de métodos, que en su conjunto se conocen como *métodos cuasi-Newton*. Denis y Moré [3] y Martínez [6] discuten la teoría y exploran variantes. Una técnica reciente en esta línea es la de Klement [5]. Acá consideraremos el primero y más simple de ellos, el método de Broyden [2].

La idea base del método es aproximar la matriz jacobiana $\mathbf{f}'(\mathbf{x}_k)$ mediante una matriz $\mathbf{B}_k$, calculada usando $\mathbf{f}(\mathbf{x}_k)$ y $\mathbf{f}(\mathbf{x}_{k-1})$ junto con $\mathbf{B}_{k-1}$. En vista de (6.2), es razonable exigir:

$$\mathbf{f}(\mathbf{x}_k) = \mathbf{f}(\mathbf{x}_{k-1}) + \mathbf{B}_k(\mathbf{x}_k - \mathbf{x}_{k-1})$$

Para simplificar notación, llamamos:

$$\mathbf{s}_k = \mathbf{x}_k - \mathbf{x}_{k-1}$$
$$\mathbf{y}_k = \mathbf{f}(\mathbf{x}_k) - \mathbf{f}(\mathbf{x}_{k-1})$$

y escribimos:

$$\mathbf{B}_k\mathbf{s}_k = \mathbf{y}_k \tag{6.4}$$

Si $n = 1$ (el número de ecuaciones, y la dimensión de las matrices $\mathbf{B}_k$), la ecuación (6.4) define $\mathbf{B}_k$ en forma única, es el método de secante. Si $n > 1$, podemos argüir que solo conocemos la variación de $\mathbf{f}$ a lo largo de $\mathbf{s}_k$; si tenemos una aproximación previa $\mathbf{B}_{k-1}$, el cambio no aporta nueva información en direcciones ortogonales a $\mathbf{s}_k$, vale decir debiéramos exigir:

$$\mathbf{B}_k\mathbf{z} = \mathbf{B}_{k-1}\mathbf{z} \quad \text{si } \langle \mathbf{s}_k, \mathbf{z}\rangle = 0 \tag{6.5}$$

Resulta que (6.4) y (6.5) determinan $\mathbf{B}_k$ en forma única. Llamemos $\mathbf{B}_k = \mathbf{B}_{k-1} + \mathbf{X}$, en esos términos (6.5) dice que cada fila de $\mathbf{X}$ es un múltiplo de $\mathbf{s}_k^T$. Para simplificar notación, temporalmente anotemos:

$$\overline{\mathbf{B}} = \mathbf{B}_k$$
$$\mathbf{B} = \mathbf{B}_{k-1}$$
$$\mathbf{s} = \mathbf{s}_k$$
$$\mathbf{y} = \mathbf{y}_k$$

La observación anterior dice que:

$$\mathbf{X} = \mathbf{v}\mathbf{s}^T$$

o sea:

$$\begin{aligned} \overline{\mathbf{B}}\mathbf{s} &= \mathbf{y} \\ (\mathbf{B} + \mathbf{v}\mathbf{s}^T)\mathbf{s} &= \mathbf{y} \\ (\mathbf{v}\mathbf{s}^T)\,\mathbf{s} &= \mathbf{y} - \mathbf{B}\mathbf{s} \\ \mathbf{v} &= \frac{\mathbf{y} - \mathbf{B}\mathbf{s}}{\mathbf{s}^T\mathbf{s}} \end{aligned}$$

Como $\mathbf{s}^T\mathbf{s} = \langle \mathbf{s}, \mathbf{s} \rangle$, en términos de la notación original:

$$\mathbf{B}_k = \mathbf{B}_{k-1} + \frac{(\mathbf{y}_k - \mathbf{B}_{k-1}\mathbf{s}_k)\mathbf{s}_k^T}{\langle \mathbf{s}_k, \mathbf{s}_k \rangle} \tag{6.6}$$

Sigue pendiente el problema de resolver:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{B}_k^{-1}\mathbf{f}(\mathbf{x}_k)$$

Un lema muestra cómo hacerlo con mínimo costo. El resultado es de Sherman y Morrison [7, 8], lo que citamos es la forma de Bartlett [1]. Como es tradicional, Hager [4] al discutir la historia y aplicaciones halla que el resultado es anterior.

Lema 6.1 (Fórmula de Sherman-Morrison)**.** *Sea* $\mathbf{A}$ *una matriz no singular, y sean* $\mathbf{u}$, $\mathbf{v}$ *vectores. Entonces* $\mathbf{A} + \mathbf{u}\mathbf{v}^T$ *es no singular si y solo si* $\sigma = 1 + \langle \mathbf{v}, \mathbf{A}^{-1}\mathbf{u} \rangle \neq 0$. *Si* $\sigma \neq 0$, *es:*

$$(\mathbf{A} + \mathbf{u}\mathbf{v}^T)^{-1} = \mathbf{A}^{-1} - \frac{1}{\sigma}\mathbf{A}^{-1}\mathbf{u}\mathbf{v}^T\mathbf{A}^{-1}$$

Demostración. Siempre que $\mathbf{A}$ no sea singular, podemos expresar:

$$\mathbf{A} + \mathbf{u}\mathbf{v}^T = \mathbf{A}(\mathbf{I} + \mathbf{A}^{-1}\mathbf{u}\mathbf{v}^T)$$

Consideremos entonces la matriz siguiente para vectores $\mathbf{x}$ e $\mathbf{y}$:

$$\mathbf{I} + \mathbf{x}\mathbf{y}^T$$

Sospechamos que su inversa es de la forma siguiente, para un escalar α:

$$\mathbf{I} + \alpha\mathbf{x}\mathbf{y}^T$$

Multiplicando por la inversa propuesta, como $\mathbf{y}\mathbf{x} = \langle \mathbf{x}, \mathbf{y} \rangle$ es un escalar:

$$\begin{aligned} (\mathbf{I} + \mathbf{x}\mathbf{y}^T) \cdot (\mathbf{I} + \alpha\mathbf{x}\mathbf{y}^T) &= \mathbf{I} + \mathbf{x}\mathbf{y}^T + \alpha\mathbf{x}\mathbf{y}^T + \alpha\mathbf{x}\mathbf{y}^T\mathbf{x}\mathbf{y}^T \\ &= \mathbf{I} + (1 + \alpha + \alpha\langle \mathbf{x}, \mathbf{y} \rangle)\mathbf{x}\mathbf{y}^T \end{aligned}$$

Esto debe ser $\mathbf{I}$, el escalar del segundo término es 0, de donde despejamos:

$$\alpha = -\frac{1}{1 + \langle \mathbf{x}, \mathbf{y} \rangle}$$

De la misma forma vemos que:

$$(\mathbf{I}+\alpha\mathbf{x}\mathbf{y}^T)\cdot(\mathbf{I}+\mathbf{x}\mathbf{y}^T)=\mathbf{I}$$

y realmente es inversa.

Con $\mathbf{x}=\mathbf{A}^{-1}\mathbf{u}$, $\mathbf{y}=\mathbf{v}$ tenemos así la inversa para:

$$\begin{aligned}(\mathbf{A}+\mathbf{u}\mathbf{v}^T)^{-1} &= (\mathbf{A}(\mathbf{I}+\mathbf{A}^{-1}\mathbf{u}\mathbf{v}^T))^{-1}\\ &= (\mathbf{I}+\mathbf{A}^{-1}\mathbf{u}\mathbf{v}^T)^{-1}\mathbf{A}^{-1}\\ &= (\mathbf{I}-\frac{1}{\sigma}\mathbf{A}^{-1}\mathbf{u}\mathbf{v}^T)\mathbf{A}^{-1}\\ &= \mathbf{A}^{-1}-\frac{1}{\sigma}\mathbf{A}^{-1}\mathbf{u}\mathbf{v}^T\mathbf{A}^{-1}\end{aligned}$$

Es claro que si $\sigma=0$, no hay inversa. □

El algoritmo parte entonces con una estimación inicial $\mathbf{x}_0$, y una estimación inicial $\widetilde{\mathbf{J}}$ de la matriz jacobiana en ese punto (por ejemplo, estimando las derivadas mediante diferencias en cada dimensión) y calculamos una aproximación inicial $\mathbf{H}_0=\widetilde{\mathbf{J}}^{-1}$. De allí la iteración procede usando la fórmula de Sherman-Morrison para actualizar la inversa. Usamos las abreviaturas $\Delta\mathbf{x}_k=\mathbf{x}_{k+1}-\mathbf{x}_k$ y $\Delta\mathbf{f}(\mathbf{x}_k)=\mathbf{f}(\mathbf{x}_{k+1})-\mathbf{f}(\mathbf{x}_k)$:

$$\begin{aligned}\mathbf{x}_{n+1} &= \mathbf{x}_n-\mathbf{H}_n\mathbf{f}(\mathbf{x}_n)\\ \mathbf{H}_{n+1} &= \mathbf{H}_n+\frac{\Delta\mathbf{x}_{n+1}-\mathbf{H}_n\Delta\mathbf{f}(\mathbf{x}_{n+1})}{\Delta\mathbf{x}_{n+1}^T\mathbf{H}_n\Delta\mathbf{f}(\mathbf{x}_{n+1})}\Delta\mathbf{f}^T(\mathbf{x}_{n+1})\end{aligned}$$

Condiciones de término apropiadas pueden ser $\|\mathbf{x}_{n+1}-\mathbf{x}_n\|\le\epsilon$, o una variante relativa de esta, como $\|\mathbf{x}_{n+1}-\mathbf{x}_n\|/\|\mathbf{x}_{n+1}\|\le\epsilon$; o $\|\mathbf{f}(\mathbf{x}_{n+1})\|\le\epsilon$.

Ejercicios

1. En su publicación, Broyden [2] prueba su método con varios sistemas de ecuaciones. Uno es:

$$\begin{aligned}f_1 &= -(3+\alpha x_1)x_1+2x_2-\beta\\ f_i &= x_{i-1}-(3+\alpha x_i)x_i-2x_{i+1}-\beta \qquad\qquad i=2,3,\ldots,n-1\\ f_n &= x_n-(3+\alpha x_n)x_n-\beta\end{aligned}$$

Valores de los parámetros usados están dados en el cuadro 6.1, valores iniciales son siempre $x_i=1{,}0$. Experimente con el método de Newton y el de Broyden con algunas de estas, o con

n	α	β
5	−0,1	1,0
5	−0,5	1,0
10	−0,5	1,0
20	−0,5	1,0

Cuadro 6.1 – Parámetros para ejercicio 1

parámetros diferentes. Compare número de evaluaciones de las funciones y el tiempo total para obtener las soluciones con cinco cifras.

2. Una propuesta para evitar el cálculo inicial de la matriz jacobiana en el método de Broyden es partir con una matriz inicial **H** arbitraria, por ejemplo **I**. Experimente con esto en el ejercicio 1.

3. Otra opción sería ordenar **f** de forma que la derivada de f_i respecto de x_i inicial sea máxima, y aproximar la matriz jacobiana inicial por la matriz diagonal con estas entradas.

Bibliografía

[1] Maurice S. Bartlett: *An inverse matrix adjustment arising in discriminant analysis.* Annals of Mathematical Statistics, 22(1):107–111, March 1951.

[2] Charles G. Broyden: *A class of methods for solving nonlinear simultaneous equations.* Mathematics of Computation, 19(92):577–593, October 1965.

[3] J. E. Dennis and Jorge J. Moré: *Quasi-Newton methods, motivation and theory.* SIAM Review, 19(1):46–89, January 1977.

[4] William W. Hager: *Updating the inverse of a matrix.* SIAM Review, 31(2):221–239, June 1989.

[5] Jan Klement: *On using quasi-Newton algorithms of the Broyden class for model-to-test correlation.* Journal of Aerospace Technology and Management, 6(4), oct/dec 2014.

[6] José Mario Martínez: *Practical quasi-Newton methods for solving nonlinear systems.* Journal of Computational and Applied Mathematics, 124(1-2):97–121, December 2000.

[7] Jack Sherman and Winifred J. Morrison: *Adjustment of an inverse matrix corresponding to changes in the elements of a given column or a given row of the original matrix (abstract).* Annals of Mathematical Statistics, 20(4):621, December 1949.

[8] Jack Sherman and Winifred J. Morrison: *Adjustment of an inverse matrix corresponding to a change in one element of a given matrix.* Annals of Mathematical Statistics, 21(1):124–127, March 1950.

[9] Sergei Treil: *Linear algebra done wrong.* https://www.math.brown.edu/~treil/papers/LADW/book.pdf, September 2017. Department of Mathematics, Brown University.

Clase 7

Interpolación

Hay situaciones en las cuales se conoce el valor de una función solo en algunos puntos dados, y queremos calcular valores en puntos intermedios. Por ejemplo, contamos solo con algunos valores medidos, o hay una tabla de valores numéricos. Este es el problema de interpolación.

Nos dan los valores exactos de una función desconocida en $n+1$ puntos $f(x_0),\dots,f(x_n)$, 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.

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

- Obtener los coeficientes de un sistema de ecuaciones.
- De forma implícita, dando el polinomio en forma no canónica.

7.1. 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:

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} f(x_0) \\ f(x_1) \\ \vdots \\ f(x_n) \end{pmatrix} \tag{7.1}$$

Con igual cantidad de ecuaciones e incógnitas, el sistema de ecuaciones (7.1) tiene solución única si y solo si el determinante es distinto de cero (ver por ejemplo Treil [3]):

$$\begin{vmatrix} 1 & \cdots & x_0^n \\ \vdots & \ddots & \vdots \\ 1 & \cdots & x_n^n \end{vmatrix} \neq 0 \tag{7.2}$$

El determinante de la ecuación (7.2) resulta ser el *determinante de Vandermonde*.

Teorema 7.1. *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. 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: El caso $n = 1$ es trivial, es simplemente:

$$\begin{vmatrix} 1 \end{vmatrix} = 1$$

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 $1 \leq i \leq k$, o sea:

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

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:

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

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

7.2. De forma implícita

Para encontrar la interpolación de f de manera implícita simplemente inventamos un polinomio que pase por los puntos.

7.2.1. Forma de Lagrange

Consiste en usar el polinomio:

$$\begin{aligned} p(x) &= f(x_0)\frac{(x-x_1)(x-x_2)\cdots(x-x_n)}{(x_0-x_1)(x_0-x_2)\cdots(x_0-x_n)} + f(x_1)\frac{(x-x_0)(x-x_2)\cdots(x-x_n)}{(x_1-x_0)(x_1-x_2)\cdots(x_1-x_n)} + \cdots \\ &\quad + f(x_n)\frac{(x-x_0)\cdots(x-x_{n-1})}{(x_n-x_0)\cdots(x_n-x_{n-1})} \\ &= \sum_{0\le k\le n} f(x_k) \prod_{\substack{0\le j\le n \\ j\neq k}} \frac{x-x_j}{x_k-x_j} \end{aligned} \tag{7.3}$$

como interpolación de f. Es claro que si evalúa p en alguno de los x_k con $k \in \{0,1,\ldots,n\}$ que nos entregan, se tiene que $p(x_k) = f(x_k)$, y el polinomio es de grado n.

Para referencia futura, llamaremos:

$$\ell_i(x) = \prod_{\substack{0\le j\le n \\ j\neq i}} \frac{x-x_j}{x_i-x_j} \tag{7.4}$$

Estos polinomios dependen de los puntos x_i, y tienen la particularidad que:

$$\ell_i(x_j) = [i=j]$$

Acá usamos la convención de Iverson:

$$[\text{condición}] = \begin{cases} 0 & \text{si la condición es falsa} \\ 1 & \text{si la condición es verdadera} \end{cases}$$

En la práctica, la fórmula (7.3) 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:

$$p(x) = \frac{\sum_{0\le j\le n} \frac{w_j f(x_j)}{x-x_j}}{\sum_{0\le j\le n} \frac{w_j}{x-x_j}} \tag{7.5}$$

donde:

$$w_j = \left(\prod_{\substack{0\le k\le n \\ k\neq j}} (x_j - x_k) \right)^{-1} \tag{7.6}$$

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

7.2.2. Forma de Newton

Podemos escribir:

$$a_0 = f(x_0) \qquad Q_0(x) = a_0$$

Luego para $k > 0$:

$$a_k = \frac{f(x_k) - Q_{k-1}(x_k)}{\prod_{0\le i\le k-1}(x_k - x_i)} \qquad Q_k(x) = Q_{k-1}(x) + a_k \prod_{0\le i\le k-1}(x - x_i) \tag{7.7}$$

donde Q_k corresponde a la interpolación de grado k.

7.2.2.1. Diferencias divididas

La forma de Newton es engorrosa. Una alternativa se obtiene con *diferencias divididas*, que para los puntos $x_0, \ldots, x_n$ se definen mediante:

$$f[x_0] = f(x_0)$$
$$f[x_0, \ldots, x_j] = \frac{f(x_j) - Q_{j-1}(x_j)}{\prod_{0\le k\le j-1}(x_j - x_k)}$$

Con esta notación (7.7) se escribe:

$$Q_n(x) = f[x_0] + f[x_0, x_1](x - x_0) + \cdots + f[x_0, x_1, \ldots, x_n] \prod_{0\le k<n}(x - x_k) \tag{7.8}$$

La utilidad de la forma (7.8) es por el siguiente resultado:

Lema 7.2. *Las diferencias divididas cumplen:*

$$f[x_0, \ldots, x_n] = \frac{f[x_1, \ldots, x_n] - f[x_0, \ldots, x_{n-1}]}{x_n - x_0} \tag{7.9}$$

Demostración. Para cualquier k, sea Q_k el polinomio de grado a lo más k que interpola f en los puntos $x_0, \ldots, x_k$, o sea:

$$Q_k(x_j) = f(x_j) \quad \text{para } 0 \le j \le k$$

Consideremos el polinomio $P(x)$ único de grado a lo más $n-1$ que interpola f en los puntos $x_1, \ldots, x_n$. Es simple verificar que:

$$Q_n(x) = P(x) + \frac{x - x_n}{x_n - x_0}(P(x) - Q_{n-1}(x)) \tag{7.10}$$

En (7.10) el coeficiente de x^n al lado izquierdo es $f[x_0, \ldots, x_n]$. El coeficiente de x^{n-1} en $P(x)$ es $f[x_1, \ldots, x_n]$, y en $Q_{n-1}(x)$ es $f[x_0, \ldots, x_{n-1}]$. O sea, el coeficiente de x^n al lado derecho de (7.10) es:

$$\frac{f[x_1, \ldots, x_n] - f[x_0, \ldots, x_{n-1}]}{x_n - x_0}$$

como aseveramos. □

x	y	Diferencias divididas				
0,10	0,8000					
		−2,4615				
0,30	0,3077		4,3140			
		−0,9516		−5,6441		
0,45	0,1649		2,0564		4,7074	
		−0,5403		−3,5258		−0,4531
0,50	0,1379		1,1749		4,4355	
		−0,4229		−1,7516		−4,2013
0,55	0,1168		0,7371		1,9148	
		−0,2754		−0,8899		
0,70	0,0755		0,3811			
		−0,1421				
0,90	0,0471					

Cuadro 7.1 – Tabla para interpolación

Nota: Algunos textos definen:

$$f[x_0,\ldots,x_n] = f[x_1,\ldots,x_n] - f[x_0,\ldots,x_{n-1}]$$

en cuyo caso las fórmulas deben ajustarse para compensar por los denominadores faltantes.

Un ejemplo (la función es la de Runge, ecuación (8.4)) es la tabla del cuadro 7.1. Las entradas (diferencias divididas) se obtienen restando los elementos inmediatamente a la izquierda divididas por la diferencia de los valores de x en las diagonales. Para interpolar, se usa la primera fila diagonal.

Ejercicios

1. Demuestre la fórmula baricéntrica (7.5). Defina:

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

   escriba (7.3) como:

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

   Use esta forma de (7.3) 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 (7.3), (7.5) y (7.7). 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. Trefethen: *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/book.pdf, September 2017. Department of Mathematics, Brown University.

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

Clase 8

Error de Interpolación

La clase pasada vimos cómo obtener la interpolación dado los pares de puntos $(x_k, f(x_k))$ con $k \in \{0, \dots, n\}$ que nos daban. Nuestro tema de interés ahora es obtener el error de esas interpolaciones (figura 8.1). En particular, nos interesa ver cómo depende el error de los puntos elegidos. Usaremos algunos resultados del análisis real, refiérase a sus apuntes o a textos como el de Chen [1] para demostraciones de los teoremas del caso.

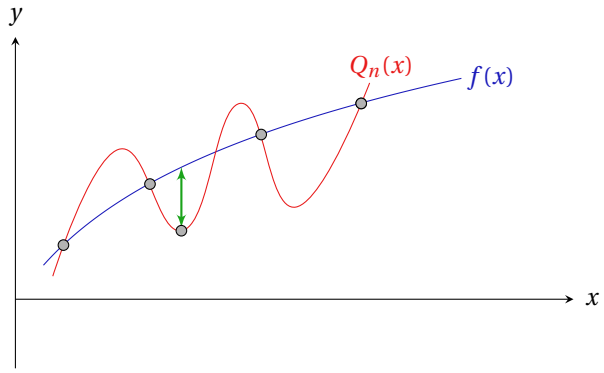


Figura 8.1 – El error es la diferencia entre $Q_n(x)$ y $f(x)$ en cada punto (flecha verde).

Teorema 8.1. *Sea $f \in C^{n+1}[a,b]$ (vale decir, f tiene $n+1$ derivadas continuas en el intervalo). 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*

$$f(x) - Q_n(x) = \underbrace{\frac{1}{(n+1)!} f^{(n+1)}(\zeta) \prod_{0 \le j \le n} (x - x_j)}_{Error} \tag{8.1}$$

Demostración. Fijemos $x \in [a,b]$. Si $x = x_j$ para algún j, el lado izquierdo y derecho de (8.1) se anulan y estamos listos.

En caso contrario, sea

$$\omega(x) = \prod_{0 \le j \le n} (x - x_j) \tag{8.2}$$

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

Definamos:

$$F(y) = f(y) - Q_n(y) - \lambda\omega(y) \tag{8.3}$$

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, en general menor será el error (vea el $(n+1)!$ como denominador). Claro que intervienen las características de la función y los puntos elegidos también.

8.1. El fenómeno de Runge

En 1901 Runge [3] observó el fenómeno que lleva su nombre. Lo ilustramos con la función que el mismo usó de ejemplo:

$$f(x) = \frac{1}{1+25x^2} \tag{8.4}$$

Se grafica (8.4) en el rango $[-1,1]$ en la figura 8.2. Se aprecia que parece ser bastante mansa, pero las figuras 8.3 muestran lo que ocurre al interpolar con 3, 5, 9 y 11 puntos igualmente espaciados entre -1 y 1. Se aprecia que alrededor de 0, el polinomio interpolante se acerca a la función, como esperábamos; sin embargo, en los extremos el error (la diferencia absoluta máxima entre la función y el polinomio) aumenta. La figura 8.4 muestra cómo evoluciona el error máximo con el número de puntos igualmente espaciados. Una explicación clara del fenómeno, aunque un tanto simplificada para efectos didácticos, es la de Epperson [2]. Resulta que depende de las singularidades de la función en el plano complejo, que no se aprecian en la línea real.

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

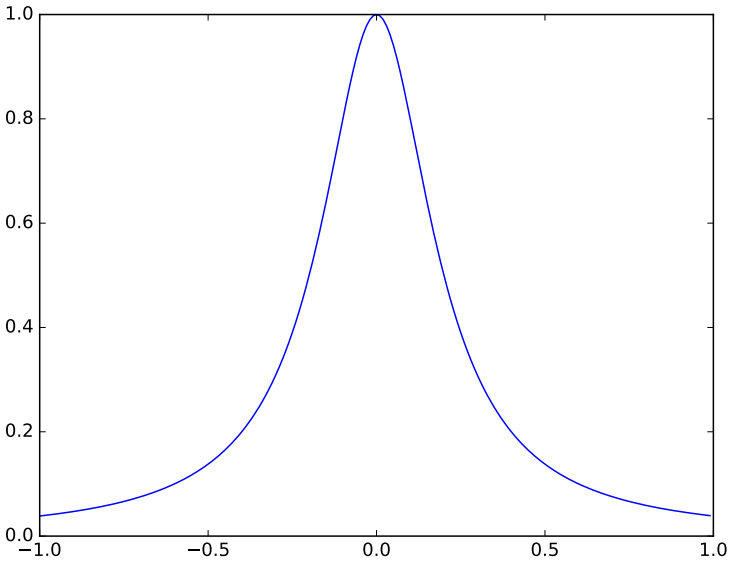


Figura 8.2 – La función de Runge

8.2. Puntos de Chebyshev

Nace entonces la pregunta de elegir los puntos de manera de evitar este comportamiento indeseable. Del teorema 8.1 el error de interpolación para los puntos distintos $x_0, \ldots, x_n \in [a, b]$ cumple:

$$f(x) - Q_n(x) = \frac{1}{(n+1)!} f^{(n+1)}(\zeta) \prod_{0 \le j \le n} (x - x_j) \tag{8.5}$$

donde $a < \zeta < b$. Debe destacarse que ζ depende de los puntos $x_0, \ldots, x_n$, minimizar esta expresión no es fácil. Nos contentaremos con minimizar el polinomio dado por la productoria. Para ello emplearemos una secuencia de polinomios especiales.

Los *polinomios de Chebyshev* se definen mediante:

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_{n+1}(x) &= 2xT_n(x) - T_{n-1}(x) \quad n \ge 1 \end{aligned} \tag{8.6}$$

En vez de la recurrencia (8.6) es posible derivar una fórmula explícita para $T_n(x)$.

Lema 8.2. *Para $x \in [-1, 1]$:*

$$T_n(x) = \cos(n \arccos x) \tag{8.7}$$

Demostración. Identidades trigonométricas simples implican que:

$$\begin{aligned} \cos(n+1)\theta &= \cos\theta \cos n\theta - \sin\theta \sin n\theta \\ \cos(n-1)\theta &= \cos\theta \cos n\theta + \sin\theta \sin n\theta \end{aligned}$$

de donde:

$$\cos(n+1)\theta = 2\cos\theta\cos n\theta - \cos(n-1)\theta$$

Sea $\theta = \arccos x$, vale decir, $x = \cos\theta$. Si definimos:

$$t_n(x) = \cos(n \arccos x) = \cos n\theta$$

Resulta así:

$$\begin{aligned} t_0(x) &= 1 \\ t_1(x) &= x \\ t_{n+1}(x) &= 2x\,t_n(x) - t_{n-1}(x) \quad n \geq 1 \end{aligned}$$

Esta es exactamente la recurrencia (8.6). □

De la ecuación (8.6) vemos que:

$$T_n(x) = 2^{n-1}x^n + \cdots$$

Conocemos los ceros de $T_n(x)$ por el lema 8.2:

$$x_k = \cos\frac{(2k-1)\pi}{2n}, k = 1, \ldots, n$$

así que:

$$T_n(x) = 2^{n-1}\prod_{1\leq k\leq n}(x - x_k)$$

¿Qué tienen de especial estos polinomios? Veremos primero un resultado general sobre polinomios mónicos (con coeficiente uno en el término de mayor grado).

Teorema 8.3. *Si $p_n(x)$ es un polinomio mónico de grado n, entonces:*

$$\max_{-1\leq x\leq 1}|p_n(x)| \geq 2^{1-n} \tag{8.8}$$

Demostración. Demostramos (8.8) por contradicción. Supongamos que para para el polinomio p_n de grado n y para todo $|x| \leq 1$ siempre es:

$$|p_n(x)| < 2^{1-n}$$

Sea

$$q_n(x) = 2^{1-n}T_n(x)$$

con lo que q_n es mónico, y sean u_j los siguientes $n+1$ puntos:

$$u_j = \cos\frac{j\pi}{n} \quad 0 \leq j \leq n$$

Como:

$$T_n\left(\cos\frac{j\pi}{n}\right) = (-1)^j$$

tenemos que:

$$(-1)^j q_n(u_j) = 2^{1-n}$$

Por lo tanto:

$$(-1)^j p_n(u_j) \leq |p_n(u_j)| < 2^{1-n} = (-1)^j q_m(u_j)$$

Concluimos que:

$$(-1)^j (q_n(u_j) - p_n(u_j)) > 0 \quad 0 \leq j \leq n$$

lo que significa que el polinomio $q_n(x) - p_n(x)$ oscila $n+1$ veces en el intervalo. Vale decir, tiene al menos n ceros en el intervalo. Pero p_n y q_n son mónicos, por lo que $q_n(x) - p_n(x)$ tiene grado a lo más $n-1$, y no puede tener más de $n-1$ ceros a menos que sea el polinomio cero. Como el máximo de p_n es menor al máximo de q_n, esto es imposible. □

Volvamos a nuestro problema de interpolación. Queremos hallar los puntos x_j que minimicen:

$$\max_{-1 \leq x \leq 1} \left| \prod_{0 \leq j \leq n} (x - x_j) \right|$$

Notamos que este polinomio es mónico de grado $n+1$, por lo que por el teorema 8.3:

$$\max_{-1 \leq x \leq 1} \left| \prod_{0 \leq j \leq n} (x - x_j) \right| \geq 2^{-n}$$

y podemos obtener el mínimo de esto si el polinomio es $2^{-n} T_{n+1}(x)$, o sea, los x_j son los ceros del polinomio de Chebyshev $T_{n+1}(x)$. Por el lema 8.2 vemos que:

$$x_j = \cos \frac{2j+1}{2n+2} \pi \quad 0 \leq j \leq n \tag{8.9}$$

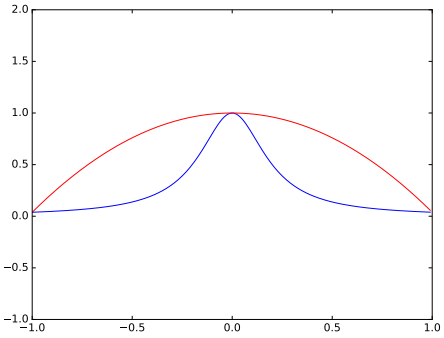
Para estos puntos nuestra cota (8.1) para el error da:

$$\begin{aligned} |f(x) - p_n(x)| &= \left| 2^{-n} T_{n+1}(x) \frac{f^{(n+1)}(\zeta(x))}{(n+1)!} \right| \\ &\leq \frac{B_{n+1}}{2^n (n+1)!} \end{aligned} \tag{8.10}$$

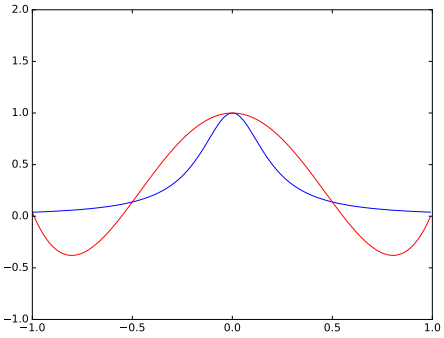
donde:

$$B_{n+1} = \max_{x \in [-1,1]} |f^{(n+1)}(x)| \tag{8.11}$$

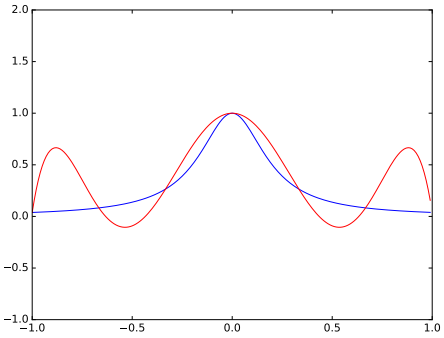
Si repetimos el ejercicio de interpolación anterior, pero ahora interpolando la función de Runge en los puntos de Chebyshev resultan las figuras 8.5. La figura 8.6 muestra cómo evoluciona el error máximo con el número de puntos de Chebyshev.



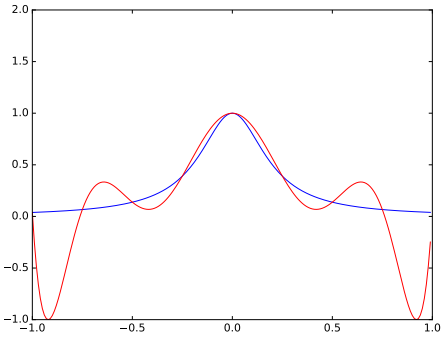
(a) 3 puntos



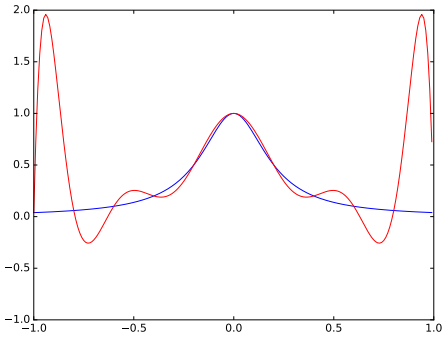
(b) 5 puntos



(c) 7 puntos



(d) 9 puntos



(e) 11 puntos

Figura 8.3 – Interpolando la función de Runge, puntos equiespaciados

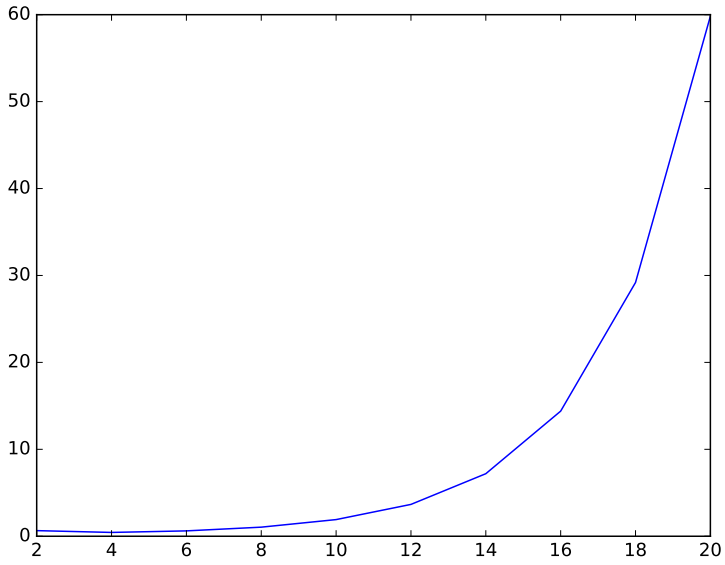
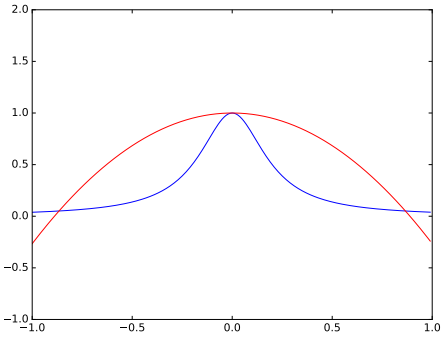
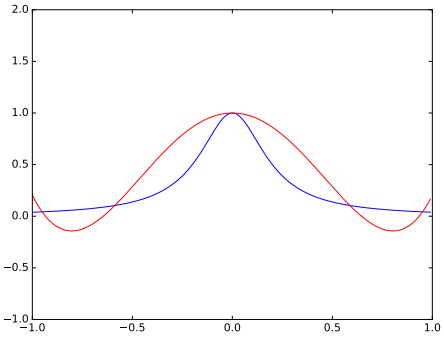


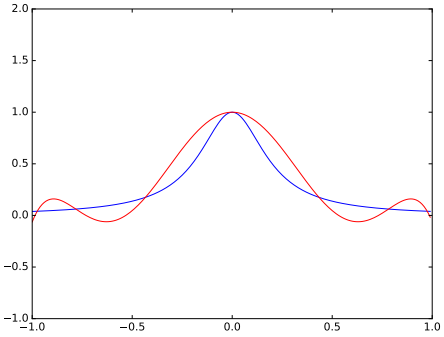
Figura 8.4 – Error al interpolar la función de Runge



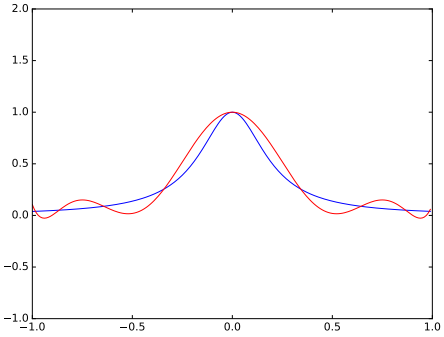
(a) 3 puntos



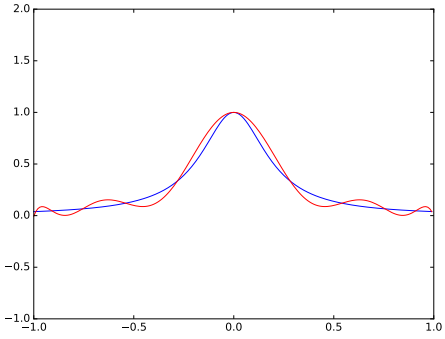
(b) 5 puntos



(c) 7 puntos



(d) 9 puntos



(e) 11 puntos

Figura 8.5 – Interpolando la función de Runge, puntos de Chebyshev

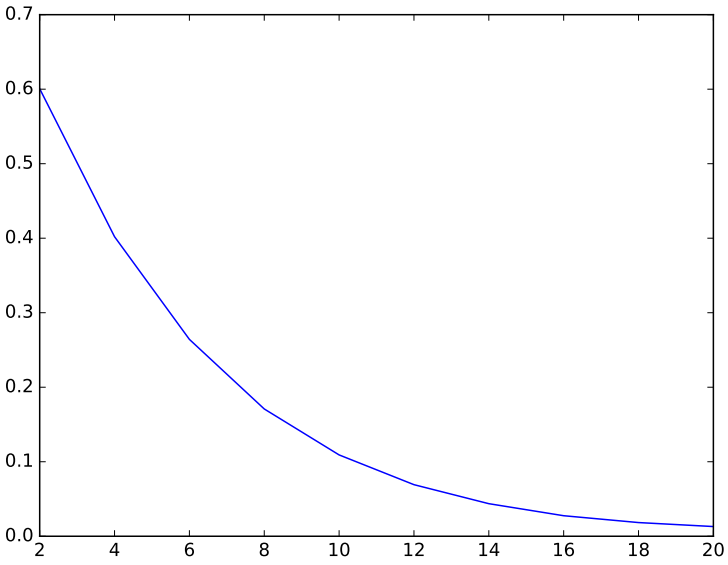


Figura 8.6 – Error al interpolar la función de Runge en puntos de Chebyshev

Bibliografía

[1] William W. L. Chen: *Fundamentals of analysis*. http://rutherglen.science.mq.edu.au/wchen/lnfafolder/lnfa.html, 2008. Department of Mathematics, Macquarie University.

[2] James F. Epperson: *On the Runge example*. The American Mathematical Monthly, 94(4):329–341, April 1987.

[3] Carl Runge: *Über empirische Funktionen und die Interpolation zwischen äquidistanten Ordinaten*. Zeitschrift für Mathematik und Physik, 46:224–243, 1901.

Clase 9

Cuadratura

Interesa desarrollar técnicas numéricas para calcular integrales. La triste realidad es que, a pesar del entusiasmo de los colegas de matemáticas por técnicas de integración, la minoría de las integrales que debemos calcular en la práctica tienen una forma cerrada, e incluso si la tienen puede ser poco manejable.

Queremos evaluar:

$$\int_a^b f(x)\mathrm{d}x \tag{9.1}$$

Supongamos f dado en $x_0, \ldots, x_n$ (los *puntos de cuadratura*). Para encontrar el valor de (9.1) simplemente interpolamos f, e integramos el polinomio interpolante.

9.1. Caso más simple: Polinomio de grado 0

Corresponde a una aproximación con rectángulos (figura 9.1) Para ello, usábamos la fórmula:

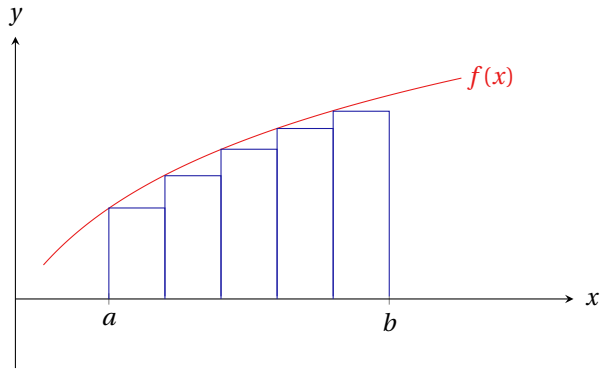


Figura 9.1 – Aproximar el área bajo una curva usando rectangulitos (integral de Riemann).

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

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

$$\begin{aligned}\int_a^b f(x)\mathrm{d}x &\approx h \sum_{0\le j<n} f(x_j) \\ &= h \sum_{0\le j<n} f(x_0 + jh)\end{aligned}$$

Consideremos la antiderivada[1]:

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

Expandiendo $F(x)$ en serie de Taylor alrededor de a y usando el teorema fundamental del cálculo:

$$F(x) = F(a) + F'(a)(x-a) + \frac{1}{2!}F''(\xi)(x-a)^2 \qquad = 0 + f(a)(x-a) + \frac{1}{2!}f'(\xi)(x-a)^2$$

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 para cada intervalo, si son n intervalos es:

$$\begin{aligned}nO(h^2) &= O\left(n\frac{(b-a)^2}{n^2}\right) \\ &= O\left(\frac{(b-a)^2}{n}\right) \\ &= O(h)\end{aligned}$$

La misma observación es válida para otras reglas compuestas.

9.2. Variante del caso simple: punto medio

En lugar de considerar una cota como se hizo en el caso de la figura 9.1, el punto de evaluación de $f(x)$ será el punto medio de la base del rectángulo (figura 9.2). Suponemos que f es integrable, y que además tiene «suficientes» derivadas continuas.

Para este caso se tiene que:

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

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

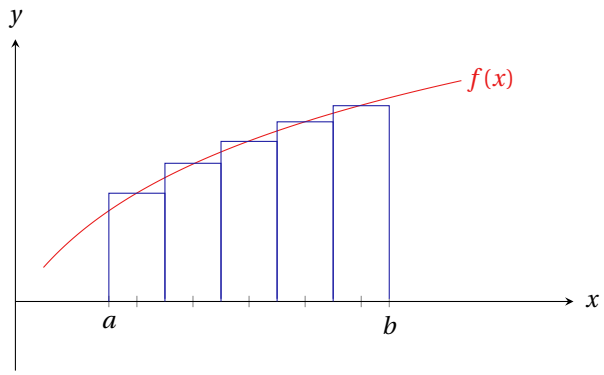


Figura 9.2 – El excedente de «triangulitos» por sobre f compensan la falta de estos que están bajo f.

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:

$$\begin{aligned} E &= \int_a^{a+h} f(x)\mathrm{d}x - hf\left(a+\frac{h}{2}\right) \\ &= hf(a) + \frac{h^2}{2}f'(a) + \frac{h^3}{6}f''(a) + O(h^4) - h\left(f(a) + \frac{h}{2}f'(a) + \frac{h^2}{8}f''(a) + O(h^3)\right) \\ &= hf(a) + \frac{h^2}{2}f'(a) + \frac{h^3}{6}f''(a) + O(h^4) - \left(hf(a) + \frac{h^2}{2}f'(a) + \frac{h^3}{8}f''(a) + O(h^4)\right) \\ &= \frac{1}{24}f''(a)h^3 + O(h^4) \end{aligned} \tag{9.2}$$

Sorprendentemente, el error es de carácter cúbico, cabía esperar un error cuadrático como en el caso anterior (sección 9.1). Si tenemos una cota para la segunda derivada de f en a, estamos listos.

9.2.1. Teorema del Valor Intermedio Ad Hoc

Considerando el error obtenido en la ecuación (9.2):

$$E = \frac{1}{24}f''(\zeta)h^3 \qquad \text{con } a \le \zeta \le a+h \tag{9.3}$$

Suponiendo intervalos $a = x_0, x_1, \ldots, x_n = b$ 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:

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

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

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

9.3. 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:

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

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:

$$E = \frac{b-a}{2}(f(a)+f(b)) - \int_a^b f(x)\mathrm{d}x \tag{9.4}$$

Sea c el centro del intervalo:

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

con lo que:

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

De (9.4) 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 $a \leq x \leq b$ tenemos la cota:

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

vemos que:

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

Sorprendentemente, el error es cúbico.

9.4. 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 $w_{-1} = w_1 = 1/3$, $w_0 = 4/3$.

Por la simetría del sistema de ecuaciones subyacente, si tenemos $2n+1$ puntos $-n, \ldots, n$ es claro que debe ser $A_{-k} = A_k$ y tendremos:

$$\begin{aligned} \sum_{-n\leq k\leq n} A_k k^{2n+1} &= A_0 0^{2n+1} + 2\sum_{1\leq k\leq n} A_k\left((-k)^{2n+1} + k^{2n+1}\right) \\ &= 0 \end{aligned}$$

Vale decir, si el número n de puntos igualmente espaciados es impar (en realidad, basta que estén están distribuidos simétricamente alrededor del 0), la regla de cuadratura es exacta hasta para polinomios de grado n. No ocurre lo mismo si n es par, en cuyo caso es exacta para polinomios hasta grado $n-1$, como plantea el sistema de ecuaciones, pero para grado n (que es par) falla.

Tenemos el siguiente resultado para la regla de Simpson:

Teorema 9.1. *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:

$$\int_{x_0}^{x_2} f(x)\mathrm{d}x = h\int_{-1}^{1} F(\tau)\mathrm{d}\tau$$

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 $H'''(\xi_3) = 0$. Note que la tercera derivada de G es:

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

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 $2\xi_3^2 = \xi_3^2 - (-\xi_3^2)$, 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 intermedio 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 $\xi = x_1 + h\xi_4 \in (a,b)$. □

Esto explica el h^5 mágico que empleamos antes: sabíamos que el error resultaría $O(h^4)$, necesitábamos uno más.

9.5. Fórmula para los coeficientes

Si tenemos puntos $x_0 < x_1 < \ldots < x_n$, sabemos que los polinomios (7.4) cumplen:

$$\ell_k(x) = \prod_{\substack{0\le j\le n\\ j\ne k}} \frac{x - x_j}{x_k - x_j} \tag{9.5}$$

$$\ell_k(x_j) = [j = k] \tag{9.6}$$

Es claro que son $n+1$ polinomios linealmente independientes, por lo que forman una base para Π_n. En consecuencia, si la fórmula:

$$\int_{x_0}^{x_n} f(x)\,\mathrm{d}x = \sum_{0\le j\le n} A_j f(x_j) \tag{9.7}$$

es exacta para todo Π_n, es exacta para los polinomios ℓ_k, lo que da:

$$\begin{aligned}\int_{x_0}^{x_n} \ell_k(x)\,\mathrm{d}x &= \sum_{0\le j\le n} A_j \ell_k(x_j)\\ &= A_k\end{aligned} \tag{9.8}$$

Esto da una fórmula general para los A_k.

9.6. Fórmulas de mayor grado

Vimos el fenómeno de Runge, usar polinomios interpolantes de alto grado con puntos igualmente espaciados puede llevar al desastre. Por esa razón en la práctica se usan fórmulas compuestas con pocos puntos, como la regla de Simpson. Una opción para disminuir el error es usar puntos de Chebyshev, pero eso complica las fórmulas. No nos detendremos en esto.

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).

3. Hemos analizado el caso de un único intervalo, en la práctica un intervalo mayor se subdividirá. Deduzca las fórmulas para el caso de n subintervalos iguales, y exprese el error para el intervalo completo.

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 10

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* (los llamaremos $x_0, x_1, \ldots, x_n$). 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 gaussianas*.

Ejemplo 10.1 (Cuadratura gaussiana con dos puntos)**.** Supongamos que queremos encontrar dos puntos de cuadratura de la ecuación:

$$\int_{-1}^{1} f(x)\mathrm{d}x \approx a_0 f(x_0) + a_1 f(x_1) \tag{10.1}$$

Como queremos encontrar los valores de a_0, a_1, x_0 y x_1, esperamos que la ecuación (10.1) 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:

$$\int_{-1}^{1} x^k \mathrm{d}x = a_0 x_0^k + a_1 x_1^k \qquad k \in \{0,1,2,3\} \tag{10.2}$$

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 (10.2):

$$\begin{aligned} 2 &= a_0 + a_1 \\ 0 &= a_0 x_0 + a_1 x_1 \\ \frac{2}{3} &= a_0 x_0^2 + a_1 x_1^2 \\ 0 &= a_0 x_0^3 + a_1 x_1^3 \end{aligned} \tag{10.3}$$

Al resolver el sistema de ecuaciones (10.3) 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 extensible 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 10.2 (Cuadratura gaussiana con tres puntos)**.** 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 10.1, podemos resumir el sistema de ecuaciones generado a través del cuadro 10.1. 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	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	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 10.1 – Comprobamos nuestras sospechas.

$$\frac{2}{3} = 2a_0 x_2^2$$
$$x_2^2 = \frac{1}{3a_0} \tag{10.4}$$

Continuamos con $k = 4$:

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

Luego, reemplazamos a_0 en (10.4):

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

Además:

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

Basta reemplazar en el sistema de ecuaciones para comprobar que esto se cumple.

10.1. 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]. La teoría requerida sobre espacios normados se reseña en el apéndice E.

10.1.1. 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 10.1. 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 $\|f\|_w = 1$

Es claro de la definición que la norma nunca es negativa. Comúnmente omitimos el subíndice, la función peso se subentiende.

Definición 10.2. 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], o refiérase al apéndice E) permite construir un conjunto ortogonal.

Teorema 10.1. *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. □

10.1.2. Cuadratura de Gauß

Buscamos reglas de cuadratura de la forma:

$$\int_a^b w(x) f(x) \mathrm{d}x = \sum_{0 \leq i \leq n} A_i f(x_i) \tag{10.5}$$

La ecuación (10.5) es exacta para $f \in \Pi_n$ si y solo si (esto es básicamente por la forma de Lagrange del polinomio interpolador):

$$A_i = \int_a^b w(x) \prod_{\substack{0 \leq j \leq n \\ j \neq i}} \frac{x - x_j}{x_i - x_j} \mathrm{d}x \tag{10.6}$$

En (10.5) 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 10.2. *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 (10.5) con los coeficientes (10.6) 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 10.3. *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 (con esto $\deg(q) = n+1$), donde $q(x_i) = 0$ en los puntos de cuadratura $\{x_i\}_{0\leq i\leq n}$:

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

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, $\deg(p) \leq n$, y $\deg(p^2) \leq 2n$, 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 que $\langle f, g \rangle_w$ es un producto interno.
2. Demostrar que los coeficientes A_i están dados por la ecuación (10.6).

Bibliografía

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

[2] Sergei Treil: *Linear algebra done wrong*. https://www.math.brown.edu/~treil/papers/LADW/book.pdf, September 2017. Department of Mathematics, Brown University.

Clase 11

Más sobre Cuadratura Gaussiana

Interesan métodos de cálculo mejores para los polinomios ortogonales, y derivar fórmulas para el error de reglas de cuadratura gaussianas. Antes de entrar en el tema, daremos un desvío.

11.1. Interpolación de Hermite

Un problema de interpolación interesante se presenta si se dan los valores de la función y adicionalmente se especifican valores para derivadas. Obtener un polinomio que cumple estas condiciones se conoce como *interpolación de Hermite*. Mayores detalles se hallan por ejemplo en el texto de Levy [1], acá requeriremos solo un caso muy particular.

Supongamos que tenemos los valores de las función f y los de su primera derivada en los puntos x_i para $1 \le i \le n$. Buscamos un polinomio que coincida con ellos. Claramente, tal polinomio será de grado $2n+1$ (tenemos $2n+2$ condiciones). Podemos escribirlo:

$$p(x) = \sum_{0 \le i \le n} f(x_i) A_i(x) + \sum_{0 \le i \le n} f'(x_i) B_i(x) \tag{11.1}$$

donde requerimos:

$$\begin{array}{ll} A_i(x_j) = [i = j] & A_i'(x_j) = 0 \\ B_i(x_j) = 0 & B_i'(x_j) = [i = j] \end{array} \tag{11.2}$$

Notamos que los polinomios l_i definidos por (7.4) para los puntos $x_0, \ldots, x_n$ cumplen:

$$l_i(x_j) = [i = j] \quad \text{para } 0 \le j \le n$$

El cuadrado de l_i tiene ceros dobles en x_j para $j \neq i$:

$$l_i^2(x_j) = 0 \quad \left(l_i^2(x)\right)' = 0$$

Como l_i es de grado n, el grado de l_i^2 es $2n$. Esto sugiere que los polinomios A_i y B_i pueden escribirse:

$$A_i(x) = r_i(x) l_i^2(x)$$
$$B_i(x) = s_i(x) l_i^2(x)$$

con polinomios lineales r_i, s_i. Ahora bien, por (11.2):

$$[i=j] = A_i(x_j) = r_i(x_j) l_i^2(x_j) = r_i(x_j)[i=j]$$

por lo que $r_i(x_i) = 1$. Por el otro lado requerimos:

$$0 = A_i'(x_j) = r_i'(x_j) l_i^2(x_j) + 2 r_i(x_j) l_i(x) l_i'(x) = r_i'(x_j)[i=j] + 2 r_i(x_j)[i=j] l_i'(x_j)$$

Esto se resume en:

$$r_i'(x_i) + 2 l_i'(x_i) = 0$$

Suponiendo r_i lineal, resulta:

$$r_i(x) = -2 l_i'(x_i) x + (1 + 2 l_i'(x_i) x_i)$$

que lleva a:

$$A_i(x) = \left(1 - 2 l_i'(x_i)(x - x_i)\right) l_i^2(x)$$

De forma similar, de (11.2):

$$\begin{aligned} 0 = B_i(x_j) &= s_i(x_j) l_i^2(x_j) \\ [i=j] = B_i'(x_j) &= s_i'(x_j) l_i^2(x_j) + 2 s_i(x_j) \left(l_i^2(x_j)\right)' \end{aligned}$$

Concluimos que:

$$s_i(x) = x - x_i$$

con lo que:

$$B_i(x) = (x - x_i) l_i^2(x)$$

y finalmente el polinomio interpolador de Hermite toma la forma:

$$p(x) = \sum_{0 \le i \le n} \left(1 - 2 l_i'(x_i)(x - x_i)\right) l_i^2(x) f(x_i) + \sum_{0 \le i \le n} (x - x_i) l_i^2(x) f'(x_i) \tag{11.3}$$

Para el error en (11.3) tenemos:

Teorema 11.1. *Sean $x_0, \ldots, x_n$ puntos distintos en $[a,b]$ y $f \in C^{2n+2}[a,b]$. Si $p \in \Pi_{2n+1}$ es tal que para $0 \le i \le n$:*

$$p(x_i) = f(x_i) \qquad p'(x_i) = f'(x_i)$$

entonces hay $\xi \in (a,b)$ tal que:

$$f(x) - p(x) = \frac{f^{(2n+2)}(\xi)}{(2n+2)!} \prod_{0 \le i \le n} (x - x_i)^2 \tag{11.4}$$

Demostración. La técnica es similar a la empleada para demostrar el teorema 8.1. Sea x un punto fijo en $[a,b]$, distinto de los x_i, y definimos:

$$w(y) = \prod_{0 \le i \le n} (y - x_i)^2$$

Definimos también:

$$\phi(y) = f(y) - p(y) - \lambda w(y)$$

donde elegimos λ de manera que $\phi(x) = 0$, o sea:

$$\lambda = \frac{f(x) - p(x)}{w(x)}$$

Entonces ϕ tiene al menos $n+2$ ceros en $[a,b]$, a saber $x, x_0, \ldots, x_n$. Por el teorema de Rolle, ϕ' tiene (al menos) $n+1$ ceros distintos de los anteriores, y sabemos que ϕ' se anula en $x_0, \ldots, x_n$, con lo que ϕ' tiene al menos $2n+2$ ceros en (a,b). Finalmente $\phi^{(2n+2)}$ tiene al menos un cero en (a,b), llamémosle ξ:

$$0 = \phi^{(2n+2)}(\xi) = f^{(2n+2)}(\xi) - p^{(2n+2)}(\xi) - \lambda w^{(2n+2)}(\xi)$$

Como p es un polinomio de grado $2n+1$ y w un polinomio mónico de grado $2n+2$, resulta lo indicado. □

Estamos en condiciones de obtener el error de la regla de cuadratura gaussiana.

Teorema 11.2. *Sea $f \in C^{2n+2}[a,b]$ y sea w una función de peso en ese intervalo. Considere la cuadratura gaussiana:*

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

Entonces existe $\zeta \in (a,b)$ tal que

$$\int_a^b w(x) f(x) \mathrm{d}x - \sum_{0 \le i \le n} A_i f(x_i) = \frac{f^{(2n+2)}(\zeta)}{(2n+2)!} \int_a^b w(x) \prod_{0 \le i \le n} (x - x_i)^2 \, \mathrm{d}x$$

Demostración. Consideremos interpolación de Hermite en puntos $x_0, \ldots, x_n$, cuyo error por el teorema 11.1 para algún $\xi \in (a,b)$ cumple:

$$f(x) - p(x) = \frac{f^{(2n+2)}(\xi)}{(2n+2)!} \prod_{0 \le i \le n} (x - x_i)^2$$

Tenemos la fórmula (11.3) para p:

$$p(x) = \sum_{0 \le i \le n} \left(1 - 2l_i'(x_i)(x - x_i)\right) l_i^2(x) f(x_i) + \sum_{0 \le i \le n} (x - x_i) l_i^2(x) f'(x_i) \tag{11.5}$$

Vemos que:

$$(x - x_i) l_i^2(x) = l_i(x) \prod_{0 \le k \le n} (x - x_k)$$

Para una cuadratura gaussiana $x_0, \ldots, x_n$ son los ceros del polinomio ortogonal de grado $n+1$, el producto es un múltiplo de ese polinomio, ortogonal a l_i de grado n. Las integrales de los términos que involucran las derivadas $f'(x_i)$ se anulan.

En consecuencia, tenemos (recuerde que ξ depende de x):

$$\begin{aligned} \int_a^b w(x) f(x) \, \mathrm{d}x - \sum_{0 \le i \le n} A_i f(x_i) &= \int_a^b w(x) f(x) \, \mathrm{d}x - \int_a^b w(x) p(x) \, \mathrm{d}x \\ &= \int_a^b w(x) \frac{f^{(2n+2)}(\xi)}{(2n+2)!} \prod_{0 \le i \le n} (x - x_i)^2 \, \mathrm{d}x \end{aligned}$$

Por el teorema del valor intermedio de la integral hay ζ tal que la última integral puede expresarse:

$$\frac{f^{(2n+2)}(\zeta)}{(2n+2)!}\int_a^b w(x)\prod_{0\le i\le n}(x-x_i)^2\,\mathrm{d}x$$

que es lo que asevera el teorema. □

11.2. La recurrencia para polinomios ortogonales

Los polinomios ortogonales tienen bastantes propiedades notables. Comentaremos una de ellas.

Teorema 11.3. *La secuencia de polinomios ortogonales P_n cumple una recurrencia de la forma:*

$$P_{n+1}(x) = (A_n x + B_n)P_n(x) + C_n P_{n-1}(x) \tag{11.6}$$

Si $P_n(x) = a_n x^n + b_n x^{n-1} + \cdots$ entonces:

$$A_n = \frac{a_{n+1}}{a_n} \quad B_n = \frac{a_{n+1}}{a_n}\left(\frac{b_{n+1}}{a_{n+1}} - \frac{b_n}{a_n}\right) \quad C_n = \frac{a_{n+1}a_{n-1}}{a_n^2} \tag{11.7}$$

Demostración. Es claro que el polinomio:

$$Q_n(x) = P_{n+1}(x) - A_n x P_n(x)$$

es de grado a lo más n, por lo que podemos escribirlo como:

$$Q_n(x) = \sum_{0\le i\le n}\alpha_i P_i(x)$$

Los coeficientes pueden expresarse usando el producto interno:

$$\alpha_i = \frac{\langle Q_n, P_i\rangle}{\langle P_i, P_i\rangle} = \frac{\langle P_{n+1} - A_n x P_n, P_i\rangle}{\langle P_i, P_i\rangle} = \frac{\langle P_{n+1}, P_i\rangle - \langle A_n x P_n, P_i\rangle}{\langle P_i, P_i\rangle} = -A_n\frac{\langle xP_n, P_i\rangle}{\langle P_i, P_i\rangle}$$

Por la definición del producto interno:

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

vemos que:

$$\langle xP_n, P_i\rangle = \langle P_n, xP_i\rangle$$

y como P_n es ortogonal a todo Π_{n-1}:

$$\langle P_n, xP_i\rangle = 0 \qquad 0 \le i \le n-2$$

O sea:

$$P_{n+1}(x) = A_n x P_n(x) + \alpha_n P_n + \alpha_{n-1}P_{n-1}(x)$$

Reordenando y comparando coeficientes de x^n y x^{n-1} a ambos lados completa los valores indicados para B_n y C_n citados. □

Bibliografía

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

Parte II

Algoritmos Combinatorios

Clase 12

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.

12.1. 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 12.1. Un conjunto de matrimonios se dice *inestable* si incluye parejas uv y xy, tales que u prefiere a y frente a v y y prefiere a u frente a x.

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. Es claro no todos pueden obtener su mejor opción, si hay un hombre que es el preferido de dos o más mujeres, una de ellas al menos deberá conformarse con otro.

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 12.1, donde simplemente damos las mujeres en orden de preferencia para cada hombre y viceversa. Una secuencia de divorcios 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 12.1. *Siempre hay un conjunto de matrimonios estable.*

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 12.1 – Contraejemplo para divorcios y matrimonios

Demostración. Consideremos el modelo tradicional, en el cual los hombres proponen matrimonio a las mujeres, y estas aceptan o no. Efectuamos varias rondas, en cada ronda los hombres sin pareja proponen matrimonio a la mujer más alta en su preferencia que no lo ha rechazado aún, 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 noviazgo 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', quien a su vez prefiere a x sobre su marido x'. 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 x', 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 esta 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 12.2. Vemos que el hombre 1 puede formar parejas estables

1: 1 2	1: ⋯ 1
2: 2 1	2: ⋯ 2
⋮ ⋮ ⋮	⋮ ⋮
$n-1$: $n-1$ n	$n-1$: ⋯ $n-1$
n: n $n-1$	n: ⋯ n
(a) Hombres	(b) Mujeres

Cuadro 12.2 – Preferencias para muchas soluciones, n par

con 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 12.3 la

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

(a) Hombres (b) Mujeres

Cuadro 12.3 – Preferencias incompletas

única posibilidad es $((x_1, y_1), (x_2, y_2), (x_3, y_3))$, pero esta es inestable por x_2 e y_3.

En realidad, se da:

Teorema 12.2. *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; si siempre se cumple durante la ejecución del algoritmo se cumple al final, y cada hombre se casa con su mejor pareja posible. 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.

12.2. 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 12.1 se demuestra que el resultado es estable (ningún estudiante se cambiaría de universidad con otro), como en el teorema 12.2 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 12.4.

2. Acote el número de propuestas de matrimonio, como menciona la demostración del teorema 12.2.

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 12.4 – Preferencias para ejercicio

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 12.1 no especifica el orden en que los hombres se proponen. Demuestre que cualquiera sea este orden, la asignación resultante es la misma.

5. Demuestre que el algoritmo como esbozado en teorema 12.1 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 13

Algoritmos voraces

Muchos problemas involucran optimización combinatoria: buscamos una configuración óptima de algún objeto discreto. En este capítulo plantearemos una técnica general simple y muy atractiva, elegir «la mejor opción local» y nunca reconsiderar elecciones previas. Esto se conoce como *algoritmos voraces* (en inglés, *greedy algorithms*; una traducción más precisa sería *algoritmos ávidos* o *algoritmos codiciosos*, pero no suena tan bien).

Un ejemplo es un problema de programación de tareas (*scheduling* en inglés).

Ejemplo 13.1. 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 traslapar.

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 13.1). Básicamente, estamos suponiendo que el observatorio se arrienda por proyec-

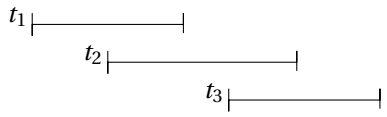


Figura 13.1 – Intervalos de tiempo que dura cada proyecto/tarea.

to y no por tiempo. ¿Cómo hacerlo? Algunas posibilidades:

Sugerencia 1: Repetidamente elegir la tarea más corta que no entra en conflicto. La figura 13.2 muestra un contraejemplo: elegiríamos la tarea t_2, dejando fuera (por conflictos) a t_1 y t_3. La

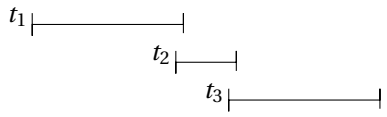


Figura 13.2 – Empezando con t_2, efectuamos 1 tarea. Con t_1 completamos 2.

solución óptima es elegir t_1, t_3. 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 13.3 muestra

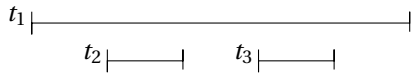


Figura 13.3 – Empezando con t_1, completa 1 tarea. Con t_2 y t_3 hacemos 2.

un contraejemplo: elegiríamos la tarea t_1, sin embargo, el óptimo es t_2, t_3. Nuevamente, 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 13.4 muestra un contraejemplo. El óptimo es

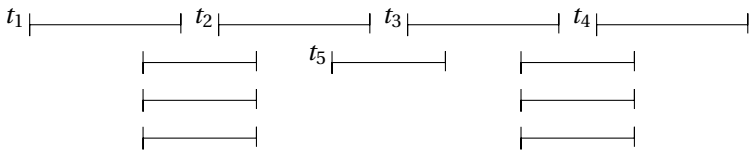


Figura 13.4 – Elije t_5, t_1 y t_4, un total de 3 tareas; pero t_1, t_2, t_3, t_4 son 4. Me echaron a perder el día.

cuatro tareas, t_1, t_2, t_3, t_4; esta estrategia elige tres tareas (t_5 y dos que no interfieren con ella).

¿Estructura común de las propuestas?

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

Es importante destacar que no siempre hay un algoritmo voraz que encuentra una solución óptima del problema, pero sí pueden ofrecer una aproximación bastante buena.

13.1. Comprobar que un algoritmo da un óptimo

Volvamos al ejemplo 13.1. En ese caso, un criterio 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?

Llamemos P a un conjunto de tareas, una solución Π es un subconjunto de P, la solución Π es viable si no incluye tareas que se traslapan. Buscamos Π viable de tamaño máximo. Llamaremos p a una tarea.

Una forma de comprobar que el algoritmo voraz retorna un óptimo es demostrar las siguientes propiedades:

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

Estructura Inductiva (*inductive structure*): 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», parte de lo que se hace al definir P' es precisamente asegurar esto).

Subestructura Óptima (*optimal substructure*): 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 13.1. *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 $|P| = 1$, 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 $\hat{p}$ por el criterio voraz, sabemos que hay una solución óptima que incluye $\hat{p}$ por Elección Voraz. Sea P' el problema que resulta al eliminar $\hat{p}$ 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, $\Pi' \cup \{\hat{p}\}$ es viable para P; por Subestructura Óptima, $\Pi' \cup \{\hat{p}\}$ es óptima para P.

□

13.1.1. Demostrando que un algoritmo voraz da un óptimo

Volvamos al ejemplo 13.1. Si quisiéramos demostrar que la sugerencia entrega un óptimo solo tenemos que demostrar que cumple las propiedades enunciadas:

Elección Voraz: Hay una solución óptima que incluye la elección voraz $\hat{p}$.

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

Estructura Inductiva: El elegir $\hat{p}$ nos deja un problema P' sin restricciones externas.

Demostración. El problema $P \setminus \{\hat{p}\}$ incluye tareas en conflicto con $\hat{p}$, hay soluciones viables para este que no son viables con $\hat{p}$. Hay restricciones externas.

Eliminar también las tareas con conflictos con $\hat{p}$ deja un problema P' sin restricciones externas. Toda solución viable para el problema resultante puede combinarse con $\hat{p}$. □

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

Demostración. Sea Π' como dado. Entonces $\Pi' \cup \{\hat{p}\}$ es viable para P (por Estructura Inductiva), y $|\Pi' \cup \{\hat{p}\}| = |\Pi'| + 1$. Sea Π^* una solución óptima para P que contiene $\hat{p}$ (existe por Elección Voraz). Entonces $\Pi^* \setminus \{\hat{p}\}$ es una solución óptima para P' (si hubiese una mayor, combinada con $\hat{p}$ daría una solución mayor que Π^* para P). Pero entonces:

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

O sea:

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

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

13.2. Problema de Asignación de Tareas

Demostraremos formalmente que nuestro algoritmo voraz entrega una solución óptima al problema de asignación de tareas, usando las tres propiedades (Elección Voraz, Estructura Inductiva y Subestructura Óptima).

Teorema 13.2. *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 $\widehat{p}$ según criterio voraz. Por Elección Voraz hay una solución óptima que lo incluye; al eliminar la tarea $\widehat{p}$ con las tareas con las que interfiere queda un problema P', claramente $|P'| \leq k$.

Por inducción, obtengo una solución óptima Π' de P'. Por Estructura Inductiva Π' junto a $\widehat{p}$ es una solución viable para P Esta es una solución óptima para P por Subestructura Óptima.

□

La demostración no depende realmente de este problema, lo que demuestra que siempre que se cumplan las tres propiedades obtendremos una solución óptima. De ahora en adelante nos contentaremos con demostrar las tres propiedades, sabiendo que podemos usar el mismo esquema para completar la demostración de que el algoritmo voraz entrega un óptimo.

13.3. Knapsack (mochila)

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

Estrategia:

- Ordenar los objetos por

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

  decreciente.

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

Falta demostrar que esta estrategia da una solución óptima, lo que quedará de ejercicio.

Una variante obvia es objetos discretos: el objeto i se agrega completo o no (no fracciones). En este caso la estrategia voraz *no* da óptimo. Incluso vimos que este problema es NP-completo. Construir un contraejemplo para la estrategia indicada queda de ejercicio.

13.4. Árbol recubridor mínimo

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 13.1, en realidad de Jarník [6], redescubierto por Prim [10] y Dijkstra [4]) y el algoritmo de

Algoritmo 13.1: Algoritmo de Prim

```
procedure Prim(G)
    Ordenar G_E en orden de c(e) 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
end
```

Kruskal [8] (algoritmo 13.2). Ambos son algoritmos voraces, como puede apreciarse, aunque usan

Algoritmo 13.2: Algoritmo de Kruskal

```
procedure Kruskal(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
end
```

criterios diferentes.

La demostración de ambos se basa en:

Proposición 13.1. *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_1v_2 tiene costo mínimo entre los arcos entre V_1 y V_2, hay un árbol recubridor mínimo de G que incluye v_1v_2.*

Este es un problema muy importante, hay una variedad de algoritmos mejores que los planteados, como el de Chazelle [2, 3]. Karger, Klein y Tarjan [7] describen un algoritmo aleatorizado (ver el capítulo 33) con tiempo de ejecución esperado lineal. Curiosamente, todos ellos usan de alguna forma el primer algoritmo publicado para resolver este problema, el de Borůvka [1]. Una revisión tutorial relativamente reciente es la de Eisner [5], Mareš [9] discute los algoritmos en detalle.

13.5. Programar tareas con plazo fatal

Hay una colección de tareas, cada una de las cuales requiere ejecutarse por una unidad de tiempo en la única máquina disponible. La tarea i trae ganancia g_i si se completa antes de su plazo fatal d_i, en caso contrario no aporta nada. Se busca la secuencia de tareas a programar de forma de obtener la máxima ganancia.

Como cada tarea demanda una unidad de tiempo, podemos considerar el tiempo dividido en ranuras, que pueden estar libres u ocupadas. Es claro que debemos ver de programar las tareas que más ganancia traen, pero de forma que interfieran lo menos posible con otras tareas de menor ganancia. Vale decir, programar las tareas en la ranura libre más tardía antes de su plazo fatal. Esto sugiere ordenar las tareas por ganancia decreciente, y si hay empate en ganancia por plazo fatal decreciente; luego asignar cada tarea a la ranura libre más tardía en la cual aún cumple su plazo fatal o descartarla. Si la programación resultante tiene tiempos muertos, podemos compactar al final adelantando tareas. El tiempo del algoritmo resultante está dominado por el ordenamiento, si hay n tareas es $O(n \log n)$.

Para demostrar que esto da una solución óptima, recurrimos a nuestras tres propiedades:

Greedy Choice: Nuestro algoritmo elige la tarea $\hat{t}$ que más ganancia da de entre las que aún pueden cumplir su plazo fatal. Demostramos por contradicción que hay una solución óptima que incluye la tarea $\hat{t}$ así elegida.

Tomemos una solución óptima. Es claro que las ranuras antes del plazo fatal de $\hat{t}$ están todas ocupadas, ya que en caso contrario podemos programar $\hat{t}$ en una libre, contradiciendo que la solución es óptima.

Si la solución óptima incluye a $\hat{t}$, estamos listos. Si no la incluye, podemos tomar una tarea que termina antes del plazo fatal de $\hat{t}$. Si su ganancia es menor que la ganancia de $\hat{t}$, intercambiándola con $\hat{t}$ mejoramos la ganancia total, contradicción con que la solución sea óptima. La única posibilidad es que tenga la misma ganancia de $\hat{t}$, podemos intercambiarlas obteniendo una solución óptima que incluye a $\hat{t}$.

Inductive Substructure: Sea P el problema original, $\hat{t}$ la tarea elegida por el criterio voraz, y el problema P' lo que queda al asignar $\hat{t}$ a la última ranura libre antes de su plazo fatal. Una solución viable a P', o sea, una colección de tareas a programar entre las restantes, nunca puede entrar en conflicto con la programación de $\hat{t}$; podemos combinar una solución a P' con $\hat{t}$ para dar una solución viable a P.

En realidad, en este caso no hay restricciones «cruzadas», esto se cumple automáticamente al eliminar tareas que ya no pueden cumplir sus plazos fatales.

Optimal Substructure: Consideremos una solución óptima Π^* al problema P. Para simplificar notación, llamemos $|\Pi|$ a la ganancia de la solución viable Π al problema P, y similarmente $|t|$ la ganancia que reporta la tarea t.

Por **Greedy Choice**, podemos suponer sin pérdida de generalidad que Π^* incluye la elección voraz $\hat{t}$. Sea P' el problema que queda al eliminar $\hat{t}$ y las tareas que ya no pueden completarse. Sea Π' una solución óptima para P', por **Inductive Substructure** es compatible con $\hat{t}$; la ganancia de esa solución es:

$$|\Pi'| \leq |\Pi^*| - |\hat{t}|$$

(si fuera mayor, junto con $\hat{t}$ daría una solución mejor que la óptima). Pero la solución $\Pi^* \setminus \{\hat{t}\}$ da el valor $|\Pi^*| - |\hat{t}|$, y la combinación Π' con $\hat{t}$ es óptima.

Como se cumplen las tres propiedades, el algoritmo da una solución óptima.

13.6. 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.

13.6.1. 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. Como el tiempo es proporcional al largo leído, usaremos largos como medidas de tiempo. 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. Para n archivos 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 T mejora en:

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

Vea la figura 13.5, donde marcamos con asterisco los supuestos óptimos. Se observa que:

$$\begin{aligned} L_a - L_a^* &= l_b \\ L_b - L_b^* &= -l_a \end{aligned}$$

El cambio en el total es:

$$\begin{aligned} T - T^* &= p_a(L_a - L_a^*) + p_b(L_b - L_b^*) \\ &= p_a l_b - p_b l_a \\ &= p_a p_b \left(\frac{l_b}{p_b} - \frac{l_a}{p_a} \right) \end{aligned}$$

Pero $p_a p_b > 0$, ya que son probabilidades; y supusimos que $l_a/p_a > l_b/p_b$. Al intercambiarlos, disminuye el tiempo promedio. Esto contradice el que haya sido óptimo antes del cambio.

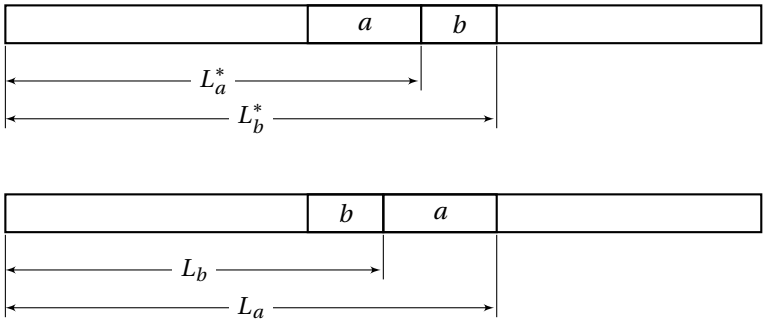


Figura 13.5 – Resultado de intercambiar los archivos a y b

Un caso particular importante de este problema se da al programar tareas que deben ejecutarse una tras otra, donde nos interesa minimizar el tiempo promedio de término de las tareas. Es el mismo problema anterior, si consideramos que las probabilidades de acceso son todas uno. Nuestro resultado indica ejecutar las tareas en orden de duración creciente (lo que llaman *Shortest Job First*, o SJF).

13.6.2. Usando un invariante

Un caso claro donde el esquema general discutido 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 13.3 describe esto informalmente. Demostramos que el algoritmo de Dijkstra es correcto demostrando por inducción sobre S que se cumple el invariante:

$$\forall v \in S, d[v] = \delta[v] \tag{13.1}$$

Base: Luego de la primera iteración tenemos $S = \{s\}$, donde es $d[s] = \delta(s) = 0$, el invariante vale.

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,

Algoritmo 13.3: Algoritmo de Dijkstra

```
foreach v ∈ V do
    d[v] ← ∞
end
d[s] ← 0
Inicialice Q como vacía
foreach v ∈ V do
    Inserte v en Q con prioridad d[v]
end
S ← ∅
while Q no vacía do
    u ← DeleteMin(Q)
    S ← S ∪ {u}
    foreach 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
```

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 $w(p^*) < w(p)$.

Como el invariante se cumple al principio del algoritmo, y se cumple luego de cada paso, se cumple al terminar. Pero al terminar el algoritmo todos los vértices están en S. Como en cada iteración se agrega un vértice a S, el algoritmo siempre termina.

Ejercicios

1. Demuestre en detalle que el algoritmo voraz da una solución óptima al problema de la mochila, demostrando las tres propiedades.

2. Demuestre la proposición 13.1.

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] Otakar Borůvka: *O jistém problému minimálním.* Práce Moravcské přírodovědecké společnosti, 3(3):37–58, 1926.

[2] Bernard Chazelle: *A minimum spanning tree algorithm with inverse Ackerman type complexity.* Journal of the ACM, 47(6):1028–1047, November 2000.

[3] Bernard Chazelle: *The soft heap: An approximate priority queue with optimal error rate.* Journal of the ACM, 47(6):1012–1027, November 2000.

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

[5] Jason Eisner: *State-of-the-art algorithms for minimal spanning trees.* https://www.cs.jhu.edu/~jason/papers/eisner.mst-tutorial.pdf, April 1997. Report for Written Preliminary Exam II, Department of Computer and Information Science, University of Pennsylvania.

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

[7] David R. Karger, Phillip N. Klein, and Robert E. Tarjan: *A randomized linear-time algorithm to find minimum spanning trees.* Journal of the ACM, 42(2):321–328, March 1995.

[8] 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.

[9] Martin Mareš: *Graph Algorithms.* PhD thesis, Department of Applied Mathematics, Faculty of Mathematics and Physics, Charles University in Prague, 2008.

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

Clase 14

Código Huffman

El código Huffman [2] es una aplicación muy importante de algoritmo voraz. Lo desarrolló como estudiante de pregrado, cuando se presentó la alternativa de dar un examen final o escribir un trabajo sobre la optimalidad de ciertos códigos en su ramo de teoría de comunicaciones. Vio que no podría resolver el problema planteado, y estaba a punto de abandonar cuando se le ocurrió la idea de este algoritmo, demostrando que es óptimo. En esto le ganó a sus profesores, que estaban desarrollando el código Shannon-Fano que resulta no ser óptimo.

Dado un texto, formado por símbolos, buscamos codificarlo eficientemente. Si cada símbolo se codifica en k bits (total 2^k símbolos posibles), de texto de largo n usa nk bits. En texto, las frecuencias son *muy* desiguales. Por ejemplo, en la novela Moby Dick aparece 117 194 veces la letra 'e', y 640 veces la 'z'. Nuestro principal objetivo es asignarle codificaciones más largas a los símbolos que menos se repiten y codificaciones más cortas a aquellos que se repiten más.

Pero hay que tener cuidado:

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

Con esta codificación escribimos:

$$ababc \rightsquigarrow 010101 \tag{14.1}$$

Pero también:

$$ccc \rightsquigarrow 010101 \tag{14.2}$$

¡Se produce ambigüedad entre (14.1) y (14.2)!

Obviamente nos interesan códigos que tengan decodificación única. Condición suficiente para evitar ambigüedades es que ningún código sea prefijo de otro («*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. Esto es importante porque hace eficiente el decodificar

14.1. 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.

Si es un código prefijo, podemos 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 símbolo rotula una de las hojas, el camino desde la raíz es el código de ese carácter (figura 14.1).

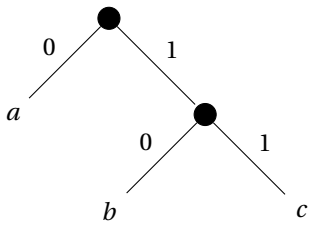


Figura 14.1 – Ejemplo de código prefijo como árbol

Definición 14.1. La *profundidad* de la hoja ℓ_i, anotada $d(\ell_i)$, es el largo del camino de la raíz a esa hoja.

En el código descrito por el árbol R el símbolo x_i queda codificado por $d(x_i)$ bits, el texto completo queda representado por el siguiente número de bits:

$$B(R) = \sum_i f_i d(x_i)$$

Observamanos que:

- Si R es óptimo, todo nodo interno tiene dos hijos.

  Si hubiese un nodo interno con un único hijo, podríamos acortar los caminos (códigos) de descendientes de su hijo haciéndolos depender directamente del nodo.

- Hay dos hojas x_a, x_b a la profundidad máxima que son hermanos.

  Por el punto anterior, no pueden haber nodos internos con un único hijo, toda hoja tiene un hermano.

Intuitivamente, buscamos letras poco frecuentes a altas profundidades, frecuentes a profundidades bajas. Lo que hace el algoritmo de Huffman es asignar desde los símbolos menos frecuentes, agrupando colecciones de símbolos. Sea $L = (\ell_1, \ldots, \ell_n)$ el conjunto de hojas para todos los símbolos, 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 14.2): Recursivamente resolver el problema con:

$$L = \{\ell_1, \ldots, \ell_n\} \setminus \{\ell_a, \ell_b\} \cup \{\ell_{ab}\} \tag{14.3}$$

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

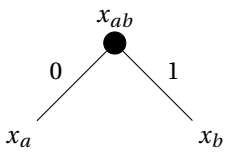


Figura 14.2 – El nodo x_{ab} es la unión entre x_a y x_b.

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

Cuadro 14.1 – Frecuencias de los símbolos a, b, c, d, e, f.

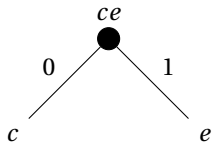


Figura 14.3 – Hojas son los símbolos que menos se repiten.

Ejemplo 14.1. Considere el cuadro 14.1. El algoritmo de Huffman encuentra los dos símbolos de menor frecuencia y crea un sub-árbol con ellos. En el cuadro 14.1, se tiene que los símbolos c y e son los de menor frecuencia. Luego, formamos un sub-árbol con ellos (figura 14.3). Agregamos este «nodo conjunto» al cuadro 14.1, cuya frecuencia es equivalente al peso del árbol de la figura 14.3 (suma de las frecuencias de c y e). El resultado se aprecia en el cuadro 14.2. Repetimos el proceso,

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

Cuadro 14.2 – Nodo conjunto ce con frecuencia la suma de las de c y e.

es decir, escogemos dos símbolos del cuadro 14.2 que tienen menor frecuencia y creamos un nuevo sub-árbol. Estos símbolos son ce y b. La figura 14.4 muestra el árbol resultante. Reemplazamos los símbolos b y ce del cuadro 14.2 con bce de frecuencia $f_{bce} = 9$. El resultado queda en el cuadro 14.3. Iteramos nuevamente. En el cuadro 14.3 se tiene que los dos símbolos con menor frecuencia son bce y a. El árbol resultante es el de la figura 14.5. Quitamos estos símbolos del cuadro 14.3 y los reemplazamos por $abce$. El resultado es el cuadro 14.4. En el cuadro 14.4 vemos que los símbolos con menor frecuencia son d y f. Tomamos estos dos símbolos y creamos un nuevo árbol que los tenga como hojas (figura 14.6). Sacamos esos símbolos y los reemplazamos por df, dando el cuadro 14.5. Tomamos los dos últimos símbolos y creamos el árbol final de la figura 14.7. La codificación

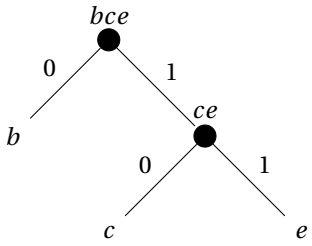


Figura 14.4 – Hojas son los símbolos que menos se repiten.

Símbolo	Frecuencia
a	9
d	15
f	17
bce	9

Cuadro 14.3 – Reemplazando los símbolos b y ce del cuadro 14.2.

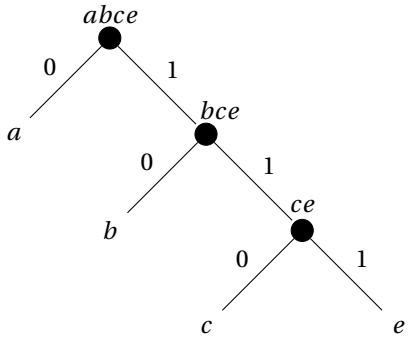


Figura 14.5 – Árbol con peso de $f_{bce} + f_a = 18$.

resultante se lee directamente del árbol, es la del cuadro 14.6. Si usáramos un código de largo fijo, requeriríamos $\lceil \log_2 6 \rceil = 3$ bits por símbolo. Nuestro código da 2,28 bits en promedio.

14.2. 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 al paso 1 hasta que nuestra tabla esté formada por solo un símbolo conjunto, que representará a todos los símbolos de Σ.

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

Símbolo	Frecuencia
d	15
f	17
$abce$	18

Cuadro 14.4 – Reemplazando los símbolos bce y a del cuadro 14.3.

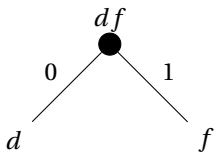


Figura 14.6 – Hojas son los símbolos de menor frecuencia del cuadro 14.4.

Símbolo	Frecuencia
d f	32
a b c e	18

Cuadro 14.5 – Reemplazando d y f del cuadro 14.4.

Demostración. Para demostrar que da un óptimo, usamos el teorema 13.1.

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}.

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.

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 (la profundidad d hace referencia al árbol original R). En el árbol R^*:

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

Pero R es óptimo. Hemos llegado a una contradicción.

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

Subestructura óptima: Sean x, y los símbolos menos frecuentes, con frecuencias f_x y f_y, respectivamente. Sea R' el árbol óptimo para el problema con el «símbolo» xy con frecuencia $f_x + f_y$. Debemos demostrar que el árbol R, con xy reemplazado por el subárbol respectivo con x e y

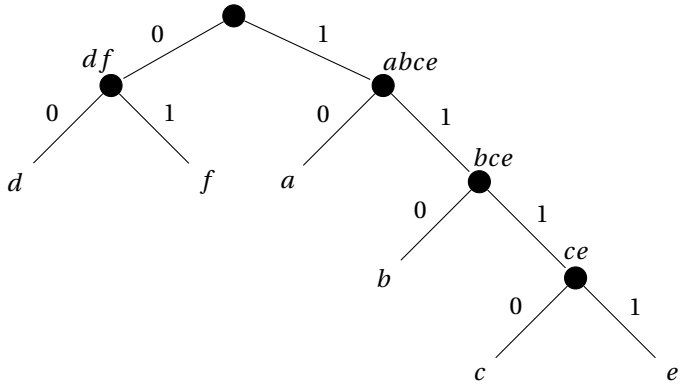


Figura 14.7 – Este árbol tiene un peso de $f_{bce} + f_a = 18$.

Símbolo	Freq	Código
a	9	10
b	4	110
c	2	1110
d	15	00
e	3	1111
f	17	01

Cuadro 14.6 – Codificación del ejemplo

es óptimo. Usaremos primas para distinguir valores en R' de valores en R. Primeramente:

$$\begin{aligned}
B(R) &= \sum_{s\in\Sigma} f_s d(s) \\
&= \sum_{s\in\Sigma\setminus\{x,y\}} f_s d(s) + f_x d(x) + f_y d(y) \\
&= \sum_{s\in\Sigma\setminus\{x,y\}} f_s d(s) + (f_x + f_y)(d'(xy) + 1) \\
&= \sum_{s\in\Sigma\setminus\{x,y\}} f_s d(s) + f'_{xy} d'(xy) + f_x + f_y \\
&= B(R') + f_x + f_y
\end{aligned}$$

Para llegar a una contradicción, suponga que R no es óptimo, y sea T un árbol óptimo con x e y de hojas hermanas (sabemos que existe por lo anterior). Sea T' el árbol que resulta de eliminar x e y. Podemos ver T' como un árbol para el alfabeto $\Sigma \setminus \{x,y\} \cup \{xy\}$ (agrupa x con y). Podemos repetir el cálculo anterior para obtener $B(T) = B(T') + f_x + f_y$. O sea:

$$\begin{aligned}
B(R') &= B(R) - f_x - f_y \\
&> B(T) - f_x - f_y \\
&= B(T')
\end{aligned}$$

Pero supusimos que R' era óptimo.

□

Ejercicios

1. Escriba un programa que lea las frecuencias de un conjunto de símbolos (para simplificar, considere caracteres ASCII únicamente) con sus frecuencias, y retorne una codificación de Huffman para ellos.

Bibliografía

[1] Guy E. Blelloch: *Introduction to data compression*. http://www.cs.cmu.edu/afs/cs/project/pscico-guyb/realworld/www/compression.pdf, January 2013. Department of Computer Science, Carnegie Mellon University.

[2] David A. Huffman: *A method for the construction of minimum-redundancy codes*. Proceedings of the IRE, 40(9):1098–1101, September 1952.

Clase 15

Sistemas de subconjuntos y matroides

Para muchos problemas hay una respuesta a la pregunta de por qué funcionan los algoritmos voraces.

Definición 15.1. Un *sistema de subconjuntos* es un conjunto $\mathscr{I}$ de subconjuntos de un conjunto $\mathscr{E}$ (el *conjunto base*) de modo que $\mathscr{I}$ es cerrado bajo inclusión.

O sea, si $\mathscr{B} \in \mathscr{I}$ y $\mathscr{A} \subseteq \mathscr{B}$ entonces $\mathscr{A} \in \mathscr{I}$. Note en particular que si $\mathscr{I} \neq \varnothing$, siempre es $\varnothing \in \mathscr{I}$.

El *problema de optimización* para un sistema de subconjuntos asigna un peso positivo a cada elemento de $\mathscr{E}$, y busca un conjunto $\mathscr{X} \in \mathscr{I}$ cuyo peso sea máximo en todos los conjuntos de $\mathscr{I}$. Es claro que podemos definir en forma afín la búsqueda de un mínimo. Retendremos esta definición para no complicar innecesariamente la discusión general. Parte de la discusión y los ejemplos que siguen vienen de Erickson [2].

Algunos ejemplos:

- Sea $\mathscr{E}$ un conjunto cualquiera de vectores de un espacio vectorial V, y sea $\mathscr{I}$ el conjunto de subconjuntos de $\mathscr{E}$ que son linealmente independientes. Claramente $\mathscr{I}$ es cerrado bajo inclusión.

  Por este caso es que al conjunto $\mathscr{I}$ se le suele llamar «de conjuntos independientes». En los casos de interés igualmente podremos identificar una relación de «dependencia» como acá.

- Considere:

  $$\mathscr{E} = \{a, b, c\}$$
  $$\mathscr{I} = \{\varnothing, \{a\}, \{b\}, \{a, b\}, \{b, c\}\}$$

  Podemos verificar directamente que es cerrado bajo inclusión. Una manera alternativa de describir $\mathscr{I}$ es como todos los conjuntos que no contienen a a y c.

- Sea $\mathscr{E}$ el conjunto de arcos de un grafo, y sean $\mathscr{I}$ los conjuntos de arcos que no comparten vértices (exactamente los conjuntos independientes del problema INDEPENDENT SET, también se les llama *matching* del grafo).

15.1. Algoritmo voraz genérico

Dado un sistema finito de subconjuntos $(\mathscr{E},\mathscr{I})$ hallamos un conjunto en $\mathscr{I}$ mediante el algoritmo 15.1. El algoritmo retorna un conjunto *maximal* (no se le pueden agregar elementos de $\mathscr{E}$

Algoritmo 15.1: Algoritmo voraz genérico

```
function Greedy(E, I)
    X ← ∅
    Ordene los elementos de E en orden de peso decreciente
    for x ∈ E do
        if X ∪ {x} ∈ I then
            X ← X ∪ {x}
        end
    end
    return X
end
```

sin salir de $\mathscr{I}$), pero no necesariamente *máximo* (no hay elementos de $\mathscr{I}$ de mayor peso). Nuestro problema de optimización pide un conjunto máximo.

Nuestro resultado es que hay una propiedad de sistemas de subconjuntos finitos que garantiza que el algoritmo voraz 15.1 da un conjunto maximal para todas las funciones de peso.

En nuestros ejemplos previos:

- Si consideramos los subconjuntos de $\{a,b,c\}$ que no tienen $\{a,c\}$ como subconjunto, y asignamos peso a los elementos, agregaremos b y aquél de $\{a,c\}$ de mayor peso.
- En los conjuntos de arcos que no forman ciclos, lo que tenemos es el problema *Maximal Weight Forest* (hallar el bosque de mayor peso que es subgrafo de G, abreviado MWF). Este problema es equivalente a MST, si el máximo peso de un arco en MST es m, asígnele peso $2m-w(e)$ al arco e. El algoritmo 15.1 para MWF aplicado a esto entrega un MST (es el algoritmo de Kruskal 13.2 disfrazado).

15.2. Matroides y algoritmos voraces

Diremos que un sistema de subconjuntos $(\mathscr{E},\mathscr{I})$ tiene la *propiedad de intercambio* si:

$$\forall \mathscr{A},\mathscr{B}\in\mathscr{I}, (|\mathscr{A}|<|\mathscr{B}|) \Longrightarrow (\exists e\in\mathscr{B}\setminus\mathscr{A} \text{ tal que } \mathscr{A}\cup\{e\}\in\mathscr{I}) \tag{15.1}$$

En estos términos:

Definición 15.2. Un *matroide* es un sistema de subconjuntos $M=(\mathscr{E},\mathscr{I})$ con la propiedad de intercambio.

Se les llama *conjuntos dependientes* (¡Sorpresa!) a los subconjuntos de $\mathscr{E}$ que no pertenecen a $\mathscr{I}$. El *rango* de $\mathscr{X}\subseteq\mathscr{E}$ es la cardinalidad de su máximo subconjunto independiente. Un conjunto independiente de dice que es una *base* de M si no es subconjunto de ningún conjunto independiente (es un conjunto independiente maximal). A un conjunto dependiente cuyos subconjuntos propios son todos independientes se le llama *circuito*.

Algunos ejemplos, varios de los cuales aparecerán nuevamente luego. Que son matroides quedará como ejercicio:

Matroide uniforme $U_{k,n}$: Un subconjunto $X \subseteq \{1,2,\ldots,n\}$ es independiente si y solo si $|X| \leq k$.

Todo subconjunto de k elementos es una base; todo conjunto de $k+1$ elementos es un circuito.

Matroide gráfico $\mathscr{M}(G)$: Sea $G = (V,E)$ un grafo. Un subconjunto de E es independiente si no contiene ciclos.

Una base del matroide es un árbol recubridor de G; un circuito es un ciclo en G.

Matroide cográfico $\mathscr{M}^*(G)$: Sea $G = (V,E)$ un grafo. Un subconjunto $I \subseteq E$ es independiente si el subgrafo complementario $(V, E \setminus I)$ es conexo.

Una base del matroide es el complemento de un árbol recubridor de G; un circuito es un *cociclo* de G, un conjunto mínimo de arcos que desconecta a G.

Matroide de correspondencias: Sea $G = (V,E)$ un grafo. Un conjunto $I \subseteq V$ es independiente si y solo si hay un *matching* (conjunto de arcos que no tienen vértices en común) que los cubre.

Caminos disjuntos: Sea $G = (V,E)$ un grafo dirigido, y sea s un vértice fijo de G. Un subconjunto $I \subseteq V$ es independiente si y solo si hay caminos que no comparten arcos desde s a cada elemento de I.

El resultado general es de Radó y Edmonds [1]:

Teorema 15.1 (Rado-Edmonds)**.** *Dado un sistema de subconjuntos $(\mathscr{E},\mathscr{I})$, las siguientes son equivalentes:*

1. *El algoritmo voraz 15.1 entrega una solución óptima para toda función de peso*
2. *El sistema de subconjuntos es un matroide*

Demostración. Demostramos implicancia en ambas direcciones.

Demostramos que si $M = (\mathscr{E},\mathscr{I})$ es un matroide entonces el algoritmo voraz entrega un óptimo por contradicción. Sea $\mathscr{A} = \{a_1, a_2, \cdots, a_k\}$ la solución entregada por el algoritmo voraz, y sea $\mathscr{B} = \{b_1, b_2, \cdots, b_{k'}\}$ una solución óptima, donde suponemos $w(\mathscr{B}) > w(\mathscr{A})$. Primero, $k = k'$, ya que si fuera $k' \neq k$ por la propiedad de intercambio podríamos agregar un elemento del conjunto mayor al otro. Esto o contradice la optimalidad de $\mathscr{B}$ o contradice el que el algoritmo terminó con $\mathscr{A}$. Luego, podemos suponer que los elementos de $\mathscr{A}$ y $\mathscr{B}$ se listan en orden de mayor a menor (en $\mathscr{A}$ es el orden en que los incluyó nuestro algoritmo), y consideremos el mínimo s tal que $w(b_s) > w(a_s)$. Sean los subconjuntos:

$$\alpha = \{a_1, \cdots, a_{s-1}\}$$
$$\beta = \{b_1, \cdots, b_s\}$$

Por ser parte de $\mathscr{I}$ los conjuntos $\mathscr{A}$ y $\mathscr{B}$, también son parte de $\mathscr{I}$ los conjuntos α y β. Por la propiedad de intercambio, hay t con $1 \leq t \leq s$ tal que $b_t \in \beta \setminus \alpha$ y $\alpha \cup \{b_t\} \in \mathscr{I}$. Pero $w(b_t) \geq w(b_s) > w(a_s)$, y nuestro algoritmo hubiese preferido b_t a a_s.

Al revés, si el algoritmo voraz siempre entrega un óptimo entonces $(\mathscr{E},\mathscr{I})$ es un matroide. Para esto basta demostrar que el algoritmo voraz puede no entregar un óptimo si $(\mathscr{E},\mathscr{I})$ no es un matroide. Si $(\mathscr{E},\mathscr{I})$ no es un matroide, entonces:

$$\exists \mathscr{A},\mathscr{B} \in \mathscr{I}, (|\mathscr{A}| < |\mathscr{B}|) \wedge (\nexists e \in \mathscr{B} \setminus \mathscr{A} \text{ tal que } \mathscr{A} \cup \{e\} \in \mathscr{I})$$

Sean $m = |\mathscr{A}|$ y $n = |\mathscr{E}|$. Defina:

$$w(e) = \begin{cases} m+2 & e \in \mathscr{A} \\ m+1 & e \in \mathscr{B} \setminus \mathscr{A} \\ 1/(2n) & \text{caso contrario} \end{cases}$$

El algoritmo voraz retorna $\mathscr{A}$, con peso a lo más $m(m+2)+1/2 = m^2+2m+1/2$; una solución mejor es $\mathscr{B}$ con peso al menos $(m+1)^2 = m^2+2m+1$. □

Otra propiedad interesante resulta de lo siguiente:

Definición 15.3. Un sistema de subconjuntos $(\mathscr{E}, \mathscr{I})$ tiene la *propiedad de cardinalidad* si:

$$\forall \mathscr{E}' \subseteq \mathscr{E}, (A, B \in \mathscr{I} \text{ subconjuntos maximales de } \mathscr{E}') \implies (|A| = |B|) \tag{15.2}$$

Decimos que $A \in \mathscr{I}$ es *subconjunto maximal* de $\mathscr{E}'$ si $A \subseteq \mathscr{E}'$ y no hay $a \in \mathscr{E}'$ tal que $A \cup \{a\} \in \mathscr{I}$. Con esto tenemos:

Teorema 15.2 (Propiedad de cardinalidad)**.** *Sea un sistema de subconjuntos $(\mathscr{E}, \mathscr{I})$. Entonces $(\mathscr{E}, \mathscr{I})$ es un matroide si y solo si cumple la propiedad de cardinalidad.*

Demostración. Es un si y solo si, demostramos implicancia en ambas direcciones.

Sean A, B subconjuntos maximales de $\mathscr{E}' \subseteq \mathscr{E}$. Debemos demostrar $|A| = |B|$, cosa que haremos por contradicción. Supongamos $|A| < |B|$, por la propiedad de intercambio:

$$\exists e \in B \setminus A, (A \cup \{e\} \in \mathscr{I})$$

Note que $A \cup \{e\} \in \mathscr{E}'$ ya que $e \in B \subseteq \mathscr{E}'$, o sea, A no sería maximal. El caso $|A| > |B|$ es simétrico.

Al revés, debemos demostrar que si $(\mathscr{E}, \mathscr{I})$ no es matroide entonces hay $\mathscr{E}'$ y $A, B \in \mathscr{I}$ con A, B maximales en $\mathscr{E}'$ con $|A| \neq |B|$. Si $(\mathscr{E}, \mathscr{I})$ no es matroide:

$$\exists A, C \in \mathscr{I}, |A| < |C| \wedge \nexists e \in C \setminus A \text{ con } A \cup \{e\} \in \mathscr{I}$$

Defina $\mathscr{E}' = A \cup C$, note que A es maximal en $\mathscr{E}'$. Hay $B \in \mathscr{I}$ tal que $C \subseteq B$ y B es maximal en $\mathscr{E}'$. Pero $|B| \geq |A| + 1$, como debíamos demostrar. □

Tenemos dos propiedades diferentes (intercambio y cardinalidad) que describen los matroides.

Note que nuestro primer ejemplo de algoritmo voraz (programar observaciones de ALMA) tiene un sistema de subconjuntos natural asociado: dos tareas son independientes si no traslapan (conjuntos independientes son los INDEPENDENT SET del grafo en el cual cada observación es un vértice, y dos vértices están unidos si traslapan). Esto *no* es un matroide, pueden haber conjuntos independientes maximales de tamaños distintos. Nuestro algoritmo voraz entrega un óptimo por la elección de función de peso (todos iguales), para otras funciones de peso falla.

Algunos ejemplos de matroides y los algoritmos voraces correspondientes:

- Los subconjuntos de un conjunto finito $\mathscr{U}$ de cardinalidad a lo más k.

  Podemos hallar el subconjunto más pesado usando el algoritmo voraz.

- Matroide de columnas. Sea **A** una matriz, $\mathscr{E} = \{\mathbf{x}\colon \mathbf{x} \text{ es columna de } \mathbf{A}\}$, y sea $\mathscr{I}$ los conjuntos de columnas linealmente independientes.

  Podemos hallar la base más pesada para las columnas de **A** usando el algoritmo voraz.

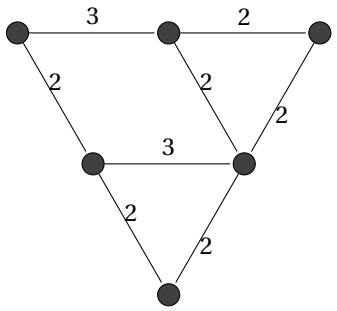
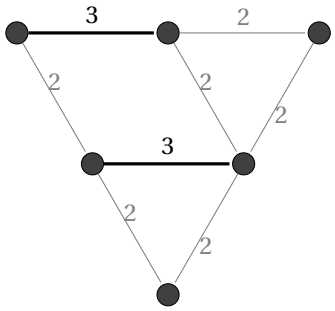
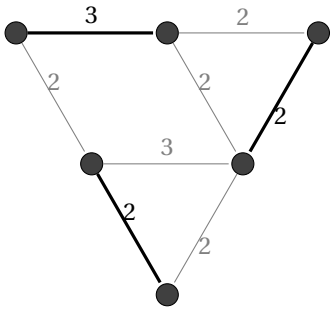


Figura 15.1 – Un grafo con arcos rotulados



(a) Resultado del algoritmo voraz



(b) Un óptimo

Nuestro último ejemplo de sistema de subconjuntos (arcos independientes en un grafo) no es un matroide, y el algoritmo voraz no siempre da un máximo. Considere el grafo de la figura 15.1. Al buscar un conjunto de arcos de máximo peso que no comparten vértices, el algoritmo voraz da el conjunto maximal que consta de los arcos de peso 3, para un total de 6 (ver 15.2a); pero el máximo es 7 (ver 15.2b).

La teoría de matroides nació de consideraciones sobre conjuntos linealmente independientes de vectores, y se vio aplicable a teoría de grafos. Rápidamente se extendió a otras áreas, un ejemplo es el texto de Lawler [3] sobre optimización combinatoria y matroides.

15.3. Programación con plazos fatales

Suponga que debe completar n tareas en n días, donde cada tarea demanda un día completo de dedicación. Cada tarea tiene un plazo fatal, si se completa después de su plazo fatal hay un costo a pagar. Se busca el orden en el cual ejecutar las tareas de forma de pagar el mínimo costo.

Formalmente, numeramos las tareas de 1 a n, dados un arreglo de plazos fatales D (un entero entre 1 y n) y uno de penalizaciones P (números reales no negativos). Un programa π es una permutación de $\{1,2,\ldots,n\}$. Buscamos un programa π que minimice:

$$\sum_{1\le i\le n} P[i]\cdot[\pi(i) > D[i]]$$

No parece para nada un ejemplo de optimización de matroide, allí solicitan un subconjunto y buscamos una permutación. Sorprendentemente, hay un matroide disfrazado en él. Para el programa π, diga que las tareas para las cuales $\pi(i) > D[i]$ están *atrasadas*, las demás *a tiempo*. La

observación trivial de que el costo de una programación queda determinada por sus tareas a tiempo lleva a revelar el matroide.

Llame *realista* a un conjunto de tareas X tal que hay una programación π en la que todas las tareas de X se completan a tiempo. Podemos caracterizar los subconjuntos realistas de la siguiente forma. Sea $X(t)$ el conjunto de tareas en X con plazo fatal en o antes de t:

$$X(t) = \{i \in X \colon D[i] \leq t\}$$

En particular, $X(0) = \varnothing$ y $X(n) = X$.

Proposición 15.1. *Sea $X \subseteq \{1, 2, \ldots, n\}$ un conjunto arbitrario de tareas. Entonces X es realista si y solo si $|X(t)| \leq t$ para todo t.*

Demostración. Es un si y solo si, demostramos implicancia en ambas direcciones.

Sea π una programación en la que todas las tareas de X están a tiempo. Sea i_t la t-ésima de X a completar. Por un lado, $\pi(i_t) \geq t$, ya que completamos $t-1$ tareas antes; por el otro, $\pi(i_t) \leq D[i_t]$ ya que i_t está a tiempo. Concluimos $D[i_t] \geq t$, por lo que $|X(t)| \leq t$.

Suponga ahora que $|X(t)| \leq t$ para todo t. Si ejecutamos las tareas de X en orden de plazo fatal, completamos las tareas con plazo fatal a más tardar t el día t. Para todo $i \in X$ estamos completando i antes de $D[i]$, X es realista. □

Llamemos *canónica* una programación para el conjunto de tareas X en la cual se ejecutan las tareas de X en orden de plazo fatal creciente, y las demás tareas en orden arbitrario. La proposición 15.1 nos dice que X es realista si y solo si todas sus tareas se completan a tiempo en su programación canónica. O sea, nuestro problema se puede reformular como hallar un subconjunto realista X que maximice:

$$\sum_{i \in X} P[i]$$

Estamos buscando un subconjunto óptimo.

Proposición 15.2. *La colección de conjuntos realistas es un matroide.*

Demostración. El conjunto vacío es realista (vacuamente), todo subconjunto de un conjunto realista es obviamente realista. Resta demostrar que se cumple la propiedad de intercambio. Sean entonces X e Y conjuntos realistas, con $|X| > |Y|$.

Sea t^* el máximo entero tal que $|X(t^*)| \leq |Y(t^*)|$. Debe existir, ya que $|X(0)| = 0 \leq 0 = |Y(0)|$ mientras $|X(n)| = |X| > |Y| = |Y(n)|$. Por la definición de t^*, hay más tareas con plazo fatal t^*+1 que t^* en X que en Y. O sea, podemos elegir $j \in X \setminus Y$ con plazo fatal t^*+1. Llamemos $Z = Y \cup \{j\}$.

Sea t arbitrario. Si $t \leq t^*$, entonces $|Z(t)| = |Y(t)| \leq t$, ya que Y es realista. Por otro lado, si $t > t^*$, $|Z(t)| = |Y(t)| + 1 \leq |X(t)|$ por la definición de t^* y dado que X es realista. Por la proposición 15.1, Z es realista. Se cumple la propiedad de intercambio. □

Por la proposición 15.2, nuestro problema de hallar la programación óptima es una optimización de matroide, el algoritmo voraz 15.2 da detalles. Falta determinar si un conjunto de tareas es realista. La proposición 15.1 da una pista cómo hacerlo, dando el algoritmo 15.3. Esto supone que X viene ordenado por plazo fatal:

$$i \leq j \implies D[X[i]] \leq D[X[j]]$$

El resultado se ejecuta en tiempo $O(n^2)$, usando estructuras de datos apropiadas esto se reduce a $O(n \log n)$. Detalles quedan de ejercicio.

Algoritmo 15.2: Algoritmo voraz para programar tareas

```
Ordene P en orden decreciente, y ordene D acorde
j ← 0
for i ← 1 to n do
    X[j+1] ← i
    if X[1..j+1] es realista then
        j ← j+1
    end
end
Retorne la programación canónica de X[1..j]
```

Algoritmo 15.3: Determinar si un conjunto es realista

```
function realista(X, D)
    N ← 0
    j ← 0
    for t ← 1 to n do
        if D[X[j]] = t then
            N ← N+1; j ← j+1
            if N > t then
                return F
            end
        end
    end
    return T
end
```

Ejercicios

1. Demuestre que los ejemplos de sistemas de subconjuntos citados realmente lo son. ¿Son matroides?

2. Demuestre que el matroide gráfico $\mathscr{M}(G)$ es un matroide para todo grafo G. Describa sus bases y circuitos.

3. Demuestre que para todo grafo G el matroide cográfico $\mathscr{M}^*(G)$ es un matroide.

4. Demuestre que para todo grafo G el matroide de correspondencias es un matroide.
   Pista: ¿Qué es la diferencia simétrica de dos correspondencias?

5. Indique qué entrega el algoritmo voraz en cada ejemplo de matroide citado.

6. Sea G un grafo. Un conjunto de ciclos $\{c_1, c_2, \ldots, c_k\}$ de G se llama *redundante* si cada arco de G aparece en un número par de c_i. Un conjunto de ciclos es *independiente* si no contiene subconjuntos redundantes. Un conjunto maximal de ciclos independientes es una *base de ciclos* de G.

   a) Sea C una base de ciclos de G. Demuestre que para cada ciclo γ de G, hay un subconjunto $A \subseteq C$ tal que $A \cap \{\gamma\}$ es redundante. O sea, γ es el «o exclusivo» de los ciclos de A.

   b) Demuestre que la colección de conjuntos de ciclos independientes es un matroide.

c) Suponga ahora que cada arco de G tiene un peso, que el peso de un ciclo es la suma de los pesos de sus arcos, y que el peso de un conjunto de ciclos es la suma de los pesos de los ciclos (note que arcos que se repiten se cuentan cada vez que aparecen). Describa y analice un algoritmo eficiente para hallar una base de ciclos de mínimo peso. (Esto no es sencillo, no es inmediato obtener los ciclos de G).

7. Demuestre que el sistema de subconjuntos del problema de programar observaciones de ALMA no es un matroide. Dé un ejemplo con una función de peso (retorno de la observación) tal que el algoritmo voraz no entregue una programación óptima.

8. Demuestre cómo programar tareas con plazo fatal en tiempo $O(n \log n)$.

   Pista: Use una estructura que permita determinar si $X \cup \{i\}$ es realista y agregar i a X en tiempo $O(\log n)$ cada operación.

Bibliografía

[1] Jack Edmonds: *Matroids and the greedy algorithm*. Mathematical Programming, 1(1):127–136, December 1971.

[2] Jeff Erickson: *Algorithms, etc.* http://jeffe.cs.illinois.edu/teaching/algorithms, January 2015. Department of Computer Science, University of Illinois at Urbana-Champaign.

[3] Eugene L. Lawler: *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, 1976.

Clase 16

Programación Dinámica

Hay situaciones que naturalmente se enfrentarían por búsquedas recursivas, pero en las cuales la búsqueda obvia termina resolviendo una y otra vez los mismos subproblemas. Hay dos opciones: la más fácil es registrar los subproblemas resueltos con sus soluciones, y revisar si un subproblema ya se resolvió antes de emprender su solución. Esto se conoce como *memoización*. La otra es *programación dinámica*, que consiste en calcular sistemáticamente las soluciones a subproblemas, de manera que cuando una de ellas se requiera ya esté calculada de antemano. La ventaja de programación dinámica es que no requiere una compleja estructura extra, nos ahorramos su administración y las búsquedas en ella. La exposición siguiente se organiza en parte siguiendo las sugerencias de Forišek [3] y usa la estructura dada por Erickson [2] para desarrollar algoritmos.

Requisitos para la aplicabilidad de programación dinámica son similares a las de algoritmos voraces (ver el capítulo 13). Suponemos un problema P, que se resuelve en etapas, eligiendo parte de la solución p en cada una de ellas.

Estructura Inductiva: Dada la elección $\hat{p}$, 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.

Elección Completa: Elegimos aquel $\hat{p}$ que da el mejor resultado combinado con Π', una solución óptima para el problema resultante de $P \setminus \{\hat{p}\}$.

A diferencia de un algoritmo voraz, no conocemos un criterio «local» que nos permita elegir $\hat{p}$, debemos considerar varias opciones. Esto lleva naturalmente a una recursión: resuelva los subproblemas recursivamente, y elija aquella combinación que da la solución global.

Lo anterior está planteado en términos de buscar un óptimo, pero puede adaptarse para determinar si hay o no soluciones.

16.1. Un primer ejemplo

Consideremos los números de Fibonacci, definidos por la recurrencia:

$$F_{n+2} = F_{n+1} + F_n \qquad F_0 = 0, F_1 = 1 \tag{16.1}$$

Esto lleva a la obvia función recursiva del listado 16.1. Si consideramos como medida de costo el

```
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)
```

Listado 16.1 – Cálculo de número de Fibonacci, recursión obvia

número de llamadas, llamando C_n al número de llamadas para calcular F_n resulta la recurrencia:

$$C_{n+2} = 1 + C_{n+1} + C_n \qquad C_0 = C_1 = 1 \tag{16.2}$$

Técnicas tradicionales de solución de recurrencias dan:

$$C_n = 2F_{n+1} - 1 \tag{16.3}$$

Sabemos que $F_n \sim \tau^n/\sqrt{5}$, donde $\tau = (1+\sqrt{5})/2$. Esto crece en forma exponencial.

Aplicar memoización es simple en Python, usamos un diccionario para registrar los valores ya calculados. Ver el listado 16.2.

```
fib = {0: 0, 1: 1} # Prefill values

def fibonacci(n):
    if(not n in fib):
        fib[n] = fibonacci(n - 1) + fibonacci(n - 2)
    return fib[n]
```

Listado 16.2 – Cálculo de número de Fibonacci, memoizado

La idea de programación dinámica es calcular sistemáticamente los valores, de forma de tenerlos disponibles cuando se requieren. Esto lleva directamente al listado 16.3. Si reconocemos además

```
def fibonacci(n):
    fib = [0, 1]
    for i in range(2, n + 1):
        fib.append(fib[i - 1] + fib[i - 2])
    return fib[n]
```

Listado 16.3 – Cálculo de número de Fibonacci, programación dinámica

que solo se necesitan los dos últimos valores, no requerimos almacenar los valores anteriores, llegamos al listado 16.4. Aún mejor, podemos simplificar si partimos la iteración con el valor $F_{-1} = 1$ (perfectamente consistente con la recurrencia), dando el programa 16.5.

Es claro que todas las formulaciones alternativas toman tiempo lineal.

Este ejemplo muestra las características salientes de la programación dinámica: tenemos un problema con solución recursiva obvia; tenemos las opciones de memoizar o usar programación dinámica; de usar programación dinámica puede ser suficiente retener solo parte de los valores calculados.

```
def fibonacci(n):
    if n <= 1:
        return n
    (a, b) = (0, 1)
    for i in range(2, n):
        (a, b) = (b, a + b)
    return a + b
```

Listado 16.4 – Cálculo de número de Fibonacci, guardando solo los últimos dos valores

```
def fibonacci(n):
    (a, b) = (1, 0)
    for i in range(0, n):
        (a, b) = (b, a + b)
    return b
```

Listado 16.5 – Cálculo simplificado de número de Fibonacci

16.2. Tome lo que más pueda

Hay n botellas en línea, numeradas de 0 a $n-1$ (¡somos computines!), la botella i contiene v_i de cerveza eslovaca. Después de un difícil semestre, queremos beber el máximo posible, pero nos ponen la restricción que no podemos beber dos botellas vecinas.

Este problema tiene una sencilla solución recursiva: debemos decidir si bebemos la última botella (en cuyo caso debemos omitir la penúltima, es claro que beberemos el máximo de las botellas 0 a $n-3$) o dejamos la última botella (en cuyo caso bebemos el máximo de las botellas 0 a $n-2$). Obtenemos la recurrencia para M_n, el máximo posible de beber si hay n botellas:

$$\begin{aligned} M_0 &= 0 \\ M_1 &= v_0 \\ M_{n+2} &= \max\{M_{n+1}, M_n + v_{n-1}\} \end{aligned}$$

Esto nos lleva al programa 16.6. Es fácil ver que el costo de este algoritmo (número de llamadas a

```
def solve(k):
    global v

    if k == 0:
        return 0
    elif k == 1:
        return v[0]
    else:
        return max(solve(k - 1), solve(k - 2) + v[k - 1])
```

Listado 16.6 – Beber cerveza recursivamente

solve) para n es F_{n+1}, que sabemos es exponencial en n.

Usar explícitamente un arreglo M lleva al programa 16.7. Notamos que solo se usan los últimos dos valores de M, no se requiere el arreglo completo. Esto lleva al programa final 16.8.

```
def solve():
    global v

    M = [0, v[0]]
    for k in range(2, len(v) + 1):
        M.append(max(M[k - 1], M[k - 2] + v[k - 1]))
    return M[len(v)]
```

Listado 16.7 – Beber cerveza por programación dinámica

```
def solve():
    global v

    (a, b) = (0, v[0])
    for k in range(2, len(v) + 1):
        (a, b) = (b, max(b, a + v[k - 1]))
    return b
```

Listado 16.8 – Beber cerveza por programación dinámica, versión final

16.3. Proyectos de plantas

Una corporación tiene US$ 5 millones a invertir este año, y planea expandir tres de sus plantas. Cada planta ha entregado a lo más tres propuestas, con sus costos y retornos estimados. Las diferentes propuestas de cada planta son excluyentes, vale decir, de las tres se puede ejecutar solo una. Además, los proyectos o se hacen (gastando el presupuesto completo) o no se hacen, no se pueden efectuar parcialmente. El cuadro 16.1 resume los costos de las propuestas y sus retornos. Algunas plantas no completaron las tres propuestas, y en todos los casos se agrega la propuesta de

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

Cuadro 16.1 – Propuestas, sus costos y retornos

«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 artificial 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 para esto. 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 16.2 resume los resultados. Estamos en condiciones de atacar la etapa 2, la mejor combinación para las

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

Cuadro 16.2 – 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 16.2). Por ejemplo, con el capital total de 5 si en la planta 2 elegimos la propuesta 3, tenemos un retorno de 9 y nos queda 2 para la planta 1, que da retorno 6, para un total de 15. El cuadro resume esto para las distintas opciones. Vamos por la etapa 3, con la misma idea

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

Cuadro 16.3 – Cómputo de la etapa 2

tenemos el cuadro 16.4. La entrada para $x = 5$ dice que el mejor retorno posible es 18. Nos indica que

Capital x	**Propuesta óptima**	**Retorno 1, 2 y 3**
0	0	0
1	0	5
2	1	9
3	0	13
4	1	17
5	1	18

Cuadro 16.4 – Cómputo de la etapa 3

la mejor opción para la planta 3 es su proyecto 1, lo que deja $5 - 1 = 4$ para las otras; del cuadro 16.3 vemos que la mejor opción para la planta 2 es la 1; queda $4 - 2 = 2$, con lo que del cuadro 16.2 vemos que la mejor opción para la planta 1 es la 2.

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.

Estamos desarrollando nuestra recurrencia «mirando hacia atrás», consideramos la etapa j suponiendo que tenemos resuelto el problema hasta $j-1$, el cálculo resultante «camina hacia adelante», aumentando j. Claramente es igualmente válido «mirar hacia adelante» resultando una recurrencia «marcha atrás», las soluciones serán equivalentes. Cuál de las opciones es más natural depende del problema (y de las inclinaciones del programador).

16.4. Estructura general

El desarrollo de los ejemplos anteriores sigue el siguiente esquema:

(a) **Plantear la recurrencia:** Esta es la parte crítica, depende íntimamente del problema. Debemos identificar los subproblemas relevantes, cómo se combinan para una solución, identificar todas las alternativas relevantes y cómo elegir la mejor de las alternativas.

(b) **Escribir un programa recursivo:** Generalmente es una traducción mecánica de la recurrencia.

(c) **Identificar subproblemas:** Vea todas las formas en las que el programa recursivo se llama a sí mismo. Determine el conjunto de posibles argumentos.

(d) **Defina una estructura de datos:** Requerimos almacenar resultados para todas las combinaciones posibles de argumentos. Esto generalmente lleva a alguna clase de tabla, pero perfectamente puede ser apropiada una estructura diferente.

(e) **Identifique dependencias:** Debemos organizar los cálculos de forma que valores requeridos ya se hayan calculado antes. Por ejemplo, considere un valor genérico, y dibuje los valores de los que depende. Formalice esto.

(f) **Determine un buen orden de cálculo:** El orden en que se obtienen los resultados en el programa recursivo es una guía, pero no es necesario seguirlo estrictamente. Interesa definir un orden que sea simple de programar. Las dependencias descubiertas en el paso anterior definen un orden parcial entre subproblemas, buscamos una extensión lineal. Esto es crítico, tenga cuidado.

(g) **Analice requerimientos de tiempo y espacio:** Depende fundamentalmente del número de subproblemas y lo que se debe hacer para cada uno de ellos. Incluso es posible de obtener directamente luego del paso (c).

(h) **Escriba el algoritmo:** Esto es inmediato si se desarrollaron cuidadosamente los pasos anteriores.

16.5. Subset Sum

La idea de programación dinámica puede emplearse siempre que en una propuesta recursiva de solución hayan subproblemas que se repiten. Comúnmente se usa para problemas de optimización, pero no es el único uso.

Dado un conjunto de enteros positivos $\mathscr{A} = \{a_1, a_2, \ldots, a_n\}$, se pide determinar si hay un subconjunto de $\mathscr{A}$ que suma s.

Definimos $p[i,t]$ como verdadero si entre $\{a_1, \ldots, a_i\}$ hay un subconjunto que suma t.

La recurrencia sobre i es bastante obvia. Al agregar a_{i+1} a los elementos considerados, podemos simplemente no usarlo (es posible obtener t con $\{a_1, \ldots, a_i\}$) o lo incluimos (debemos obtener $t - a_{i+1}$ con $\{a_1, \ldots, a_i\}$). O sea:

$$p[i+1,t] = p[i,t] \vee p[i,t-a_{i+1}]$$

Tenemos condiciones iniciales:

$$p[i,t] = \begin{cases} F & \text{si } t > 0 \text{ y } i = 0 \\ T & \text{si } t = 0 \end{cases}$$

Es simple llevar esto al algoritmo 16.1 recursivo, donde hemos aprovechado cálculo en corto circuito de la expresión. Debemos tener cuidado de no considerar incluir a_i si $a_i > t$, como se muestra. Obtenemos el resultado como HaySuma(n, S). En el peor caso, este algoritmo considera los 2^n sub-

Algoritmo 16.1: Hay subconjunto de $\{a_1, \ldots, a_i\}$ con la suma t dada

```
function HaySuma(i, t)
    if t = 0 then
        return T
    end
    if i = 0 then
        return F
    end
    if HaySuma(i − 1, t) then
        return T
    end
    if t ≤ a_i then
        return HaySuma(i − 1, t − a_i)
    else
        return F
    end
end
```

conjuntos de los a_i.

Una solución por programación dinámica llena el arreglo desde $t = 0$, para cada $t = 0, 1, \ldots$ calculamos $p[t,i]$ sistemáticamente, con lo que los valores requeridos los habremos calculado antes. Ver el algoritmo 16.2. Luego de ejecutar este algoritmo, $p[n,s]$ nos dice si es posible o no la suma dada. Si quisiéramos además obtener un subconjunto que logra la suma pedida, habría que registrar con los $p[t,i]$ verdaderos cuál fue la opción que dio verdadero, y revisar hacia atrás desde el resultado final.

La complejidad de este algoritmo es $O(ns \log s)$, estamos calculando esencialmente ns elementos, cada uno de los cuales significa algunas operaciones entre palabras de $\log_2 s$ bits. Nótese que

Algoritmo 16.2: Subconjunto de $\{a_1, \ldots, a_n\}$ que suma s, programación dinámica

```
for t ← 1 to s do
    p[0, t] = F
end
for i ← 0 to n do
    p[i, 0] ← T
end
for t ← 1 to s do
    for i ← 1 to n do
        if a_i > t then
            p[i, t] ← p[i − 1, t]
        else
            p[i, t] ← p[i − 1, t] ∨ p[i − 1, t − a_i]
        end
    end
end
```

si s es substancialmente mayor a 2^n, el algoritmo recursivo es menos costoso. Programación dinámica no siempre es la mejor opción. Vimos en *Informática Teórica* (INF-155) que este problema (SUBSETSUM) es NP-completo. En términos del largo de la representación en binario este algoritmo no es polinomial, pero sí lo es en términos de los valores de los datos de entrada (los n elementos a_i y s). A tales algoritmos se les llama *pseudopolinomiales*.

16.6. Subsecuencia creciente más larga

Dado una secuencia de n elementos, nos interesa determinar el largo de la subsecuencia creciente más larga. Por ejemplo, una subsecuencia creciente más larga está marcada con negrilla en la siguiente secuencia:

$$80, \mathbf{10}, \mathbf{22}, 9, \mathbf{33}, 21, \mathbf{50}, 41, \mathbf{60}, 23, 7$$

Sea a_i el i-ésimo elemento de la secuencia ($0 \le i < n$), y sea L_i el largo de la secuencia creciente más larga *que termina en* a_i. Nos interesa el valor máximo de L_i.

Note que nuestros subproblemas no son del mismo tipo que el problema inicial, estamos poniendo la condición de que el último elemento de la secuencia sea parte de la secuencia creciente, y el resultado no es simplemente el valor registrado en una posición fija.

Pensando en la composición de la subsecuencia creciente más larga que termina con a_i, antes de a_i hay una subsecuencia creciente más larga que termina en a_j, con $a_j < a_i$. Esto hace plantear la recurrencia:

$$L_i = \begin{cases} 1 + \max\limits_{\substack{0<j<i \\ a_j \le a_i}} \{L_j\} & \text{si tal } j \text{ existe} \\ 1 & \text{caso contrario} \end{cases} \tag{16.4}$$

Como mencionamos, el valor buscado es $\max_{0 \le i < n}\{L_i\}$.

Hay subproblemas repetidos, vale la pena pensar en programación dinámica. En nuestro caso, calcular los L_i sistemáticamente. El programa Python del listado 16.9 da detalles. Esto entrega el largo de la subsecuencia más larga, obtener una subsecuencia más larga queda de ejercicio.

```
def LIS(a):
    n = len(a)
    L = [0 for i in range(n)]
    L[0] = 1
    for i in range(n):
        max = 0
        for j in range(i):
            if a[j] < a[i] and L[j] > max:
                max = L[j]
        L[i] = 1 + max

    max = L[0]
    for i in range(1, n):
        if L[i] > max:
            max = L[i]

    return max
```

Listado 16.9 – Subsecuencia creciente más larga

16.7. Producto de matrices

Queremos calcular el producto de n matrices, $\mathbf{A}_1 \cdot \mathbf{A}_2 \cdots \mathbf{A}_n$, donde $\mathbf{A}_i$ es $n_i \times n_{i+1}$ (para que sea posible el producto $\mathbf{A}_i \cdot \mathbf{A}_{i+1}$).

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 esto.

Sabemos que la multiplicación de matrices es asociativa:

$$\mathbf{A} \cdot (\mathbf{B} \cdot \mathbf{C}) = (\mathbf{A} \cdot \mathbf{B}) \cdot \mathbf{C}$$

El trabajo total depende del orden. Por ejemplo, si tenemos matrices:

$\mathbf{A}$: 2×12

$\mathbf{B}$: 12×3

$\mathbf{C}$: 3×4

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

$$2 \cdot 12 \cdot 3 + 2 \cdot 3 \cdot 4 = 96$$

El primer término corresponde al producto $\mathbf{A} \cdot \mathbf{B}$, el segundo a multiplicar esto por $\mathbf{C}$. Continuamos con el cálculo del costo de obtener $\mathbf{A} \cdot (\mathbf{B} \cdot \mathbf{C})$:

$$2 \cdot 12 \cdot 4 + 12 \cdot 3 \cdot 4 = 240$$

Para obtener el óptimo, consideremos «de afuera adentro». Si el último producto es:

$$\underbrace{(\mathbf{A}_1 \cdot \mathbf{A}_2 \cdots \mathbf{A}_i)}_{\text{óptimo}} \underbrace{(\mathbf{A}_{i+1} \cdots \mathbf{A}_n)}_{\text{óptimo}} \tag{16.5}$$

Nótese que $(\mathbf{A}_1 \cdot \mathbf{A}_2 \cdots \mathbf{A}_i)$ y $(\mathbf{A}_{i+1} \cdots \mathbf{A}_n)$ 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, 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 $\mathbf{A}_i \cdot \ldots \cdot \mathbf{A}_j$.

Inicialmente:

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

Sabemos que:

$$T[i,j] = \min_{i \leq k < j} \{T[i,k] + T[k+1,j] + \underbrace{n_i \cdot n_{k+1} \cdot n_{j+1}}_{\text{costo del producto}}\} \tag{16.6}$$

Donde k corresponde a la k-ésima matriz que hicimos el corte (Donde dividimos con los paréntesis. Por ejemplo, en (16.5) se tiene que $k = i$.) y que dio el valor mínimo de (16.6). Además, n corresponde al valor izquierdo de la dimensión de la matriz. Por ejemplo, si nuestra $\mathbf{A}_i$ tiene dimensión $a \times b$, $\mathbf{A}_{k+1} : c \times d$ y $\mathbf{A}_{j+1} : e \times f$, 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 16.1. Supongamos que queremos calcular el producto de matrices:

$$\mathbf{A} \cdot \mathbf{B} \cdot \mathbf{C} \cdot \mathbf{D} \cdot \mathbf{E} \cdot \mathbf{F} \cdot \mathbf{G} \cdot \mathbf{H}$$

donde las dimensiones son:

- $\mathbf{A} = \mathbf{A}_1 : 2 \times 3$
- $\mathbf{B} = \mathbf{A}_2 : 3 \times 4$
- $\mathbf{C} = \mathbf{A}_3 : 4 \times 1$
- $\mathbf{D} = \mathbf{A}_4 : 1 \times 9$
- $\mathbf{E} = \mathbf{A}_5 : 9 \times 3$
- $\mathbf{F} = \mathbf{A}_6 : 3 \times 7$
- $\mathbf{G} = \mathbf{A}_7 : 7 \times 2$
- $\mathbf{H} = \mathbf{A}_8 : 2 \times 8$

	1	2	3	4	5	6	7	8
1	0 1							
2		0 2						
3			0 3					
4				0 4				
5					0 5			
6						0 6		
7							0 7	
8								0 8

Cuadro 16.5 – 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 $\mathbf{A}_i$. Como 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 16.5. Nótese que cada casilla del cuadro 16.5 contiene $T[i,j]$ y el k que dio el mínimo. Para la segunda iteración, tenemos que calcular todos los $T[i,i+1]$, es decir, la mínima cantidad de multiplicaciones para obtener $\mathbf{A}_i \cdot \mathbf{A}_{i+1}$. Como solo tenemos dos matrices involucradas, es bastante fácil realizar este cálculo:

- $T[1,2] = 2 \cdot 3 \cdot 4 = 24$
- $T[2,3] = 3 \cdot 4 \cdot 1 = 12$
- $T[3,4] = 4 \cdot 1 \cdot 9 = 36$
- $T[4,5] = 1 \cdot 9 \cdot 3 = 27$
- $T[5,6] = 9 \cdot 3 \cdot 7 = 189$
- $T[6,7] = 3 \cdot 7 \cdot 2 = 42$
- $T[7,8] = 7 \cdot 2 \cdot 8 = 112$

Luego, agregamos estos valores al cuadro 16.5. Los cambios se pueden apreciar en el cuadro 16.6. Seguimos con la tercera iteración, completando la diagonal siguiente. En esta ocasión, tendremos que calcular los $T[i,i+2]$, es decir, la cantidad de multiplicaciones mínima para obtener $\mathbf{A}_i \cdot \mathbf{A}_{i+1} \cdot \mathbf{A}_{i+2}$. Para ello, comenzamos calculando $T[1,3]$, es decir:

$$T[1,3] = \min_{1 \le k < 3} \{T[1,k] + T[k+1,3] + n_1 \cdot n_{k+1} \cdot n_4\} \tag{16.7}$$

Vamos por partes:

- Para $k = 1$:

$$T[1,1] + T[2,3] + n_1 \cdot n_2 \cdot n_4 = 0 + 12 + 2 \cdot 3 \cdot 1 = 18$$

	1	2	3	4	5	6	7	8
1	0 1	24 1						
2		0 2	12 2					
3			0 3	36 3				
4				0 4	27 4			
5					0 5	189 5		
6						0 6	42 6	
7							0 7	112 7
8								0 8

Cuadro 16.6 – Segunda iteración.

- Para $k = 2$:

$$T[1,2] + T[3,3] + n_1 \cdot n_3 \cdot n_4 = 24 + 0 + 2 \cdot 4 \cdot 1 = 32$$

Por lo tanto, de acuerdo a lo anterior la ecuación (16.7) obtiene el mínimo 18 cuando hacemos el corte en $k = 1$. Es decir, obtenemos el mínimo de productos para $\mathbf{A}_1 \cdot \mathbf{A}_2 \cdot \mathbf{A}_3$ si hacemos un corte

$$\mathbf{A}_1 \cdot (\mathbf{A}_2 \cdot \mathbf{A}_3) = \mathbf{A} \cdot (\mathbf{B} \cdot \mathbf{C}) \tag{16.8}$$

Agregamos estos datos a la tabla.

Para dejar más en claro cómo calcular (16.6) repetiremos los pasos anteriores, pero para calcular $T[4,7]$, el que se puede obtener a través de:

$$T[4,7] = \min_{4 \le k < 7} \{T[4,k] + T[k+1,7] + n_4 \cdot n_{k+1} \cdot n_7\}$$

Vamos viendo:

$$\begin{aligned}
k = 4\colon\ & T[4,4] + T[5,7] + 1 \cdot 9 \cdot 2 = \ \ 0 + 96 + 18 = 114 \\
k = 5\colon\ & T[4,5] + T[6,7] + 1 \cdot 3 \cdot 2 = 27 + 42 + \ \ 6 = \ \ 75 \\
k = 6\colon\ & T[4,6] + T[7,7] + 1 \cdot 7 \cdot 2 = 48 + \ \ 0 + 14 = \ \ 62
\end{aligned}$$

El mínimo se obtiene para $k = 6$, de costo 62, es decir:

$$(\mathbf{A}_4 \cdot \mathbf{A}_5 \cdot \mathbf{A}_6) \cdot \mathbf{A}_7 = (\mathbf{D} \cdot \mathbf{E} \cdot \mathbf{F}) \cdot \mathbf{G} \tag{16.9}$$

Note que calculando sistemáticamente las diagonales, cuando se requieran valores ya estarán calculados de antes.

El objetivo de este ejemplo es mostrar cómo funciona el algoritmo, por lo que no es necesario explicar paso a paso cómo obtener el resto de las casillas de la tabla. El resultado final es el dado en el cuadro 16.7.

	1	2	3	4	5	6	7	8
1	0 1	24 1	18 1	36 3	51 3	80 3	84 3	112 3
2		0 2	12 2	39 3	48 3	81 3	80 3	114 3
3			0 3	36 3	39 3	76 3	70 3	110 3
4				0 4	27 4	48 5	62 6	78 7
5					0 5	189 5	96 5	240 7
6						0 6	42 6	90 7
7							0 7	112 7
8								0 8

Cuadro 16.7 – Tabla final

Demostración. Como siempre, demostramos que cumple con:

Estructura inductiva: Dada la selección k (última) se subdivide en problemas, cuyas soluciones viables junto con 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».

□

Por el momento solo nos interesaba obtener el valor del costo mínimo. Con los valores de k registrados en la tabla podemos calcular el orden óptimo. La entrada $(1,8)$ nos dice que el mejor corte final está luego de $\mathbf{A}_3 = C$, o sea:

$$(\mathbf{A} \cdot \mathbf{B} \cdot \mathbf{C}) \cdot (\mathbf{D} \cdot \mathbf{E} \cdot \mathbf{F} \cdot \mathbf{G} \cdot \mathbf{H})$$

Seguimos viendo la subdivisión óptima para $\mathbf{A} \cdot \mathbf{B} \cdot \mathbf{C}$ en $1,3$, que resulta ser:

$$\mathbf{A} \cdot (\mathbf{B} \cdot \mathbf{C})$$

Similarmente, $4,8$ nos indica:

$$(\mathbf{D} \cdot \mathbf{E} \cdot \mathbf{F} \cdot \mathbf{G}) \cdot \mathbf{H}$$

Continuamos de la misma forma, obteniendo finalmente:

$$(\mathbf{A} \cdot (\mathbf{B} \cdot \mathbf{C})) \cdot ((((\mathbf{D} \cdot \mathbf{E}) \cdot \mathbf{F}) \cdot \mathbf{G}) \cdot \mathbf{H})$$

Es sencillo escribir un programa recursivo que recorra la tabla para extraer la subdivisión óptima.

16.8. Subsecuencia común más larga

En inglés conocido como *Longest Common Subsequence*, LCS. Dadas dos palabras X e Y, de símbolos $x_1, x_2, \ldots, x_m$ e $y_1, y_2, \ldots, y_n$ respectivamente, buscamos la secuencia más larga de símbolos $x_{i_k} = y_{j_k}$ tales que i_k y j_k son ambas crecientes.

Esta es la base del comando Unix `diff(1)`, que compara dos archivos X e Y (se consideran las líneas como «símbolos»), y entrega las líneas eliminadas e insertadas para cambiar X a Y:

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

Supongamos que tenemos dos archivos: X e Y, cuyo contenido se muestra en el cuadro 16.8. En

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

Cuadro 16.8 – Los archivos X e Y. Cada fila de la tabla es una línea del archivo.

consecuencia, si hacemos un `diff X Y` obtenemos como resultado, la columna «Resultado» del cuadro 16.9. Esta operación es crítica en sistemas de control de versiones (para ahorrar espacio

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 16.9 – La columna «Resultado» resume las operaciones.

almacenan solo las diferencias, que suelen ser pequeñas entre versiones sucesivas), y es fundamental para mostrar diferencias entre versiones. Mucho de la biología molecular es determinar diferencias entre secuencias de aminoácidos de proteínas o de genes similares

16.9. Aspectos formales

Nos dan 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:

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

Interesa la secuencia más larga (máximo q, correspondiente a la cantidad de 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 `(+)`
3. Si $X[n] = Y[m]$, hacer $x_q = n$, $y_q = m$.

Pueden quedar fuera ambos, pero eso resulta automáticamente de eliminar uno y luego el otro. Notar que si $X[n] \neq Y[m]$ no pueden pertenecer ambos a la LCS.

En resumen, hay tres opciones que dan lugar a los siguientes subproblemas:

1. $X[1,\ldots,n-1]$, $Y[1,\ldots,m]$
2. $X[1,\ldots,n]$, $Y[1,\ldots,m-1]$
3. Solo si $X[n] = Y[m]$ considerar $X[1,\ldots,n-1]$, $Y[1,\ldots,m-1]$ (y contabilizar una coincidencia)

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):

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

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

$$L[i,j] = |LCS(X[1,\ldots,i),Y[1,\ldots,j|] \tag{16.10}$$

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 $O(n\cdot m)$.

16.10. Árboles binarios óptimos

Nos dan un arreglo ordenado de claves $A[1,\ldots,n]$ con sus respectivas frecuencias de búsqueda $f[1,\ldots,n]$. Buscamos crear un árbol binario para el cual el costo total de las búsquedas sea mínimo.

Requerimos una forma de plantear la cantidad a optimizar. Fijemos un árbol binario de búsqueda T de las claves A, con la clave $A[i]$ en el nodo v_i, el costo total de las búsquedas en T es proporcional a:

$$C(T,f) = \sum_i f[i]\cdot \text{número de ancestros de } v_i \text{ en } T$$

Podemos particionar C según subárboles de v_r:

$$C(T,f) = \sum_i f[i] + \sum_{1\le i<r} f[i]\cdot \#\text{ ancestros de } v_i \text{ en left}(T) + \sum_{r<i\le n} f[i]\cdot \#\text{ ancestros de } v_i \text{ en left}(T)$$

Las dos sumas son de la misma forma de nuestro original, obtenemos la recursión:

$$C(T,f) = \sum_i f[i] + C(\text{left}(T)) + C(\text{right}(T))$$

El caso base es para $n = 0$, buscar en el árbol vacío tiene costo cero.

Es claro que las claves no inciden, dada la raíz $A[r]$ el subárbol derecho y el izquierdo deben ser óptimos para las claves en sus respectivos rangos. Fijemos el arreglo de frecuencias, y sea $\text{Opt}(i,k)$ el costo total de buscar en el árbol óptimo para las claves $A[i,\ldots,k]$. Nuestra recurrencia se transforma en:

$$\text{Opt}(i,k) = \begin{cases} 0 & i > k \\ \sum_{i\le j\le k} f[j] + \min_{i\le r\le k}\{\text{Opt}(i,r-1) + \text{Opt}(r+1,k)\} & \text{caso contrario} \end{cases}$$

El programa resulta más simple (y más eficiente) si precalculamos el primer término:

$$F[i,k] = \sum_{i\le j\le k} f[j]$$

Podemos calcular los valores requeridos en tiempo $O(n^2)$ mediante (¡sorpresa!) programación dinámica, algoritmo 16.3. (En realidad, bastaría calcular $\tilde{F}[k] = \sum_{1\le j\le k} f[j]$, y obtener $F[i,k] = \tilde{F}(k) -$

Algoritmo 16.3: Calcular $F[i,k]$

```
procedure InitF(f)
    for i ← 1 to n do
        F[i, i−1] ← 0
        for k ← i to n do
            F[i,k] ← F[i,k−1] + f[k]
        end
    end
end
```

$\tilde{F}[i-1]$, ahorrando espacio en el proceso; pero esto es irrelevante en este caso. Por lo demás, no es el momento de preocuparse de optimizaciones.) Nuestra recurrencia se simplifica a:

$$\text{Opt}(i,k) = \begin{cases} 0 & i > k \\ F[i,k] + \min_{i\le r\le k}\{\text{Opt}(i,r-1) + \text{Opt}(r+1,k)\} & \text{caso contrario} \end{cases}$$

Estamos listos para seguir nuestra estrategia.

Subproblemas: Cada subproblema queda descrito por dos enteros, $1 \le i \le n+1$ y $0 \le k \le n$.

Estructura de datos: Se requiere un arreglo $\text{Opt}[1..n+1, 0..n]$ para registrar los valores (solo se usan las entradas $\text{Opt}[i,j]$ con $j \ge i-1$, pero da lo mismo; si realmente se requiriera ahorrar espacio se puede almacenar solo la mitad usada del arreglo).

Dependencias: Cada entrada $\text{Opt}[i,k]$ depende de las entradas $\text{Opt}[i,j-1]$ y $\text{Opt}[j+1,k]$ para j tales que $i \le j \le k$. Vale decir, las que están directamente a la izquierda o debajo. El algoritmo 16.4 calcula $\text{Opt}[i,k]$.

Algoritmo 16.4: Cómputo de Opt$[i,k]$

```
procedure ComputeOpt(i, k)
    Opt[i, k] ← ∞
    for r ← i to n do
        tmp ← Opt[i, r − 1] + Opt[r + 1, k]
        if Opt[i, k] > tmp then
            Opt[i, k] ← tmp
        end
    end
    Opt[i, k] ← Opt[i, k] + F[i, k]
end
```

Algoritmo 16.5: Llenar en diagonal

```
function OptimalBST(f)
    InitF(f)
    for i ← 1 to n + 1 do
        Opt[i, i − 1] ← 0
    end
    for d ← 0 to n − 1 do
        for i ← 1 to n − d do
            ComputeOpt(i, i + d)
        end
    end
    return Opt[1, n]
end
```

Orden de evaluación: Hay varias opciones. Una es llenar el arreglo una diagonal a la vez, partiendo con los valores triviales Opt$[i,i-1]$ y llegando a Opt$[1,n]$, algoritmo 16.5. Otras opciones son llenar por filas de abajo arriba, algoritmo 16.6; o llenar por columnas de izquierda a derecha, llenando cada columna de abajo arriba, algoritmo 16.6.

Algoritmo 16.6: Llenar por filas

```
function OptimalBST(f)
    InitF(f)
    for i ← n + 1 downto 1 do
        Opt[i, i − 1] ← 0
        for j ← i to n do
            ComputeOpt(i, j)
        end
    end
    return Opt[1, n]
end
```

Es claro que el algoritmo resultante toma tiempo $O(n^3)$ y usa espacio $O(n^2)$.

Algoritmo 16.7: Llenar por columnas

```
function OptimalBST(f)
    InitF(f)
    for j ← 0 to n + 1 do
        Opt[j + 1, j] ← 0
        for i ← j downto 1 do
            ComputeOpt(i, j)
        end
    end
    return Opt[1, n]
end
```

16.11. Máximo conjunto independiente de un árbol

Definimos *conjunto independiente* en un grafo como un conjunto de vértices que no están conectados entre sí, y vimos que determinar si un grafo cualquiera tiene un conjunto independiente de tamaño k (el problema INDEPENDENT SET) es NP-completo. Sin embargo, en el caso de árboles hay una solución eficiente.

Sin pérdida de generalidad, supongamos un árbol T con raíz r. Observamos que el tamaño del máximo conjunto independiente de T puede incluir la raíz r (en cuyo caso no incluye a ninguno de sus hijos) o la excluye (en cuyo caso los hijos de r pueden ser miembros), Esto lleva a considerar como subproblemas determinar el máximo conjunto independiente de árboles, incluyendo y excluyendo a la raíz. Si llamamos $I(T)$ al tamaño del máximo conjunto independiente que incluye la raíz de T, y $E(T)$ al tamaño del máximo conjunto independiente que no incluye la raíz de T, tenemos que el tamaño del máximo conjunto independiente de T es máx$\{I(T), E(T)\}$, y podemos escribir las recurrencias:

$$I(T) = \begin{cases} 0 & \text{si } T \text{ es vacío} \\ 1 + \sum_{T' \text{ subárbol de } T} E(T') & \text{caso contrario} \end{cases}$$

$$E(T) = \begin{cases} 0 & \text{si } T \text{ es vacío} \\ \sum_{T' \text{ subárbol de } T} \text{máx}\{I(T'), E(T')\} & \text{caso contrario} \end{cases}$$

En este caso, para cada vértice debemos almacenar dos valores ($I(T)$ y $E(T)$), la estructura natural para hacerlo es el mismo árbol. Un orden de cálculo obvio de los valores según las recurrencias es en postorden.

Ejercicios

1. Derive la complejidad del algoritmo para determinar el orden óptimo de multiplicación de matrices, suponiendo que le dan a multiplicar n matrices. Compare con comparar todos los órdenes posibles.

2. 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 16.8.

Algoritmo 16.8: 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
end
function CalculeB(n)
    if n < 2 then
        return 2
    else
        return CalculeA(n − 1) + CalculeB(n − 2)
    end
end
```

a) Demuestre que el tiempo que demanda 16.8 para calcular a_n es exponencial.

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

c) Escriba una versión memoizada del algoritmo.

d) Escriba una versión basada en programación dinámica. ¿Qué datos deben retenerse?

3. 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?

4. ¿Cuál es la complejidad del algoritmo para resolver el problema de subsecuencia creciente más larga, programa 16.9?

5. Extienda el programa 16.9 para entregar una secuencia creciente más larga.

6. Escriba un programa que construya el árbol binario de búsqueda óptimo.

7. Decida una representación como estructura de datos de árboles con raíz, y dada esa estructura escriba un programa que dé un conjunto independiente de tamaño máximo para su árbol. ¿Cuál es la complejidad de su algoritmo?

8. 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).

9. 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.

10. 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 $\langle i_1, i_2, \ldots, i_n \rangle$. Los azulejos pueden repetirse, suponemos que hay suficientes de cada uno de ellos.

11. 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).

12. Considere una gramática de contexto libre G en forma normal de Chomsky (ver por ejemplo Hopcroft, Motwani y Ullman [4, capítulo 7]) y una palabra $\sigma = a_1 a_2 \ldots a_n$, donde queremos determinar si $\sigma \in \mathscr{L}(G)$. Esto puede hacerse mediante programación dinámica, registrando para el rango $a_i..a_j$ el conjunto de no-terminales que generan esa palabra, si el símbolo de partida pertenece al conjunto de no-terminales que generan $a_1..a_n = \sigma$, la respuesta es si (y podemos extraer una derivación).

a) Detalle este algoritmo (se le conoce como Cocke-Younger-Kasami, o CYK, por sus inventores, que independientemente plantearon esencialmente la misma idea [1, 5, 6]).

b) ¿Cuál es la complejidad de su algoritmo?

13. Considere el problema de asignación de tareas, pero ahora cada tarea tiene un valor, y nos interesa la colección de tareas de máximo valor. Formalmente, la tarea i comienza en el instante s_i y tiene duración ℓ_i (está activa en el intervalo abierto $[s_i, s_i + \ell_i)$), y tiene valor v_i. Buscamos una colección de tareas que no traslapan tal que la suma de los valores es máxima.

Bibliografía

[1] John Cocke and Jacob T. Schwartz: *Programming languages and their compilers: Preliminary notes*. Technical report, Courant Institute of Mathematical Sciences, New York University, 1970. (Second revised version).

[2] Jeff Erickson: *Algorithms, etc.* http://jeffe.cs.illinois.edu/teaching/algorithms, January 2015. Department of Computer Science, University of Illinois at Urbana-Champaign.

[3] Michal Forišek: *Towards a better way to teach dynamic programming*. Olympiads in Informatics, 9:45–55, 2015.

[4] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman: *Introduction to Automata Theory, Languages, and Computation*. Pearson/Addison-Wesley, third edition, 2007.

[5] Tadao Kasami: *An efficient recognition and syntax-analysis algorithm for context-free languages*. Technical Report 65-758, Air Force Cambridge Research Laboratories, 1965.

[6] Daniel H. Younger: *Recognition and parsing of context-free languages in time n^3*. Information and Control, 10(2):189–208, February 1967.

Clase 17

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 [2].

17.1. 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 17.1. *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:

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

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 17.1 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 altura 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.

17.2. 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 sea 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:

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

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.

17.3. Ahorrar espacio

Consideramos el problema de máxima subsecuencia común (sección 16.8). Vimos que el tiempo requerido por programación dinámica es $O(mn)$ al comparar secuencias de largos m y n, y que el espacio es también $O(mn)$. Es rutinario querer comparar secuencias de muchos miles de líneas, el espacio requerido se puede hacer prohibitivo. Si se analiza el algoritmo esbozado, solo se requieren algunas entradas del arreglo, no se necesita el arreglo completo. Basándose en esta observación, Hirschberg [1] construye un algoritmo que requiere espacio lineal. Suponemos palabras X e Y, de largos m y n, respectivamente. Partimos con el algoritmo de programación dinámica directo, algoritmo 17.1. Observamos que el algoritmo 17.1 para calcular la fila i del arreglo $\mathbf{L}$ solo hace referencia a la fila $i - 1$. Una pequeña modificación da el algoritmo 17.2, que calcula el vector $\tilde{\mathbf{L}}$,

Algoritmo 17.1: Máxima común subsecuencia por programación dinámica directa

```
procedure A(m, n, X, Y, L)
    L_{i,0} ← 0   [i = 0, ..., m]
    L_{0,j} ← 0   [j = 0, ..., n]
    for i ← 1 to m do
        for j ← 1 to n do
            if X_i = Y_j then
                L_{i,j} ← L_{i-1,j-1} + 1
            else
                L_{i,j} ← máx{L_{i,j-1}, L_{i-1,j}}
            end
        end
    end
end
```

Algoritmo 17.2: Largo de máxima común subsecuencia por programación dinámica ahorrando espacio

```
procedure B(m, n, X, Y, L̃)
    K_{1,j} ← 0   [j = 0, ..., n]
    for i ← 1 to n do
        K_{0,j} ← K_{1,j}   [j = 0, ..., n]
        for j ← 1 to n do
            if X_i = Y_j then
                K_{1,j} ← K_{0,j-1} + 1
            else
                K_{1,j} ← máx{K_{1,j-1}, K_{0,j}}
            end
        end
        L̃_j ← K_{1,j}   [i = 0, ..., n]
    end
end
```

donde $\tilde{L}_j = L_{m,j}$. Básicamente mantenemos en la matriz **K** los valores requeridos. Lo malo es que el algoritmo 17.2 solo entrega el largo, no tenemos cómo recuperar la subsecuencia máxima. Veremos cómo usar el algoritmo 17.2 sobre subpalabras para recuperar la máxima común subsecuencia en espacio lineal.

Llamemos X_{rs} a la subsecuencia $x_r, x_{r+1}, \ldots, x_s$. Para explicitar que estamos marchando en reversa, anotamos $\widehat{X}_{rs}$ con $r > s$ para $x_s, x_{s+1}, \ldots, x_r$. Sea $L^*_{i,j}$ el largo de la subsecuencia máxima entre $X_{i+1,m}$ e $Y_{j+1,n}$. Notamos que L_{ij} para $j = 0, \ldots, n$ son los largos máximos de subsecuencias comunes de $X_{1,i}$ y prefijos de Y. Podemos interpretarlos igualmente en términos de las palabras reversas y sufijos en $\widehat{X}$ e $\widehat{Y}$. Definamos:

$$M_i = \max_{0 \le j \le n} \{L_{i,j} + L^*_{i,j}\} \tag{17.1}$$

Haremos uso del siguiente teorema:

Teorema 17.1. *Para* $1 \le i \le m$ *es* $M_i = L_{m,n}$.

Demostración. Sea j tal que $M_i = L_{i,j} + L^*_{i,j}$. Sea tanbién S_{ij} una subsecuencia máxima común de X_{1i} e Y_{1j}; y sea S^*_{ij} una subsecuencia máxima común de $X_{i+1,m}$ e $Y_{j+1,n}$. Entonces $Z = S_{ij} \cdot S^*_{ij}$ es una subsecuencia común de X e Y, y su largo es M_i. O sea, $L_{mn} \geq M_i$.

Por otro lado, sea S_{mn} cualquier subsecuencia común más larga entre X e Y. Podemos escribir $S_{mn} = S_1 \cdot S_2$, donde S_1 es subsecuencia de X_{1i} para algún i y S_2 es subsecuencia de $X_{i+1,m}$. Hay un j tal que S_1 es subsecuencia de Y_{1j} y S_2 es subsecuencia de $Y_{j+1,n}$. Por las definiciones de L y L^*, $|S_1| \leq L_{ij}$ y $|S_2| \leq L^*_{ij}$. O sea:

$$\begin{aligned} L_{mn} &= |S_{mn}| \\ &= |S_1| + |S_2| \\ &\leq L_{ij} + L^*_{ij} \\ &\leq M_i \end{aligned}$$

Concluimos $L_{mn} = M_i$. □

Usamos el teorema 17.1 recursivamente para dividir el problema original en subproblemas similares hasta obtener problemas triviales (ver también el capítulo 22). Nuestro algoritmo final 17.3 construye la palabra Z que es la subsecuencia común más larga de X e Y. El algoritmo B toma

Algoritmo 17.3: Máxima común subsecuencia por programación dinámica ahorrando espacio

```
procedure C(m, n, X, Y, Z)
    if n = 0 then
        Z ← ϵ
    else if m = 1 then
        if ∃ j ≤ n, X_1 = Y_j then
            Z ← A_1
        else
            Z ← ϵ
        end
    else
        i ← ⌊m/2⌋
        Calcule L_{i,j} y L*_{i,j} para 0 ≤ j ≤ n
        B(i, n, X_{1i}, Y_{1,n}, L')
        B(m − 1, n, X̂_{n,i+1}, Ŷ_{n,1}, L'')
        Halle j tal que L_{ij} + L*_{ij} = L_{mn} usando el teorema 17.1:
        M ← máx_{0≤j≤n}{L'_j + L''_{n−j}}
        k ← j tal que M = L'_j + L''_{n−j}
        C(i, k, X_{1i}, Y_{1k}, Z')
        C(m − 1, n − k, X_{i+1,m}, Y_{k+1,n}, Z'')
        return Z' · Z''
    end
end
```

tiempo $O(mn)$ y usa espacio $O(m)$ (suponemos que X e Y y sus subpalabras se manejan con índices a principio y fin de espacio común).

El algoritmo C se ejecuta a lo más $2m-1$ veces, por inducción: Sea $m \leq 2^r$. Si $r = 0$, es $m = 1$ y hay $2^0 = 1$ llamada a C. Suponga ahora que para $m \leq 2^r = M$ hay $2m-1$ llamadas a C. Para

$m' \leq 2^{r+1} = 2M$, i a lo más toma el valor M, hay dos llamadas a C con m_1 y m_2 tales que $m_1 + m_2 = m'$ y con m_1 y m_2 ambos menores a M. Cada cual ejecutará $2m_1 - 1$ y $2m_2 - 1$ llamadas a C por inducción, agregando la llamada original a C da un total de $2m_1 - 1 + 2m_2 + 1 = 2m' - 1$, como se quería demostrar.

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] Daniel S. Hirschberg: *A linear space algorithm for computing maximal common subsequences*. Communications of the ACM, 18(6):341–343, June 1975.

[2] 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. Department of Computer Science and Engineering, University of New South Wales.

Clase 18

Máxima subsecuencia común, otra mirada

Discutimos el problema de máxima subsecuencia común (LCS, por el inglés *Longest Common Subsequence*) dando algoritmos basados en programación dinámica (sección 16.8, el algoritmo de Wagner-Fischer [18], aunque Navarro [15] halla múltiples autores independientes de la misma idea) y las variantes de Hirschberg [7] para ahorrar espacio (discutido en la sección 17.3). Otros algoritmos para este problema incluyen el de Hunt-Szymanski [9]. Una visión distinta es la de Heckel [5], quien intenta reconstruir diferencias fijándose en líneas únicas en ambos archivos, con el objetivo de obtener diferencias intuitivamente relevantes. Una discusión bastante completa de diferentes algoritmos es la de Hirschberg [6]. Aho, Hirschberg y Ullman [1] derivan cotas inferiores para el problema general, concluyen que en caso de solo comparar símbolos por igualdad y alfabeto ilimitado (la situación de comparar archivos, donde una línea representada por un *hash* es un símbolo) en el caso general comparar dos secuencias de largo N demanda tiempo $\Omega(N^2)$.

El problema halla aplicación práctica en muchas áreas, prominente en las cuales es comparar archivos (como describen Hunt y McIllroy [8] y Miller y Myers [13]). Los sistemas de control de versiones deben mostrar diferencias entre archivos de diferentes versiones, y muchos usan internamente las diferencias entre versiones para ahorrar espacio al almacenar la historia. Ejemplos tempranos son SCCS [16] y RCS [17]. Claro que en esta aplicación (y al comprimir datos) se suele usar además la idea de hacer referencia a copias anteriores de los mismos datos. El algoritmo básico es de Bentley y McIllroy [3], se estandarizó [10] y hay herramientas código abierto [12].

Para aplicaciones prácticas (archivos de muchos miles de líneas, secuencias de genes de muchos millones de pares de bases) los algoritmos presentados no son adecuados. Hunt y McIllroy [8] comparan algunas alternativas tempranas, y plantean técnicas para mejorar el rendimiento, como usar un *hash* (ver el capítulo 32 para la teoría del caso) en vez de la línea completa para acelerar las comparaciones y ahorrar espacio (esto es usar huellas digitales, tema que discutiremos en el capítulo 33). Concluyen que los algoritmos disponibles en ese entonces no hacen diferencia para los casos a su alcance (archivos de 3 500 líneas), el tiempo de ejecución está dominado por la lectura de los datos y manipulación de caracteres individuales. Advertimos, eso sí, que los distintos algoritmos tienen comportamientos diferentes, no siempre este es el más adecuado. Barabalucci et al [2] discuten varios escenarios e intentan formalizar «calidad» aplicada específicamente a diferencias entre documentos en XML.

Hoy el algoritmo empleado con mayor frecuencia es el de Myers [14], cuya variante debida a Wu, Manber, Myers y Miller [19] discutiremos a continuación. Bergroth, Hakonen y Raita [4] comparan

varios algoritmos, para situaciones en las cuales las secuencias son similares este algoritmo parece ser el mejor.

Suponemos dos textos, X e Y, de largos $M = |X|$ y $N = |Y|$, donde sin pérdida de generalidad asumimos $N \geq M$, con subsecuencia común máxima de largo L. La diferencia entre los largos es $\Delta = N - M$. Sea $D = M + N - 2L$, el largo de la secuencia de edición (operaciones agregar/eliminar) más corta (de X debemos eliminar $M - L$ símbolos, luego hay que insertar $N - L$ para crear Y). A este valor se le llama *distancia de Levenshtein* [11]. Una medida relacionada, que llamaremos P, es el número de símbolos eliminados de X:

$$\begin{aligned} P &= M - L \\ &= M - \frac{M + N - D}{2} \\ &= \frac{D - \Delta}{2} \end{aligned}$$

El algoritmo planteado tiene tiempo de ejecución $O(NP)$, lo que lo hace aplicable en situaciones comunes en que las palabras a comparar son muy similares. Se basa en una formulación intuitiva de grafo de edición, es un algoritmo de programación dinámica que aprovecha un criterio voraz para limitar las opciones a considerar.

18.1. Grafo de edición

Sean palabras $X = x_1 x_2 \ldots x_M$ e $Y = y_1 y_2 \ldots y_N$, de largos respectivos M y N con $N \geq M$. El *grafo de edición* para X e Y tiene nodos en los puntos de la cuadrícula (i, j) para $0 \leq i \leq M$ y $0 \leq j \leq N$. Los nodos se conectan mediante arcos dirigidos verticales, horizontales y diagonales, formando un grafo dirigido acíclico. Arcos horizontales conectan $(i-1, j)$ con (i, j), arcos verticales conectan $(i, j-1)$ con (i, j), hay un arco diagonal $(i-1, j-1)$ a (i, j) siempre que $x_i = y_j$. Nuestro problema es llegar desde la *fuente* $(0, 0)$ al *sumidero* (M, N) en este grafo. Un camino más corto usará el máximo posible de pasos diagonales, nos da la subsecuencia común más larga. Nuestra medida de distancia es el número de eliminaciones en X, que corresponden a pasos verticales. La figura 18.1 ilustra este grafo para $X = acbdeacbed$ e $Y = acebdabbabed$, ejemplo de [19]. En la figura 18.1 se remarca un camino de $(0,0)$ a $(12,10)$, correspondiente a $acbdabed = x_1 x_2 x_4 x_5 x_6 x_7 x_{11} x_{12} = y_1 y_2 y_3 y_4 y_6 y_8 y_9 y_{10}$. Una movida vertical corresponde a eliminar un símbolo de X, una horizontal inserta un símbolo de Y, un paso diagonal es retener el símbolo de ambos (parte del camino común más largo). En este ejemplo hay $P = 2$ eliminaciones de X y un total de $D = 6$ ediciones, la secuencia común más larga es de $L = 8$. Hay caminos más cortos alternativos, como se ven en la figura 18.1.

18.2. Preliminares

Considere el grafo de edición dibujado como grilla. Sea la diagonal k los puntos (i, j) tales que $i - j = k$. La fuente está sobre la diagonal 0, el sumidero sobre la diagonal $\Delta = N - M$. Tenemos diagonales numeradas $-M$ a N. El algoritmo presente considera los nodos en la franja de diagonales entre $-P$ y $\Delta + P$. Esto porque todo camino que salga de esta franja incluye más de P pasos verticales: si pasa por un vértice bajo $-P$, para llegar a él desde la fuente da más de P pasos verticales; si pasa por un vértice sobre $\Delta + P$ requiere más de P pasos para llegar al sumidero.

La *distancia de edición* a (i, j), anotada $D(i, j)$, es el costo (número total de inserciones y eliminaciones) de un camino óptimo de $(0,0)$ a (i, j) sobre la diagonal $k = i - j$. Si tal camino contiene v pasos verticales y h horizontales, el número de pasos no diagonales es $v + h = D(i, j)$, y debe terminar en la diagonal $h - v = k$. El número de pasos verticales, $V(i, j)$, en este camino es una cantidad

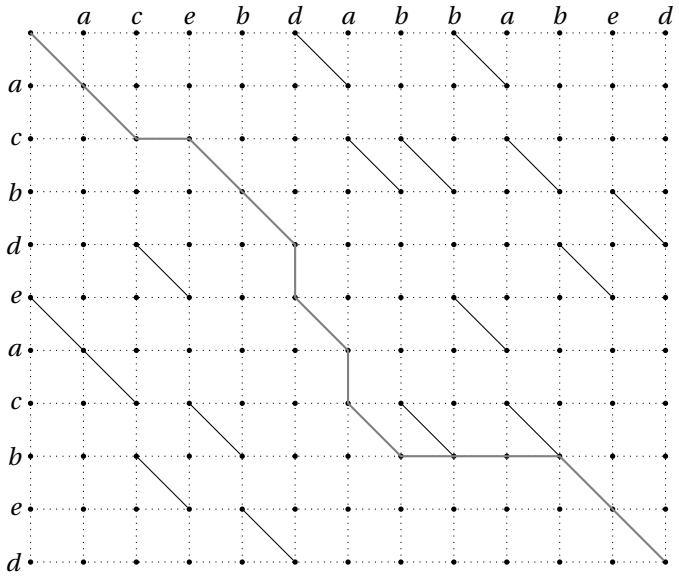


Figura 18.1 – Un grafo de edición y un camino óptimo

bien definida: es $V(i,j) = (D(i,j)+k)/2$. Sea la *distancia comprimida* como definida a continuación:

$$P(i,j) = \begin{cases} V(i,j) & \text{si } (i,j) \text{ está bajo la diagonal } \Delta \\ V(i,j) + (k-\Delta) & \text{caso contrario} \end{cases} \tag{18.1}$$

La distancia comprimida es la distancia vertical $V(i,j)$ más una cota inferior al número de pasos verticales para llegar de (i,j) al sumidero. Bajo la diagonal no se requieren pasos verticales, sobre la diagonal son al menos $k-\Delta$.

El algoritmo se centra en calcular los vértices más lejanos del origen en orden de distancia hasta hallar el sumidero. Sea el *d-punto más lejano sobre la diagonal k* el vértice sobre esa diagonal con valor de $D(i,j) = d$ y máximo i (o j). Llamaremos $\text{fd}(k,d)$ a la coordenada j:

$$\text{fd}(k,d) = \text{máx}\{j : D(j-k,j) = d\}. \tag{18.2}$$

El conjunto $\text{FD}(d)$ es la frontera de los vértices con distancia d. Usamos distancias comprimidas, con frontera $\text{FP}(p)$ con definición análoga, para $\text{fp}(k,p) = \text{máx}\{j : P(j-k,j) = p\}$:

$$\text{FP}(p) = \{(j-k,j) : y = \text{fp}(k,p) \wedge -p \le k \le p+\Delta\} \tag{18.3}$$

18.3. El algoritmo $O(NP)$

Calculamos el conjunto $\text{FP}(p)$ desde $\text{FP}(p-1)$ hasta que $(M,N) \in \text{FP}(p)$, con lo que conocemos P y también $D = \Delta + 2P$. Daremos primero una descripción informal, para luego formalizarlo en una recurrencia que demostramos correcta. Sea q_k el $(p-1)$-punto más lejano en la diagonal k (vale decir, el punto $(j-k,j)$ tal que $j = \text{fp}(k,p-1)$) y sea g_k el p-punto más lejano en la diagonal k. Suponga que ya conocemos $\text{FP}(p-1) = \{q_{-(p-1)}, q_{-(p-2)}, \ldots, q_{\Delta+(p-1)}\}$. Primero calculamos $g_{-p}, g_{-(p-1)}, \ldots, g_{\Delta-1}$ en este orden. Para calcular g_k de g_{k-1} y q_{k+1} procedemos como sigue. Sea a el vértice inmediatamente a la derecha de g_{k-1} y b el vértice inmediatamente debajo de q_{k+1} (ver la

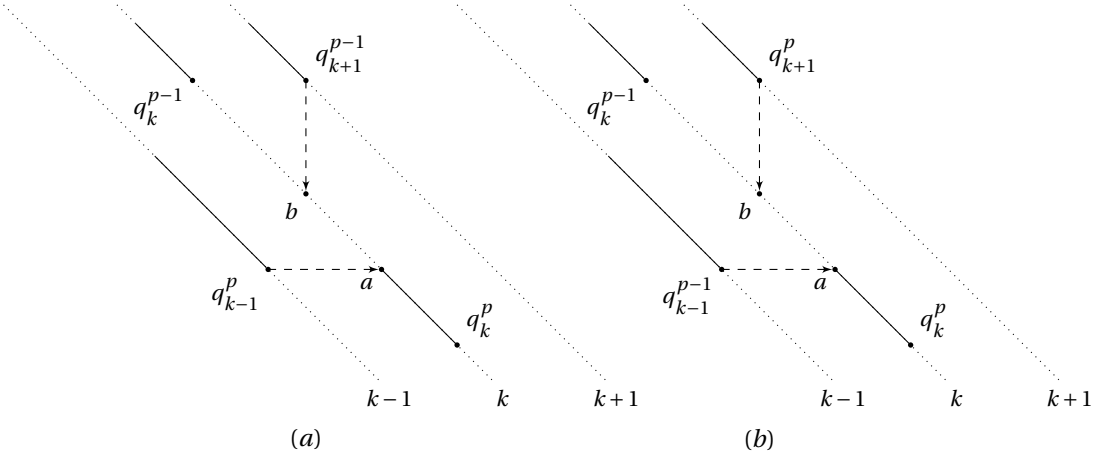


Figura 18.2 – Obtener FP(p) desde FP($p-1$)

figura 18.2). Ambos están sobre la diagonal k. Del vértice con máxima coordenada j seguimos arcos diagonales hasta un vértice sin arco diagonal saliente (o llegamos al borde inferior de la grilla). Este es el vértice g_k, cosa que demostraremos en el lema 18.1. Luego se obtienen $g_{\Delta+p}, g_{\Delta+(p-1)}, \ldots, g_{\Delta+1}$, esta vez usando q_{k-1} y g_{k+1} para calcular g_k. Finalmente se calcula g_Δ de $g_{\Delta-1}$ y $g_{\Delta+1}$.

El cálculo de FP(p) a partir de FP($p-1$) se formaliza en el lema 18.1, que da una recurrencia para $\text{fp}(k,p)$ en términos de la coordenada j de puntos más lejanos calculados previamente. Sea $\text{snake}(k,j)$ la coordenada j del punto más lejano sobre la diagonal k que puede alcanzarse desde $(j-k,j)$ atravesando arcos diagonales. Formalmente:

$$\text{snake}(k,j) = \text{máx}\{r : x_{j+1-k} \ldots x_{r-k} = y_{j+1} \ldots y_r\}$$

Que la recurrencia es correcta depende del manejo de casos límite: $p=0$, $k=-p$ y $k=\Delta+p$. Estas se resuelven limpiamente definiendo $\text{fp}(k,p)=-1$ siempre que $p<0$ o $k \notin [-p, \Delta+p]$.

Lema 18.1.

$$\text{fp}(k,p) = \begin{cases} \text{snake}(k, \text{máx}\{\text{fp}(k-1,p)+1, \text{fp}(k+1,p-1)\}) & k \in [-p, \Delta-1] \\ \text{snake}(k, \text{máx}\{\text{fp}(k-1,p)+1, \text{fp}(k+1,p)\}) & k = \Delta \\ \text{snake}(k, \text{máx}\{\text{fp}(k-1,p-1)+1, \text{fp}(k+1,p)\}) & k \in [\Delta+1, \Delta+p] \end{cases}$$

Demostración. Demostramos el caso $k<\Delta$, los demás casos son similares. Sea g el p-punto más lejano en la diagonal $k-1$, y sea q el $(p-1)$-punto más lejano en la diagonal $k+1$. Sean a el vértice inmediatamente a la derecha de g, b el vértice inmediatamente inferior a q, y d el vértice más lejano alcanzable desde el más lejano de a y b por arcos diagonales. La coordenada j de a es $\text{fp}(k-1,p)+1$, la de b es $\text{fp}(k+1,p-1)$, la de d es la dada por el primer caso del lema.

La figura 18.3 ilustra los casos en que a está arriba de b (vale decir, $\text{fp}(k+1,p)+1 \le \text{fp}(k,p-1)$) y el caso en que b está sobre a. Trataremos solo el primer caso, el otro es similar. El valor P de d tiene que ser p dado que hay un camino a d con distancia comprimida p (el que pasa por q y b), si hubiese uno más corto, el vértice c de la figura tendría un valor de P menor a $p-1$, contradiciendo la elección de q. Debemos demostrar además que d es el más lejano posible. Un camino de distancia p más largo no puede pasar por d, eso contradiría la elección de d. Eso quiere decir que pasa por un vértice a distancia $p-1$ en la diagonal $k-1$ bajo g o uno a distancia $p-1$ sobre la diagonal k bajo q, contradiciendo las elecciones de g y q, respectivamente. O sea, tal camino no existe y d es el más lejano. □

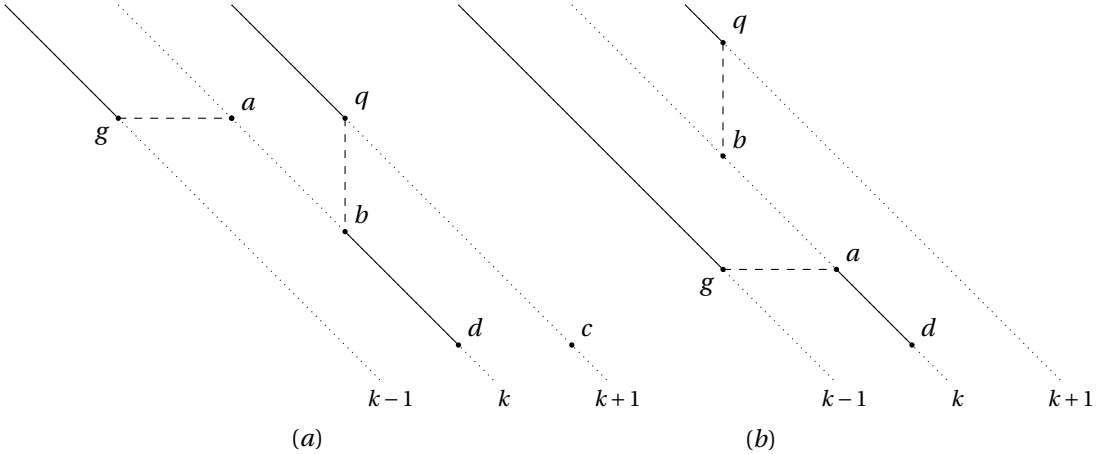


Figura 18.3 – Los dos casos del lema 18.1

El algoritmo 18.1 viene directamente del lema 18.1. Note que para $k \in [-p, \Delta - 1]$ se usan solo los valores $\text{fp}[k+1, p-1]$ y $\text{fp}[k-1, p]$ al calcular $\text{fp}[k, p]$. Podemos almacenar fp en un único arreglo con índice k si calculamos en orden de k creciente en este rango. Análogamente, si $k \in [\Delta + 1, \Delta + p]$ conviene calcular en orden de k decreciente. El algoritmo 18.1 toma tiempo $O((M+N)P)$: el ciclo externo se ejecuta P veces, y es claro que para cada diagonal (de largo acotado por $M+N$) cada nodo se considera a lo más una vez.

18.4. Obtener la subsecuencia común más larga

Nuevamente nuestro algoritmo solo da el largo buscado, no la secuencia común que buscamos (o las operaciones de edición que transforman X en Y). Registrar los valores necesarios para poder reconstruir directamente el camino óptimo requiere espacio $O(NP)$, en el peor caso $O(NM)$. Esto es inaceptable. Siguiendo la pista de Hirschberg [7], podemos reducir el espacio requerido a $O(M+N)$ a costa de procesamiento adicional.

La idea es aplicar el algoritmo 18.1 desde ambos extremos. Identificamos así la serpiente central de una subsecuencia común más larga, lo que divide el problema en determinar recursivamente subsecuencias comunes más largas llegando a ambos extremos de la serpiente identificada. O sea, usamos dividir y conquistar (ver el capítulo 22).

Es claro que la estrategia planteada deberá hallar una serpiente central en aproximadamente $P/2$ pasos. Podemos plantear la recurrencia para el tiempo $T(R,P)$ que toma el algoritmo para hallar las serpientes. Acá $R = M+N$ y P es el número de pasos verticales. Para constantes α, β apropiadas, y $R_1 + R_2 \le R$ tenemos:

$$T(R,P) \le \begin{cases} \alpha RP + T(R_1, \lceil P/2 \rceil) + T(R_2, \lfloor P/2 \rfloor) & P > 1 \\ \beta R & P \le 1 \end{cases} \tag{18.4}$$

Como $\lceil P/2 \rceil \le 2P/3$ siempre que $P \ge 2$, una simple inducción demuestra que $T(R,P) \le 3\alpha RP + \beta R$. O sea, esta división recursiva sigue tomando tiempo $O(NP)$. Se requieren dos arreglos de tamaño $M+N$, uno en cada dirección. Pero el espacio se requiere antes de la recursión, puede reutilizarse.

Algoritmo 18.1: El algoritmo de Wu-Manber-Myers-Miller

```
function LCS(X, Y, M, N)
    fp[−(M + 1), ..., (N + 1)] ← −1
    Δ ← N − M
    p ← −1
    repeat
        p ← p + 1
        for k ← −p to Δ − 1 do
            fp[k] ← snake(k, máx{fp[k − 1] + 1, fp[k + 1]})
        end
        for k ← Δ + p downto Δ + 1 do
            fp[k] ← snake(k, máx{fp[k − 1] + 1, fp[k + 1]})
        end
        fp[Δ] ← snake(Δ, máx{fp[Δ − 1] + 1, fp[Δ + 1]})
    until fp[Δ] = N
    return M − p)
end

function snake(k, j)
    i ← j − k
    while j < M ∧ i < N ∧ X[j + 1] = Y[i + 1] do
        i ← i + 1;   j ← j + 1
    end
    return j
end
```

Bibliografía

[1] A. V. Aho, D. S. Hirschberg, and J. D. Ullman: *Bounds on the complexity of the longest common subsequence problem.* Journal of the ACM, 23(1):1–12, January 1976.

[2] Giole Barabucci, Paolo Ciancarini, Angelo Di Ioirio, and Fabio Vitali: *Measuring the quality of diff algorithms: A formalization.* Computer Standards & Interfaces, 46:52–65, May 2016.

[3] Jon Bentley and Douglas McIllroy: *Data compression using long common strings.* In *Proceedings of the Data Compression Conference*, pages 287–295, Snowbird, UT, USA, March 1999. IEEE.

[4] L. Bergroth, H. Hakonen, and T. Raita: *A survey of longest common subsequence algorithms.* In *Proceedings Seventh International Symposium on String Processing and Information Retrieval*, pages 39–48, 2000.

[5] Paul Heckel: *A technique for isolating differences between files.* Communications of the ACM, 21(4):264–268, April 1978.

[6] D. S. Hirschberg: *Serial computations of Levenshtein distances.* In Alberto Apostolico and Zvi Galill (editors): *Pattern Matching Algorithms*, pages 123–141. Oxford University Press, 1997.

[7] Daniel S. Hirschberg: *A linear space algorithm for computing maximal common subsequences.* Communications of the ACM, 18(6):341–343, June 1975.

[8] James W. Hunt and M. Douglas McIllroy: *An algorithm for differential file comparison.* Computing science technical report 41, Bell Laboratories, June 1976.

[9] James W. Hunt and Thomas G. Szymanski: *A fast algorithm for computing longest common subsequences.* Communications of the ACM, 20(5):350–353, May 1977.

[10] D. Korn, J. MacDonald, J. Mogul, and K. Vo: *The VCDIFF generic differencing and compression data format.* Request for comments 3284, Internet Engineering Task Force, June 2002.

[11] Vladimir I. Levenshtein: *Binary codes capable of correcting deletions, insertions, and reversals.* Soviet Physics Doklady, 10(8):707–710, February 1966.

[12] Joshua P. MacDonald: *xdelta: Open-source binary diff, differential compression tools.* http://xdelta.org, January 2016. Version 3.11.

[13] Webb Miller and Eugene W. Myers: *A file comparison program.* Software – Practice & Experience, 15(11):1025–1040, November 1985.

[14] Eugene W. Myers: *An $O(ND)$ difference algorithm and its variations.* Algorithmica, 1(1-4):251–266, November 1986.

[15] Gonzalo Navarro: *A guided tour to approximate string matching.* ACM Computing Surveys, 33(1):31–88, March 2001.

[16] Marc J. Rochkind: *The Source Code Control System.* IEEE Transactions on Software Engineering, SE-1(4):364–370, December 1975.

[17] Walter F. Tichy: *RCS – a system for version control.* Software – Practice & Experience, 15(7):637–654, July 1985.

[18] Robert A. Wagner and Michael J. Fischer: *The string-to-string correction problem.* Journal of the ACM, 21(1):168–173, January 1974.

[19] Sun Wu, Udi Manber, Gene Myers, and Webb Miller: *An $O(NP)$ sequence comparison algorithm.* Information Processing Letters, 35(6):317–323, September 1990.

Clase 19

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. Una discusión lúcida es de Hetland [2, capítulo 4]; otra opción es Smid [3, capítulo 4] quien discute varios ejemplos de definiciones por inducción que llevan a programas recursivos nada obvios.

19.1. Torres de Hanoi

Descripción del problema: Hay 64 placas redondas ubicadas de mayor a menor (figura 19.1). Solo se puede mover una placa a la vez y nunca se debe ubicar una placa mayor sobre una placa menor. Lo que buscamos es mover todas las placas de A a C, posiblemente usando B como auxiliar.

19.1.1. Solución (recursiva)

Una solución (recursiva) es como se muestra en la figura 19.2. Esto es solución porque traduce el problema de mover n piezas a mover $n-1$ recursivamente, luego mover 1 (trivial, figura 19.2), luego mover $n-1$ recursivamente.

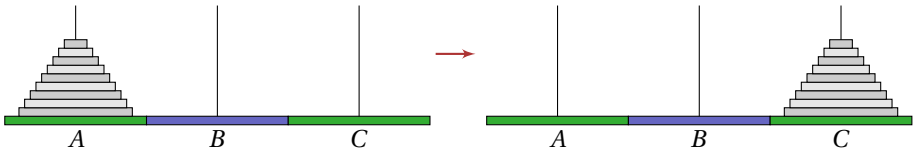


Figura 19.1 – Mover las placas desde la plataforma A a la C.

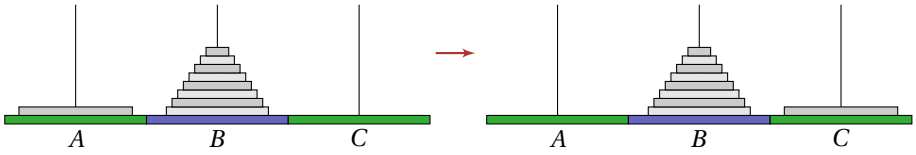


Figura 19.2 – Solo nos queda mover el último disco (más grande) de A a C directamente.

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 C (con B de «apoyo»):

- Movemos las k piezas superiores de A a B.
- Movemos la pieza inferior de A a C.
- Movemos las k piezas de B a C (con A de «apoyo»).

□

Pseudocódigo es el algoritmo 19.1. Usamos las variables *src* para el origen, *dst* para el destino y *tmp* para el auxiliar (libre). Estas variables valen A, B o C, según sea. Por la estructura de la solución,

Algoritmo 19.1: 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
end
```

siempre cumplimos con las restricciones.

¿Cuántas movidas se requieren? Sea $T(n)$ el número de movidas para transferir n platos usando el algoritmo 19.1.

$$T(0) = 0$$
$$T(n+1) = 2T(n) + 1, \qquad n \geq 0$$

Técnicas estándar de solución de recurrencias entregan:

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

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

Demostración. **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, este es óptimo.

19.2. Mergesort

Queremos ordenar $A[1,\ldots,n]$. Nuestro pseudocódigo para mergesort es el del algoritmo 19.2. De la misma forma que lo hicimos con las torres de Hanoi: ¿Cuántas operaciones se requieren? Sea

Algoritmo 19.2: 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
end
```

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

$$M(0) = M(1)$$
$$M(n) = M(\left\lfloor \frac{n}{2} \right\rfloor) + M(\left\lceil \frac{n}{2} \right\rceil) + n \tag{19.1}$$

Discutiremos recurrencias como esta en el capítulo 22, donde concluiremos que:

$$M(n) = O(n \log n)$$

Dentro de un factor constante, mergesort es óptimo (vimos en el apunte de *Fundamentos d Informática* [1] que si solo se comparan elementos para ordenar n elementos se requieren $\Omega(n \log n)$ comparaciones).

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.

Bibliografía

[1] Horst H. von Brand: *Fundamentos de Informática*. git://csrg.inf.utfsm.cl/vonbrand/Ramos/trainwreck, Septiembre 2017. Versión 0.84.

[2] Magnus Lie Hetland: *Python Algorithms: Mastering Basic Algorithms in the Python Language*. Apress, second edition, 2014.

[3] Michiel Smid: *Discrete structures for computer science: Counting, recursion, and probability*. http://cglab.ca/~michiel/DiscreteStructures/DiscreteStructures.pdf, July 2018.

Clase 20

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 20.1 (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 20.1 muestra los casilleros que amenaza una reina en el ajedrez. Un problema clásico es determinar si se pueden ubicar ocho reinas en el

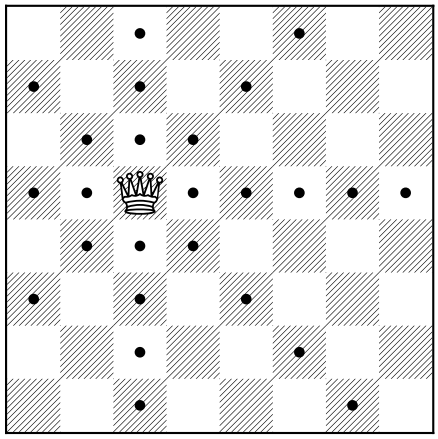


Figura 20.1 – Los casilleros amenazados por una reina

tablero de manera que ninguna pueda amenazar 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 20.2. Vemos que las filas y diagonales amenazadas por estas tres restringen

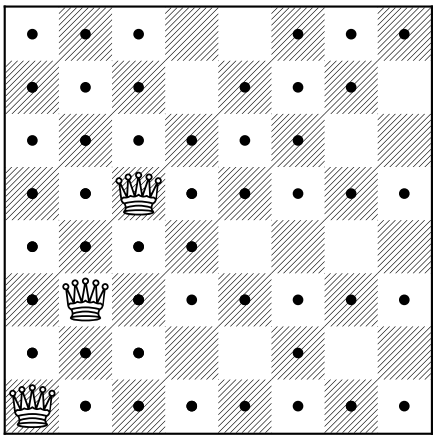


Figura 20.2 – Configuración con tres reinas

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 20.1. La figura 20.3 muestra una de las 92 soluciones.

```
#!/usr/bin/env 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 = c + 7 - r
dd = [True for i in range(15)]         # Diagonal i = c + r free

def solve(c):
    global solutions

    if c == 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[c + r] and du[c + 7 - r]:
                queen[c] = r
                rfree[r] = dd[c + r] = du[c + 7 - r] = False
                solve(c + 1)
                rfree[r] = dd[c + r] = du[c + 7 - r] = True

solutions = 0
solve(0)

print()
print(solutions, "solutions")
```

Listado 20.1 – Ocho reinas en Python

20.1. Formalizando backtracking

Nos basamos en la presentación sobre backtracking de Steven Skiena [3]. 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 20.1. La idea de backtracking es usarlo cuando no tengamos subproblemas 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).

20.2. Sudoku

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 20.4. Estrategias para elegir siguiente casillero a llenar:

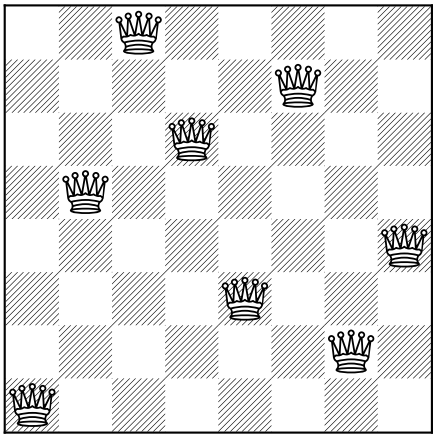


Figura 20.3 – Una solución para el problema de 8 reinas.

Algoritmo 20.1: 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
end
```

- Al azar/primero libre.
- Más restringido.

Estrategias para podar:

- Cuenta local (revisar si quedan opciones fila/columna/cuadrante)
- *Look ahead* (revisar si quedan casilleros sin opciones)

Skiena [4] reporta resultados del cuadro 20.1 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 20.4 (tiene 17 pistas, es de los con menos pistas que tiene una única solución). El programa que obtiene estos valores es [2]. En 2012 McGuire, Tugeman y Civario [1] demostraron mediante una búsqueda exhaustiva que tomó 7,1 millones de horas-núcleo en un supercomputador que no hay problemas con solo 16 pistas con solución única.

		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 20.4 – Sudoku muy difícil

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 20.1 – Rendimiento de variantes de backtracking en Sudoku

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.

2. Determine si es posible que un caballo del ajedrez visite exactamente una vez cada una de las casillas del tablero.

Bibliografía

[1] Gary McGuire, Bastian Tugemann, and Gilles Civario: *There is no 16-clue Sudoku: Solving the Sudoku minimum number of clues problem*. http://arxiv.org/abs/1201.0749, September 2012.

[2] Steven S. Skiena: *A backtracking program to solve Sudoku*. http://www3.cs.stonybrook.edu/~skiena/algorist/book/programs/sudoku.c, August 2006. Department of Computer Science, Stony Brook University.

[3] Steven S. Skiena: *Backtracking*. http://www3.cs.stonybrook.edu/~algorith/video-lectures/2007/lecture15.pdf, 2007. Department of Computer Science, Stony Brook University.

[4] Steven S. Skiena: *The Algorithm Design Manual*. Springer, second edition, 2008.

Clase 21

Búsqueda en Grafos

Hay una variedad de situaciones en las que se debe explorar algún grafo en busca de un vértice (nodo) meta. Antes se han discutido técnicas básicas, como búsqueda/recorrido de árboles, búsqueda en profundidad y a lo ancho en grafos, y algoritmos como los de Dijkstra y Bellman-Ford. Pero estos tienen la desventaja de considerar todos los nodos, interesan técnicas que elijan inteligentemente los nodos a revisar para minimizar el trabajo.

Las técnicas son variantes de *backtracking* (ver el capítulo 20). 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].

21.1. 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 21.1, 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 grafo dirigido acíclico, DAG). Una aplicación es resolver el problema del vendedor viajero (TSP, un conocido problema NP-completo). El problema contempla un grafo $G = (V, E)$ y un peso $w\colon E \to \mathbb{R}$, y busca un ciclo hamiltoniano de G (un camino simple que visita cada vértice exactamente una vez) de peso mínimo. Modela el caso de un vendedor viajero que debe visitar a sus clientes, partiendo de la oficina y volviendo a ella, con mínimo costo.

La idea es partir con un vértice cualquiera, y a partir de allí ir extendiendo el camino un arco a la vez. Analizando el algoritmo 21.1 vemos que $f(x)$ debe representar un valor que puede lograrse (en nuestro ejemplo, el costo del camino que se considera hasta x y un camino posible a través del resto del grafo, como el que resulta del algoritmo voraz que visita cada vez el vértice más cercano),

Algoritmo 21.1: Esquema de *Branch and Bound*

```
function BB
    x ← initial guess
    B ← f(x)
    enqueue(Q, x)

    while Q not empty do
        x ← dequeue(Q)
        if f(x) < B then
            B ← f(x)
        end
        foreach v ∈ Γ(x) do
            if g(v) < B then
                enqueue(Q, v)
            end
        end
    end
    return B
end
```

Asimismo, $g(x)$ debe ser una cota inferior al costo del camino que lleva del último vértice en el camino parcial al vértice inicial. Podemos usar para $g(x)$ el costo del árbol recubridor mínimo del grafo que resulta de eliminar los vértices intermedios ya visitados (el camino óptimo es un árbol recubridor de este grafo, con la particularidad que es un camino simple, su costo no puede ser menor al del árbol recubridor mínimo).

21.2. El algoritmo A^*

Un algoritmo genérico de búsqueda es A^*, desarrollado inicialmente para planificar rutas de robots moviéndose en un ambiente con obstáculos. En el algoritmo, vértices son posibles posiciones del robot, que comienza su viaje en s, y debe llegar a la más cercana de las posiciones en T (pueden haber varios destinos alternativos). Desde la posición r pueden alcanzarse directamente las posiciones en $\Gamma(r)$; si $u \in \Gamma(r)$, conocemos el costo $c(r,u)$ para moverse directamente de r a u. Los proponentes de A^*, 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 se cumple que para una constante $\delta > 0$ siempre es $c(e) \geq \delta$, 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; suponemos además que cuando Γ nos entrega los descendientes de v simultáneamente entrega los costos desde el nodo v 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 $\min_{t\in T}\{h(v,t)\}$

($h(v)$ es el costo del camino óptimo desde v a un destino).

Podemos imaginar muchos algoritmos que expanden vértices y exploran los caminos que nacen de ellos, podando la búsqueda. 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. Nos interesan algoritmos admisibles y eficientes.

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 21.2, que contempla una cola de prioridad Q. Diremos que nodos en Q (al igual que nodos aún no considerados) están *abiertos*, y marcaremos ciertos nodos como *cerrados* para no considerarlos nuevamente. En realidad nos interesa el camino

Algoritmo 21.2: El algoritmo A^*

```
function A*(G, s, T)
    enqueue(Q, s, f̂(s))
    while Q not empty do
        v ← dequeue(Q)
        Mark v closed
        if v ∈ T then
            return v
        end
        foreach 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
end
```

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 alcanzado hacia atrás para reconstruir el camino buscado.

21.2.1. 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 21.2.

Podemos escribir f como una suma:

$$f(v) = g(v) + h(v) \tag{21.1}$$

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, definimos funciones $\widehat{h}$ apropiadas, por el momento demostramos que si $\widehat{h}(v) \leq h(v)$, el algoritmo 21.2 es admisible.

Lema 21.1. *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), tome $s = v'$, 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 (el último nodo cerrado de P), donde $v^* \neq v$ porque v está abierto. Sea v' el sucesor de v^* en P. Entonces:

$$\begin{aligned}
&\widehat{g}(v') \leq \widehat{g}(v^*) + c(v^* v') && \text{por definición de } \widehat{g} \\
&\widehat{g}(v^*) = g(v^*) && \text{porque } v^* \in \Delta \\
&g(v') = g(v^*) + c(v^* v') && \text{dado que } P \text{ es óptimo}
\end{aligned}$$

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 21.2. *Suponga que para todo v es $\widehat{h}(v) \leq h(v)$, 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 21.1, 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 21.3. *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 21.2 habrá un nodo abierto v'en un camino óptimo tal que $\widehat{f}(v') \leq f(s) < f(v)$. El algoritmo elegirá 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 $\widehat{f}(t) = \widehat{g}(t) > f(s)$. Por el corolario 21.2 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.

□

21.2.2. 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*:

$$h(u,v) + \widehat{h}(u) \geq \widehat{h}(v) \tag{21.2}$$

La condición (21.2) 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 21.4. *Suponga que se cumple la condición de monotonía* (21.2)*, 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 21.1, 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 (21.2) 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. □

21.3. 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.

21.3.1. 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 21.3, 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).

Algoritmo 21.3: 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
        foreach child of node do
            v ← minmax(child, depth −1, B)
            best ← max(best, v)
        end
    else
        best ← +1
        foreach child of node do
            v ← minmax(child, depth −1, A)
            best ← min(best, v)
        end
    end
    return best
end
```

21.3.2. 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 21.4 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 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 21.4 no especifica

Algoritmo 21.4: 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
        foreach 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
        foreach 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
end
```

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 22

Dividir y Conquistar

Una de las mejores estrategias para diseñar algoritmos. Muchos de los algoritmos importantes se basan en esto, y su análisis presenta problemas matemáticos interesantes.

La idea general es dado un problema grande, reducirlo a varios problemas menores del mismo tipo, y combinar resultados.

Ejemplo 22.1 (Merge Sort)**.** Ya discutido en el capítulo 19. Para ordenar N elementos:

- Dividir en «mitades» de $\lfloor \frac{N}{2} \rfloor$ y $\lceil \frac{N}{2} \rceil$ elementos.
- Ordenarlas recursivamente.
- Intercalar resultados.

Ejemplo 22.2 (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 22.3 (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:

$$A \cdot B = 10^{2n} ac + 10^n (ad + bc) + bd \tag{22.1}$$

Notando que:

$$(a+b)\cdot(c+d) = ac + ad + bc + bd - (ad + bc) + ac + bd$$

Podemos calcular los coeficientes de (22.1) con 3 (no 4) multiplicaciones (este truco se le atribuye a Gauß, quien lo empleaba para multiplicar números complejos):

$$ac$$
$$bd$$
$$(a+b)(c+d) - ac - bd$$

Ejemplo 22.4. Otra aplicación 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 $2^n \times 2^n$.

22.1. 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 de 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 $f(n)$:

$$t(n) = at(n/b) + f(n) \qquad t(1) = t_1 \tag{22.2}$$

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:

$$n = b^k \qquad t(b^k) = T(k) \qquad k = \log_b n$$
$$T(k+1) = aT(k) + f(b^{k+1})$$

Definimos la función generatriz ordinaria:

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

Por propiedades:

$$\begin{aligned}\frac{g(z)-t_1}{z} &= ag(z)+\sum_{r\geq 0} f(b^{r+1})z^r\\ g(z) &= \frac{t_1}{1-az}+\frac{z}{1-az}\sum_{r\geq 0} f(b^{r+1})z^r\\ [z^k]g(z) &= t_1 a^k+\sum_{0\leq r\leq k-1} a^{k-1-r}f(b^{r+1})\\ &= t_1 a^k + a^k\sum_{1\leq r\leq k} a^{-r}f(b^r)\end{aligned}$$

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:

$$t(n) = t_1 n^{\log_b a} + n^{\log_b a}\sum_{1\leq r\leq \log_b n} a^{-r}f(b^r)$$

Si la segunda suma converge cuando n tiende a infinito, vale decir, si $f(n) = O(n^c)$ para algún $c < \log_b a$, es determinante el factor:

$$t(n) = \Theta(n^{\log_b a})$$

En caso que la suma no converja, dominará el segundo término. Analicemos primeramente el caso en que $f(n) = \Omega(n^c)$, donde $c > \log_b a$. Es central la suma:

$$\sum_{1\leq r\leq \log_b n} a^{-r}f(b^r)$$

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 una serie geométrica convergente, que podemos acotar por su límite:

$$\begin{aligned}\sum_{1\leq r\leq \log_b n} a^{-r}f(b^r) &= \frac{1}{a^{\log_b n}}\sum_{1\leq r\leq \log_b n} a^r f(n/b^r)\\ &\leq \frac{1}{an^{\log_b a}}\sum_{1\leq r\leq \log_b n} k^r f(n)\\ &< \frac{1}{an^{\log_b a}}\frac{f(n)}{1-k}\end{aligned}$$

Esto con la cota inferior anterior se resume en:

$$t(n) = \Theta(f(n))$$

El caso intermedio de mayor interés es $f(n) = \Theta(n^{\log_b a}\log^\alpha n)$:

$$\begin{aligned}\sum_{1\leq r\leq \log_b n} a^{-r}f(b^r) &= \sum_{1\leq r\leq \log_b n} a^{-r}\Theta(a^r r^\alpha)\\ &= \sum_{1\leq r\leq \log_b n}\Theta(r^\alpha)\\ &= \Theta\left(\sum_{1\leq r\leq \log_b n} r^\alpha\right)\end{aligned}$$

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 22.1 (Teorema Maestro)**.** *La recurrencia:*

$$t(n) = at(n/b) + f(n) \qquad t(1) = t_1 \tag{22.3}$$

con constantes $a \geq 1$, $b > 1$, $f(n) > 0$ tiene solución:

$$t(n) = \begin{cases} \Theta(n^{\log_b a}) & f(n) = O(n^c) \text{ para } c < \log_b a \\ \Theta(n^{\log_b a}) & f(n) = \Theta(n^{\log_b a} \log^{\alpha} n) \text{ con } \alpha < -1 \\ \Theta(n^{\log_b a} \log\log n) & f(n) = \Theta(n^{\log_b a} \log^{\alpha} n) \text{ con } \alpha = -1 \\ \Theta(n^{\log_b a} \log^{\alpha+1} n) & f(n) = \Theta(n^{\log_b a} \log^{\alpha} n) \text{ con } \alpha > -1 \\ \Theta(f(n)) & f(n) = \Omega(n^c) \text{ con } c > \log_b a \text{ y } af(n/b) < kf(n) \text{ para } n \text{ grande con } k < 1 \end{cases}$$

Nuestros ejemplos se resumen en el cuadro 22.1. 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 22.1 – Complejidad de nuestros ejemplos

teorema de Akra-Bazzi, del que reportamos la versión de Leighton [5].

Teorema 22.2 (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 22.2, 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 23

Diseño de Algoritmos

Para demostrar cómo se aplican las técnicas de diseño descritas, discutiremos un problema planteado por Bentley [1]. Dado el arreglo $a[n]$, hallar la máxima suma de un rango:

$$\max_{i,j}\left\{\sum_{i\leq k\leq j} a[k]\right\} \tag{23.1}$$

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.

23.1. Algoritmo ingenuo

La solución obvia, traducción directa de la especificación dada por la ecuación (23.1), es la mostrada en el programa C del listado 23.1. La complejidad del algoritmo 1 es $O(n^3)$. Lo que buscamos es mejorarlo.

23.2. No recalcular sumas

Hay dos ideas sencillas para evitar recalcular sumas.

23.2.1. Extender sumas

En vez de calcular la suma del rango cada vez, extendemos la suma anterior. Esto da el programa del listado 23.2. La complejidad del algoritmo 2 es $O(n^2)$.

23.2.2. Sumas cumulativas

Una manera de manejar rangos es usar sumas cumulativas, y obtener el valor para el rango restando. Esta idea da el listado 23.3. La complejidad del algoritmo 3 es $O(n^2)$. Comparado a nuestro algoritmo original resulta una mejora, pero no respecto a la segunda variante.

```
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 23.1 – Algoritmo 1: Versión ingenua

```
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 23.2 – Algoritmo 2: Evitar recalcular sumas

23.3. Dividir y Conquistar

Aplicar la estrategia vista la clase pasada lleva a la figura 23.1a. Pero debemos también considerar que el rango con máxima suma esté a hojarcadas, cruzando el punto central, como en la figura 23.1b. El algoritmo es el dado en el listado 23.4. Usando el teorema maestro (teorema 22.1), para el algoritmo 4 son $a = 2$, $b = 2$ y $f(n) = O(n)$, por lo tanto la complejidad es $O(n \log n)$.

23.4. 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 23.2. Gries [3] deriva el algoritmo sistemáticamente y demuestra su correctitud. Esto da el algoritmo 5,

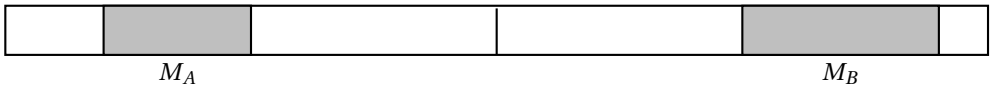
```
double MaxSum(double a[], int n)
{
        double CumArray[n + 1]; /* Sum of a[0] to a[i - 1] */
        double MaxSoFar;

        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 23.3 – Algoritmo 3: Usar arreglo cumulativo



(a) Máximos en las mitades

(b) Máximo al medio

Figura 23.1 – Dividir y conquistar

del listado 23.5

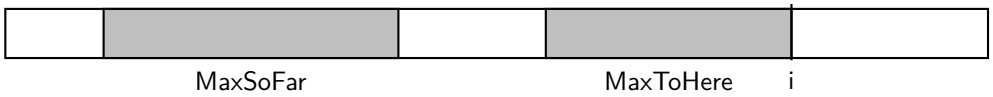


Figura 23.2 – Extender la solución

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 23.1 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 23.4 – Algoritmo 4: Dividir y conquistar

23.5. Mayoría de una secuencia

Se dice que un elemento de una secuencia de n elementos es *mayoría* si es más de $\lfloor n/\rfloor$ elementos de ella. Nuestro problema es, dada una secuencia que sabemos tiene una mayoría, determinar cuál es ese elemento.

Algoritmos obvios para resolver este problema son ordenar la secuencia y ver el elemento del medio (es claro que si cierto elemento es al menos la mitad de la secuencia ordenada, estará al

```
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 23.5 – 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 23.1 – Comparativa de Bentley [1] entre las variantes

medio, independiente de dónde comience la repetición), o ir contabilizando los elementos (por ejemplo, en una tabla *hash* o un árbol) e ir contabilizando las veces que cada uno aparece.

La primera idea demora $O(n\log n)$, y exige tener la secuencia completa a la mano. La segunda puede procesar la secuencia conforme llega, usando tablas *hash*, demora $O(n)$, pero requiere $O(n)$ espacio adicional.

Boyer y Moore [2] propusieron un algoritmo que lee la secuencia una sola vez y usa una cantidad mínima de espacio adicional. El algoritmo es el 23.1. La idea es ir contabilizando elementos iguales

Algoritmo 23.1: Algoritmo MJRTY de Boyer-Moore

```
count ← 0
foreach x ∈ S do
    if count = 0 then
        m ← x
        count ← 1
    else if m = x then
        count ← count+1
    else
        count ← count−1
    end
    return m
end
```

al candidato a mayoría, si aparecen más elementos diferentes que iguales a la mayoría propuesta, cambiamos de candidato.

La demostración de que el misterioso algoritmo 23.1 es correcto es como sigue: sea c una variable fantasma cuyo valor es el de count si m es la mayoría, $-$count en caso contrario. Cada vez que el algoritmo se encuentra con un valor igual a la mayoría, c aumenta (si m es la mayoría, aumenta count en uno; si no es la mayoría, count disminuye en uno); cada vez que encuentra un valor diferente a la mayoría, c puede aumentar o disminuir en uno. Habrán más aumentos que disminuciones de c, con lo que c al final del algoritmo es positivo. Pero esto solo es posible si count es positivo, y esto a su vez es solo si m es la mayoría.

Está claro que este algoritmo siempre retorna un valor, no detecta si la secuencia no tiene mayoría. Verificar esto requiere una segunda pasada por la secuencia.

Bibliografía

[1] Jon Louis Bentley: *Algorithm design techniques*. Communications of the ACM, 27(9):865–871, September 1984.

[2] Robert S. Boyer and J. Strother Moore: *MJRTY – a fast majority vote algorithm*. In Robert S. Boyer (editor): *Automated Reasoning: Essays in Honor of Woody Bledsoe*, volume 1 of *Automated Reasoning*, chapter 5, pages 105–117. Kluver Academic Publishers, 1991. Originally published as a technical report in 1981.

[3] David Gries: *A note on the standard strategy for developing loop invariants and loops*. Science of Computer Programming, 2(3):207–214, December 1982.

Clase 24

Programación lineal

Muchos de los problemas que queremos resolver son problemas de *optimización*: hallar el camino *más corto*, un árbol recubridor de *costo mínimo*, la subsecuencia común *más larga*. Tenemos ciertas reglas: hay que cumplir *restricciones* (solo usar arcos del grafo, los objetos deben ir en el orden de las secuencias) y debe ser *mejor posible* según un criterio bien definido.

Un conjunto amplio de problemas de optimización es la *programación lineal*. Como nota al margen, en esto «programación» se usa en el sentido de «planificación» o «asignar recursos», no «elaborar programas para computadora». Tenemos una colección de restricciones lineales a cumplir, para ellas nos interesa obtener el óptimo de una función objetivo también lineal. Muchísimos problemas pueden plantearse en esta forma, su solución es un problema muy importante que ha dado lugar a extensas investigaciones y extensiones. Mucho más detalle del que podremos dar en este limitado espacio dan Ferguson [7], Dasgupta, Papadimitriou y Vazirani [4, capítulo 7] y también Erickson [6], quien reseña un poco de la historia del problema. Sabemos que la variante del problema con variables enteras es NP-completo, nos interesa el caso de variables reales.

24.1. Solución gráfica

Como un primer ejemplo, consideremos el problema de hallar x_1 y x_2 tales que $x_1 \geq 0$, $x_2 \geq 0$ y:

$$\begin{aligned} x_1 + \ \ x_2 &<= 9 \\ x_1 + 3x_2 &<= 12 \\ -x_1 + 2x_2 &<= 2 \end{aligned} \tag{24.1}$$

tal que sea máximo $x_1 + 2x_2$. Lo relevante en este ejemplo es que las restricciones y la función objetivo son todas lineales. Gráficamente podemos visualizar este problema como en la figura 24.1. El área en amarillo es el área factible, donde se cumplen las restricciones. La geometría de la situación nos indica que el máximo de la función objetivo se encuentra en uno de los vértices del área factible, en este caso en la intersección entre $x_1 + 3x_2 = 12$ y $x_1 + x_2 = 9$, que es $x_1 = 15/2$ y $x_2 = 3/2$, donde vale $21/2$.

El área factible está limitada por las restricciones (cada restricción define un semiplano, el área factible es la intersección entre ellos), puede ser un polígono convexo (y hay soluciones factibles), puede no haber soluciones factibles (la intersección entre los semiplanos es vacía), o el área puede ser ilimitada, en cuyo caso puede o no haber óptimo.

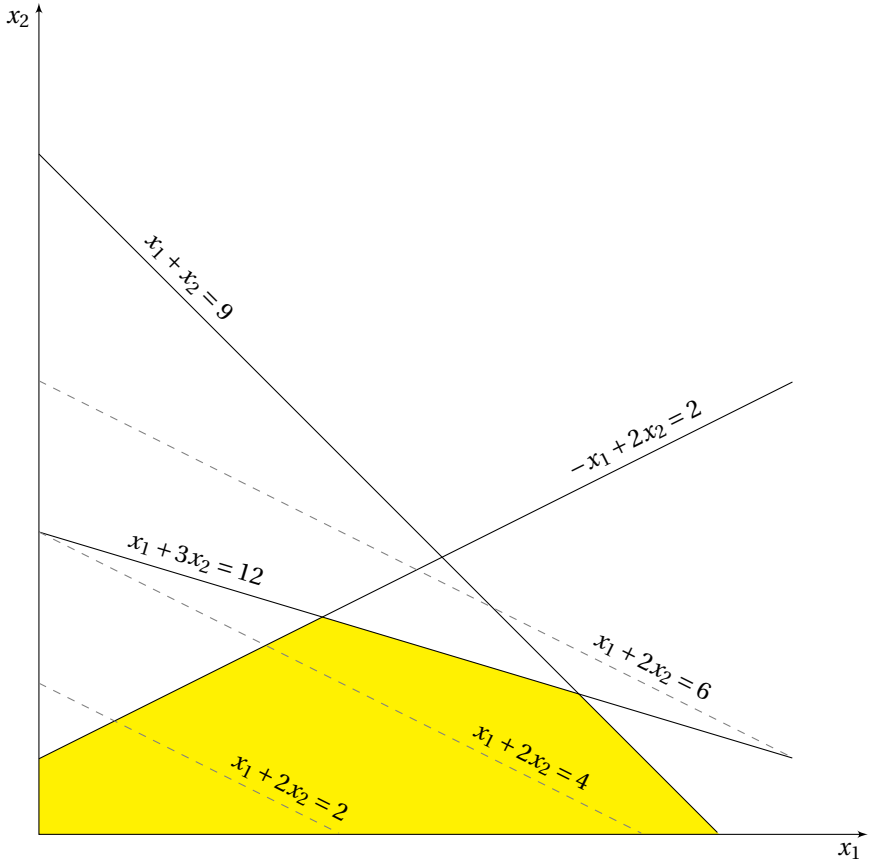


Figura 24.1 – Un problema de programación lineal

24.2. Problemas estándar

En nuestro ejemplo, tenemos restricciones de que las variables son no negativas, que ciertas expresiones lineales en las variables son menores o iguales que una constante, y buscamos el máximo de una expresión lineal. Nuestro ejemplo es un *problema estándar de máximo*, si son solo restricciones de mayores o iguales y minimizamos la función objetivo se le llama *problema estándar de mínimo*. Todo problema de programación lineal puede llevarse a una de estas formas usando las siguientes: Si la variable x no tiene restricciones, podemos reemplazarla por x^+ y x^-, ambas no negativas, y la reemplazamos por $x^+ - x^-$. Si tenemos restricciones lineales de mayor o igual, basta multiplicarlas por -1. Si hay igualdades, podemos despejar alguna de las variables de ellas y reemplazar en las demás restricciones para eliminarlas. Si buscamos un mínimo, nuevamente basta multiplicar por -1 la función objetivo.

Para simplificar notación, para vectores $\mathbf{x}$ e $\mathbf{y}$ escribiremos por ejemplo $\mathbf{x} \geq \mathbf{y}$ si cada componente de $\mathbf{x}$ es mayor o igual al elemento correspondiente de $\mathbf{y}$. Así el problema máximo estándar puede expresarse como:

$$\begin{aligned} &\text{maximizar } \mathbf{c}^T\mathbf{x} \\ &\text{sujeto a las restricciones } \mathbf{Ax} \leq \mathbf{b} \text{ y } \mathbf{x} \geq \mathbf{0} \end{aligned} \tag{24.2}$$

Una manera alternativa de ver el problema 24.2 es buscar una cota superior al valor buscado. Esto

lleva a considerar combinaciones lineales de las restricciones, sujetas a la condición que los coeficientes de las variables no pueden sobrepasar los coeficientes respectivos en la función objetivo, y buscamos la combinación que nos da el mínimo valor posible. O sea, tenemos un sistema para el vector **y**:

$$(\text{restricción 1})\, y_1 + (\text{restricción 2})\, y_2 + \cdots + (\text{restricción m})\, y_m$$

Decir que este valor es mínimo dados los lados derechos de las restricciones se traduce en:

$$\mathbf{y}^T\mathbf{c} \text{ mínimo}$$

Las restricciones de no sobrepasar los coeficientes en la función objetivo se traducen en:

$$\mathbf{A}^T\mathbf{y} \leq \mathbf{b}$$

Obtuvimos un problema estándar de mínimo, el *dual* del problema (24.2):

$$\begin{aligned} &\text{minimizar } \mathbf{y}^T\mathbf{b} \\ &\text{sujeto a las restricciones } \mathbf{y}^T\mathbf{A} \geq \mathbf{c}^T \text{ e } \mathbf{y} \geq \mathbf{0} \end{aligned} \tag{24.3}$$

Los vectores *b* y *c* cambiaron de posición, y la matriz se traspone. Repetir el ejercicio para el sistema resultante nos lleva de vuelta al problema origina. El punto interesante es que los valores óptimos de ambos problemas están relacionados. Se dice que (24.2) y (24.3) son *duales*. El problema original suele llamarse *primal* para distinguirlo del *dual*.

La relación entre problemas duales está dada por los siguientes resultados.

Teorema 24.1. *Si* **x** *es factible para el problema estándar de maximización* (24.2) *y es factible* **y** *para su dual* (24.3)*, entonces:*

$$\mathbf{c}^T\mathbf{x} \leq \mathbf{y}^T\mathbf{b} \tag{24.4}$$

Demostración.

$$\mathbf{c}^T\mathbf{x} \leq \mathbf{y}^T\mathbf{A}\mathbf{x} \leq \mathbf{y}^T\mathbf{b}$$

La primera desigualdad es por $\mathbf{c}^T \leq \mathbf{y}^T\mathbf{A}$ con $\mathbf{x} \geq \mathbf{0}$, la segunda de $\mathbf{A}\mathbf{x} \leq \mathbf{b}$ con $\mathbf{y} \geq \mathbf{0}$. □

Corolario 24.2. *Si un problema estándar y su dual son ambos factibles, ambos son factibles acotados.*

Demostración. Si **y** es factible para el problema mínimo, por (24.4) sabemos que $\mathbf{y}^T\mathbf{b}$ es una cota superior para los valores de $\mathbf{c}^T\mathbf{x}$ de un **x** factible del problema máximo. El converso es similar. □

Corolario 24.3. *Si hay vectores factibles* $\mathbf{x}^*$ *para el problema estándar máximo* (24.2) *e* $\mathbf{y}^*$ *para el problema dual* (24.3) *tales que* $\mathbf{c}^T\mathbf{x}^* = \mathbf{y}^{*T}\mathbf{b}$*, ambos son óptimos.*

Demostración. Si **x** es una solución factible de (24.2), entonces $\mathbf{c}^T\mathbf{x} \leq \mathbf{y}^{*T}\mathbf{b} = \mathbf{c}^T\mathbf{x}^*$, por lo que $\mathbf{x}^*$ es óptimo. Un argumento simétrico se aplica a $\mathbf{y}^*$. □

Demostraremos el resultado fundamental siguiente usando el método Simplex para resolver problemas de programación lineal más adelante. En resumen, si el problema o su dual es factible acotado, se cumplen las condiciones del corolario 24.3.

Teorema 24.4 (Dualidad)**.** *Si un problema estándar es factible acotado, también lo es su dual, sus óptimos son iguales y hay vectores óptimos para ambos.*

Básicamente, de las nueve combinaciones posibles de factible acotado, factible sin cota y no factible para el problema y su dual por el corolario 24.2 tres no son posibles (si el problema y su dual son factibles, ambos son factibles acotados), mientras el teorema 24.4 elimina dos más. Las combinaciones restantes (ambos factibles no acotados, uno factible no acotado y el otro no factible y ambos no factibles) son posibles.

Como corolario del teorema de dualidad, tenemos:

Teorema 24.5 (Equilibrio)**.** *Sean* $\mathbf{x}^*$ *e* $\mathbf{y}^*$ *factibles para el problema* (24.2) *y su dual* (24.3)*, respectivamente. Entonces* $\mathbf{x}^*$ *e* $\mathbf{y}^*$ *son óptimos si y solo si:*

$$y_i^* = 0 \quad \textit{para todo } i \textit{ para los cuales } \sum_j a_{ij} x_j^* < b_i \tag{24.5}$$

$$x_j^* = 0 \quad \textit{para todo } j \textit{ para los cuales } \sum_i y_i^* a_{ij} > c_j \tag{24.6}$$

Demostración. Demostramos implicancia en ambas direcciones. Primero, solo si para el índice i es $\sum_j a_{ij} x_j^* = b_i$ puede ser $y_i^* \neq 0$, y simétricamente solo si para j es $\sum_i y_i^* a_{ij} = c_j$ es posible $x_j^* \neq 0$, con lo que:

$$\sum_i y_i^* b_i = \sum_{i,j} y_i^* a_{ij} x_j^* = \sum_j c_j x_j^*$$

Pero eso es ${\mathbf{y}^*}^T \mathbf{b} = \mathbf{c}^T \mathbf{x}^*$, por el corolario 24.3 ambos son óptimos.

En la dirección contraria, por el corolario 24.3 si $\mathbf{x}^*$ e $\mathbf{y}^*$ son óptimos, es ${\mathbf{y}^*}^T \mathbf{b} = \mathbf{c}^T \mathbf{x}^*$. Como $\mathbf{x}^* \geq \mathbf{0}$ e $\mathbf{y}^* \geq \mathbf{0}$, esta igualdad es imposible a menos que se cumplan (24.5) y (24.6). □

Esto hace preguntarse el significado del vector $\mathbf{y}^*$ óptimo del problema dual. Consideremos para ello el tipo de problema de programación lineal conocido como *problema de dieta*: tenemos ciertas necesidades diarias de componentes de dieta (carbohidratos, proteínas, diversas vitaminas, minerales, ácidos grasos y aminoácidos esenciales). Sea b_j la cantidad mínima diaria que debemos ingerir del componente j. Cada alimento provee ciertas cantidades de cada componente, el alimento i provee a_{ij} del nutriente j por gramo, su costo es c_i por gramo. Si nos interesa la dieta de mínimo costo que cubre nuestras necesidades alimenticias, este es un problema de mínimo estándar, definido por la matriz $\mathbf{A}$, el vector de restricciones $\mathbf{b}$ y el vector de costos $\mathbf{c}$. El vector óptimo $\mathbf{x}^*$ expresa la cantidad de cada alimento a ingerir diariamente para cubrir las necesidades alimenticias a mínimo costo, el valor correspondiente de la función objetivo $\mathbf{c}^T \mathbf{x}^*$ es el costo diario de la dieta más barata. La función objetivo del problema dual en su óptimo, ${\mathbf{y}^*}^T \mathbf{b}$ tiene el mismo valor (por el corolario 24.3), los y_j^* representan lo que estamos pagando por unidad del componente j en la dieta óptima. El hecho que los y_j^* se anulan cuando hay holgura en una restricción significa que adquirir una unidad adicional de j sobre la necesaria es gratis.

24.3. Geometría de programación lineal

Muchos problemas de asignación de recursos son naturalmente programas lineales. Hay muchos otros problemas que se pueden traducir en programación lineal. Es importante contar con algoritmos eficientes para resolver esta clase de problemas. El método Simplex [3] es eficiente en la práctica (aunque es necesario tener algunas precauciones que discutiremos luego). Claro que Klee y Minty [11] hallaron una familia infinita de modelos (hipercubos en d dimensiones) en los cuales Simplex usando la regla de pivote de Dantzig requiere 2^d iteraciones para llegar al óptimo. En la práctica es mucho mejor, se ha demostrado que en hipercubos de d dimensiones, partiendo de un vértice al azar en promedio son d iteraciones, ver por ejemplo a Schrijver [13] o a Borgwardt [2].

Sigue siendo un problema abierto si hay una regla de elección de pivotes que garantiza tiempo polinomial. Recién en 1979 Khachiyan [10] dio un algoritmo polinomial, aunque en la práctica es muy lento. Karmarkar [9] luego dio un algoritmo polinomial que en algunos casos es competitivo en la práctica.

Discusión detallada de Simplex provee Reveliotis [12], parte de lo que sigue se adapta de allí.

Si tenemos n variables, estamos operando en un espacio con n dimensiones. Las restricciones corresponden a hiperplanos, cada una define un semi-espacio en el cual la restricción se cumple. La intersección de tales semi-espacios (donde se cumplen las restricciones) define lo que se denomina *politopo*, un politopo acotado se llama *poliedro*.

Dados puntos $\mathbf{x}_1 = (x_{11}, x_{12}, \ldots, x_{1n})^T$ y $\mathbf{x}_2 = (x_{21}, x_{22}, \ldots, x_{2n})^T$ la línea recta que pasa por ellos puede expresarse como:

$$\mathbf{x}_1 + t(\mathbf{x}_2 - \mathbf{x_1}), t \in \mathbb{R}$$

El segmento de línea recta entre $\mathbf{x}_1$ y $\mathbf{x}_2$ es el conjunto:

$$\mathbf{x}_1 + t(\mathbf{x}_2 - \mathbf{x_1}), t \in [0,1]$$

Equivalentemente:

$$\mu\mathbf{x}_1 + \kappa\mathbf{x}_2, \mu + \kappa = 1, \mu, \kappa \geq 0$$

Decimos que esto define una *combinación convexa* de $\mathbf{x}_1$ y $\mathbf{x}_2$. Se dice que un conjunto de puntos es *convexo* si el segmento que conecta dos puntos del conjunto está enteramente en su interior. Vale decir, si $\mathbf{x}_1, \mathbf{x}_2 \in S$, entonces $(1-t)\mathbf{x}_1 + t\mathbf{x}_2 \in S$ para todo $0 \leq t \leq 1$. Es claro que el semiespacio definido por una restricción es convexo, y por tanto lo es todo politopo resultante de sus intersecciones. Como último concepto, un *punto extremo* de un conjunto convexo S es un punto $\mathbf{x}_0$ tal que todo segmento que está enteramente en S y que contiene $\mathbf{x}_0$ tiene a $\mathbf{x}_0$ como uno de sus extremos. O sea, si $\mathbf{x}_1, \mathbf{x}_2 \in S$, y para $\mathbf{x}_0 \in S$ podemos escribir $\mathbf{x}_0 = (1-\kappa)\mathbf{x}_1 + \kappa\mathbf{x}_2$, entonces $\mathbf{x}_0 = \mathbf{x}_1$ o $\mathbf{x}_0 = \mathbf{x}_2$. En el caso particular de politopos convexos se habla de *vértices* para referirse a los puntos extremos. Esto nos lleva a:

Teorema 24.6 (Fundamental de la programación lineal)**.** *Si un programa lineal tiene una solución óptima acotada, existe un punto extremo de la región factible que es óptimo.*

Note que pueden haber varios vértices óptimos si resulta que un arista o una cara del politopo es paralela al plano que define la función objetivo.

24.3.1. El método Simplex

El método Simplex revisa distintos puntos extremos (vértices) del politopo definido por las restricciones. El nombre del método viene de que esos puntos extremos normalmente coinciden con vértices de poliedros con el mínimo número de caras si se omiten restricciones no involucradas (en dos dimensiones, un triángulo; en tres es tetraedro; en general, en n dimensiones tiene $n+1$ caras). A tales poliedros se les llama *simplex*.

Volvamos a nuestro primer ejemplo, maximizar $x_1 + 2x_2$ sujeto a:

$$\begin{aligned} x_1 + \ \ x_2 &<= 9 \\ x_1 + 3x_2 &<= 12 \\ -x_1 + 2x_2 &<= 2 \end{aligned} \tag{24.7}$$

Trabajar con desigualdades es incómodo, introducimos variables holgura (en inglés, *slack variables*) para cada desigualdad. Agregamos una igualdad adicional con holgura z para la función objetivo:

$$\begin{aligned} x_1 + x_2 + s_1 &= 9 \\ x_1 + 3x_2 + s_2 &= 12 \\ -x_1 + 2x_2 + s_3 &= 2 \\ -x_1 - 2x_2 + z &= 0 \end{aligned} \tag{24.8}$$

De (24.8) una solución factible al problema es obvia: haga $x_1 = x_2 = 0$, y $s_1 = 9$, $s_2 = 12$, $s_3 = 2$. Se dice que s_1 a s_3 están en la base las demás variables no (estamos trabajando en el espacio vectorial definido por las variables, las variables en la base están siendo usadas como base para el subespacio definido por las ecuaciones). Claro que esto nos da $z = 0$ para la función objetivo. De la última fila vemos que podemos aumentar z aumentando x_1 o x_2 (aparecen con coeficientes negativos). Analicemos x_2. De la primera ecuación vemos que podemos aumentar x_2 hasta 9, dejando $s_1 = 0$; la segunda indica que $x_2 = 4$ deja $s_2 = 0$; la tercera dicta $x_2 = 1$ para $s_3 = 0$. Debemos elegir el mínimo de estos (no se permiten holguras negativas). Usamos la tercera ecuación para eliminar x_2 de las demás ecuaciones:

$$\begin{aligned} \tfrac{3}{2}x_1 + s_1 - \tfrac{1}{2}s_3 &= 8 \\ \tfrac{5}{2}x_1 + s_2 - \tfrac{3}{2}s_3 &= 9 \\ -\tfrac{1}{2}x_1 + x_2 + \tfrac{1}{2}s_3 &= 1 \\ -2x_1 + s_3 + z &= 2 \end{aligned} \tag{24.9}$$

El sistema resultante tiene la misma forma que el original, claro que las variables independientes (que aparecen en una ecuación solamente) ahora son x_2, s_1, s_2. A esta operación le llaman *pivotear* (alrededor de (x_2, s_3)), x_2 entra a la base mientras s_3 sale. El resultado corresponde a la solución factible $s_1 = 8, s_2 = 9, x_2 = 1$ con $x_1 = s_3 = 0$; la función objetivo es $z = 2$. Ahora la única variable con coeficiente negativo en la última fila es x_1. Debemos elegir entre 16/3 y 18/5, es la segunda ecuación:

$$\begin{aligned} s_1 - \tfrac{3}{5}s_2 + \tfrac{2}{5}s_3 &= \tfrac{13}{5} \\ x_1 + \tfrac{2}{5}s_2 - \tfrac{3}{5}s_3 &= \tfrac{18}{5} \\ x_2 + \tfrac{1}{5}s_2 + \tfrac{1}{5}s_3 &= \tfrac{14}{5} \\ \tfrac{4}{5}s_2 - \tfrac{1}{5}s_3 + z &= \tfrac{46}{5} \end{aligned} \tag{24.10}$$

Esto corresponde a $x_1 = 18/5$, $x_2 = 14/5$, $s_1 = 13/5$ y función objetivo $z = 46/5$. Coeficiente negativo en la última fila tiene s_3; como antes elegimos el menor, en este caso 13/2 de la primera ecuación, resultando:

$$\begin{aligned} \tfrac{5}{2}s_1 - \tfrac{3}{2}s_2 + s_3 &= \tfrac{13}{2} \\ x_1 + \tfrac{3}{2}s_1 - \tfrac{1}{2}s_2 &= \tfrac{15}{2} \\ x_2 - \tfrac{1}{2}s_1 + \tfrac{1}{2}s_2 &= \tfrac{3}{2} \\ \tfrac{1}{2}s_1 + \tfrac{1}{2}s_2 + z &= \tfrac{21}{2} \end{aligned} \tag{24.11}$$

No hay coeficientes negativos en la última fila, es óptimo. Es $x_1 = 15/2, x_2 = 3/2, s_3 = 13/2$ con función objetivo $z = 21/2$. Esto es lo mismo que habíamos obtenido de nuestra solución gráfica. Si analizamos el camino seguido por nuestra técnica, caminamos de un vértice a un vecino hasta alcanzar el óptimo. En cada paso aumenta la función objetivo.

Es obvio abreviar esto registrando solo los coeficientes en una matriz. Vemos también que en la matriz aparecen muchos ceros, y hay columnas con un único 1. En vez de representar esto explícitamente, podemos indicar para cada fila cuál es la columna (variable) que tiene coeficiente uno y cuyo

coeficiente es cero en el resto de la columna. Podemos reutilizar el espacio que se abre al eliminar una variable, llenándolo con los coeficientes de la nueva columna. Para ello debemos indicar a qué variable corresponde cada columna, y qué variable es la que aparece como independiente en la fila. A esta representación se llama un *tableau* (plural del francés es *tableaux*). En esta representación se puede ilustrar el efecto de la operación de pivote alrededor de p sobre elementos en su misma fila y columna y otros elementos mediante:

$$\boxed{\begin{array}{cc} p & a \\ b & c \end{array}} \rightsquigarrow \boxed{\begin{array}{cc} 1/p & a/p \\ -c/p & c-ab/p \end{array}} \tag{24.12}$$

Ilustramos el proceso de trabajar con *foreignlanguagefrenchtableau* con el ejemplo de maximizar $5x_1 + 2x_2 + x_3$ con las restricciones:

$$\begin{aligned} x_1 + \ 3x_2 - x_3 &<= 6 \\ x_2 + x_3 &<= 4 \\ 3x_1 + x_2 &< = 7 \end{aligned} \tag{24.13}$$

El *foreignlanguagefrenchtableau* inicial es:

	x_1	x_2	x_3	
s_1	1	3	-1	6
s_2	0	1	1	4
s_3	3	1	0	7
	-5	-2	-1	0

Pivoteando alrededor de s_3, x_1 (intercambia el papel de estas variables) da:

	s_3	x_2	x_3	
s_1	$-1/3$	$8/3$	-1	$11/3$
s_2	0	1	1	4
x_1	$1/3$	$1/3$	0	$7/3$
	$5/3$	$-1/3$	-1	$35/3$

Ahora pivotear (s_2, x_3) entrega:

	s_3	x_2	s_2	
s_1	$-1/3$	$11/3$	1	$23/3$
x_3	0	1	1	4
x_1	$1/3$	$1/3$	0	$7/3$
	$5/3$	$2/3$	1	$47/3$

Como no hay coeficientes negativos en la última fila, es óptimo. Leemos $x_1 = 7/3$, $x_2 = 0$ y $x_3 = 4$; la función objetivo es $47/3$.

Si recordamos el problema dual, su solución se obtiene de leer por columnas y no por filas. El óptimo para este es $y_1 = 0$, $y_2 = 1$, $y_3 = 5/3$, con el mismo valor de la función objetivo.

24.3.2. Reglas de pivote

Sistematizaremos la elección de pivotes. Hay varios casos a considerar. Supongamos que después de pivotear un rato tenemos el *foreignlanguagefrenchtableau*:

	y	
x	**A**	**b**
	$-\mathbf{c}$	v

Si $\mathbf{b} \geq \mathbf{0}$, una solución factible para el problema máximo del que partimos es $\mathbf{x} = \mathbf{b}$, $\mathbf{y} = \mathbf{0}$, y tenemos el valor v de la función objetivo. Si $-\mathbf{c} \geq \mathbf{0}$, una solución factible para el problema dual está dada por $\mathbf{y} = -\mathbf{c}$ y $\mathbf{x} = \mathbf{0}$, también con valor v. Sabemos que si $\mathbf{b} \geq \mathbf{0}$ y $-\mathbf{c} \geq \mathbf{0}$ tenemos los óptimos.

Consideraremos primero el caso en que ya tenemos un punto factible.

Caso 1: Si $\mathbf{b} \geq \mathbf{0}$, tome cualquier columna s que contenga un elemento negativo en la última fila, $-c_s < 0$. En esa columna elija r tal que $a_{rs} > 0$ y b_r / a_{rs} sea mínimo, eligiendo cualquiera en caso de empate. Pivotee alrededor de x_r, y_s.

Si esta regla no es aplicable, puede ser porque $-\mathbf{c} \geq \mathbf{0}$, en cuyo caso tenemos el óptimo; o en la columna s todos los $a_{is} \leq 0$. En este último caso, el problema de maximización es factible no acotado. Para verlo, considere un vector $\mathbf{x} \geq \mathbf{0}$ con $x_s > 0$ y $x_j = 0$ para $j \neq s$. Entonces $\mathbf{x}$ es factible para el problema de maximización ya que para todo i:

$$y_i = \sum_j (-a_{ij} x_j + b_i = -a_{is} x_s + b_i \geq 0$$

Este vector da el valor $\sum c_j x_j = c_s x_s$, que podemos aumentar a gusto a través de r_s.

Esta regla de pivote es conveniente por las siguientes.

Proposición 24.1. *Si* $\mathbf{b} \geq \mathbf{0}$ *antes de aplicar la regla, entonces se cumple después de pivotear.*

Demostración. Marcamos los valores luego del pivote por circunflejos. Debemos demostrar $\widehat{b}_i \geq 0$ para todo i. Para $i = s$ es $\widehat{b}_s = b_s / a_{rs}$, no negativo ya que $b_s \geq 0$ y $a_{rs} > 0$. Para $i \neq s$ resulta:

$$\widehat{b}_i = b_i - \frac{a_{is}}{b_s} a_{rs}$$

Si $a_{is} < 0$, estamos restando un valor negativo, $\widehat{b}_i > b_i$; si $a_{is} > 0$ por la regla para elegir el pivote $a_{is} \leq a_{rs}$, estamos restando una fracción de b_i que se mantiene no negativo. □

Proposición 24.2. *El valor del nuevo* foreignlanguagefrenchtableau *nunca es menor que el original.*

Demostración. Tenemos que $-c_s < 0$, $a_{rs} > 0$ y $b_r \geq 0$, por lo que:

$$\widehat{v} = v - (-c_s) \frac{b_r}{a_{rs}} \geq v$$

□

Con estas dos propiedades, como nos movemos de un vértice a uno vecino, si en cada paso aumenta el valor hallaremos el óptimo en un número finito de pasos (el politopo tiene un número finito de vértices) o concluiremos que es factible no acotado.

Caso 2: Si hay b_i negativos, elija el primero de ellos, llamémosle b_k (con esta elección tenemos $b_1 \geq 0, b_2 \geq 0, \ldots, b_{k-1} \geq 0$). Elija alguna entrada negativa en la fila k, digamos $a_{ks} < 0$. Esta será la columna pivote. Compare b_k / a_{ks} con los b_i / a_{is} con $b_i > 0$ y $a_{is} > 0$, sea r el índice que da la razón más pequeña (podría ser $r = k$), eligiendo cualquiera en caso de empate. Pivotee alrededor de (r, s).

Si esta regla no es aplicable, el problema máximo no tiene soluciones factibles. La fila k dice:

$$-y_k = \sum_j a_{kj} x_j - b_k$$

Para vectores factibles $\mathbf{x} \geq \mathbf{0}$, $\mathbf{y} \geq \mathbf{0}$ el lado izquierdo es negativo o cero, el lado derecho es positivo.

Lo que buscamos es aumentar ese b_k, de manera de llegar al caso 1 que sabemos manejar. Esto por las siguientes.

Proposición 24.3. *Los b_i no negativos se mantienen no negativos al pivotear.*

Demostración. Suponga que $b_i \geq 0$, o sea $i \neq k$. Si $i = r$, entonces $\widehat{b}_r = b_r / a_{rs} \geq 0$. Si $i \neq r$, entonces:

$$\widehat{b}_i = b_i - \frac{a_{is}}{a_{rs}} b_r$$

Pero $b_r / a_{rs} \geq 0$. Si $a_{is} < 0$ entonces $\widehat{b}_i \geq b_i \geq 0$; mientras que si $a_{is} > 0$ entonces $b_s / a_{rs} \leq b_i / a_{is}$, con lo que $\widehat{b}_i \geq b_i - b_i = 0$. □

Proposición 24.4. *Al aplicar la regla, b_k negativo no disminuye.*

Demostración. Si $k = r$, entonces $\widehat{b}_k = b_k / a_{rs} > 0$ (ambos son negativos). Si $k \neq r$, entonces:

$$\widehat{b}_k = b_k - \frac{a_{ks}}{a_{rs}} b_r \geq b_k$$

□

Si aplicamos la regla del caso 2, y en cada paso aumenta b_k, en algún momento se hará positivo y eliminamos un b_k negativo.

24.3.3. Ciclos

Lamentablemente, nuestras reglas no garantizan mejoras. Existe la posibilidad de entrar en un ciclo, como ilustra el siguiente ejemplo, maximizar $3x_1 - 5x_2 + x_3 - 2x_4$ sujeto a $x_i \geq 0$ y:

$$\begin{aligned} x_1 - 2x_2 - \quad x_3 + 2x_4 &<= 0 \\ 2x_1 - 3x_2 - \quad x_3 + \quad x_4 &<= 0 \\ x_3 &<= 1 \end{aligned} \tag{24.14}$$

Si se pivotea en el orden (x_1, y_1), (x_2, y_2), (x_3, x_1), (x_4, x_2), (y_1, x_3), (y_2, x_4) volvemos al *foreignlanguagefrenchtableau* original.

El problema de ciclos en la práctica es muy raro. Pero resulta que hay una regla simple y eficiente, debida a Bland [1], que evita ciclos. Esta regla, llamada *mínimo subíndice*, dice que en caso de empate de fila o columna pivote, se seleccione la fila (o columna) en la cual la variable x tenga el mínimo subíndice; si no hay variables x elija la y con el mismo criterio.

Finalmente tenemos una demostración constructiva para el teorema de dualidad: Use el método Simplex para obtener la solución al problema de máximo acotado; automáticamente obtenemos la solución al problema dual.

24.3.4. Comentarios finales

Esto en realidad es una familia de algoritmos. No hemos especificado cómo elegir la columna pivote si hay varios coeficientes negativos en la última fila. Una regla simple, propuesta por Dantzig, es usar aquella columna con el coeficiente más negativo. Hay varias otras opciones, algunas de ellas discute Eigen [5], una comúnmente usada y generalmente efectiva es la de Devex, de Harris [8]. Lo que buscan es la arista que comienza el camino más corto al óptimo, pero hallarla es más difícil que resolver el problema, se emplean diversas heurísticas. Tampoco hay una regla para elegir en caso de empate (aunque hemos dado la regla de Bland).

Un vértice de un politopo en n dimensiones (si hay n variables) está definido por $n-1$ hiperplanos, si hay m restricciones esto está acotado por $\binom{m}{n-1}$, lo que es más que cualquier polinomio en mn (el número de coeficientes que definen el modelo). Klee y Minty [11] hallaron una familia infinita de modelos en los cuales Simplex usando la regla de pivote de Dantzig toma tiempo exponencial. En la práctica, el método es notablemente eficiente, particularmente con reglas de pivote elegidas cuidadosamente.

Ejercicios

1. Dé y muestre gráficamente problemas de programación lineal que no tienen soluciones factibles, que tienen soluciones factibles acotadas y soluciones factibles no acotadas. ¿Es posible tener áreas no acotadas con soluciones acotadas?

2. Considere el problema de programación lineal:

$$\begin{aligned} x_1 + 2x_2 - \quad x_3 + 3x_4 &<= 7 \\ 2x_1 - 3x_2 - \quad x_3 + \quad x_4 &<= 3 \\ \qquad\qquad x_3 &< \quad\quad = 1 \\ x_1 - \quad x_2 + \quad x_3 - \quad x_4 &= 3 \end{aligned}$$

   a) Resuélvalo usando nuestra técnica inicial (mencionando explícitamente las ecuaciones completas), sin utilizar la ecuación para reducir el problema.

   b) Describa las modificaciones necesarias a la técnica usando *foreignlanguagefrenchtableaux* para representar esto.

3. Considere el problema de maximizar $3x_1 - 5x_2 + x_3 - 2x_4$ sujeto a $x_i \geq 0$ y:

$$\begin{aligned} x_1 - 2x_2 - \quad x_3 + 2x_4 &<= 0 \\ 2x_1 - 3x_2 - \quad x_3 + \quad x_4 &<= 0 \\ \qquad\qquad x_3 &< \quad\quad = 1 \end{aligned}$$

   a) Escriba el *foreignlanguagefrenchtableau* inicial para este problema.

   b) Verifique que pivotear en el orden (x_1, y_1), (x_2, y_2), (x_3, x_1), (x_4, x_2), (y_1, x_3), (y_2, x_4) respeta las reglas planteadas, pero volvemos a un *foreignlanguagefrenchtableau* equivalente al original.

4. Repita el ejercicio 24.3.4, pero esta vez usando la regla de Bland.

Bibliografía

[1] Robert G. Bland: *New finite pivoting rules for the simplex method.* Mathematics of Operations Research, 2(2):103–107, May 1977.

[2] Karl Heinz Borgwardt: *The Simplex Method: A Probabilistic Analysis*, volume 1 of *Algorithms and Combinatorics.* Spŕinger, 1987.

[3] George B. Dantzig: *Maximization of a linear function of variables subject to linear inequalities.* In Tjalling C. Koopmans (editor): *Activity Analysis of Production and Allocation*, pages 339–347. Wiley and Chapman-Hall, 1947.

[4] Sanjoy Dasgupta, Christos H. Papadimitrou, and Umesh V. Vazirani: *Algorithms.* McGraw Hill Higher Education, 2006.

[5] David Eigen: *Pivot rules for the simplex method.* https://cs.nyu.edu/courses/fall12/CSCI-GA.2945-002/pastproject2.pdf, May 2011. Computer Science, Courant Institute of Mathematical Sciences, New York University.

[6] Jeff Erickson: *Algorithms, etc.* http://jeffe.cs.illinois.edu/teaching/algorithms, January 2015. Department of Computer Science, University of Illinois at Urbana-Champaign.

[7] Thomas S. Ferguson: *Linear programming: A concise introduction.* https://www.math.ucla.edu/~tom/LP.pdf, 2015. Department of Mathematics, University of California at Los Angeles.

[8] Paula M. J. Harris: *Pivot selection metods of the Devex LP code.* Mathematical Programming, 5(1):1–28, December 1973.

[9] Narendra Karmarkar: *A new polynomial-time algorithm for linear programming.* Combinatorica, 4(4):373–395, December 1984.

[10] Leonid G. Khachiyan: *A polynomial algorithm in linear programming.* Doklady Akademiia Nauk SSSR, 244:1093–1096, 1979. (English translation: Soviet Mathematics Doklady, 20(1):191–194, 1979).

[11] Victor Klee and George J. Minty: *How good is the simplex algorithm?* In *Inequalities III (Proceedings of the Third Symposium on Inequalities)*, pages 159–175, September 1969.

[12] Spyros Reveliotis: *An introduction to linear programming and the Simplex Method.* http://www2.isye.gatech.edu/~spyros/LP/LP.html, June 1997. School of Industrial and Systems Engineering, Georgia Institute of Technology.

[13] Alexander Schrijver: *Theory of Linear and Integer Programming.* John Wiley & Sons, 1998.

Clase 25

La transformada rápida de Fourier

Un algoritmo importantísimo basado en dividir y conquistar es la transformada rápida de Fourier, abreviada FFT (por el nombre en inglés, *Fast Fourier Transform*). Es relevante no solo por sus aplicaciones directas (particularmente en procesamiento de señales), también es la base para algunos otros algoritmos importantes y es el corazón de los algoritmos asintóticamente más rápidos conocidos para algunos problemas. La motivación siguiente se adapta de Dasgupta, Papadimitrou y Vazirani [1].

Vimos que dividir y conquistar ayuda al multiplicar números y matrices, ahora consideraremos polinomios. Sabemos que un polinomio de grado n queda determinado por sus valores en $n+1$ puntos distintos, lo que nos da una representación alternativa a la secuencia de coeficientes. Podemos dar el polinomio $A(x)$ como:

- Su secuencia de coeficientes, $a_0, a_1, \ldots, a_n$
- Sus valores en $n+1$ puntos distintos, $A(x_0), A(x_1), \ldots, A(x_n)$

La segunda representación es muy atractiva a la hora de multiplicar polinomios, $A(x) \cdot B(x)$ es simplemente $A(x_0) \cdot B(x_0), A(x_1) \cdot B(x_1), \ldots, A(x_n) \cdot B(x_n)$, son $n+1$ multiplicaciones, no $(n+1)(n+2)/2$ como en el esquema tradicional:

$$[x^k]A(x)B(x) = \sum_{0 \le j \le k} a_j b_{k-j}$$

Esto lleva a la idea de partir con los coeficientes de dos polinomios de grado d; evaluar los polinomios en n puntos $x_0, x_1, \ldots, x_{n-1}$ elegidos, donde $n \ge 2d+1$; multiplicar los valores; y extraer los coeficientes del producto.

Una primera idea es evaluar cada polinomio en pares positivo/negativo, vale decir:

$$\pm x_0, \pm x_1, \ldots, \pm x_{n/2-1}$$

porque de esa forma las computaciones para calcular $A(x_i)$ y $A(-x_i)$ traslapan en gran medida, ya que las potencias pares de x_i coinciden con las de $-x_i$. Separando potencias pares e impares podemos escribir:

$$A(x) = A_e(x^2) + xA_o(x^2)$$

donde A_e y A_o tienen la mitad del grado de A, con lo que:

$$A(x) = A_e(x^2) + xA_e(x^2)$$
$$A(-x) = A_e(x^2) - xA_e(x^2)$$

y basta entonces evaluar A_e y A_o en los $n/2$ puntos $x_0^2, x_1^2, \ldots, x_{n/2-1}^2$ y un poco de trabajo adicional. Lo malo es que este truco positivo/negativo solo puede aplicarse una vez, no podemos aplicarlo recursivamente *a menos que usemos números complejos*.

La pregunta ahora es, ¿qué números complejos elegir? Ingeniería reversa del proceso indica que partimos con un número de puntos que es una potencia de 2, que en cada recursión se dividen en dos hasta terminar con un único valor final, que podemos arbitrariamente fijar como 1. O sea, en cada nivel tenemos las raíces cuadradas de los puntos del nivel previo, tenemos las n-ésimas raíces complejas de 1, para n una potencia de 2. Recordamos que podemos escribirlas $1, \omega, \omega^2, \ldots, \omega^{n-1}$, donde $\omega = e^{2\pi i/2}$. Si n es par, están pareadas positivo/negativo, o sea $\omega^{n/2+j} = -\omega^j$, y sus cuadrados son las $(n/2)$-ésimas raíces de 1. Si partimos con las n-ésimas raíces de 1, con n una potencia de 2, en cada nivel tendremos las raíces $(n/2^k)$-ésimas para $k = 0, 1, \ldots$. Sucesivos niveles de la recursión funcionan perfectamente. El algoritmo 25.1 resultante es la transformada rápida de Fourier. Tenemos

Algoritmo 25.1: Transformada rápida de Fourier

```
function FFT(A, ω)
    if ω = 1 then
        return A(1)
    end
    Exprese A(x) = A_e(x^2) + x A_o(x^2)
    Llame FFT(A_e, ω^2) para evaluar A_e en potencias pares de ω
    Llame FFT(A_o, ω^2) para evaluar A_o en potencias pares de ω
    for j ← 0 to n − 1 do
        Compute A(ω^j) = A_e(ω^{2j}) + ω^j A_o(ω^{2j})
    end
    return A(ω^0), A(ω^1), ..., A(ω^{n−1})
end
```

así un algoritmo $O(n \log n)$ para evaluar un polinomio en los n puntos ω^j, podemos multiplicar polinomios expresados como valores en tiempo $O(n)$, falta obtener los coeficientes del producto, la operación inversa. Sorprendentemente, si:

$$\langle \text{valores} \rangle = \text{FFT}(\langle \text{coeficientes} \rangle, \omega)$$

es:

$$\langle \text{coeficientes} \rangle = \frac{1}{n} \text{FFT}(\langle \text{valores} \rangle, \omega^{-1})$$

Esto porque si:

$$b_r = \sum_{0 \le k \le n} a_k \omega^{kr}$$

tenemos que:

$$\begin{aligned}\sum_{0\le k<n} b_k\omega^{-kr} &= \sum_{0\le k<n}\left(\sum_{0\le j<n} a_j\omega^{jk}\right)\omega^{-kr}\\ &= \sum_{0\le j<n} a_j \sum_{0\le k<n} \omega^{(j-r)k}\end{aligned}$$

Ahora bien, sabemos que:

$$\sum_{0\le k<n} \omega^{km} = \begin{cases} n & n \mid m \\ 0 & n \nmid m \end{cases}$$

Esto porque la suma es una serie geométrica. Si $m \mid n$, es $\omega^{km} = 1$, dando el primer caso; cuando $n \nmid m$, tenemos que $\omega^m \neq 1$, por definición es $\omega^n = 1$ y:

$$\begin{aligned}\sum_{0\le k<n} \omega^{km} &= \frac{1-\omega^{mn}}{1-\omega^m}\\ &= 0\end{aligned}$$

En nuestra suma solo cuando $j = r$ es $n \mid (j-r)$, dando el resultado anunciado:

$$\sum_{0\le k<n} b_k\omega^{-kr} = a_r$$

Esto completa un elegante algoritmo $O(n\log n)$ para multiplicar polinomios.

Multiplicar enteros, por ejemplo en notación decimal, es multiplicar dos polinomios evaluados en la base del caso. Puede adaptarse el algoritmo de multiplicación de polinomios para obtener un algoritmo $O(n\log n)$ para números de n dígitos.

Bibliografía

[1] Sanjoy Dasgupta, Christos H. Papadimitrou, and Umesh V. Vazirani: *Algorithms*. McGraw Hill Higher Education, 2006.

Clase 26

Métodos simples de ordenamiento

Como una ilustración de técnicas de análisis de algoritmos más avanzadas, analizaremos los métodos de ordenamiento más simples. Tenemos los métodos de la burbuja (listado 26.1), selección, (listado 26.2), e inserción (listado 26.3). 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 26.1 – Método de la burbuja

```
void
sort(double a[], int n)

{
        for(int i = n; i; i--) {
                int imax = 0; double max = a[0];
                for(int j = 1; j < i; j++)
                        if(a[j] > a[imax])
                                imax = j;

                double tmp = a[i]; a[i] = a[imax]; a[imax] = tmp;
        }
}
```

Listado 26.2 – 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 26.3 – Método de inserción

26.1. Rendimiento de métodos simples de ordenamiento

Nos interesa obtener información más detallada que esta 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 26.1. 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 en la posición i participa en $i-1$ inversiones con elementos previos, 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 26.1. Nuevamente, si suponemos que el espacio vacío

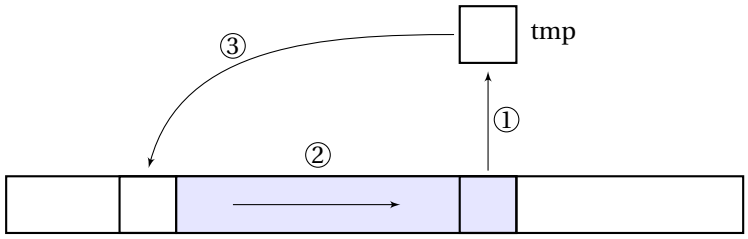


Figura 26.1 – Operación del método de inserción

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.

26.2. 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. Véase el apéndice F o el apunte de Fundamentos de Informática [1].

26.3. 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:

$$I(z) = \sum_{\pi\in\mathscr{P}} \iota(\pi)\frac{z^{|\pi|}}{|\pi|!} \tag{26.1}$$

En particular, nos interesa el número promedio de inversiones para permutaciones de tamaño n.

Podemos describir permutaciones mediante la expresión simbólica:

$$\mathscr{P} = \mathscr{E} + \mathscr{P}\star\mathscr{Z} \tag{26.2}$$

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:

$$(|\pi|+1)\iota(\pi) + \sum_{0\le k\le|\pi|} k = (|\pi|+1)\iota(\pi) + \frac{|\pi|(|\pi|+1)}{2} \tag{26.3}$$

Con esto tenemos la descomposición para la función generatriz cumulativa (la permutación de cero elementos no tiene inversiones):

$$\begin{aligned} I(z) &= \iota(\epsilon) + \sum_{\pi\in\mathscr{P}} \left((|\pi|+1)\iota(\pi) + \frac{|\pi|(|\pi|+1)}{2} \right) \frac{z^{|\pi|+1}}{(|\pi|+1)!} \\ &= \sum_{\pi\in\mathscr{P}} \iota(\pi) \frac{z^{|\pi|+1}}{|\pi|!} + \frac{1}{2} \sum_{\pi\in\mathscr{P}} \frac{z^{|\pi|+1}}{|\pi|!} |\pi| \end{aligned} \tag{26.4}$$

Como hay $k!$ permutaciones de tamaño k, sumando sobre tamaños resulta:

$$\begin{aligned} &= zI(z) + \frac{1}{2} z \sum_{k\ge0} k z^k \\ &= zI(z) + \frac{z^2}{2(1-z)^2} \end{aligned} \tag{26.5}$$

Despejando:

$$I(z) = \frac{1}{2} \frac{z^2}{(1-z)^3} \tag{26.6}$$

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:

$$\mathsf{E}_n[\iota] = [z^n] I(z) \tag{26.7}$$

$$\begin{aligned} &= \frac{1}{2} [z^n] \frac{z^2}{(1-z)^3} \\ &= \frac{1}{2} [z^{n-2}] (1-z)^{-3} \\ &= \frac{1}{2} \binom{-3}{n-2} \\ &= \frac{1}{2} \binom{n-2+(3-1)}{3-1} \\ &= \frac{1}{2} \binom{n}{2} \\ &= \frac{n(n-1)}{4} \end{aligned} \tag{26.8}$$

Podemos resumir las anteriores como número asintótico de asignaciones al ordenar n elementos en el cuadro 26.1. Queda claro (salvo para optimistas incurables) que el método de inserción es mejor.

26.4. 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.

Algoritmo	Min	Prom	Máx
Burbuja	0	$3n^2/4$	$3n^2/2$
Inserción	$2n$	$n^2/4$	$n^2/2$

Cuadro 26.1 – Comparación entre métodos de burbuja e inserción

```
1  double
2  maximum(const double a[], const int n)
3  {
4          double max;
5
6          max = a[0];
7          for(int i = 1; i < n; i++)
8                  if(a[i] > max)
9                          max = a[i];
10         return max;
11 }
```

Listado 26.4 – Hallar el máximo

Para el método de selección el número de asignaciones está dado por el número de veces que hallamos un elemento mayor, ver el listado 26.4. o sea, el número de máximos de izquierda a derecha en la permutación. Todas las operaciones se efectúan n veces, salvo las actualizaciones a la variable max. Es evidente que el número de veces que se actualiza max es $O(n)$, pero interesa una respuesta más precisa. Para obtener este valor, podemos optar por funciones generatrices cumulativas o bivariadas. Exploraremos ambas opciones como ejemplos en lo que sigue.

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:

$$\mathscr{P} = \mathscr{E} + \mathscr{P} \star \mathscr{Z} \tag{26.9}$$

Llamaremos $\chi(\pi)$ al número de máximos de izquierda a derecha en la permutación π.

26.4.1. Función generatriz cumulativa

Definimos la función generatriz cumulativa exponencial:

$$\widehat{C}(z) = \sum_{\pi \in \mathscr{P}} \chi(\pi) \frac{z^{|\pi|}}{|\pi|!} \tag{26.10}$$

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:

$$\chi(\pi) + [j = |\pi| + 1] \tag{26.11}$$

con lo que en total para el conjunto de permutaciones $\pi \star (1)$ es:

$$\sum_{1 \le j \le |\pi|+1} (\chi(\pi) + [j = |\pi| + 1]) = (|\pi| + 1)\chi(\pi) + 1$$

con lo que de (26.9) resulta:

$$\begin{aligned}\widehat{C}(z) &= \chi(\epsilon)\frac{z^0}{0!} + \sum_{\pi\in\mathscr{P}}((|\pi|+1)\chi(\pi)+1)\frac{z^{|\pi|+1}}{(|\pi|+1)!}\\ &= 0 + z\sum_{\pi\in\mathscr{P}}\chi(\pi)\frac{z^{|\pi|}}{|\pi|!} + \sum_{\pi\in\mathscr{P}}\frac{z^{|\pi|+1}}{(|\pi|+1)!}\end{aligned}$$

Como hay $n!$ permutaciones de largo n, la última suma se simplifica:

$$= z\widehat{C}(z) + \sum_{n\geq 0} n!\frac{z^{n+1}}{(n+1)!}$$

$$\begin{aligned}&= z\widehat{C}(z) + \sum_{n\geq 0}\frac{z^{n+1}}{n+1}\\ &= z\widehat{C}(z) + \ln\frac{1}{1-z}\end{aligned}$$

Despejamos:

$$\widehat{C}(z) = \frac{1}{1-z}\ln\frac{1}{1-z} \tag{26.12}$$

Nos interesa el coeficiente de z^n de (26.12), que casualmente es directamente el valor promedio que buscamos (ver por ejemplo el apunte de Fundamentos de Informática [1] para desarrollo de las funciones generatrices empleadas):

$$[z^n]\widehat{C}(z) = H_n \tag{26.13}$$

En promedio a max se asigna un nuevo valor $H_n = \ln n + \gamma + O(1/n)$ veces al buscar el máximo de n valores.

26.4.2. Función generatriz bivariada

La función generatriz de probabilidad de que una permutación de tamaño n tenga k máximos de izquierda a derecha es:

$$M(z,u) = \sum_{\pi\in\mathscr{P}}\frac{z^{|\pi|}}{|\pi|!}u^{\chi(\pi)} \tag{26.14}$$

Esto casualmente es la función generatriz exponencial bivariada correspondiente a la clase (26.9).

Razonando como en la sección 26.4.1 obtenemos:

$$M(z,u) = \frac{z^{|\epsilon|}}{|\epsilon|!}u^{\chi(\epsilon)} + \sum_{\pi\in\mathscr{P}}\sum_{1\leq j\leq|\pi|+1}\frac{z^{|\pi|+1}}{(|\pi|+1)!}u^{\chi(\pi)+[j=|\pi|+1]} \tag{26.15}$$

$$\begin{aligned}&= \frac{z^0}{0!}u^0 + \sum_{\pi\in\mathscr{P}}\frac{z^{|\pi|+1}}{(|\pi|+1)!}u^{\chi(\pi)}\sum_{1\leq j\leq|\pi|+1}u^{[j=|\pi|+1]}\\ &= 1 + \sum_{\pi\in\mathscr{P}}\frac{z^{|\pi|+1}}{(|\pi|+1)!}u^{\chi(\pi)}(|\pi|+u)\end{aligned} \tag{26.16}$$

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:

$$(1-z)M_z(z,u) - uM(z,u) = 0 \tag{26.17}$$

En (26.17) 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:

$$\begin{aligned} M_u(z,u) &= \frac{1}{(1-z)^u}\ln\frac{1}{1-z} \\ [z^n]M_u(z,1) &= H_n \\ M_{uu}(z,u) &= \frac{1}{(1-z)^u}\ln^2\frac{1}{1-z} \\ [z^n]M_{uu}(z,1) &= \sum_{1\le k\le n} H_k \\ &= (n+1)H_n - n \end{aligned}$$

Obtenemos:

$$\mathbb{E}[\chi_n] = H_n \tag{26.18}$$

$$\begin{aligned} \mathrm{var}[\chi_n] &= ((n+1)H_n - n) + H_n - H_n^2 && (26.19)\\ &= (n+2)H_n - n - H_n^2 && (26.20) \end{aligned}$$

26.4.3. Ordenamiento por selección

Volvamos a nuestro algoritmo de ordenamiento. Llamemos X_i al número de asignaciones a max en las distintas rondas del algoritmo (ciclo externo sobre i). Nos interesan:

$$\begin{aligned} \mathbb{E}\left[\sum_{1\le i\le n} X_i\right] &= \sum_{1\le i\le n} \mathbb{E}[X_i] \\ &= \sum_{1\le i\le n} H_i \\ &= (n+1)H_n - n \end{aligned} \tag{26.21}$$

Vemos que los X_i son variables independientes, así que:

$$\begin{aligned} \mathrm{var}\left[\sum_{1\le i\le n} X_i\right] &= \sum_{1\le i\le n} \mathrm{var}[X_i] \\ &= \sum_{1\le i\le n} (((i+2)H_i - i - H_i^2) \\ &= \sum_{1\le i\le n} iH_i + 2((n+1)H_n - n) - \frac{n(n+1)}{2} - \sum_{1\le i\le n} H_i^2 \end{aligned}$$

Atacamos las distintas sumas:

$$\begin{aligned}
\sum_{1\le i\le n} i H_i &= [z^n]\frac{z}{1-z}\mathrm{D}\frac{1}{1-z}\ln\frac{1}{1-z}\\
&= [z^n]\left(\frac{z}{(1-z)^3}\ln\frac{1}{1-z}-\frac{1}{(1-z)^3}\right)\\
&= [z^{n-1}]\frac{1}{(1-z)^3}\ln\frac{1}{1-z}-\binom{-3}{n}\\
&= \binom{n-1+2}{2}(H_{n-1+2}-H_2)-\binom{n+3-1}{3-1}\\
&= \frac{n(n+1)}{2}\left(H_{n+1}-\frac{3}{2}\right)-\frac{n(n+1)}{2}\\
&= \frac{n(n+1)}{2}\left(H_{n+1}-\frac{5}{2}\right)
\end{aligned}$$

La otra suma se puede resolver sumando por partes:

$$\begin{aligned}
\sum_{1\le i\le n} H_i^2 &= ((n+1)H_n-n)H_n-\sum_{1\le i\le n-1}\frac{(i+1)H_i-i}{i+1}\\
&= (n+1)H_n^2-nH_n-(n+1)H_n+n+H_n+(n-1)-H_n+1\\
&= (n+1)H_n^2-(2n+1)H_n+2n
\end{aligned}$$

Uniendo las piezas y simplificando:

$$\operatorname{var}\left[\sum_{1\le i\le n} X_i\right] = \frac{n^2+9n+6}{2}H_n-(n+1)H_n^2-\frac{5n(n+1)}{4}+\frac{n}{2} \tag{26.22}$$

O sea, asintóticamente el número de asignaciones (recuerde que un intercambio son tres asignaciones) en el método de selección es mínimo 0, máximo $3n$, promedio $3n\ln n$, varianza $3n^2\ln n/2$.

Bibliografía

[1] Horst H. von Brand: *Fundamentos de Informática*. git://csrg.inf.utfsm.cl/vonbrand/Ramos/trainwreck, Septiembre 2017. Versión 0.84.

Clase 27

Quicksort

Quicksort, debido a Hoare [5], 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 este, y todos los elementos mayores queden después. Con

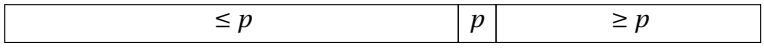


Figura 27.1 – Idea de Quicksort

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 27.2 indica una manera popular de

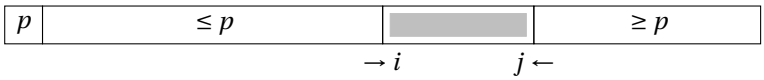


Figura 27.2 – Particionamiento en Quicksort

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 27.1. El listado 27.1 muestra una versión simple del programa, que elige siempre el primer elemento del rango como pivote.

27.1. 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

```
static double *a;

static int partition(const int lo, const int hi)
{
        int i = lo, j = hi + 1;
        double tmp;

        for(;;) {
                while(a[++i] < a[lo])
                        if(i == hi) break;
                while(a[lo] < a[--j])
                        if(j == lo) break;
                if(i >= j) break;
                tmp = a[i]; a[i] = a[j]; a[j] = tmp;
        }
        tmp = a[lo]; a[lo] = 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 27.1 – Versión simple de Quicksort

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 esto, 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:

$$C(n) = n - 1 + \frac{1}{n} \sum_{1 \le k \le n} (C(k-1) + C(n-k)) \quad C(0) = 0$$

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 [3, capítulo 19]), el segundo término da un coeficiente binomial:

$$\begin{aligned} C(n) &= 2 \sum_{0 \le k \le n} H_k - 2\binom{n}{1} \\ &= 2 \sum_{0 \le k \le n} H_k - 2n \end{aligned}$$

Para el primer término, consideremos cuántas veces aparece $1/k$ en la suma:

$$\begin{aligned} \sum_{0 \le k \le n} H_k &= \sum_{0 \le k \le n} \sum_{1 \le r \le k} \frac{1}{r} \\ &= \sum_{1 \le r \le n} \frac{n-r+1}{r} \\ &= (n+1) \sum_{1 \le r \le n} \frac{1}{r} - n \end{aligned} \tag{27.1}$$

Otra opción es sumar por partes (ver el apunte de Fundamentos de Informática, ecuación (1.34)). Esto da finalmente:

$$C(n) = 2(n+1)H_n - 4n \tag{27.2}$$

Sabemos que (ver [3, capítulo 18]) que $H_n = \ln n + \gamma + O(1/n)$, donde $\gamma = 0{,}57721\,56649\,01532\,65120$ con lo que $C(n) = 2n \ln n + O(n)$.

27.2. 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.

27.3. 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 [9].

Por el otro lado, McIllroy [7] 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 [2]. 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 [10] (garantizadamente $O(n \log n)$, pero mucho más lento que Quicksort) si se detecta un caso malo, como propone Musser [8].

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 pasamos 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 27.1.

Pero lo que realmente nos conviene es particionar en *tres* tramos, como muestra la figura 27.3. Esto es una variante del *problema de la bandera holandesa* (*Dutch national flag problem*, ver la

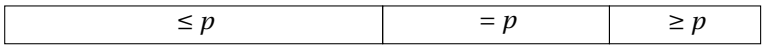


Figura 27.3 – Particionamiento ancho

figura 27.4) propuesto por Dijkstra [4]: 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

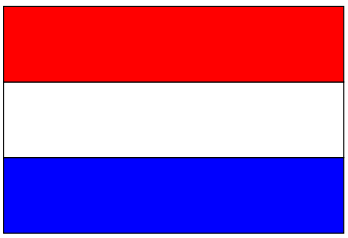


Figura 27.4 – La bandera holandesa

es usar el invariante de la figura 27.5. Código es como indica el listado 27.2.

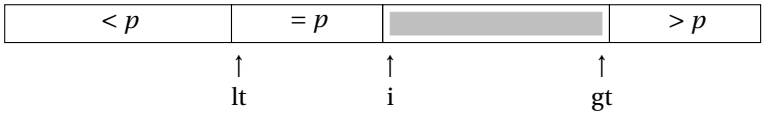


Figura 27.5 – Invariante para particionamiento ancho

Durante mucho tiempo se consideró poco práctico Quicksort con más de un pivote, hasta que en 2009 se adoptó una variante con dos pivotes en Java 7, el algoritmo de Yaroslavskiy, Bentley y

```
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 27.2 – Partición ancha

Bloch [11]. Desde entonces se han analizando opciones con más de un pivote, una selección de referencias recientes es [1,6]. Resulta que su mejor rendimiento se debe a efectos de memoria caché.

Bibliografía

[1] Martin Aumüller, Martin Dietzfelbinger, and Pascal Klaue: *How good is multi-pivot quicksort?* ACM Transactions on Algorithms, 13(1):8:1 – 8:47, December 2016.

[2] Jon Louis Bentley and M. Douglas McIllroy: *Engineering a sort function.* Software: Practice and Experience, 23(11):1249–1265, November 1993.

[3] Horst H. von Brand: *Fundamentos de Informática.* git://csrg.inf.utfsm.cl/vonbrand/Ramos/trainwreck, Septiembre 2017. Versión 0.84.

[4] Edsger W. Dijkstra: *A Discipline of Programming.* Prentice Hall, 1976.

[5] C. A. R. Hoare: *Quicksort.* Computer Journal, 5(1):10–15, April 1962.

[6] Shrinu Kushagra, Alejandro López-Ortiz, J. Ian Munro, and Aurick Qiao: *Multi-pivot quicksort: Theory and experiments.* In *Proceedings of the Meeting on Algorithm Engineering & Expermiments, ALENEX*, pages 47–60, Portland, OR, USA, January 2014. SIAM.

[7] M. Douglas McIllroy: *A killer adversary for Quicksort.* Software: Practice and Experience, 29(4):341–344, April 1999.

[8] David Musser: *Introspective sorting and selection algorithms.* Software: Practice and Experience, 27(8):983–993, August 1997.

[9] Robert Sedgewick and Philipe Flajolet: *An Introduction to the Analysis of Algorithms.* Addison-Wesley, second edition, 2013.

[10] J. W. J. Williams: *Algorithm 232 – Heapsort.* Communications of the ACM, 7(6):347–348, June 1964.

[11] Vladimir Yaroslavskiy: *Dual-pivot quicksort algorithm.* http://codeblab.com/wp-content/uploads/2009/09/DualPivotQuicksort.pdf, September 2009.

Clase 28

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 (suma de los costos de los arcos) mínimo. Una solución a este problema da el algoritmo de Kruskal [3], (algoritmo 28.1, que ya discutimos en el capítulo 13). 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 un programa eficiente (y

Algoritmo 28.1: 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)
```

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].

28.1. 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 un subconjunto

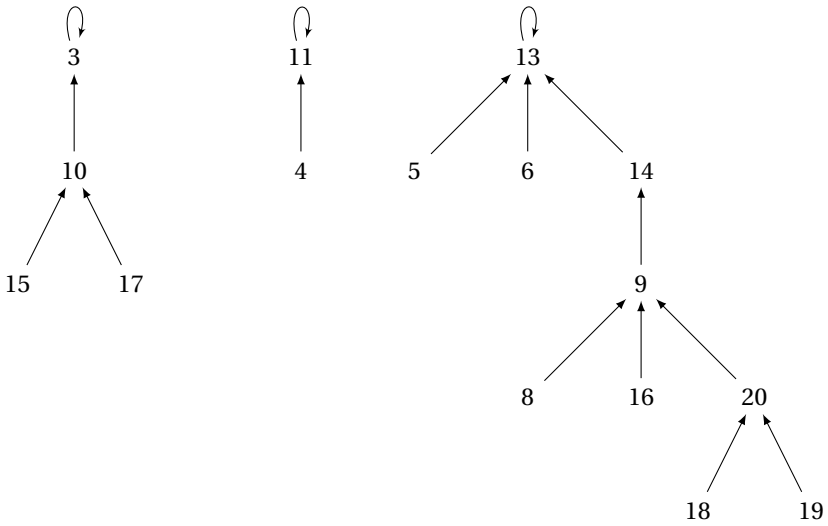


Figura 28.1 – Esquema de la estructura para *union-find*.

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 28.1. 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, esta 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 28.2, 28.3, y 28.4.

Algoritmo 28.2: Algoritmo para crear conjuntos

```
procedure MakeSets(V)
    for v ∈ V do
        rank[v] ← 0
        parent[v] ← v
    end
end
```

Algoritmo 28.3: Algoritmo para unir conjuntos

```
procedure union(u, v)
    u ← find(u)
    v ← find(v)
    if rank[u] > rank[v] then
        parent[v] ← u
    else
        parent[u] ← v
        if rank[u] = rank[v] then
            rank[v] ← rank[v] + 1
        end
    end
end
```

Algoritmo 28.4: Algoritmo para encontrar representante

```
function find(v)
    while v ≠ parent[v] do
        v ← parent[v]
    end
    return v
end
```

Bibliografía

[1] Jeff Erickson: *Algorithms, etc.* http://jeffe.cs.illinois.edu/teaching/algorithms, January 2015. Department of Computer Science, University of Illinois at Urbana-Champaign.

[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 29

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.

29.1. Análisis de la versión simple

Veamos algunas propiedades cruciales de nuestra estructura inicial bajo las operaciones mencionadas:

Propiedad 1: *Para todo v que no es raíz,* $\text{rank}[v] < \text{rank}[\text{parent}[v]]$.

> *Demostración.* 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.*

> *Demostración.* 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 29.1) tiene 2^k nodos y altura k.

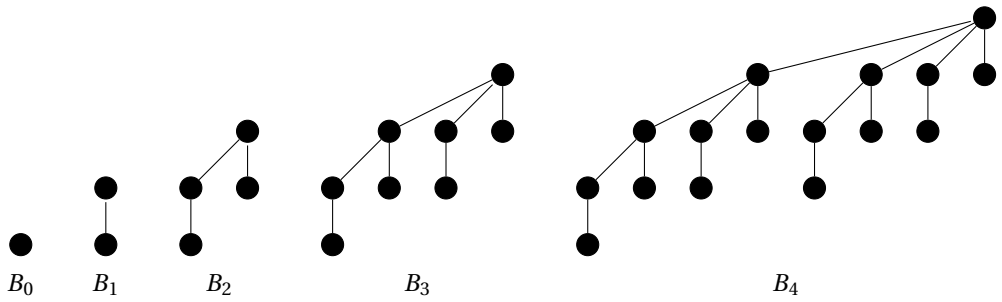


Figura 29.1 – Árboles binomiales

29.2. 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 29.2 ilustra el efecto de buscar 20 y comprimir caminos. Los árboles de las ilustraciones definitivamente no son resultado de nuestros algoritmos, simplemente sirven para mostrar el efecto de las operaciones. El algoritmo resultante 29.1 es la versión modificada de find. La idea es pagar un costo extra en

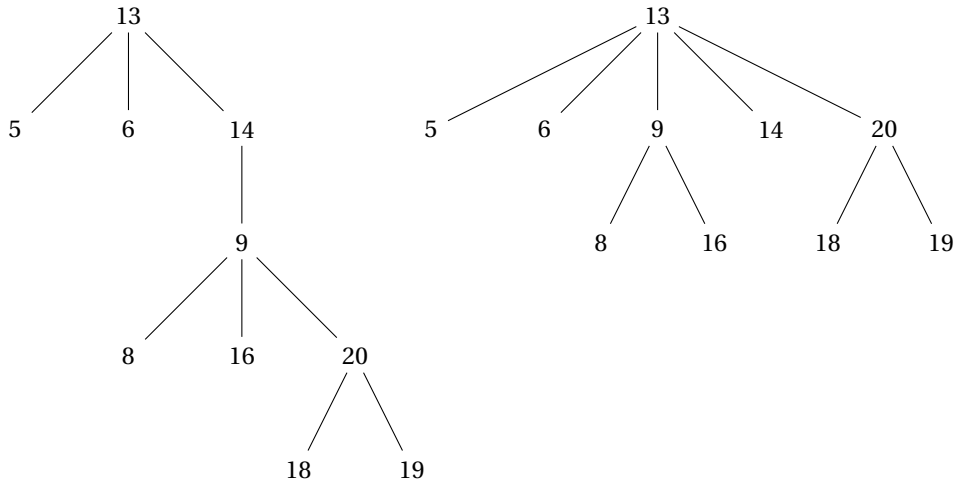


Figura 29.2 – Acortar caminos (*path compression*) al buscar 20.

las operaciones find en la esperanza de ahorrar en operaciones futuras. Una variante más simple es cambiar abuelos por padres, ver el algoritmo 29.2, que difiere del original en una única línea. La figura 29.3 ilustra el efecto al buscar 19.

29.3. 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.

Algoritmo 29.1: 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
end
```

Algoritmo 29.2: 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
end
```

Para $g\colon \mathbb{R} \to \mathbb{R}$ tal que para $x > 1$ siempre es $g(x) < x$ definimos:

$$g^*(x) = \begin{cases} 0 & x \leq 1 \\ 1 + g^*(g(x)) & x > 1 \end{cases} \tag{29.1}$$

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 $\log_2^* x$, donde el logaritmo es en base 2 (¡somos computines!). Es claro que $\log_2^* n$ crece extremadamente lento:

$$\log_2^* n = \begin{cases} 0 & n \leq 1 \\ 1 & 1 < n \leq 2 \\ 2 & 2 < n \leq 2^2 \\ 3 & 2^2 < n \leq 2^4 \\ 4 & 2^4 < n \leq 2^{16} \\ 5 & 2^{16} < n \leq 2^{65\,536} \end{cases}$$

El análisis clásico de esta estructura (Hopcroft y Ullman [3], Tarjan [7]) es complejo. Acá seguimos la idea de Seidel y Sharir [6], 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

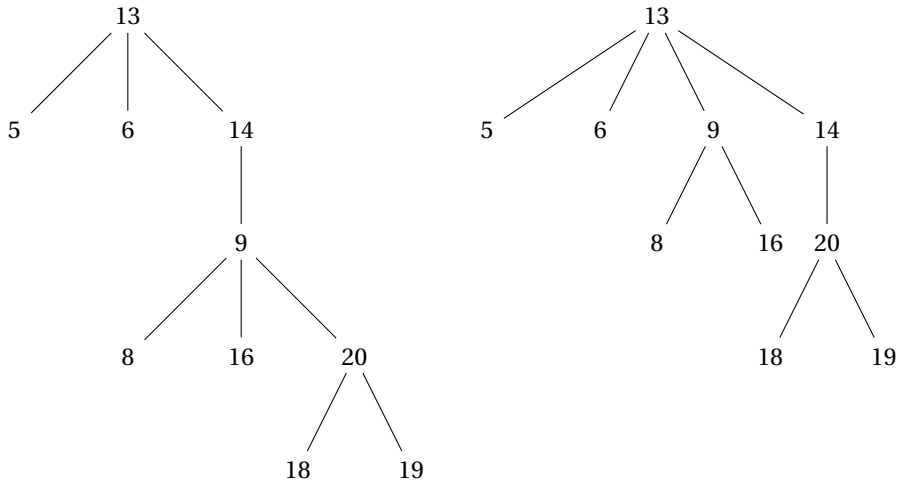


Figura 29.3 – Acortar caminos (*path compression*) con abuelos desde 20.

Algoritmo 29.3: 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
end
```

Algoritmo 29.4: 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
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.

Lema 29.1. $T(m,n,r) \leq nr$

Demostración. Cada nodo puede cambiar padre a lo más r veces, ya que rank siempre aumenta. □

Sea $\mathscr{F}$ un bosque de n nodos con rank máximo r y una secuencia C de m operaciones compress sobre $\mathscr{F}$, y sea $T(\mathscr{F}, C)$ el número total de asignaciones de punteros ejecutados por esta secuencia. Divida el bosque en dos sub-bosques, un bosque «bajo» $\mathscr{F}_-$ con los nodos de rank$[v] \le s$ y el bosque «alto» $\mathscr{F}_+$ 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 29.4 que muestra un bosque (un único árbol en el ejemplo) dividido en sub-bosques.

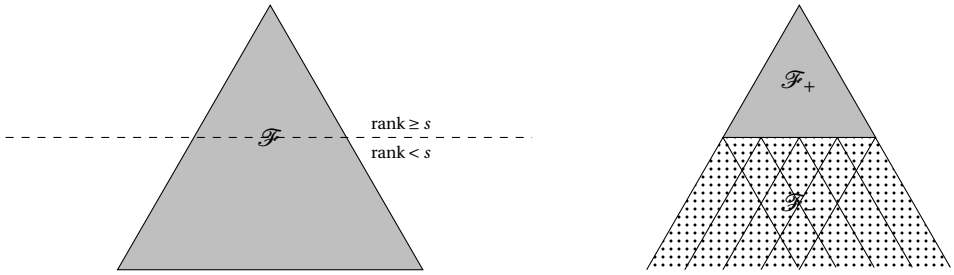


Figura 29.4 – Dividiendo el bosque según rank

Cualquier secuencia de operaciones compress sobre $\mathscr{F}$ puede descomponerse en una secuencia de operaciones compress sobre $\mathscr{F}_+$ y una secuencia de operaciones compress y shatter sobre $\mathscr{F}_-$ con el mismo costo. La modificación es prohibir a un nodo bajo tener un padre alto, ver la figura 29.5. El punto de hacer esto es descomponer la secuencia en secuencias menores. Al dividir el bosque en

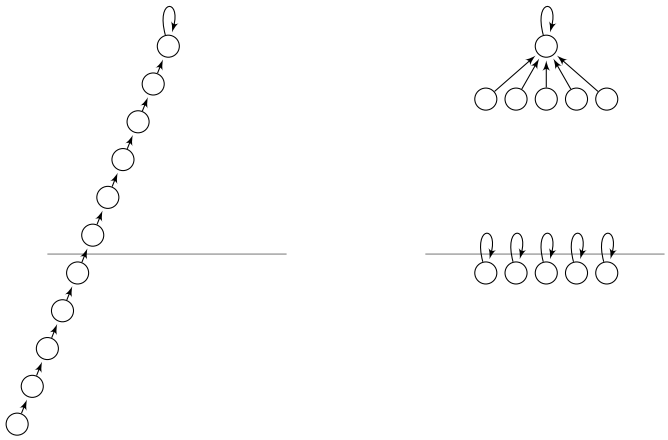


Figura 29.5 – División de una operación compress

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 que acota el costo de la secuencia de operaciones.

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$[v] \le s$), o cruza el rango y hay que subdividir la operación.

La última asignación del algoritmo 29.5 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

Algoritmo 29.5: 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
end
```

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 operaciones shatter. En vista de las consideraciones anteriores se cumple la desigualdad:

$$T(\mathscr{F}, C) \leq T(\mathscr{F}_-, C_-) + T(\mathscr{F}_+, C_+) + m_+ + n \tag{29.2}$$

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 lema 29.1 implica:

$$T(\mathscr{F}_+, C_+) \leq \frac{rn}{2^s}$$

Fijemos $s = \lfloor \log_2 r \rfloor$, 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 vale en *cualquier* bosque $\mathscr{F}$ y para toda secuencia C, y como $T(m, n, r)$ es creciente con sus dos primeros argumentos, hemos demostrado que para $T'(m, n, r) = T(m, n, r) - m$:

$$\begin{aligned} T'(m, n, r) &\leq T'(m_-, n, \lfloor \log_2 r \rfloor) + 2n \\ &\leq T'(m, n, \lfloor \log_2 r \rfloor) + 2n \end{aligned}$$

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 29.2. $T(m,n,r) \leq m + 2n \log_2^* r$

El teorema 29.2 puede mejorarse. En la demostración usamos la cota del lema 29.1, que nuestro teorema 29.2 mejora, y puede usarse recursivamente, aumentando cada vez algo la dependencia de m mientras disminuye la de r. Erickson [2, clase 17] completa el desarrollo. Se concluye lo siguiente: La función de Ackermann [1] (aunque usamos la forma de dos argumentos definida por Péter [4] y Robinson [5]) se define como:

$$A(m,n) = \begin{cases} n+1 & m = 0 \\ A(m-1,1) & m > 0 \wedge n = 0 \\ A(m-1, A(m,n-1)) & m > 0 \wedge n > 0 \end{cases}$$

Para algunos ejempplos, vemos que:

$$\begin{aligned}
A(0,n) &= n+1 \\
A(1,n) &= A(0, A(1,n-1)) \\
&= A(1,n-1)+1 \\
&\vdots \\
&= A(1,0)+n \\
&= A(0,1)+n \\
&= n+2 \\
&= 2+(n+3)-3 \\
A(2,n) &= A(1, A(2,n-1)) \\
&= A(1, A(1, A(2,n-2))) \\
&\vdots \\
&= A(1, A(1, A(1, \ldots, A(1, A(2,0))))) \\
&= A(1, A(1, A(1, \ldots, A(1,3)) \\
&\vdots \\
&= 2n+3 \\
&= 2(n+3)-3 \\
A(3,n) &= A(2, A(3,n-1)) \\
&\vdots \\
&= A(2, A(2, A(2, \ldots, A(2, A(2,0))) \\
&= A(2, A(2, A(2, \ldots, A(2,3))) \\
&\vdots \\
&= 2^{n+3} - 3
\end{aligned}$$

Esencialmente, cada aumento de m significa «aplique la función anterior n veces», $A(1,n)$ es «$n+3$ veces sumar 1», $A(2,n)$ es «$n+3$ veces sumar 2», $A(3,n)$ es «$n+3$ veces multiplicar por 2», $A(4,n)$ es «$n+3$ veces elevar a la potencia 2», lo que da:

$$\begin{aligned} A(4,n) &= {}^{n+3}2-3 \\ &= \underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{n+3}-3 \end{aligned}$$

O sea, por ejemplo:

$$A(4,3) = 2^{2^{65536}} - 3$$

Se ve que esta función crece extremadamente rápido.

Se definen dos funciones inversas, que crecen extremadamente lento:

$$\alpha(n) = A^{-1}(n,n) \tag{29.3}$$

$$\alpha(m,n) = \min\{i \geq 1 : A(i, \lfloor m/n \rfloor) \geq \log_2 n\} \tag{29.4}$$

De los valores anteriores para $A(m,n)$ vemos que para todos los valores imaginables de m y n es $\alpha(m,n) \leq 5$. Claro que en estricto rigor no está acotada por una constante.

En estos términos, Tarjan [7] demuestra que toda secuencia intercalada de $m \geq n$ operaciones find y $n-1$ operaciones union toma $O(m\alpha(m,n))$ tiempo.

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 $\log_2^* r = \log_2^* V - 1$. Las operaciones con clases de equivalencia aportan $O(V)$ para los union, los find aportan $O(2E + 2V \log_2^* V)$ para un costo total de:

$$O(E \log E) + O(V + 2E + 2V \log_2^* V) = O(E \log E)$$

Esto es dominado por el costo de ordenar los arcos. Usar la cota de Tarjan no mejora esto, cambia el segundo término a $O(2E\alpha(2E,V))$.

Bibliografía

[1] Wilhelm Ackermann: *Zum Hilbertschen Aufbau der reellen Zahlen.* Mathematische Annalen, 99(1):118–133, Dezember 1928.

[2] Jeff Erickson: *Algorithms, etc.* http://jeffe.cs.illinois.edu/teaching/algorithms, January 2015. Department of Computer Science, University of Illinois at Urbana-Champaign.

[3] John E. Hopcroft and Jeffrey D. Ullman: *Set merging algorithms.* SIAM Journal of Computing, 2(4):294–303, 1973.

[4] Rózsa Péter: *Konstruktion nichtrekursiver Funktionen.* Mathematische Annalen, 111(1):42–60, Oktober 1934.

[5] Raphael M. Robinson: *Recursion and double recursion.* Bulletin of the American Mathematical Society, 54(10):987–993, October 1948.

[6] Raimund Seidel and Micha Sharir: *Top-down analysis of path compression.* SIAM Journal of Computing, 34(3):515–525, 2005.

[7] Robert E. Tarjan: *Efficiency of a good but not linear set union algorithm.* Journal of the ACM, 22(2):215–225, April 1975.

Clase 30

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». Otro caso es el que vimos recién, pagamos un costo ahora en la esperanza de ahorrar más adelante. La definición es simple:

Definición 30.1. 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). Incluso la biblioteca estándar de plantillas de C++ (STL, *Standard Template Library*, revisar por ejemplo el clásico de Stroustrup [5], o la documentación de SGI [3, 4]) ofrece una colección de estructuras de datos y algoritmos, especificando complejidades promedio y, donde aplicable, amortizadas para cada uno.

Hay tres métodos principales (comparar con CLRS [2, 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.

30.1. 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 30.1. Hemos omitido el verificar si el *stack* contiene elementos o llenó el arreglo. La pregunta es qué hacer

```
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 30.1 – Operaciones sobre un *stack*

si el arreglo se llena. La biblioteca de C ofrece la opción de solicitar memoria, copiar el contenido de un área al comienzo de esta y liberar el original (ver `realloc(3)` en su Unix preferido o refiérase a Wikipedia [6]). 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 objeto, 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 objetos). 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:

$$1+(1+1)+(2+1)+1+(4+1)+1+1+1+\cdots+(2^k+1)+1+\cdots$$

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:

$$\begin{aligned}n+\sum_{0\le k\le\lfloor\log_2(n-1)\rfloor}2^k &= n+2^{\lfloor\log_2(n-1)\rfloor+1}-1\\ &\le n+2^{\log_2(n-1)+1}-1\\ &= n+2(n-1)-1\\ &= 3n-3\\ &< 3n\end{aligned}$$

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.

30.2. 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.

30.2.1. 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 30.1. *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 de una secuencia de operaciones nunca sobrepasa 2 cambios de bit por incremento. □

30.2.2. Método agregado

Para contraste, una demostración usando el método agregado sería:

Demostración. Consideremos una secuencia de incrementos, y consideremos cuántas veces cambia cada bit. Si el contador llega a n (n operaciones) el bit más alto es $\lfloor \log_2 n \rfloor + 1$. Claramente, $\mathsf{a}[0]$ cambia cada vez, $\mathsf{a}[1]$ cambia cada dos incrementos, ..., $\mathsf{a}[\mathsf{k}]$ cambia cada 2^k incrementos, y así sucesivamente. El costo total de los cambios a $\mathsf{a}[0]$ es n, los cambios a $\mathsf{a}[1]$ tienen costo total el piso de $n/2, \ldots,$ cambios de $\mathsf{a}[\mathsf{k}]$ cuestan el piso de $n/2^k$, y así sigue.

En total, el costo es:

$$\begin{aligned}
\lfloor n \rfloor + \left\lfloor \frac{n}{2} \right\rfloor + \cdots + \left\lfloor \frac{n}{2^{\lfloor \log_2 n \rfloor + 1}} \right\rfloor &= \sum_{0 \le k \le \log_2 n + 1} \left\lfloor \frac{n}{2^k} \right\rfloor \\
&\le n \sum_{0 \le k \le \log_2 n + 1} 2^{-k} \\
&< n \sum_{k \ge 0} 2^{-k} \\
&= 2n
\end{aligned}$$

De aquí el costo amortizado es menor a 2. □

30.2.3. 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 $\Phi\colon \mathscr{S} \to \mathbb{R}$ de estados $\mathscr{S}$ 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 $\Phi(n) \geq 0$. 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* (número de reservas de dígitos) 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.

30.2.3.1. Diseño de una función potencial

Esto claramente es la parte más difícil de esta técnica. En el caso del método contable, hay que definir un cobro adicional, que también es complejo de hacer correctamente, posiblemente más que lo que discutimos. La discusión presente es de Belleville [1].

Si tenemos una estructura que maneja inserciones y eliminaciones, usualmente podemos construir un potencial adecuado considerando el costo adicional que significa el elemento una vez que se ha añadido. Por ejemplo, consideremos un *stack* con operaciones `push`, `pop` y `multipop` (la última elimina un número dado de objetos del *stack*). Una vez que un elemento se ha añadido al *stack*, lo que queda por hacer con él es eliminarlo nuevamente, lo que sugiere usar como potencial el número de elementos actualmente presentes (asociar un valor 1 a cada elemento añadido). Esto sugiere:

$$\Phi(D_i) = \text{número de elementos en } D_i$$

Es fácil demostrar con esto que con esto el costo amortizado por operación resulta constante. En el contador binario, nuestra función potencial es el número de unos, debemos ahorrar cada vez que un cero pasa a uno para volverlo a cero.

En estos casos el potencial se distribuye entre los objetos de la estructura, pero hay casos en que no conviene tener una asociación demasiado estrecha, el potencial viene de la conformación de la estructura misma.

Como ejemplo, consideremos una estructura simple que soporta operaciones de inserción y búsqueda. Si usamos un arreglo desordenado, si hay n elementos el costo de una inserción es constante y el costo de una búsqueda es $O(n)$; si el arreglo se mantiene ordenado la búsqueda tiene costo $O(\log n)$ mientras la inserción tiene costo $O(n)$. Buscamos una estructura simple que de mejor rendimiento amortizado.

Supongamos n elementos, y sea $k = \lceil \log_2(n+1) \rceil$ (requerimos k bits para representar números hasta n), y sea $\langle n_{k-1} n_{k-2} \ldots n_0 \rangle$ la representación de n en binario. Usaremos k arreglos ordenados $A_0, A_1, \ldots, A_{k-1}$. El arreglo A_i o es vacío o contiene exactamente 2^i elementos, dependiendo del valor de n_i. Aunque cada arreglo está ordenado, no hay relación entre los elementos de los arreglos.

Para buscar un objeto en esta estructura, buscamos en los k arreglos. Hay $O(\log n)$ arreglos, la búsqueda en cada uno de ellos toma $O(\log n)$, con lo que el costo total de la búsqueda es $O(\log^2 n)$ (un análisis más fino da una cota algo mejor).

Para insertar, creamos un nuevo arreglo con un único elemento, y vamos intercalando sucesivamente con A_0, el resultado se intercala con A_1 si no está vacío, y así sucesivamente hasta llegar a una posición vacía. El algoritmo 30.1 da el detalle. El peor caso de la operación `insert` es $\Theta(n)$, por

Algoritmo 30.1: Inserción en la estructura propuesta

```
procedure insert(x)
    i ← 0
    A ← x
    while A_i not empty do
        A ← merge(A_i, A)
        A_i ← empty
        i ← i + 1
    end
    A_i ← A
end
```

ejemplo si la estructura ya contiene $n = 2^t - 1$ elementos.

Para analizar el costo de una secuencia de n operaciones `insert` Definimos la función potencial:

$$\Phi(D_i) = \sum_{0 \le j \le k-1} (k-j) n_j 2^j$$

donde $k = \lceil \log_2(n+1) \rceil$ y $n_j = [A_j \text{not empty}]$. La justificación es que los 2^j elementos actualmente en el arreglo j potencialmente deben participar en operaciones `merge` para migrar a los arreglos $j+1, j+2, \ldots, k$.

Dividimos la operación en dos partes: crear el arreglo A con un elemento y las operaciones `merge` hasta que haya un solo arreglo de tamaño 2^j para cada j. El costo de la primera parte es constante, y aumenta el potencial en k. Su aporte al costo amortizado es $k+1$.

Para las operaciones de intercalación, cada vez que se invoca sobre un par de arreglos de 2^j elementos el costo es 2^{j+1} (el tiempo por elemento es constante). Como se mueven 2^{j+1} elementos de A_j a A_{j+1}, el potencial disminuye en 2^{j+1}. El costo amortizado de cada `merge` 0.

En resumen, el costo amortizado de insert es $O(k+1)$, que es $O(\log n)$.

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. Repita el análisis del arreglo extensible (*stack*) usando el método potencial.

3. Consideremos una cola de prioridad con operaciones dadas en el algoritmo 30.2.

Algoritmo 30.2: Operaciones sobre la cola de prioridad

```
procedure Init(n)
    for i ← 1 to n do
        a[i] ← true
    end
end

procedure Delete(i)
    a[i] ← false
end

function DeleteMin()
    i ← 1
    while ¬a[i] do
        i ← i + 1
    end
    if i ≤ n then
        Delete(i)
        return i
    else
        return 0
    end
end
```

a) Analice el tiempo de ejecución de cada procedimiento.

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 un algoritmo diferente con costos $O(1)$ en el peor caso para Delete y DeleteMin.

4. 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 $k = L/2$?

5. 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 $m = -1$.

Algoritmo 30.3: 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
end
```

a) Dé un ejemplo de dos representaciones diferentes de un número.

b) Demuestre que los procedimientos de los algoritmos 30.3 y 30.4 son correctos.

c) Usando los procedimientos de los algoritmos 30.3 y 30.4 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 30.4: 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
end
```

Bibliografía

[1] Patrice Belleville: *Amortized analysis: A summary*. https://www.ugrad.cs.ubc.ca/~cs320/2010W2/handouts/aa-nutshell.pdf, February 2011.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald R. Rivest, and Clifford Stein: *Introduction to Algorithms*. MIT Press, third edition, 2009.

[3] Silicon Graphics International: *Standard Template Library programmer's guide*. http://www.sgi.com/tech/stl, June 2000. (version 3.3).

[4] Silicon Graphics International: *STL complexity specifications*. http://www.sgi.com/tech/stl/complexity.html, June 2000.

[5] Bjarne Stroustrup: *The C++ Programming Language*. Addison-Wesley, fourth edition, 2013.

[6] Wikipedia: *C Standard Library*. https://en.wikipedia.org/wiki/C_standard_library, November 2016. Accessed 2016-11-07.

Clase 31

Ejemplos de análisis amortizado

Trataremos en detalle algunos ejemplos adicionales de análisis amortizado para estructuras simples, de aplicación práctica. Parte de lo siguiente se adapta de Fiebrink [1].

31.1. Listas autoorganizantes

Este es un ejemplo interesante en que el análisis compara respeto del algoritmo óptimo. El modelo no es demasiado realista, Es parte de la discusión de Tarjan [7], quien resume resultados previos y discute varias otras situaciones afines, de aplicación directa.

El modelo es de una lista, que se accede secuencialmente. En muchas aplicaciones, las referencias tienen localidad, un objeto accedido al instante t es más probable de ser accedido nuevamente poco después de t. Para favorecer accesos futuros, el elemento buscado se mueve al principio de la lista (esta heurística se llama *Move To Front*, abreviado MTF).

Acceder al objeto en la posición i tiene costo i, y un par de objetos vecinos pueden intercambiarse en tiempo constante. Con esto, el acceder al objeto en posición i tiene costo $2i-1$ (i para hallarlo, luego $i-1$ intercambios para llevarlo al comienzo).

Usamos análisis amortizado para demostrar que el rendimiento de MTF siempre está dentro de un factor de 4 de *cualquier* algoritmo, incluso del óptimo, sin suposiciones sobre localidad de referencia. Sea A un algoritmo cualquiera, definimos el potencial de MTF al instante t como dos veces el número de objetos cuyo orden en la lista de MTF difiere del orden de la lista de A (el número de inversiones). Por ejemplo, si la lista de A es $\langle a,b,c,d,e\rangle$ y la de MTF es $\langle a,d,b,c,e\rangle$, $Phi = 2\cdot 2 = 4$ por b y c respecto de d. Claramente, el potencial nunca es negativo, y a $t=0$ es cero, ambos comienzan con la misma secuencia. El costo amortizado es una cota superior al costo de una secuencia de operaciones.

Considere el acceder al objeto x, que está en la posición k de la lista de MTF y en la posición i en la de A. El costo en MTF es $2k-1$, el costo en A es i. Mover x al comienzo invierte el orden de los $k-1$ pares de elementos en posiciones 1 a $k-1$ con x, todos los demás pares se mantienen. En la lista de A hay $i-1$ objetos antes de x, todos ellos terminan después de x luego de promover a x en MTF. Se añaden a lo más $\min\{k-1, i-1\}$ inversiones entre las listas. Las demás reorganizaciones (a lo menos $k-1-\min\{k-1,i-1\}$) resultan en eliminar inversiones (las posiciones ahora concuerdan entre A y MTF). En consecuencia, el cambio de potencial en este acceso está acotado por arriba por:

$$2(\min\{k-1,i-1\} - (k-1-\min\{k-1,i-1\})) = 4\min\{k-1,i-1\} - 2(k-1)$$

En consecuencia, el costo amortizado de esta operación es:

$$\begin{aligned} a &= c + \Delta\Phi \\ &\leq 2k - 1 + 4\,\text{mín}\{k-1, i-1\} - 2(k-1) \\ &\leq 4\,\text{mín}\{k-1, i-1\} \\ &\leq 4i \end{aligned}$$

O sea, el costo de acceso amortizado de MTF está acotado por 4 veces el de A.

Pero lo anterior no considera reorganizaciones que hace A. Sea que A intercambia dos objetos. Esto no introduce costo adicional para MTF, pero el potencial aumenta o disminuye en 2 (dos objetos cambian de posición), y aumenta el costo de acceso en A en 1. La cota se mantiene, el costo amortizado aumenta a lo más en 2 y la cota aumenta en 4. Esto vale independiente del número de intercambios que hace A.

31.2. Splay trees

Supondremos que la terminología sobre árboles binarios de búsqueda junto con los algoritmos relevantes de búsqueda, inserción y eliminación son conocidos. Si no es así, revise sus apuntes de estructuras de datos.

Recuerde que la *profundidad* de un nodo en un árbol binario es su distancia a la raíz, y su *altura* es la distancia a su hoja descendiente más lejana. La altura del árbol es simplemente la altura de su raíz. Llamaremos *tamaño* de un nodo al número de nodos en su subárbol. El tamaño del árbol es el tamaño de su raíz.

Un árbol de altura h tiene a lo más 2^h hojas (un simple ejercicio de inducción), por lo que un árbol con n hojas tiene altura al menos $\lceil \log_2 n \rceil$. En el peor caso, el tiempo para una búsqueda, inserción o eliminación es proporcional a la altura del árbol, por lo que nos interesa minimizar la altura. Lo mejor que podemos hacer es mantener *árboles perfectamente balanceados*, en los cuales cada subárbol (recursivamente) tiene lo más exactamente posible la mitad de los nodos. Esto asegura que la altura sea exactamente $\lceil \log_2 n \rceil$ (otro ejercicio simple de inducción), con lo que el peor caso del tiempo de búsqueda es $O(\log n)$. Sin embargo, si partimos con un árbol perfectamente balanceado una secuencia maliciosa de inserciones y eliminaciones puede hacerlo arbitrariamente desbalanceado, llevando los tiempos de búsqueda a $\Theta(n)$. Para evitar esto, debemos modificar el árbol periódicamente para mantener el balance (al menos aproximadamente). Hay varios métodos para hacer esto, que llevan a árboles binarios con nombres diversos: árboles AVL, árboles rojo-negro, árboles balanceados en altura o en peso, y otros más. Plavec, Vranesic y Brown [6] resumen evaluación experimental de estas y otras alternativas. El problema de todas ellas es que almacenan información adicional necesaria para mantener el balance (pueden reusarse bits en desuso para ello, pero eso lo hace poco portable), y operaciones engorrosas de programar.

Una alternativa simple, que no requiere espacio adicional, relativamente simple de programar y con tiempo de ejecución amortizado $O(\log n)$ para todas las operaciones son los *splay trees* propuestos por Sleator y Tarjan [8]. La idea básica es promover a la raíz al nodo resultado de una búsqueda, reorganizando el árbol en el proceso. Resulta un árbol agradablemente balanceado, y en el caso común en que los accesos se aglomeran, los elementos buscados frecuentemente estarán cerca de la raíz. Eso sí que tienen la desventaja que incluso operaciones de «solo lectura» (búsquedas) reorganizan el árbol, lo que hace que esta estructura lamentablemente no pueda usarse en forma concurrente.

31.3. 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 [3]. Pero estas estructuras son pesadas y complejas de programar, además de lentas en la práctica; Fredman, Sedgewick, Sleator y Tarjan [2] 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 [9], una estructura afín nueva es el *rank-pairing heap* de Haeupler, Sen y Tarjan [4], quienes revisan las diversas estructuras propuestas; el resumen y análisis más reciente de estas y otras variantes es el de Iacono y Özkan [5]. El consenso parece ser que las diferencias son menores, con *pairing heaps* la más simple y algo más eficiente en la práctica.

31.3.1. 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 objetos, cada uno con una *clave*. Las operaciones a soportar por una cola de prioridad son:

make_heap: Retorna un nuevo *heap* vacío.

empty(H): Retorna si el *heap* H es vacío.

insert(H, x): Inserte el objeto x en el *heap* H, que no contiene x previamente. La clave se supone que es parte de x.

find_min(H): Retorna un objeto con clave mínima en H, sin cambiar este. Es un error invocar esta operación sobre un *heap* vacío.

delete_min(H): Retorna un objeto con clave mínima en H, eliminándolo del *heap*. Es un error invocar esta operación sobre un *heap* vacío.

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 prerrequisito que H_1 y H_2 no tengan objetos 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 $\Omega(\log n)$ (una manera de ordenar n elementos es insertarlos en un *heap* y luego extraerlos en orden).

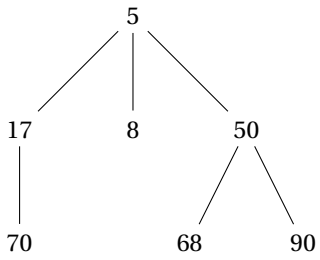


Figura 31.1 – Un ejemplo de *pairing heap*

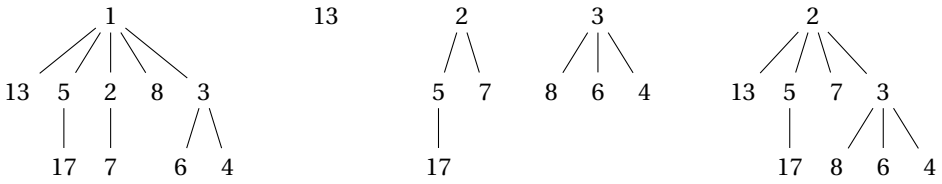
31.3.2. La estructura *pairing heap*

La idea es representar el *heap* en forma de árbol, con cada nodo conteniendo un objeto con clave menor que sus descendientes, como muestra la figura 31.1. Por ahora supondremos esta representación 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 (esencialmente la operación delete_min(H) en ese subárbol) y uniéndolo con el resto. Luego insertamos el objeto x modificado. 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 objetos 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 31.2 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.



(a) Original (b) Parear (c) Acumular

Figura 31.2 – Operación delete_min

31.3.3. 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 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_key.

31.3.4. 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:

$$\Phi(H) = \sum_{x \in H} r(x) \tag{31.1}$$

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] Rebecca Fiebrink: *Amortized analysis explained*. https://www.cs.princeton.edu/~fiebrink/423/AmortizedAnalysisExplained_Fiebrink.pdf, September 2007. Department of Computer Science, Princeton University.

[2] 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.

[3] 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.

[4] Bernhard Haeupler, Siddharta Sen, and Robert E. Tarjan: *Rank-pairing heaps*. SIAM Journal of Computing, 40(6):1463–1485, 2011.

[5] 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.

[6] Franjo Plavec, Zvonko G. Vranesic, and Stephen Dean Brown: *On Digital Search Trees – A simple method for constructing balanced binary trees*. In *ICSOFT*, 2007.

[7] Daniel D. Sleator and Robert E. Tarjan: *Amortized efficiency of list updates and paging rules*. Communications of the ACM, 28(2):202–208, February 1985.

[8] Daniel D. Sleator and Robert E. Tarjan: *Self-adjusting binary search trees*. Journal of the ACM, 32(3):625–686, July 1985.

[9] John T. Stasko and Jeffrey Scott Vitter: *Pairing heaps: Experiments and analysis*. Communications of the ACM, 30(3):234–249, March 1987.

Clase 32

Hashing

Una *tabla hash* es una estructura que almacena un conjunto de objetos, permitiendo determinar rápidamente si un objeto dado está o no presente. Generalmente se asocia una *clave* que define el objeto, como es un nombre, un rol o un número de inventario. La idea central es elegir una *función de hashing* h que mapea toda clave posible a un entero pequeño $h(x)$. Almacenamos el objeto asociado a x en la posición $h(x)$ de un arreglo, este arreglo es la *tabla hash*. La importancia de esta idea es que ofrece algoritmos que dan tiempos de búsqueda constantes, independientes del número de datos almacenados. Por esta razón es la técnica de búsqueda preferida. Lo malo es que el buen rendimiento es solo en valor esperado, los peores casos son *muy* malos (pero extremadamente poco probables si se toman precauciones apropiadas).

El análisis de los algoritmos discutidos requiere rudimentos de probabilidades, para notación y discusión de los conceptos usados revise el apéndice G. Usaremos varias cotas simples, pero extraordinariamente útiles, que no se han tratado anteriormente. Las demostraciones se encuentran en el apéndice.

32.1. Algunos resultados previos

Requeriremos algunos resultados auxiliares para analizar *hashing*. Primero, un límite clásico:

Lema 32.1. *Para todo $x \in \mathbb{R}$ se tiene que:*

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

Demostración. Como e^x es continua, podemos escribir:

$$\begin{aligned}
\lim_{n\to\infty}\left(1+\frac{x}{n}\right)^n &= \exp\left(\lim_{n\to\infty} n\ln\left(1+\frac{x}{n}\right)\right)\\
&= \exp\left(\lim_{n\to\infty}\frac{\ln(1+x/n)}{1/n}\right)\\
&= \exp\left(\lim_{n\to\infty}\frac{\frac{-x/n^2}{1+x/n}}{-1/n^2}\right)\\
&= \mathrm{e}^x
\end{aligned}$$

Acá usamos l'Hôpital para calcular el límite interno. □

Luego un par de cotas extremadamente útiles.

Lema 32.2. *Para todo $x \geq 0$:*

$$1 + x \leq \left(1 + \frac{x}{n}\right)^n \leq \mathrm{e}^x$$

Demostración. Para la primera desigualdad, del teorema del binomio:

$$\begin{aligned}\left(1 + \frac{x}{n}\right)^n &= \sum_k \binom{n}{k} \left(\frac{x}{n}\right)^k \\ &= 1 + x + \sum_{k \geq 2} \binom{n}{k} \left(\frac{x}{n}\right)^k \\ &\geq 1 + x\end{aligned}$$

Para la segunda desigualdad escribimos por el teorema del binomio:

$$\begin{aligned}\left(1 + \frac{x}{n}\right)^n &= \sum_{k \geq 0} \binom{n}{k} \left(\frac{x}{n}\right)^k \\ &= \sum_{k \geq 0} \frac{n^{\underline{k}}}{n^k} \frac{x^k}{k!}\end{aligned}$$

Comparando con la serie para e^x, ambas son series de términos positivos ya que x no es negativo, como $n^{\underline{k}}/n^k \leq 1$ los términos de la segunda acotan a los de la primera por arriba. □

Otro resultado que requeriremos es la cota siguiente:

Lema 32.3. *Para $n > t > 0$ se cumple:*

$$\binom{n}{t} \leq \frac{n^n}{t^t (n-t)^{n-t}}$$

Demostración. Primeramente, sabemos que:

$$\binom{n}{t} = \frac{n!}{t!(n-t)!}$$

Usando la fórmula de Stirling para factoriales:

$$n! = \sqrt{2\pi n} \left(\frac{n}{\mathrm{e}}\right)^n \mathrm{e}^{r(n)} \tag{32.1}$$

donde usaremos las cotas de Robbins [13] para $r(n)$:

$$\frac{1}{12n+1} < r(n) < \frac{1}{12n} \tag{32.2}$$

Con la fórmula de Stirling obtenemos:

$$\begin{aligned}\binom{n}{t} &= \frac{(n/\mathrm{e})^n \sqrt{2\pi n} \mathrm{e}^{r(n)}}{(t/\mathrm{e})^t \sqrt{2\pi t} \mathrm{e}^{r(t)} ((n-t)/\mathrm{e})^{n-t} \sqrt{2\pi(n-t)} \mathrm{e}^{r(n-t)}} \\ &= \frac{n^n}{t^t (n-t)^{n-t}} \cdot \frac{\sqrt{n} \mathrm{e}^{r(n)-r(t)-r(n-t)}}{\sqrt{2\pi}\sqrt{t(n-t)}}\end{aligned}$$

Nos interesa demostrar que el segundo factor es menor a 1. De partida, vemos que el mínimo de $t(n-1)$ en el rango de interés $1 \le t \le n-1$ se da para $t=1$ o $t=n-1$, aporta un factor:

$$\begin{aligned}\frac{\sqrt{n}}{\sqrt{t(n-t)}} &\le \frac{\sqrt{n}}{\sqrt{n-1}} \\ &\le \sqrt{2}\end{aligned}$$

Con las cotas de Robbins para $r(n)$ podemos escribir:

$$\frac{1}{12n+1} - \frac{1}{12(n-t)} - \frac{1}{12t} \le r(n) - r(t) - r(n-t) \le \frac{1}{12n} - \frac{1}{12t+1} - \frac{1}{12(n-t)+1}$$

El mínimo de la cota izquierda se da para $n=2$, $t=1$, donde vale $-19/150$; la cota derecha es menor a 0 (cada uno de los términos que se restan a $1/12n$ es mayor a este). Juntando todas las piezas, vemos que:

$$\begin{aligned}\binom{n}{t} &\le \frac{n^n}{t^t(n-t)^{n-t}} \cdot \frac{\sqrt{2}\mathrm{e}^0}{\sqrt{2\pi}} \\ &= \frac{1}{\sqrt{\pi}} \cdot \frac{n^n}{t^t(n-t)^{n-t}} \\ &< \frac{n^n}{t^t(n-t)^{n-t}}\end{aligned}$$

dado que el primer factor definitivamente es menor a 1. □

32.2. La escena de hashing

Seamos más específicos. Nos interesa almacenar n objetos, que pertenecen a un universo $\mathscr{U}$, La tabla *hash* es un arreglo $T[0..m-1]$, donde m es el tamaño de la tabla. En estos términos:

$$h\colon \mathscr{U} \to [0, m-1]$$

Obviamente, si $|\mathscr{U}| = m$, podemos usar simplemente x como índice de la tabla. El caso de interés nuestro es que n (el número de objetos a almacenar) es muchísimo menor que el total de objetos posibles; buscamos que m no sea mucho mayor que n para ahorrar espacio. Pero en caso que $m < |\mathscr{U}|$ necesariamente debemos manejar *colisiones*, situaciones en que queremos almacenar dos objetos $x \neq y$ tales que $h(x) = h(y)$. Para esto hay tres alternativas de solución:

Direccionamiento cerrado: Los objetos que colisionan se almacenan en una estructura secundaria, como una lista o un árbol binario de búsqueda.

Direccionamiento abierto: Si se produce una colisión, almacenamos uno de los objetos en alguna otra ubicación libre de la tabla.

Hashing perfecto: Si conocemos los objetos a almacenar de antemano, podemos elegir h de forma que no hayan colisiones. Pero las funciones de *hashing* perfectas (esencialmente permutaciones) son relativamente raras. Para n elementos hay $n!$ permutaciones y n^n funciones en total, o sea, usando la fórmula de Stirling la proporción es aproximadamente:

$$\frac{n!}{n^n} approx \frac{\sqrt{2\pi n}(n/\mathrm{e})^n}{n^n} \tag{32.3}$$

$$= \sqrt{2\pi n}\mathrm{e}^{-n} \tag{32.4}$$

Fredman, Komlós y Szemerédi [9] describen una técnica eficiente para construir funciones de *hashing* perfectas.

32.3. Importancia del azar

Por el principio del palomar, sea cual sea la función de *hashing* elegida para una tabla de tamaño m hay al menos $\lceil |\mathcal{U}|/m \rceil$ objetos que dan a la misma posición. Si en una aplicación aparecen muchos objetos que caen en la misma posición, ya sea por mala suerte o por elección maliciosa, el rendimiento será horrible. Este es un riesgo de seguridad importante cuando se procesan datos controlados por un potencial adversario (los llamados ataques de complejidad algorítmica, ver por ejemplo Crosby y Wallach [6]). La única solución práctica es elegir la función al azar entre una colección suficientemente grande. Específicamente, fijamos una colección de funciones $\mathcal{H}$ de $\mathcal{U}$ a $[0, m-1]$, y elegimos $h \in \mathcal{H}$ al azar según alguna distribución. Distintos conjuntos de funciones y distribuciones dan garantías teóricas diferentes.

El análisis teórico es más simple si se suponen funciones *hash ideales al azar*, se elige h uniformemente al azar entre todas las funciones de $\mathcal{U}$ a $[0, m-1]$. Es un modelo simple y limpio, que da las máximas garantías posibles, pero requiere elegir el valor de $h(x)$ al azar para cada x, lo que significaría tener que registrar su valor... y volvemos a nuestro problema inicial. En la práctica, nos restringimos a familias de funciones «lo suficientemente al azar» para dar buen rendimiento.

Una propiedad que intuitivamente parece útil es *uniformidad*. Se dice que la familia de funciones $\mathcal{H}$ es uniforme si eligiendo una función h uniformemente al azar de $\mathcal{H}$ cada valor es igualmente probable para cada objeto del universo:

$$\Pr_{h \in \mathcal{H}}[h(x) = i] = \frac{1}{m} \quad \text{para todo } x \in \mathcal{U} \text{ y todo } i \in [0, m-1] \tag{32.5}$$

Sin embargo, esto no es particularmente relevante. Considere la familia $\mathcal{K}$ de funciones constantes definidas mediante $\text{const}_a(x) = a$ para todo x. La familia $\mathcal{K}$ es perfectamente uniforme y totalmente inútil.

Lo que buscamos realmente es minimizar colisiones. Se dice que la familia $\mathcal{H}$ es *universal* si la probabilidad de colisión entre dos objetos diferentes es la menor posible:

$$\Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq \frac{1}{m} \quad \text{para todo } x \neq y \tag{32.6}$$

La mayor parte del los análisis elementales requieren una versión menos exigente, se dice que la familia $\mathcal{H}$ es *cercana a universal* si la probabilidad de colisión es cercana a la ideal:

$$\Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq \frac{2}{m} \quad \text{para todo } x \neq y \tag{32.7}$$

No hay nada especial en la constante 2, toda constante explícita mayor a 1 sirve.

Análisis más detallado requiere considerar colisiones en grupos mayores de objetos. Para cada entero k se dice que la familia $\mathcal{H}$ es *fuertemente k-universal* o *k-uniforme* si para toda colección de k objetos y de k índices:

$$\Pr_{h \in \mathcal{H}}\left[\bigwedge_{1 \leq j \leq k} h(x_j) = i_j\right] = \frac{1}{m^k} \quad \text{para todos } x_1, \ldots x_k \text{ distintos y todos } i_1, \ldots i_k \tag{32.8}$$

Funciones *hash* ideales al azar son k-uniformes para todo k.

32.4. Una familia de funciones hash universal

Hay varias construcciones de familias *hash* universales, presentamos una de ellas. Una discusión más detallada y ejemplos adicionales se encuentra por ejemplo en el apunte de Erickson [7].

Teorema 32.4. *Considere un primo p. Para cualquier par de enteros (a,b) con $1 \leq a < p$ y $0 \leq b < p$ defina la función $h_{a,b}: \mathscr{U} \to [0, m-1]$ mediante:*

$$h_{a,b}(x)((ax+b) \bmod p) \bmod m \tag{32.9}$$

Esta familia es universal.

Demostración. Fije enteros r, s, x, y con $x \not\equiv y \pmod{p}$ y $r \not\equiv s \pmod{p}$. El sistema lineal:

$$\begin{aligned} ax + b &\equiv r \pmod{p} \\ ay + b &\equiv s \pmod{p} \end{aligned}$$

tiene solución única (a,b) módulo p si $x \not\equiv y \pmod{p}$. Se sigue que:

$$\Pr_{a,b}[(ax+b) \bmod p = r \wedge (ay+b) \bmod p = s] = \frac{1}{p(p-1)}$$

por lo que:

$$\Pr_{a,b}[h_{a,b}(x) = h_{a,b}(y)] = \frac{N}{p(p-1)}$$

donde N es el número de pares ordenados $(r,s) \in \mathbb{Z}_p^2$ tales que $r \neq s$ pero $r \equiv s \pmod{m}$. Para cada $r \in \mathbb{Z}_p$ fijo, hay a lo más $\lfloor p/m \rfloor$ enteros $s \in \mathbb{Z}_p$ tales que $r \neq s$ pero $r \bmod m = s \bmod m$. Como p es primo, $\lfloor p/m \rfloor \leq (p-1)/m$, con lo que $N \leq p(p-1)/m$. Concluimos

$$\Pr_{a,b}[h_{a,b}(x) = h_{a,b}(y)] \leq \frac{1}{m}$$

como queríamos demostrar. □

Para aplicar esto en la práctica, al momento de crear la tabla se eligen a y b, uniformemente al azar y se usan durante su existencia.

32.5. Direccionamiento cerrado

Esta situación es la más sencilla de analizar matemáticamente, usando herramientas simples de probabilidades. Nuestro desarrollo sigue al de Bogart, Stein y Drysdale [2]. Suponemos *hashing* ideal.

32.5.1. Posiciones libres y ocupadas

Sea X_i el número de objetos en la posición i de la tabla. Es claro que:

$$\sum_i X_i = n$$

Por linealidad sabemos entonces:

$$\sum_i \mathbb{E}[X_i] = n$$

Por simetría, $\mathbb{E}[X_i]$ no depende de i:

$$\mathbb{E}[X_i] = \frac{n}{m}$$

Hemos demostrado:

Teorema 32.5. *Con* hashing *ideal, al almacenar n objetos a una tabla de tamaño m, el número esperado de objetos en cada posición es n/m.*

Después de agregar un objeto a la tabla, la probabilidad que la posición i esté vacía es $1-1/m$. Después de n objetos agregados a la tabla, la probabilidad que siga vacío es $(1-1/m)^n$ (son n intentos independientes). Consideremos la misma secuencia de objetos, y sea $X_i = 1$ si la posición i está libre, $X_i = 0$ en caso contrario. El número de posiciones libres es:

$$\sum_i X_i$$

Nuevamente por linealidad, como $\mathbb{E}[X_i] = (1-1/m)^n$, tenemos:

$$\begin{aligned}\mathbb{E}\left[\sum_i X_i\right] &= \sum_i \mathbb{E}[X_i] \\ &= \sum_i \left(1-\frac{1}{m}\right)^n \\ &= m\left(1-\frac{1}{m}\right)^n\end{aligned}$$

Hemos demostrado:

Teorema 32.6. *Al hashear n objetos a una tabla de m ubicaciones, el número esperado de ubicaciones vacías es $m(1-1/m)^n$.*

Incidentalmente, si $n = m$, la fracción esperada de espacios libres es $(1-1/n)^n$. Cuando $n \to \infty$, por el límite clásico del lema 32.1 cuando $n = m$ y $n \to \infty$, la fracción de espacios libres es e^{-1}, esperamos n/e espacios libres.

32.5.2. Problema del coleccionista de cupones

Es de interés calcular el número de objetos requeridos para llenar la tabla de tamaño m. Esto se conoce como el *problema del coleccionista de cupones* (*coupon collector problem*, alguien recibe cupones elegidos al azar entre m y busca armar la colección completa). Planteamos el problema como ir llenando sucesivamente k posiciones, estamos interesados en ir de k llenas a $k+1$. Como los cupones llegan al azar, si hay k posiciones ocupadas la probabilidad de tener éxito (llenar una libre) en cada intento será:

$$p_k = \frac{m-k}{m}$$

por lo que el número esperado de pasos requeridos en esta etapa es:

$$\frac{1}{p_k} = \frac{m}{m-k}$$

Por la linealidad del valor esperado el valor esperado del número de pasos total T es:

$$\begin{aligned}\mathbb{E}[T] &= \sum_{0\le k\le m-1} \frac{1}{p_k} \\ &= m \sum_{0\le k\le m-1} \frac{1}{m-k} \\ &= m \sum_{1\le k\le m} \frac{1}{k} \\ &= m H_m \\ &\sim m \ln m\end{aligned}$$

Acá hemos usado la definición de números harmónicos:

$$H_n = \sum_{1\le k\le n} \frac{1}{k}$$

Se sabe (ver por ejemplo el apunte de Fundamentos de Informática [3, capítulo 18]) que:

$$H_n = \ln n + \gamma + O\left(\frac{1}{n}\right)$$

donde $\gamma \approx 0{,}57721$ es la constante de Euler.

Para la varianza, como el número de cupones recibidos en cada paso son variables aleatorias independientes:

$$\begin{aligned}\operatorname{var}[T] &= \operatorname{var}[t_1] + \operatorname{var}[t_2] + \cdots + \operatorname{var}[t_m] \\ &= \frac{1-p_1}{p_1^2} + \frac{1-p_2}{p_2^2} + \cdots + \frac{1-p_m}{p_m^2} \\ &\le \frac{m^2}{m^2} + \frac{m^2}{(m-1)^2} + \cdots + \frac{m^2}{1^2} \\ &= m^2\left(\frac{1}{1^2} + \frac{1}{2^2} + \cdots + \frac{1}{m^2}\right) \\ &< \frac{\pi^2 m^2}{6}\end{aligned}$$

Esto último por la solución al famoso problema de Basilea (ver por ejemplo el apunte de Fundamentos de Informática [3, capítulo 18]):

$$\sum_{k\ge 1} \frac{1}{k^2} = \frac{\pi^2}{6}$$

32.5.3. Número esperado de colisiones

El número de colisiones es el número total n de objetos menos el número de ubicaciones ocupadas, ya que cada posición ocupada contiene un objeto que no ha colisionado (el primero en llegar). Por los teoremas 32.5 y 32.6 tenemos:

$$\begin{aligned}\mathbb{E}[\text{colisiones}] &= n - \mathbb{E}[\text{posiciones ocupadas}] \\ &= n - m + \mathbb{E}[\text{posiciones libres}]\end{aligned}$$

y tenemos otro teorema:

Teorema 32.7. *Usando* hashing *ideal, al agregar n objetos a una tabla de tamaño m, el número esperado de colisiones es:*

$$n - m + m\left(1 - \frac{1}{m}\right)^n$$

Esto va contra la intuición: en promedio, al agregar n objetos a una tabla de tamaño n cuando n es grande, esperamos un objeto en cada posición, pero una fracción de $\mathrm{e}^{-1} = 0{,}3679$ queda libre, y hay $n\mathrm{e}^{-1}$ colisiones.

32.5.4. Largo esperado de la lista más larga

Suponiendo que los elementos que dan en una posición dada de la tabla se almacenan en una lista, tiene sentido preguntarse por el peor caso de esta. Es claro que lo peor que puede ocurrir es que todos den en la misma posición, con lo que el peor largo es directamente n. Pero esto es extremadamente poco probable, no es una medida realista. Usar listas es una buena opción, en promedio para tablas no demasiado llenas las listas serán cortas e importa que sean simples de administrar. Otras posibilidades son usar a su vez tablas *hash* con direccionamiento cerrado, que se hacen crecer cuando se llenen demasiado.

Nos interesa acotar el valor esperado del tamaño de la lista más larga, cuando se ingresan n objetos a una tabla de tamaño n. Usaremos la fórmula de Stirling (32.1):

$$x! = \left(\frac{x}{\mathrm{e}}\right)^x \sqrt{2\pi x}\mathrm{e}^{r(x)}$$

donde las cotas de Robbins (32.2) son:

$$\frac{1}{12x+1} < r(x) < \frac{1}{12x}$$

Nos dice que, en términos gruesos, $(x/\mathrm{e})^x$ es una buena aproximación para $x!$. Por lo demás, el factor de error es casi uno, podemos omitirlo.

Partiremos por una cantidad que sabemos cómo calcular: sea H_{it} el evento que t claves dan en la posición i. Entonces $\Pr[H_{it}]$ es la probabilidad de exactamente t éxitos en n intentos independientes con probabilidad de éxito $1/n$, o sea:

$$\Pr[H_{it}] = \binom{n}{t}\left(\frac{1}{n}\right)^t\left(1 - \frac{1}{n}\right)^{n-t} \tag{32.10}$$

Sea M_t el evento que la lista más larga sea de largo t. Tenemos:

Lema 32.8. *Sea M_t el evento que la cola más larga al hashear n objetos en una tabla de tamaño n sea de largo t, y sea H_{it} el evento que exactamente t claves dan en la posición i. Entonces:*

$$\Pr[M_t] \leq n\Pr[H_{1t}]$$

Demostración. Sea M_{it} el evento que el largo máximo sea t y que está en la posición i. Es claro que $M_{it} \subseteq H_{it}$, por lo que $\Pr[M_{it}] \leq \Pr[H_{it}]$. Además:

$$M_t = M_{1t} \cup M_{2t} \cup \cdots \cup M_{nt}$$

Por simetría, $\Pr[H_{it}]$ son todas iguales, y por la cota de la unión resulta lo prometido. □

Podemos usar (32.10) y el lema 32.3 para acotar:

$$\begin{aligned}\Pr[H_{it}] &= \binom{n}{t}\left(\frac{1}{n}\right)^t\left(1-\frac{1}{n}\right)^{n-t}\\ &\leq \frac{n^n}{t^t(n-t)^{n-t}}\left(\frac{1}{n}\right)^t\left(1-\frac{1}{n}\right)^{n-t}\end{aligned}$$

Simplificando, como $(1-1/n)^{n-t}<1$:

$$\begin{aligned}\Pr[H_{it}] &\leq \frac{n^n}{t^t(n-t)^{n-t}}\left(\frac{1}{n}\right)^t\\ &= \left(\frac{n}{n-t}\right)^{n-t}\frac{1}{t^t}\\ &= \left(\left(1+\frac{t}{n-t}\right)^{\frac{n-t}{t}}\right)^t\frac{1}{t^t}\\ &\leq \frac{\mathrm{e}^t}{t^t}\end{aligned}$$

Juntando los anteriores resultados:

Lema 32.9.

$$\Pr[M_t] \leq \frac{n\mathrm{e}^t}{t^t}$$

Tenemos una cota, que podemos usar para acotar la cantidad de interés, el largo promedio de la lista más larga:

$$\sum_{0\leq t\leq n} t\Pr[M_t] \tag{32.11}$$

Pero hay un problema: la cota del lema 32.9 da valores absurdos para t pequeño. Por ejemplo, para $t=1$ da $\Pr[M_1]\leq n\mathrm{e}$, lo que está totalmente fuera de proporción. Para valores grandes de t, el lema indica que $\Pr[M_t]$ es pequeño, y la cota resulta ajustada. El truco para estimar la suma (32.11) es dividir en una suma sobre valores «chicos» y «grandes», y estimar las sumas por separado. Suponiendo que $n\geq 3$ (para evitar logaritmos de números negativos) cortamos en $t^* = \lfloor 5\ln n/\ln\ln n\rfloor$:

$$\sum_{0\leq t\leq n} t\Pr[M_t] = \sum_{0\leq t\leq t^*} t\Pr[M_t] + \sum_{t^*<t\leq n} t\Pr[M_t] \tag{32.12}$$

Para la primera suma, observamos que $t\leq t^*$, de manera que:

$$\begin{aligned}\sum_{0\leq t\leq t^*} t\Pr[M_t] &\leq \sum_{0\leq t\leq t^*} t^*\Pr[M_t]\\ &= t^*\sum_{0\leq t\leq t^*}\Pr[M_t]\\ &\leq t^*\end{aligned} \tag{32.13}$$

Lo último resulta simplemente porque las probabilidades de eventos disjuntos suman a lo más a uno.

Para la segunda usamos nuestro lema 32.9. Dijimos que esperamos que $\Pr[M_t]$ acá sea pequeño, y la cota disminuye. Por el lema, para $t\geq t^*$ después de algún álgebra tediosa vemos que:

$$\Pr[M_t] \leq \frac{1}{n^2}$$

Podemos entonces acotar la suma para t mayores:

$$\begin{aligned}\sum_{t^*<t\le n} t\Pr[M_t] &\le \sum_{t^*<t\le n} n\frac{1}{n^2}\\ &= \sum_{t^*<t\le n}\frac{1}{n}\\ &\le 1\end{aligned} \tag{32.14}$$

Combinando (32.13) con (32.14) obtenemos:

$$\sum_{0\le t\le n} t\Pr[M_t] \le 5\ln n/\ln\ln n + 1 = O(\ln n/\ln\ln n) \tag{32.15}$$

La elección de t^* para cortar la suma parece magia. Nos pusimos una cota para $\Pr[M_t]$ cercana a $1/n^2$, que da la cómoda cota (32.14). Esto da la ecuación:

$$\frac{ne^t}{t^t} = \frac{1}{n^2} \tag{32.16}$$

Esto no puede resolverse para t, pero es más o menos similar a (tome logaritmos y omita constantes molestas):

$$\begin{aligned}t^t &= n\\ t &= \frac{\ln n}{\ln t}\end{aligned} \tag{32.17}$$

Tomando una aproximación inicial $t = \ln n$, iterar una vez nos da:

$$t \approx \frac{\ln n}{\ln\ln n} \tag{32.18}$$

Sospechamos que la solución de nuestra ecuación (32.16) es algo como:

$$t = c\frac{\ln n}{\ln\ln n} \tag{32.19}$$

y un poco de experimentación muestra que $c = 5$ da una división manejable en (32.12).

32.5.5. Usar más de una función de hashing

Una idea sorprendentemente efectiva es usar varias funciones de *hashing* independientes y elegir aquella posición en la cual la lista es más corta. Mitzenmacher [11] demostró que al usar dos el largo de la lista más larga es $\Theta(\log\log n)$ con alta probabilidad. Esto es una reducción exponencial frente a la cota (32.15). El desarrollo es complejo, no lo repetiremos acá. Una revisión reciente de este resultado y afines, junto a una discusión de aplicaciones, dan Mitzenmacher, Richa y Sitaraman [12]. Es de interés práctico porque disminuye el tiempo de búsqueda, además que las búsquedas en ambas listas pueden paralelizarse fácilmente.

32.5.6. Análisis de direccionamiento cerrado

Consideremos los programas para direccionamiento cerrado, listado 32.1. Usamos listas sin ordenar para cada casillero, por ser lo más simple. Usaremos como medida de costo los nodos considerados en cada caso. Como antes, sea m el tamaño de la tabla y n el número de elementos que contiene, y suponemos *hashing* ideal. Abreviaremos:

$$\alpha = \frac{n}{m} \tag{32.20}$$

```
#include <stdlib.h>

#include "item.h"
#include "hash.h"

#define M 10267

struct node{
        struct item item;
        struct node *next;
};

static struct node *table[M];

struct item *search(struct item *pi)
{
        unsigned long h = hash(pi);

        struct node *p;

        for(p = table[h]; p; p = p->next)
                if(cmp(pi, &p->item) == 0)
                        return &p->item;
        return NULL;
}

void insert(struct item *pi)
{
        unsigned long h = hash(pi);

        struct node *p = malloc(sizeof(struct node));

        p->item = *pi;
        p->next = table[h];
        table[h] = p;
}
```

Listado 32.1 – Hashing cerrado

32.5.6.1. Inserción

Es claro (comparar con el listado 32.1) que el número de posiciones consideradas es constante ($O(1)$), no verificamos que el elemento a insertar no esté ya presente.

32.5.6.2. Búsqueda exitosa

Sea la secuencia en que se ingresaron los objetos $\langle x_i \rangle$, que suponemos todos diferentes entre sí. Buscamos hallar el costo promedio de búsqueda de los objetos, suponiendo que cada uno es bus-

cado con la misma frecuencia. Como insertamos nuevos objetos a cada lista al comienzo, debemos comparar y pasar sobre los insertados después.

Sean X_{ij} variables indicadoras, $X_{ij} = [h(x_i) = h(x_j)]$. Por la suposición sobre la función h para todo $i \neq j$ tenemos:

$$\mathbb{E}[X_{ij}] = \frac{1}{m}$$

El valor esperado del número de posiciones revisadas es:

$$\begin{aligned}
\mathbb{E}\left[\frac{1}{n}\sum_{1\leq n\leq n}\left(1+\sum_{i+1\leq j\leq n} X_{ij}\right)\right] &= \frac{1}{n}\sum_{1\leq i\leq n}\left(1+\sum_{i+1\leq j\leq n}\mathbb{E}[X_{ij}]\right) \\
&= \frac{1}{n}\sum_{1\leq i\leq n}\left(1+\frac{1}{m}(n-i)\right) \\
&= 1+\frac{1}{mn}\left(\sum_{1\leq i\leq n} n - \sum_{1\leq i\leq n} i\right) \\
&= 1+\frac{n}{2m}-\frac{1}{2m} \qquad (32.21) \\
&= 1+\frac{\alpha}{2}-\frac{\alpha}{2n} \\
&\sim 1+\frac{\alpha}{2} \qquad (32.22)
\end{aligned}$$

32.5.6.3. Búsqueda fallida

La situación es la misma anterior, solo que en este caso debemos revisar la lista completa:

$$\begin{aligned}
\mathbb{E}\left[\frac{1}{n}\sum_{1\leq n\leq n}\left(1+\sum_{1\leq j\leq n} X_{ij}\right)\right] &= \frac{1}{n}\sum_{1\leq i\leq n}\left(1+\sum_{1\leq j\leq n}\mathbb{E}[X_{ij}]\right) \\
&= \frac{1}{n}\sum_{1\leq i\leq n}\left(1+\frac{1}{m}n\right) \\
&= 1+\frac{n}{m} \\
&= 1+\alpha \qquad (32.23)
\end{aligned}$$

32.5.7. Variantes de interés

Algunas variantes que vale la pena considerar es manejar las listas ordenadas (esto encarece la inserción de nuevos elementos, pero disminuye el costo de búsquedas fallidas), o utilizar estructuras más sofisticadas en cada casillero, como árboles binarios o derechamente nuevas tablas *hash*. Claro que por lo discutido antes, estas generalmente solo tienen sentido en caso que n sea substancialmente mayor a m.

32.6. Direccionamiento abierto

Una alternativa es usar posiciones libres de la tabla cuando ocurren colisiones. El análisis de Knuth [10] de esta técnica es considerado por muchos como el nacimiento del análisis de algoritmos. Una discusión detallada de estas técnicas ofrecen Sedgewick y Flajolet [14, capítulo 8]. Acá veremos un estudio mucho más somero, adaptado de Erickson [7].

Algoritmo 32.1: Direccionamiento abierto

```
for i ← 0 to m − 1 do
    if T[h_i(x)] = x then
        return h_i(x)
    else if T[h_i(x)] = ∅ then
        return Absent
    end
end
```

El algoritmo es 32.1, supone una secuencia de funciones *hash* $\langle h_0, h_1, \ldots, h_{m-1}\rangle$ tales que para todo $x \in \mathscr{U}$ tenemos que $\langle h_0(x), h_1(x), \ldots, h_{m-1}(x)\rangle$ es una permutación de $\{0,1,2,\ldots,m-1\}$. En otras palabras, para $1 \le i \le m-1$ las funciones h_i mapean x a posiciones diferentes de la tabla.

Simplifica el análisis que la secuencia de intentos sea verdaderamente al azar, usamos la *suposición de hash fuerte uniforme*: para cada objeto $x \in \mathscr{U}$ la secuencia $\langle h_0(x), h_1(x), \ldots, h_{m-1}(x)\rangle$ es una permutación elegida uniformemente al azar entre las permutaciones del conjunto $\{0,1,2,\ldots,m-1\}$.

Calculemos el tiempo esperado de una búsqueda infructuosa bajo esta suposición. La suposición tiene dos consecuencias importantes:

Uniformidad: Para cada objeto x y posición i, el valor de $h_i(x)$ toma los valores $\langle 0,1,2,\ldots,m-1\rangle$ con igual probabilidad.

Independencia: Para cada objeto x, ignorando el intento $h_0(x)$, los intentos $\langle h_1(x), h_2(x), \ldots, h_{m-1}(x)\rangle$ tienen la misma probabilidad de ser cualquier permutación de $\{0,1,2,\ldots,m-1\} \setminus \{h_0(x)\}$.

Por uniformidad, la probabilidad de que $T(h_0(x))$ esté libre es exactamente m/n. Por independencia, podemos suponer que si nuestro algoritmo halla ocupada la posición $h_0(x)$, *busca recursivamente en el resto de la tabla*. Llamemos $p(m,n)$ al número promedio de intentos al buscar en una tabla de tamaño m que contiene n elementos. Lo anterior sugiere la recurrencia:

$$p(m,n) = 1 + \frac{n}{m}\,p(m-1,n-1)$$

La condición de borde es $p(m,0) = 1$, si no hay nada en la tabla tendremos éxito con el primer intento.

Podemos demostrar por inducción que $p(m,n) \le m/(m-n)$:

$$\begin{aligned} p(m,n) &= 1 + \frac{n}{m}\,p(m-1,n-1) \\ &\le 1 + \frac{n}{m}\cdot\frac{m-1}{(m-1)-(n-1)} \\ &< 1 + \frac{n}{m}\cdot\frac{m}{m-n} \\ &= \frac{m}{m-n} \end{aligned}$$

En términos del *factor de carga* $\alpha = n/m$, tenemos:

$$p(m,n) \le \frac{1}{1-\alpha}$$

O sea, el tiempo esperado de búsqueda es $O(1)$, pero crece rápidamente cuando α se acerca a 1.

El supuesto de secuencias de intentos independientes es irreal, en la práctica debemos usar secuencias más manejables. Tal vez la más sencilla es *prueba lineal*, usar una única función h e intentar las posiciones $(h(x)+i)$ mód m. Además de ser simple, tiene la ventaja que intenta posiciones consecutivas, lo que funciona bien en arquitecturas de memoria con caché. Por otro lado, esto tiene el efecto de apiñar las posiciones ocupadas (una secuencia de k posiciones ocupadas tiene una probabilidad de k/m de alargarse porque el nuevo elemento cae en alguna de sus posiciones, lo que en realidad es aún peor ya que puede llegar a llenar posiciones entre secuencias), alargando las búsquedas al aumentar α. Propuestas para evitar este efecto son *doble hashing*, que calcula una nueva función de hashing independiente $h_1(x)$ y revisa las posiciones $(h(x)+i\,h_1(x))$ mód m; o usar *prueba cuadrática*, que intenta $(h(x)+c_1 i+c_2 i^2)$ mód m para c_1, c_2 elegidos adecuadamente. Otra opción es *prueba binaria*, que usa $h_i(x) = h(x) \oplus i$, donde $\oplus$ es la operación de o exclusivo bit a bit. Tiene la ventaja de revisar un rango de entradas contiguas antes de pasar a otra (o sea, recorre líneas de caché completas), pero no las recorre secuencialmente (evitando apiñamiento). Detalles da Erickson [7].

32.6.1. Análisis de direccionamiento abierto

Como antes, sea m el tamaño de la tabla y n el número de elementos que contiene. Nuevamente la medida de costo es el número de posiciones revisadas, y promediamos sobre todos los elementos con probabilidad uniforme. Debemos hallar un espacio vacío para el elemento a insertar en caso que $h(x)$ ya esté ocupado. La suposición más simple es que cada intento sucesivo considera las posiciones aún no consideradas en forma uniforme, esencialmente *hashing* ideal.

32.6.1.1. Búsqueda fallida

Sea X el número de intentos al buscar x, y sea A_i el evento que hay i intentos y la posición intentada en el intento i está ocupada. Entonces:

$$\begin{aligned}\Pr[X \geq i] &= \Pr\left[\bigcap_{1\leq j\leq i-1} A_j\right] \\ &= \Pr[A_1]\cdot\Pr[A_2 \mid A_1]\cdot\Pr[A_3 \mid A_1\cap A_2]\cdot\cdots\cdot\Pr[A_{i-1} \mid A_1\cap A_2\cap\cdots\cap A_{i-2}]\end{aligned}$$

En el intento i ya descartamos $i-1$ posiciones a revisar, ocupadas por los $i-1$ elementos que están en ellas, quedan $m-i+1$ posiciones de las cuales hay $n-i+1$ ocupadas, y estamos eligiendo uniformemente al azar entre ellas:

$$\Pr[A_i] = \frac{n-i+1}{m-i+1}$$

Como son intentos independientes:

$$\begin{aligned}\Pr[X \geq i] &= \prod_{0\leq j\leq i-2} \frac{n-j}{m-j} \\ &\leq \left(\frac{n}{m}\right)^{i-1} \\ &= \alpha^{i-1}\end{aligned}$$

Por lo tanto:

$$\begin{aligned}\mathbb{E}[X] &= \sum_{i\geq 1} \Pr[X \geq i] \\ &\leq \sum_{i\geq 1} \alpha^{i-1} \\ &= \sum_{i\geq 0} \alpha^{i} \\ &= \frac{1}{1-\alpha}\end{aligned} \tag{32.24}$$

Insertar un nuevo elemento en la tabla es una búsqueda fallida para hallar su lugar.

32.6.1.2. Búsqueda exitosa

Las posiciones revisadas al insertar x_{i+1} son exactamente las mismas que las búsquedas fallidas que llevaron a insertar ese elemento, cuando la tabla tenía i elementos, o sea, se revisaron a lo más $1/(1-i/m) = m/(m-i)$ posiciones. El promedio cumple:

$$\begin{aligned}\frac{1}{n}\sum_{0\leq i\leq n-1}\frac{m}{m-i} &= \frac{m}{n}\sum_{0\leq i\leq n-1}\frac{1}{m-i} \\ &= \frac{1}{\alpha}(H_m - H_{m-n}) \\ &\leq \frac{1}{\alpha}\int_{m-n}^{m}\frac{\mathrm{d}x}{x} \\ &= \frac{1}{\alpha}\ln\frac{m}{m-n} \\ &= \frac{1}{\alpha}\ln\frac{1}{1-\alpha}\end{aligned} \tag{32.25}$$

32.6.2. Resumen del análisis

El cuadro 32.1 resume los resultados anteriores. Estos análisis se basa en el desarrollo de CLRS [5]. Sedgewick y Flajolet [14] analizan estas mismas situaciones usando el método simbólico, obteniendo resultados más ajustados y también varianzas.

Técnica	Exitosa	Fallida
Direccionamiento cerrado	$1+\frac{\alpha}{2}$	$1+\alpha$
Direccionamiento abierto	$\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$	$\frac{1}{1-\alpha}$

Cuadro 32.1 – Resumen de posiciones revisadas en hashing

32.7. Otras aplicaciones

La misma idea tiene otras aplicaciones. Exploraremos un par de ellas.

32.7.1. Filtro de Bloom

Hay situaciones en las que interesa representar un conjunto grande en forma compacta. Si aceptamos la posibilidad de errores, en particular *falso positivo* (respondemos «sí», pero la respuesta correcta es «no»), una opción es un filtro de Bloom [1]. Una discusión detallada, particularmente aplicaciones a sistemas distribuidos, dan Broder y Mitzenmacher [4]. La idea es representar el conjunto mediante un arreglo de m bits, para agregar el elemento x aplicarle k funciones *hash* independientes h_i, poniendo en 1 los bits respectivos. Ver si x pertenece al conjunto es revisar los bits respectivos, si todos son 1 probablemente pertenece; si alguno es 0, definitivamente no pertenece.

Interesa derivar la probabilidad de error si hay n elementos del conjunto representado con n bits y k funciones *hash*. Para ello primero derivaremos la probabilidad que luego de insertar n elemento, un bit dado siga en 0. Esto corresponde a nk intentos fallidos de apuntarle a ese bit, con probabilidad de éxito $1/m$ en cada intento, o sea:

$$\begin{aligned}\Pr[\text{bit } i \text{ en cero luego de agregar } n] &= \left(1-\frac{1}{m}\right)^{nk} \\ &= \left(\left(1-\frac{1}{m}\right)^{m}\right)^{nk/m} \\ &\approx \mathrm{e}^{-nk/m}\end{aligned}$$

Un falso positivo, por otro lado, corresponde a tener k éxitos (encontrar bit en 1) en k intentos independientes. La probabilidad de éxito en cada intento viene del punto anterior:

$$\begin{aligned}\Pr[\text{falso positivo}] &= \prod_{1\le i\le k} \Pr[\text{bit } h_i(x) \text{ es } 1] \\ &= \left(1-\left(1-\frac{1}{m}\right)^{nk}\right)^{k} \\ &\approx \left(1-\mathrm{e}^{-nk/m}\right)^{k}\end{aligned}$$

Los filtros de Bloom balancean varios efectos contrarios: queremos usar poca memoria (m pequeño, pero eso hace más probables las colisiones, falsos positivos), poco tiempo de cómputo (k pequeño, usa menos cómputo pero hace más probables las colisiones). Un análisis aproximado es como sigue. Sea $f(k)$ la función que nos interesa, queremos minimizar la probabilidad de falso positivo para m y n fijos, donde suponemos m lo suficientemente grande para poder aplicar la aproximación por la exponencial. Minimizar $f(k)$ es minimizar $\ln f(k)$:

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}k}\ln f(k) &= \frac{\mathrm{d}}{\mathrm{d}k}k\ln\left(1-\mathrm{e}^{-nk/m}\right) \\ &= \ln\left(1-\mathrm{e}^{-nk/m}\right)+\frac{nk}{m}\cdot\frac{\mathrm{e}^{-nk/m}}{1-\mathrm{e}^{-nk/m}}\end{aligned}$$

Igualando a cero la derivada, hallamos que el mínimo se da con $k=\frac{m}{n}\ln 2$, y este mínimo es global. En el mínimo, la probabilidad de falso positivo es:

$$\left(\frac{1}{2}\right)^{k} = 0{,}6185^{m/n}$$

Conforme m crece respecto de n, disminuye la tasa de falsos positivos.

Un problema de los filtros de Bloom es que podemos *agregar* elementos al conjunto, pero no *eliminarlos*. Una posibilidad es no usar un simple bit, sino un contador. Este es el caso que discuten Fan et al [8].

32.7.2. Contar elementos distintos

En muchas aplicaciones hay un gran flujo de elementos, e interesa estimar cuántos elementos distintos hay. Por ejemplo, en páginas web interesa saber el número de visitantes diferentes. Una posibilidad es registrarlos y procesar la lista. Acá discutiremos una posibilidad más simple (aunque menos precisa).

La idea básica es calcular una función *hash* de cada elemento e ir recordando el mínimo visto. Resulta que el mínimo tiene una relación simple con el número de elementos diferentes.

Teorema 32.10. *Si $X_1, X_2, \ldots, X_n$ son variables aleatorias independientes idénticamente distribuidas con función de distribución cumulativa F, la función cumulativa del mínimo de los X_i es:*

$$F_{min}(y) = 1 - (1 - F(y))^n$$

Demostración. Nos interesa:

$$\begin{aligned}
F_{\min}(y) &= \Pr[\text{mín}\{X_1, \ldots, X_n\} \leq y] \\
&= 1 - \Pr[\text{mín}\{X_1, \ldots, X_n\} > y] \\
&= 1 - \Pr[X_1 > y \wedge \cdots \wedge X_n > y] \\
&= 1 - \Pr[X_1 > y] \cdot \cdots \cdot \Pr[X_n > y] \\
&= 1 - (\Pr[X_1 > y])^n \\
&= 1 - (1 - \Pr[X_1 \leq y])^n \\
&= 1 - (1 - F(y))^n
\end{aligned}$$

□

En particular, si $X_i \sim \mathbf{U}(0,1)$ (distribución uniforme continua) la función cumulativa es simplemente $F(y) = y$ para $0 \leq y \leq 1$ y resulta:

$$F_{\min}(y) = 1 - (1 - y)^n$$

De acá la función de densidad de probabilidad es la derivada:

$$f_{\min}(y) = n(1 - y)^{n-1}$$

Es rutina calcular:

$$\begin{aligned}
\mathbb{E}[\text{mín}\{X_1, \ldots, X_n\}] &= \int_0^1 y f(y)\, \mathrm{d}y \\
&= \frac{1}{n+1} \\
\text{var}[\text{mín}\{X_1, \ldots, X_n\}] &= \int_0^1 \left(y - \frac{1}{n+1}\right)^2 f(y)\, \mathrm{d}y \\
&= \frac{n}{(n+1)^2(n+2)}
\end{aligned}$$

O sea, registrando el mínimo de $h(x)$ (normalizado a $[0,1]$) para los elementos conforme llegan, obtenemos una estimación de n, el número de elementos distintos observados. Lamentablemente la desviación es bastante grande, pero podemos usar el promedio de varias funciones para mejorar la estimación.

Ejercicios

1. Las direcciones IPv4 se escriben tradicionalmente en la forma $a.b.c.d$, donde a, b, c, d están entre 0 y 255. Se propone la siguiente familia de funciones *hash*: se elige un primo p, y se eligen al azar $\alpha, \beta, \gamma, \delta \in \mathbb{Z}_p$, y:

   $$h(a,b,c,d) = (\alpha a + \beta b + \gamma c + \delta d) \text{ mód } p$$

   a) Discuta porqué algunos miembros de la familia son malos. En particular, vea los casos $\alpha = 1, \beta = \gamma = \delta = 0$ y $\alpha = \beta = \gamma = 0, \delta = 1$.

   b) ¿Es universal la familia completa?

   c) ¿Qué pasa con versiones restringidas, como $\alpha, \beta, \gamma, \delta \neq 0$? ¿Al menos un coeficiente no cero? ¿Exactamente un coeficiente no cero?

2. Considere una variante de filtros de Bloom, en la cual los m bits se dividen en k grupos de m/k, cada uno reservado a una de las funciones *hash*. Esto tiene la ventaja que pueden calcularse en paralelo, incluso con memorias separadas. Compare la tasa de falsos positivos con la derivada en el texto.

Bibliografía

[1] Burton H. Bloom: *Space/time trade-offs in hash coding with allowable errors*. Communications of the ACM, 13(7):422–426, July 1970.

[2] Kenneth P. Bogart, Clifford Stein, and Robert L. Drysdale: *Discrete Mathematics for Computer Science*. Addison-Wesley, 2010.

[3] Horst H. von Brand: *Fundamentos de Informática*. `git://csrg.inf.utfsm.cl/vonbrand/Ramos/trainwreck`, Septiembre 2017. Versión 0.84.

[4] Andrei Broder and Michael Mitzenmacher: *Network applications of Bloom filters: A survey*. Internet Mathematics, 1(4):485–509, 2004.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald R. Rivest, and Clifford Stein: *Introduction to Algorithms*. MIT Press, third edition, 2009.

[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] Jeff Erickson: *Algorithms, etc.* `http://jeffe.cs.illinois.edu/teaching/algorithms`, January 2015. Department of Computer Science, University of Illinois at Urbana-Champaign.

[8] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder: *Summary cache: A scalable wide-area web cache sharing protocol*. IEEE/ACM Transactions on Networking, 8(3):281–293, June 2000.

[9] 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.

[10] Donald E. Knuth: *Notes on "open" addressing*. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.4899`, July 1963.

[11] Michael David Mitzenmacher: *The Power of Two Choices in Randomized Load Balancing*. PhD thesis, Computer Science, University of California at Berkeley, Fall 1996.

[12] Michael David Mitzenmacher, Andréa W. Richa, and Ramesh Sitaraman: *The power of two random choices: A survey of techniques and results*. In Sanguthevar Rajasekaran, Panos M. Pardalos, John H. Reif, and José Rolim (editors): *Handbook of Randomized Computing*, volume I, pages 255–312. Kluwer Academic Publishers, 2001.

[13] Herbert Robbins: *A remark on Stirling's formula*. The American Mathematical Monthly, 62(1):26–29, January 1955.

[14] Robert Sedgewick and Philipe Flajolet: *An Introduction to the Analysis of Algorithms.* Addison-Wesley, second edition, 2013.

Clase 33

Algoritmos Aleatorizados

Un área activa de investigación reciente son los *algoritmos aleatorizados* [14, 21] (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. Un uso importante es algoritmos aleatorizados para aproximar soluciones a problemas NP-completos. Véase también el capítulo 34. Textos en el área son el clásico de Motwani y Raghavan [20] y el más accesible de Hromkovič [13]

33.1. 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 [4], en analogía a los métodos de Monte Carlo usados en análisis numérico, física estadística y simulación, aplicados ya en el proyecto Manhattan). Algunos hablan también de algoritmos de *Atlantic City*, otra ciudad famosa por sus casinos.

33.1.1. 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.

33.1.2. 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, este es una variable aleatoria (dependiente de los valores elegidos). Usaremos las siguientes definiciones para discutirlos.

Definición 33.1. Llamaremos $\mathrm{RT}(u)$ al tiempo de ejecución del algoritmo con entrada u.

Definición 33.2. 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 33.1 es una variable con distribución geométrica de probabilidad $1/2$, con lo que, medido en número de invocaciones de RandBit(), es $\mathbb{E}[\mathrm{RT}] = 2$. Sin embargo, en el peor caso nunca termina (si RandBit() siempre retorna 0).

Algoritmo 33.1: Perder el tiempo

```
while RandBit() = 1 do
    Nada
end
```

Otra definición relevante es:

Definición 33.3. 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 esta, mientras no haya consenso.

33.1.3. Algoritmos de Atlantic City

Este término fue introducido por Finn [10], para referirse a algoritmos que dan la respuesta correcta al menos 75 % de las veces (algunas otras versiones de la definición especifican alguna otra cota mayor a 50 %).

33.2. Ámbitos de aplicación

Algoritmos aleatorizados son usados en el ámbito de teoría de números (el método de Miller-Rabin [18, 24], 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* [23] y *treaps* [2, 26] (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.

33.3. 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 [7] y McIllroy y Douglas [17]).

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).

33.4. 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 (ver el apéndice G) 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:

$$\begin{aligned}\Pr[T \geq 600] &= \Pr\left[T \geq \frac{m}{10} \cdot \mathbb{E}[T]\right] \\ &\leq \mathrm{e}^{-\beta(m/10)\cdot 6000/m}\end{aligned}$$

donde:

$$\beta(c) = c \ln c - c + 1$$

Por la cota de la unión, la probabilidad de que *alguna* de las máquinas se sobrecargue es:

$$\begin{aligned}\Pr[\text{alguna máquina se sobrecarga}] &\leq \sum_{1 \leq i \leq m} \Pr[\text{el servidor } i \text{ se sobrecarga}] \\ &= m \Pr[\text{el servidor 1 se sobrecarga}] \\ &\leq m\mathrm{e}^{-\beta(m/10)\cdot 6000/m}\end{aligned}$$

Algunos valores se tabulan a continuación:

$$\begin{array}{ll} m = 11: & 0{,}784\ldots \\ m = 12: & 0{,}000999\ldots \\ m = 13: & 0{,}0000000760\ldots \end{array}$$

O sea, con 11 máquinas alguna puede caerse casi inmediatamente, 12 debieran durar unos días, y 13 dan para un siglo o dos.

Un resultado relevante es que si se revisan al azar dos de las máquinas, y se elige aquella con menos carga, la carga máxima esperada disminuye muy substancialmente (ver por ejemplo Mitzenmacher, Richa y Sitaraman [19]). Esto es importante porque no es necesario imponer la carga extra de revisar todas las máquinas, además que es fácil de hacer las consultas en paralelo.

El resultado clásico es que si se distribuyen m bolas en m casilleros, con alta probabilidad el casillero más lleno contiene:

$$(1 + o(1))\frac{\ln m}{\ln \ln m}$$

Azar, Broder, Karlin y Upfal [3] demuestran que si hay m casilleros y se distribuyen n ($n > m$) bolas entre ellos, si se eligen d casilleros al azar cada vez y se pone la bola en el más vacío donde $d \geq 2$, el largo de la cola más larga con alta probabilidad es:

$$(1 + o(1))\frac{\ln \ln m}{\ln d} + \Theta(n/m)$$

La derivación es compleja, no la repetiremos acá. Vale decir, de $d = 1$ a $d = 2$ hay una mejora exponencial; para $d \geq 2$ el cambio es solo en un factor constante moderado.

33.5. Cotas inferiores a números de Ramsey

Una aplicación del método probabilístico es la demostración de Erdős [9] de una cota inferior para el número de Ramsey $R(r,r)$.

El teorema de Ramsey [25] 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 33.1. $R(3,3) = 6$

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 33.1 muestra K_5 con arcos coloreados de rojo y azul que no contiene K_3 del mismo color, mostrando que $R(3,3) > 5$. □

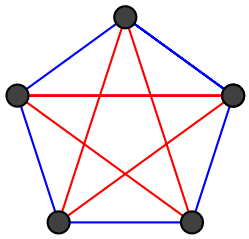


Figura 33.1 – K_5 sin K_3 monocromático

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.

33.6. 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 [11] 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 $\mathbf{a}$ 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 $\mathbf{a}$ no requiere multiplicaciones, solo sumar o no los coeficientes.

33.7. 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:

$$\begin{aligned}
\mathbb{E}\left[\sum_{i<j} X_{ij}\right] &= \sum_{i<j} \mathbb{E}[X_{ij}] \\
&= \sum_{1\le i<j\le n} \frac{2}{j-i+1} \\
&= \sum_{1\le i\le n-1} \sum_{i<j\le n} \frac{2}{j-i+1} \\
&= 2\sum_{1\le i\le n-1} \sum_{2\le k\le n-i+1} \frac{1}{k} \\
&= 2\sum_{1\le i\le n-1} (H_{n-i+1}-1) \\
&= 2\sum_{1\le i\le n-1} H_{n-i+1} - 2(n-1) \\
&= 2\sum_{2\le k\le n} H_k - 2(n-1) \\
&= 2\sum_{1\le k\le n} H_k - 2n \\
&= 2\left((n+1)H_n - n\right) - 2n \\
&= 2(n+1)H_n - 4n
\end{aligned}$$

Igual que lo que obtuvimos antes (27.2).

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:

$$\begin{aligned}
\Pr\left[\text{número de buenas divisiones} < \frac{1}{3}a\ln n\right] &\le \mathrm{e}^{-\beta(1/c)\cdot\frac{1}{3}ca\ln n} \\
&= n^{-\beta(1/c)ca/3}
\end{aligned}$$

Podemos elegir c de manera que $\beta(1/c)ca/3 \ge 2$ (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.

33.8. 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 [22]; un rápido resumen da por ejemplo Williams [28]).

Concretamente, supongamos que el archivo de marras es de 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 $\mathbf{a}$ y $\mathbf{b}$ van a detectar si no son iguales. Acá la garantía es sobre una distribución de $\mathbf{a}$ y $\mathbf{b}$. Para los algoritmos comunes hay técnicas para «falsificar» CRC (ver por ejemplo Stigge y otros [27]), 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 $\mathbf{a}$ y $\mathbf{b}$, 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 $\mathbf{a}$ y $\mathbf{b}$, 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 [6, sección 9.3]):

Teorema 33.2. *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 [8] 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 33.2.

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 $O(m \log n)$.

Algoritmo 33.2: 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
```

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 [14] 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.

33.9. 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 [16] y el de Boyer-Moore [5] con la modificación de Galil [12], pero son muy complicados.

Discutiremos el algoritmo de Karp y Rabin [15]. 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 33.3. Podemos verificar probables calces comparando símbolo a símbolo.

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 esto en muchas posiciones, haciendo que debamos recurrir a comparaciones inútiles (llegando a tiempo $O(nm)$).

Algoritmo 33.3: El algoritmo de Karp-Rabin para calces de patrones

Elija p al azar como indicado

$h \leftarrow 0$
for $k \leftarrow 1$ **to** n **do**
    $h \leftarrow h \cdot b + \sigma_k$
end

$s \leftarrow 0$
for $k \leftarrow 1$ **to** n **do**
    $s \leftarrow s \cdot b + \omega_k$
end

$k \leftarrow n$
while $(s \neq h) \wedge (k < m)$ **do**
    $k \leftarrow k + 1$
    $s \leftarrow (s - \omega_{k-n} \cdot b^{n-1}) \cdot b + \omega_k$
end
if $s = h$ **then**
    return probable calce en $k - n$
else
    return no hay calce
end

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] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal: *Balanced allocations*. SIAM Journal on Computing, 29(1):180–200, February 2000.

[4] Lázló Babai: *Monte-Carlo algorithms in graph isomorphism testing*. Technical Report D.M.S. 79-10, Université de Montreal, 1979.

[5] R. S. Boyer and J. Strother Moore: *A fast string searching algorithm*. Communications of the ACM, 20(10):762–772, October 1977.

[6] Horst H. von Brand: *Fundamentos de Informática*. `git://csrg.inf.utfsm.cl/vonbrand/Ramos/trainwreck`, Septiembre 2017. Versión 0.84.

[7] 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.

[8] Paul Erdős: *Beweis eines Satzes von Tschebyschef*. Acta Scientiarum Mathematicarum Szegedensis, 5(3-4):194–198, 1930-1932.

[9] Paul Erdős: *Some remarks on the theory of graphs*. Bulletin of the American Mathematical Society, 53(4):292–294, April 1947.

[10] J. Finn: *Comparison of probabilistic tests for primality*. Unpublished manuscript, 1982.

[11] 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.

[12] 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.

[13] Juraj Hromkovič: *Design and Analysis of Randomized Algorithms*. Texts in Theoretical Computer Science. Springer, 2005.

[14] Richard M. Karp: *An introduction to randomized algorithms*. Discrete Applied Mathematics, 34(1-3):165–201, November 1991.

[15] Richard M. Karp and Michael O. Rabin: *Efficient randomized pattern-matching algorithms*. IBM Journal of Research and Development, 31(2):249–260, March 1987.

[16] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt: *Fast pattern matching in strings*. SIAM Journal of Computing, 6(2):323–350, 1977.

[17] M. Douglas McIllroy: *A killer adversary for Quicksort*. Software: Practice and Experience, 29(4):341–344, April 1999.

[18] Gary L. Miller: *Riemann's hypothesis and tests for primality*. Journal of Computer and System Sciences, 13(3):300–317, December 1976.

[19] Michael David Mitzenmacher, Andréa W. Richa, and Ramesh Sitaraman: *The power of two random choices: A survey of techniques and results*. In Sanguthevar Rajasekaran, Panos M. Pardalos, John H. Reif, and José Rolim (editors): *Handbook of Randomized Computing*, volume I, pages 255–312. Kluwer Academic Publishers, 2001.

[20] Rajeev Motwani and Prabhakar Raghavan: *Randomized Algorithms*. Cambridge University Press, 1995.

[21] Rajeev Motwani and Prabhakar Raghavan: *Randomized algorithms*. ACM Computing Surveys, 28(1):33–37, March 1996.

[22] W. W. Peterson and D. T. Brown: *Cyclic codes for error detection*. Proceedings of the IRE, 49(1):228–235, January 1961.

[23] William Pugh: *Skip Lists: A probabilistic alternative to balanced trees*. Communications of the ACM, 33(6):668–676, June 1990.

[24] Michael O. Rabin: *Probabilistic algorithm for testing primality*. Journal of Number Theory, 12(1):128–138, February 1980.

[25] Frank P. Ramsey: *On a problem in formal logic*. Proceedings of the London Mathematical Society, s2-30(1):264–268, 1930.

[26] Raimund G. Seidel and Cecilia A. Aragon: *Randomized search trees*. Algorithmica, 16(4/5):464–497, October 1996.

[27] 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.

[28] 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.

Clase 34

Algoritmos aproximados

Vimos que muchos problemas de búsqueda importantes son NP-completos, con lo que la esperanza de obtener soluciones exactas es muy remota para instancias grandes. Nos contentaremos entonces con aproximaciones, y nos interesa saber qué tan buenas son. La presente clase se adapta de Dasgupta, Papadimitou y Vazirani [3] y de Mount [6]. Un texto que cubre el área en detalle es el de Vazirani [8], un borrador previo es [8].

Siendo importantes problemas prácticos, simplemente abandonar no es opción. Nuestras alternativas generales son:

Usar fuerza bruta: O al menos algún mecanismo de búsqueda exhaustiva, como *branch and bound* o A^*. Hasta con los computadores paralelos más grandes, esto es viable solo con instancias pequeñas del problema.

Heurísticas: Una *heurística* es una técnica que construye una solución válida, sin garantía de qué tan cerca está del óptimo. La heurística se basa en algún criterio que sugiere que la solución es buena. Se aplica a falta de otras opciones, o si basta una solución válida y no es muy relevante qué tan cerca del óptimo está.

Usar un algoritmo aleatorizado: Esto se discute en el capítulo 33. Generalmente entregan una solución aproximada con costo moderado, una opción es correr el algoritmo varias veces y retener la mejor solución.

Algoritmo aproximado: Es un algoritmo que se ejecuta en tiempo polinomial (idealmente), y entrega una solución que está dentro de un factor garantizado del óptimo.

34.1. Cotas de rendimiento

Por razones teóricas, planteamos los problemas NP-completos como problemas de decisión (¿Hay una *clique* de tamaño k en el grafo G?, ¿Es satisfacible la fórmula ϕ?), pero vimos que el problema de búsqueda relacionado (Dé una *clique* de tamaño k del grafo G, si la hay. Dé una asignación de valores a las variables de ϕ que la hacen verdadera, si existe.) o de optimización (Dé una *clique* de máximo tamaño del grafo G. Indique el ciclo de mínimo costo en un grafo dirigido con arcos rotulados con costos.) son «equivalentes» para problemas NP-completos. No nos extenderemos en esto, detalles dan Bellare y Goldwasser [2], un resumen accesible es el de Bellare [1]. Nótese que a

veces maximizamos y otras minimizamos. Un algoritmo aproximado entrega una solución válida, no necesariamente óptima.

¿Cómo medir qué tan buena es la aproximación? Dada una instancia I del problema, llamamos $C(I)$ al costo de la solución provista por el algoritmo, y $C^*(I)$ al costo óptimo respectivo. Supondremos que los costos son estrictamente positivos, con lo que buscamos $C(I)/C^*(I)$ pequeño si es minimización, para maximización $C^*(I)/C(I)$ pequeño. Para datos de tamaño n, decimos que el algoritmo logra *cota de razón* $\rho(n)$ si para todas las instancias I:

$$\max_{|I|=n}\left\{\frac{C(I)}{C^*(I)}, \frac{C^*(I)}{C(I)}\right\} \leq \rho(n)$$

Note que $\rho(n)$ siempre es mayor o igual a 1, siendo 1 solo si la solución aproximada es el óptimo.

Algunos problemas NP-completos pueden aproximarse arbitrariamente. Para ellos hay un algoritmo que toma una instancia I y un real $\epsilon > 0$, se ejecuta en tiempo polinomial en $n = |I|$ y retorna una solución cuya cota de razón es a lo más $1+\epsilon$. A este tipo de algoritmo se le llama *polynomial time approximation scheme* o PTAS. El tiempo de ejecución depende de n y ϵ, en el caso de problemas NP-completos en muchos casos al disminuir ϵ el tiempo de ejecución crece más allá de polinomial, como es $O(n^{1/\epsilon})$. Si el tiempo de ejecución es polinomial en n y $1/\epsilon$, se habla de *fully polynomial time approximation scheme* o FPTAS. Ejemplo es tiempo de ejecución $O((1/\epsilon)^2 n^3)$, mientras $O(n^{1/\epsilon})$ y $O(2^{1/\epsilon} n^2)$ no lo son.

Aunque los problemas NP-completos son equivalentes respecto de si sus peores casos pueden resolverse exactamente en tiempo polinomial, su aproximabilidad varía considerablemente.

- Para algunos problemas, es muy poco probable que haya algoritmos aproximados. Por ejemplo, en el caso general del problema del vendedor viajero (TSP), si hay un algoritmo aproximado con una cota de razón menor que ∞, entonces P = NP.
- Algunos problemas pueden aproximarse, pero la cota es una función que crece lentamente con n. Por ejemplo SET COVER puede aproximarse dentro de un factor de $\ln n$.
- Hay problemas que se pueden aproximar dentro de un factor fijo, veremos un par de ejemplos más adelante.
- Hay problemas que tienen PTAS o FPTAS.

De hecho, similar al caso de los problemas NP-completos, hay colecciones de problemas que «se cree» que son equivalentes en que son difíciles de aproximar, y que si uno puede aproximarse en tiempo polinomial todos ellos pueden serlo. Pero el estudio de algoritmos aproximados sirve para llenar varios cursos...

34.2. Algunos algoritmos aproximados

Veremos algunos ejemplos de algoritmos aproximados, y las demostraciones de su rendimiento. Volveremos sobre algunos de los ejemplos ya tratados para ello.

34.2.1. El problema VERTEX COVER

El problema de optimización VERTEX COVER toma un grafo $G = (V, E)$ y solicita un subconjunto mínimo $C \subseteq V$ tal que todos los arcos de G contienen al menos un vértice en C.

Sabemos que el problema de decisión correspondiente (¿tiene G un *vertex cover* de tamaño k?) es NP-completo. Demostraremos que hay un algoritmo con cota de razón 2, o sea, obtiene un conjunto que a lo más tiene el doble tamaño del mínimo.

Una manera de diseñar un algoritmo aproximado es tomar una heurística, alguna «estrategia razonable», técnica que en muchos casos da buenos resultados. Acá tenemos un algoritmo muy simple, que se basa en una observación obvia: si consideramos el arco $uv \in E$, al menos uno de los dos vértices pertenece al conjunto, no sabemos cuál. Agregamos ambos a nuestro conjunto (más tonto, imposible), luego eliminamos los arcos que contienen los vértices u, v, y repetimos el proceso con el resto. Llamemos ApproxVC al algoritmo 34.1. Note que es un algoritmo aleatorizado (elige un arco cualquiera en cada paso).

Algoritmo 34.1: El algoritmo ApproxVC

```
function ApproxVC(G)
    C ← ∅
    while E ≠ ∅ do
        Sea uv ∈ E un arco cualquiera
        C ← C ∪ {u, v}
        Elimine de E todos los arcos incidentes a u o v
    end
    return C
end
```

Proposición 34.1. *El algoritmo* ApproxVC *tiene cota de razón* 2 *para* VERTEX COVER.

Demostración. Sea C el conjunto que entrega ApproxVC, y sea C^* el mínimo. Sea A el conjunto de arcos seleccionados por ApproxVC, vemos que $|C| = 2|A|$ ya que cada arco seleccionado aporta 2 vértices a C. Pero el *vertex cover* mínimo tiene que contener al menos uno de esos dos vértices, con lo que $|A| \leq |C^*|$. Tenemos:

$$\frac{|C|}{2} = |A| \leq |C^*|$$

vale decir:

$$\frac{|C|}{|C^*|} \leq 2$$

□

Este ejemplo es típico: necesitamos hallar alguna cota para la solución óptima en términos de lo que entrega o hace el algoritmo.

Hay una familia infinita de grafos para los cuales ApproxVC da cota de razón exactamente 2: en el grafo bipartito completo $K_{n,n}$ el algoritmo ApproxVC elige todos los vértices, para un total de $2n$; el óptimo es uno de los conjuntos de vértices, con n vértices. A esto se le llama *ejemplo ajustado* (*tight example* en inglés) para el algoritmo.

Otra idea es una estrategia voraz: ¿porqué no concentrarse en vértices que cubren el máximo número de arcos? O sea, ir agregando vértices de grado máximo. Esto da el algoritmo 34.2. Este algoritmo no siempre da una solución mejor que la estrategia estúpida de dos-por-uno. Sorprendentemente, actualmente ApproxVC es el algoritmo aproximado que garantiza la mejor cota de razón para este problema. Es un ejercicio moderadamente difícil demostrar que para GreedyVC(G) la cota de razón crece como $\Theta(\log n)$, donde n es el número de vértices del grafo, ni siquiera es una constante. Vale decir, puede comportarse arbitrariamente peor que ApproxVC. Por otro lado, en «grafos típicos» suele dar mejores resultados que ApproxVC, vale la pena correr ambos (y ya que ApproxVC

Algoritmo 34.2: El algoritmo GreedyVC

```
function GreedyVC(G)
    C ← ∅
    while E ≠ ∅ do
        Sea u el vértice de mayor grado en G
        C ← C ∪ {u}
        Elimine de E todos los arcos que contienen a u
    end
    return C
end
```

es aleatorizado, correr este varias veces) y quedarse con el conjunto más pequeño. O combinar: usar ApproxVC, pero elegir siempre el arco uv con máximo $\delta(u)+\delta(v)$, con la idea de eliminar los más arcos posibles.

La moraleja es que no siempre la heurística «obvia» da el mejor resultado, puede ser necesario considerar varias alternativas.

34.2.2. El problema del vendedor viajero

El problema del vendedor viajero (*Travelling Salesman Problem*, o su versión «sexistamente correcta» *Travelling Salesperson Problem*, abreviado TSP) es el niño símbolo de lo NP-completo. Tenemos un grafo $G=(V,E)$ con costos de los arcos, $c\colon E\to\mathbb{R}^+$. Buscamos un ciclo de costo mínimo (sumando los costos de los arcos) que visite cada vértice exactamente una vez.

34.2.2.1. No hay algoritmo aproximado para TSP

El problema TSC es similar al problema de ciclo Hamiltoniano (abreviado HAM), que dado un grafo $G=(V,E)$ pregunta si existe un ciclo que pasa por cada vértice exactamente una vez. Usaremos esta similitud para demostrar que es imposible aproximar TSP.

Teorema 34.1. *No hay un algoritmo aproximado polinomial para* TSP *que dé una cota de razón menor que* ∞ *a menos que* $\mathsf{P}=\mathsf{NP}$.

Demostración. Esto lo haremos reduciendo el problema HAM a obtener una aproximación a TSP. Sea $G=(V,E)$ una instancia de HAM, y sea $n=|V|$, elegimos una constante C y creamos una instancia de TSP con el grafo $G'=K_n$ sobre los vértices V, con:

$$c(u,v)=\begin{cases}1 & uv\in E\\ C & uv\notin E\end{cases}$$

Es claro que nuestra instancia de TSP tiene solución de costo total n si y solo si la instancia de HAM tiene solución. Si HAM no tiene solución, el costo de la travesía es a lo menos $n-1+C$. Si hubiese un algoritmo polinomial que dé $\rho(n)\le(C+n-1)/n$, tendríamos un algoritmo polinomial para HAM (si da una solución de costo menor a $C+n-1$, tiene costo n, quiere decir que HAM tiene solución). Pero C es arbitrario y HAM es NP-completo. □

34.2.2.2. Vendedor viajero con desigualdad triangular

Una variante de TSP, que llamaremos TSP_Δ, se da si para todos los vértices los costos cumplen la desigualdad triangular:

$$c(u,v) \leq c(u,x) + c(x,v)$$

Un caso especial de esto es cuando los vértices son puntos en el plano y los costos son las distancias entre ellos.

Teorema 34.2. *Para* TSP_Δ *hay un algoritmo polinomial que asegura cota de razón* 2.

Demostración. Primeramente, sea C^* el costo del ciclo óptimo para la instancia de TSP. Si eliminamos uno de los arcos del ciclo obtenemos un árbol recubridor del grafo, cuyo costo es a lo menos el del árbol recubridor mínimo del grafo, llamémosle T^*. Por otro lado, podemos construir un circuito que visita cada vértice dos veces partiendo de un árbol recubridor mínimo: recorrer el árbol «por fuera» da un circuito cuyo costo es $2T^*$; si en el recorrido (básicamente un recorrido en preorden) evitamos los vértices ya visitados, obtenemos un ciclo, como estamos evitando vértices, por la desigualdad triangular el costo es menor. Resumiendo, obtenemos un recorrido de costo C que cumple:

$$T^* < C^* \leq C \leq 2T^* < 2C^*$$

de donde tenemos:

$$\frac{C}{C^*} < 2$$

□

34.2.3. El problema SET COVER

Otro problema NP-completo famoso es SET COVER: Dado un conjunto universo finito $\mathscr{U}$ y una familia finita de subconjuntos $\mathscr{S}_i$, $1 \leq i \leq n$, tal que $\bigcup_i \mathscr{S}_i = \mathscr{U}$ (los $\mathscr{S}_i$ *no* son necesariamente disjuntos), se busca la colección mínima de los $\mathscr{S}_i$ tal que:

$$\bigcup_{i \in \mathscr{C}} \mathscr{S}_i = \mathscr{U}$$

El problema VERTEX COVER es un caso particular, los arcos de G son subconjuntos de tamaño 2 de los vértices, y nos interesa la colección mínima de arcos que incluyen a todos los vértices. La estrategia que llevó a ApproxVC acá no es aplicable, los conjuntos $\mathscr{S}_i$ no son necesariamente del mismo tamaño.

Una heurística plausible es la estrategia voraz de elegir en cada paso aquel subconjunto que más elementos aún no considerados incluye. Esta heurística puede ser engañada, como ilustra la figura 34.1. La estrategia voraz elegiría el conjunto de 8 elementos, luego el de 4, y finalmente el de

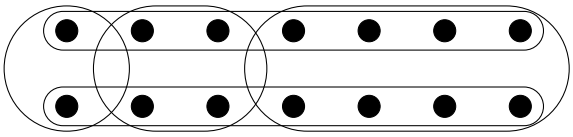


Figura 34.1 – Engañando a la heurística para SET COVER

2; la solución óptima es elegir simplemente ambos conjuntos de 7.

Antes de ir al resultado, recordamos una cota importante.

Lema 34.3. *Para todo $c > 0$:*

$$\left(1 - \frac{1}{c}\right)^c \leq \frac{1}{\mathrm{e}}$$

Demostración. Usamos que para todo x es $1 + x \leq \mathrm{e}^x$ (son iguales para $x = 0$ únicamente). Substituyendo $x = -1/c$ tenemos $1 - 1/c \leq \mathrm{e}^{-1/c}$, elevando a la potencia c tenemos lo prometido. □

Tenemos:

Teorema 34.4. *El algoritmo voraz para* SET COVER *tiene una cota de razón de a lo más* $\ln m$*, donde* $m = |\mathscr{U}|$.

Demostración. Sea c el tamaño del *set cover* óptimo, y sea g el tamaño del *set cover* entregado por el algoritmo voraz menos uno. Demostraremos que $g/c \leq \ln m$, lo que no es exactamente lo prometido, pero está a 1 de distancia.

Al iniciar el algoritmo voraz, tenemos $m_0 = m$ elementos a cubrir. Sabemos que hay una cobertura de tamaño c (la óptima), con lo que por el principio del palomar hay a lo menos un conjunto de m_0/c elementos (si todos fueran menores a m_0/c, en total c conjuntos no cubrirían m_0 elementos). Al elegir el conjunto más grande, el algoritmo voraz elige un conjunto con al menos m_0/c elementos, quedando $m_1 \leq m_0 - m_0/c = m_0(1 - 1/c)$ elementos a cubrir. Aplicando el mismo argumento, hay forma de cubrir los m_1 elementos restantes con $c - 1$ conjuntos, quedan $m_2 \leq m_1(1 - 1/(c-1)) \leq m_0(1 - 1/c)^2$, y así sucesivamente. Aplicando este argumento g veces, cada vez cubrimos una fracción de al menos $1 - 1/c$ de los elementos, los elementos que restan al final del algoritmo voraz es a lo más $m(1 - 1/c)^g$. El valor máximo de g para el cual queda al menos un elemento sin cubrir es:

$$\begin{aligned} 1 &\leq m\left(1 - \frac{1}{c}\right)^g \\ &= m\left(\left(1 - \frac{1}{c}\right)^c\right)^{g/c} \end{aligned}$$

Por el lema 34.3:

$$1 \leq m\mathrm{e}^{-g/c}$$

Multiplicando por $\mathrm{e}^{g/c}$ y tomando logaritmos:

$$\frac{g}{c} \leq \ln m$$

□

En la práctica, la cota del teorema 34.4 resulta ser demasiado pesimista.

34.2.4. El problema de la mochila

El problema de la mochila (KNAPSACK) es otro problema NP-completo importante. Tenemos una mochila de capacidad M y n objetos, con el objeto i de peso p_i y valor v_i. Buscamos el conjunto de objetos que da el valor máximo. Este problema podemos plantearlo como:

$$\begin{aligned} &\text{máx} \sum_i x_i v_i \\ &\text{tal que} \sum_i x_i p_i \leq M \\ &x_i \in \{0, 1\} \end{aligned}$$

Claramente objetos de peso mayor que M nunca pueden formar parte de la solución, podemos suponer sin pérdida de generalidad que $p_i < M$. Suponemos además que tanto M como los p_i son enteros. Nuestra discusión sigue a Nelson [7].

34.2.4.1. Algoritmos voraces

Sabemos (sección 13.3) que si se permiten fracciones de objetos, el algoritmo voraz de incluir los objetos en orden de v_i/p_i decreciente, agregando todo lo que se puede del último es óptimo. Sin embargo, en este caso esto no funciona. Por ejemplo, con $p_1 = 1, v_1 = 1+\epsilon, p_2 = M, v_2 = M$ elige solo el objeto 1, para un valor $1+\epsilon$; siendo que eligiendo 2 obtiene un valor M. La aproximación no es mejor que M, y M es arbitrario.

34.2.4.2. Un FPTAS para KNAPSACK

La idea central es usar redondeo. Si $V = \sum_i v_i$, sabemos que hay un algoritmo de programación dinámica que resuelve el problema en tiempo $O(nV)$. Para obtener un FPTAS, defina:

$$v'_i = \left\lfloor \frac{n}{\epsilon} \cdot \frac{v_i}{v_{\max}} \right\rfloor$$

Ejecute el algoritmo de programación dinámica para obtener la solución óptima OPT$'$ al problema modificado, con costo $O(nV')$. Sabemos:

$$V' \le n \max_i v'_i \le \frac{n^2}{\epsilon}$$

Es claro que $\text{OPT}' \ge v'_{\max} \ge n/\epsilon$. Después de redondear, todo conjunto de k objetos pierde a lo más $k < n$ en valor, lo que (por lo anterior) es a lo más $\epsilon\,\text{OPT}'$.

Proposición 34.2. *Para $\epsilon > 0$, la solución óptima de la instancia redondeada tiene valor al menos* $(1-\epsilon)\,\text{OPT}$.

Demostración. Sea $v(S)$ el valor del conjunto S de objetos, $v'(S)$ el valor del conjunto con valores redondeados. Para todo conjunto S:

$$\alpha\, v(S) - |S| \le v'(S) \le \alpha\, v(S)$$

donde $\alpha = n/(\epsilon\, v_{\max})$. Sea A^* algún conjunto óptimo para el problema original sin redondear y A un conjunto óptimo del problema redondeado, entonces:

$$v(A) \ge \frac{1}{\alpha} v'(A) \ge \frac{1}{\alpha} v'(A^*) \ge \frac{1}{\alpha} v(A^*) - \frac{1}{\alpha}|A^*| \ge \text{OPT} - \frac{n}{\alpha} \ge \text{OPT} - \epsilon\, v_{\max} \ge \text{OPT} - \epsilon\,\text{OPT}$$

□

Dado $\epsilon > 0$, el esquema de redondeo nos da un algoritmo que entrega una solución de valor $(1-\epsilon)\,\text{OPT}$ en tiempo $O(n^3/\epsilon)$, es un FPTAS.

Lawler [5] da un FPTAS de tiempo $O(n\log(1/\epsilon) + 1/\epsilon^4)$, una versión preliminar es [4].

Ejercicios

1. Usando los grafos construidos de la siguiente forma: tenemos un nivel base de m vértices, un segundo nivel de $m/2$ vértices, un tercer nivel de $m/3$ vértices, ..., un último nivel de 1 vértice. Todos los arcos terminan en el nivel base; cada vértice del nivel base está conectado a un vértice de cada nivel siguiente, distribuidos de forma lo más equitativa posible.

   Demuestre que en estos grafos la cota de razón para el algoritmo GreedyVC(G) cumple $\Theta(\log n)$, donde n es el número de vértices del grafo.

2. Muestre cómo extender el ejemplo de la figura 34.1 para construir un ejemplo ajustado para la heurística voraz para SET COVER.

Bibliografía

[1] Mihir Bellare: *Decision versus search.* https://cseweb.ucsd.edu/~mihir/cse200/decision-search.pdf, January 2010.

[2] Mihir Bellare and Shafi Goldwasser: *The complexity of decision versus search.* SIAM Journal of Computing, 23(1):97–119, February 1994.

[3] Sanjoy Dasgupta, Christos H. Papadimitrou, and Umesh V. Vazirani: *Algorithms.* McGraw Hill Higher Education, 2006.

[4] Eugene L. Lawler: *Fast approximation algorithms for knapsack problems.* In *18tn Annual Symposium on Foundations of Computer Science*, 1977.

[5] Eugene L. Lawler: *Fast approximation algorithms for knapsack problems.* Mathematics of Operations Research, 4(4):339–356, 1979.

[6] David M. Mount: *CMSC 451: Design and analysis of computer algorithms.* https://www.cs.umd.edu/class/fall2013/cmsc451/Lects/cmsc451-fall13-lects.pdf, Fall 2013. Department of Computer Science, University of Maryland.

[7] Jelani Nelson: *CS 224: Advanced algorithms.* http://people.seas.harvard.edu/~minilek/cs224/fall14/lec.html, Fall 2014. Department of Computer Science, Harvard University.

[8] Vijay V. Vazirani: *Approximation Algorithms.* Springer, 2003.

Apéndice A

Píldoras de funciones generatrices

La idea de funciones generatrices es usar una serie para representar una secuencia. Resulta que en muchos casos manipular la serie es mucho más fácil que trabajar con la secuencia, y se pueden obtener resultados sorprendentes en forma muy sencilla. Véase el apéndice F para ejemplos. Mucho más detalle se da en los apuntes de Fundamentos de Informática [1], en los textos de Wilf [9] y de Flajolet y Sedgewick [3], y muy particularmente aplicado a nuestro tema en el de Sedgewick y Flajolet [8].

Wilf [9] expresa que la función generatriz es una línea de ropa de la cual se cuelgan los coeficientes para exhibición. Entendemos el exponente de z como un contador, índice del coeficiente correspondiente. Como veremos, operaciones sobre la función generatriz corresponden a actuar sobre la secuencia, en muchos casos resulta más sencillo manipular la serie que trabajar con la secuencia. Para nuestros efectos, en general basta manipularlos como si fueran «polinomios infinitos». Si tenemos la suerte que la serie converge para algún rango alrededor de $z = 0$ (como en nuestros ejemplos), podremos aplicar las herramientas del cálculo.

A.1. Ejemplos combinatorios

Al lanzar dos dados tradicionales las sumas 2 y 12 se pueden obtener de una única manera, mientras para 4 hay tres ($1+3 = 2+2 = 3+1$). Representamos un dado mediante el polinomio:

$$D(z) = z + z^2 + z^3 + z^4 + z^5 + z^6 \tag{A.1}$$

con lo cual para lanzamientos de dos dados:

$$D^2(z) = z^2 + 2z^3 + 3z^4 + 4z^5 + 5z^6 + 6z^7 + 5z^8 + 4z^9 + 3z^{10} + 2z^{11} + z^{12} \tag{A.2}$$

Interesa hallar dados marcados en forma diferente que den la misma distribución de las sumas («dados locos»). Para construirlos buscamos polinomios $D_1(z)$ y $D_2(z)$ que den el producto (A.2). Queremos además que ambas representen dados, o sea tengan 6 caras, y que cada cara debe estar marcada por al menos un punto. El número de caras es simplemente el valor de la función en $z = 1$; que cada cara esté marcada con al menos un punto se traduce en que la función generatriz sea divisible por z (el número de caras marcadas con cero, o sea el coeficiente de z^0, debe ser cero), O sea:

$$D_1(1) = D_2(1) = 6 \tag{A.3}$$

Factorizamos (A.1):

$$D(z) = z(z+1)(z^2-z+1)(z^2+z+1) \tag{A.4}$$

Los factores z y z^2-z+1 tienen valor 1 para $z=1$, $z+1$ da 2 y z^2+z+1 da 3. Tanto $D_1(z)$ como $D_2(z)$ deben tener los factores z, $z+1$ y z^2+z+1; solo quedan por redistribuir los dos factores z^2-z+1 en $D^2(z)$:

$$\begin{aligned} D_1(z) &= z(z+1)(z^2+z+1) \\ &= z+2z^2+2z^3+z^4 \end{aligned} \tag{A.5}$$

$$\begin{aligned} D_2(z) &= z(z+1)(z^2-z+1)^2(z^2+z+1) \\ &= z+z^3+z^4+z^5+z^6+z^8 \end{aligned} \tag{A.6}$$

Los dados marcados con $\{1,2,2,3,3,4\}$ y $\{1,3,4,5,6,8\}$ se conocen como *dados de Sicherman* [4].

A.2. Definiciones formales

Sea una secuencia $\langle a_n\rangle_{n\geq 0} = \langle a_0, a_1, a_2, \ldots, a_n, \ldots\rangle$. La *función generatriz* (ordinaria) de la secuencia es la serie de potencias:

$$A(z) = \sum_{0\leq n} a_n z^n$$

Anotaremos $A(z) \overset{\text{ogf}}{\longleftrightarrow} \langle a_n\rangle_{n\geq 0}$ en este caso (*ogf* es por *Ordinary Generating Function*).

La *función generatriz exponencial* de la secuencia es la serie:

$$\widehat{A}(z) = \sum_{0\leq n} a_n \frac{z^n}{n!}$$

Anotaremos $\widehat{A}(z) \overset{\text{egf}}{\longleftrightarrow} \langle a_n\rangle_{n\geq 0}$ en este caso (*egf* es por *Exponential Generating Function*).

Por comodidad, a veces escribiremos estas relaciones con la función generatriz al lado derecho.

A.2.1. Reglas OGF

Las propiedades siguientes de funciones generatrices ordinarias son directamente las definiciones del caso o son muy simples de demostrar, sus justificaciones detalladas quedarán de ejercicios.

Linealidad: Si $A(z) \overset{\text{ogf}}{\longleftrightarrow} \langle a_n\rangle_{n\geq 0}$ y $B(z) \overset{\text{ogf}}{\longleftrightarrow} \langle b_n\rangle_{n\geq 0}$, y α y β son constantes, entonces:

$$\alpha A(z) + \beta B(z) \overset{\text{ogf}}{\longleftrightarrow} \langle \alpha a_n + \beta b_n\rangle_{n\geq 0}$$

Secuencia desplazada a la izquierda: Si $A(z) \overset{\text{ogf}}{\longleftrightarrow} \langle a_n\rangle_{n\geq 0}$, entonces:

$$\frac{A(z) - a_0 - a_1 z - \cdots - a_{k-1}z^{k-1}}{z^k} \overset{\text{ogf}}{\longleftrightarrow} \langle a_{n+k}\rangle_{n\geq 0}$$

Multiplicar por n: Consideremos:

$$\begin{aligned} A(z) &\overset{\text{ogf}}{\longleftrightarrow} \langle a_n\rangle_{n\geq 0} \\ z\frac{\mathrm{d}}{\mathrm{d}z}A(z) &\overset{\text{ogf}}{\longleftrightarrow} \langle n a_n\rangle_{n\geq 0} \end{aligned}$$

Esta operación se expresa en términos del operador $z\mathrm{D}$ (acá D es por derivada, para abreviar). Además:

$$(z\mathrm{D})^2 A(z) = zD(zDA(z)) \overset{\text{ogf}}{\longleftrightarrow} \left\langle n^2 a_n \right\rangle_{n\geq 0}$$

Nótese que $(z\mathrm{D})^2 = z\mathrm{D} + z^2\mathrm{D}^2$ es diferente de $z^2\mathrm{D}^2$.

Multiplicar por un polinomio en n: Si $p(n)$ es un polinomio, usando el operador $z\mathrm{D}$ varias veces para potencias de n y por linealidad:

$$p(z\mathrm{D})A(z) \overset{\text{ogf}}{\longleftrightarrow} \left\langle p(n)a_n \right\rangle_{n\geq 0}$$

Convolución: Si $A(z) \overset{\text{ogf}}{\longleftrightarrow} \langle a_n \rangle_{n\geq 0}$ y $B(z) \overset{\text{ogf}}{\longleftrightarrow} \langle b_n \rangle_{n\geq 0}$ entonces:

$$A(z)\cdot B(z) \overset{\text{ogf}}{\longleftrightarrow} \left\langle \sum_{0\leq k\leq n} a_k b_{n-k} \right\rangle_{n\geq 0}$$

Sea k un entero positivo y $A(z) \overset{\text{ogf}}{\longleftrightarrow} \langle a_n \rangle_{n\geq 0}$, entonces:

$$(A(z))^k \overset{\text{ogf}}{\longleftrightarrow} \left\langle \sum_{n_1+n_2+\cdots+n_k=n} \left(a_{n_1}\cdot a_{n_2}\cdots a_{n_k}\right) \right\rangle_{n\geq 0}$$

Vale la pena tener presente el caso especial:

$$(A(z))^2 \overset{\text{ogf}}{\longleftrightarrow} \left\langle \sum_{0\leq i\leq n} a_i a_{n-i} \right\rangle_{n\geq 0}$$

Sumas parciales: Supongamos:

$$A(z) \overset{\text{ogf}}{\longleftrightarrow} \langle a_n \rangle_{n\geq 0}$$

Podemos escribir:

$$\sum_{0\leq k\leq n} a_k = \sum_{0\leq k\leq n} 1\cdot a_k$$

Esto no es más que la convolución de las secuencias $\langle 1 \rangle_{n\geq 0}$ y $\langle a_n \rangle_{n\geq 0}$, y la función generatriz de la primera es nuestra vieja conocida, la serie geométrica, con lo que:

$$\frac{A(z)}{1-z} \overset{\text{ogf}}{\longleftrightarrow} \left\langle \sum_{0\leq k\leq n} a_k \right\rangle_{n\geq 0} \tag{A.7}$$

A.2.2. Reglas EGF

Las siguientes resumen propiedades de las funciones generatrices exponenciales. Son simples de demostrar, y las justificaciones que no se dan acá quedarán de ejercicios.

Linealidad: Si $\widehat{A}(z) \overset{\text{egf}}{\longleftrightarrow} \langle a_n \rangle_{n\geq 0}$ y $\widehat{B}(z) \overset{\text{egf}}{\longleftrightarrow} \langle b_n \rangle_{n\geq 0}$, y α y β son constantes, entonces:

$$\alpha\widehat{A}(z) + \beta\widehat{B}(z) \overset{\text{egf}}{\longleftrightarrow} \left\langle \alpha a_n + \beta b_n \right\rangle_{n\geq 0}$$

Secuencia desplazada a la izquierda: Si $\widehat{A}(z) \overset{\text{egf}}{\longleftrightarrow} \langle a_n \rangle_{n\geq 0}$, entonces, usando nuevamente D para el operador derivada:

$$\mathrm{D}^k \widehat{A}(z) \overset{\text{egf}}{\longleftrightarrow} \langle a_{n+k} \rangle_{n\geq 0}$$

Multiplicación por un polinomio en n: Si es $\widehat{A}(z) \overset{\text{egf}}{\longleftrightarrow} \langle a_n \rangle_{n\geq 0}$, y p es un polinomio, entonces:

$$p(z\mathrm{D})\widehat{A}(z) \overset{\text{egf}}{\longleftrightarrow} \big\langle p(n) a_n \big\rangle_{n\geq 0}$$

Es la misma que en funciones generatrices ordinarias, ya que la operación $z\mathrm{D}$ no altera el exponente en z^n.

Convolución binomial: Si $\widehat{A}(z) \overset{\text{egf}}{\longleftrightarrow} \langle a_n \rangle_{n\geq 0}$ y $\widehat{B}(z) \overset{\text{egf}}{\longleftrightarrow} \langle b_n \rangle_{n\geq 0}$ entonces:

$$\begin{aligned}\widehat{A}(z)\cdot\widehat{B}(z) &= \sum_{n\geq 0}\Big(\sum_{0\leq k\leq n} \frac{a_k}{k!}\frac{b_{n-k}}{(n-k)!}\Big) z^n \\ &= \sum_{n\geq 0}\Big(\sum_{0\leq k\leq n} \binom{n}{k} a_k b_{n-k}\Big)\frac{z^n}{n!}\end{aligned}$$

Vale decir:

$$\widehat{A}(z)\cdot\widehat{B}(z) \overset{\text{egf}}{\longleftrightarrow} \left\langle \sum_{0\leq k\leq n} \binom{n}{k} a_k b_{n-k} \right\rangle_{n\geq 0}$$

Términos individuales: Es fácil ver que si $\widehat{A}(z) \overset{\text{egf}}{\longleftrightarrow} \langle a_n \rangle_{n\geq 0}$ entonces:

$$a_n = \widehat{A}^{(n)}(0)$$

Esto en realidad no es más que el teorema de Maclaurin.

A.3. El truco $z\mathrm{D}\log$

Los logaritmos ayudan a simplificar expresiones con exponenciales y potencias. Pero terminamos con el logaritmo de una suma si el argumento es una serie, que es algo bastante feo de contemplar. Eliminar el logaritmo se logra derivando:

$$\frac{\mathrm{d}\ln(A)}{\mathrm{d}z} = \frac{A'}{A}$$

Esto es mucho más decente. Multiplicamos por z para reponer la potencia «perdida» al derivar.

A.4. Algunas series útiles

Las expansiones en serie que más aparecen son las siguientes.

A.4.1. Serie geométrica

Es la serie más común en aplicaciones. Si $|z| < 1$, se cumple:

$$\sum_{n\geq 0} z^n = \frac{1}{1-z} \tag{A.8}$$

Una variante importante es la siguiente, expansión válida para $|az| < 1$ (con la convención $0^0 = 1$):

$$\sum_{n\geq 0} a^n z^n = \frac{1}{1-az} \tag{A.9}$$

A.4.2. Teorema del binomio

Una de las series más importantes es la expansión de la potencia de un binomio:

$$\sum_{n\geq 0}\binom{\alpha}{n}z^n=(1+z)^\alpha \tag{A.10}$$

Siempre que $|z|<1$ esto vale incluso para α complejos, si definimos:

$$\binom{\alpha}{k}=\frac{\alpha}{1}\cdot\frac{\alpha-1}{2}\cdot\frac{\alpha-2}{3}\cdot\cdots\cdot\frac{\alpha-k+1}{k}=\frac{\alpha^{\underline{k}}}{k!} \tag{A.11}$$

y (consistente con la convención que productos vacíos son 1) siempre es:

$$\binom{\alpha}{0}=1 \tag{A.12}$$

A los coeficientes (A.11) se les llama *coeficientes binomiales* por su conexión con la potencia de un binomio. La expansión (A.10) (también conocida como *fórmula de Newton* si α no es un natural) es fácil de demostrar por el teorema de Maclaurin. Resulta que (A.8) es un caso particular de (A.10).

Si α es un entero positivo, la serie (A.10) se reduce a un polinomio (el factor $\alpha^{\underline{k}}$ se anula si $k>\alpha$) y la relación es válida para todo z. Además, en caso que n sea natural podemos escribir:

$$\binom{n}{k}=\frac{n!}{k!(n-k)!} \tag{A.13}$$

Es claro que:

$$\binom{n}{k}=0 \text{ si } k<0 \text{ o } k>n \tag{A.14}$$

Esto con (A.13) sugiere la convención:

$$\frac{1}{k!}=0 \quad \text{si } k<0 \tag{A.15}$$

Nótese la simetría:

$$\binom{n}{k}=\binom{n}{n-k} \tag{A.16}$$

Casos especiales notables de coeficientes binomiales para $\alpha\notin\mathbb{N}$ son los siguientes:

Caso $\alpha=1/2$: Tenemos, como siempre:

$$\binom{1/2}{0}=1 \tag{A.17}$$

Cuando $k \geq 1$:

$$\begin{aligned}
\binom{1/2}{k} &= \frac{\frac{1}{2}\cdot(\frac{1}{2}-1)\cdots(\frac{1}{2}-k+1)}{k!} \\
&= \frac{1}{2^k}\cdot\frac{1\cdot(1-2)\cdot(1-4)\cdots(1-2k+2)}{k!} \\
&= \frac{(-1)^{k-1}}{2^k k!}\cdot(1\cdot 3\cdots(2k-3)) \\
&= \frac{(-1)^{k-1}}{2^k k!}\cdot\frac{1\cdot 2\cdot 3\cdot 4\cdot\cdots\cdot(2k-3)\cdot(2k-2)}{2\cdot 4\cdot 6\cdots(2k-2)} \\
&= \frac{(-1)^{k-1}}{2^k k!}\cdot\frac{(2k-2)!}{2^{k-1}(k-1)!} \\
&= \frac{(-1)^{k-1}}{2^{2k-1}\cdot k}\cdot\frac{(2k-2)!}{(k-1)!(k-1)!} \\
&= \frac{(-1)^{k-1}}{2^{2k-1}\cdot k}\cdot\binom{2k-2}{k-1}
\end{aligned} \tag{A.18}$$

Hay que tener cuidado, la última fórmula no cubre el caso $k = 0$.

Caso $\alpha = -1/2$: Mucha de la derivación es similar a la del caso anterior. Tenemos, para $k > 0$:

$$\begin{aligned}
\binom{-1/2}{k} &= \frac{(-1/2)\cdot(-1/2-1)\cdot\cdots\cdot(-1/2-k+1)}{k!} \\
&= (-1)^k\frac{1}{2^k}\cdot\frac{1\cdot 3\cdots(2k-1)}{k!} \\
&= (-1)^k\frac{1}{2^k}\cdot\frac{(2k)!}{k!2^k k!} \\
&= (-1)^k\frac{1}{2^{2k}}\binom{2k}{k}
\end{aligned} \tag{A.19}$$

Esta fórmula con $k = 0$ da:

$$\binom{-1/2}{0} = 1$$

así no se necesita hacer un caso especial acá.

Caso $\alpha = -n$: Cuando α es un entero negativo, podemos escribir:

$$\binom{-n}{k} = \frac{(-n)^{\underline{k}}}{k!} = (-1)^k\frac{n^{\overline{k}}}{k!} = (-1)^k\frac{(n+k-1)^{\underline{k}}}{k!} = (-1)^k\binom{k+n-1}{n-1} \tag{A.20}$$

Una fórmula cómoda es:

$$\frac{1}{(1-z)^{n+1}} = \sum_{k\geq 0}\binom{n+k}{n}z^k \tag{A.21}$$

Interesantes resultan las series al variar n, no k. Sabemos que $\binom{n}{k} = 0$ si no es que $0 \leq k \leq n$, podremos ahorrarnos los límites de las sumas para simplificar. El índice n o $n+k$ para k fijo da lo mismo al

sumar sobre todo $n \in \mathbb{Z}$:

$$\begin{aligned}\sum_n \binom{n}{k} z^n &= \sum_n \binom{n+k}{k} z^{n+k} \\ &= z^k \sum_n \binom{n+k}{n} z^n \\ &= \frac{z^k}{(1-z)^{k+1}}\end{aligned} \tag{A.22}$$

Al final usamos (A.21). Omitir los rangos de los índices ahorró interminables ajustes.

A.4.3. Otras series

Una serie común es la exponencial:

$$\mathrm{e}^z = \sum_{n\geq 0} \frac{z^n}{n!} \tag{A.23}$$

A veces aparecen funciones trigonométricas:

$$\sin z = \sum_{n\geq 0} (-1)^n \frac{z^{2n+1}}{(2n+1)!} \qquad \cos z = \sum_{n\geq 0} (-1)^n \frac{z^{2n}}{(2n)!}$$

o hiperbólicas:

$$\sinh z = \sum_{n\geq 0} \frac{z^{2n+1}}{(2n+1)!} \qquad \cosh z = \sum_{n\geq 0} \frac{z^{2n}}{(2n)!}$$

Una relación útil es la fórmula de Euler:

$$\mathrm{e}^{u+\mathrm{i}v} = \mathrm{e}^u(\cos v + \mathrm{i}\sin v) \tag{A.24}$$

Es frecuente la serie para el logaritmo, que podemos hallar integrando término a término:

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}z}\ln(1-z) &= -\frac{1}{1-z} = -\sum_{n\geq 0} z^n \\ \ln(1-z) &= -\sum_{n\geq 1} \frac{z^n}{n}\end{aligned} \tag{A.25}$$

Muchos ejemplos adicionales de series útiles se hallan en el texto de Wilf [9].

A.5. Notación para coeficientes

Comúnmente extraeremos el coeficiente de un término de una serie. Generalmente no hay términos con potencias negativas de z, tales coeficientes serán cero. Para esto, dadas las series:

$$A(z) = \sum_{n\geq 0} a_n z^n \qquad B(z) = \sum_{n\geq 0} b_n z^n$$

usaremos la notación:

$$\left[z^n\right] A(z) = a_n$$

Tenemos algunas propiedades simples:

$$\left[z^n\right] z^k A(z) = \begin{cases} 0 & n < k \\ \left[z^{n-k}\right] A(z) & n \geq k \end{cases}$$

$$\left[z^n\right] (\alpha A(z) + \beta B(z)) = \alpha \left[z^n\right] A(z) + \beta \left[z^n\right] B(z)$$

A.6. Aceite de serpiente

La manera tradicional de simplificar sumatorias (particularmente las que involucran coeficientes binomiales) es aplicar identidades u otras manipulaciones de los índices, como magistralmente exponen Knuth [6] y Graham, Knuth y Patashnik [5]. Acá mostramos un método alternativo, que no requiere saber y aplicar una enorme variedad de identidades. Wilf [9] le llama *Snake Oil Method*, por la cura milagrosa que se ve en las películas del viejo oeste. La técnica es bastante simple:

1. Identificar la variable libre, llamémosle n, de la que depende la suma. Sea $f(n)$ nuestra suma.
2. Sea $F(z)$ la función generatriz ordinaria de la secuencia $\langle f(n)\rangle_{n\geq 0}$.
3. Multiplique la suma por z^n y sume sobre n. Tenemos $F(z)$ expresado como una doble suma, sobre n y la variable de la suma original.
4. Intercambie el orden de las sumas, y exprese la suma interna en forma simple y cerrada.
5. Encuentre los coeficientes, son los valores de $f(n)$ buscados.

Sorprende la alta tasa de éxitos de la técnica. Tiene la ventaja de que no requiere mayor creatividad; resulta claro cuándo funciona y es obvio cuando falla.

Usaremos la convención que toda suma sin restricciones es sobre el rango $-\infty$ a ∞. Como los coeficientes binomiales $\binom{n}{k}$ que usaremos en los ejemplos se anulan cuando k no está en el rango $[0,n]$, esto evita interminables ajustes de índices.

Nuestro ejemplo viene de Riordan [7], donde se resuelve mediante delicadas maniobras. Nuestro desarrollo sigue a Dobrushkin [2].

Evaluar:

$$h_n = \sum_{0\leq k\leq n} (-1)^{n-k} 4^k \binom{n+k+1}{2k+1}$$

Definimos $H(z)$ como la función generatriz de los h_n; multiplicamos por z^n, sumamos para $n\geq 0$ e intercambiamos orden de suma:

$$\begin{aligned} H(z) &= \sum_{n\geq 0} z^n \sum_{0\leq k\leq n} (-1)^{n-k} 4^k \binom{n+k+1}{2k+1} \\ &= \sum_{n\geq 0}\sum_{0\leq k\leq n} (-4)^k (-z)^n \binom{n+k+1}{2k+1} \\ &= \sum_{k\geq 0} (-4)^k \sum_{n\geq k} \binom{n+k+1}{2k+1} (-z)^n \end{aligned}$$

Para completar el trabajo necesitamos la suma interna. Haciendo el cambio de variable $r = n-k$:

$$\sum_{n\geq k} \binom{n+k+1}{2k+1} (-z)^n = (-z)^k \sum_{r\geq 0} \binom{r+2k+1}{2k+1} (-z)^r = \frac{(-z)^k}{(1+z)^{2k+2}}$$

Substituyendo en lo anterior:

$$H(z) = \sum_{k\geq 0} \frac{(4z)^k}{(1+z)^{2k+2}} = \frac{1}{(1+z)^2} \cdot \frac{1}{1-\frac{4z}{(1+z)^2}} = \frac{1}{(1-z)^2}$$

Resta extraer los coeficientes, lo que da:

$$h_n = (-1)^n \binom{-2}{n} = \binom{n+1}{1} = n+1$$

A.7. La fórmula de inversión de Lagrange

Es común encontrarse con ecuaciones de la forma

$$u = t\phi(u)$$

donde ϕ es una función dada de u. Esta relación define u en función de t, y «estamos despejando u en términos de t». Fue demostrada por Lagrange y casi simultáneamente por Bürmann, la forma dada acá es la de Bürmann.

Teorema A.1 (Fórmula de inversión de Lagrange)**.** *Sean $f(u)$ y $\phi(u)$ series de potencias en u, con $\phi(0) = 1$. Entonces hay una única serie $u = u(t)$ que cumple:*

$$u = t\phi(u)$$

Además, el valor $f(u(t))$ de f en el cero $u = u(t)$ cumple:

$$\left[t^n\right]\left\{f(u(t))\right\} = \frac{1}{n}\left[u^{n-1}\right]\left\{f'(u)\phi(u)^n\right\}$$

Dadas f y ϕ, esta fórmula da los coeficientes de $f(u(t))$ en bandeja. No demostraremos este resultado, ya que nos llevaría demasiado fuera del rango de este ramo. La demostración del teorema puede encontrarse en el texto de Wilf [9].

Bibliografía

[1] Horst H. von Brand: *Fundamentos de Informática*. git://csrg.inf.utfsm.cl/vonbrand/Ramos/trainwreck, Septiembre 2017. Versión 0.84.

[2] Vladimir A. Dobrushkin: *Methods in Algorithmic Analysis*. Chapman & Hall/CRC Computer and Information Science Series. Chapman & Hall/CRC, 2010.

[3] Philipe Flajolet and Robert Sedgewick: *Analytic Combinatorics*. Cambridge University Press, 2009.

[4] Martin Gardner: *Mathematical games*. Scientific American, 238(2):19–32, February 1978.

[5] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik: *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Professional, second edition, 1994.

[6] Donald E. Knuth: *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, third edition, 1997.

[7] John Riordan: *Combinatorial Identities*. John Wiley & Sons, 1968.

[8] Robert Sedgewick and Philipe Flajolet: *An Introduction to the Analysis of Algorithms*. Addison-Wesley, second edition, 2013.

[9] Herbert S. Wilf: *Generatingfunctionology*. A. K. Peters, Ltd., third edition, 2006.

Apéndice B

Recurrencias

Una *recurrencia* es una descripción recursiva de una función, la describe en términos de ella misma. Como toda definición recursiva, hay algunos *casos base* y *casos recursivos*. Cada caso es una igualdad o desigualdad, donde los casos base dan valores explícitos y los casos recursivos relacionan el valor de la función con valores anteriores.

Decimos que una función *satisface* la recurrencia, o es *solución* de la recurrencia, si para ella cada uno de los casos se cumple. En la mayoría de los casos que encontramos en la práctica hay una solución – podemos escribir una función recursiva de la recurrencia, y la función que esta compute es la solución. Aún más, en casos de interés, si cada uno de los casos es una igualdad la solución es única.

Por sí misma la recurrencia no es una descripción satisfactoria de la función. Buscamos una *forma cerrada*, una descripción no recursiva de la función. Pero una forma cerrada exacta puede no existir, o ser demasiado complicada para uso práctico. Muchas veces nos conformamos con una solución asintótica del tipo $f(n) = \Theta(g(n))$, aunque Sedgewick y Flajolet [3] arguyen que debiéramos ir más allá y dar asintóticas de la forma $f(n) \sim g(n)$.

Para desigualdades recursivas preferimos una solución ajustada, que se cumple incluso si las desigualdades se reemplazan por las igualdades correspondientes. Nuevamente, soluciones ajustadas exactas pueden no existir o ser demasiado complicadas, y nos conformamos con $f(n) = \Omega(g(n))$ o $f(n) = O(g(n))$, según sea el caso.

B.1. El método definitivo: adivinar y verificar

Siempre podemos aplicar la idea de adivinar la solución y verificar que cumple la recurrencia. Esto es esencialmente una demostración por inducción: verificamos que cumple los casos base y los casos recursivos (inducción). Claro que esto traslada el problema a adivinar la solución. Algunas de las secciones siguientes presentan guías que ayudan a adivinar correctamente.

B.1.1. Calcular valores

En el caso de las torres de Hanoi, tenemos la recurrencia:

$$H(n) = 2H(n-1) + 1 \quad H(0) = 0$$

Viendo la recurrencia, sospechamos que la solución es algo como $H(n) = 2^n$, probando algunos valores obtenemos $0, 1, 3, 7, 15, \ldots$, parece que $H(n) = 2^n - 1$. Substituyendo, vemos que cumple la recurrencia.

Acá es relevante la forma en que se dejan los elementos. Si se simplifican demasiado, puede ocultar su forma. No simplificar lo suficiente también puede ser contraproducente.

B.1.2. Desenrollar la recurrencia

Otra alternativa es desenrollar la recurrencia, substituyéndola en sí misma:

$$\begin{aligned} H(n) &= 2H(n-1) + 1 \\ &= 2(2H(n-2) + 1) + 1 = 4H(n-2) + 3 \\ &= 4(2H(n-3) + 1) + 3 = 8H(n-3) + 7 \\ &= \cdots \end{aligned}$$

Da la impresión que al desenrollar resulta $H(n) = 2^k H(n-k) + 2^k - 1$. Esto podemos demostrarlo por inducción, y finalmente deducir $H(n) = 2^n H(0) + 2^n - 1 = 2^n - 1$. Incidentalmente, nos da la dependencia de $H(n)$ del valor inicial.

Los números de Fibonacci dan otro ejemplo:

$$F_{n+2} = F_{n+1} + F_n \quad F_0 = 0, F_1 = 1$$

Los primeros valores $0, 1, 1, 2, 3, 5, \ldots$ no muestran un patrón claro, pero la forma hace sospechar algo de la forma $c\alpha^n$. Intentemos demostrar por inducción que $F_n \leq c\alpha^n$ y ver dónde llegamos:

$$\begin{aligned} F_{n+2} &= F_{n+1} + F_n \\ &\leq c\alpha^{n+1} + c\alpha^n \\ &\leq c\alpha^{n+2} \end{aligned}$$

La última desigualdad se cumple si:

$$\alpha^2 \geq \alpha + 1$$

El mínimo α que funciona es $\alpha = \tau = (1 + \sqrt{5})/2$ (el otro cero de la ecuación es menor, y podemos ignorarlo). Faltan los casos base, que determinan la constante c. Resultan:

$$\begin{aligned} F_0 &= 0 \leq c\tau^0 \\ F_1 &= 1 \leq c\tau^1 \end{aligned}$$

Ambas se cumplen si elegimos $c = 1/\tau$, hemos demostrado

$$F_n \leq \tau^{n-1}$$

Vamos por una cota inferior ahora. Podemos usar la misma estrategia con $F_n \geq d\tau^n$, pero el único valor que funciona en todos los casos es el trivial $d = 0$. Pero nos interesa *que funcione para n grande*, podemos omitir el caso $n = 0$, que nos lleva a concluir que para $n \geq 1$ funciona $d = 1/\tau^2$, y:

$$\tau^{n-2} \leq F_n \leq \tau^{n-1}$$

Uniendo ambas cotas y ajustando la constante es:

$$F_n = \Theta(\tau^n)$$

Veamos la recurrencia:

$$T(n) = \sqrt{n}T(\sqrt{(n)}) + n$$

Es claro que $T(n) > n$, pero no mucho. Si intentamos $T(n) \leq an\log_2 n$, la demostración funciona bien:

$$\begin{aligned} T(n) &= \sqrt{n}T(\sqrt{n}) + n \\ &\leq \sqrt{n}\cdot a\sqrt{n}\log_2\sqrt{n} + n \\ &= \frac{a}{2}n\log_2 n + n \\ &\leq an\log_2 n \end{aligned}$$

siempre que $1 \leq a/2\log_2 n$, o $n \geq 2^{2/a}$. Pero esto funciona para cualquier a, no importa qué tan chico. Esto indica que nuestro intento es muy grande. Hallar b tal que $T(n) \geq bn\log_2 n$ falla, lo que confirma que estamos mal. Debemos buscar alguna función entre n y $n\log_2 n$. Intentemos $n\log_2\log_2 n$:

$$\begin{aligned} T(n) &\leq \sqrt{n}\cdot a\sqrt{n}\log_2\log_2\sqrt{n} + n \\ &= an\log_2\log_2 n - an + n \\ &\leq an\log_2\log_2 n \end{aligned}$$

Lo último siempre y cuando $a \geq 1$, tener una cota inferior para a es fuerte indicación que vamos por el camino correcto. Intentando $T(n) \geq bn\log_2\log_2 n$ tenemos éxito siempre que $b \leq 1$, concluimos que $T(n) = \Theta(n\log\log n)$.

B.1.3. Coeficientes indeterminados

La idea es combinar soluciones parciales, dependiendo de parámetros a determinar. Volvemos a los números de Fibonacci, que llevaron a sospechar soluciones de las formas $c_1\tau^n$ y $c_2\phi^n$, donde τ y ϕ son los ceros de $x^2 = x + 1$:

$$\begin{aligned} \tau &= \frac{1+\sqrt{5}}{2} \\ \phi &= \frac{1-\sqrt{5}}{2} \end{aligned}$$

Substituyendo el intento $F_n = c_1\tau^n + c_2\phi^n$ en la recurrencia vemos que cumple el caso recursivo para todo c_1 y c_2, los casos base dan dos ecuaciones que permiten determinar las constantes:

$$\begin{aligned} F_n &= \frac{\tau^n - \phi^n}{\tau - \phi} \\ &= \frac{(1+\sqrt{5})^n - (1-\sqrt{5})^n}{2^n\sqrt{5}} \end{aligned}$$

Es sorprendente que los números de Fibonacci, claramente enteros, se expresen en términos de números irracionales.

Una variante es lo que Knuth llama *método del repertorio*. La idea es armar un repertorio de secuencias $f(n)$ con sus correspondientes resultados de aplicar los casos recursivos, para construir la solución que nos interesa mediante una combinación lineal. Un ejemplo tomado de Rossmanith [2] es hallar a_n tal que:

$$a_n = na_{n-1} + n^2a_{n-2} - n^4 - 3n^2 + 5 \quad a_0 = 5, a_1 = 9$$

a_n	$a_n - na_{n-1} - n^2 a_{n-2}$
1	$-n^2 - n + 1$
n	$-n^3 + n^2 + 2n$
n^2	$-n^4 + 3n^3 - n^2 - n$

Cuadro B.1 – Valores del operador $G = a_n - na_{n-1} - n^2 a_{n-2}$

Es claro que si a_n es un polinomio en n, lo es $a_n - na_{n-1} - n^2 a_{n-2}$. No tiene sentido ir más allá de n^2, daría potencias mayores a n^4. Armamos el cuadro B.1 con algunos polinomios simples. Buscamos entonces constantes α, β, γ tales que:

$$\alpha G(1) + \beta G(n) + \gamma G(n^2) = -n^4 - 3n^2 + 5$$

Resultan $\alpha = 5$, $\beta = 3$, $\gamma = 1$. Tenemos la suerte que $5 + 3n + n^2$ cumple las condiciones iniciales, y la respuesta es:

$$a_n = n^2 + 3n + 5$$

B.2. Recurrencia lineal de primer orden

Un caso especial importante es la recurrencia lineal de primer orden:

$$a_{n+1} = u_n a_n + f_n$$

Vemos que si dividimos por el *factor sumador*:

$$s_n = \prod_{0 \le k \le n} u_k$$

(lo que presupone que $u_k \neq 0$ en el rango de interés) queda:

$$\frac{a_{n+1}}{s_n} - \frac{a_n}{s_{n-1}} = \frac{f_n}{s_n}$$

Sumando ambos lados obtenemos la solución.

B.3. Transformaciones

En algunos casos las otras técnicas no pueden aplicarse directamente, pero una transformación de la recurrencia la lleva a una forma familiar. Las más importantes son las siguientes:

Transformación de dominio: Defina una nueva función $S(n) = T(f(n))$ para alguna función f que dé una recurrencia simple para S.

Por ejemplo, para mergesort si $M(n)$ cuenta el número de comparaciones la recurrencia exacta es:

$$M(n) = M(\lfloor n/2 \rfloor) + M(\lceil n/2 \rceil) + n - 1 \quad M(0) = 0$$

El cambio de variables $n = 2^k$, $m(k) = M(2^k)$ y la observación $m(0) = M(1) = 0$ lleva esto a la forma manejable:

$$m(k) = 2m(k-1) + 2^k - 1 \quad m(0) = 0$$

Transformación de rango: Defina una nueva función $S(n) = f(T(n))$ para alguna función f que dé una recurrencia simple para S.

Un ejemplo es la solución de Brand [1] de la recurrencia de Ricatti:

$$w_{n+1} = \frac{aw_n + b}{cw_n + d} \quad w_0 \text{ dado}$$

donde $c \neq 0$ y $ad - bc \neq 0$ (si $c = 0$ es una recurrencia lineal de primer orden; si $ad = bc$ se reduce a $w_{n+1} =$ constante). Si substituimos $y_n \mapsto cw_n + d$, queda:

$$y_n = \alpha - \frac{\beta}{y_{n-1}}$$

donde:

$$\alpha = a + d$$
$$\beta = ad - bc$$

Claramente eso solo vale si $ad - bc \neq 0$. Substituyendo ahora:

$$y_n \mapsto \frac{x_{n+1}}{x_n}$$

resulta:

$$x_{n+2} - \alpha x_{n+1} + \beta x_n = 0$$

Necesitamos dos valores iniciales para resolverla, podemos elegir bastante arbitrariamente $x_0 = 1$, dando $x_1 = y_0$, que a su vez podemos obtener de la condición inicial original.

Diferencias: Si aparecen sumas en los casos recursivos puede ser útil aplicar diferencias y buscar una recurrencia para $S(n) = \Delta T(n) = T(n+1) - T(n)$.

B.4. Funciones generatrices

Podemos usar las herramientas de funciones generatrices. Dada la recurrencia, definimos una función generatriz adecuada, y aplicamos las propiedades respectivas para obtener una ecuación de la recurrencia. Resuelta la ecuación, extraemos coeficientes para obtener la solución.

Por ejemplo, los números de Catalan C_n cumplen la recurrencia:

$$C_{n+1} = \sum_{0 \leq k \leq n} C_k C_{n-k} \quad C_0 = 1$$

Definiendo la función generatriz ordinaria:

$$c(z) = \sum_{n \geq 0} C_n z^n$$

de las propiedades vemos que cumple:

$$\frac{c(z) - C_0}{z} = c^2(z)$$

Despejando queda:

$$zc^2(z) - c(z) + 1 = 0$$

Obtenemos:

$$c(z) = \frac{1 \pm \sqrt{1-4z}}{2z}$$

Sabemos que debe ser $c(0) = C_0 = 1$, lo que descarta el signo positivo:

$$c(z) = \frac{1 - \sqrt{1-4z}}{2z}$$

Usando la expansión de$(1-4z)^{1/2}$ por el teorema del binomio y simplificando se obtiene finalmente:

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

Alternativamente, podemos definir:

$$u(z) = c(z) - 1$$

con lo que la ecuación toma la forma:

$$u(z) = z(u(z)+1)^2$$

Usando la fórmula de inversión de Lagrange con $\phi(u) = (u+1)^2$ y $f(u) = u$ se obtiene directamente para $n \geq 1$:

$$\begin{aligned} C_n &= [z^n]u(z) \\ &= \frac{1}{n}[u^{n-1}](1+u)^{2n} \\ &= \frac{1}{n}\binom{2n}{n-1} \\ &= \frac{1}{n+1}\binom{2n}{n} \end{aligned}$$

Casualmente coincide $C_0 = 1$ con esta fórmula.

El número D_n de desarreglos (permutaciones sin puntos fijos) de n elementos cumple la recurrencia:

$$D_{n+1} = nD_n + nD_{n-1} \quad D_0 = 1, D_1 = 0$$

Definimos la función generatriz exponencial:

$$\widehat{d}(z) = \sum_{n\geq 0} D_n \frac{z^n}{n!}$$

Las propiedades dan:

$$\begin{aligned} \widehat{d}'(z) &\overset{\text{egf}}{\longleftrightarrow} \langle D_{n+1}\rangle_{n\geq 0} \\ z\widehat{d}'(z) &\overset{\text{egf}}{\longleftrightarrow} \langle nD_n\rangle_{n\geq 0} \end{aligned}$$

Falta el segundo término del lado derecho:

$$\sum_{n\geq 1} nD_{n-1}\frac{z^n}{n!} = z\sum_{n\geq 1} D_{n-1}\frac{z^{n-1}}{(n-1)!} = z\widehat{d}(z)$$

Combinando las anteriores:

$$\widehat{d}'(z) = z\widehat{d}'(z) + z\widehat{D}(z) \qquad \widehat{d}(0) = D_0 = 1$$
$$\frac{\widehat{d}'(z)}{\widehat{d}(z)} = \frac{z}{1-z}$$

La solución de esta ecuación diferencial es:

$$\widehat{d}(z) = \frac{\mathrm{e}^{-z}}{1-z}$$

Por las propiedades, esto es las sumas parciales de la serie exponencial:

$$\begin{aligned} D_n &= n![z^n]\widehat{d}(z) \\ &= n! \sum_{0\le k\le n} \frac{(-1)^k}{k!} \end{aligned}$$

B.5. Perturbación

Una técnica aplicable para aproximar o acotar la solución a algunas recurrencias es el método de perturbación, que ilustramos mediante un ejemplo tomado de Sedgewick y Flajolet [3]. Sea la recurrencia:

$$a_{n+1} = 2a_n + \frac{a_{n-1}}{n^2} \quad a_0 = 1, a_1 = 2$$

Para valores grandes de n el último término tendrá poco efecto. Consideremos entonces la recurrencia aproximada:

$$b_{n+1} = 2b_n \quad b_0 = 1$$

que tiene la solución obvia $b_n = 2^n$, y comparemos los valores aproximados y exactos:

$$p_n = \frac{a_n}{b_n} = 2^{-n}a_n$$

Substituyendo en la recurrencia exacta:

$$2^{n+1}p_{n+1} = 2\cdot 2^n p_n + \frac{2^{n-1}p_{n-1}}{n^2}$$

Esto se simplifica a:

$$p_{n+1} = p_n + \frac{p_{n-1}}{4n^2} \quad p_0 = p_1 = 1$$

Es claro que los p_n aumentan, por lo que:

$$\begin{aligned} p_{n+1} &\le p_n\left(1 + \frac{1}{4n^2}\right) \\ &\le \prod_{1\le k\le n}\left(1 + \frac{1}{4k^2}\right) \\ &< \prod_{k\ge 1}\left(1 + \frac{1}{4k^2}\right) \end{aligned}$$

Resulta que el producto infinito converge al valor 1,46505, con lo que obtenemos:

$$a_n \sim \alpha 2^n$$

donde sabemos que $1 < \alpha < 1{,}46505$.

Bibliografía

[1] Louis Brand: *A sequence defined by a difference equation.* American Mathematical Monthly, 62(7):489–492, September 1955.

[2] Peter Rossmanith: *Analysis of algorithms.* http://tcs.rwth-aachen.de/lehre/Analyse/SS2014/script.ps, Summer 2012. Theoretical Computer Science, RWTH Aachen University.

[3] Robert Sedgewick and Philipe Flajolet: *An Introduction to the Analysis of Algorithms.* Addison-Wesley, second edition, 2013.

C

Apéndice C

Breve introducción a asintóticas

Mucho de las matemáticas de pregrado se concentran en hallar cantidades exactas: aprendemos a obtener las raíces exactas de la ecuación, nos enseñan cómo obtener fórmulas (en términos de «funciones elementales») para el valor de las integrales de ciertas clases de funciones, una fórmula cerrada para una suma o la solución de una recurrencia, el número de subconjuntos de un conjunto dado o el número de árboles binarios de n nodos.

Sin embargo, en el mundo real las situaciones en las que hay solución exacta o hay una fórmula son la excepción, no la regla general. Hay muchos problemas en combinatoria, probabilidades, y otras áreas, donde son de interés cantidades (funciones) que nacen naturalmente, pero para las cuales no se conocen fórmulas exactas (simples), y es probable que tales fórmulas no existan. Por ejemplo, hay un resultado notable, el teorema de Risch, que muestra que ciertas integrales no pueden expresarse en términos de funciones elementales, entre las que se cuentan las integrales de $\ln^{-1} x$ y e^{-x^2}, de importancia en teoría de números y probabilidades respectivamente.

Lo que se puede hacer en estos casos es derivar *estimaciones asintóticas* para estas cantidades, aproximaciones «simples» que muestran cómo estas cantidades se comportan asintóticamente (vale decir, cuando un argumento x o n tiende a infinito). En muchas aplicaciones estas estimaciones son tanto o más útiles que valores exactos.

El análisis asintótico, o asintótica para abreviar, se preocupa de hallar estas estimaciones asintóticas. Provee un conjunto de herramientas y técnicas útiles para este propósito. En algunos casos obtener la estimación es rutinario, otros requieren creatividad y métodos *ad hoc*. Buena parte de este apéndice se adapta de un curso corto de Hildebrand [2], una magistral introducción general. Más detalle de aplicaciones combinatorias dan Flajolet y Sedgewick [1], aplicaciones concretas al análisis de algoritmos se hallan en el texto de Sedgewick y Flajolet [3].

C.1. Muestras de aplicaciones

Tal vez la mejor respuesta a las preguntas «¿Qué es asintóticas?» y «¿Para qué sirve?» es dar algunos ejemplos de las muchas áreas en que estos métodos se han aplicado exitosamente. Hildebrand [2] usa estos ejemplos (y otros más) como ilustración de las técnicas más comunes. Desarrollaremos algunos de ellos luego.

C.1.1. Combinatoria: Estimar factoriales

Es fácil ver que $n!$ crece más rápido que a^n para todo a, pero a su vez crece más lento que n^n. La pregunta es cuán rápido crece *realmente*. La respuesta la da la *fórmula de Stirling*, $n! \sim (n/\mathrm{e})^n \sqrt{2\pi n}$. Esta es una de las estimaciones asintóticas más famosas, y la base de muchas otras.

C.1.2. Probabilidades: La paradoja de los cumpleaños

La famosa «paradoja del cumpleaños» dice que la probabilidad de hallar un par de personas con el mismo cumpleaños en un grupo de 23 es más de 50 %. La fórmula para la probabilidad de una coincidencia en un grupo de n personas puede escribirse explícitamente, pero es engorrosa y no ayuda a entender el fenómeno. En particular, ¿porqué es tan pequeño el punto de corte 23 frente al número 356 de días en el año? Técnicas asintóticas dan aproximaciones simples que muestran claramente cómo se comportan las probabilidades en términos del tamaño del grupo y el largo del año, y permiten calcular valores de corte apropiados fácilmente.

C.1.3. Combinatoria/probabilidades: Estimaciones de números harmónicos

Los números harmónicos:

$$H_n = \sum_{1 \le k \le n} \frac{1}{k}$$

aparecen naturalmente en muchos problemas combinatorios, y por tanto son frecuentes en probabilidades y análisis de algoritmos. No hay fórmulas exactas para esta suma, pero métodos asintóticos dan aproximaciones notablemente precisas.

C.1.4. Análisis/probabilidades: La función error

La integral:

$$E(t) = \int_x^\infty \mathrm{e}^{-t^2} \,\mathrm{d}t$$

representa (dentro de un factor constante) la probabilidad de la cola de la distribución normal (se le llama «normal» precisamente porque aparece tan frecuentemente en aplicaciones). No se puede evaluar en forma exacta, análisis asintótico da aproximaciones bastante precisas en términos de funciones elementales.

C.1.5. Teoría de números: La integral logarítmicas

La función:

$$\mathrm{Li}(x) = \int_2^x \frac{\mathrm{d}t}{\ln t}$$

aparece en teoría de números como la «mejor» aproximación a la función $\pi(x)$ que cuenta el número de primos hasta x. La integral no puede expresarse en funciones elementales, aproximaciones asintóticas describen su comportamiento.

C.1.6. Análisis: La función W

Resolver la ecuación:

$$w^w = x$$

define una función $w = W(x)$, que aparece en varios contextos. La ecuación no tiene solución exacta en funciones elementales, métodos asintóticos dan buenas aproximaciones.

C.2. Notaciones asintóticas

Hay una variedad de notaciones asintóticas en uso razonablemente común, definiremos las más importantes.

C.2.1. O mayúscula

Esta es la más común y la más importante. En la mayoría de las aplicaciones interesa el comportamiento de funciones de una variable real o entera que tiende a infinito. Dadas dos funciones $f(x)$ y $g(x)$, definidas para valores suficientemente grandes de x, anotamos:

$$f(x) = O(g(x)) \tag{C.1}$$

si hay constantes x_0 y c tales que:

$$|f(x)| \le c|g(x)| \quad (x \ge x_0) \tag{C.2}$$

En este caso decimos que $f(x)$ *es de orden* $O(g(x))$, y llamamos a (C.2) una *estimación O-mayúscula* (en inglés *big-Oh estimate*). La constante c es *la constante de O*, y el rango $x \ge x_0$ es el *rango de validez* de la estimación. Ocasionalmente se da un rango, en cuyo caso se entiende que la estimación vale en ese rango:

$$|f(x)| \le c|g(x)| \quad (0 \le x \le 1) \tag{C.3}$$

En la mayor parte de las aplicaciones interesan estimaciones válidas para valores grandes de x, o cuando x tiende a infinito. Usamos la convención que si no se indica un rango, se subentiende que vale en algún rango de la forma $x \ge x_0$ para un x_0 apropiado.

Las constantes involucradas en la definición (C.3) generalmente dependerán de parámetros propios de la función. Por ejemplo, para explicitar dependencia del parámetro α, ocasionalmente se anota $f(\alpha, x) = O_\alpha(x)$.

En una ecuación, un término $O(\ldots)$ debe interpretarse como «*en el rango de interés, hay una función que, en valor absoluto, está acotada por una constante veces* $\ldots$». Por ejemplo:

$$\ln(1+x) = x + O(x^2) \quad (|x| \le 1/2)$$

se traduce en: «$\ln(1+x) = x + f(x)$, *donde la función* $f(x)$ *satisface* $|f(x)| \le cx^2$ *para una constante* c *y todo* x *en el rango* $|x| \le 1/2$». Se ve que es una abreviatura útil.

Como O mayúscula da una cota superior, Ω indica una cota inferior:

$$f(x) = \Omega(g(x)) \tag{C.4}$$

si hay constantes x_0 y $c > 0$ tales que:

$$|f(x)| \ge c|g(x)| \quad (x \ge x_0) \tag{C.5}$$

Finalmente, frecuentemente queremos indicar que la misma función da una cota superior e inferior:

$$f(x) = \Theta(g(x)) \tag{C.6}$$

si se cumplen $f(x) = O(g(x))$ y $f(x) = \Omega(g(x))$.

También anotamos:

$$f(x) = o(g(x)) \tag{C.7}$$

si para x suficientemente grande $g(x) = 0$ y tenemos:

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0 \tag{C.8}$$

En este caso decimos que $f(x)$ es *de orden menor* que $g(x)$. Esto es mucho más fuerte de $f(x) = O(g(x))$.

Afín es:

$$f(x) = \omega(g(x)) \tag{C.9}$$

si para todo $c > 0$ hay x_0 tal que:

$$|f(x)| \geq c|g(x)|$$

Alternativamente:

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} \neq 0 \tag{C.10}$$

Otra notación común es:

$$f(x) \sim g(x)$$

que significa que para x suficientemente grande $g(x) \neq 0$ y es:

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = 1$$

Decimos que $f(x)$ y $g(x)$ son *asintóticas* o *asintóticamente equivalentes*. Llamamos *fórmula asintótica* para $f(x)$ a una relación $f(x) \sim g(x)$, donde $g(x)$ es una función «simple».

Algunos ejemplos de la notación O mayúscula, con demostraciones, son:

1. $x = O(\mathrm{e}^x)$
2. $\log(x^2 + 1) = O(\log x)$
3. $\ln(1 + x) = x - x^2/2 + O(x^3)$ para $|x| \leq 1/2$. Más en general, esto vale para $|x| \leq c$ si $c < 1$, con la constante O dependiendo de c.
4. $1/(1 + x) = 1 - x + O(x^2)$ para $|x| \leq 1/2$. Nuevamente, esto vale para $|x| \leq c$ si $c < 1$, con la constante O dependiendo de c.
5. $\ln x = O_\epsilon(x^\epsilon)$

La demostración de tales estimaciones suele ser aplicación directa del cálculo. Ilustraremos las demostraciones de 1 y 3, dando valores explícitos de las constantes.

Para 1, por la definición de una estimación O, debemos demostrar que hay c y x_0 tales que $x \le ce^x$ siempre que $x \ge x_0$. Equivalentemente:

$$\sup_{x \ge x_0} \frac{x}{e^x} \le c$$

Consideremos la función $q(x) = xe^{-x}$. Vemos que:

$$q'(x) = e^{-x}(1-x)$$

Esto es positivo para $x < 1$ y negativo para $x > 1$, o sea $q(x)$ tiene un máximo en $x = 1$. Allí es $q(x) = e^{-1}$, y:

$$\begin{aligned} \frac{x}{e^{-x}} &\le e^{-1} \\ x &\le e^{-1} \cdot e^x \end{aligned}$$

Esto es válido para todo x, podemos tomar arbitrariamente $x_0 = 0$.

Para demostrar 3, partimos de la expansión de Taylor, válida siempre que $|x| < 1$:

$$\ln(1+x) = \sum_{n \ge 1} \frac{(-1)^{n+1} x^n}{n}$$

Bajo el supuesto $|x| \le 1/2$:

$$\begin{aligned} \left| \ln(1+x) - x + \frac{x^2}{2} \right| &\le \sum_{n \ge 3} \frac{|x|^n}{n} \\ &\le \frac{1}{3} \sum_{n \ge 3} |x|^n \\ &= \frac{|x|^3}{3(1-|x|)} \\ &\le \frac{|x|^3}{3(1-1/2)} \\ &= \frac{2}{3}|x|^3 \end{aligned}$$

que es equivalente a:

$$\ln(1+x) = x - \frac{x^2}{2} + O(|x|^3)$$

C.2.2. Trabajando con estimaciones asintóticas

Partimos con algunas estimaciones útiles. Acá α es una constante real arbitraria, y C un número real positivo cualquiera. La constante $1/2$ en los rangos $|z| \le 1/2$ puede reemplazarse por $c < 1$ (en

cuyo caso, la constante de O dependerá de c).

$$\begin{aligned}
\frac{1}{1+z} &\le 1+O(|z|) && (|z|\le 1/2)\\
(1+z)^{\alpha} &\le 1+O_{\alpha}(|z|) && (|z|\le 1/2)\\
(1+z)^{\alpha} &\le 1+\alpha z+O_{\alpha}(|z|^2) && (|z|\le 1/2)\\
\ln(1+z) &\le O(|z|) && (|z|\le 1/2)\\
\ln(1+z) &\le z+O(|z|^2) && (|z|\le 1/2)\\
\ln(1+z) &\le z-\frac{z^2}{2}+O(|z|^3) && (|z|\le 1/2)\\
\mathrm{e}^z &\le 1+O_C(|z|) && (|z|\le C)\\
\mathrm{e}^z &\le 1+z+O_C(|z|^2) && (|z|\le C)\\
\mathrm{e}^z &\le 1+z+\frac{z^2}{2}+O_C(|z|^3) && (|z|\le C)
\end{aligned}$$

C.2.2.1. Propiedades de estimaciones O

Algunas propiedades básicas ayudan al manipular estimaciones. Sus demostraciones son aplicaciones simples de la definición.

Constantes dentro de términos *O*: Si C es una constante positiva, la estimación $f(x)=O(Cg(x))$ es equivalente a $f(x)=O(g(x))$. En particular, la estimación $f(x)=O(C)$ es equivalente a $f(x)=O(1)$. No tiene sentido tener constantes dentro de términos O.

Sumas y productos de términos *O*: Estimaciones O pueden sumarse y multiplicarse en el sentido siguiente: si $f_1(x)=O(g_1(x))$ y $f_2(x)=O(g_2(x))$, entonces $f_1(x)+f_2(x)=O(|g_1(x)|+|g_2(x)|)$ y $f_1(x)\cdot f_2(x)=O(|g_1(x)g_2(x)|)$. Relaciones afines valen para cualquier número *fijo* de términos O.

Diferencias de términos *O*: Si es $f_1(x)=O(g_1(x))$ y $f_2(x)=O(g_2(x))$, entonces la diferencia cumple $f_1(x)-f_2(x)=O(|g_1(x)|+|g_2(x)|)$. Note que al lado derecho tenemos la *suma* (no la diferencia) de los valores absolutos de las funciones cota. En particular, términos O nunca se cancelan.

Sumas de un número arbitrario de términos *O*: La propiedad de suma se extiende a un número arbitrario o infinito de términos O si las estimaciones individuales valen *uniformemente*, vale decir, con los mismos valores de c y x_0. O sea, si para todo n vale $f_n(x)\le g_n(x)$ si $x\ge x_0$, entonces también:

$$\begin{aligned}
\sum_n f_n(x) &= \sum_n O(|g_n(x)|)\\
&= O\left(\sum_n |g_n(x)|\right)
\end{aligned}$$

Distribuyendo términos *O*: Si $f_1(x)=O(g_1(x))$ y $f_2(x)=O(g_2(x))$, para cualquier función $h(x)$ tenemos que $h(x)(f_1(x)+f_2(x))=O(|h(x)|\cdot(|g_1(x)|+|g_2(x)|))$. Esto se extiende en forma obvia a cualquier número *fijo* de términos.

Como ejemplo, demostraremos:

$$\left(1+\frac{1}{x}+O\left(\frac{1}{x^2}\right)\right)^2 = 1+\frac{2}{x}+O\left(\frac{1}{x^2}\right) \tag{C.11}$$

La interpretación correcta de una ecuación como (C.11) es que *toda función estimada por el lado izquierdo también es estimada por el lado derecho.*

Para demostrar (C.11), comenzamos con el lado derecho, y usamos las propiedades repetidas veces:

$$\begin{aligned}\left(1+\frac{1}{x}+O\left(\frac{1}{x^2}\right)\right)^2 &= 1+\frac{1}{x}+O\left(\frac{1}{x^2}\right)+\frac{1}{x}\left(1+\frac{1}{x}+O\left(\frac{1}{x^2}\right)\right)+O\left(\frac{1}{x^2}\right)\left(1+\frac{1}{x}+O\left(\frac{1}{x^2}\right)\right)\\ &= 1+\frac{2}{x}+\left(\frac{1}{x^2}+\frac{1}{x}\cdot O\left(\frac{1}{x^2}\right)\right)+O\left(\frac{1}{x^2}\right)+\frac{1}{x}\cdot O\left(\frac{1}{x^2}\right)+O\left(\frac{1}{x^2}\right)\cdot O\left(\frac{1}{x^2}\right)\\ &= 1+\frac{2}{x}+O\left(\frac{1}{x^2}\right)+O\left(\frac{1}{x^3}\right)+O\left(\frac{1}{x^4}\right)\\ &= 1+\frac{2}{x}+O\left(\frac{1}{x^2}\right)\end{aligned}$$

C.2.2.2. Trucos asintóticos

Concluimos con algunos trucos útiles al manipular expresiones asintóticas.

El truco del logaritmo: Enfrentados a productos de muchos factores o altas potencias, muchas veces ayuda considerar el logaritmo de la expresión. El resultado se manipula siguiendo las propiedades mencionadas arriba, para finalmente exponenciar y simplificar el resultado. Para esto las estimaciones de $\ln(1+z)$ y e^z son útiles.

El truco del factor: Otro truco útil es identificar la contribución dominante en la expresión, extraer este como factor para obtener una expresión de la forma $f(x)(1+e(x))$, donde $f(x)$ es el factor dominante y $e(x)$ es el error relativo. Por ejemplo, al estimar $(x+1/x+1/x^2)^x$, un primer paso natural es factorizar el término x del paréntesis, para obtener $x^x(1+1/x^2+1/x^3)^x$, y tratar el segundo término usando por ejemplo el truco del logaritmo.

El truco del máximo término: Una cota simple a una suma de un número finito de términos es el número de términos veces el máximo término. Esto puede dar estimaciones útiles con poco esfuerzo. Por ejemplo, así tenemos:

$$\sum_{1\le k\le n}\sqrt{\ln k} = O(n\sqrt{\ln n})$$

El truco del término único: La suma de términos no negativos está acotada por abajo por cualquiera de los términos. Esta es otra observación trivial que resulta sorprendentemente efectiva. Por ejemplo:

$$\sum_{1\le k\le n} k^k \ge n^n$$

Puede demostrarse que esta simple cota inferior es cercana a mejor posible, los términos crecen muy rápidamente.

C.2.3. Un caso de estudio: comparar $(1+1/n)^n$ con e

Es sabido que $(1+1/n)^n$ converge a e cuando $n\to\infty$. ¿Qué tan rápido converge? Por ejemplo, ¿converge lo suficientemente rápido para que la diferencia:

$$\epsilon_n = \mathrm{e}-\left(1+\frac{1}{n}\right)^n$$

se acerque tan rápido a 0 como para que converja $\sum \epsilon_n$?

Podemos obtener estimaciones suficientemente precisas de ϵ_n para responder a preguntas como esta. Específicamente:

$$\left(1+\frac{1}{n}\right)^n = \mathrm{e} - \frac{\mathrm{e}}{2n} + O\left(\frac{1}{n^2}\right) \qquad (n \geq 2) \tag{C.12}$$

Como consecuencia de (C.12):

$$\sum_{n\geq 2} \epsilon_n = \sum_{n\geq 2} \left(\frac{\mathrm{e}}{2n} + O\left(\frac{1}{n^2}\right)\right)$$

Como la suma harmónica diverge, diverge esta suma.

Para demostrar (C.12) usamos logaritmos para estimar la expresión del lado izquierdo, y exponenciamos el resultado.

Sea $n \geq 2$. Aplicando la estimación $\ln(1+z) = z - z^2/2 + O(|z|^3)$ para $|z| \leq 1/2$ con $z = 1/n$ obtenemos:

$$\begin{aligned}
\ln\left(1+\frac{1}{n}\right)^n &= n\ln\left(1+\frac{1}{n}\right) \\
&= n\left(\frac{1}{n} - \frac{1}{2n^2} + O\left(\frac{1}{n^3}\right)\right) \\
&= 1 - \frac{1}{2n} + O\left(\frac{1}{n^2}\right)
\end{aligned}$$

Exponenciando:

$$\begin{aligned}
\left(1+\frac{1}{n}\right)^n &= \exp\left(1 - \frac{1}{2n} + O\left(\frac{1}{n^2}\right)\right) \\
&= \mathrm{e}\cdot \mathrm{e}^{-1/(2n)} \cdot \mathrm{e}^{O(1/n^2)}
\end{aligned}$$

Podemos simplificar estas dos exponenciales con las estimaciones $\mathrm{e}^z = 1 + z + O(|z|^2)$ y $\mathrm{e}^z = 1 + O(z)$, respectivamente:

$$\begin{aligned}
\mathrm{e}^{-1/(2n)} &= 1 - \frac{1}{2n} + O\left(\frac{1}{(2n)^2}\right) \\
\mathrm{e}^{O(1/n^2)} &= 1 + O\left(\frac{1}{n^2}\right)
\end{aligned}$$

Substituyendo estas estimaciones y simplificando:

$$\begin{aligned}
\left(1+\frac{1}{n}\right)^n &= \mathrm{e}\left(1 - \frac{1}{2n} + O\left(\frac{1}{(2n)^2}\right)\right)\left(1 + O\left(\frac{1}{n^2}\right)\right) \\
&= \mathrm{e}\left(1 - \frac{1}{2n} + O\left(\frac{1}{n^2}\right)\right)
\end{aligned}$$

Pueden construirse aproximaciones mejores usando más términos en las estimaciones usadas. Note también que tuvimos cuidado de no expandir a términos que finalmente iban a ser absorbidos.

C.3. El problema de los cumpleaños

El problema del cumpleaños trata la probabilidad de que en un grupo de k hayan dos pesonas con el mismo cumpleaños. Resulta que es mucho mayor de lo que uno ingenuamente piensa, para

un grupo de 23 personas ya es mayor a 50 %. Para verlo, la probabilidad que en un grupo de k personas *no* hayan cumpleaños repetidos (suponiendo 365 días en el año, con cumpleaños distribuidos uniformemente) es la que la segunda persona no coincida con la primera $(1-1/365)$, la tercera no coincida con ninguna de las dos anteriores $(1-2/365)$, ..., y finalmente la k-ésima no coincida con ninguna de las $k-1$ anteriores $((1-(k-1)/365)$. O sea, esta probabilidad está dada por el producto:

$$\left(1-\frac{1}{365}\right)\cdot\left(1-\frac{2}{365}\right)\cdot\left(1-\frac{3}{365}\right)\cdots\left(1-\frac{k-1}{365}\right)=\frac{365\cdot 364\cdot\cdots\cdot(365-k+1)}{365^k}$$

(hemos agregado un factor $365/365=1$ para simplificar). Generalizando, si son n días:

$$\begin{aligned}P(n,k)&=\frac{n(n-1)\cdots(n-k+1)}{n^k}\\&=\prod_{0\le i<k}\left(1-\frac{i}{n}\right)\end{aligned}$$

Una generalizacón obvia del problema original es preguntar por el mínimo valor de k tal que $P(n,k)<0{,}5$, $k^*(n)$. ¿Cómo se comporta k^* al crecer n? Desarrollaremos una estimación de $P(n,k)$ que permita responder tales preguntas.

Proposición C.1. *Suponiendo que los cumpleaños se distribuyen uniformente, para $k\le n/2$:*

$$P(n,k)=\exp\left(-\frac{k^2}{2n}+O\left(\frac{k}{n}\right)+O\left(\frac{k^3}{n^2}\right)\right) \tag{C.13}$$

Note que tenemos dos términos O, las variables n y k inciden en forma separada. En particular, para el problema clásico la aproximación (C.13) nos da $\sqrt{2\ln 2\cdot 365}=22{,}494\ldots$, cuando el valor correcto es $k^*(365)=23$.

Demostración. Para $k\le n/2$ tenemos:

$$\begin{aligned}\ln P(n,k)&=\sum_{0\le i<k}\ln\left(1-\frac{i}{n}\right)\\&=\sum_{0\le i<k}\left(-\frac{i}{n}+O\left(\frac{i^2}{n^2}\right)\right)\\&=\frac{1}{n}\cdot\frac{k(k-1)}{2}+O\left(k\cdot\frac{k^2}{n^2}\right)\\&=-\frac{k^2}{2n}+O\left(\frac{k}{n}\right)+O\left(\frac{k^3}{n^2}\right)\end{aligned}$$

Exponenciar da la estimación (C.13). □

Se ve que son manipulaciones bastante rutinarias.

C.4. La integral error

Un ejemplo instructivo es la integral error:

$$E(x)=\int_x^\infty \mathrm{e}^{-t^2}\,\mathrm{d}t$$

Esta representa (dentro de un factor de escala) la probabilidad de la cola de una distribución Gaussiana. Esta función no expresarse en términos de funciones elementales, se requieren métodos numéricos o asintóticos para evaluarla.

Nos interesa $E(x)$ para valores grandes de x. Demostraremos:

Teorema C.1.

$$E(x) = \mathrm{e}^{-x^2}\left(\frac{1}{2x} + O\left(\frac{1}{x^3}\right)\right) \tag{C.14}$$

Demostración. Observe que el integrando e^{-t^2} toma su máximo valor en $t = x$, y después decae muy rápidamente. Esperamos que la mayor contribución al valor de la integral venga de una pequeña vecindad al comienzo del rango. Efectuemos el cambio de variables $t = x + s$ y expresemos el integrando como su valor máximo por un factor:

$$\mathrm{e}^{-t^2} = \mathrm{e}^{-x^2} \cdot \mathrm{e}^{-2sx-s^2}$$

Nuestra integral queda:

$$E(x) = \mathrm{e}^{-x^2}\int_0^\infty \mathrm{e}^{-2sx}\mathrm{e}^{-s^2}\,\mathrm{d}s = \mathrm{e}^{-x^2} I(x)$$

Para deducir la estimación, debemos demostrar que $I(x)$ cumple:

$$I(x) = \frac{1}{2x} + O\left(\frac{1}{x^3}\right) \tag{C.15}$$

Una estimación simple resulta de descartar el factor $\mathrm{e}^{-s^2} \le 1$, que deja:

$$\begin{aligned} I(x) &\le \int_0^\infty \mathrm{e}^{-2sx}\,\mathrm{d}s \\ &= \frac{1}{2x} \end{aligned}$$

Mejorar esta estimación requiere estimar el error cometido al descartar este factor. Para ello dividimos el rango de integración en $s = 1$:

$$I(x) = \int_0^1 \mathrm{e}^{-2sx}\mathrm{e}^{-s^2}\,\mathrm{d}s + \int_1^\infty \mathrm{e}^{-2sx}\mathrm{e}^{-s^2}\,\mathrm{d}s = I_1(x) + I_2(x)$$

Este tipo de división se ve motivado porque el factor e^{-2sx} decae muy rápidamente para x grande, con lo que la contribución vendrá de valores cercanos al límite inferior. Esperamos que $I_1(x)$ dé el término principal de (C.15).

En la integral $I_1(x)$ usamos la estimación $\mathrm{e}^{-s^2} = 1 + O(s^2)$ válida para $0 \le s \le 1$, con lo que:

$$I_1(x) = \int_0^1 \mathrm{e}^{-2sx}\mathrm{e}^{-s^2}\,\mathrm{d}s = \int_0^1 \mathrm{e}^{-2sx}(1 + O(s^2))\,\mathrm{d}s$$

Con el cambio de variables $u = 2sx$ obtenemos:

$$\begin{aligned} I_1(x) &= \frac{1}{2x}\int_0^{2x} \mathrm{e}^{-u}\left(1 + O\left(\frac{u^2}{x^2}\right)\right)\mathrm{d}u \\ &= \frac{1}{2x}\left((1 - \mathrm{e}^{-2x}) + O\left(\frac{1}{x^2}\int_0^\infty u^2 - \mathrm{e}^{-u}\,\mathrm{d}u\right)\right) \\ &= \frac{1}{2x}\left((1 - \mathrm{e}^{-2x}) + O\left(\frac{1}{x^2}\int_0^\infty u^2\mathrm{e}^{-u}\,\mathrm{d}u\right)\right) \\ &= \frac{1}{2x} + O\left(\frac{1}{x^3}\right) \end{aligned}$$

Resta demostrar que $I_2(x)$ es despreciable. Basta una cota superior bastante burda:

$$\begin{aligned} I_2(x) &= \int_1^\infty \mathrm{e}^{-2sx}\mathrm{e}^{-s^2}\,\mathrm{d}s \\ &\le \int_1^\infty \mathrm{e}^{-2sx}\,\mathrm{d}s \\ &= \frac{\mathrm{e}^{-2x}}{2x} \\ &= O\left(\frac{1}{x^2}\right) \end{aligned}$$

Combinando estas estimaciones da lo prometido. □

Ejercicios

1. Demuestre que nuestras notaciones asintóticas pueden resumirse mediante el siguiente cuadro:

Definición	$\boxed{?}\, c > 0$	$\boxed{?}\, x_0$	$\lvert f(x)\rvert \boxed{?}\, c \cdot \lvert g(x)\rvert$
$O()$	$\exists$	$\exists$	$\le$
$o()$	$\forall$	$\exists$	$<$
$\Omega()$	$\exists$	$\exists$	$\ge$
$\omega()$	$\forall$	$\exists$	$>$

2. Demuestre las estimaciones 2, 4 y 5 de la sección C.2.
3. Demuestre las estimaciones asintóticas básicas de la sección C.2.2.
4. Demuestre las propiedades dadas en la sección C.2.2.1.
5. Complete la estimación de $(x+1/x+1/x^2)^x$ hasta $O(x^{-2})$.

Bibliografía

[1] Philipe Flajolet and Robert Sedgewick: *Analytic Combinatorics*. Cambridge University Press, 2009.

[2] Adolf J. Hildebrand: *Short course on asymptotics*. https://faculty.math.illinois.edu/~hildebr/lecture-notes/asymptotics.pdf, July 2015. Mathematics Summer REU Programs, University of Illinois.

[3] Robert Sedgewick and Philipe Flajolet: *An Introduction to the Analysis of Algorithms*. Addison-Wesley, second edition, 2013.

Apéndice D

Rendimiento de programas

Formalizaremos algunas técnicas que hemos usado para evaluar rendimiento de programas. Detalle exhaustivo de la idea da Knuth [1], una discusión más accesible y sistematizada es la de Rossmanith [2].

D.1. Número de veces que se ejecuta cada línea

El primer paso para una evaluación rigurosa de los recursos gastados es determinar cuántas veces se ejecuta cada línea del algoritmo. Contando con costos precisos de cada una de ellas puede calcularse el costo total. Dividimos el programa en bloques básicos (designados por letras mayúsculas, $A, B, \ldots$, secuencias de instrucciones que siempre se ejecutan juntas, y las unimos por arcos dirigidos e_i entre bloques siempre que existe la posibilidad de que el control pase de un bloque a otro. Habrá un bloque en el que comienza el programa, y bloques en los cuales termina. Los conectamos por arcos ficticios. Llamaremos E_i al número de veces que el control pasa por el arco e_i. La técnica se basa en lo siguiente, resultado bastante evidente si se considera que el control nunca se ve atrapado en un ciclo infinito:

Teorema D.1 (Ley de Kirchhoff)**.** *Sea X un bloque básico, y sean $\mathscr{I}$ el conjunto de i tales que el arco e_i entra a X y $\mathscr{O}$ el conjunto de i tales que el arco e_i sale de X. Entonces:*

$$\sum_{i \in \mathscr{I}} E_i = \sum_{i \in \mathscr{O}} E_i$$

La suma es el número de veces que se ejecutan las instrucciones en X.

El primer paso es descomponer en bloques básicos, e identificar los arcos e_i. Luego se halla un árbol recubridor del grafo, obviando que los arcos son dirigidos. Arcos que no pertenecen al árbol recubridor crean ciclos, que llamaremos *ciclos fundamentales*. Describimos los ciclos eligiendo un arco perteneciente al árbol recubridor como positivo y recorremos el ciclo en esa dirección, asignando signos positivo o negativo según si el arco va en la dirección en que seguimos el ciclo o no. Lo interesante es que cada ciclo da una solución a la ley de Kirchhoff, si asignamos $E_i = 0$ a los arcos que no pertenecen al ciclo y $E_i = 1$ a los que pertenecen. Como las ecuaciones son lineales, combinaciones lineales de soluciones nuevamente son soluciones. Podemos expresar los flujos a través del ciclo C_i como λ_i, planteando un sistema de ecuaciones. Esto nos permite expresar todos los flujos E_i en términos de un subconjunto de ellos.

D.1.1. Descomposición en bloques básicos

El primer paso es descomponer el programa en *bloques básicos*, secuencias de instrucciones que se ejecutan siempre en orden. Comienzan en el destino de control de flujo y terminan con alguna instrucción que (condicionalmente o no) cambia el flujo de control. Consideremos por ejemplo el algoritmo de Prim, algoritmo D.1, para un árbol recubridor mínimo de grafo. Datos de entrada son un grafo conexo $G = (V, E)$, una función de peso $w\colon E \to \mathbb{R}$ y un vértice inicial $s \in V$ (arbitrario). Anotaremos $N(v)$ para el conjunto de vértices vecinos a v en G.

Representamos la solución mediante arreglos, el arreglo $\pi[v]$ contendrá el padre en el árbol recubridor mínimo con s de raíz, key$[v]$ es el costo del camino más corto conocido entre el árbol actual y ese vértice. Suponemos la operación min-from(M) que extrae un vértice del conjunto de vértices M con mínimo valor de key$[v]$. Este programa se representa en forma de diagrama de flujo

Algoritmo D.1: Algoritmo de Prim

```
foreach u ∈ V do
    key[u] ← ∞
    π[u] ← nil
end
key[s] ← 0
M ← V
while M ≠ ∅ do
    u ← min-from(M)
    foreach v ∈ N(u) do
        if (v ∈ M) ∧ (w(u, v) < key[v]) then
            π[v] ← u
            key[v] ← w(u, v)
        end
    end
end
```

(grafo de bloques básicos) en la figura D.1. La convención es que flujos «hacia abajo» representan salida con condición al final del bloque verdadera, flujos «hacia el lado» representan condiciones falsas. Note que hemos separado la inicialización de los ciclos de sus cuerpos, se ejecutan aparte. Para representar el elegir un vértice cualquiera estamos usando la operación any-from(M), que elige un elemento cualquiera del conjunto M, lo elimina de este y lo retorna.

D.1.2. Aplicar ley de Kirchhoff

En la figura D.1 hemos marcado los arcos $e_1, e_2, e_4, e_5, e_6, e_7, e_8, e_9$ y e_{10} (estamos identificando $e_0 = e_{15}$, un flujo ficticio a través del «exterior») del árbol recubridor elegido en negrita. Esto da los ciclos fundamentales representados como:

$$C_0 = e_0 + e_1 + e_4 + e_5$$
$$C_1 = e_2 + e_3$$
$$C_2 = e_6 + e_7 + e_{14}$$
$$C_3 = e_8 + e_{13}$$
$$C_4 = e_8 + e_9 + e_{11}$$
$$C_5 = e_8 + e_9 + e_{10} + e_{12}$$

Por la ley de Kirchhoff podemos representar los flujos como:

$$\begin{pmatrix} E_0 \\ E_1 \\ E_2 \\ E_3 \\ E_4 \\ E_5 \\ E_6 \\ E_7 \\ E_8 \\ E_9 \\ E_{10} \\ E_{11} \\ E_{12} \\ E_{13} \\ E_{14} \end{pmatrix} = \lambda_0 \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \lambda_1 \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \lambda_2 \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} + \lambda_3 \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} + \lambda_4 \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \lambda_5 \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} \lambda_0 \\ \lambda_1 \\ \lambda_2 \\ \lambda_3 \\ \lambda_4 \\ \lambda_5 \end{pmatrix}$$

Interesa identificar un conjunto de seis flujos cuyas filas en la matriz sean linealmente independientes, y expresar las demás en términos de ellas. Notamos que $E_0 = E_1 = E_4 = E_5$, $E_2 = E_3$, también $E_6 = E_7 = E_{14}$ y finalmente $E_{10} = E_{12}$ ya que sus filas son iguales. Una mirada al diagrama de flujo D.1 confirma estas igualdades.

Buscamos ahora expresar los demás flujos en términos de un conjunto de flujos linealmente independientes. Tenemos varias igualdades ya, completando con las expresiones para E_8 y E_9 todos

quedan expresados en términos de $E_0, E_2, E_6, E_{10}, E_{11}$ y E_{13}:

$$
\begin{aligned}
E_1 &= E_0 \\
E_3 &= E_2 \\
E_4 &= E_0 \\
E_5 &= E_0 \\
E_7 &= E_6 \\
E_8 &= E_9 + E_{13} \\
E_9 &= E_{10} + E_{11} \\
E_{12} &= E_{10} \\
E_{14} &= E_6
\end{aligned}
$$

Sabemos que el programa se invoca una única vez, por lo que $E_0 = 1$. Los números de veces que se repiten los demás ciclos deberemos determinarlos. Conociendo los flujos, podemos calcular cuántas veces se ejecuta cada vértice (bloque básico). Llamando como el bloque en el diagrama de flujo al número de veces que se le ejecuta resulta:

$$
\begin{aligned}
A &= E_0 \\
B &= E_0 + E_3 \\
&= E_0 + E_2 \\
C &= E_2 \\
D &= E_4 \\
&= E_0 \\
E &= E_5 + E_{14} \\
&= E_0 + E_6 \\
F &= E_6 \\
G &= E_7 + E_{11} + E_{12} + E_{13} \\
&= E_6 + E_{10} + E_{11} + E_{13} \\
H &= E_8 \\
&= E_9 + E_{13} \\
I &= E_9 \\
J &= E_{10}
\end{aligned}
$$

Del programa vemos que:

$$
\begin{aligned}
E_0 &= 1 \\
E_2 &= E_6 = |V|
\end{aligned}
$$

Los demás valores dependen del grafo, del orden en que se eligen nodos y de la función w.

D.1.3. Llamadas a funciones

En el caso que el programa llame a una función, simplemente el costo de esa línea será el costo de esa función para esos argumentos específicos. En caso que sea una llamada recursiva, esto da lugar a una recurrencia. Por ejemplo, considere el programa D.1.

```
void
snapl(int n)
{
        if(n == 0 || n == 2)
                return;
        if(n == 1)
                std::cout << 42 << std::endl;
        else {
                snapl(n - 2);
                snapl(n - 3);
                snapl(n - 2);
                snapl(n - 3);
                snapl(n - 2);
        }
}
```

Listado D.1 – El programa `snapl`

Nos interesa el número a_n de números que escribe la llamada `snapl(n)`.

Resulta la recurrencia:

$$a_n = 3a_{n-2} + 2a_{n-3} \quad a_0 = 0, a_1 = 1, a_2 = 0$$

Técnicas estándar de solución de recurrencias dan:

$$a_n = \frac{2^{n+1} - (-1)^n(3n+2)}{9}$$

D.2. Análisis probabilístico

Analicemos el programa D.2.

```
1  double
2  maximum(const double a[], const int n)
3  {
4          double max;
5
6          max = a[0];
7          for(int i = 1; i < n; i++)
8                  if(a[i] > max)
9                          max = a[i];
10         return max;
11 }
```

Listado D.2 – Hallar el máximo

Es claro que...

Nos interesa específicamente el número de asignaciones a la variable max. Al efecto, supondremos que los elementos son todos distintos, son una permutación elegida uniformemente al azar de n elementos.

El mínimo es 1 (si `a[0]` es el máximo). La probabilidad de este caso es $1/n$ (hay n valores posibles de `a[0]`, todos igualmente probables, solo uno de ellos corresponde al evento de interés).

El máximo es n (si vienen en orden). La probabilidad de este caso es $1/n!$ (hay $n!$ permutaciones, solo una de ellas con los elementos en orden).

Para el promedio, consideremos el caso que para i se ejecuta la línea 9. Podemos describirlo diciendo que elegimos $i+1$ elementos para `a[0]` a `a[i]` entre los n, uno de ellos es el máximo (en `a[i]`). Se permutan los demás, i antes de `a[i]` y $n-i-1$ después. Como en total hay $n!$ permutaciones la probabilidad de este evento es:

$$\begin{aligned}\frac{\binom{n}{i+1} i!(n-i-1)!}{n!} &= \frac{n!}{(i+1)!(n-i-1)!} \cdot \frac{i!(n-i-1)!}{n!} \\ &= \frac{1}{i+1}\end{aligned}$$

Por la linearidad del valor esperado, el número esperado de veces que se ejecuta la línea 9 es:

$$\begin{aligned}\sum_{1\le i<n} \Pr[\text{línea 9 se ejecuta para } i] &= \sum_{1\le i<n} \frac{1}{i+1} \\ &= \sum_{2\le i\le n} \frac{1}{i} \\ &= H_n - 1\end{aligned}$$

El valor esperado del número total de asignaciones a `max` es $H_n \sim \ln n$.

En resumen, las líneas 6 y 10 se ejecutan siempre 1 vez, las líneas 7 y 8 siempre se ejecutan $n-1$ veces, mientras la línea 9 se ejecuta en promedio $H_n - 1$ veces.

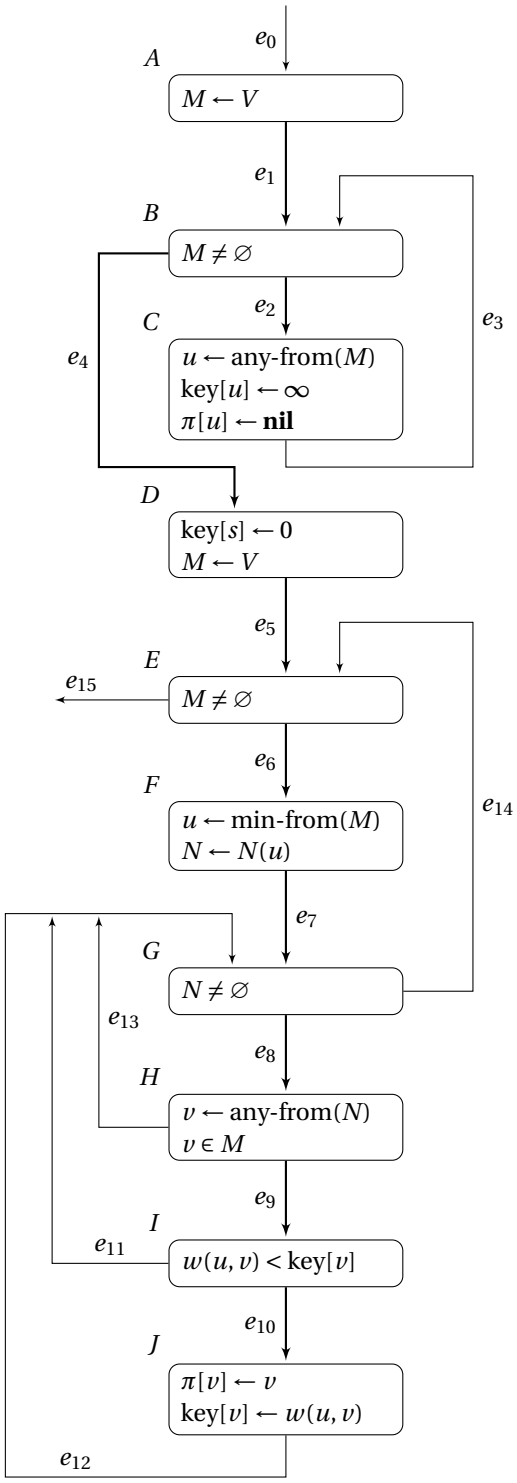


Figura D.1 – Diagrama de flujo del algoritmo de Prim

Bibliografía

[1] Donald E. Knuth: *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, third edition, 1997.

[2] Peter Rossmanith: *Analysis of algorithms*. http://tcs.rwth-aachen.de/lehre/Analyse/SS2014/script.ps, Summer 2012. Theoretical Computer Science, RWTH Aachen University.

Apéndice E

Espacios normados

Nos interesan espacios vectoriales a los cuales les adjuntamos un concepto *producto interno*, y también de *norma*. Partiremos con las definiciones del caso, para luego mostrar algunas consecuencias simples que usamos en el texto.

E.1. Espacio vectorial

Definición E.1. Un *espacio vectorial* V sobre un campo $\mathbb{F}$ (para nosotros casi siempre $\mathbb{R}$, ocasionalmente $\mathbb{C}$) es un conjunto de elementos, $\mathbf{x}, \mathbf{y}, \ldots \in V$ con una operación de *suma* $\mathbf{x}+\mathbf{y}$ que entrega un nuevo vector, y una operación de *multiplicación escalar* que dados $\alpha \in \mathbb{F}$ y $\mathbf{x} \in V$ retorna un vector $\alpha\mathbf{x}$, que cumplen los siguientes axiomas:

1. La suma es *asociativa*: para todo $\mathbf{x}, \mathbf{y}, \mathbf{z} \in V$ es $(\mathbf{x}+\mathbf{y})+\mathbf{z} = \mathbf{x}+(\mathbf{y}+\mathbf{z})$.
2. La suma es *conmutativa*: para todo $\mathbf{x}, \mathbf{y} \in V$ es $\mathbf{x}+\mathbf{y} = \mathbf{y}+\mathbf{x}$.
3. Hay un *elemento neutro* $\mathbf{0} \in V$ tal que para todo $\mathbf{x} \in V$ es $\mathbf{x}+\mathbf{0} = \mathbf{0}+\mathbf{x} = \mathbf{x}$.
4. Para cada $\mathbf{x} \in V$ existe un *inverso aditivo* $-\mathbf{x} \in V$ tal que $\mathbf{x}+(-\mathbf{x}) = \mathbf{0}$.
5. La multiplicación escalar es *compatible*: para todos $\alpha, \beta \in \mathbb{F}$ y todo $\mathbf{x} \in V$ es $\alpha(\beta\mathbf{x}) = (\alpha\beta)\mathbf{x}$.
6. Elemento *identidad* para multiplicación escalar: si $1 \in \mathbb{F}$ es la identidad multiplicativa, entonces para todo $\mathbf{x} \in V$ es $1\mathbf{x} = \mathbf{x}$.
7. La multiplicación escalar distribuye sobre la suma vectorial: para todo $\alpha \in \mathbb{F}$ y todos $\mathbf{x}, \mathbf{y} \in V$ es $\alpha(\mathbf{x}+\mathbf{y}) = \alpha\mathbf{x}+\alpha\mathbf{y}$.
8. La multiplicación escalar distribuye sobre la suma escalar: para todo $\alpha, \beta \in \mathbb{F}$ y todo $\mathbf{x} \in V$ es $(\alpha+\beta)\mathbf{x} = \alpha\mathbf{x}+\beta\mathbf{x}$.

Comúnmente se abrevia $\mathbf{x}+(-\mathbf{y}) = \mathbf{x}-\mathbf{y}$, y omitimos paréntesis dadas la asociatividad y la compatibilidad.

De las anteriores pueden demostrarse consecuencias simples, como las siguientes:

1. El elemento neutro $\mathbf{0} \in V$ es único.

2. Para $\mathbf{x} \in V$, el inverso aditivo $-\mathbf{x}$ es único.

3. Para todo $\mathbf{x} \in V$, $0\mathbf{x} = \mathbf{0}$.

4. Si $\alpha\mathbf{x} = \mathbf{0}$, es $\alpha = 0$ o $\mathbf{x} = \mathbf{0}$.

E.2. Espacios normados

Definición E.2. Sea V un espacio vectorial sobre el campo $\mathbb{F}$ (reales o complejos). Una *norma* en V es una función $\|\cdot\| : V \to \mathbb{R}$ que cumple los axiomas:

1. Homogeneidad: para todo $\alpha \in \mathbb{F}$ y todo $\mathbf{x} \in V$ es $\|\alpha\mathbf{x}\| = |\alpha|\|\mathbf{x}\|$.

2. Desigualdad triangular: para todo $\mathbf{x}, \mathbf{y} \in V$ se cumple $\|\mathbf{x}+\mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$

3. Si $\|\mathbf{x}\| = 0$ entonces $\mathbf{x} = \mathbf{0}$.

De acá se demuestra:

1. Identidad triangular reversa: para todo $\mathbf{x}, \mathbf{y} \in V$ es $|\|\mathbf{x}\| - \|\mathbf{y}\|| \leq \|\mathbf{x}-\mathbf{y}\|$

Intuitivamente, interpretamos $\|\mathbf{x}-\mathbf{y}\|$ como la *distancia* entre $\mathbf{x}$ e $\mathbf{y}$.

E.3. Producto interno

Nos interesan espacios vectoriales sobre el campo escalar $\mathbb{F}$ (reales o complejos) con la operación de *producto interno* entre vectores.

Definición E.3. Sea V un espacio vectorial sobre el campo $\mathbb{F}$ (reales o complejos). Un *producto interno* en V es una operación $\langle\cdot,\cdot\rangle : V \times V \to \mathbb{F}$ que cumple los siguientes axiomas:

1. Simetría conjugada: para todo $\mathbf{x}, \mathbf{y} \in V$ es $\langle \mathbf{x}, \mathbf{y}\rangle = \overline{\langle \mathbf{y}, \mathbf{x}\rangle}$

2. Lineal en el primer argumento: para todo $\mathbf{x}, \mathbf{y}, \mathbf{z} \in V$ y $\alpha, \beta \in \mathbb{F}$ es $\langle \alpha\mathbf{x} + \beta\mathbf{y}, \mathbf{z}\rangle = \alpha\langle \mathbf{x}, \mathbf{z}\rangle + \beta\langle \mathbf{y}, \mathbf{z}\rangle$

3. Positivo definido: para todo $\mathbf{x} \in V$ es $\langle \mathbf{x}, \mathbf{x}\rangle \geq 0$ (por simetría conjugada es siempre real) y $\langle \mathbf{x}, \mathbf{x}\rangle = 0$ si y solo si $\mathbf{x} = 0$.

Consecuencias simples son:

1. $\langle \mathbf{x}, \mathbf{y} + \mathbf{z}\rangle = \langle \mathbf{x}, \mathbf{y}\rangle + \langle \mathbf{x}, \mathbf{z}\rangle$

Vemos que $\|\mathbf{x}\| = \langle \mathbf{x}, \mathbf{x}\rangle$ es una norma en V.
Podemos interpretar $\langle \mathbf{x}, \mathbf{y}\rangle$ como indicativo del «ángulo» entre $\mathbf{x}$ e $\mathbf{y}$, si $\langle \mathbf{x}, \mathbf{y}\rangle = 0$ son ortogonales.
El teorema siguiente relaciona normas con productos internos.

Teorema E.1 (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 E.2 (Desigualdad triangular)**.** $\|\mathbf{x}+\mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$

Demostración.

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

□

E.4. Construir conjunto de vectores ortogonales

Es común querer construir un conjunto de vectores ortogonales dado un conjunto de vectores linealmente independientes. Esto se logra por el proceso de Gram-Schmidt (ver cualquier texto de álgebra lineal, como [1]).

Supongamos entonces un conjunto de vectores linealmente independientes $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \ldots$, y queremos construir un conjunto de vectores ortogonales $\mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3, \ldots$. La construcción se basa en la siguiente observación: sean $\mathbf{x}_1$ y $\mathbf{x}_2$ linealmente independientes, buscamos un par de vectores ortogonales $\mathbf{y}_1$ y $\mathbf{y}_2$ que abarca el espacio definido por $\mathbf{x}_1$ y $\mathbf{x}_2$. Vale decir, $\mathbf{y}_1$ y $\mathbf{y}_2$ se pueden describir como combinaciones lineales de $\mathbf{x}_1$ y $\mathbf{x}_2$. Podemos arbitrariamente elegir $\mathbf{y}_1 = \mathbf{x}_1$, queda por hallar un posible $\mathbf{y}_2$. Por ejemplo, buscamos expresar:

$$\mathbf{y}_2 = \mathbf{x}_2 + \alpha\mathbf{y}_1$$

tal que:

$$\langle \mathbf{y}_1, \mathbf{y}_2 \rangle = 0$$

Vale decir:

$$\begin{aligned} \langle \mathbf{y}_1, \mathbf{y}_2 \rangle &= \langle \mathbf{y}_1, \mathbf{x}_2 + \alpha_2\mathbf{y}_1 \rangle \\ &= \langle \mathbf{y}_1, \mathbf{x}_2 \rangle + \langle \mathbf{y}_1, \alpha_2\mathbf{y}_1 \rangle \\ &= \langle \mathbf{y}_1, \mathbf{x}_2 \rangle + \alpha_2\langle \mathbf{y}_1, \mathbf{y}_1 \rangle \end{aligned}$$

Igualando a cero, despejamos:

$$\alpha_2 = -\frac{\langle \mathbf{y}_1, \mathbf{x}_2 \rangle}{\langle \mathbf{y}_1, \mathbf{y}_1 \rangle}$$

Así:

$$\mathbf{y}_2 = \mathbf{x}_2 - \frac{\langle \mathbf{y}_1, \mathbf{x}_2 \rangle}{\langle \mathbf{y}_1, \mathbf{y}_1 \rangle} \mathbf{y}_1$$

Podemos usar esta misma técnica para eliminar los componentes a lo largo de los anteriores $\mathbf{y}_k$ de los demás vectores. En resumen, calculamos sucesivamente para $i = 1, 2, \ldots$:

$$\mathbf{y}_i = \mathbf{x}_i - \sum_{1 \leq k < i} \frac{\langle \mathbf{y}_k, \mathbf{x}_i \rangle}{\langle \mathbf{y}_k, \mathbf{y}_k \rangle} \mathbf{y}_k$$

Bibliografía

[1] Sergei Treil: *Linear algebra done wrong.* https://www.math.brown.edu/~treil/papers/LADW/book.pdf, September 2017. Department of Mathematics, Brown University.

Apéndice F

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 [3]. Mucho más detalle dan Flajolet y Sedgewick [2], Sedgewick y Flajolet [4] se concentran en aplicaciones al análisis de algoritmos.

La idea es tener una *clase* de objetos, que anotaremos mediante letras caligráficas, como $\mathscr{A}$. La clase $\mathscr{A}$ consta de *objetos*, $\alpha \in \mathscr{A}$. 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 $\mathscr{A}_n$ para referirnos al conjunto de objetos de la clase $\mathscr{A}$ de tamaño n, con lo que $a_n = |\mathscr{A}_n|$. 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:

$$A(z) = \sum_{\alpha \in \mathscr{A}} z^{|\alpha|} = \sum_{n \geq 0} a_n z^n$$

$$\widehat{A}(z) = \sum_{\alpha \in \mathscr{A}} \frac{z^{|\alpha|}}{|\alpha|!} = \sum_{n \geq 0} a_n \frac{z^n}{n!}$$

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; $\mathscr{E} = \{\epsilon\}$, la clase que contiene únicamente el objeto vacío ϵ (de tamaño nulo); y la clase que comúnmente llamaremos $\mathscr{Z}$, conteniendo un único objeto de tamaño uno (que llamaremos ζ por consistencia). Luego definimos operaciones que combinan las clases $\mathscr{A}$ y $\mathscr{B}$ mediante *unión combinatoria* $\mathscr{A} + \mathscr{B}$, en que aparecen los α y los β con sus tamaños (los objetos individuales se «decoran» con su proveniencia, de forma que $\mathscr{A}$ y $\mathscr{B}$ no necesitan ser disjuntos; pero generalmente nos preocuparemos que $\mathscr{A}$ y $\mathscr{B}$ 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 (esta-

mos dejando fuera ciertos elementos, simplemente). Usaremos *producto cartesiano* $\mathscr{A} \times \mathscr{B}$, 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.

F.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 F.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]. □

F.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 $f(u) = u$:

$$\begin{aligned}
[z^n]\, u(z) &= \frac{1}{n}\left[u^{n-1}\right]\phi(u)^n \\
&= \frac{1}{n}\left[u^{n-1}\right](u+1)^{2n} \\
&= \frac{1}{n}\binom{2n}{n-1} \\
&= \frac{1}{n+1}\binom{2n}{n}
\end{aligned}$$

Tenemos, como $u(z) = B(z) - 1$ (sabemos que $[z^0]u(z) = 0$) y de la condición inicial $b_0 = 1$:

$$b_n = \begin{cases} \dfrac{1}{n+1}\dbinom{2n}{n} & \text{si } n \geq 1 \\ 1 & \text{si } n = 0 \end{cases}$$

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 $\mathcal{A}$ 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:

$$\mathcal{A} = \mathcal{Z} \times \textsc{Seq}(\mathcal{A})$$

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 $\mathbb{N}_0 = \textsc{Seq}(\mathcal{Z})$. Para calcular el número de multiconjuntos de k elementos tomados entre n, un multiconjunto queda representado por las cuentas de cada uno 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:

$$4 = 3+1 = 2+2 = 2+1+1 = 1+3 = 1+2+1 = 1+1+2 = 1+1+1+1$$

Sea $c(n)$ el número de combinaciones de n. Como antes, tenemos $\mathbb{N} = \mathcal{Z} \times \textsc{Seq}(\mathcal{Z})$, 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.

F.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 F.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 F.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 $\mathrm{MSET}(\mathscr{A})$, que podemos considerar como una secuencia de elementos numerados obviando el orden. Cuidado, muchos textos le llaman $\mathrm{SET}()$ a esta operación.

Tenemos el siguiente teorema:

Teorema F.2 (Método simbólico, EGF)**.** *Sean $\mathscr{A}$ y $\mathscr{B}$ clases de objetos rotulados, 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 $\mathrm{SEQ}(\mathscr{A})$:*

   $$\frac{1}{1 - \widehat{A}(z)}$$

6. *Para enumerar $\mathrm{MSET}(\mathscr{A})$:*

   $$\mathrm{e}^{\widehat{A}(z)}$$

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.

□

F.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:

$$\text{MSET}(\text{CYC}(\mathcal{Z}))$$

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 ordenados para comenzar con sus elementos máximos.

Podemos describir permutaciones como un conjunto de elementos que quedan fijos combinado con otros elementos que están fuera de orden (un *desarreglo*, clase $\mathscr{D}$). O sea:

$$\begin{aligned}\mathscr{P} &= \textsc{MSet}(\mathscr{Z}) \star \mathscr{D}\\ \frac{1}{1-z} &= \mathrm{e}^z \cdot \widehat{D}(z)\\ \widehat{D}(z) &= \frac{\mathrm{e}^{-z}}{1-z}\end{aligned}$$

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:

$$\mathscr{I} = \textsc{MSet}(\textsc{Cyc}_{\le 2}(\mathscr{Z}))$$

Revisando la derivación, vemos que $\textsc{Cyc}_{\le 2}(\mathscr{Z})$ 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:

$$\mathscr{D} = \textsc{MSet}(Cyc_{>1}(\mathscr{Z}))$$

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 obtuvimos antes.

F.3. Funciones generatrices cumulativas

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:

$$A(z) = \sum_{\alpha\in\mathscr{A}} z^{|\alpha|} \tag{F.1}$$

$$= \sum_{n\geq 0} a_n z^n \tag{F.2}$$

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*:

$$C(z) = \sum_{\alpha\in\mathscr{A}} \chi(\alpha) z^{|\alpha|} \tag{F.3}$$

Vale decir, los coeficientes son la suma de la medida χ para un tamaño dado:

$$[z^n]C(z) = \sum_{|\alpha|=n} \chi(\alpha) \tag{F.4}$$

Así tenemos el valor promedio para objetos de tamaño n:

$$\mathbb{E}_n[\chi] = \frac{[z^n]C(z)}{[z^n]A(z)} \tag{F.5}$$

La discusión precedente es aplicable si tenemos objetos no rotulados entre manos. Si corresponden objetos rotulados, podemos definir las respectivas funciones generatrices exponenciales:

$$\widehat{A}(z) = \sum_{\alpha\in\mathscr{A}} \frac{z^{|\alpha|}}{|\alpha|!} \tag{F.6}$$

$$= \sum_{n\geq 0} a_n \frac{z^n}{n!} \tag{F.7}$$

$$\widehat{C}(z) = \sum_{\alpha\in\mathscr{A}} \chi(\alpha)\frac{z^{|\alpha|}}{|\alpha|!} \tag{F.8}$$

Nuevamente, como los factoriales en los coeficientes se cancelan:

$$\mathbb{E}_n[\chi] = \frac{[z^n]\widehat{C}(z)}{[z^n]\widehat{A}(z)} \tag{F.9}$$

F.4. 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:

$$A(z,u) = \sum_{\alpha\in\mathscr{A}} z^{|\alpha|} u^{\chi(\alpha)} \tag{F.10}$$

De la misma forma, si los átomos son distinguibles es apropiada la función generatriz exponencial:

$$\widehat{A}(z,u) = \sum_{\alpha\in\mathscr{A}} \frac{z^{|\alpha|}}{|\alpha|!} u^{\chi(\alpha)} \tag{F.11}$$

Es común que nos interese el valor promedio de $\chi(\alpha)$ para objetos de tamaño dado. Nótese que:

$$\frac{\partial A}{\partial u} = \sum_{\alpha\in\mathscr{A}} \chi(\alpha) u^{\chi(\alpha)-1} z^{|\alpha|} \tag{F.12}$$

Así podemos calcular los valores promedios a partir de los coeficientes de las siguientes sumas:

$$\sum_{\alpha\in\mathscr{A}} z^{|\alpha|} = A(z,1) \tag{F.13}$$

$$\sum_{\alpha\in\mathscr{A}} \chi(\alpha) z^{|\alpha|} = \left.\frac{\partial A}{\partial u}\right|_{u=1} \tag{F.14}$$

Vemos que (F.13) no es más que la función generatriz del número de objetos, mientras (F.14) es la función generatriz cumulativa.

Bibliografía

[1] Horst H. von Brand: *Fundamentos de Informática*. git://csrg.inf.utfsm.cl/vonbrand/Ramos/trainwreck, Septiembre 2017. Versión 0.84.

[2] Philipe Flajolet and Robert Sedgewick: *Analytic Combinatorics*. Cambridge University Press, 2009.

[3] Jérémie Lumbroso and Basile Morcrette: *A gentle introduction to analytic combinatorics*. http://lipn.univ-paris13.fr/~nicodeme/nablus14/nafiles/gentle.pdf, September 2012.

[4] Robert Sedgewick and Philipe Flajolet: *An Introduction to the Analysis of Algorithms*. Addison-Wesley, second edition, 2013.

Apéndice G

Una pizca de probabilidades

Requeriremos una pizca de probabilidades discretas, ofrecemos un rápido repaso de los resultados principales con sus derivaciones. Incluimos un par de resultados simples que generalmente no se tratan en ramos de probabilidades, pero que son de utilidad para análisis de algoritmos. La discusión general se adapta de Ash [1].

G.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.

G.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:

$$\Omega \in \mathscr{F} \tag{G.1}$$

$$A_1, A_2, \ldots \in \mathscr{F} \text{ implica } \bigcup_n A_n \in \mathscr{F} \tag{G.2}$$

$$A \in \mathscr{F} \text{ implica } \overline{A} \in \mathscr{F} \tag{G.3}$$

O sea, es cerrado respecto de unión finita o infinita numerable (G.2) y complemento (G.3). Por (G.1) y (G.3) concluimos que $\varnothing \in \mathscr{F}$. Por de Morgan, de (G.2) y (G.3) también es cerrado respecto de intersección finita o infinita numerable. O sea, si la pregunta «¿Ocurrió A_i?» tiene respuesta definitiva para $i = 1, 2, \ldots$ también 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:

$$0 \le \Pr[A] \le 1 \tag{G.4}$$

$$\Pr[\Omega] = 1 \tag{G.5}$$

$$\Pr[A \cup B] = \Pr[A] + \Pr[B] \qquad \text{si } A \cap B = \varnothing \tag{G.6}$$

$$\Pr\left[\bigcup_i A_i\right] = \sum_i \Pr[A_i] \qquad \text{para colecciones contables de } A_i \text{ disjuntos} \tag{G.7}$$

$$\tag{G.8}$$

Definición G.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. $\Pr[\varnothing] = 0$

   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 $\Pr[A \cap B] \ge 0$, tenemos la *cota de unión* $\Pr[A \cup B] \le \Pr[A] + \Pr[B]$

3. Si $A \subseteq B$, entonces $\Pr[A] \le \Pr[B]$, específicamente $\Pr[A \setminus B] = \Pr[A] - \Pr[B]$.

4. Para una colección numerable de eventos A_i se cumple $\Pr[\bigcup_i A_i] \le \sum_i \Pr[A_i]$

Las demostraciones de estas aseveraciones quedan de ejercicios.

Se dice que los eventos A y B son *mutuamente exclusivos* si $A \cap B = \varnothing$ (no pueden ocurrir ambos a la vez). El evento $\varnothing$ se dice *imposible* (jamás ocurre).

G.1.2. Probabilidades condicionales

Dados dos eventos A y B con $\Pr[B] > 0$, la *probabilidad condicional* de A dado B se define como:

$$\Pr[A \mid B] = \frac{\Pr[A \cap B]}{\Pr[B]} \tag{G.9}$$

Estamos limitando nuestro universo al evento B.

G.1.3. 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:

$$\mathbb{E}[X] = \sum_x x \Pr[X = x] \tag{G.10}$$

y una medida importante de la dispersión es la *varianza*:

$$\text{var}[X] = \mathbb{E}[(X - \mathbb{E}[X])^2] \tag{G.11}$$

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.

Si se supone que los distintos resultados son igualmente probables se habla de *distribución uniforme*. Así se habla comúnmente de *elegir uniformemente al azar* para indicar que hay un conjunto de posibles resultados, de los que se elige uno al azar siendo igualmente probable que el elemento sea cualquiera del conjunto.

Un resultado extremadamente importante es:

Teorema G.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:

$$\begin{aligned}\mathbb{E}[\alpha X_1 + \beta X_2] &= \sum_x (\alpha x \Pr[X_1 = x] + \beta x \Pr[X_2 = x]) \\ &= \alpha \sum_x x \Pr[X_1 = x] + \beta \sum_x x \Pr[X_2 = x] \\ &= \alpha \mathbb{E}[X_1] + \beta \mathbb{E}[X_2]\end{aligned}$$

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.

G.1.4. Independencia

Decimos que dos variables X e Y son *independientes* si para todo par de valores x e y:

$$\Pr[(X = x) \wedge (Y = y)] = \Pr[X = x] \cdot \Pr[Y = y] \tag{G.12}$$

Una colección de variables es *independiente a pares* si para todo $i \neq j$ y todo par de valores x_i, x_j:

$$\Pr[(X_i = x_i) \wedge (X_j = x_j) = \Pr[X_i = x_i] \cdot \Pr[X_j = x_j] \tag{G.13}$$

Una colección de variables es *mutuamente independiente* si para todo subconjunto $S \subseteq N$ y toda colección de valores x_i:

$$\Pr[(X_{i_1} = x_{i_1}) \wedge (X_{i_2} = x_{i_2}) \wedge \cdots \wedge (X_{i_s} = x_{i_s})] = \Pr[X_{i_1} = x_{i_1}] \cdot \Pr[X_{i_2} = x_{i_2}] \cdots \Pr[X_{i_s} = x_{i_s}] \tag{G.14}$$

Un teorema importante sobre variables independientes es:

Teorema G.2. *Si X_1, X_2 son independientes:*

$$\mathbb{E}[X_1 X_2] = \mathbb{E}[X_1] \cdot \mathbb{E}[X_2]$$
$$\mathbb{E}[f(X_1) f(X_2)] = \mathbb{E}[f(X_1)] \cdot \mathbb{E}[f(X_2)]$$

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. □

De la misma forma obtenemos:

Teorema G.3. *Si X_1 y X_2 son variables aleatorias independientes, entonces:*

$$\text{var}[X_1 + X_2] = \text{var}[X_1] + \text{var}[X_2].$$

Demostración. Abreviamos $\mu_1 = \mathbb{E}[X_1]$ y $\mu_2 = \mathbb{E}[X_2]$, y sabemos $\mathbb{E}[X_1 + X_2] = \mu_1 + \mu_2$:

$$\begin{aligned}\text{var}[X_1 + X_2] &= \mathbb{E}[(X_1 + X_2 - (\mu_1 + \mu_2))^2] \\ &= \mathbb{E}[(X_1 - \mu_1)^2 + 2(X_1 - \mu_1)(X_2 - \mu_2) + (X_2 - \mu_2)^2] \\ &= \mathbb{E}[(X_1 - \mu_1)^2] + 2\mathbb{E}[(X_1 - \mu_1)(X_2 - \mu_2)] + \mathbb{E}[(X_2 - \mu_2)^2] \\ &= \mathbb{E}[(X_1 - \mu_1)^2] + 2\mathbb{E}[X_1 - \mu_1]\mathbb{E}[X_2 - \mu_2] + \mathbb{E}[(X_2 - \mu_2)^2] \\ &= \text{var}[X_1] + \text{var}[X_2]\end{aligned}$$

El término mixto se anula ya que X_1 y X_2 son independientes. □

Un resultado interesante es el siguiente:

Teorema G.4. *Considere un experimento con probabilidad de éxito p. El número de repeticiones independientes hasta tener éxito tiene valor esperado y varianza:*

$$\mathbb{E}[X] = \frac{1}{p}$$
$$\text{var}[X] = \frac{1-p}{p^2}$$

Demostración. La variable X toma valores naturales. Si tiene éxito en la x-ésima repetición quiere decir que falló $x-1$ veces y tuvo éxito una vez. O sea, la probabilidad de este evento es:

$$(1-p)^{x-1} p$$

La función generatriz de probabilidad relevante es:

$$\begin{aligned}G(z) &= \sum_{x \geq 1} (1-p)^{x-1} p z^x \\ &= pz \sum_{x \geq 0} ((1-p)z)^x \\ &= \frac{pz}{1-(1-p)z}\end{aligned}$$

Podemos calcular las estadísticas del caso:

$$\begin{aligned}\mathbb{E}[X] &= G'(1)\\ &= \frac{1}{p}\\ \operatorname{var}[X] &= G''(1) + G'(1) - \left(G'(1)\right)^2\\ &= \frac{1-p}{p^2}\end{aligned}$$

□

G.2. Relaciones elementales

Suele ser útil considerar si las funciones entre manos son *convexas* (o *cóncavas*).

Definición G.2. La función f se dice *convexa* si para $0 \le \alpha \le 1$:

$$\alpha f(x) + (1-\alpha) f(y) \ge f(\alpha x + (1-\alpha) y)$$

Vale decir, la gráfica de f está por debajo de una cuerda que une dos puntos, ver la figura G.1. Por inducción, si $\alpha_i \ge 0$ con $\sum_i \alpha_i = 1$, es:

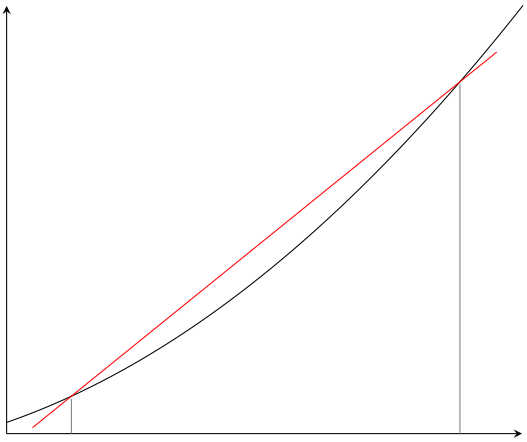


Figura G.1 – Una función convexa

$$\sum_i \alpha_i f(x_i) \ge f\left(\sum_i \alpha_i x_i\right)$$

Aplicando esto a una variable aleatoria finita, con los α_i las probabilidades de los valores de X, obtenemos:

Teorema G.5 (Desigualdad de Jensen)**.** *Si la función f es convexa:*

$$f(\mathbb{E}[X]) \le \mathbb{E}[f(X)] \tag{G.15}$$

G.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 G.6 (Desigualdad de Markov)**.**

$$\Pr[X \geq c] \leq \frac{\mu}{c} \tag{G.16}$$

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.

G.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 G.7 (Desigualdad de Chebyshev)**.**

$$\Pr[|X - \mu| \geq c\sigma] \leq \frac{1}{c^2} \tag{G.17}$$

G.5. Momentos superiores

Hemos definido la varianza de una variable aleatoria X con media μ como:

$$\sigma^2 = \mathbb{E}[(X - \mu)^2]$$

Podemos extender esto a otras potencias, obteniendo los *momentos centrales*:

$$\mu_k = \mathbb{E}[(X-\mu)^k]$$

Note que:

$$\mu_1 = \mathbb{E}[(X-\mu)^1] = 0$$
$$\mu_2 = \mathbb{E}[(X-\mu)^2] = \sigma^2$$

Los momentos centrales dan una medida de cuánto se dispersa la distribución de la media. Mayores k dan mayor importancia a valores más alejados.

Podemos usar el mismo truco que usamos para deducir la desigualdad de Chebyshev para obtener una desigualdad involucrando μ_4.

Sea X una variable aleatoria con media μ y cuarto momento central μ_4. Entonces:

$$\Pr[|X-\mu| \geq c\sqrt[4]{\mu_4}] = \Pr[|X-\mu|^4 \geq c^4\mu_4]$$

Aplicando la desigualdad de Markov a $(X-\mu)^4$, sabiendo que $\mathbb{E}[(X-\mu)^4] = \mu_4$:

$$= \Pr[(X-\mu)^4 \geq c^4 E[(X-\mu)^4]]$$
$$\leq \frac{1}{c^4}$$

Un desarrollo afín es posible siempre que k es par.

G.6. 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 G.8 (Cota superior de Chernoff)**.** *Sean $X_1, \ldots, X_n$ variables aleatorias discretas mutuamente independientes con $0 \leq X_i \leq 1$ para todo i. Sea $X = X_1 + \cdots + X_n$ y sea $\mu = \mathbb{E}[X]$. Entonces para todo $c \geq 1$:*

$$\Pr[X \geq c\mu] \leq \mathrm{e}^{-\beta(c)\mu} \tag{G.18}$$

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 $X \geq c\mu$ y aplicar la desigualdad de Markov:

$$\begin{aligned}
\Pr[X \geq c\mu] &= \Pr[c^X \geq c^{c\mu}] \\
&\leq \frac{\mathbb{E}[c^X]}{c^{c\mu}} && \text{(cota de Markov)} \\
&\leq \frac{e^{(c-1)\mu}}{e^{\mu c \ln c}} && \text{(ver lema G.9)} \\
&= e^{-\beta(c)\mu}
\end{aligned}$$

□

Hemos usado el siguiente resultado, expresado con las mismas variables anteriores:

Lema G.9.

$$\mathbb{E}[c^X] \leq e^{(c-1)\mu}$$

Demostración.

$$\begin{aligned}
\mathbb{E}\left[c^X\right] &= \mathbb{E}\left[c^{X_1+\cdots+X_n}\right] && \text{(definición de } X\text{)} \\
&= \mathbb{E}\left[c^{X_1}\cdots c^{X_n}\right] \\
&= \mathbb{E}\left[c^{X_1}\right]\cdots\mathbb{E}\left[c^{X_n}\right] && \text{(producto de valores independientes)} \\
&\leq e^{(c-1)\mathbb{E}[X_1]}\cdots e^{(c-1)\mathbb{E}[X_n]} && \text{(ver lema G.10 más adelante)} \\
&= e^{(c-1)(\mathbb{E}[X_1]+\cdots+\mathbb{E}[X_n])} \\
&= e^{(c-1)(\mathbb{E}[X_1+\cdots+X_n])} && \text{(linearidad del valor esperado)} \\
&= e^{(c-1)\mathbb{E}[X]}
\end{aligned}$$

□

Finalmente:

Lema G.10.

$$\mathbb{E}\left[c^{X_i}\right] \leq e^{(c-1)\mathbb{E}[X_i]}$$

Demostración. En lo que sigue, las sumas son sobre valores x tomados por la variable X_i. Por la definición de X_i, $x \in [0,1]$.

$$\begin{aligned}
\mathbb{E}\left[c^{X_i}\right] &= \sum_x c^x \Pr[X_i = x] && \text{(definición de valor esperado)} \\
&\leq \sum_x (1+(c-1)x)\Pr[X_i = x] && \text{(convexidad – ver abajo)} \\
&= \sum_x (\Pr[X_i = x] + (1-c)x\Pr[X_i = x]) \\
&= 1 + (c-1)\mathbb{E}[X_i] \\
&\leq e^{(c-1)\mathbb{E}[X_i]} && \text{(porque } 1+z \leq e^z\text{)}
\end{aligned}$$

El segundo paso usa la desigualdad:

$$c^x \leq 1 + (c-1)x$$

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 G.9. □

Ocasionalmente interesa una cota superior, provista por el siguiente teorema.

Teorema G.11. *Con las mismas suposiciones del teorema G.8:*

$$\Pr[X < \mu/c] \leq \mathrm{e}^{-\beta(1/c)\mu} \tag{G.19}$$

Demostración. La demostración es esencialmente igual a la del teorema G.8.

$$\begin{aligned}
\Pr[X < \mu/c] &= \Pr\left[c^{-X} > c^{-\mu/c}\right] \\
&\leq \frac{\mathbb{E}[c^{-X}]}{c^{-\mu/c}} && \text{(cota de Markov)} \\
&\leq \frac{\mathrm{e}^{-(1-1/c)\mu}}{\mathrm{e}^{-\mu \ln c/c}} && \text{(ver lema G.12)}
\end{aligned}$$

El resultado resulta reorganizando esto. □

Tenemos las variantes de los lemas G.9 y G.10:

Lema G.12.

$$\mathbb{E}[c^{-X}] \leq \mathrm{e}^{-(1-1/c)\mu}$$

Demostración.

$$\begin{aligned}
\mathbb{E}\left[c^{-X}\right] &= \mathbb{E}\left[c^{-X_1-\cdots-X_n}\right] && \text{(definición de } X\text{)} \\
&= \mathbb{E}\left[c^{-X_1}\cdots c^{-X_n}\right] \\
&= \mathbb{E}\left[c^{-X_1}\right]\cdots\mathbb{E}\left[c^{-X_n}\right] && \text{(producto de valores independientes)} \\
&\leq \mathrm{e}^{-(1-1/c)\mathbb{E}[X_1]}\cdots\mathrm{e}^{-(1-1/c)\mathbb{E}[X_n]} && \text{(ver lema G.13 más adelante)} \\
&= \mathrm{e}^{-(1-1/c)(\mathbb{E}[X_1]+\cdots+\mathbb{E}[X_n])} \\
&= \mathrm{e}^{-(1-1/c)(\mathbb{E}[X_1+\cdots+X_n])} && \text{(linearidad del valor esperado)} \\
&= \mathrm{e}^{-(1-1/c)\mathbb{E}[X]}
\end{aligned}$$

□

Lema G.13.

$$\mathbb{E}\left[c^{-X_i}\right] \leq \mathrm{e}^{-(1-1/c)\mathbb{E}[X_i]}$$

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 G.12. □

Ejercicios

1. Complete las demostraciones de las propiedades de probabilidades de eventos de la sección G.1.1.

2. Se da a entender en la sección G.5 que la técnica empleada no es aplicable si k es impar. Explique.

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/fall17/mcs.pdf, September 2017. Massachusetts Institute of Technology.