

9

Técnicas de análisis de algoritmos

¿Qué es un buen algoritmo? No es fácil responder a esta pregunta. Muchos criterios para un buen algoritmo incluyen cuestiones muy subjetivas como simplicidad, claridad y adecuación a los datos manejados. Una meta más objetiva, lo cual no significa que sea más importante, es la eficiencia en tiempo de ejecución. En la sección 1.5 se abordaron las técnicas básicas para establecer el tiempo de ejecución de programas simples. Sin embargo, en casos más complejos se requieren técnicas nuevas, como cuando los programas son recursivos. Este corto capítulo presenta algunas técnicas generales para resolver ecuaciones de recurrencia que se presentan en el análisis de los tiempos de ejecución de algoritmos recursivos.

9.1 Eficiencia de los algoritmos

Una forma de determinar la eficiencia del tiempo de ejecución de un algoritmo es programarlo y medir el tiempo que lleva la versión en particular, en un computador específico, para un conjunto seleccionado de entradas. Aunque es popular y útil, este enfoque tiene algunos problemas inherentes. Los tiempos de ejecución dependen no sólo del algoritmo base, sino también del conjunto de instrucciones del computador, de la calidad del compilador y de la destreza del programador. El programa también puede adaptarse para trabajar correctamente sobre el conjunto particular de entradas de prueba. Estas dependencias pueden ser muy notorias con un computador, un compilador, un programador o un conjunto de entradas de prueba distinto. Para superar esos inconvenientes, los expertos en computación han adoptado la complejidad de tiempo asintótica como una medida fundamental del rendimiento de un algoritmo. El término *eficiencia* se referirá a esta medida y, en especial, a la complejidad de tiempo en el peor caso (en contraposición al promedio).

Recuérdense del capítulo 1 las definiciones de $O(f(n))$ y $\Omega(f(n))$. La eficiencia o complejidad en el peor caso de un algoritmo es $O(f(n))$, o $f(n)$ para abreviar, si $O(f(n))$ es la función de n que da el máximo, en todas las entradas de longitud n , del número de pasos dados por el algoritmo en esas entradas. En otras palabras, existe alguna constante c tal que para una n suficientemente grande, $cf(n)$ es una cota superior del número de pasos dados por el algoritmo con cualquier entrada de longitud n .

En la aseveración de que «la eficiencia de un algoritmo dado es $f(n)$ », existe la implicación de que la eficiencia también es $\Omega(f(n))$, de forma que $f(n)$ es la función

de crecimiento más lento de n que acota el tiempo de ejecución en el peor caso por arriba. Sin embargo, este último requisito no es parte de la definición de $O(f(n))$, y algunas veces no es posible asegurar que se tenga la cota de crecimiento superior más lenta.

Esta definición de eficiencia ignora factores constantes en el tiempo de ejecución y existen varias razones prácticas para ello. Primero, dado que la mayoría de los algoritmos están escritos en lenguajes de alto nivel, se deben describir en función de «pasos», los cuales emplean una cantidad constante de tiempo cuando se traducen al lenguaje de máquina de cualquier computador. Sin embargo, exactamente cuánto tiempo requiere un paso depende no sólo del paso mismo, sino del proceso de traducción y del conjunto de instrucciones de la máquina. Así, intentar tener más precisión que para decir que el tiempo de ejecución de un algoritmo es «del orden de $f(n)$ », esto es, $O(f(n))$, podría empantanar al usuario en detalles de máquinas específicas y sólo sería aplicable a esas máquinas.

Una segunda razón importante para tratar con la complejidad asintótica e ignorar factores constantes es que, más que estos factores, es la complejidad asintótica lo que determina para qué tamaño de entradas puede usarse el algoritmo para producir soluciones en un computador. En el capítulo 1 se analizó con detalle este aspecto. Sin embargo, es necesario tener cautela acerca de la posibilidad de que para problemas muy importantes, como los de clasificación, se pueda considerar adecuado analizar los algoritmos con tal detalle que sean factibles ciertas proposiciones como «el algoritmo A debe ejecutarse con el doble de rapidez que el algoritmo B en un computador típico».

Una segunda situación en la cual conviene desviarse de la noción de eficiencia en el peor caso ocurre cuando se conoce la distribución esperada de las entradas a un algoritmo. En estas situaciones, el análisis del caso promedio puede ser mucho más significativo que el análisis del peor caso. Por ejemplo, en el capítulo previo se analizó el tiempo de ejecución promedio de la clasificación rápida en el supuesto de que todas las permutaciones del orden de clasificación correcto tienen la misma probabilidad de presentarse como entradas.

9.2 Análisis de programas recursivos

En el capítulo 1 se mostró cómo analizar el tiempo de ejecución de un programa que no se llama a sí mismo en forma recursiva. El análisis de un programa recursivo es bastante distinto, y suele implicar la solución de una ecuación de diferencias. Las técnicas para la solución de ecuaciones de diferencias algunas veces son sutiles, y tienen una considerable semejanza con los métodos de solución de ecuaciones diferenciales, alguna de cuya terminología se utiliza.

Considérese el programa de clasificación presentado en la figura 9.1. Ahí, el procedimiento *mergesort* (clasificación por intercalación) toma una lista de longitud n como entrada, y devuelve una lista ordenada como salida. El procedimiento *combi-na*(L_1, L_2) toma como entrada las listas clasificadas L_1 y L_2 , y recorre cada una, elemento por elemento, desde el inicio. En cada paso, el mayor de los dos elementos

delanteros se borra de esta lista y se emite como salida. El resultado es una sola lista clasificada con los elementos de L_1 y L_2 . Los detalles de *combina* no tienen importancia alguna en este momento, puesto que este algoritmo de clasificación se analizará con detalle en el capítulo 11. Lo que importa es que el tiempo empleado por *combina* en listas de longitud $n/2$ es $O(n)$.

```

function mergesort (  $L$ : LISTA;  $n$ : integer ) : LISTA;
{  $L$  es una lista de longitud  $n$ . Se devuelve una versión clasificada de  $L$ .
  Se supone que  $n$  es una potencia de 2. }
var
   $L_1, L_2$ : LIST
begin
  if  $n = 1$  then
    return ( $L$ )
  else begin
    partir  $L$  en dos mitades,  $L_1$  y  $L_2$ , cada una de longitud  $n/2$ ;
    return (combina(mergesort( $L_1, n/2$ ), mergesort( $L_2, n/2$ )))
  end
end; { mergesort }
```

Fig. 9.1. Procedimiento recursivo *mergesort*.

Sea $T(n)$ el tiempo de ejecución en el peor caso del procedimiento *mergesort* de la figura 9.1. Se escribe una ecuación de *recurrencia* (o *diferencias*) que acote $T(n)$ por arriba, como sigue

$$T(n) \leq \begin{cases} c_1 & \text{si } n = 1 \\ 2T(n/2) + c_2n & \text{si } n > 1 \end{cases} \quad (9.1)$$

El término c_1 en (9.1) representa el número constante de pasos dados cuando L tiene longitud 1. En el caso de que $n > 1$, el tiempo requerido por *mergesort* puede dividirse en dos partes. Las llamadas recursivas a *mergesort* con listas de longitud $n/2$ cada una llevan un tiempo $T(n/2)$, de aquí el término $2T(n/2)$. La segunda parte consiste en la prueba para descubrir que $n \neq 1$, la división de la lista L en dos partes iguales y el procedimiento *combina*. Esas tres operaciones requieren un tiempo que puede ser una constante, en el caso de la prueba, o ser proporcional a n al dividir y combinar. Así, la constante c_2 puede escogerse de modo que el término c_2n sea una cota superior del tiempo requerido por *mergesort* para hacer todo excepto las llamadas recursivas. La ecuación (9.1) representa todo lo anterior.

Obsérvese que (9.1) se aplica sólo cuando n es par, por lo que generará una cota superior en forma cerrada (esto es, como una fórmula para $T(n)$ que no implica ningún $T(m)$ para $m < n$) sólo cuando n es una potencia de 2. Sin embargo, aunque sólo se conozca $T(n)$ cuando n es una potencia de 2, se tiene una buena idea de $T(n)$ para toda n . En particular, para casi todos los algoritmos, se puede suponer que $T(n)$ está entre $T(2^i)$ y $T(2^{i+1})$ si n está entre 2^i y 2^{i+1} . Y con un poco más de esfuerzo

para encontrar la solución, se puede reemplazar el término $2T(n/2)$ de (9.1) por $T((n+1)/2) + T((n-1)/2)$ para $n > 1$ impar. Después, se podría resolver la nueva ecuación de diferencia para obtener una solución cerrada para toda n .

9.3 Resolución de ecuaciones de recurrencia

Existen tres enfoques distintos para resolver una ecuación de recurrencia.

1. Suponer una solución $f(n)$ y usar la recurrencia para mostrar que $T(n) \leq f(n)$. Algunas veces sólo se supone la forma de $f(n)$, dejando algunos parámetros sin especificar (por ejemplo, supóngase $f(n) = an^2$ para alguna a) y deduciendo valores adecuados para los parámetros al intentar demostrar que $T(n) \leq f(n)$ para toda n .
2. Utilizar la recurrencia misma para sustituir $m < n$ por cualquier $T(m)$ en la derecha, hasta que todos los términos $T(m)$ para $m > 1$ se hayan reemplazado por fórmulas que impliquen sólo $T(1)$. Como $T(1)$ siempre es constante, se tiene una fórmula para $T(n)$ en función de n y de algunas constantes. A esta fórmula se ha denominado «forma cerrada» para $T(n)$.
3. Emplear la solución general para ciertas ecuaciones de recurrencia de tipos comunes de esta sección o de otra (véanse las notas bibliográficas).

En esta sección se examinan los dos primeros métodos.

Suposición de una solución

Ejemplo 9.1. Considérese el método (1) aplicado a la ecuación (9.1). Supóngase que para alguna a , $T(n) = an\log n$. Al sustituir $n = 1$, se observa que esta suposición no funcionará, porque $an\log n$ tiene valor 0, independiente del valor de a . Así, se intenta a continuación $T(n) = an\log n + b$. Ahora, $n = 1$ requiere que $b \geq c_1$.

Para la inducción, se supone que

$$T(k) \leq ak\log k + b \quad (9.2)$$

para toda $k < n$ y se intenta establecer que

$$T(n) \leq an\log n + b$$

Para iniciar la demostración, se supone que $n \geq 2$. De (9.1),

$$T(n) \leq 2T(n/2) + c_2n$$

De (9.2), con $k = n/2$, se obtiene

$$T(n) \leq 2[a\frac{n}{2}\log\frac{n}{2} + b] + c_2n \quad (9.3)$$

$$\begin{aligned} &\leq an \log n - an + c_2 n + 2b \\ &\leq an \log n + b \end{aligned}$$

siempre que $a \geq c_2 + b$.

Así, $T(n) \leq an \log n + b$ siempre y cuando se satisfagan dos restricciones: $b \geq c_1$ y $a \geq c_2 + b$. Por fortuna, existen valores que se pueden escoger para a que satisfacen ambas restricciones. Por ejemplo, al elegir $b = c_1$ y $a = c_1 + c_2$, por inducción sobre n , se concluye que para toda $n \geq 1$

$$T(n) \leq (c_1 + c_2)n \log n + c_1 \quad (9.4)$$

En otras palabras, $T(n)$ es $O(n \log n)$. \square

Son útiles dos observaciones acerca del ejemplo 9.1. Si se supone que $T(n)$ es $O(f(n))$, y si el intento de probar que $T(n) \leq cf(n)$ por inducción falla, se sigue que $T(n)$ no sea $O(f(n))$. De hecho, puede funcionar una hipótesis inductiva de la forma $T(n) \leq cf(n) - 1$.

En segundo lugar, aún no se ha determinado la tasa exacta de crecimiento asintótica para $f(n)$, aunque se ha demostrado que no es peor que $O(n \log n)$. Si se conjectura una solución más lenta, como $f(n) = an$, o $f(n) = an \log \log n$, no se puede demostrar la validez de $T(n) \leq f(n)$. La cuestión sólo se puede establecer concluyentemente examinando mergesort y demostrando que necesita realmente un tiempo $\Omega(n \log n)$; de hecho, requiere un tiempo proporcional a $n \log n$ en todas las entradas, no sólo en las peores entradas posibles. Se deja esta observación como ejercicio.

El ejemplo 9.1 expone una técnica general para demostrar que alguna función es una cota superior en el tiempo de ejecución de un algoritmo. Supóngase la ecuación de recurrencia

$$\begin{aligned} T(1) &= c \\ T(n) &\leq g(T(n/2), n), \text{ para } n > 1 \end{aligned} \quad (9.5)$$

Obsérvese que (9.5) generaliza (9.1), donde $g(x, y)$ es $2x + c_2y$. También obsérvese que podrían imaginarse ecuaciones más generales que (9.5). Por ejemplo, la fórmula g puede comprender todos los $T(n-1)$, $T(n-2)$, ..., $T(1)$, y no sólo $T(n/2)$. También, es posible tener valores para $T(1)$, $T(2)$, ..., $T(k)$, y la recurrencia sólo sería aplicable para $n > k$. A manera de ejercicio, sería interesante considerar cómo resolver estas recurrencias más generales por el método (1), suponiendo una solución y verificándola.

Considérese ahora (9.5), en vez de sus generalizaciones. Supóngase una función $f(a_1, \dots, a_j, n)$, donde a_1, \dots, a_j son parámetros, e inténtese demostrar por inducción en n que $T(n) \leq f(a_1, \dots, a_j, n)$. Por ejemplo, la suposición del ejemplo 9.1 fue $f(a_1, a_2, n) = a_1 n \log n + a_2$, pero se utilizaron a y b por a_1 y a_2 . Para corroborar que para algunos valores de a_1, \dots, a_j se tiene $T(n) \leq f(a_1, \dots, a_j, n)$ para toda $n \geq 1$, se debe cumplir

$$\begin{aligned} f(a_1, \dots, a_j, 1) &\geq c \\ f(a_1, \dots, a_j, n) &\geq g(f(a_1, \dots, a_j, n/2), n) \end{aligned} \quad (9.6)$$

Esto es, por la hipótesis inductiva, se puede sustituir f por T en el lado derecho de la recurrencia (9.5) para obtener

$$T(n) \leq g(f(a_1, \dots, a_j, n/2), n) \quad (9.7)$$

Cuando se cumple la segunda línea de (9.6), se combina con (9.7) para demostrar que $T(n) \leq f(a_1, \dots, a_n, n)$, que es lo que se deseaba demostrar por inducción en n .

Por ejemplo, en el ejemplo 9.1, $g(x, y) = 2x + c_2y$ y $f(a_1, a_2, n) = a_1 n \log n + a_2$. Aquí se intenta satisfacer

$$\begin{aligned} f(a_1, a_2, 1) &= a_2 \geq c_1 \\ f(a_1, a_2, n) &= a_1 n \log n + a_2 \geq 2(a_1 \left(\frac{n}{2}\right) \log \left(\frac{n}{2}\right) + a_2) + c_2 n \end{aligned}$$

Como se analizó, $a_2 = c_1$ y $a_1 = c_1 + c_2$ son una elección satisfactoria.

Expansión de recurrencias

Si no se puede suponer una solución, o no existe la seguridad de tener la mejor cota en $T(n)$, se usa un método que, en principio, siempre es adecuado para la solución exacta de $T(n)$, aunque en la práctica a menudo surgen problemas al sumar series y hay que recurrir al cálculo de una cota superior en la suma. La idea general es tomar una recurrencia como (9.1), que indica que $T(n) \leq 2T(n/2) + c_2n$, y emplearla para obtener una cota en $T(n/2)$ sustituyendo $n/2$ por n . Esto es,

$$T(n/2) \leq 2T(n/4) + c_2n/2 \quad (9.8)$$

Al sustituir el lado derecho de (9.8) por $T(n/2)$ en (9.1), se obtiene

$$T(n) \leq 2(2T(n/4) + c_2n/2) + c_2n = 4T(n/4) + 2c_2n \quad (9.9)$$

Del mismo modo, se sustituye $n/4$ por n en (9.1) y se emplea para obtener una cota superior en $T(n/4)$. Esa cota, $2T(n/8) + c_2n/4$, se sustituye en el lado derecho de (9.9) para obtener

$$T(n) \leq 8T(n/8) + 3c_2n \quad (9.10)$$

Ahora es necesario percibir un patrón. Por inducción en i se obtiene la relación

$$T(n) \leq 2^i T(n/2^i) + i c_2 n \quad (9.11)$$

para cualquier i . Si se supone que n es una potencia de 2, como 2^k , este proceso de expansión terminará tan pronto como se alcance $T(1)$ en el lado derecho de (9.11), lo que ocurrirá cuando $i = k$, quedando (9.11) como

$$T(n) \leq 2^k T(1) + k c_2 n \quad (9.12)$$

Entonces, como $2^k = n$, se sabe que $k = \log n$. Como $T(1) \leq c_1$, (9.12) es

$$T(n) \leq c_1 n + c_2 n \log n \quad (9.13)$$

La ecuación (9.13) es en realidad la mejor cota que se pudo colocar en $T(n)$, y demuestra que $T(n)$ es $O(n \log n)$.

9.4 Solución general para una clase grande de recurrencias

Considérese la recurrencia que surge al dividir un problema de tamaño n en a subproblemas de tamaño n/b . Por conveniencia, se supone que un problema de tamaño 1 requiere una unidad de tiempo y que el tiempo para reunir las soluciones de los subproblemas y obtener una solución del problema de tamaño n es $d(n)$, en las mismas unidades de tiempo. Para el ejemplo de *mergesort*, se tienen $a = b = 2$ y $d(n) = -c_2 n / c_1$, en unidades de c_1 . Entonces, si $T(n)$ es el tiempo para resolver un problema de tamaño n , se tiene

$$\begin{aligned} T(1) &= 1 \\ T(n) &= aT(n/b) + d(n) \end{aligned} \quad (9.14)$$

Obsérvese que (9.14) sólo se aplica a las n que sean una potencia entera de b , pero si se presume que $T(n)$ es continua, al tomar una cota superior ajustada sobre $T(n)$ para aquellos valores de n , nos indica cómo crece $T(n)$ en general.

Obsérvese también que se utiliza la igualdad en (9.14), mientras que en (9.1) hay desigualdades. La razón es que aquí $d(n)$ puede ser arbitraria y, por tanto, exacta, mientras que en (9.1) la suposición de que $c_2 n$ fue el peor caso en tiempo de combinación, para una constante c_2 y toda n , fue sólo una cota superior; el peor caso real del tiempo de ejecución con entradas de tamaño n pudo haber sido menor que $2T(n/2) + c_2 n$. En realidad, hay muy poca diferencia entre utilizar $=$ o \leq en la recurrencia, ya que de cualquier modo se obtiene una cota superior para el peor caso del tiempo de ejecución.

Para resolver (9.14) se aplica la técnica de sustituciones repetidas para T en el lado derecho, igual que se hizo para un ejemplo específico en la exposición anterior de la expansión de recurrencias. Esto es, la sustitución n/b^i por n en la segunda línea de (9.14) da

$$T\left(\frac{n}{b^i}\right) = aT\left(\frac{n}{b^{i+1}}\right) + d\left(\frac{n}{b^i}\right) \quad (9.15)$$

Así, al empezar con (9.14) y sustituir (9.15) para $i = 1, 2, \dots$, se tiene

$$\begin{aligned} T(n) &= aT\left(\frac{n}{b}\right) + d(n) \\ &= a[aT\left(\frac{n}{b^2}\right) + d\left(\frac{n}{b}\right)] + d(n) = a^2T\left(\frac{n}{b^2}\right) + ad\left(\frac{n}{b}\right) + d(n) \end{aligned}$$

$$\begin{aligned}
 &= a^2[aT\left(\frac{n}{b^3}\right) + d\left(\frac{n}{b^2}\right)] + ad\left(\frac{n}{b}\right) + d(n) = a^3T\left(\frac{n}{b^3}\right) + a^2d\left(\frac{n}{b^2}\right) + ad\left(\frac{n}{b}\right) + d(n) \\
 &= \dots \\
 &= a^iT\left(\frac{n}{b^i}\right) + \sum_{j=0}^{i-1} a^j d\left(\frac{n}{b^j}\right)
 \end{aligned}$$

Ahora, suponiendo que $n = b^k$, se puede utilizar el hecho de que $T(n/b^k) = T(1) = 1$, para tener de lo anterior, con $i = k$, la fórmula

$$T(n) = a^k + \sum_{j=0}^{k-1} a^j d(b^{k-j}) \quad (9.16)$$

Si se aplica el hecho de que $k = \log_b n$, el primer término de (9.16) puede escribirse como $a^{\log_b n}$ o, de forma equivalente, $n^{\log_b a}$ (se toman logaritmos de base b de ambas expresiones para ver que son lo mismo). Esta expresión es n a una potencia constante. Por ejemplo, en el caso de *mergesort*, donde $a = b = 2$, el primer término es n . En general, cuanto mayor sea a , es decir, cuantos más subproblemas haya que resolver, tanto mayor será el exponente; cuanto mayor sea b , es decir, cuanto menor sea cada subproblema, tanto menor será el exponente.

Soluciones homogéneas y particulares

Es interesante ver los diferentes papeles que desempeñan los dos términos de (9.16). El primero, a^k o $n^{\log_b a}$, se conoce como *solución homogénea*, en analogía con la terminología de las ecuaciones diferenciales. La solución homogénea es la solución exacta cuando $d(n)$, conocida como *función motriz*, es 0 para toda n . En otras palabras, la solución homogénea representa el costo de resolver todos los subproblemas, aunque puedan combinarse sin costo.

Por otra parte, el segundo término de (9.16) comprende el costo de creación de los subproblemas y la combinación de sus resultados. Este término se denomina *solución particular*, que está afectada tanto por la función motriz como por el número y tamaño de los subproblemas. Como regla general, si la solución homogénea es mayor que la función motriz, la solución particular tendrá la misma tasa de crecimiento que la solución homogénea. Si la función motriz crece más rápido que la solución homogénea en más que n^ϵ para alguna $\epsilon > 0$, la solución particular tendrá la misma tasa de crecimiento que la función motriz. Si la función motriz tiene la misma tasa de crecimiento que la solución homogénea, o crece más rápido que $\log^k n$ como mucho para alguna k , entonces la solución particular crecerá $\log n$ veces la función motriz.

Es importante reconocer que cuando se buscan mejoras en un algoritmo es necesario saber si la solución homogénea es mayor que la función motriz. Por ejemplo, si la solución homogénea es mayor, prácticamente no tendrá efecto encontrar una forma más rápida de combinar los subproblemas en la eficiencia de todo el al-

goritmo. Lo mejor, en este caso, es encontrar una forma de dividir el problema en menos o menores subproblemas. Eso afectará a la solución homogénea y reducirá el tiempo total de ejecución.

Si la función motriz excede a la solución homogénea, entonces es necesario tratar de reducir la función motriz. Por ejemplo, en el caso de *mergesort*, donde $a = b = 2$, y $d(n) = cn$, la solución particular es $O(n\log n)$. Sin embargo, reducir $d(n)$ a una función ligeramente sublineal, por ejemplo, $n^{0.9}$, hará que la solución particular sea también menos que lineal y que se reduzca el tiempo de ejecución total a $O(n)$, que es la solución homogénea †.

Funciones multiplicativas

La solución particular de (9.16) es difícil de evaluar, aun sabiendo lo que es $d(n)$. Sin embargo, para ciertas funciones $d(n)$ comunes, se puede resolver (9.16) con exactitud, y hay otras para las cuales se puede conseguir una buena cota superior. Se dice que una función f en enteros es *multiplicativa* si $f(xy) = f(x)f(y)$ para todos los enteros positivos x e y .

Ejemplo 9.2. Las funciones multiplicativas de mayor interés son de la forma n^a para cualquier a positiva. Para demostrar que $f(n) = n^a$ es multiplicativa, sólo hay que observar que $(xy)^a = x^a y^a$. □

Ahora, si $d(n)$ de (9.16) es multiplicativa, entonces $d(b^{k-j}) = (d(b))^{k-j}$, y la solución particular de (9.16) es

$$\begin{aligned} \sum_{j=0}^{k-1} a^j (d(b))^{k-j} &= d(b)^k \sum_{j=0}^{k-1} \left(\frac{a}{d(b)} \right)^j \\ &= d(b)^k \frac{\left(\frac{a}{d(b)} \right)^k - 1}{\frac{a}{d(b)} - 1} \\ &= \frac{a^k - d(b)^k}{\frac{a}{d(b)} - 1} \end{aligned} \tag{9.17}$$

Hay tres casos a considerar, dependiendo de si a es mayor, menor o igual que $d(b)$.

† Pero no debe esperarse descubrir una manera de combinar dos listas clasificadas de $n/2$ elementos en un tiempo menor que el lineal; en ese caso, no sería posible siquiera ver todos los elementos de la lista.

- Si $a > d(b)$, la fórmula (9.17) es $O(a^k)$, que como se recuerda es $n^{\log_b a}$, ya que $k = \log_b n$. En este caso, las soluciones particular y homogénea son iguales, y sólo dependen de a y b , y no de la función motriz d . Así, las mejoras en el tiempo de ejecución deben proceder de disminuir a o aumentar b ; la disminución de $d(n)$ no es muy útil.
- Si $a < d(b)$, (9.17) es $O(d(b)^k)$ o, en forma equivalente, $O(n^{\log_b d(b)})$. En este caso, la solución particular excede a la homogénea, y se puede dirigir la atención también hacia la función motriz $d(n)$, además de a y b , para obtener mejoras. Obsérvese el importante caso especial en que $d(n) = n^a$. Entonces, $d(b) = b^a$, y $\log_b(b^a) = a$. Así, la solución particular es $O(n^a)$ o bien $O(d(n))$.
- Si $a = d(b)$, se reconsideran los cálculos implicados en (9.17), pues la fórmula para la suma de una serie geométrica no es ahora apropiada. En este caso,

$$\begin{aligned} \sum_{j=0}^{k-1} a^j (d(b))^{k-j} &= d(b)^k \sum_{j=0}^{k-1} \left(\frac{a}{d(b)} \right)^j \\ &= d(b)^k \sum_{j=0}^{k-1} 1 \\ &= d(b)^k k \\ &= n^{\log_b d(b)} \log_b n \end{aligned} \quad (9.18)$$

Como $a = d(b)$, la solución particular dada por (9.18) es $\log_b n$ veces la solución homogénea, y de nuevo la solución particular excede a la homogénea. En el caso especial $d(n) = n^a$, (9.18) se reduce a $O(n^a \log n)$, por observaciones similares a las del caso (2).

Ejemplo 9.3. Considérense las siguientes recurrencias, con $T(1) = 1$.

- $T(n) = 4T(n/2) + n$
- $T(n) = 4T(n/2) + n^2$
- $T(n) = 4T(n/2) + n^3$

En cada caso, $a = 4$, $b = 2$, y la solución homogénea es n^2 . En la ecuación (1), con $d(n) = n$, se tiene $d(b) = 2$. Como $a = 4 > d(b)$, la solución particular también es n^2 , y $T(n)$ es $O(n^2)$ en (1).

En la ecuación (3), $d(n) = n^3$, $d(b) = 8$, y $a < d(b)$. Así, la solución particular es $O(n^{\log_2 d(b)}) = O(n^3)$, y $T(n)$ de la ecuación (3) es $O(n^3)$. Se puede deducir que la solución particular es del mismo orden que $d(n) = n^3$, aplicando las observaciones anteriores acerca de los $d(n)$ de la forma n^a y analizando el caso $a < d(b)$ de (9.17). En la ecuación (2), se tiene $d(b) = 4 = a$, con lo que se aplica (9.18). Como $d(n)$ es de la forma n^a , la solución particular y, por tanto, $T(n)$, es $O(n^2 \log n)$. \square

Otras funciones motrices

Existen otras funciones motrices no multiplicativas, por medio de las cuales se obtienen soluciones para (9.16) e incluso para (9.17). Se considerarán dos ejemplos. El primero generaliza cualquier función que sea el producto de una función multiplicativa y una constante mayor o igual que uno. La segunda es típica de un caso donde es preciso examinar (9.16) con detalle y obtener una cota superior ajustada a la solución particular.

Ejemplo 9.4. Considérese

$$\begin{aligned}T(1) &= 1 \\T(n) &= 3T(n/2) + 2n^{1.5}\end{aligned}$$

Ahora, $2n^{1.5}$ no es multiplicativa, pero $n^{1.5}$ sí lo es. Sea $U(n) = T(n)/2$ para toda n . Entonces,

$$\begin{aligned}U(1) &= 1/2 \\U(n) &= 3U(n/2) + n^{1.5}\end{aligned}$$

La solución homogénea, si $U(1)$ fuera 1, sería $n^{\log 3} = n^{1.59}$; como $U(1) = 1/2$, se demuestra con facilidad que la solución homogénea es $n^{1.59}/2$; y es $O(n^{1.59})$. Para la solución particular se ignora el hecho de que $U(1) \neq 1$, dado que al incrementar $U(1)$ seguramente no se reducirá la solución particular. Como $a = 3$, $b = 2$ y $b^{1.5} = 2.82 < a$, la solución particular también es $O(n^{1.59})$, que es la tasa de crecimiento de $U(n)$. Como $T(n) = 2U(n)$, $T(n)$ también es $O(n^{1.59})$ o $O(n^{\log 3})$. \square

Ejemplo 9.5. Considérese

$$\begin{aligned}T(1) &= 1 \\T(n) &= 2T(n/2) + n\log n\end{aligned}$$

Es fácil observar que la solución homogénea es n , pues $a = b = 2$. Sin embargo, $d(n) = n\log n$ no es multiplicativa, y se debe sumar la fórmula de la solución particular de (9.16) por medios apropiados. Esto es, se desea evaluar

$$\begin{aligned}\sum_{j=0}^{k-1} 2^{j-k} \log(2^{k-j}) &= 2^k \sum_{j=0}^{k-1} (k-j) \\&= 2^{k-1}k(k+1)\end{aligned}$$

Como $k = \log n$, la solución particular es $O(n\log^2 n)$, y esta solución, al ser mayor que la homogénea, también es el valor que se obtiene de $T(n)$. \square

Ejercicios

- 9.1 Escribanse algunas de las ecuaciones de recurrencia para las complejidades de tiempo y espacio correspondientes al siguiente algoritmo, suponiendo que n es una potencia de 2.

```

function camino (s, t, n: integer): boolean;
begin
  if n = 1 then
    if arista (s, t) then
      return (true)
    else
      return (false);
  { si se llega aquí es que n > 1 }
  for i := 1 to n do
    if camino (s, i, n div 2) and camino (i, t, n div 2) then
      return (true);
    return (false)
  end; { camino }

```

La función *arista*(*i, j*) devuelve verdadero si los vértices *i* y *j* de un grafo de *n* vértices están conectados por una arista o si *i* = *j*; de lo contrario, *arista*(*i, j*) devuelve falso. ¿Qué hace el programa?

- 9.2 Resuélvanse las siguientes recurrencias, donde $T(1) = 1$ y $T(n)$ para $n \geq 2$ satisface:
- $T(n) = 3T(n/2) + n$
 - $T(n) = 3T(n/2) + n^2$
 - $T(n) = 8T(n/2) + n^3$
- 9.3 Resuélvanse las siguientes recurrencias, donde $T(1) = 1$ y $T(n)$ para $n \geq 2$ satisface:
- $T(n) = 4T(n/3) + n$
 - $T(n) = 4T(n/3) + n^2$
 - $T(n) = 9T(n/3) + n^2$
- 9.4 Obténgase las cotas o mayúscula y omega mayúscula en los $T(n)$ definidos por las siguientes recurrencias. Supóngase que $T(1) = 1$
- $T(n) = T(n/2) + 1$
 - $T(n) = 2T(n/2) + \log n$
 - $T(n) = 2T(n/2) + n$
 - $T(n) = 2T(n/2) + n^2$

*9.5 Resuélvanse las siguientes recurrencias suponiendo una solución y comprobando la respuesta

a) $T(1) = 2$

$$T(n) = 2T(n - 1) + 1 \text{ para } n \geq 2$$

b) $T(1) = 1$

$$T(n) = 2T(n-1) + n \text{ para } n \geq 2$$

9.6 Verifíquense las respuestas del ejercicio 9.5 resolviendo las recurrencias por sustitución repetida.

9.7 Generalícese el ejercicio 9.6 resolviendo todas las recurrencias de la forma

$$T(1) = 1$$

$$T(n) = aT(n - 1) + d(n) \text{ para } n \geq 1$$

en función de a y $d(n)$.

*9.8 Supóngase en el ejercicio 9.7 que $d(n) = c^n$ para alguna constante $c \geq 1$. ¿Qué dependencia tiene la solución de $T(n)$ de la relación entre a y c ? ¿Qué es $T(n)$?

**9.9 Resuélvase para $T(n)$:

$$T(1) = 1$$

$$T(n) = \sqrt{n}T(\sqrt{n}) + n \text{ para } n \geq 2$$

9.10 Encuéntrense expresiones en su forma cerrada para las siguientes sumas.

a) $\sum_{i=0}^n i$

b) $\sum_{i=0}^n i^k$

c) $\sum_{i=0}^n 2^i$

d) $\sum_{i=0}^n \binom{n}{i}$

*9.11 Muéstrese que el número de órdenes distintos en que se puede multiplicar una secuencia de n matrices, está dado por la recurrencia

$$T(1) = 1$$

$$T(n) = \sum_{i=1}^{n-1} T(i)T(n-i)$$

Muéstrese que $T(n+1) = \frac{1}{n+1} \binom{2n}{n}$. Los $T(n)$ se conocen como *números de Catalán*.

**9.12 Muéstrese que el número de comparaciones requeridas para clasificar n elementos con clasificación por intercalación (*mergesort*) está dado por

$$T(1) = 0$$

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n - 1$$

donde $\lfloor x \rfloor$ denota la parte entera de x , y $\lceil x \rceil$, el entero más pequeño $\geq x$. Muéstrese que la solución a esta recurrencia es

$$T(n) = n\lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1$$

- 9.13** Muéstrese que el número de funciones booleanas de n variables está dado por la recurrencia

$$T(1) = 4$$

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

Resuélvase para $T(n)$.

- **9.14** Muéstrese que el número de árboles binarios de altura $\leq n$ está dado por la recurrencia

$$T(1) = 1$$

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

Muéstrese que $T(n) = \lfloor k^{2^n} \rfloor$ para alguna constante k . ¿Cuál es el valor de k ?

Notas bibliográficas

Bentley, Haken, y Saxe [1978], Greene y Knuth [1983], Liu [1968] y Lueker [1980] contienen material adicional sobre la solución de recurrencias. En Aho y Sloane [1973] se demuestra que muchas recurrencias no lineales de la forma $T(n) = (T(n-1))^2 + g(n)$ tienen una solución de la forma $T(n) = \lfloor k^{2^n} \rfloor$ donde k es una constante, como en el ejercicio 9.14.

10

Técnicas de diseño de algoritmos

A través de los años, los científicos de la computación han identificado diversas técnicas generales que a menudo producen algoritmos eficientes para la resolución de muchas clases de problemas. Este capítulo presenta algunas de las técnicas más importantes como dividir para vencer, programación dinámica, técnicas ávidas, el método de retroceso y búsqueda local. En el intento de diseñar un algoritmo para resolver un problema dado, a menudo es útil plantear cuestiones como «¿qué clase de solución se puede obtener de la programación dinámica, del enfoque ávido, de la técnica dividir para vencer o de otra técnica estándar?».

Sin embargo, debe subrayarse que hay algunos problemas, como los NP completos, para los cuales ni éstas ni otras técnicas conocidas producirán soluciones eficientes. Cuando se encuentra algún problema de este tipo, suele ser útil determinar si las entradas al problema tienen características especiales que se puedan explotar en la búsqueda de una solución, o si puede usarse alguna solución aproximada sencilla, en vez de la solución exacta, difícil de calcular.

10.1 Algoritmos dividir para vencer

Tal vez la técnica más importante y aplicada para el diseño de algoritmos eficientes sea la estrategia llamada dividir para vencer, que consiste en la descomposición de un problema de tamaño n en problemas más pequeños, de modo que a partir de la solución de dichos problemas sea posible construir con facilidad una solución al problema completo. Ya se han visto varias aplicaciones de esta técnica, como la clasificación por intercalación o los árboles binarios de búsqueda.

Para ilustrar el método, considérese el conocido acertijo de las «torres de Hanói». Consta de tres postes A , B y C . Inicialmente, el poste A tiene cierta cantidad de discos de distintos tamaños, con el más grande en la parte inferior y otros sucesivamente más pequeños encima, como se muestra en la figura 10.1. El objeto del acertijo es pasar un disco a la vez de un poste a otro, sin colocar nunca un disco grande sobre otro más pequeño, hasta que todos los discos estén en el poste B .

Pronto se descubre que el acertijo puede resolverse con el sencillo algoritmo siguiente. Se imaginan los postes dispuestos en un triángulo. Con un número de movimientos impar, se mueve el disco más pequeño hacia un poste en el sentido de las manecillas del reloj. Con un número de movimientos pares, se hace el único movimiento válido que no implique al disco más pequeño.

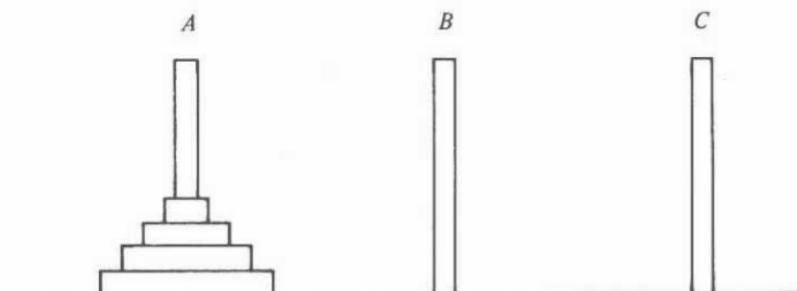


Fig. 10.1. Posición inicial en el acertijo de las torres de Hanoi.

El algoritmo anterior es conciso y correcto, pero es difícil entender por qué funciona y de descubrir intuitivamente. En cambio, considérese el siguiente enfoque de dividir para vencer. El problema de pasar los n discos más pequeños de A a B puede considerarse compuesto de dos problemas de tamaño $n - 1$. Primero se mueven los $n - 1$ discos más pequeños del poste A al C , dejando el n -ésimo disco más pequeño en el poste A . Se mueve ese disco de A a B . Después se pasan los $n - 1$ discos más pequeños de C a B . El movimiento de los $n - 1$ discos más pequeños se efectuará por medio de la aplicación recursiva del método. Como los n discos implicados en los movimientos son más pequeños que los demás discos, no es necesario preocuparse de los que se encuentran debajo de ellos en los postes A , B o C . Aunque el movimiento real de los discos individuales no es obvio y la simulación a mano es difícil debido al apilamiento de llamadas recursivas, el algoritmo es conceptualmente fácil de entender, de probar que es correcto y, tal vez, de considerarlo como primera opción. Es probable que la facilidad de descubrir los algoritmos de división haga que esta técnica sea tan importante, aunque en muchos casos los algoritmos son también más eficientes que otros más convencionales †.

Problema de la multiplicación de enteros grandes

Considérese el problema de la multiplicación de dos enteros X e Y de n bits. Recuérdese que el algoritmo para la multiplicación de enteros de n bits (o n dígitos), que se suele enseñar en la escuela primaria, implica el cálculo de n productos parciales de tamaño n , de aquí que sea un algoritmo $O(n^2)$, si se toman las multiplicaciones y las adiciones de un sólo bit o dígito como un paso. Un enfoque de clasificación del resultado por división en la multiplicación de enteros dividiría X e Y en dos enteros de $n/2$ bits cada uno, como se muestra en la figura 10.2. (Por simplicidad, se supone que n es una potencia de 2.)

† En el caso de las torres de Hanoi, el algoritmo dividir para vencer es en realidad el mismo que el dado inicialmente.

$$X := \boxed{A \quad B}$$

$$X = A2^{n/2} + B$$

$$Y := \boxed{C \quad D}$$

$$Y = C2^{n/2} + D$$

Fig. 10.2. División de un entero de n bits en segmentos de $n/2$ bits.

El producto de X e Y puede escribirse como

$$XY = AC2^n + (AD+BC)2^{n/2} + BD \quad (10.1)$$

Si se evalúa XY de esta forma directa, es necesario realizar cuatro multiplicaciones de enteros de $n/2$ bits (AC, AD, BC, BD), tres sumas de enteros con un máximo de $2n$ bits (correspondientes a los tres signos + de (10.1)), y dos desplazamientos (multiplicaciones por 2^n y por $2^{n/2}$). Como esas sumas y desplazamientos requieren $O(n)$ pasos, se puede escribir la siguiente recurrencia para $T(n)$, el número total de operaciones de bits necesarias para multiplicar enteros de n bits de acuerdo con (10.1).

$$T(1) = 1$$

$$T(n) = 4T(n/2) + cn \quad (10.2)$$

Por un razonamiento similar al del ejemplo 9.4, se toma la constante c de (10.2) como 1, de modo que la función motriz $d(n)$ es tan sólo n , y se deduce que la solución homogénea y la particular son $O(n^2)$.

En caso de que la fórmula (10.1) se utilice para multiplicar enteros, la eficiencia asintótica no es mayor que para el método de la escuela primaria. Pero recuérdese que para ecuaciones como (10.2) se consigue una mejora asintótica si se reduce el número de subproblemas. Puede ser sorprendente que se haga eso, pero considérese la siguiente fórmula para multiplicar X por Y .

$$XY = AC2^n + [(A-B)(D-C) + AC + BD]2^{n/2} + BD \quad (10.3)$$

Aunque (10.3) parece más complicada que (10.1), sólo requiere tres multiplicaciones de enteros con $(n/2)$ bits, AC, BD y $(A-B)(D-C)$, seis sumas o restas y dos desplazamientos. Como todas las operaciones, excepto las multiplicaciones, requieren $O(n)$ pasos, el tiempo $T(n)$ para multiplicar enteros de n bits por (10.3) está dado por

$$T(1) = 1$$

$$T(n) = 3T(n/2) + cn$$

cuya solución es $T(n) = O(n^{\log_2 3}) = O(n^{1.59})$.

El algoritmo completo, incluyendo los detalles implicados por el hecho de que (10.3) requiere multiplicaciones de enteros de $(n/2)$ bits tanto negativos como positivos, está dado en la figura 10.3. Obsérvese que las líneas (8) a (11) se realizan co-

piando bits, y la multiplicación por 2^n y $2^{n/2}$ de la línea (16), por desplazamientos. Del mismo modo, la multiplicación por s de la línea (16) tan sólo introduce el signo adecuado en el resultado.

```

function mult ( X, Y, n: integer ): integer;
  { X e Y son enteros con signo  $\leq 2^n$ .
    n es una potencia de 2. La función devuelve XY }
  var
    s: integer; { contiene el signo de XY }
    m1, m2, m3: integer; { contiene los tres productos }
    A, B, C, D: integer; { contiene las mitades izquierda y derecha de X e Y }
  begin
(1)    s := sign(X) * sign(Y);
(2)    X := abs(X);
(3)    Y := abs(Y); { hace positivos a X e Y }
(4)    if n = 1 then
(5)      if (X = 1) and (Y = 1) then
(6)        return (s)
      else
(7)        return (0)
    else begin
(8)      A := n/2 bits izquierdos de X;
(9)      B := n/2 bits derechos de X;
(10)     C := n/2 bits izquierdos de Y;
(11)     D := n/2 bits derechos de Y;
(13)     m1 := mult(A, C, n/2);
(14)     m2 := mult(A - B, D - C, n/2);
(15)     m3 := mult(B, D, n/2);
(16)     return (s * (m1 * 2n + (m1 + m2 + m3) * 2n/2 + m3))
  end
end; { mult }

```

Fig. 10.3. Algoritmo dividir para vencer de la multiplicación de enteros.

Obsérvese que el algoritmo de la figura 10.3 es asintóticamente más rápido que el método estudiado en la escuela primaria, al requerir $O(n^{1.59})$ pasos en vez de $O(n^2)$. Puede plantearse la pregunta de por qué no se imparte en la escuela primaria si es mejor. Existen dos respuestas: primero, aunque es fácil de implantar en un computador, la descripción del algoritmo es lo bastante compleja que si se intenta enseñarlo en la escuela primaria, los estudiantes no aprenderán a multiplicar. Más aún, se han ignorado las constantes de proporcionalidad. Mientras que el procedimiento *mult* de la figura 10.3 es asintóticamente superior al método usual, las constantes son tales que para problemas pequeños (de hecho hasta de 500 bits) el método de la escuela primaria es mejor, y en raras ocasiones se pide a estudiantes de ese nivel que multipliquen números de esa magnitud.

Programación de torneos de tenis

La técnica de dividir para vencer tiene varias aplicaciones, no sólo en el diseño de algoritmos, sino también en el diseño de circuitos, construcción de demostraciones matemáticas y en otros campos del quehacer humano. Se brinda un ejemplo para ilustrar esto. Considérese el diseño del programa de un torneo de tenis para $n = 2^k$ jugadores. Cada jugador debe enfrentarse a todos los demás, y debe tener un encuentro diario durante $n - 1$ días, que es el número mínimo de días necesarios para completar el torneo.

El programa del torneo es una tabla de n filas por $n - 1$ columnas cuya posición en fila i y columna j es el jugador con quien debe contender i el j -ésimo día.

El enfoque dividir para vencer construye un programa para la mitad de los jugadores, que está diseñado por una aplicación recursiva del algoritmo buscando un programa para una mitad de esos jugadores, y así sucesivamente. Cuando se llega hasta dos jugadores, se tiene el caso base y simplemente se emparejan.

Supóngase que hay ocho jugadores. El programa para los jugadores 1 a 4 ocupa la esquina superior izquierda (4 filas por 3 columnas) del programa que se está construyendo. La esquina inferior izquierda (4 filas por 3 columnas) debe enfrentar a los jugadores con numeración más alta (5 a 8). Este subprograma se obtiene agregando 4 a cada entrada de la esquina superior izquierda.

Ahora se ha simplificado el problema, y lo único que falta es enfrentar a los jugadores de baja numeración con los de alta numeración. Esto es fácil si los jugadores 1 a 4 contienen con los 5 a 8, respectivamente, en el día 4, y se permutan cíclicamente 5 a 8 en los días siguientes. El proceso se ilustra en la figura 10.4. El lector debe ser capaz de generalizar las ideas de esta figura para obtener un programa de 2^k jugadores para cualquier k .

Balanceo de subproblemas

En el diseño de algoritmos siempre hay que afrontar varios compromisos. Ha surgido la regla de que suele ser ventajoso balancear los costos de competencia siempre que sea posible. Por ejemplo, en el capítulo 5 se vio que los árboles 2-3 balanceaban los costos de búsqueda con los de inserción, mientras que los métodos más directos requieren $O(n)$ pasos por cada búsqueda o por cada inserción, aun cuando la otra operación pueda efectuarse en un número constante de pasos.

En forma semejante, para los algoritmos de dividir para vencer, casi siempre es mejor que los subproblemas tengan un tamaño aproximadamente igual. Por ejemplo, la clasificación por inserciones puede considerarse como la división de un problema en dos subproblemas, uno de tamaño 1 y otro de tamaño $n - 1$, con un costo máximo de n pasos para combinarlos. Esto da la recurrencia

$$T(n) = T(1) + T(n - 1) + n$$

que tiene una solución $O(n^2)$. La clasificación por intercalación, por otra parte, divide el problema en dos subproblemas de tamaño $n/2$ y tiene un rendimiento

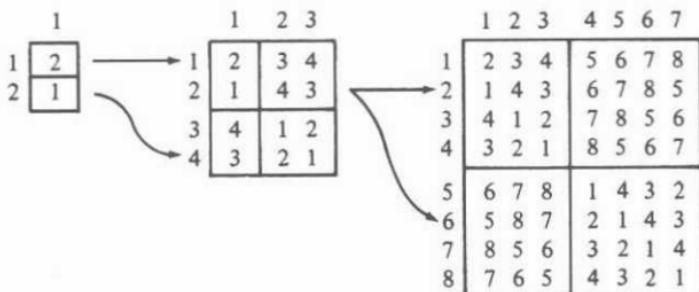


Fig. 10.4. Torneo para ocho jugadores.

$O(n \log n)$. Como principio general, a menudo resulta que dividir un problema en subproblemas iguales o casi iguales es un factor crucial para la obtención de un buen rendimiento.

10.2 Programación dinámica

A menudo sucede que no hay manera de dividir un problema en un pequeño número de subproblemas cuya solución pueda combinarse para resolver el problema original. En tales casos, se puede intentar dividir el problema en tantos subproblemas como sea necesario, dividir cada subproblema en subproblemas más pequeños, y así sucesivamente. Si eso es todo lo que se hace, quizás se produzca un algoritmo de tiempo exponencial.

No obstante, con frecuencia, sólo hay un número polinomial de subproblemas, de aquí que se deba resolver algún subproblema muchas veces. Si, en cambio, se conserva la solución a cada subproblema resuelto, y tan sólo se toma la respuesta cuando se requiere, se obtiene un algoritmo de tiempo polinomial.

Desde el punto de vista de la realización, algunas veces es más fácil crear una tabla de las soluciones de todos los subproblemas que se tengan que resolver. Se rellena la tabla sin tener en cuenta si se necesita realmente un subproblema particular en la solución total. La formación de la tabla de subproblemas para alcanzar una solución a un problema dado se denomina *programación dinámica*, nombre procedente de la teoría de control.

La forma de un algoritmo de programación dinámica puede variar, pero hay un esquema común: una tabla a llenar y un orden en el cual se hacen las entradas. Se ilustrarán las técnicas mediante dos ejemplos, el cálculo de apuestas en un encuentro como la Serie Mundial de Béisbol y el «problema de la triangulación».

Apuestas en la Serie Mundial de Béisbol

Supóngase que dos equipos, *A* y *B*, tienen un enfrentamiento para ver quién es el primero en ganar *n* partidos para una *n* en particular. La Serie Mundial es uno de

tales enfrentamientos, con $n = 4$. Se supone que A y B son igualmente competentes, de modo que cada uno tiene un 50% de oportunidades de ganar cualquier partido. Sea $P(i, j)$ la probabilidad de que A gane el encuentro, considerando que A necesita i partidos para ganar, y B necesita j . Por ejemplo, si los Dodgers han ganado dos partidos y los Yankees uno, entonces $i = 2$, $j = 3$, y se observa que $P(2, 3)$ es 11/16.

Para calcular $P(i, j)$, se emplea una ecuación de recurrencia en dos variables. Primero, si $i = 0$ y $j > 0$, el equipo A ya ha ganado el encuentro, por lo que $P(0, j) = 1$. Igualmente, $P(i, 0) = 0$ para $i > 0$. Si i y j son mayores que cero, deberá jugarse al menos otro partido más y los dos equipos ganan la mitad de las veces. Así, $P(i, j)$ debe ser el promedio de $P(i - 1, j)$ y $P(i, j - 1)$, siendo la primera de ellas la probabilidad de que A gane el encuentro si gana el siguiente partido, y la segunda, la probabilidad de que A gane el encuentro aun cuando pierda el siguiente partido. En resumen:

$$\begin{aligned} P(i, j) &= 1 \quad \text{si } i = 0 \text{ y } j > 0 \\ &= 0 \quad \text{si } i > 0 \text{ y } j = 0 \\ &= (P(i - 1, j) + P(i, j - 1))/2 \quad \text{si } i > 0 \text{ y } j > 0 \end{aligned} \quad (10.4)$$

Si se utiliza (10.4) recursivamente como una función, se demuestra que $P(i, j)$ requiere un tiempo no mayor que $O(2^{i+j})$. Sea $T(n)$ el tiempo máximo requerido por una llamada a $P(i, j)$, donde $i + j = n$. Entonces, de (10.4),

$$T(1) = c$$

$$T(n) = 2T(n - 1) + d$$

para algunas constantes c y d . Es útil verificar por los medios analizados en el capítulo anterior, que $T(n) \leq 2^{n-1}c + (2^{n-1} - 1)d$, lo cual es $O(2^n)$ u $O(2^{i+j})$.

Así, se ha obtenido una cota superior exponencial para el tiempo requerido por el cálculo recursivo de $P(i, j)$. Sin embargo, para tener la convicción de que la fórmula recursiva para $P(i, j)$ es una mala forma de calcularla, se toma una cota inferior omega mayúscula. Se deja como ejercicio demostrar que al llamar a $P(i, j)$, el número total de llamadas a P que se hace es $\binom{i+j}{i}$, el número de formas de escoger i objetos a partir de $i + j$. Si $i = j$, ese número es $\Omega(2^n/\sqrt{n})$, donde $n = i + j$. Así, $T(n)$ es $\Omega(2^n/\sqrt{n})$ y, de hecho, se demuestra que también es $O(2^n/\sqrt{n})$. Mientras $2^n/\sqrt{n}$ crece asintóticamente más lento que 2^n , la diferencia no es muy grande, y $T(n)$ crece demasiado rápido como para que el cálculo recursivo de $P(i, j)$ sea práctico.

El problema con el cálculo recursivo es que se calcula la misma $P(i, j)$ muchas veces. Por ejemplo, si se desea obtener $P(2, 3)$, se calcula, por (10.4), $P(1, 3)$ y $P(2, 2)$. $P(1, 3)$ y $P(2, 2)$ requieren el cálculo de $P(1, 2)$, por lo que se calcula ese valor dos veces.

Una forma mejor de calcular $P(i, j)$ es llenar la tabla sugerida en la figura 10.5. La fila inferior está llena de ceros y la columna más a la derecha está llena de unos, debido a las dos primeras filas de (10.4). Por la última fila de (10.4), cada una de las otras entradas es el promedio de la entrada de abajo y de la entrada de la derecha. Así, una forma apropiada de llenar la tabla es proceder en diagonales a partir

la esquina inferior derecha, y procediendo hacia arriba y hacia la izquierda en diagonales que representan las posiciones con valor constante de $i + j$, como se sugiere en la figura 10.6. Este programa se presenta en la figura 10.7, suponiendo que opera en un arreglo bidimensional P de tamaño apropiado.

1/2	21/32	13/16	15/16	1	4	
11/32	1/2	11/16	7/8	1	3	t
3/16	5/16	1/2	3/4	1	2	j
1/16	1/8	1/4	1/2	1	1	
0	0	0	0		0	
4	3	2	1	0		
			$\leftarrow i$			

Fig. 10.5. Tabla de apuestas.

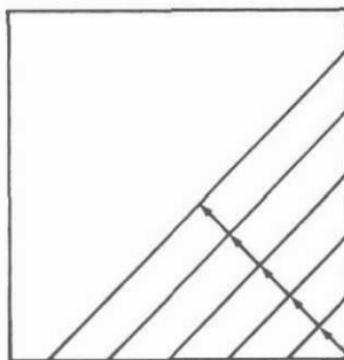


Fig. 10.6. Patrón de computación.

El análisis de la función *apuestas* es fácil. El ciclo de las líneas (4) a (5) requiere un tiempo $O(s)$, y domina el tiempo $O(1)$ de las líneas (2) a (3). Así, el ciclo externo requiere un tiempo $O(\sum_{s=1}^n s)$ u $O(n^2)$, donde $i + j = n$. Con ello, la programación dinámica requiere un tiempo $O(n^2)$, comparado con $O(2^n / \sqrt{n})$ para el enfoque directo. Puesto que $2^n / \sqrt{n}$ crece mucho más rápido que n^2 , casi siempre es preferible utilizar la programación dinámica que el enfoque recursivo.

```
function apuestas ( i, j: integer ) : real;
var
  s, k: integer;
begin
```

```

(1)      for s := 1 to i + j do begin
          { calcular la diagonal de las posiciones cuyos índices se suman a s }
(2)          P[0, s] := 1.0;
(3)          P[s, 0] := 0.0;
(4)          for k := 1 to s - 1 do
(5)              P[k, s - k] := (P[k - 1, s - k] + P[k, s - k - 1])/2.0
end;
(6)      return (P[i, j])
end; { apuestas }

```

Fig. 10.7. Cálculo de apuestas.

Problema de la triangulación

Como otro ejemplo de programación dinámica, considérese el problema de la *triangulación* de un polígono. Se dan los vértices de un polígono y una medida de la distancia entre cada par de vértices. La distancia puede ser la normal (euclidiana) en el plano, o puede ser una función de costo arbitraria dada por una tabla. El problema consiste en seleccionar un conjunto de *cuerdas* (líneas entre vértices no adyacentes) de modo que dos cuerdas no se crucen entre sí y que todo el polígono quede dividido en triángulos. La longitud total (distancia entre puntos finales) de las cuerdas seleccionadas debe ser mínima. Ese conjunto de cuerdas se llama *triangulación mínimal*.

Ejemplo 10.1. La figura 10.8 muestra un polígono de siete lados y las coordenadas (x, y) de sus vértices. La función de distancia es la distancia normal euclidiana. Una triangulación no minimal, se muestra por medio de las líneas de puntos. Su costo es la suma de las longitudes de las cuerdas (v_0, v_2) , (v_0, v_3) , (v_0, v_5) y (v_3, v_5) , o $\sqrt{8^2+16^2} + \sqrt{15^2+16^2} + \sqrt{22^2+2^2} + \sqrt{7^2+14^2} = 77.56$. \square

Además de ser interesante por sí mismo, el problema de la triangulación tiene diversas aplicaciones útiles. Por ejemplo, Fuchs, Kedem, y Uselton [1977] usaron una generalización del problema de la triangulación para el propósito siguiente. Considerese el problema de sombrear la imagen bidimensional de un objeto cuya superficie está definida por una colección de puntos en un espacio tridimensional. La fuente de luz proviene de una dirección dada, y el brillo de un punto en la superficie depende de los ángulos entre la dirección de la luz, la dirección de la vista del observador y una perpendicular a la superficie en ese punto. Para estimar la dirección de la superficie en un punto, se calcula una triangulación mínima de los puntos que definen la superficie.

Cada triángulo define un plano en el espacio tridimensional, y dado que se encontró una triangulación mínima, es de esperarse que los triángulos sean muy pequeños. Es fácil encontrar la dirección de una perpendicular a un plano, así que se calcula la intensidad de la luz para los puntos de cada triángulo en el supuesto de que la superficie puede considerarse como un plano triangular en una región dada. Si los triángulos no son suficientemente pequeños como para hacer que la intensidad de la luz aparezca suave, entonces obtener el promedio local puede mejorar la imagen.

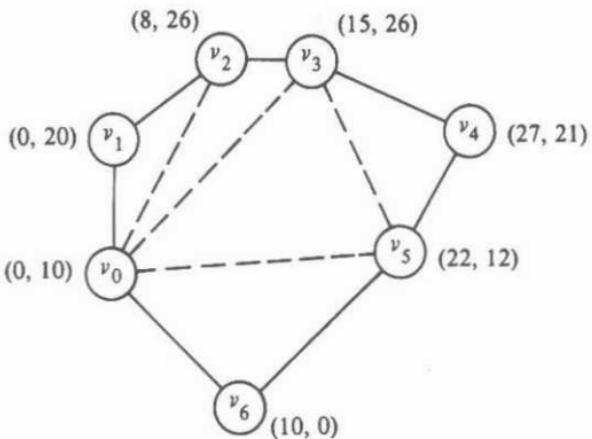


Fig. 10.8. Heptágono y triangulación.

Antes de proseguir con la solución de la programación dinámica al problema de la triangulación, se dejarán establecidas dos observaciones acerca de las triangulaciones que ayudarán a diseñar el algoritmo. Se supone que hay un polígono con n vértices v_0, v_1, \dots, v_{n-1} , en el sentido de las manecillas del reloj.

Hecho 1. En cualquier triangulación de un polígono con más de tres vértices, cada par de vértices adyacentes es tocado al menos por una cuerda. Para ver esto, supóngase que ni v_i ni v_{i+1} † fueron tocados por una cuerda. Entonces la región que la arista (v_i, v_{i+1}) limita debe incluir las aristas (v_{i-1}, v_i) , (v_{i+1}, v_{i+2}) y al menos una arista adicional. Esta región no sería un triángulo.

Hecho 2. Si (v_i, v_j) es una cuerda de una triangulación, debe haber algún v_k tal que (v_i, v_k) y (v_k, v_j) sean aristas del polígono o cuerdas. De otra forma, (v_i, v_j) puede limitar una región que no sea un triángulo.

Para iniciar la búsqueda de una triangulación minimal, se escogen dos vértices adyacentes, como v_0 y v_1 . Por los dos hechos, se sabe que en cualquier triangulación y, por tanto, en la triangulación mínima, debe haber un vértice v_k tal que (v_1, v_k) y (v_k, v_0) sean cuerdas o aristas en la triangulación. Así pues, se debe considerar la bondad de la triangulación que se obtendrá después de seleccionar cada valor posible de k . Si el polígono tiene n vértices, se puede hacer un total de $(n - 2)$ selecciones.

Cada selección de k da lugar como máximo a dos subproblemas, que son polígonos formados por una cuerda y las aristas del polígono original que van de un extremo a otro de la cuerda. Por ejemplo, la figura 10.9 muestra los dos subproblemas que resultan al seleccionar el vértice v_3 .

A continuación, se deben encontrar triangulaciones mínimas para los polígonos de la figura 10.9(a) y (b). Lo primero que hay que hacer es considerar otra vez todas

† En lo sucesivo, se considerará que todos los subíndices están calculados módulo n . Así, en la figura 10.8, v_i y v_{i+1} pueden ser v_6 y v_0 , respectivamente, puesto que $n = 7$.

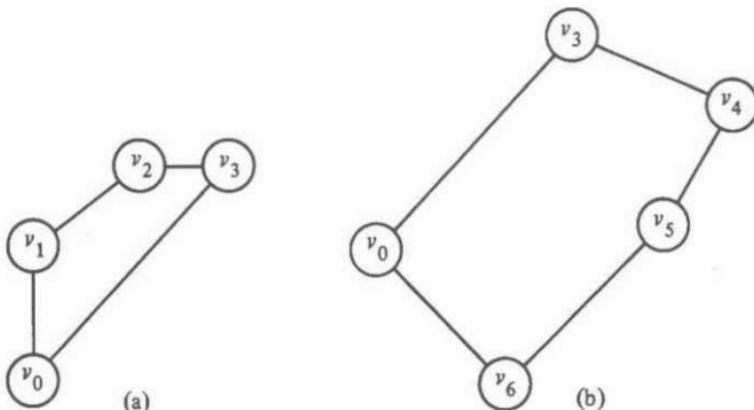


Fig. 10.9. Los dos subproblemas después de seleccionar v_3 .

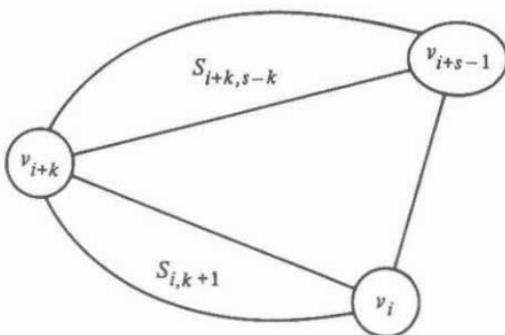
las cuerdas que parten de dos vértices adyacentes. Por ejemplo, en la solución de la figura 10.9(b), se considera la selección de la cuerda (v_3, v_5) , la cual deja el subproblema (v_0, v_3, v_5, v_6) , un polígono en el cual dos lados, (v_0, v_3) y (v_3, v_5) , son cuerdas del polígono original. Este camino conduce a un algoritmo de tiempo exponencial.

Sin embargo, considerando el triángulo que incluye la cuerda (v_0, v_k) nunca se tienen que considerar polígonos en los cuales más de un lado sean cuerdas del polígono original. El hecho 2 dice que, en la triangulación mínima, la cuerda del subproblema, como (v_0, v_3) de la figura 10.9(b), debe formar un triángulo con uno de los otros vértices. Por ejemplo, si se selecciona v_4 , se forma el triángulo (v_0, v_3, v_4) y el subproblema (v_0, v_4, v_5, v_6) , que sólo tiene una cuerda del polígono original. Al probar con v_5 , se forman los subproblemas (v_3, v_4, v_5) y (v_0, v_5, v_6) , con las cuerdas (v_3, v_5) y (v_0, v_5) nada más.

En general, se define el *subproblema de tamaño s* partiendo del vértice v_i , denotado S_{is} , como el problema de triangulación minimal para el polígono formado por los s vértices que comienzan en v_i y siguen en el sentido de las manecillas del reloj, esto es, $v_i, v_{i+1}, \dots, v_{i+s-1}$. La cuerda en S_{is} es (v_i, v_{i+s-1}) . Por ejemplo, en la figura 10.9(a) es S_{04} y en la figura 10.9 (b) es S_{35} . Para resolver S_{is} debemos considerar las tres opciones siguientes.

1. Tomar el vértice v_{i+s-2} para formar un triángulo con las cuerdas (v_i, v_{i+s-1}) , (v_i, v_{i+s-2}) y con el tercer lado (v_{i+s-2}, v_{i+s-1}) , y después resolver el subproblema $S_{i,s-1}$.
2. Tomar el vértice v_{i+1} para formar un triángulo con las cuerdas (v_i, v_{i+s-1}) , (v_{i+1}, v_{i+s-1}) y con el tercer lado (v_i, v_{i+1}) , y después resolver el subproblema $S_{i+1,s-1}$.
3. Para alguna k entre 2 y $s - 3$, tomar el vértice v_{i+k} y formar un triángulo con los lados (v_i, v_{i+k}) , (v_{i+k}, v_{i+s-1}) , y (v_i, v_{i+s-1}) , y después resolver los subproblemas $S_{i,k+1}$ y $S_{i+k,s-k}$.

Si se recuerda que la «solución» a cualquier subproblema de tamaño tres o menor no requiere acción alguna, se puede resumir (1) a (3) diciendo que se toma al-

Fig. 10.10. División de S_{is} en subproblemas.

gún vértice k entre 1 y $s - 2$ y se resuelven los subproblemas $S_{i,k+1}$ y $S_{i+k,s-k}$. La figura 10.10 ilustra esta división en subproblemas.

Si se emplea el algoritmo recursivo implícito en las reglas anteriores para resolver subproblemas de tamaño 4 o mayores se podrá mostrar que cada llamada en un subproblema de tamaño s provoca un total de 3^{s-4} llamadas recursivas, si se «resuelven» directamente los subproblemas de tamaño 3 o menores y sólo se cuentan las llamadas en subproblemas de tamaño 4 o mayores. Así, el número de subproblemas a resolver es exponencial en s . Dado que el problema inicial es de tamaño n , donde n es el número de vértices del polígono dado, el número total de pasos realizados por este procedimiento recursivo es exponencial en n .

Es evidente que todavía hay algo erróneo en este análisis, porque se sabe que sin contar el problema original, sólo hay $n(n - 4)$ subproblemas diferentes que necesitan solución, y se representan por S_{is} , donde $0 \leq i < n$ y $4 \leq s < n$. Es obvio que no todos los subproblemas resueltos por el procedimiento recursivo son diferentes. Por ejemplo, si en la figura 10.8 se escoge la cuerda (v_0, v_3) , y después, en el subproblema de la figura 10.9(b), v_4 , hay que resolver el subproblema S_{44} . Pero también se tendría que resolver este problema si se tomara primero la cuerda (v_0, v_4) , o la cuerda (v_1, v_4) , y entonces, al resolver el subproblema S_{45} , se tomaría el vértice v_0 para completar un triángulo con v_1 y v_4 .

Esto sugiere una forma eficiente de resolver el problema de la triangulación. Se prepara una tabla que contenga el costo C_{is} de la triangulación S_{is} para toda i y s . Dado que la solución a cualquier problema dado sólo depende de la solución a problemas de tamaño menor, el orden lógico en el cual se debe llenar la tabla es de acuerdo con el tamaño. Esto es, para tamaños $s = 4, 5, \dots, n - 1$ se introduce el costo mínimo para los subproblemas S_{is} , para todos los vértices i . Es conveniente incluir también los problemas de tamaño $0 \leq s < 4$, pero recuérdese que S_{is} tiene costo 0 si $s < 4$.

De las reglas anteriores (1) a (3) para encontrar subproblemas, la fórmula para calcular C_{is} para $s \geq 4$ es:

$$C_{is} = \min_{1 \leq k \leq s-2} [C_{i,k+1} + C_{i+k,s-k} + D(v_i, v_{i+k}) + D(v_{i+k}, v_{i+s-1})] \quad (10.5)$$

donde $D(v_p, v_q)$ es la longitud de la cuerda entre los vértices v_p y v_q , si v_p y v_q no son puntos adyacentes en el polígono; $D(v_p, v_q) = 0$ si v_p y v_q son adyacentes.

Ejemplo 10.2. La figura 10.11 contiene la tabla de costos de S_{is} para $0 \leq i \leq 6$ y $4 \leq s \leq 6$, basada en el polígono y las distancias de la figura 10.8. Los costos de las filas con $s < 3$ son todos cero. Se ha llenado la posición C_{07} , en la columna 0 y la fila para $s = 7$. Este valor, como todos los de la misma fila, representa la triangulación del polígono completo. Para ver esto, obsérvese que se puede, si se desea, considerar la arista (v_0, v_6) como una cuerda de un polígono mayor y el polígono de la figura 10.8 como un subproblema de este polígono, el cual tiene una serie de vértices adicionales que se extienden en el sentido de las manecillas del reloj desde v_6 hasta v_0 . Obsérvese que toda la fila para $s = 7$ tiene el mismo valor que C_{07} , para conservar la exactitud de cálculo.

Como ejemplo, se proporciona la demostración de cómo se introdujo el valor 38.09 de la columna para $i = 6$ y la fila para $s = 5$. De acuerdo con (10.5), el valor de esta posición, C_{65} , es el mínimo de tres sumas correspondientes a $k = 1, 2$ ó 3 . Esas sumas son:

$$C_{62} + C_{04} + D(v_6, v_0) + D(v_0, v_3)$$

$$C_{63} + C_{13} + D(v_6, v_1) + D(v_1, v_3)$$

$$C_{64} + C_{22} + D(v_6, v_2) + D(v_2, v_3)$$

7	$C_{07} = 75.43$						
6	$C_{06} = 53.54$	$C_{16} = 55.22$	$C_{26} = 57.58$	$C_{36} = 64.69$	$C_{46} = 59.78$	$C_{56} = 59.78$	$C_{66} = 63.62$
5	$C_{05} = 37.54$	$C_{15} = 31.81$	$C_{25} = 35.49$	$C_{35} = 37.74$	$C_{45} = 45.50$	$C_{55} = 39.98$	$C_{65} = 38.09$
4	$C_{04} = 16.16$	$C_{14} = 16.16$	$C_{24} = 15.65$	$C_{34} = 15.65$	$C_{44} = 22.69$	$C_{54} = 22.69$	$C_{64} = 17.89$
s	$i=0$	1	2	3	4	5	6

Fig. 10.11. Tabla de C_{is} .

Las distancias requeridas se calculan a partir de las coordenadas de los vértices:

$$D(v_2, v_3) = D(v_6, v_0) = 0$$

(puesto que son aristas del polígono, no cuerdas, y se presentan en forma «gratuita»)

$$D(v_6, v_2) = 26.08$$

$$D(v_1, v_3) = 16.16$$

$$D(v_6, v_1) = 22.36$$

$$D(v_0, v_3) = 21.93$$

Las tres sumas anteriores dan 38.09, 38.52 y 43.97, respectivamente. Se concluye que el costo mínimo del subproblema S_{65} es 38.09. Más aún, como la primera suma fue la menor, se sabe que para alcanzar este valor es necesario utilizar los subproblemas S_{62} y S_{04} , esto es, seleccionar la cuerda (v_0, v_3) y resolver S_{64} como se pude; la cuerda (v_1, v_3) es la elección preferida para este subproblema. \square

Existe un truco útil para llenar la tabla de la figura 10.11 de acuerdo con la fórmula (10.5). Cada término de la operación \min de (10.5) requiere un par de valores. El primer par, para $k = 1$, puede encontrarse en la tabla a) en la parte «inferior» (la fila de $s = 2$) † de la columna del elemento que se está calculando, y b) justo abajo y a la derecha ‡ del elemento calculado. El segundo par está a) junto a la parte inferior de la columna, y b) dos posiciones más abajo y a la derecha. La figura 10.12 muestra las dos líneas de posiciones que se siguen para obtener todos los pares de valores que es necesario considerar simultáneamente. El patrón —subir por la columna y bajar por la diagonal— es común para el llenado de tablas durante la programación dinámica.

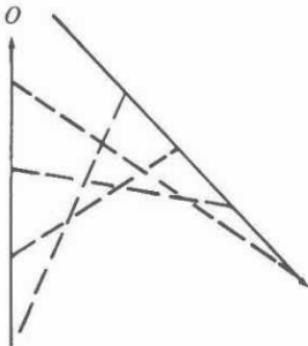


Fig. 10.12. Patrón de seguimiento de la tabla para calcular un elemento.

Para encontrar soluciones a partir de la tabla

Aunque la figura 10.11 ofrece el costo de la triangulación mínima, no proporciona de inmediato la triangulación misma. Para cada posición se necesita el valor de k que produjo el mínimo en (10.5). Luego se puede deducir que la solución consta de las cuerdas (v_i, v_{i+k}) y (v_{i+k}, v_{i+s-1}) (a menos que una de ellas no sea cuerda, porque $k = 1$ o $k = s - 2$), más las cuerdas que estén implicadas por las soluciones a $S_{i,k+1}$ y $S_{i+k,s-k}$. Al calcular un elemento de la tabla, es útil incluir el valor de k que brinde la mejor solución.

Ejemplo 10.3. En la figura 10.11, el valor C_{07} , que representa la solución al problema completo de la figura 10.8, procede de los términos de $k = 5$ en (10.5). Esto es,

† Recuérdese que la tabla de la figura 10.11 tiene filas de ceros por debajo de las que se muestran.

‡ Por «a la derecha» se quiere decir que la tabla se lee en forma circular. Así, si uno se encuentra en la columna de más a la derecha, la columna «a la derecha» es la que está más a la izquierda.

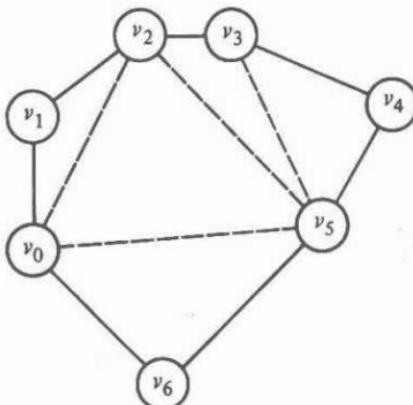


Fig. 10.13. Triangulación de costo minimal.

el problema S_{07} se divide en S_{06} y S_{52} ; el primero es el problema con seis vértices v_0, v_1, \dots, v_5 , y el último es un «problema» trivial de costo 0. Así, se introduce la cuerda (v_0, v_5) de costo 22.09 y se debe resolver S_{06} .

El costo mínimo de C_{06} procede de los términos para $k = 2$ en (10.5), en tanto el problema S_{06} se divide en S_{03} y S_{24} . El primero es el triángulo con vértices v_0, v_1 y v_2 , mientras que el último es un cuadrilátero definido por v_2, v_3, v_4 y v_5 . S_{03} no necesita ser resuelto, pero S_{24} sí, y deben incluirse los costos de las cuerdas (v_0, v_2) y (v_2, v_5) , los cuales son 17.89 y 19.80, respectivamente. El valor mínimo para C_{24} se supone cuando $k = 1$ en (10.5), dando los subproblemas C_{22} y C_{33} , que tienen tamaño menor o igual que tres y, por tanto, costo 0. Se introduce la cuerda (v_3, v_5) , con un costo de 15.65. □

10.3 Algoritmos ávidos

Considérese el problema de dar un cambio. Supónganse monedas de 25 € (un cuarto) 10 € (un décimo), 5 € (un vigésimo) y 1 € (un centavo), y que se desea dar un cambio de 63 €. Sin pensar, se convierte esta cantidad a dos cuartos, un décimo y tres centavos. No sólo se determinó rápidamente una lista de monedas con el valor correcto, sino que se produjo la lista más corta de monedas con ese valor.

El algoritmo empleado probablemente fue seleccionar la moneda mayor cuyo valor no excedía de 63 € (un cuarto), agregarla a la lista y sustraer su valor de 63, quedando 38 €. De nuevo, se seleccionó la moneda más grande cuyo valor no fuera mayor de 38 € (otro cuarto) y se añadió a la lista, y así sucesivamente.

Este método de dar cambio es un *algoritmo ávido*. En cualquier etapa individual, un algoritmo ávido selecciona la opción que sea «localmente óptima» en algún sentido particular. Obsérvese que el algoritmo ávido para dar cambio produce una solución óptima general debido a las propiedades de las monedas. Si las monedas tuvieran valores 1 €, 5 € y 11 € y se deseara dar un cambio de 15 €, el

algoritmo ávido seleccionaría primero la moneda de 11 € y después cuatro de 1 €, para un total de cinco monedas. Sin embargo, tres monedas de 5 € bastan.

Ya se han visto varios algoritmos ávidos, como el algoritmo del camino más corto de Dijkstra y el algoritmo de árboles abarcadores de costo mínimo de Kruskal. El algoritmo de Dijkstra es «ávido» en el sentido de que siempre escoge el vértice más cercano al origen de entre aquellos cuyo camino más corto todavía se desconoce. El algoritmo de Kruskal también es «ávido»: de las aristas restantes, elige la más corta de entre aquellas que no crean un ciclo.

Es necesario subrayar el hecho de que no todos los enfoques ávidos llegan a dar mejor resultado. Como sucede en la vida real, una estrategia ávida puede dar un buen resultado durante un tiempo, pero el resultado general puede ser pobre. Como ejemplo, se considera lo que sucede al permitir aristas ponderadas negativas en los algoritmos de Dijkstra y de Kruskal. Resulta que el algoritmo de árboles abarcadores de Kruskal no se ve afectado y continuará produciendo el árbol de costo mínimo, pero el algoritmo de Dijkstra en algunos casos no produce los caminos más cortos.

Ejemplo 10.4. En la figura 10.14 se presenta un grafo con una arista entre b y c cuyo costo es negativo. Al aplicar el algoritmo de Dijkstra con origen s , primero se descubrirá, correctamente, que el camino mínimo hacia a tiene longitud 1. Ahora, considerando sólo las aristas de s o a a b o c , se espera que b tenga el camino más corto desde s , es decir, $s \rightarrow a \rightarrow b$, de longitud 3. Así, se descubre que c tiene un camino más corto desde s de longitud 1.

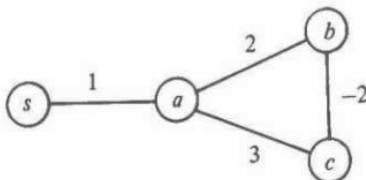


Fig. 10.14. Grafo con una arista ponderada negativa.

Sin embargo, esta selección «ávida» de b antes que c fue errónea desde un punto de vista general. Dado que el camino $s \rightarrow a \rightarrow c \rightarrow b$ tiene una longitud de sólo 2, la distancia mínima de 3 para b fue errónea †. □

Algoritmos ávidos como procedimientos heurísticos

Para algunos problemas, no existen algoritmos ávidos conocidos que produzcan soluciones óptimas; con todo, existen algunos que pueden llegar a producir soluciones «buenas» con una alta probabilidad. A menudo, una solución semióptimal

† De hecho, se debe tener cuidado con el significado de «camino más corto» cuando existen aristas negativas. Si se permiten ciclos con costo negativo, entonces se pueden recorrer repetidamente ese ciclo para alcanzar distancias negativas arbitrariamente grandes, por lo que se desea ceñirse a caminos acíclicos.

con un costo de un pequeño porcentaje sobre la óptima es muy satisfactoria. En esos casos, un algoritmo ávido con frecuencia brinda la forma más rápida para llegar a una solución «buena». De hecho, si el problema es tal que la única forma de alcanzar una solución óptima es mediante el uso de una técnica de búsqueda exhaustiva, un algoritmo ávido u otro procedimiento heurístico para llegar a una buena solución, pero no necesariamente óptima, puede ser la única opción real.

Ejemplo 10.5. Se presenta un conocido problema donde los únicos algoritmos conocidos que producen soluciones óptimas son del tipo «intentar todas las posibilidades» y pueden tener tiempos de ejecución con entradas de tamaño exponencial. El problema, llamado *problema del agente viajero*, o *PAV*, es encontrar, en un grafo no dirigido con pesos en las aristas, un *recorrido* (un ciclo simple que incluya todos los vértices) donde la suma de todos los pesos de las aristas sea un mínimo. Un recorrido se suele denominar *ciclo de Hamilton* (o *ciclo hamiltoniano*).

La figura 10.15(a) muestra un ejemplo del problema del agente viajero, un grafo con seis vértices (a veces llamados «ciudades»). Se dan las coordenadas de cada vértice, y se toma el peso de cada arista como su longitud. Obsérvese que, por convención con el PAV, se supone que todas las aristas existen, esto es, el grafo es completo. En ejemplos más generales, donde los pesos de las aristas no estén basados en la distancia euclíadiana, se puede encontrar un peso infinito sobre aristas que en realidad no están presentes.

Las figuras 10.15(b) a (e) muestran cuatro recorridos de las seis «ciudades» de la figura 10.15(a). El lector puede preguntarse cuál de los cuatro es el óptimo, si lo es alguno. Las longitudes de los cuatro recorridos son 50.00, 49.73, 48.39 y 49.78, respectivamente; (d) es el más corto de todos los recorridos posibles.

El PAV tiene varias aplicaciones prácticas. Como su nombre indica, puede usarse para planear la ruta de una persona que debe visitar varios sitios y re-

$$\begin{array}{ll}
 c \bullet (1, 7) & d \bullet (15, 7) \\
 b \bullet (4, 3) & e \bullet (15, 4) \\
 a \bullet (0, 0) & f \bullet (18, 0) \\
 \end{array}$$

(a) seis «ciudades»

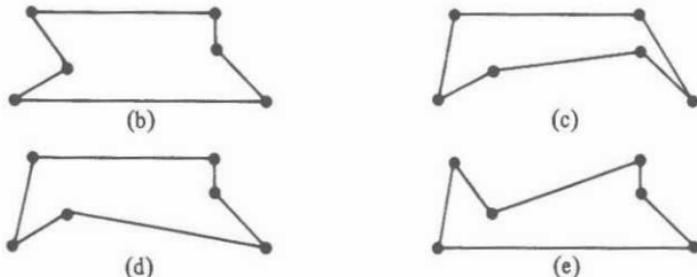


Fig. 10.15. Ejemplo del problema del agente viajero.

gresar al punto inicial. Por ejemplo, el PAV se ha usado para seleccionar la ruta de los recolectores de dinero de teléfonos públicos. Los vértices son los teléfonos y la oficina de cobro. El costo de cada arista es el tiempo del viaje entre los dos sitios en cuestión.

Otra aplicación del PAV es la resolución del *problema del recorrido del caballo* para encontrar una secuencia de movimientos de modo que el caballo pueda visitar cada cuadro del tablero de ajedrez sólo una vez y regrese al punto de partida. Se dan como vértices los cuadros del tablero de ajedrez y las aristas entre dos cuadros que son un movimiento del caballo con peso 0; las demás aristas tienen peso infinito. Un recorrido óptimal tiene peso 0 y debe ser un recorrido del caballo. Sorprendentemente, los procedimientos heurísticos buenos para el PAV no tienen dificultad en encontrar los recorridos del caballo, pero encontrar uno «a mano» es un reto.

El algoritmo ávido para el PAV que se tiene en mente es una variante del algoritmo de Kruskal. Como en dicho algoritmo, primero se considerarán las aristas más cortas. En el algoritmo de Kruskal, se acepta una arista en su turno si no forma un ciclo con las que ya han sido tomadas; en caso de no cumplir esto, rechaza la arista. Para el PAV, el criterio de aceptación es que una arista sometida a consideración, junto con las que ya se han seleccionado,

1. no haga que un vértice tenga grado 3 o mayor, y
2. no forme un ciclo, a menos que el número de aristas seleccionadas sea igual al número de vértices del problema.

Las colecciones de aristas seleccionadas con ese criterio, formarán una colección de caminos no conexos, hasta el último paso, cuando el único camino existente esté cerrado y forme un recorrido.

En la figura 10.15(a), se puede tomar primero la arista (d, e) , puesto que es la más corta, con longitud 3. Después, las aristas (b, c) , (a, b) y (e, f) , que tienen longitud 5. No importa en qué orden se tomen, todas cumplen con las condiciones de selección, y han de seleccionarse si se quiere seguir el enfoque ávido. La siguiente arista más corta es (a, c) , con longitud 7.08. Sin embargo, esta arista puede formar un ciclo con (a, b) y (b, c) , por lo que se rechaza. La arista (d, f) es la siguiente que se rechaza por motivos similares. La arista (b, e) es la siguiente a considerar, pero debe rechazarse porque puede aumentar los grados de los vértices b y e a 3, y nunca formaría un recorrido con lo seleccionado. De modo similar, se rechaza (b, d) . A continuación se considera (c, d) , que sí se acepta. Ahora existe un camino, $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f$, y por último se acepta (a, f) para completar el recorrido. El recorrido resultante está en la figura 10.15(b), y es el cuarto mejor de todos los posibles, pero es más costoso que el óptimo en menos del 4%. □

10.4 Método de retroceso (*backtracking*)

Algunas veces surge el problema de encontrar una solución óptima a un subproblema, pero sucede que no existe una teoría que pueda aplicarse para ayudar a encontrar lo óptimo, si no es recurriendo a una búsqueda exhaustiva. Esta sección se de-

dica a una técnica de búsqueda exhaustiva sistemática conocida como método de retroceso (*backtracking*) y una técnica llamada poda alfa-beta, que suele reducir mucho la búsqueda.

Considérese un juego como el ajedrez, damas o tres en raya (gato), donde hay dos jugadores. Los jugadores alternan las jugadas, y el estado del juego puede representarse por una posición del tablero. Se hace la suposición de que hay un número finito de posiciones del tablero y que el juego tiene algún tipo de regla para asegurar la terminación. Con cada uno de esos juegos se asocia un árbol llamado *árbol de juego*. Cada nodo del árbol representa una posición en el tablero, y se asocia la raíz con la posición de partida. Si la posición del tablero x se asocia con el nodo n , los hijos de n corresponderán al conjunto de movimientos posibles desde la posición x del tablero, y cada hijo se asociará con la posición resultante. Por ejemplo, la figura 10.16 muestra parte del árbol del juego de tres en raya.

Las hojas del árbol corresponden a las posiciones del tablero donde ya no existen movimientos posibles, porque uno de los jugadores ha ganado o porque todos los cuadros están ocupados y se produjo un empate. Se asocia un valor con cada nodo del árbol, y primero se asignan valores a las hojas. Si el juego es tres en raya, una hoja tendrá un valor -1 , 0 ó 1 , dependiendo de si la posición del tablero corresponde a una derrota, empate o victoria para el jugador 1 (jugando con X).

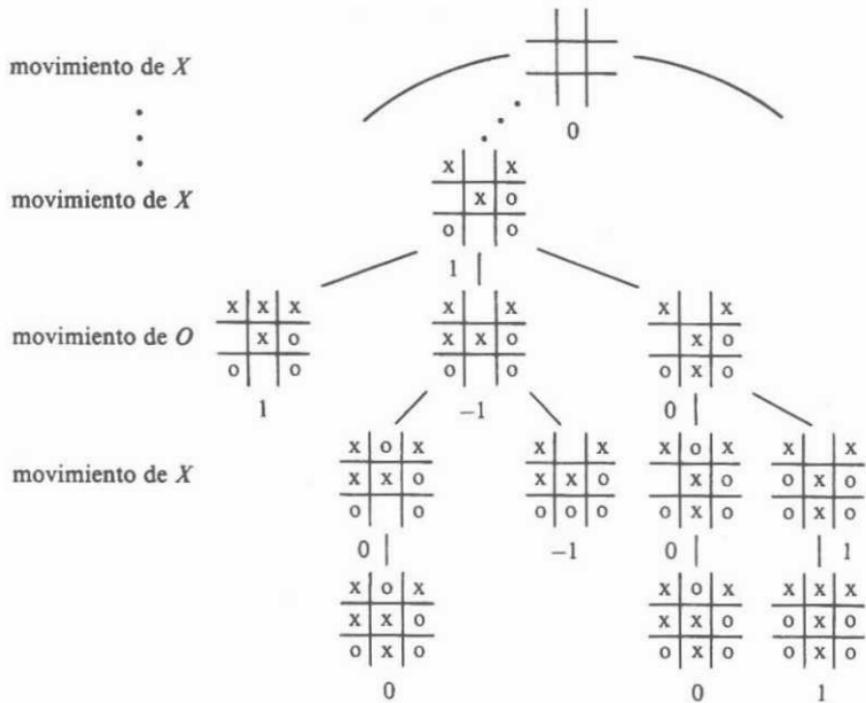


Fig. 10.16. Parte del árbol del juego de tres en raya.

Los valores se propagan hacia arriba en el árbol de acuerdo con la siguiente regla: si un nodo corresponde a una posición del tablero donde hay movimiento del jugador 1, el valor será el máximo de los valores de los hijos de ese nodo. Esto es, se supone que el jugador 1 hará el movimiento más favorable para él, o sea, el que produce el resultado con mayor valor. Si el nodo corresponde al movimiento del jugador 2, entonces el valor será el mínimo de los valores de los hijos. Es decir, se presume que el jugador 2 hará su movimiento más favorable, consiguiendo, de ser posible, una derrota para el jugador 1, o bien un empate como siguiente preferencia.

Ejemplo 10.6. Los valores de los tableros se encuentran en la figura 10.16. La hojas que significan triunfos para O tienen el valor -1 , los empates toman el 0 , y los triunfos de X tienen $+1$. En seguida se procede hacia arriba en el árbol. En el nivel 8, donde sólo queda un cuadro vacío, y es un movimiento de X , los valores de los tableros no resueltos son el «máximo» del hijo en el nivel 9.

En el nivel 7, correspondiente a un movimiento de O , hay dos opciones y se toma como valor para un nodo interno el mínimo de los valores de sus hijos. El tablero de la extrema izquierda, mostrado en el nivel 7, es una hoja y tiene un valor 1 , porque corresponde a un triunfo de X . El segundo tablero del mismo nivel tiene un valor $\min(0, -1) = -1$, mientras que el tercero tiene valor $\min(0, 1) = 0$. El único tablero mostrado en el nivel 6, que corresponde a un movimiento de X , tiene un valor $\max(1, -1, 0) = 1$, lo que significa que existe una selección que X puede hacer para ganar; en tal caso, el triunfo es inmediato. \square

Nótese que si la raíz tiene valor 1 , el jugador 1 tiene una estrategia ganadora; cualquiera que sea su turno, tiene la garantía de seleccionar un movimiento que le dé una posición del tablero con valor 1 . Cuando el turno corresponde al jugador 2, no tiene más opción que seleccionar un movimiento que deje una posición del tablero con valor 1 , lo cual significa una derrota para él. El hecho de que se suponga que el juego termina, garantiza que el primer jugador podrá ganar. Si la raíz tiene valor 0 , como sucede en el juego tres en raya, ningún jugador tiene una estrategia ganadora, pero por lo menos puede garantizarse un empate si juega lo mejor posible. Si la raíz tiene valor -1 , entonces el jugador 2 tiene una estrategia ganadora.

Funciones de utilidad

La idea de un árbol de juego, donde los nodos tienen valores -1 , 0 y 1 , puede generalizarse a árboles donde a las hojas se les da un número cualquiera (llamado la *utilidad*) como valor, y se aplican las mismas reglas de evaluación de nodos interiores: toma el máximo de los hijos en los niveles donde corresponda un movimiento del jugador 1, y el mínimo de los hijos en los niveles donde mueve el jugador 2.

Como ejemplo de donde son provechosas las utilidades generales, considérese un juego complejo como el ajedrez, donde el árbol de juego, aunque finito, es tan inmenso que la evaluación de abajo arriba no es posible. Los programas de ajedrez operan construyendo el árbol de juego con ese tablero como raíz para cada posición del tablero desde la cual se debe mover extendiéndose hacia abajo por varios niveles; el número exacto de niveles depende de la velocidad con que el computador puede tra-

bajar. Puesto que la mayor parte de las hojas del árbol son ambiguas, no indican triunfos, derrotas, ni empates, cada programa usa una función de las posiciones del tablero que intenta evaluar la probabilidad de que el computador gane desde esa posición. Por ejemplo, la diferencia entre las piezas afecta en gran medida tal estimación, al igual que algunos factores, como el poder defensivo alrededor de los reyes. Al usar esta función de utilidad, el computador puede estimar la probabilidad de un triunfo después de hacer cada uno de sus siguientes movimientos posibles en el supuesto de que se hará la mejor jugada siguiente de cada lado, y se escogerá el movimiento de mayor utilidad †.

Realización de la búsqueda con retroceso

Supóngase que se proporcionan las reglas de un juego ‡, esto es, sus movimientos válidos y reglas de terminación. Se desea construir su árbol de juego y evaluar la raíz, en la manera más obvia, y después visitar los nodos en orden final. El recorrido en orden final asegura que se visita un nodo interior n después de sus hijos, y entonces es posible evaluar n , tomando el mínimo o el máximo de los valores de sus hijos, lo que sea más apropiado.

El espacio de almacenamiento del árbol puede ser demasiado grande, pero con cuidado nunca se necesitará almacenar más de un camino, desde la raíz hasta un nodo, en cualquier momento. En la figura 10.17 se presenta el esbozo de un programa recursivo que representa el camino en el árbol por medio de la secuencia de las llamadas a procedimientos activos en cualquier instante. Ese programa supone lo siguiente:

1. Las utilidades son números reales en un intervalo limitado, por ejemplo -1 a $+1$.
2. La constante ∞ es mayor que cualquier utilidad positiva y su negativo es menor que cualquier utilidad negativa.
3. El tipo tipo_modo se declara

```
type
  tipo_modo = (MIN, MAX)
```

4. Existe un tipo tipo_tablero declarado de alguna forma adecuada para la representación de posiciones en el tablero.

† Inicialmente, otras cosas que hacen los programas para jugar al ajedrez son las siguientes.

1. Usar técnicas heurísticas para eliminar la consideración de ciertos movimientos que tal vez no serán buenos. Esto ayuda a expandir el árbol a más niveles en un tiempo dado.
2. Expandir «cadenas de captura» que son secuencias de movimientos de captura más allá del último nivel al que se expande el árbol normalmente. Esto ayuda a estimar con mayor exactitud la fuerza material relativa de las posiciones.
3. Podar el árbol de búsqueda por medio de la técnica alfa-beta, como se analizará más adelante en esta sección.

‡ Esto no implica que de esta forma sólo se puedan resolver «juegos». Como se verá en ejemplos posteriores, el «juego» puede representar en realidad la solución de un problema práctico.

5. Existe una función UTILIDAD que calcula la utilidad de cualquier tablero que sea una *hoja* (esto es, situación de triunfo, derrota o empate).

```

function busca ( B: tipo_tablero; modo: tipo_modo ) : real;
  { evalúa la utilidad del tablero B, en el supuesto
    de que es el movimiento del jugador 1 si modo = MAX, y es
    el movimiento del jugador 2 si modo = MIN. Devuelve la utilidad }
  var
    C: tipo_tablero; { un hijo del tablero B }
    valor: real; { valor mínimo o máximo temporal }
  begin
    (1)   if B es una hoja then
        return (utilidad(B))
    (2)   else begin
        { asigna el valor inicial mínimo o máximo de los hijos }
        (3)   if modo = MAX then
            valor := -∞
        (4)   else
            valor := ∞;
        (5)   for cada hijo C del tablero B do
            (6)     if modo = MAX then
                (7)       valor := máx(valor, busca(C, MIN))
            (8)     else
                (9)       valor := mín(valor, busca(C, MAX));
        (10)   return (valor)
    end
  end; { busca }

```

Fig. 10.17. Programa de búsqueda recursiva con el método de retroceso.

Otra implantación a considerar es por medio de un programa no recursivo que contenga una pila de tableros correspondientes a la secuencia de llamadas activas de *busca*. Las técnicas estudiadas en la sección 2.6 pueden usarse para construir dicho programa.

Poda alfa-beta

Hay una observación simple que permite dejar de considerar buena parte de un árbol de juego típico. Partiendo de la figura 10.17, el ciclo **for** de la línea (6) puede saltar ciertos hijos, con frecuencia muchos. Supóngase que se tiene un nodo *n*, como en la figura 10.18, y ya se ha determinado que *c*₁, el primero de los hijos de *n*, tiene valor 20. Como *n* es un nodo máx, se sabe que su valor es por lo menos 20. Ahora, supóngase que al continuar con la búsqueda se observa que *d*, un hijo de *c*₂, tiene valor 15. Como *c*₂, otro hijo de *n*, es un nodo mín, se sabe que el valor de *c*₂ no pue-

de exceder de 15. Así, cualquiera que sea el valor c_2 no afecta al valor de n ni de cualquier parente de n .

Así, en la situación de la figura 10.18, es posible ignorar la consideración de los hijos de c_2 que aún no se han examinado. Las reglas generales para pasar por alto o «podar» nodos implican la noción de valores tentativos y finales para los nodos. El valor *final* es el que se ha denominado hasta ahora simplemente «valor». Un valor

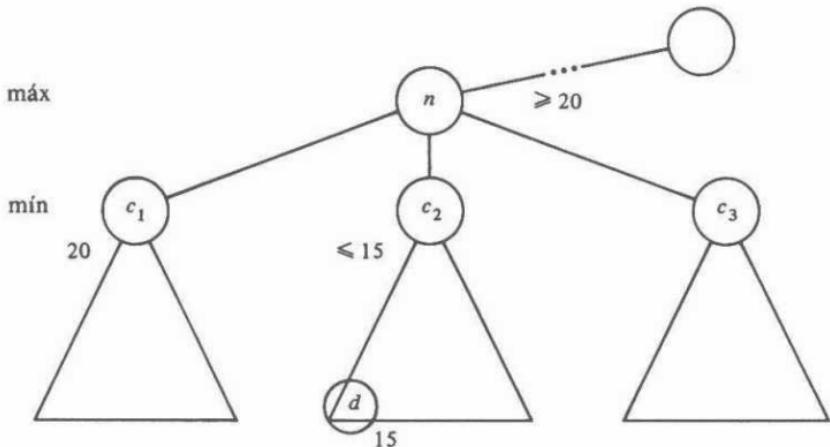


Fig. 10.18. Poda de los hijos de un nodo.

tentativo es una cota superior al valor del nodo *mín*, o una cota inferior al valor del nodo *máx*. Las reglas para el cálculo de los valores finales y tentativos son las siguientes.

1. Si todos los hijos de un nodo n se consideraron o se podaron, conviértase el valor tentativo de n en final.
2. Si un nodo *máx* n tiene valor tentativo v_1 y un hijo con valor final v_2 , entonces el valor tentativo de n se hace $\max(v_1, v_2)$. Si n es un nodo *mín*, se asigna su valor tentativo a $\min(v_1, v_2)$.
3. Si p es un nodo *mín* con parente q (un nodo *máx*), y p y q tienen valores tentativos v_1 y v_2 , respectivamente, con $v_1 \leq v_2$, entonces se pueden podar todos los hijos no considerados de p . También se pueden podar los hijos no considerados de p si p es un nodo *máx* (y por tanto, q es un nodo *mín*) y $v_2 \leq v_1$.

Ejemplo 10.7. Considérese el árbol de la figura 10.19. Suponiendo los valores mostrados en las hojas, se desea calcular el valor de la raíz; se empieza con un recorrido en orden final. Después de alcanzar el nodo D , por la regla (2) se asigna un valor tentativo de 2 al nodo C , que es el valor final de D . Entonces se explora E y se vuelve a C y después a B . Por la regla (1), el valor final de C se fija en 2 y el valor de B se hace tentativamente 2. La búsqueda desciende hasta G y después regresa a F . El valor de F se hace tentativamente 6. Por la regla (3), con p y q iguales a F y B , respec-

tivamente, se puede podar H . Esto es, no se necesita explorar el nodo H , ya que el valor tentativo de F nunca puede decrecer y ya es mayor que el valor de B , que no puede aumentar.

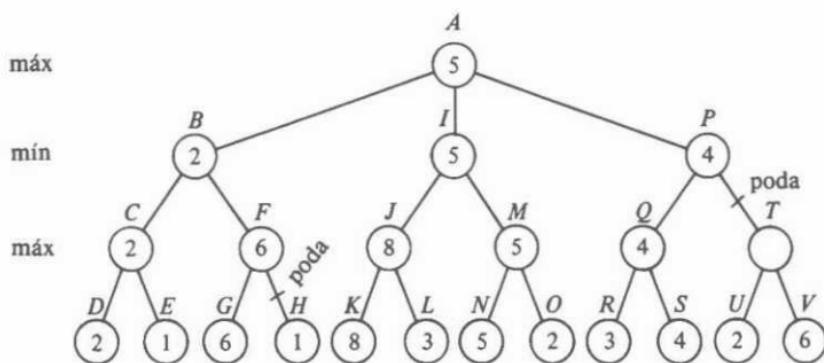


Fig. 10.19. Árbol de juego.

Continuando con el ejemplo, A tiene asignado un valor tentativo de 2 y la búsqueda prosigue hasta K . J tiene asignado un valor tentativo de 8. L no determina el valor del nodo máx J . I recibe un valor tentativo de 8. La búsqueda sigue hacia abajo hasta N , y a M se le asigna un valor tentativo de 5. El nodo O debe ser explorado, ya que 5, el valor tentativo de M , es menor que el valor tentativo de I . Se revisan los valores tentativos de I y A y la búsqueda continúa hasta R . Finalmente, se exploran R y S , y P recibirá un valor tentativo de 4. No es necesario buscar en T ni más abajo, ya que eso sólo disminuiría el valor de P , que ya es demasiado pequeño para afectar al valor de A . \square

Búsqueda de ramificación y acotamiento

Los juegos no son la única clase de «problemas» que pueden resolverse explorando exhaustivamente un árbol completo de posibilidades. Hay muchos problemas que exigen encontrar una configuración mínima o máxima de alguna clase que son susceptibles de resolución por medio de la búsqueda de retroceso de un árbol de todas las posibilidades. Los nodos del árbol pueden considerarse como conjuntos de configuraciones y los hijos de un nodo n representan cada uno un subconjunto de las configuraciones que n representa. Finalmente, cada hoja representa configuraciones sencillas o soluciones al problema, y se puede evaluar cada una de las configuraciones con el fin de saber si es la mejor solución encontrada hasta ese momento.

Si se es hábil en el diseño de la búsqueda, los hijos de un nodo representarán muchas menos configuraciones que el nodo mismo, así que no es necesario profundizar mucho para alcanzar las hojas. A fin de evitar que esta noción de búsqueda parezca muy vaga, he aquí un ejemplo concreto.

Ejemplo 10.8. Recuérdese de la sección anterior el problema del agente viajero, en el que se dio un «algoritmo ávido» para encontrar un buen recorrido, aunque no necesariamente el óptimo. Ahora se analiza cómo encontrar el recorrido óptimo por medio de la consideración sistemática de todos los recorridos. Una forma es tener en cuenta todas las permutaciones de los nodos y evaluar los recorridos que visitan los nodos en ese orden, recordando el mejor encontrado hasta el momento. El tiempo para esta aproximación es $O(n!)$ en un grafo con n nodos, ya que se deben considerar $(n-1)!$ permutaciones distintas †, y cada permutación toma un tiempo $O(n)$.

Se presentará un enfoque un poco distinto que no es mejor que el anterior en el peor caso, pero en el promedio, cuando se acopla con una técnica llamada «ramificación y acotamiento» —que se estudiará a continuación— produce la respuesta mucho más rápido que el método de «intentar con todas las permutaciones». Se empieza construyendo un árbol, con una raíz que representa todos los recorridos. Los recorridos son lo que se denominó «configuraciones» en el material preliminar. Cada nodo tiene dos hijos, y los recorridos que representa un nodo están divididos por esos dos hijos en dos grupos: los que tienen una arista particular y los que no. Por ejemplo, la figura 10.20 muestra porciones del árbol para el caso del PAV de la figura 10.15(a).

En la figura 10.20 se han considerado las aristas en orden lexicográfico (a, b) , (a, c) , ..., (a, f) , (b, c) , ..., (b, f) , (c, d) , y así sucesivamente. Claro está que se puede tomar cualquier otro orden. Obsérvese que no todos los nodos del árbol tienen dos hijos. Se pueden eliminar algunos hijos, porque las aristas seleccionadas no forman parte de un recorrido. Así, no hay nodo para los «recorridos que contienen (a, b) , (a, c) y (a, d) », porque a tendría grado 3 y el resultado no sería un recorrido. De forma semejante, al ir descendiendo en el árbol, se eliminan algunos nodos, porque alguna ciudad puede tener grado menor que 2. Por ejemplo, no habrá nodos para recorridos sin (a, b) , (a, c) , (a, d) o (a, e) . □

Procedimientos heurísticos necesarios para ramificación y acotamiento

Al utilizar ideas similares a las de la poda alfa-beta, se eliminan muchos más nodos del árbol de búsqueda que los mostrados en el ejemplo 10.8. Supóngase, para ser específicos, que el problema es minimizar alguna función, como el costo de un recorrido en el PAV. Supóngase, también, que se tiene un método para obtener una cota inferior en el costo de cualquier solución entre las del conjunto de soluciones representado por algún nodo n . Si la mejor solución encontrada hasta ahora cuesta menos que la cota inferior para el nodo n , no es necesario explorar ninguno de los nodos por debajo de n .

Ejemplo 10.9. Se revisará una forma de alcanzar una cota inferior en ciertos conjuntos de soluciones para el PAV, los representados por nodos en un árbol de solu-

† Obsérvese que no es necesario considerar todas las $n!$ permutaciones, ya que el punto de partida de un recorrido no importa. Por tanto, se consideran sólo aquellas permutaciones que empiecen con 1.

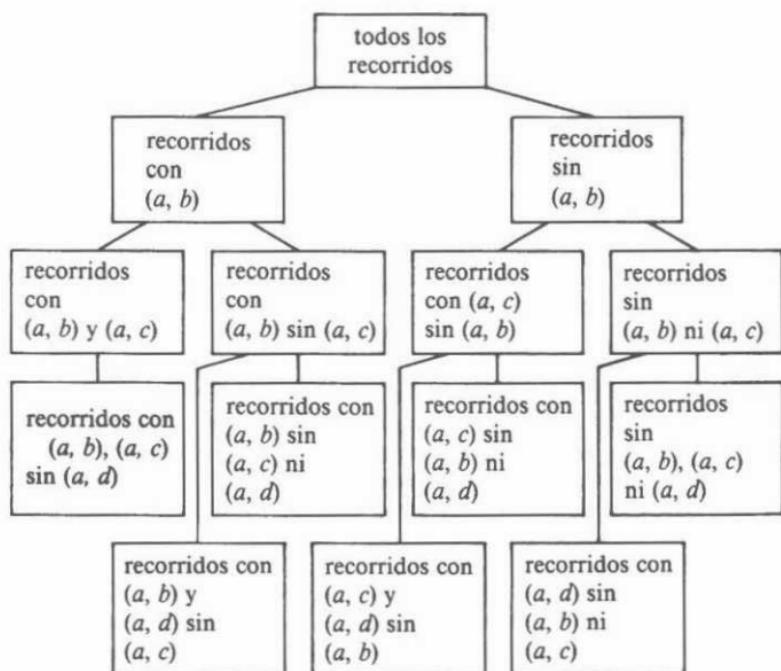


Fig. 10.20. Principio de un árbol solución para un caso del PAV.

ciones, como se sugirió en la figura 10.20. Primero, supóngase que se desea una cota inferior para todas las soluciones de un caso dado del PAV. Obsérvese que el costo de cualquier recorrido puede expresarse como la semisuma, sobre todos los nodos n , del costo de las dos aristas del recorrido adyacentes a n . Esta observación da lugar a la siguiente regla: la suma de las dos aristas de recorrido adyacentes al nodo n no es menor que la suma de las dos aristas de costo menor adyacentes a n . Así, ningún recorrido puede costar menos que la semisuma sobre todos los nodos n de las dos aristas de menor costo que inciden sobre n .

Por ejemplo, considérese el caso del PAV que se muestra en la figura 10.21. A diferencia del caso de la figura 10.15, la medida de la distancia de las aristas no es euclíadiana, es decir, no tiene relación con la distancia en el plano entre las ciudades que conecta. Tal medida del costo puede ser el tiempo o la tarifa del viaje, por ejemplo. En este caso, las aristas de menor costo adyacentes al nodo a son (a, d) y (a, b) , con un costo total de 5. Para el nodo b , se encuentran (a, b) y (b, e) con un costo total de 6. De igual forma, las dos aristas de menor costo adyacentes a c , d y e , totalizan 8, 7 y 9, respectivamente. Así pues, la cota inferior del costo de un recorrido es $(5+6+8+7+9)/2 = 17.5$.

Ahora, supóngase que se pide una cota inferior en el costo de un subconjunto de recorridos definidos por algún nodo en el árbol de búsqueda. Si el árbol de búsqueda se construye como en el ejemplo 10.8, cada nodo representa recorridos definidos

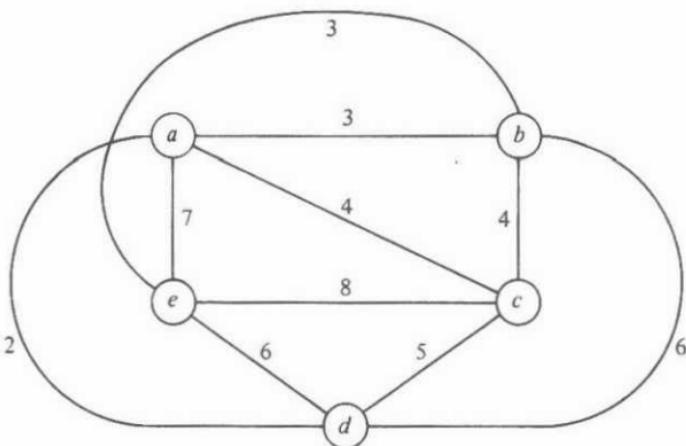


Fig. 10.21. Un caso del PAV.

por un conjunto de aristas que deben estar en el recorrido y un conjunto de aristas que tal vez no estén. Esas restricciones alteran las opciones para escoger las dos aristas de menor costo en cada nodo. Es evidente que una arista restringida a permanecer en cualquier recorrido debe incluirse entre las dos aristas seleccionadas, sin importar si son del menor o segundo menor costo o no †. Asimismo, una arista restringida a estar fuera no puede seleccionarse, aunque su costo sea menor.

Así, si existe la restricción de incluir la arista (a, e) y excluir (b, c) , las dos aristas para el nodo a son (a, d) y (a, e) , con un costo total de 9. Para b , se seleccionan (a, b) y (b, e) , igual que antes, con un costo total de 6. Para c , no es posible elegir (b, c) , y se eligen (a, c) y (c, d) , con un costo total de 10. Para d , se seleccionan (a, d) y (c, d) , como antes, mientras que para e , se deben escoger (a, e) y (b, e) . La cota inferior para esas restricciones es $(9+6+9+7+10)/2 = 20.5$. □

Ahora se construye el árbol de búsqueda a partir de lo sugerido en el ejemplo 10.8. Se consideran las aristas en orden lexicográfico, como en ese ejemplo. Por cada ramificación, al considerar los dos hijos de un nodo, se intenta inferir decisiones adicionales sobre las aristas que deben incluirse o excluirse de los recorridos representados por esos nodos. Las reglas utilizadas para esas inferencias son:

1. Si la exclusión de una arista (x, y) hiciera imposible para x o y tener dos aristas adyacentes en el recorrido, entonces debe incluirse (x, y) .
2. Si incluir (x, y) hace que x o y tengan más de dos aristas adyacentes en el recorrido, o que se complete un ciclo que no sea recorrido con las aristas ya incluidas, entonces debe excluirse (x, y) .

† Se verá que las reglas para la construcción del árbol de búsqueda eliminan cualquier conjunto de restricciones que no puedan producir un recorrido, por ejemplo, porque se requieran tres aristas adyacentes a un nodo para formar parte del recorrido.

Al ramificar, después de hacer todas las inferencias posibles, se calculan las cotas inferiores de ambos hijos. Si la cota inferior de un hijo es tan grande o mayor que el recorrido de costo menor encontrado hasta ese momento, se «poda» ese hijo y no se construyen ni consideran sus descendientes. Curiosamente, hay situaciones donde la cota inferior de un nodo n es menor que la mejor solución existente encontrada, y aún así ambos hijos de n se pueden podar, porque sus cotas inferiores exceden del costo de la mejor solución encontrada hasta el momento.

Si no es posible podar ningún hijo, como regla heurística se considerará primero el hijo con la cota inferior más pequeña, con la esperanza de alcanzar más rápido una solución más económica que la mejor encontrada hasta el momento †. Después de considerar un hijo, se debe evaluar otra vez si puede podarse su hermano, ya que pudo haberse encontrado una mejor solución nueva. Para el caso de la figura 10.21, se obtiene árbol de la figura 10.22. Para interpretar los nodos de ese árbol, es útil entender que las letras mayúsculas son los nombres de los nodos del árbol de búsqueda; los números son las cotas inferiores, y se listan las restricciones aplicadas al nodo, pero no sus antecesores, escribiendo xy si la arista (x, y) debe incluirse, y \bar{xy} , si (x, y) debe excluirse. Obsérvese también que las restricciones introducidas en un nodo se aplican a todos sus descendientes. Así, para obtener todas las restricciones aplicables en un nodo debe seguirse el camino desde ese nodo hasta la raíz.

Por último, se hace hincapié en que, como para la búsqueda de retroceso general, se construye el árbol nodo por nodo, reteniendo sólo un camino, como en el algoritmo recursivo de la figura 10.17, o su contraparte no recursiva. Quizá se prefiera la versión no recursiva, para mantener dentro de una pila la lista de restricciones.

Ejemplo 10.10. La figura 10.22 muestra el árbol de búsqueda para el caso del PAV de la figura 10.22. Con el fin de ver cómo se construye, se parte de la raíz A de la figura 10.22. La primera arista en orden lexicográfico es (a, b) , así que se consideran los hijos B y C , correspondientes a las restricciones ab y \bar{ab} , respectivamente. De momento no hay una «mejor solución hasta aquí», por lo que se considerarán B y C ‡. Forzar la inclusión de (a, b) no sube la cota inferior, pero excluir tal arista produce la cota 18.5, ya que las dos aristas válidas más económicas para los nodos a y b totalizan 6 y 7, respectivamente, comparadas con 5 y 6 sin restricciones. Siguiendo el procedimiento heurístico, se considerarán primero los descendientes del nodo B .

La siguiente arista en orden lexicográfico es (a, c) . Se introducen los hijos D y E correspondientes a los recorridos en que (a, c) está incluida y excluida, respectivamente. En el nodo D , se infiere que ni (a, d) ni (a, e) pueden estar en un recorrido; de otra forma, a puede tener demasiadas aristas incidentes. De acuerdo con el procedimiento heurístico, se analiza E antes que D , para ramificar sobre la arista (a, d) . Los hijos F y G se introducen con cotas inferiores 18 y 23, respectivamente. Para cada uno de esos hijos se conocen unas 3 aristas incidentes sobre a , por lo que se puede deducir algo acerca de la arista restante (a, e) .

† Una alternativa es usar un procedimiento heurístico para obtener una buena solución con las restricciones requeridas para cada hijo. Por ejemplo, se deberá poder modificar el algoritmo ávido del PAV para respetar las restricciones.

‡ Se puede empezar con alguna solución encontrada heurísticamente, como la ávida, aunque eso no afecte a nuestro ejemplo. La solución ávida para la figura 10.21 tiene un costo de 21.

Considérense primero los hijos de F . La primera arista restante en orden lexicográfico es (b, c) , y si se incluye, al haber incluido ya (a, b) , no se pueden incluir (b, d) ni (b, e) . Al haber eliminado (a, e) y (b, e) , se deben tener (c, e) y (d, e) . No se puede tener (c, d) , porque entonces c y d tendrían 3 aristas incidentes. Queda, por último un recorrido (a, b, c, e, d, a) , cuyo costo es 23. Igualmente, el nodo I , donde (b, c) está excluida, puede probarse que representa sólo el recorrido (a, b, e, c, d, a) , de costo 21. Este recorrido tiene el menor costo encontrado hasta el momento.

Ahora se retrocede a E y se considera su segundo hijo, G . Pero G tiene una cota inferior de 23, la cual excede al mejor costo actual, 21. Así, se poda G . Ahora, se retrocede hasta B y se considera su otro hijo, D . La cota inferior en D es 20.5, pero debido a que los costos son enteros, se sabe que ningún recorrido representado por D tiene costo menor que 21. Dado que ya existe un recorrido tan económico, no es necesario explorar los descendientes de D y, por tanto, se poda. Ahora, se retrocede hasta A y se considera su segundo hijo, C .

En el nivel del nodo C , sólo se ha considerado la arista (a, b) . Los nodos J y K se introducen como hijos de C . J corresponde a aquellos recorridos que tienen a (a, c) , pero no a (a, b) , y su cota inferior es 18.5. K corresponde a los recorridos que no contienen (a, b) ni (a, c) , y se infiere que esos recorridos tienen a (a, d) y (a, e) . La cota inferior de K es 21 y se puede podar de inmediato, dado que ya se conoce un recorrido de menor costo.

A continuación se consideran los hijos en J , L y M , y se poda M porque su cota inferior excede del costo del mejor recorrido actual. Los hijos de L son N y P , correspondientes a recorridos que tienen (b, c) , y que excluyen a (b, c) . Considerando el grado de los nodos b y c , y recordando que las aristas seleccionadas no pueden formar un ciclo con menos de cinco ciudades, se infiere que los nodos N y P representan recorridos individuales. Uno de esos, (a, c, b, e, d, a) , tiene menor costo que cualquier otro, 19. Ya se ha explorado o podado todo el árbol y, por tanto, se ha terminado. □

10.5 Algoritmos de búsqueda local

Algunas veces, la siguiente estrategia producirá una solución óptima para un problema.

1. Empezar con una solución aleatoria.
2. Aplicar a la solución actual una transformación de un conjunto dado de transformaciones para mejorar la solución; la mejora es la nueva solución «actual».
3. Repetir hasta que ninguna transformación del conjunto pueda mejorar la solución actual.

La solución resultante puede ser óptima o no. En principio, si «el conjunto de transformaciones dado» incluye todas las que toman una solución y la reemplazan por otra, entonces no se parará hasta alcanzar la solución óptima. Pero, en ese caso, el tiempo para aplicar (2) es el mismo que el necesario para examinar todas las soluciones, y todo el enfoque no tiene mucho sentido.

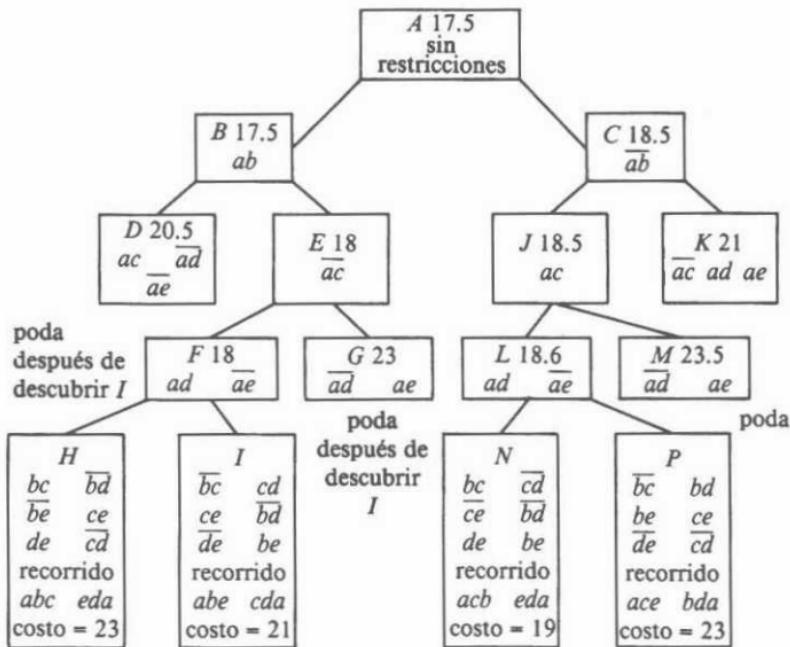


Fig. 10.22. Árbol de búsqueda para la solución del PAV.

El método tiene sentido cuando se puede restringir el conjunto de transformaciones a un conjunto pequeño, de modo que en poco tiempo se puedan considerar todas las transformaciones posibles; tal vez deben permitirse las $O(n^2)$ u $O(n^3)$ transformaciones cuando el problema es de «tamaños n . Si el conjunto de transformaciones es pequeño, es natural considerar las soluciones, que pueden ser transformadas entre sí en un paso como «cercanas». Las transformaciones se llaman «transformaciones locales», y el método se conoce como *búsqueda local*.

Ejemplo 10.11. Un problema que se puede resolver con exactitud mediante búsqueda local es el de los árboles abarcadores minimales. Las transformaciones locales son aquellas en las que se toma alguna arista que no se encuentra en el árbol abarcador actual, se agrega al árbol, que debe producir un ciclo único, y después eliminar exactamente una arista del ciclo (probablemente la de costo mayor) para formar un árbol nuevo.

Por ejemplo, considérese el grafo de la figura 10.21. Se empieza con el árbol mostrado en la figura 10.23(a). Una transformación que se puede realizar es agregar la arista (d, e) y quitar otra del ciclo formado, el cual es (e, a, c, d, e) . Si se elimina la arista (a, e) , se reduce el costo del árbol de 20 a 19. Esta transformación puede hacerse dejando el árbol de la figura 10.23(b), al cual se intenta aplicar de nuevo una transformación de mejora, como insertar la arista (a, d) y eliminar la (c, d) del ciclo

formado. El resultado se muestra en la figura 10.23(c). Después es posible introducir (a, b) y eliminar (b, c) , como en la figura 10.23(d), y más tarde introducir (b, e) en favor de (d, e) . El árbol resultante de la figura 10.23(e) es minimal. Se puede comprobar si cada arista ausente en ese árbol tiene costo mayor que cualquier otra arista dentro del ciclo que forma. Así, ninguna transformación es aplicable a la figura 10.23(c). \square

El tiempo requerido por el algoritmo del ejemplo 10.11 en un grafo de n nodos y a aristas depende del número de veces que se necesite mejorar la solución. La sola prueba de que ninguna transformación es aplicable puede llevar un tiempo $O(na)$, ya que deben probarse a aristas, y cada una puede formar un ciclo de longitud cercana a n . Así, el algoritmo no es tan bueno como los algoritmos de Prim o de Kruskal, pero sirve como ejemplo de que puede obtenerse una solución óptima por búsquedas locales.

Algoritmos de aproximación por búsqueda local

Los algoritmos de búsqueda local han tenido gran efectividad como métodos heurísticos para la solución de problemas cuyas soluciones exactas requieren tiempo exponencial. Un método común de búsqueda es empezar con un número de soluciones aleatorias y aplicar las transformaciones locales a cada una, hasta alcanzar una solución *localmente óptima*, que ninguna transformación pueda mejorar. Con frecuencia se alcanzarán diferentes soluciones localmente óptimas, a partir de la mayor parte o de todas las soluciones iniciales aleatorias, como se sugiere en la figura

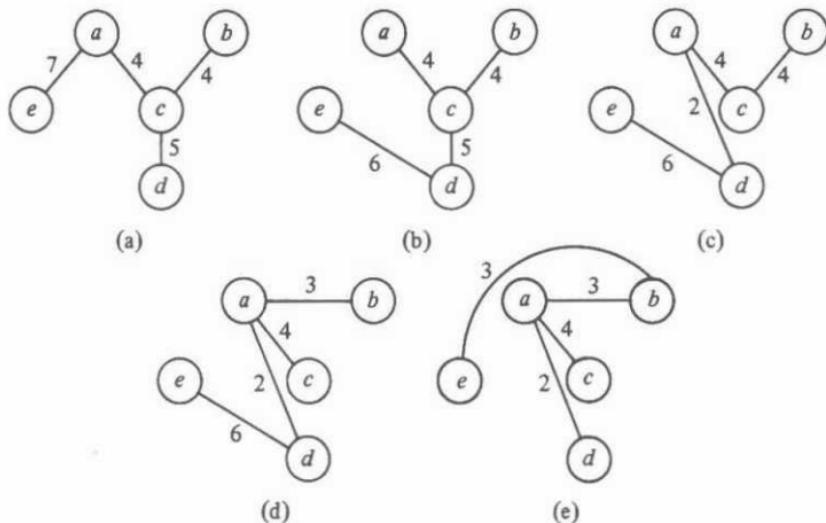


Fig. 10.23. Búsqueda local para un árbol abarcador minimal.

10.24. Si se tiene suerte, una de ellas será *globalmente optimal*, esto es, tan buena como cualquier otra solución.

En la práctica, tal vez no se pueda encontrar una solución globalmente óptima como la que se sugiere en la figura 10.24, ya que el número de soluciones localmente óptimas puede ser enorme. Sin embargo, es posible escoger por lo menos aquella solución localmente óptima que tenga el menor costo entre todas las que se encontraron. Como el número de clases de transformaciones locales utilizadas para resolver varios problemas es muy grande, se cerrará la sección con dos ejemplos: el PAV y un problema simple de «colocación de paquetes».

Problema del agente viajero

El PAV es uno de los problemas en que las técnicas de búsqueda local han sido muy satisfactorias. La transformación más simple que se ha usado se conoce como «opción doble». Consiste en tomar dos aristas cualesquiera, tales como (A, B) y (C, D) de la figura 10.25, quitándolas y reconectando sus extremos para formar un recorrido nuevo. En esta figura, el recorrido nuevo va desde B hasta C en el sentido de las manecillas del reloj, después sigue por la arista (C, A) , desde A hasta D en sentido contrario al de las manecillas del reloj y finalmente por la arista (D, B) . Si la suma de las longitudes de (A, C) y (B, D) es menor que la suma de las longitudes de (A, B) y (C, D) , se tendrá un recorrido mejorado †. Obsérvese que no se pueden conectar

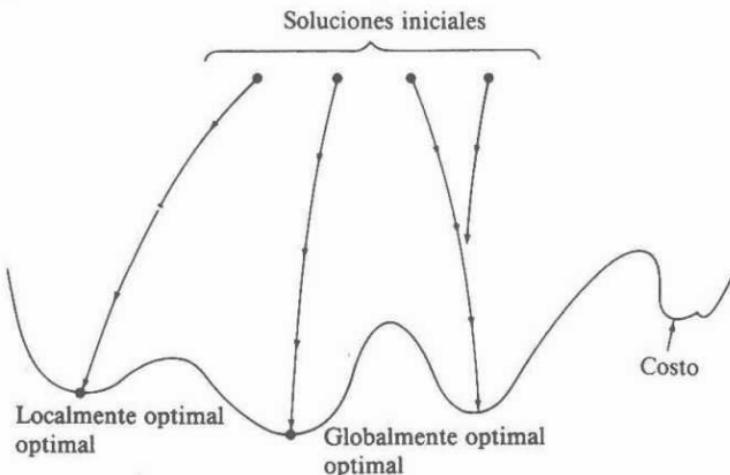


Fig. 10.24. Búsqueda local en el espacio de soluciones.

† No hay que dejarse engañar por el dibujo de la figura 10.25. Es cierto que si las longitudes de las aristas son distancias en el plano, las aristas con puntos deben ser mayores que las que reemplazan. De hecho, las opciones dobles en el plano mejoran el costo exactamente cuando las aristas eliminadas, (A, B) y (C, D) , se cruzan, mientras que las reemplazadas no lo hacen. Sin embargo, no existe motivo alguno para suponer que las distancias de la figura 10.25 sean distancias en un plano, o si lo son, puede suceder que se hayan cruzado (A, B) y (C, D) y no (A, C) y (B, D) .

A con *D* y *B* con *C*, pues, el resultado no sería un recorrido, sino dos ciclos disjuntos. Para encontrar un recorrido localmente óptimal, se comienza con un recorrido aleatorio, y se consideran todos los pares de aristas no adyacentes, como (*A*, *B*) y (*C*, *D*) de la figura 10.25. Si el recorrido puede mejorarse reemplazando esas aristas por (*A*, *C*) y (*B*, *D*), se hace así, y se prosigue considerando los pares de aristas que no se consideraron anteriormente. Obsérvese que las aristas introducidas (*A*, *C*) y (*B*, *D*) deben emparejarse cada una con todas las demás del recorrido, ya que pueden resultar mejoras adicionales.

Ejemplo 10.12. Reconsidérese la figura 10.21, para comenzar con el recorrido de la figura 10.26(a). Se reemplazan (*a*, *e*) y (*c*, *d*), cuyo costo total es 12, por (*a*, *d*) y (*c*, *e*), con un costo total de 1, como se muestra en la figura 10.26(b). Despues se sustituyen (*a*, *b*) y (*c*, *e*) por (*a*, *c*) y (*b*, *e*), dando el recorrido óptimo mostrado en la figura 10.26(c). Se puede comprobar que no haya ningún par de aristas que se pueda eliminar de esta figura y ser reemplazado con ventaja por aristas cruzadas con los mismos extremos. Como un caso adicional, (*b*, *c*) y (*d*, *e*) juntas tienen el costo relativamente alto de 10. Pero (*c*, *e*) y (*b*, *e*) son peores, al costar juntas 14. □

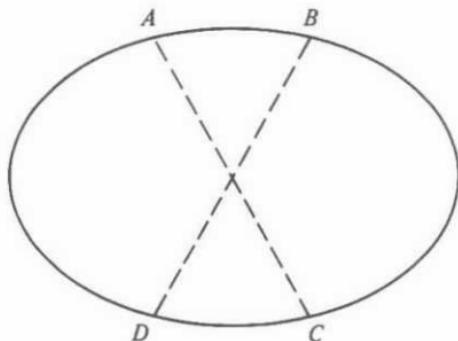


Fig. 10.25. Opción doble.

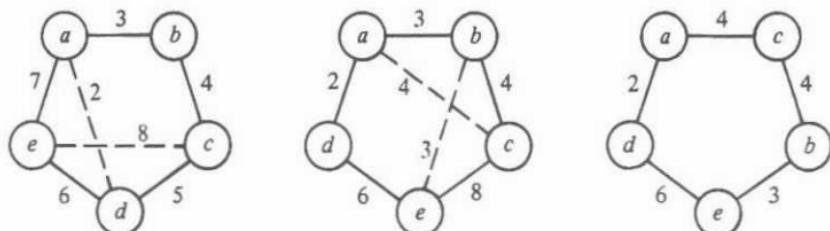


Fig. 10.26. Optimación de un caso del PAV con opción doble.

Se puede generalizar la opción doble a opción k para alguna k constante, donde se eliminan hasta k aristas y se reconectan los segmentos restantes en cualquier orden hasta producir un recorrido. Obsérvese que no se requiere que las aristas suprimidas no sean adyacentes en general, si bien para el caso de la opción doble no tenía sentido considerar la supresión de dos aristas adyacentes. Obsérvese también que para $k > 2$, existe más de una forma de conectar los segmentos. Por ejemplo, la figura 10.27 muestra el proceso general de opción triple utilizando cualquiera de los ocho conjuntos de aristas siguientes.

1. $(A, F), (D, E), (B, C)$ (como fue el recorrido)
2. $(A, F), (C, E), (D, B)$ (una opción doble)
3. $(A, E), (F, D), (B, C)$ (otra opción doble)
4. $(A, E), (F, C), (B, D)$ (una opción triple real)
5. $(A, D), (C, E), (B, F)$ (otra opción triple real)
6. $(A, D), (C, F), (B, E)$ (otra opción triple real)
7. $(A, C), (D, E), (B, F)$ (una opción doble)
8. $(A, C), (D, F), (B, E)$ (una opción triple)

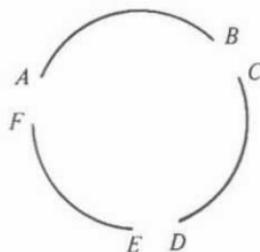


Fig. 10.27. Segmentos de un recorrido después de la supresión de tres aristas.

Es fácil verificar que, para una k fija, el número de diferentes transformaciones con opción k que se necesita considerar, si existen n vértices, es $O(n^k)$. Por ejemplo, el número exacto es $n(n-3)/2$ para $k = 2$. Sin embargo, el tiempo requerido para obtener un recorrido localmente óptimo puede ser mucho mayor que esto, ya que se pudieron haber realizado muchas transformaciones locales antes de alcanzar un recorrido localmente óptimo, y cada transformación de mejora introduce aristas nuevas que pueden participar en transformaciones posteriores que mejoran aún más el recorrido fijado. Lin y Kernighan [1973] han encontrado que la opción de profundidad variable es un método muy poderoso en la práctica, y tiene una buena oportunidad de obtener el recorrido óptimo en problemas de ciudades entre 40 y 100.

Colocación de paquetes

El problema de la colocación unidimensional de paquetes puede plantearse como sigue. Se tiene un grafo no dirigido, cuyos vértices se denominan «paquetes». Las aristas están etiquetadas por sus «pesos», y el peso $w(a, b)$ de la arista (a, b) es el nú-

mero de hilos existente entre los paquetes a y b . El problema es ordenar los vértices p_1, p_2, \dots, p_n de manera que la suma de $|i - j| w(p_i, p_j)$ en todos los pares i y j se minimice. Esto es, se desea minimizar la suma de las longitudes de los hilos necesarios para conectar todos los paquetes con el número requerido de hilos.

Al problema de la colocación de paquetes se le han dado muchas aplicaciones. Por ejemplo, los «paquetes» pueden ser tarjetas lógicas en un fichero, y el peso de una interconexión entre tarjetas es el número de hilos que las conectan. Un problema similar surge con el diseño de circuitos integrados a partir de disposiciones de módulos estándar e interconexiones entre ellos. Una generalización del problema de la colocación unidimensional de paquetes permite la colocación de los «paquetes», que tienen alto y ancho, en una región bidimensional, al tiempo que se minimiza la suma de las longitudes de los hilos entre los paquetes. Este problema también es aplicable en el diseño de circuitos integrados, entre otras áreas.

Hay ciertas transformaciones locales que se podrían utilizar para encontrar los óptimos locales para diversos casos del problema de la colocación unidimensional de paquetes. He aquí algunas:

1. Intercambiar los paquetes adyacentes p_i y p_{i+1} si el orden resultante es de costo menor. Sea $L(j)$ la suma de pesos de las aristas que se extienden hacia la izquierda de p_j , esto es, $\sum_{k=j+1}^n w(p_k, p_j)$. Igualmente, sea $R(j) \sum_{k=j+1}^n w(p_k, p_j)$. Se obtienen mejoras si $L(i) - R(i) + R(i+1) - L(i+1) + 2w(p_i, p_{i+1})$ es negativa. Sería útil verificar esta fórmula calculando los costos antes y después del intercambio y tomar la diferencia.
2. Tomar un paquete p_i e insertarlo entre p_i y p_{i+1} para algunas i y j .
3. Intercambiar dos paquetes cualesquiera p_i y p_j .

Ejemplo 10.13. Supóngase que se tiene el grafo de la figura 10.21 para representar un caso de la colocación de paquetes. Se considerará sólo conjunto de transformaciones simples (1). Una colocación inicial, a, b, c, d, e , se muestra en la figura 10.28(a); tiene un costo de 97. Obsérvese que la función de costo pesa las aristas por su distancia, de modo que (a, e) contribuye $4 \times 7 = 28$ al costo de 97. Considérese el intercambio de d y e . Se tiene que $L(d) = 13$, $R(d) = 6$, $L(e) = 24$, y $R(e) = 0$. Así, $L(d) - R(d) + R(e) - L(e) + 2w(d, e) = -5$, y se puede intercambiar d y e para mejorar la colocación a (a, b, c, e, d) , con un costo de 92, como se muestra en la figura 10.28(b).

En la figura 10.28(b), se intercambian c y e con cierta ventaja, produciendo la figura 10.28(c), cuya colocación (a, b, e, c, d) tiene un costo 91. La figura 10.28(c) es localmente óptima para el conjunto de transformaciones (1). No es globalmente óptima; (a, c, e, d, b) tiene un costo de 84. □

Como en el PAV, es difícil estimar exactamente el tiempo que lleva alcanzar un óptimo local. Sólo se observa que para el conjunto de transformaciones (1), hay sólo $n-1$ transformaciones a considerar. Más aún, si se calculan $L(i)$ y $R(i)$ una vez, tan sólo es necesario cambiarlas cuando p_i se intercambia con p_{i-1} o p_{i+1} . Además, calcular de nuevo es fácil. Si p_i y p_{i+1} se intercambian, por ejemplo, los nuevos $L(i)$ y

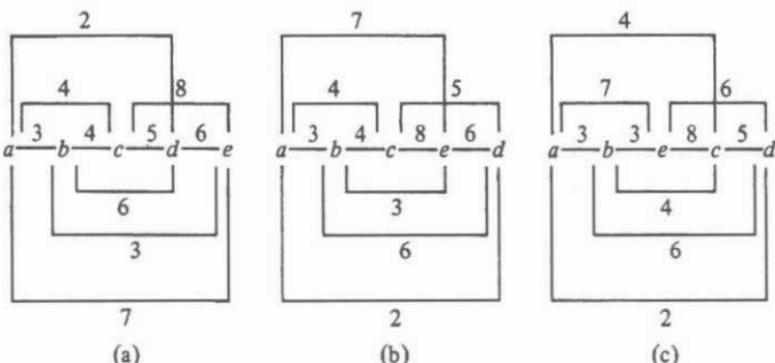


Fig. 10.28. Optimaciones locales.

$R(i)$ son, respectivamente, $L(i+1) - w(p_i, p_{i+1})$ y $R(i+1) + w(p_i, p_{i+1})$. Así, un tiempo $O(n)$ es suficiente para probar una transformación de mejora y calcular de nuevo los $L(i)$ y $R(i)$. También se requiere un tiempo $O(n)$ para asignar valores iniciales a $L(i)$ y $R(i)$, si se emplea la recurrencia

$$L(1) = 0$$

$$L(i) = L(i-1) + w(p_{i-1}, p_i)$$

y una recurrencia similar para R .

En comparación, los conjuntos de transformaciones (2) y (3) tienen, cada uno, $O(n^2)$ miembros. Por tanto, llevará un tiempo $O(n^2)$ tan sólo confirmar que se tiene una solución localmente óptima. Sin embargo, como sucedió con el conjunto (1), no es posible acotar con exactitud el tiempo total requerido al efectuar una secuencia de mejoras, ya que cada mejora puede crear oportunidades adicionales de perfeccionamiento.

Ejercicios

- 10.1 ¿Cuántos movimientos realizan los algoritmos para desplazar n discos en el problema de las torres de Hanoi?
- *10.2 Pruébese que el algoritmo recursivo (dividir para vencer) de las torres de Hanoi y el algoritmo simple no recursivo descrito al principio de la sección 10.1 efectúan los mismos pasos.
- 10.3 Muéstrense las acciones del algoritmo de la multiplicación de enteros de dividir para vencer de la figura 10.3 al multiplicar 1011 por 1101.
- *10.4 Generalícese la programación del torneo de tenis de la sección 10.1 para torneos en los que el número de jugadores no sea una potencia de dos. Suge-

rencia. Si el número de jugadores n es impar, un jugador debe quedar sin jugar algún día, y el torneo necesitará n días para terminar, en vez de $n - 1$. No obstante, si hay dos grupos con un número impar de jugadores, los que obtienen el día sin jugar de cada grupo pueden jugar entre sí.

- 10.5** El número de combinaciones de m objetos de n , denotado por $\binom{n}{m}$, para $n \geq 1$ y $0 \leq m \leq n$, se puede definir recursivamente por

$$\binom{n}{m} = 1 \text{ si } m = 0 \text{ o } m = n$$

$$\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1} \text{ si } 0 < m < n$$

- a) Proporcione una función recursiva para calcular $\binom{n}{m}$.
- b) ¿Cuál es el tiempo de ejecución en el peor caso, como una función de n ?
- c) Obténgase un algoritmo de programación dinámica para calcular $\binom{n}{m}$. *Sugerencia:* El algoritmo construye una tabla conocida como triángulo de Pascal.
- d) ¿Cuál es el tiempo de ejecución de la respuesta a c) como una función de n ?

- 10.6** Otra forma de calcular el número de combinaciones de m objetos de n es mediante $(n)(n-1)(n-2)\dots(n-m+1)/(1)(2)\dots(m)$.

- a) ¿Cuál es el tiempo de ejecución en el peor caso de este algoritmo como una función de n ?
- *b) ¿Es posible calcular la función de las «apuestas de la Serie Mundial de Béisbol» $P(i, j)$ de la sección 10.2 de una forma similar? ¿Con qué rapidez puede realizarse este cálculo?

- 10.7** a) Escríbase otra vez el cálculo de las apuestas de la figura 10.7 para tener en cuenta el hecho de que el primer equipo tiene una probabilidad p de ganar cualquier partido dado.
 b) Si los Dodgers han ganado un partido, y los Yankees, dos, pero los Dodgers tienen una probabilidad 0.6 de ganar cualquier juego, ¿quién es el ganador más probable de la Serie Mundial?

- 10.8** El cálculo de las apuestas de la figura 10.7 requiere un espacio $O(n^2)$. Reescríbase el algoritmo para usar sólo un espacio $O(n)$.

- ***10.9** Pruébese que de la ecuación (10.4) resultan exactamente $\binom{i+j}{i}$ llamadas a P .

- 10.10** Encuéntrese una triangulación minimal para un octágono regular, suponiendo que las distancias son euclidianas.

- 10.11** El problema de división en párrafos, en una forma muy simple, puede formularse como sigue: se da una secuencia de palabras p_1, p_2, \dots, p_k de longitudes l_1, l_2, \dots, l_k , que se desea agrupar en líneas de longitud L . Las palabras están separadas por espacios cuya amplitud ideal es b , pero los espa-

cios pueden reducirse o ampliarse si es necesario (pero sin solapamiento de palabras), de tal forma que una línea $p_i, p_{i+1} \dots p_j$ tenga exactamente longitud L . Sin embargo, la multa por reducción o ampliación es la magnitud de la cantidad total que los espacios se comprimen o amplían. Esto es, el costo de fijar la línea $p_i, p_{i+1} \dots p_j$ es $(j-i) |b' - b|$, donde b' , el ancho real de los espacios, es $(L - l_i - l_{i+1} - \dots - l_j) / (j-i)$. No obstante, si $j = k$ (la última línea), el costo es cero, a menos que $b' < b$, ya que no es necesario ampliar la última línea. Plantéese un algoritmo de programación dinámica para encontrar la separación de menor costo de p_1, p_2, \dots, p_k en líneas de longitud L . *Sugerencia:* Para $i = k, k-1, \dots, 1$, calcúlese el menor costo de fijar p_i, p_{i+1}, \dots, p_k .

- 10.12** Supóngase que se dan n elementos x_1, x_2, \dots, x_n relacionados por un orden lineal $x_1 < x_2 < \dots < x_n$, y que se desea disponer los m elementos dentro de un árbol binario de búsqueda. Supóngase que p_i es la probabilidad que se requiere de que una solicitud para encontrar un elemento se refiera a x_i . Entonces, para cualquier árbol binario de búsqueda, el costo promedio de una búsqueda es $\sum_{i=1}^n p_i(d_i + 1)$, donde d_i es la profundidad del nodo que contiene a x_i . Dados los p_i y en el supuesto de que los x_i nunca cambian, es posible encontrar un árbol binario de búsqueda que reduzca el costo. Dése un algoritmo de programación dinámica para hacerlo. ¿Cuál es el tiempo de ejecución de ese algoritmo? *Sugerencia:* Calcúlese para todas las i y j el costo de búsqueda óptimal entre todos los árboles que contienen sólo $x_i, x_{i+1}, \dots, x_{i+j-1}$, esto es, los j elementos comenzando desde x_i .
- **10.13** ¿Para qué valores de monedas produce una solución óptima el algoritmo ávido que calcula cambios de la sección 10.3?
- 10.14**
- Escríbase el algoritmo de triangulación recursivo analizado en la sección 10.2.
 - Muéstrese que el algoritmo recursivo produce exactamente 3^{s-4} llamadas a problemas no triviales cuando se inicia en un problema de tamaño $s \geq 4$.
- 10.15** Describáse un algoritmo ávido para
- el problema de la colocación unidimensional de paquetes, y
 - el problema de la separación en párrafos (Ejercicio 10.11). Proporciójense un ejemplo donde el algoritmo no produzca una respuesta óptima, o demuéstrese que tal ejemplo no existe.
- 10.16** Plantéese una versión no recursiva del algoritmo del árbol de búsqueda de la figura 10.17.
- 10.17** Considérese un árbol de juego en el cual hay seis fichas y los jugadores 1 y 2 se turnan de una a tres fichas. El jugador que tome la última ficha pierde el juego.

- a) Dibújese el árbol de juego completo.
 - b) Si el árbol de juego completo se explorara mediante la técnica de poda alfa-beta y se analizaran primero los nodos que representan las configuraciones con el menor número de fichas, ¿qué nodos se podarían?
 - c) ¿Quién resulta vencedor si ambos jugadores hacen sus mejores jugadas?
- *10.18 Desarróllese un algoritmo de ramificación y acotamiento para el PAV, basado en la idea de comenzar un recorrido en el vértice 1 y de que en cada nivel la ramificación se basa en el nodo que sigue en el recorrido (en vez de en la elección de una arista en particular, como en la Fig. 10.22). ¿Qué estimación es correcta para la cota inferior de las configuraciones, que son listas de vértices $1, v_1, v_2, \dots$ que empiezan un recorrido? ¿Cómo se comporta el algoritmo en la figura 10.21, suponiendo que a es el vértice 1?
- *10.19 Un posible algoritmo de búsqueda local para el problema de la separación en párrafos es permitir transformaciones locales que pasen la primera palabra de una línea a la línea anterior o la última palabra de una línea a la línea siguiente. ¿Este algoritmo será localmente óptimal, en el sentido de que toda solución localmente óptima es una solución globalmente óptima?
- 10.20 Si las transformaciones locales sólo consisten en opciones dobles, ¿hay algún recorrido localmente óptimo en la figura 10.21 que no sea globalmente óptimo?

Notas bibliográficas

Hay muchos ejemplos importantes de los algoritmos de clasificación del resultado por división, incluyendo la transformada rápida de Fourier $O(n\log n)$ de Cooley y Tukey [1965], el algoritmo de la multiplicación de enteros $O(n\log n \log \log n)$ de Schonhage y Strassen [1971], y el algoritmo de multiplicación de matrices $O(n^{2.81})$ de Strassen [1969]. El algoritmo de multiplicación de enteros $O(n^{1.59})$ es de Karatsuba y Ofman [1962]. Moenck y Borodin [1972] desarrollaron diferentes algoritmos eficientes del de dividir para vencer para aritmética modular y evaluación e interpolación polinomiales.

La programación dinámica fue popularizada por Bellman [1957]. La aplicación de programación dinámica a la triangulación se debe a Fuchs, Kedem y Uzelton [1977]. El ejercicio 10.11 es de Knuth [1981]. Knuth [1971] da una solución al problema de los árboles de búsqueda binaria óptimales del ejercicio 10.12.

En Lin y Kernighan [1973] se describe una técnica heurística eficiente para el problema del agente viajero.

Véanse Aho, Hopcroft y Ullman [1974], y Garey y Johnson [1979] sobre un análisis de problemas NP-completos y otros problemas computacionalmente difíciles.

11

Estructuras de datos y algoritmos para almacenamiento externo

Este capítulo comienza considerando las diferencias en las formas de acceso entre la memoria principal y los dispositivos de almacenamiento externo como los discos. Despues se presentan varios algoritmos para clasificación de archivos de datos almacenados en forma externa. Se concluye el capítulo con un análisis de estructuras de datos y algoritmos, como los archivos indizados y los árboles B, que son muy adecuados para el almacenamiento y recuperación de información en dispositivos de almacenamiento secundario.

11.1 Un modelo para cómputos con almacenamiento externo

En los algoritmos estudiados en capítulos anteriores, se ha supuesto que la cantidad de datos de entrada es lo bastante pequeña como para que quepan en la memoria al mismo tiempo. Pero, ¿qué sucede si se desea clasificar a todos los empleados de gobierno de acuerdo con su antigüedad, o almacenar toda la información de los impuestos de la nación? En problemas como éstos, la cantidad de datos por procesar supera la capacidad de la memoria principal. La mayor parte de los grandes sistemas de cómputo tienen dispositivos de almacenamiento externo conectados en línea, como discos o dispositivos de almacenamiento masivo, en los cuales se pueden almacenar cantidades muy grandes de datos. Sin embargo, esos dispositivos tienen características de acceso que difieren mucho de las de la memoria principal. Se han desarrollado diversas estructuras de datos y algoritmos para utilizar con más eficiencia esos dispositivos. Este capítulo comprende las estructuras de datos y algoritmos para clasificar y recuperar la información almacenada en memoria secundaria.

El lenguaje Pascal, y algunos otros, tienen el tipo de datos archivo, destinado a representar datos almacenados en memoria secundaria. Aunque el lenguaje utilizado no tenga este tipo de datos, es indudable que el sistema operativo debe manejar la noción de archivos en memoria secundaria. Tanto si se habla de los archivos de Pascal como de los manipulados directamente por el sistema operativo, se encuentran limitaciones a la forma en que puede accederse a los archivos. El sistema operativo divide la memoria secundaria en *bloques* de igual tamaño. El tamaño de los bloques varía entre los distintos sistemas operativos, pero de 512 a 4096 bytes es lo típico.

Se considera un archivo como si estuviera almacenado en una lista enlazada de bloques, aunque lo más común es que el sistema operativo utilice una disposición

tipo árbol, donde los bloques que contienen el archivo son hojas, y cada uno de los nodos interiores apunta a varios bloques del archivo. Si, por ejemplo, cuatro bytes son suficientes para contener la dirección de un bloque y los bloques tienen 4096 bytes de longitud, entonces un bloque raíz puede tener apuntadores hasta para 1024 bloques. Así, los archivos de hasta 1024 bloques, es decir, cerca de cuatro millones de bytes, pueden representarse con un bloque raíz y los bloques que contienen el archivo. Los archivos de hasta 2^{20} bloques, o 2^{32} bytes, pueden representarse con un bloque raíz que apunte a 1024 bloques en un nivel intermedio, cada uno de los cuales apunta a 1024 bloques...hoja que contienen una parte del archivo, y así sucesivamente.

La operación básica en archivos consiste en llevar un solo bloque a un *buffer* (almacenamiento temporal) de la memoria principal; un buffer no es más que un área reservada de memoria principal cuyo tamaño es idéntico al de un bloque. Un sistema operativo típico facilita la lectura de los bloques de acuerdo con el orden en que aparecen en la lista de bloques que conforman el archivo. Esto es, al principio se lee el primer bloque y se guarda en el buffer de ese archivo, después se reemplaza por el segundo, que queda escrito dentro del mismo buffer, y así sucesivamente.

Ahora se puede ver la razón de las reglas para la lectura de archivos en Pascal. Cada archivo está almacenado en una secuencia de bloques, con un número entero de registros en cada bloque. (Se puede desperdiciar espacio, puesto que se evita dividir un registro entre bloques.) El cursor de lectura siempre apunta a uno de los registros del bloque que se encuentra en el buffer en ese instante. Cuando ese cursor se deba mover a un registro que no esté en el buffer, es el momento de leer el siguiente bloque del archivo.

Igualmente, se puede considerar el proceso de escritura de archivos de Pascal como la creación de un archivo en un buffer. Cuando se «escriben» los registros en archivo, se colocan en el buffer de ese archivo, en la posición inmediata siguiente a los registros colocados previamente. Cuando el buffer no puede contener otro registro completo, se copia el buffer en un bloque disponible de memoria secundaria y ese bloque se agrega al final de la lista de bloques del archivo. Se considera ahora que el buffer está vacío, y que se pueden escribir más registros en él.

Costo de las operaciones con almacenamiento secundario

Dada la naturaleza de los dispositivos de almacenamiento secundario, como los discos, el tiempo para encontrar un bloque y leerlo a la memoria principal es muy grande, comparado con el tiempo que lleva procesar el dato. Por ejemplo, supóngase que se tiene un bloque de 1000 enteros dentro de un disco que gira a 1000 rpm. El tiempo que lleva colocar la cabeza sobre la pista que contenga este bloque (*tiempo de búsqueda*), más el tiempo consumido en esperar que el bloque quede bajo la cabeza (*tiempo de latencia*), puede ser de 100 milisegundos en promedio. El proceso de escritura de un bloque en un lugar particular dentro del almacenamiento secundario lleva una cantidad similar de tiempo. Sin embargo, la máquina puede efectuar 100 000 instrucciones en esos 100 milisegundos. Este tiempo es más que suficiente

para hacer un procesamiento simple a los mil enteros una vez que están en la memoria principal, como la suma de ellos o la ubicación del máximo; incluso puede ser suficiente para clasificación rápida de los enteros.

Cuando se evalúa el tiempo de ejecución de los algoritmos que operan sobre datos almacenados en forma de archivos, es de primordial importancia considerar el número de veces que se lee un bloque a la memoria principal o se escribe un bloque en la memoria secundaria; esa operación se denomina *acceso a bloques*. Se supone que el tamaño de los bloques lo fija el sistema operativo, así que no se puede hacer que un algoritmo se ejecute con mayor rapidez incrementando el tamaño del bloque, para reducir el número de accesos a los bloques. Como consecuencia, lo que se debe considerar para los algoritmos que emplean almacenamiento externo será el número de accesos a los bloques. Se inicia el estudio de algoritmos para almacenamiento externo con la clasificación externa.

11.2 Clasificación externa

La clasificación de datos organizados como archivos o de forma más general, la clasificación de datos almacenados en memoria secundaria, se conoce como clasificación «externa». Este estudio de clasificación externa parte del supuesto de que los datos están almacenados en un archivo de Pascal. Se presenta la forma en que un algoritmo de «clasificación por intercalación» puede ordenar un archivo de n registros en sólo $O(\log n)$ pasadas por el archivo; esta cifra es mucho mejor que los $O(n)$ pasos requeridos por los algoritmos estudiados en el capítulo 8. Después se estudia cómo la utilización de ciertas características del sistema operativo para controlar la lectura y escritura de bloques en tiempos adecuados puede acelerar la clasificación al reducir el tiempo en que el computador está ocioso, esperando la lectura o escritura de un bloque hacia o desde la memoria principal.

Clasificación por intercalación

La idea esencial en la clasificación por intercalación es organizar un archivo en *fragmentos* cada vez más grandes, esto es, secuencias de registros r_1, \dots, r_k , donde la clave de r_i no es mayor que la clave de r_{i+1} para $1 \leq i < k$. Se dice que un archivo r_1, \dots, r_m de registros está *organizado en fragmentos de longitud k* si para toda $i \geq 0$ tal que $ki \leq m$, $r_{k(i-1)+1}, r_{k(i-1)+2}, \dots, r_{ki}$ es un fragmento de longitud k , y además si m no es divisible entre k , y $m = pk + q$, donde $q < k$, por lo que la secuencia de registros $r_{m-q+1}, r_{m-q+2}, \dots, r_m$, llamada *cola*, es un fragmento de longitud q . Por ejemplo, la secuencia de enteros mostrada en la figura 11.1 está organizada en fragmentos de longitud 3. Obsérvese que la cola es de longitud menor que 3, pero consta de registros ordenados, a saber, 5 y 12.

7	15	29	8	11	13	16	22	31	5	12
---	----	----	---	----	----	----	----	----	---	----

Fig. 11.1. Archivo con fragmentos de longitud tres.

El paso básico en una clasificación por intercalación en archivos es empezar con dos archivos, como f_1 y f_2 , organizados en fragmentos de longitud k . Suponiendo que

- el número de fragmentos, incluyendo las colas, en f_1 y f_2 difiere a lo sumo en 1,
- como máximo uno de f_1 y f_2 tiene cola, y
- el único con cola tiene por lo menos tantos fragmentos como el otro.

Es un proceso sencillo leer un fragmento de f_1 y f_2 , combinar ambos y unir el fragmento resultante, de longitud $2k$, en alguno de los archivos g_1 y g_2 , que se han organizado en fragmentos de longitud $2k$. Alternando entre g_1 y g_2 , se hace que esos archivos no sólo se organicen en fragmentos de longitud $2k$, sino que satisfagan (1), (2) y (3). Para ver que (2) y (3) se satisfacen, es útil observar que la cola entre los fragmentos de f_1 y f_2 queda combinada en (o quizás ya está) el último fragmento creado.

Se empieza por dividir los n registros en dos archivos f_1 y f_2 , lo más uniformemente posible. Cualquier archivo se puede considerar organizado en fragmentos de longitud 1. Después, se combinan en fragmentos de longitud 1 y se distribuyen en los archivos g_1 y g_2 , organizados en fragmentos de longitud 2. Se vacían f_1 y f_2 , y se combinan g_1 y g_2 en f_1 y f_2 , ahora estarán organizados en fragmentos de longitud 4. Después, se combinan f_1 y f_2 para crear g_1 y g_2 organizados en fragmentos de longitud 8, y así sucesivamente.

Después de i pasadas de esta naturaleza, se tendrán dos archivos constituidos por fragmentos de longitud 2^i . Si $2^i \geq n$, uno de los dos archivos estará vacío y el otro tendrá un solo fragmento de longitud n , esto es, estará clasificado. Como $2^i \geq n$ cuando $i \geq \log n$, $\lceil \log n \rceil$ pasadas son suficientes. Cada pasada requiere la lectura de dos archivos y la escritura de otros dos, longitud aproximada $n/2$. El número total de bloques leídos o escritos en cada paso es aproximadamente $2n/b$, donde b es el número de registros que caben en un bloque. De este modo, el número de bloques leídos y escritos por el proceso de clasificación completo es $O((n \log n)/b)$ o, de otra forma, la cantidad de lecturas y escrituras es casi la misma que la requerida para hacer $O(\log n)$ pasadas por los datos almacenados en un solo archivo. Esta cifra supone una importante mejora de las $O(n)$ pasadas requeridas por muchos de los algoritmos de clasificación tratados en el capítulo 8.

La figura 11.2 muestra el proceso de combinación en Pascal. Se leen dos archivos organizados en fragmentos de longitud k y se escriben dos archivos organizados en fragmentos de longitud $2k$. Como ejercicio, se deja la especificación de un algoritmo que siga las ideas anteriores y que utilice el procedimiento *combina* de la figura 11.2 $\log n$ veces para clasificar un archivo de n registros.

```

procedure combina ( k: integer; { longitud del fragmento de entrada }
                   f1, f2, g1, g2: file of tipo_registro );
var
  commuta_salida: boolean; { indica si la escritura es en g1 (verdadero) o en
                            g2 (falso) }
  ganador: integer; { elige el archivo con la menor clave del registro actual }
  usado: array [1..2] of integer; { usado[j] indica cuántos registros se han
                                 leído hasta ahora en el fragmento actual del archivo f_j }
  fin: array [1..2] of boolean; { fin[j] es verdadero si ha terminado el
                                archivo f_j }

```

```

fragmento de  $f_1$  – se han leído  $k$  registros o se ha alcanzado el fin del
archivo de  $f_1\}$ 
actual: array [1..2] of tipo_registro; { los registros actuales de ambos
archivos }
procedure toma_registro ( i: integer ); { avanza el archivo  $f_i$ , pero no más allá
del final del archivo o del fragmento. Fija  $fin[i]$  si se encuentra el fin del
archivo o del fragmento }
begin
  usado[i] := usado[i] + 1;
  if (usado[i] = k) or
    ( $i = 1$ ) and eof(1) or
    ( $i = 2$ ) and eof(2) then fin[i] := true
  else if  $i = 1$  then read(f1, actual[1])
  else read(f2, actual[2])
end; { toma_registro }

begin { combina }
  conmuta_salida := true; { los primeros fragmentos intercalados van hacia g1 }
  rewrite(g1); rewrite(g2);
  reset(1); reset(2);
  while not eof(1) or not eof(2) do begin { intercala dos archivos }
    { asigna valor inicial }
    usado[1] := 0; usado[2] := 0;
    fin[1] := false; fin[2] := false;
    toma_registro(1); toma_registro(2);
    while not fin[1] or not fin[2] do begin { intercala dos fragmentos }
      { elige ganador }
      if fin[1] then ganador := 2
        {  $f_2$  gana por "omisión"; el fragmento de  $f_1$  se terminó }
      else if fin[2] then ganador := 1
        {  $f_1$  gana por omisión }
      else { ningún fragmento se ha terminado }
        if actual[1].clave < actual[2].clave then ganador := 1
        else ganador := 2;
      { escribe el registro del ganador }
      if conmuta_salida then write(g1, actual[ganador])
      else write(g2, actual[ganador]);
      { avanza el archivo ganador }
      toma_registro(ganador)
    end;
    { se ha terminado la intercalación de dos fragmentos; se conmuta el
    archivo de salida y se repite }
    conmuta_salida := not conmuta_salida
  end
end; { combina }

```

Fig. 11.2. Procedimiento *combina*.

Obsérvese que el procedimiento *combina* de la figura 11.2 nunca requiere un fragmento completo en memoria; lee y escribe un registro de cada vez. No se desea almacenar los fragmentos completos que se encuentran en memoria principal que obliguen a tener dos archivos de entrada. En otro caso, se podrían leer dos fragmentos al mismo tiempo desde un solo archivo.

Ejemplo 11.1. Considérese la lista de 23 números que se muestra dividida en dos archivos en la figura 11.3(a). Se empieza por combinar fragmentos de longitud 1 para crear los dos archivos de la figura 11.3(b). Por ejemplo, los primeros fragmentos de longitud uno son 28 y 31, y se combinan tomando primero 28 y luego 31. Los dos fragmentos siguientes de longitud uno, 3 y 5, se combinan para formar uno de longitud dos, que se coloca en el segundo archivo de la figura 11.3(b). Los fragmentos se separan en la figura 11.3(b) por medio de líneas verticales que no son parte del archivo. Obsérvese que el segundo archivo de la figura 11.3(b) tiene una cola de longitud uno, el registro 22, mientras que el primer archivo no tiene cola.

Se pasa, de la figura 11.3(b) a la figura 11.3(c) combinando los fragmentos de longitud dos. Por ejemplo, los fragmentos 28, 31 y 3, 5, se combinan para formar 3, 5, 28, 31 de la figura 11.3(c). Al alcanzar los fragmentos de longitud 16 de la figura 11.3(e), un archivo tiene un fragmento completo y el otro sólo tiene una cola, de longitud 7. En la última etapa, donde los archivos están en apariencia organizados como fragmentos de longitud 32, sucede que se tiene un archivo formado sólo por una cola, de longitud 23, y el segundo se encuentra vacío. El fragmento sencillo de longitud 23 es, por supuesto, la clasificación que se buscaba. □

Aceleración de la clasificación por intercalación

Por medio de un ejemplo simple se ha mostrado el proceso de la clasificación por intercalación partiendo de fragmentos de longitud 1. Se aprovechará más el tiempo si se empieza con un paso que, para una k apropiada,lea grupos de k registros dentro de la memoria principal, los ordene, por ejemplo, con clasificación rápida, y los devuelva a la memoria secundaria como un fragmento de longitud k .

Por ejemplo, un millón de registros, necesitarían 20 pasadas por los datos para hacer la clasificación empezando con fragmentos de longitud 1. Sin embargo, si se pueden tener 10 000 registros a la vez en la memoria principal, también se pueden leer 100 grupos de 10 000 registros en una pasada, clasificar cada grupo, y dejar 100 fragmentos de longitud 10 000 distribuidos uniformemente entre dos archivos. Siete pasadas de intercalación más culminarán con un archivo organizado como un fragmento de longitud $10\,000 \times 2^7 = 1\,280\,000$, lo que es mayor que un millón y significa que los datos están clasificados.

Minimización del tiempo transcurrido

28	3	93	10	54	65	30	90	10	69	8	22
31	5	96	40	85	9	39	13	8	77	10	

(a) archivos iniciales

28	31	93	96	54	85	30	39	8	10	8	10
3	5	10	40	9	65	13	90	69	77	22	

(b) organizados en fragmentos de longitud 2

3	5	28	31	9	54	65	85	8	10	69	77
10	40	93	06	13	30	39	90	8	10	22	

(c) organizados en fragmentos de longitud 4

3	5	10	28	31	40	93	96	8	8	10	10	22	69	77
9	13	30	39	54	65	85	90							

(d) organizados en fragmentos de longitud 8

3	5	9	10	13	28	30	31	39	40	54	65	85	90	93	96
8	8	10	10	22	69	77									

(e) organizados en fragmentos de longitud 16

3 5 8 8 9 10 10 13 22 28 30 31 39 40 54 65 69 77 85 90 93 96

(f) organizados en fragmentos de longitud 32

Fig. 11.3. Clasificación por intercalación de una lista.

chivo que se está leyendo, tal como sucede durante el proceso de clasificación por intercalación. Sin embargo, el hecho es que el tiempo transcurrido por dicho proceso es mayor que el tiempo empleado en el cálculo efectuado con los datos presentes en la memoria principal. Si se clasifican archivos realmente grandes, donde la operación lleva horas, el tiempo transcurrido resulta un factor importante, aun si no se paga, y debe buscarse la forma de que el proceso de la clasificación por intercalación lleve un tiempo total mínimo.

Como se ha mencionado, es común que el tiempo para leer datos de disco o cin-

ta sea mayor que el tiempo consumido en hacer cálculos simples con esos datos, como la intercalación. Por tanto, es de esperar que si sólo hay un canal dedicado a la transferencia de datos entre la memoria principal y la memoria auxiliar, este canal formará un embotellamiento, el canal de datos estará ocupado todo el tiempo, y el tiempo total transcurrido se igualará con el tiempo utilizado en el movimiento de los datos. Esto es, todos los cálculos se efectuarán en cuanto los datos se encuentren disponibles, mientras se leen o escriben datos adicionales.

Aun en este ambiente simple, debe tenerse cuidado para asegurar la terminación en una cantidad mínima de tiempo. Para ver qué puede fallar, supóngase que se lee, cada vez un bloque, de dos archivos de entrada f_1 y f_2 , en forma alterna. Los archivos están organizados en fragmentos de longitud bastante mayor que el tamaño de un bloque, así que para intercalar dos fragmentos es necesario leer muchos bloques de cada archivo. Sin embargo, supóngase que todos los registros contenidos en el fragmento del archivo f_1 preceden a todos los registros del archivo f_2 . Entonces, como se leen bloques en forma alterna, todos los bloques de f_2 tienen que permanecer en memoria. Es posible que no haya espacio para conservar todos esos bloques en la memoria principal, y aun si lo hubiera, es necesario, después de leer todos los bloques del fragmento, esperar mientras se copia y escribe todo el fragmento que proviene de f_2 .

Para evitar esos problemas, se consideran las claves de los últimos registros de los últimos bloques leídos de f_1 y f_2 , por ejemplo k_1 y k_2 , respectivamente. Si algún fragmento se agota, es obvio que se lee la siguiente de otro archivo. Sin embargo, si un fragmento no se agota, se continúa leyendo un bloque de f_1 si $k_1 < k_2$, y de f_2 , en caso contrario. Esto es, debe determinarse cuál de los dos fragmentos tendrá primero todos sus registros seleccionados en ese momento en la memoria principal, y se llenan primero registros de ese fragmento. Si la selección de registros se efectúa más rápido que la lectura, se sabe que al haber leído el último bloque de los dos fragmentos, no puede haber más que dos bloques llenos de registros sin intercalar; los registros pueden estar distribuidos en tres bloques, pero no en más.

Intercalación múltiple

Si la lectura y escritura entre las memorias principal y secundaria es el «embottellamiento», puede ahorrarse tiempo si se tiene más de un canal de datos. Supóngase que se tienen $2m$ unidades de disco, cada una con su propio canal de comunicación. Se podrían colocar m archivos, f_1, f_2, \dots, f_m en m de las unidades de disco, organizados como fragmentos de longitud k . Entonces, se pueden leer m fragmentos, uno de cada archivo, y combinarlos en un fragmento de longitud mk , el cual se coloca en alguno de los m archivos de salida, g_1, g_2, \dots, g_m , cada uno tomando un fragmento a la vez.

El proceso de intercalación en memoria principal puede llevarse a cabo en $O(\log m)$ pasos por registro si se organizan los m registros candidatos, esto es, los registros más pequeños de cada archivo que hasta ese momento no se habían seleccionado, en un árbol parcialmente ordenado u otra estructura de datos que maneje

las operaciones en colas de prioridad INSERTA y SUPRIME-MIN en tiempo logarítmico. Para seleccionar el registro con la clave más pequeña de la cola de prioridad, se realiza SUPRIME-MIN, y después INSERTA, en la cola de prioridad del siguiente registro del archivo del ganador, como reemplazo del registro seleccionado.

Si hay n registros, y la longitud de los fragmentos se multiplica por m en cada paso, después de i pasadas los fragmentos serán de longitud m^i . Si $m^i \geq n$, esto es, después de $i = \log_m n$ pasadas, la lista completa estará clasificada. Como $\log_m n = -\log_2 n / \log_2 m$, se ahorra en un factor de $\log_2 m$ el número de veces que se lee cada registro. Más aún, si m es el número de unidades de disco utilizadas para los archivos de entrada, y m son usadas para la salida, es posible procesar datos m veces tan rápido como si existiera sólo una unidad de disco para la entrada y una para la salida, o $2m$ veces tan rápido como si los archivos de entrada y salida estuvieran almacenados en una unidad de disco. Lamentablemente, incrementar m en forma indefinida no acelera el procesamiento en un factor de $\log m$. La razón de esto es que para una m suficientemente grande, el tiempo requerido por la intercalación en memoria principal, que en realidad se incrementa como $\log m$, será superior al de lectura o escritura de datos. En este punto, si hay incrementos posteriores en m , aumentarán el tiempo transcurrido, ya que el cálculo en memoria principal se convertirá en el embotellamiento.

Clasificación en varias fases

Es posible realizar una clasificación por intercalación de m -caminos con sólo $m + 1$ archivos, como alternativa a la estrategia descrita antes, que emplea $2m$ archivos. Se efectúa una secuencia de pasadas al intercalar fragmentos de m de los archivos en otros más largos en el archivo restante. Es necesario tener en cuenta los siguientes:

1. En una pasada, cuando los fragmentos de cada uno de los m archivos se intercalan en fragmentos del $(m + 1)$ -ésimo archivo, no es necesario usar todos los fragmentos en cada uno de los m archivos de entrada. Antes bien, cada archivo, cuando es de salida, se ocupa con fragmentos de cierta longitud. Se usan algunos de esos fragmentos para ayudar a llenar cada uno de los otros m archivos cuando les llegue el turno de ser archivos de salida.
2. Cada paso produce archivos de longitud diferente. Puesto que cada uno de los archivos cargados con fragmentos en las m pasadas previas contribuye a los fragmentos del paso actual, la longitud en una pasada es la suma de las longitudes de los fragmentos producidos en las m pasadas previas. (Si se han dado menos de m pasadas, habrá que considerar todos los anteriores como si produjeran fragmentos de longitud 1.)

Este proceso de clasificación por intercalación se conoce como *clasificación en varias fases (polyphase sorting)*. El cálculo exacto de los números de pasadas necesarias como una función de m y n (el número de registros), y el cálculo de la distribución inicial de los fragmentos en m archivos se dejan como ejercicio. Sin embargo, aquí se dará un ejemplo para dar idea del caso general.

Ejemplo 11.2. Si $m = 2$, se comienza con dos archivos f_1 y f_2 , organizados en fragmentos de longitud 1. Los registros de f_1 y f_2 se intercalan para hacer fragmentos de longitud 2 en un tercer archivo, f_3 . Sólo se intercalan suficientes fragmentos para vaciar f_1 . Despues, se intercalan los restantes de longitud 1 de f_2 con un número igual de fragmentos de longitud 2 de f_3 . Los resultantes, de longitud 3, quedarán colocados en f_1 . Despues, deben intercalarse los de longitud 2 de f_3 con los de longitud 3 de f_1 . Esos fragmentos, de longitud 5, se colocan en f_2 , que quedó vacío en la pasada anterior.

La secuencia de longitudes de los fragmentos 1, 1, 2, 3, 5, 8, 13, 21, ..., es la sucesión de Fibonacci. Esta secuencia se genera por medio de la relación de recurrencia $F_0 = F_1 = 1$, y $F_i = F_{i-1} + F_{i-2}$, para $i \geq 2$. Obsérvese que la razón entre los números de Fibonacci consecutivos F_{i+1}/F_i se acerca a la «razón dorada» $(\sqrt{5} + 1)/2 = 1.618\dots$ conforme i crece.

De aquí se deduce que para que el proceso continúe hasta que la lista quede clasificada, los números iniciales de registros en f_1 y f_2 deben ser dos números consecutivos de Fibonacci. Por ejemplo, la figura 11.4 muestra qué sucede al empezar con $n = 34$ registros (34 es el número de Fibonacci F_8), distribuidos 13 en f_1 y 21 en f_2 . (13 y 21 son el sexto y el séptimo números de Fibonacci, así que la razón F_7/F_6 es muy cercana a 1.618; de hecho, es 1.615.) El estado de un archivo se representa en la figura 11.4 como $a(b)$, lo cual significa que tiene a fragmentos de longitud b . □

después de la pasada	f_1	f_2	f_3
inicial	13(1)	21(1)	vacío
1	vacío	8(1)	13(2)
2	8(3)	vacío	5(2)
3	3(3)	5(5)	vacío
4	vacío	2(5)	3(8)
5	2(13)	vacío	1(8)
6	1(13)	1(21)	vacío
7	vacío	vacío	1(34)

Fig. 11.4. Ejemplo de clasificación en varias fases.

Un caso en el que la velocidad de entrada/salida no es un cuello de botella

Cuando la lectura de archivos es el cuello de botella, el siguiente bloque debe escogerse con sumo cuidado. Como ya se dijo, la situación a evitar es aquella en la que se tienen que almacenar varios bloques de un fragmento, debido a que ese fragmento tenía registros con claves grandes, los cuales serán escogidos después de la mayor parte o de todos los registros del otro fragmento. El truco para evitar este problema consiste en determinar con rapidez qué fragmento agotará primero sus registros en memoria principal, comparando los últimos de esos registros de cada archivo.

Cuando el tiempo requerido para leer datos en la memoria principal es comparable o menor que el tiempo requerido para procesar los datos, se vuelve aún más crítica la selección del archivo de entrada desde el cual leer un bloque con cuidado, ya que no hay esperanza de construir una reserva de registros dentro de la memoria principal en caso de que el proceso de intercalación empiece repentinamente al tomar más registros de un fragmento que del otro. El «truco» mencionado antes resulta útil en diversas situaciones, como se verá a continuación.

Considérese el caso donde la intercalación es el cuello de botella, y no la lectura o la escritura, por dos razones.

1. Como se ha visto, al tener disponibles varias unidades de disco o cinta, es posible acelerar la entrada/salida lo suficiente para que el tiempo requerido por la intercalación supere al tiempo de entrada o al de salida.
2. Los canales de alta velocidad pueden estar pronto disponibles en el mercado.

Por tanto, se considerará un modelo simple del problema que se puede presentar cuando la intercalación *«es el cuello de botella en una clasificación realizada con datos almacenados en la memoria secundaria.* Específicamente, se supone que

- a) Se intercalan fragmentos mucho más grandes que los bloques.
- b) Hay dos archivos de entrada y dos de salida. Los archivos de entrada están almacenados en un disco (o algún otro dispositivo conectado a la memoria principal a través de un solo canal) y los archivos de salida están en otra unidad similar con un solo canal.
- c) Los tiempos para
 - I) leer un bloque
 - II) escribir un bloque, y
 - III) seleccionar suficientes registros con las claves más pequeñas entre los dos fragmentos que se encuentran en ese momento en memoria principal, para llenar un bloque.

son todos iguales.

En dichos supuestos, se considera una clase de estrategias de intercalación donde varios buffers de entrada (espacios para contener un bloque) se ubican en la memoria principal. En todo momento, alguno de esos buffers contendrá los registros no seleccionados de los dos fragmentos de entrada, y uno de ellos estará en el proceso de ser leído de uno de los archivos de entrada. Los otros dos contendrán la salida, es decir, los registros seleccionados en el orden de intercalación adecuado. En todo momento, uno de esos buffers se encontrará en el proceso de escritura en alguno de los archivos de salida y el otro se estará llenando con registros seleccionados de los buffers de entrada.

Una *transferencia* consiste en hacer lo siguiente (quizá todo al mismo tiempo):

1. la lectura de un bloque de entrada en un buffer de entrada.
2. el llenado de uno de los buffers de salida con los registros seleccionados, es decir, los registros con las claves más pequeñas, entre todos los que se tengan en ese momento en el buffer de entrada.

3. la escritura del otro buffer de salida en uno de los dos archivos de salida que se esté generando.

Por las suposiciones, (1), (2) y (3) requieren el mismo tiempo. Para obtener el máximo de eficiencia, se deben efectuar en paralelo. Se puede hacer esto, a menos que (2), la selección de registros con las claves más pequeñas, incluya algunos de los registros que se están leyendo en ese momento †. Así, se debe buscar una estrategia para seleccionar los buffers que se van a leer, de modo que al principio de cada transferencia los b registros no seleccionados con las claves menores ya se encuentren listos en los buffers de entrada, donde b es el número de registros que llenan un bloque o buffer.

Las condiciones en las cuales la intercalación puede efectuarse en paralelo con la lectura son simples. Sean k_1 y k_2 las claves más grandes de todos los registros no seleccionados en memoria principal del primero y segundo fragmentos, respectivamente. Entonces, en la memoria principal deben encontrarse por lo menos b registros no seleccionados cuyas claves no excedan de $\min(k_1, k_2)$. En primer lugar se mostrará cómo hacer la intercalación con seis buffers, tres para cada archivo, y después se mostrará que es suficiente con cuatro buffers si se reparten entre los dos archivos.

Esquema de seis buffers de entrada

Este primer esquema se representa en el dibujo de la figura 11.5. Los dos buffers de salida no se muestran; existen tres buffers para cada archivo; cada uno tiene capacidad para b registros. El área sombreada representa los registros disponibles, y las claves están en orden ascendente en el sentido de las manecillas del reloj. Siempre, el número total de registros no seleccionados es $4b$ (a menos que el número de registros que permanecen en los fragmentos que se encuentran intercalando sea menor). Inicialmente, se leen los dos primeros bloques de cada fragmento en los buffers ††. Como siempre hay $4b$ registros disponibles, y a lo sumo $3b$ pueden proceder de un archivo, se sabe que hay por lo menos b registros que vienen de cada archivo. Si k_1 y k_2 son las claves más grandes disponibles de los dos fragmentos, debe haber b registros con las claves iguales o menores que k_1 y b registros con las claves menores o iguales que k_2 . Así, hay b registros con claves iguales o menores que $\min(k_1, k_2)$.

† Es tentador suponer que si (1) y (2) requieren el mismo tiempo, entonces la selección no podría nunca coincidir con la lectura; si el bloque completo no se hubiera leído aún, podrían seleccionarse de entre los primeros registros del bloque aquellos que tuvieran las claves menores. Sin embargo, por su naturaleza, en las lecturas de disco transcurre un período largo antes de que se encuentre el bloque y pueda realizarse la lectura. Por tanto, la única suposición segura es que nada del bloque que se está leyendo en una transferencia está disponible en ese momento para la selección.

†† Si éstos no son los primeros fragmentos tomados de cada archivo, entonces este paso inicial únicamente puede hacerse después de leer los anteriores fragmentos y sus últimos $4b$ registros se estén intercalando.

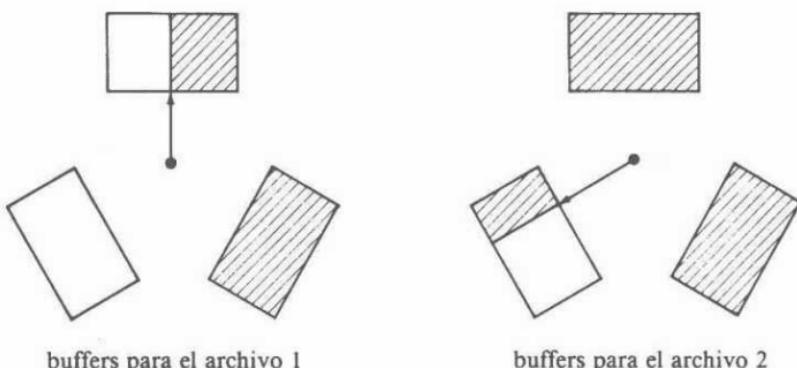


Fig. 11.5. Intercalación con seis buffers de entrada.

La pregunta sobre qué archivo leer a continuación es trivial. Por lo general, dado que dos buffers estarán llenos parcialmente, como en la figura 11.5, habrá sólo un buffer vacío y deberá llenarse. Si sucede, como cuando se está empezando, que cada fragmento tiene dos buffers completamente llenos y uno vacío, se toma cualquiera de los que se encuentran vacíos. Obsérvese que la demostración de que no es posible agotar un fragmento [existen b registros con claves iguales o menores que $\min(k_1, k_2)$] dependía sólo del hecho de que estuvieron presentes $4b$ registros.

Además, las flechas de la figura 11.5 representan apuntadores a los primeros registros disponibles (cuyas claves son menores) en ambos fragmentos. En Pascal, se representaría ese apuntador con dos enteros. El primero, en el intervalo $1..3$, representa el buffer apuntado, y el segundo, en el intervalo $1..b$, representa el registro dentro del buffer. En forma alternativa, es posible dejar que los buffers sean el primero, el medio y el último tercio de un arreglo y usar un entero en el intervalo $1..3b$. En otros lenguajes, donde los apuntadores pueden apuntar hacia elementos de arreglos, puede preferirse un apuntador de tipo \uparrow tipo_registro.

Esquema de cuatro buffers

La figura 11.6 sugiere un esquema con cuatro buffers. En el principio de cada transferencia, están disponibles $2b$ registros. Dos de los buffers de entrada están asignados a uno de los archivos; B_1 y B_2 de la figura 11.6 están asignados al archivo uno. Uno de estos buffers estará parcialmente lleno (vacío en el caso extremo), y el otro, lleno. El cuarto buffer no está comprometido, y se llenará desde uno de los archivos durante la transferencia.

Se mantendrá, por supuesto, la propiedad que permite intercalar en paralelo con la lectura; al menos b registros de la figura 11.6 deben tener claves menores o iguales que $\min(k_1, k_2)$, donde k_1 y k_2 son las claves de los últimos registros disponibles en los dos archivos, como se indica en la figura 11.6. Se denomina *segura* a la configuración que cumple esa propiedad. Al principio, se lee un bloque de cada frag-

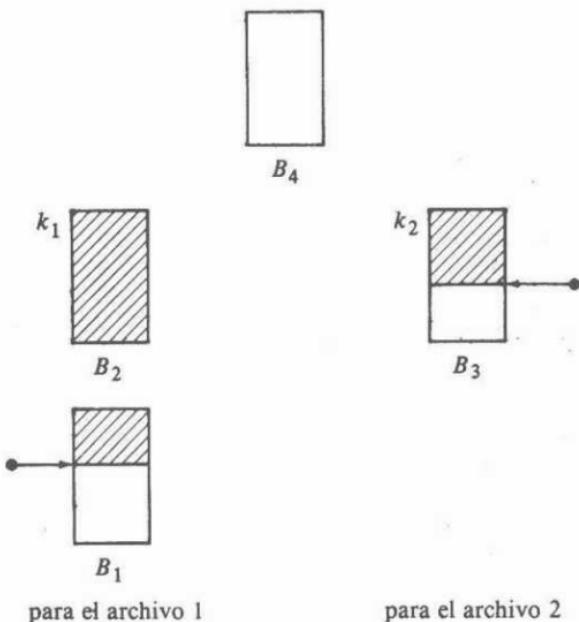


Fig. 11.6. Intercalación con cuatro buffers de entrada.

mento (este es el caso extremo, donde B_1 está vacío y B_3 está lleno en la Fig. 11.6), de modo que la configuración inicial esté segura. En el supuesto de que la figura 11.6 es segura, es necesario mostrar que la configuración lo será después de completar la siguiente transferencia.

Si $k_1 < k_2$, se escoge B_4 para llenarlo con el siguiente bloque del archivo uno y, en otro caso, llenarlo con el del archivo dos. Supóngase primero que $k_1 < k_2$. Ya que B_1 y B_3 de la figura 11.6 tienen exactamente b registros, se debe, durante la siguiente transferencia, agotar B_1 ; de otra forma, es necesario agotar B_3 y contradecir la seguridad de la figura 11.6. Así, después de una transferencia, la configuración se ve como en la figura 11.7(a).

Para comprobar que la figura 11.7(a) es segura, considérense dos casos. Primero, si k_3 , la última clave del bloque recién leído B_4 , es menor que k_2 , entonces, como B_4 está lleno, es muy probable que haya b registros iguales o menores que $\min(k_3, k_2)$, y la configuración es segura. Si $k_2 \leq k_3$, y dado que se supuso que $k_1 < k_2$ (de otro modo se hubiera llenado B_4 del archivo dos), los b registros de B_2 y B_3 tienen claves menores o iguales que $\min(k_2, k_1) = k_2$.

Ahora se estudiará el caso donde $k_1 \geq k_2$ en la figura 11.6. Se escoge leer el siguiente bloque del archivo dos. La figura 11.7(b) muestra la situación resultante. Como en el caso $k_1 < k_2$, se argumenta que B_1 debe agotarse y, por eso, se muestra que el archivo uno tiene asignado sólo el buffer B_2 de la figura 11.7(b). La demostración de que esta figura es segura es igual que la de la figura 11.7(a).

Obsérvese que, como en el esquema con seis buffers, no se lee un archivo más allá del fin de un fragmento. Sin embargo, si no es necesario leer un bloque desde uno de los fragmentos actuales, puede leerse un bloque del siguiente fragmento de ese archivo. Así, existe la oportunidad de leer un bloque de cada uno de los fragmentos siguientes, y entonces será posible iniciar la intercalación de los fragmentos tan pronto como se hayan seleccionado los últimos registros del fragmento anterior.

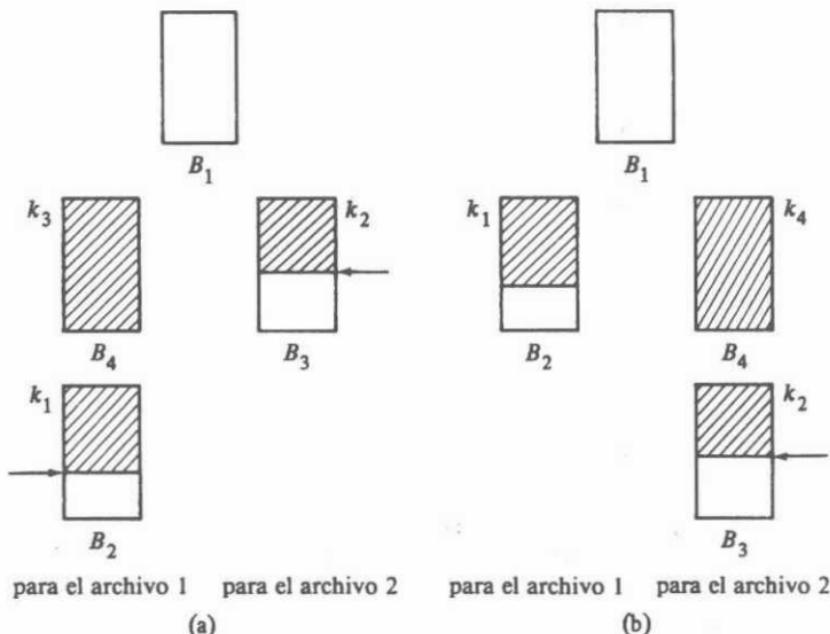


Fig. 11.7. Configuración después de una transferencia.

11.3 Almacenamiento de información en archivos

En esta sección, se analizan las estructuras de datos y los algoritmos para el almacenamiento y recuperación de información en archivos almacenados en forma externa. Se considerará un archivo como una secuencia de registros, donde cada registro consiste en la misma secuencia de campos. Los campos pueden ser de *longitud fija*, con un número predeterminado de bytes, o de *longitud variable*, con un tamaño arbitrario. Los archivos con registros de longitud fija se suelen utilizar en los sistemas de administración de bases de datos para almacenar datos muy estructurados. Los archivos con registros de longitud variable se usan típicamente para almacenar información de textos; no están disponibles en Pascal. En esta sección, se supondrá

que sólo hay campos de longitud fija; las técnicas empleadas en los registros de longitud fija pueden modificarse con facilidad para trabajar con registros de longitud variable.

Las operaciones con archivos que se considerarán son las siguientes:

1. **INSERTA** un registro determinado en un archivo en particular.
2. **SUPRIME** de un archivo en particular todos los registros que tengan un valor asignado en cada campo de conjunto de campos designado.
3. **MODIFICA** todos los registros de un archivo en particular asignando valores asignados a ciertos campos en los registros que tengan un valor asignado en otro conjunto de campos.
4. **RECUPERA** todos los registros que tengan valores asignados en cada uno de los campos de un conjunto asignado.

Ejemplo 11.3. Por ejemplo, si se tiene un archivo cuyos registros constan de tres campos: *nombre*, *dirección* y *teléfono*. Se podrá preguntar por todos los registros con *teléfono* = 555-1234, insertar el registro (Juan Pérez, calle Manzana 12, 555-1234) o eliminar todos los registros con *nombre* = «Juan Pérez» y *dirección* = «calle Manzana 12». Como otro ejemplo, se puede desear la modificación de todos los registros con *nombre* = «Juan Pérez» para fijar el campo *teléfono* a 555-1234. □

En buena medida, se pueden considerar las operaciones con archivos como si los archivos fueran conjuntos de registros y las operaciones fueran las que se analizaron en los capítulos 4 y 5. Sin embargo, existen dos diferencias importantes. Primero, cuando se habla de archivos en dispositivos de almacenamiento externo, es forzoso utilizar la medición del costo comentada en la sección 11.1, en la evaluación de las estrategias de organización de archivos. Esto es, se supone que los archivos están almacenados en cierta cantidad de bloques físicos, y que el costo de una operación es el número de bloques que se van a leer en memoria principal o escribir desde memoria principal en el almacenamiento externo.

La segunda diferencia es que los registros, siendo tipos de datos concretos en la mayoría de los lenguajes de programación, puede esperarse que tengan apunadores a ellos, mientras que los elementos abstractos de un conjunto, normalmente no tendrán «apunadores» hacia ellos. En particular, los sistemas de bases de datos con frecuencia hacen uso de apunadores a registros cuando organizan datos. La consecuencia de dichos apunadores es que los registros suelen considerarse *adheridos*; no pueden moverse por el almacenamiento, debido a la posibilidad de que un apuntador de algún lugar desconocido no consiga apuntar al registro si éste se ha movido.

Una forma simple de representar apunadores a registros es la siguiente. Cada bloque tiene una *dirección física*, que es la localización del inicio del bloque en el dispositivo de almacenamiento externo; es función del sistema de archivos cuidar las direcciones físicas. Una forma de representar las direcciones de los registros es usar la dirección física del bloque que contiene el registro junto con un *desplazamiento*, que da el número de bytes que preceden en el bloque al principio del registro. Esos pares físicos dirección-desplazamiento pueden almacenarse en campos de tipo «apuntador a registro».

Organización simple

La forma más simple, y también menos eficiente, de realizar las operaciones de archivos anteriores es usar primitivas de lectura y escritura de archivos como las de Pascal. En esta «organización» (que en realidad es una «falta de organización»), los registros pueden almacenarse en cualquier orden. La recuperación de un registro con valores especificados en ciertos campos se efectúa rastreando el archivo y viendo en cada registro si se encuentran los valores especificados. Una inserción en un archivo puede realizarse agregando el registro al final del archivo.

Para la modificación de los registros, se rastrea el archivo y se prueba cada registro para ver si corresponde a los campos designados, y de ser así, se hacen los cambios necesarios en el registro. Una operación de eliminación trabaja casi siempre de la misma forma, pero al encontrar un registro cuyos campos corresponden a los valores requeridos para que la eliminación se lleve a cabo, se debe encontrar la forma de eliminar el registro. Una posibilidad es correr todos los registros siguientes, una posición hacia adelante en sus respectivos bloques, y pasar el primer registro de cada bloque siguiente a la última posición del bloque anterior del archivo. Sin embargo, este enfoque no funciona si los registros están adheridos, porque un apuntador al i -ésimo registro del archivo apuntaría entonces al $(i + 1)$ -ésimo registro.

Si los registros están adheridos, es necesario usar un enfoque distinto. Se marcan de alguna manera los registros eliminados, pero sin mover registros para llenar el hueco, ni insertar un registro nuevo en ese espacio. Así, lógicamente, el registro se elimina, pero su espacio continúa ocupado en el archivo. Esto es necesario para que, si se sigue un apuntador a un registro eliminado, se descubra que el registro apuntado fue eliminado y se tome la acción apropiada, como hacer que el apuntador sea NIL y no se le vuelva a seguir. Dos formas de marcar los registros cuando se eliminan son:

1. Sustituir el registro por algún valor que nunca pueda ser el valor de un registro «real», y cuando se siga un apuntador, suponer que el registro está eliminado si tiene ese valor.
2. Que cada registro tenga un *bit de eliminación*, un solo bit que es 1 en registros que se han eliminado, y 0 en caso contrario.

Aceleración de operaciones con archivos

La desventaja obvia de los archivos secuenciales es que las operaciones son lentas. Cada operación requiere la lectura del archivo completo, y algunos bloques pueden requerir ser reescritos también. Por fortuna, existen organizaciones de archivos que permiten acceder a un registro, leyendo en la memoria principal sólo una pequeña fracción del archivo completo.

Para hacer posible dichas organizaciones, se supone que cada registro de archivo tiene una *clave*, un conjunto de campos que identifica de manera única cada registro. Por ejemplo, el campo *nombre* del archivo *nombre-dirección-teléfono* puede considerarse una clave. Esto es, se puede suponer que no existen simultáneamente

dos registros en el archivo con el mismo valor en el campo *nombre*. La recuperación de un registro dando valores a sus campos clave, es una operación habitual que se realiza con gran facilidad en muchas organizaciones de archivo comunes.

Otro elemento necesario para lograr operaciones rápidas con archivos es la capacidad de acceder a bloques en forma directa, en vez de recorrer en secuencia los bloques que contienen el archivo. Muchas de las estructuras de datos utilizadas para operaciones rápidas con archivos usarán apuntadores a los propios bloques, los cuales son direcciones físicas de los bloques, como se describió antes. Desafortunadamente, no es posible escribir programas en Pascal, ni en muchos otros lenguajes, que se ocupen de los datos en el nivel de bloques físicos y sus direcciones; dichas operaciones se efectúan de ordinario con mandatos del sistema operativo. Sin embargo, se hará una breve descripción informal de la forma de trabajo de las organizaciones que utilizan el acceso directo a bloques.

Archivos con función de dispersión

La dispersión (*hashing*) es una técnica muy utilizada para tener acceso rápido a información almacenada en archivos secundarios. La idea básica es similar a la dispersión abierta estudiada en la sección 4.7. Los registros de un archivo se dividen entre varias *cubetas*, y cada una consiste en una lista enlazada de uno o más bloques de almacenamiento externo. La organización es similar a la presentada en la figura 4.10. Existe una tabla de cubetas que contiene B apuntadores, uno para cada cubeta. En la tabla de cubetas, cada apuntador es la dirección física del primer bloque de la lista enlazada de bloques para esa cubeta.

Las cubetas están numeradas 0, 1, ..., $B - 1$. Una función de dispersión h hace corresponder cada valor de clave con uno de los enteros 0 a $B - 1$. Si x es una clave, $h(x)$ es el número de la cubeta que contiene el registro con la clave x , si tal registro está presente en alguna. Los bloques que forman cada cubeta se encuentran encadenados y forman una lista enlazada. Así, el encabezado del i -ésimo bloque de una cubeta contiene un apuntador a las direcciones físicas del $(i + 1)$ -ésimo bloque. El último bloque de una cubeta contiene un apuntador NIL en su encabezado.

Esta organización se ilustra en la figura 11.8. La principal diferencia entre las figuras 11.8 y 4.10 es que aquí, los elementos almacenados en un bloque de una cubeta no tienen que estar encadenados con apuntadores; sólo los bloques deben estar encadenados.

Si el tamaño de la tabla de cubetas es pequeño, puede quedar almacenada en memoria principal. De otra forma, puede almacenarse en forma secuencial, en tantos bloques como sea necesario. Para buscar el registro con clave x , se calcula $h(x)$, para encontrar el bloque de la tabla de cubetas que contengan el apuntador al primer bloque de la cubeta $h(x)$. Después, se leen los bloques de la cubeta $h(x)$ de manera sucesiva, hasta encontrar el que contenga el registro con la clave x . Si se agotan todos los bloques de la lista enlazada para la cubeta $h(x)$, se concluye que x no es la clave de ningún registro.

Esta estructura es eficiente en operaciones donde se especifican los valores de

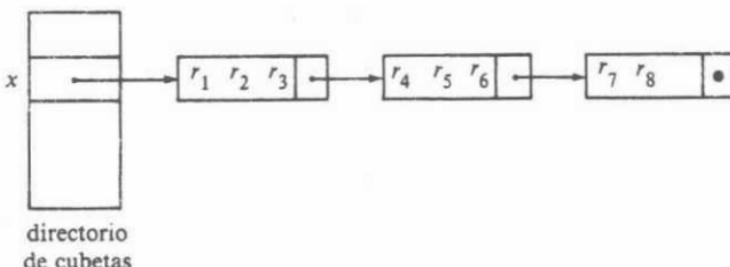


Fig. 11.8. Función de dispersión con cubetas que contienen bloques encadenados.

los campos de la clave, como la recuperación de un registro con un valor especificado en la clave o una inserción de un registro (lo que, como es obvio, especifica el valor de la clave para ese registro). El número promedio de accesos a bloques requerido para una operación que especifica la clave de un registro es aproximadamente el número promedio de bloques en una cubeta, que es n/bk si n es el número de registros, un bloque contiene b registros, y k es el número de cubetas. Así, en promedio, las operaciones basadas en claves son k veces más rápidas con esta organización que con la de archivos no organizados. Lamentablemente, las operaciones no basadas en claves no se aceleran, puesto que se deben examinar esencialmente todas las cubetas durante esas otras operaciones. La única forma general de incrementar la velocidad en operaciones que no se basan en claves es usar índices secundarios, que se analizan en el final de esta sección.

Para insertar un registro con la clave x , primero se verifica si ya existe un registro con la misma clave. De ser así, se informa de un error, dado que se supone que la clave identifica de manera unívoca cada registro. Si no existe el registro con la clave x , se inserta el registro nuevo en el primer bloque de la cadena para la cubeta $h(x)$ en el que pueda caber. Si el registro no cabe en ningún bloque de los existentes para la cubeta $h(x)$, se llama al sistema de archivos para encontrar un bloque nuevo en el cual se pueda colocar el registro. Este bloque nuevo se agrega al final de la cadena de la cubeta $h(x)$.

Para eliminar un registro con clave x , primero debe localizarse, y después se le asigna su bit de eliminación. Otra estrategia posible (que no puede usarse cuando los registros están adheridos) es sustituir el registro eliminado por el último de la cadena de $h(x)$. Si la eliminación del último registro causa el vaciado del último bloque de la cadena se puede devolver el bloque vacío al sistema de archivos para usarlo de nuevo más tarde.

Una organización de archivos de acceso con función de dispersión bien diseñada, requiere sólo unos cuantos accesos a bloques para cada operación de archivo. Si la función de dispersión es buena y el número de cubetas es más o menos igual al número de registros en el archivo, dividido entre el número de registros que pueden caber en un bloque, entonces la cubeta promedio consta de un bloque. Excluyendo el número de accesos a bloques para buscar la tabla de cubetas, una recuperación típica basada en claves hará un solo acceso a un bloque, y una inserción, eliminación o modificación típicas efectuarán dos accesos a bloques. Si el número prome-

dio de registros por cubeta excede en mucho de la cantidad que cabe en un bloque, se puede reorganizar periódicamente la tabla de dispersión incrementando al doble la cantidad de cubetas y dividiendo cada cubeta en dos. Esas ideas se estudiaron al final de la sección 4.8.

Archivos indizados

Otra forma común de organizar un archivo de registros es mantener el archivo clasificado de acuerdo con los valores de las claves. Entonces es posible buscar en el archivo como se haría en un diccionario o en un directorio telefónico, revisando sólo el primer nombre o palabra de cada página. Para facilitar la búsqueda, se crea un segundo archivo, llamado *índice disperso*, que consta de pares (x, b) , donde x es un valor de una clave y b es la dirección física del bloque en el cual el primer registro tiene la clave con valor x . El índice disperso se mantiene clasificado de acuerdo con los valores de las claves.

Ejemplo 11.4. En la figura 11.9, se observa un archivo y su archivo de índice disperso. Se supone que tres registros del archivo principal, o tres pares del archivo de índices, caben en un bloque. Sólo se muestran los valores de las claves de los registros del archivo principal, que se suponen enteros. □

Para recuperar un registro con una clave dada x , primero se busca el par (x, b) en el archivo índice. Lo que en realidad se busca es la z más grande tal que $z \leq x$ y exista un par (z, b) en el archivo índice. Después, la clave x aparece en el bloque b , si está presente en el archivo principal.

Hay varias estrategias para buscar en el archivo de índices; la más simple es la *búsqueda lineal*. Se lee el archivo índice desde el principio hasta encontrar el par (x, b) o el primer par (y, b) , donde $y > x$. En el último caso, el par precedente (z, b) debe tener $z < x$, y si el registro con la clave x está en algún sitio, es en el bloque b' .

La búsqueda lineal sólo es posible para archivos de índices pequeños. Un método más rápido es la *búsqueda binaria*. Supóngase que el archivo de índices está almacenado en los bloques b_1, b_2, \dots, b_n . Para buscar la clave x , se toma el bloque medio $b_{\lceil n/2 \rceil}$ y se compara x con la clave y del primer par de ese bloque. Si $x < y$, se repite la búsqueda en los bloques $b_1, b_2, \dots, b_{\lceil n/2 \rceil - 1}$. Si $x \geq y$, pero x es menor que la clave del bloque $b_{\lceil n/2 \rceil + 1}$, (o si $n = 1$ de modo que no hay tal bloque), se utiliza la búsqueda lineal para ver si x corresponde al primer componente de un par de índi-

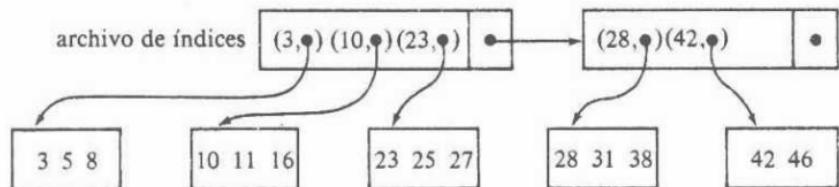


Fig. 11.9. Un archivo principal y su índice disperso.

ces en el bloque $b_{\lfloor n/2 \rfloor}$. De otra forma, se repite la búsqueda en los bloques $b_{\lfloor n/2 \rfloor + 1}, b_{\lfloor n/2 \rfloor + 2}, \dots, b_n$. Con la búsqueda binaria es necesario examinar sólo $\lceil \log_2(n+1) \rceil$ bloques del archivo índice.

Para asignar valor inicial a un archivo indizado, se clasifican los registros de acuerdo con los valores de sus claves, y después se distribuyen los registros entre los bloques, en ese orden; se puede decidir empaquetar tantos como quepan en cada bloque. Otra posibilidad sería preferir dejar espacio para unos registros adicionales que puedan insertarse más tarde. La ventaja es que después es menos probable que las inserciones sobrecarguen el bloque en el cual la inserción se lleva a cabo, con el consecuente requerimiento de tener acceso a los bloques adyacentes. Después de la división de los registros en los bloques de alguna de esas formas, se crea el archivo de índices examinando cada bloque y encontrando la primera clave de cada bloque. Como en el caso del archivo principal, puede dejarse algún espacio para crecimiento posterior en los bloques que contienen el archivo de índices.

Supóngase que se tiene un archivo clasificado con registros almacenados en los bloques b_1, b_2, \dots, b_m . Para insertar un nuevo registro en este archivo, se emplea el archivo de índices para determinar qué bloque b_i debe contener el registro nuevo. Si el registro nuevo debe quedar en b_i , se coloca en el orden adecuado. Después, si el registro nuevo queda como primer registro del bloque b_i , se ajusta el archivo índice.

Si el registro nuevo no cabe en b_i , hay varias estrategias posibles. Tal vez la más simple sea ir al bloque b_{i+1} , el cual puede encontrarse por medio del archivo de índices para ver si el último registro de b_i puede pasarse al principio de b_{i+1} . De ser así, este último registro se pasa a b_{i+1} , y el nuevo puede insertarse en la posición adecuada en b_i . El elemento del archivo de índices correspondiente a b_{i+1} , y posiblemente el de b_i , debe ajustarse en forma apropiada.

Si b_{i+1} también está lleno, o si b_i es el último bloque ($i = m$), entonces se obtiene un bloque nuevo del sistema de archivos. El registro nuevo se inserta en el bloque nuevo a continuación de b_i . Ahora se emplea este mismo procedimiento para insertar un registro que corresponda al nuevo bloque en el archivo de índices.

Archivos no clasificados con índice denso

Otra forma de organizar un archivo de registros es mantener el archivo en orden aleatorio y tener otro, llamado *índice denso*, para ayudar a ubicar los registros. El índice denso consta de pares (x, p) , donde p es un apuntador al registro con la clave x en el archivo principal. Esos pares están clasificados por el valor de la clave, de modo que una estructura como el índice disperso mencionado antes, o el árbol B mencionado en la siguiente sección, puede usarse para ayudar a encontrar las claves en el índice denso.

Con esta organización se emplean los índices densos para encontrar la localización en el archivo principal de un registro con una clave dada. Para insertar un nuevo registro, se sigue la pista del último bloque del archivo principal y ahí se insertan los registros nuevos, tomando un nuevo bloque del sistema de archivos si el último bloque está lleno. También se inserta un apuntador a ese registro en el archivo del

índice denso. Para eliminar un registro, tan sólo se ajusta el bit de eliminación en el registro, y se elimina el elemento correspondiente en el índice denso (tal vez, ajustando también un bit de eliminación).

Índices secundarios

Mientras que las estructuras dispersas e indizadas incrementan bastante la velocidad de las operaciones basadas en claves, ninguna de ellas sirven de ayuda cuando las operaciones implican una búsqueda de registros a partir de valores en campos que no sean campos clave. Si se desea encontrar los registros con valores dados en algún conjunto de campos F_1, \dots, F_k , es necesario un *índice secundario* de aquellos campos. Un índice secundario es un archivo formado de pares (v, p) , donde v es una lista de valores, uno para cada uno de los campos F_1, \dots, F_k , y p es un apuntador a un registro. Puede haber más de un par con una v dada, y cada apuntador asociado indica un registro del archivo principal que tiene a v como lista de valores de los campos F_1, \dots, F_k .

Para recuperar registros dados los valores de los campos F_1, \dots, F_k , en el índice secundario se buscan el o los registros con esa lista de valores. El índice secundario mismo puede organizarse en cualquiera de las formas comentadas para la organización de archivos por valores de claves. Esto es, se presume que v sea la clave de (v, p) .

Por ejemplo, una organización con función de dispersión en realidad no depende de que las claves sean únicas, aunque si hubiera muchos registros con la misma «clave», éstos podrían distribuirse en cubetas de manera no uniforme, con el efecto de que la función de dispersión no incrementaría mucho la velocidad de acceso. En el caso extremo, como cuando hay sólo dos valores para los campos de un índice secundario, todas las cubetas excepto dos estarían vacías, y la tabla de dispersión sólo podría incrementar la velocidad de las operaciones en un factor de dos a lo sumo, sin importar cuántas cubetas haya. De modo semejante, un índice disperso no requiere que las claves sean únicas, pero si no lo son, puede haber dos o más bloques del archivo principal que tengan la misma «clave» menor, y todos esos bloques se recorren al buscar registros con ese valor.

Con cualquier organización, de índice disperso o de índice con función de dispersión para el archivo de índices secundarios, puede desearse ahorrar espacio agrupando todos los registros con el mismo valor. Esto es, los pares $(v, p_1), (v, p_2), \dots, (v, p_m)$ pueden reemplazarse por v seguida de la lista p_1, p_2, \dots, p_m .

Cabe preguntarse si el mejor tiempo de respuesta a las operaciones aleatorias no puede obtenerse al crear un índice secundario para cada campo, o aun para todos los subconjuntos de los campos. Lamentablemente, hay un precio por cada índice secundario que se deseé crear. Primero, está el espacio necesario para almacenar el índice secundario, y eso puede ser o no un problema, dependiendo de si el espacio es una prioridad.

Además, cada índice secundario creado disminuye la velocidad de todas las inserciones y eliminaciones. La razón es que al insertar un registro se debe insertar también un elemento para ese registro en cada índice secundario, para que los índices

secundarios sigan representando exactamente el archivo. La actualización de un índice secundario requiere por lo menos dos accesos a bloques, ya que se debe leer y escribir un bloque. Sin embargo, puede requerir mucho más de dos accesos a bloques, ya que es necesario encontrar ese bloque, y cualquier organización empleada para el archivo del índice secundario requerirá unos cuantos accesos adicionales, en promedio, para encontrar cualquier bloque. Algo parecido sucede en cada eliminación. La conclusión es que la selección de los índices secundarios requiere seguir un criterio, puesto que debe determinarse qué conjuntos de campos se especificarán en operaciones muy frecuentes de modo que el tiempo ahorrado teniendo esos índices secundarios disponibles compense los costos de la actualización de los índices en cada inserción y eliminación.

11.4 Árboles de búsqueda externa

La estructura de datos tipo árbol presentada en el capítulo 5 para representar conjuntos puede usarse también para representar archivos externos. El árbol B, una generalización de los árboles 2-3 analizados en el capítulo 5, es especialmente adecuada para almacenamiento externo, y se ha convertido en la organización estándar para índices en sistemas de bases de datos. Esta sección presenta las técnicas básicas de recuperación, inserción y eliminación de información en árboles B.

Árboles de búsqueda múltiple

Un árbol de búsqueda m -ario es una generalización del árbol binario de búsqueda en el cual cada nodo tiene como máximo m hijos. Generalizando la propiedad del árbol binario de búsqueda, se requiere que si n_1 y n_2 son dos hijos de algún nodo, y n_1 está a la izquierda de n_2 , los elementos descendientes de n_1 sean todos menores que los de n_2 . Las operaciones MIEMBRO, INSERTA y SUPRIME de un árbol de búsqueda m -ario se realizan con una generalización de las operaciones en árboles binarios de búsqueda, estudiadas en la sección 5.1.

Sin embargo, aquí interesa el almacenamiento de registros en archivos, donde los archivos se depositan en bloques de almacenamiento externo. La adaptación correcta de la idea de árboles múltiples consiste en pensar en los nodos como bloques físicos. Un nodo interior contiene apuntadores a sus m hijos y también contiene $m-1$ claves que separan los descendientes del hijo. Los nodos hojas son bloques también, y contienen los registros del archivo principal.

Si se emplea un árbol binario de búsqueda de n nodos para representar un archivo almacenado en forma externa, pueden requerirse, en promedio, $\log_2 n$ accesos a bloques para recuperar un registro del archivo. En cambio, si se utiliza un árbol de búsqueda m -ario para representar el archivo, requerirá, en promedio, sólo $\log_m n$ accesos a bloques para recuperar un registro. Para $n = 10^6$, el árbol binario de búsqueda puede requerir cerca de 20 accesos a bloques, mientras que un árbol de 128 caminos requerirá sólo 3.

No se puede hacer m arbitrariamente grande, pues cuanto mayor sea m , mayor deberá ser el bloque. Más aún, es más lento leer y procesar un bloque más grande, así que hay un valor óptimo de m que reduce la cantidad de tiempo necesario para recorrer un árbol m -ario de búsqueda externa. En la práctica, se obtiene un valor cercano al mínimo para un amplio intervalo de m . (Véase Ejercicio 11.18.)

Arboles B

Un árbol B es una clase especial de árbol m -ario balanceado que permite recuperar, eliminar e insertar registros de un archivo externo con buen rendimiento en el peor caso. Es una generalización de los árboles 2-3 de la sección 5.4. Formalmente, un *árbol B de orden m* es un árbol de búsqueda m -ario con las siguientes propiedades:

1. La raíz es una hoja o tiene al menos dos hijos.
2. Cada nodo, excepto la raíz y las hojas, tiene entre $[m/2]$ y m hijos.
3. Cada camino desde la raíz hasta una hoja tiene la misma longitud.

Obsérvese que todo árbol 2-3 es un árbol B de orden 3. La figura 11.10 muestra un árbol B de orden 5, en el cual se supone que caben como máximo tres registros en un bloque hoja.

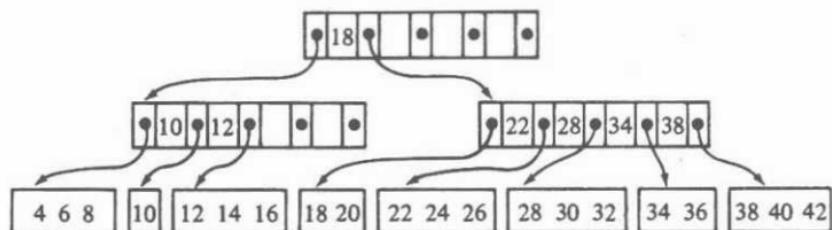


Fig. 11.10. Árbol B de orden 5.

Se puede considerar un árbol B como un índice jerárquico en el cual cada nodo ocupa un bloque en el almacenamiento externo. La raíz del árbol B es el índice de primer nivel. Cada nodo que no es hoja en el árbol B tiene la forma

$$(p_0, k_1, p_1, k_2, p_2, \dots, k_n, p_n)$$

donde p_i es un apuntador al i -ésimo hijo del nodo, $0 \leq i \leq n$, y k_i es una clave, $1 \leq i \leq n$. Las claves dentro de un nodo están clasificadas en orden tal que $k_1 < k_2 < \dots < k_n$. Todas las claves del subárbol al que apunta p_0 son menores que k_1 . Para $1 \leq i < n$, todas las claves del subárbol apuntado por p_i tienen valores mayores o iguales que k_i y menores que k_{i+1} . Todas las claves en el subárbol apuntado por p_n son mayores que k_n .

Hay varias formas de organizar las hojas. Aquí se supondrá que el archivo principal está almacenado sólo en las hojas, y que cada hoja ocupa un bloque.

Recuperación

Para recuperar un registro r con clave x , se sigue el camino desde la raíz hasta la hoja que contiene a r , si es que existe en el archivo. Se sigue este camino pasando sucesivamente nodos interiores ($p_0, k_1, p_1, \dots, k_n, p_n$) del almacenamiento externo a la memoria principal y buscando la posición de x relativa a las claves k_1, k_2, \dots, k_n . Si $k_i \leq x < k_{i+1}$, a continuación se busca el nodo apuntado por p_i y se repite el proceso. Si $x < k_1$, se aplica p_0 para alcanzar el siguiente nodo; si $x \geq k_n$, se utiliza p_n . Cuando este proceso llega a una hoja, debe buscarse el registro con la clave x . Si el número de elementos de un nodo es pequeño, es posible utilizar la búsqueda lineal dentro del nodo, de otra forma, conviene utilizar una búsqueda binaria.

Inserción

La inserción en un árbol B es similar a la inserción en árboles 2-3. Para insertar un registro r con clave x en un árbol B, primero se aplica el procedimiento de búsqueda para localizar la hoja L a la cual corresponde r . Si existe espacio suficiente para r en L , se inserta r en el orden correspondiente. En este caso no se requieren modificaciones a los antecesores de L .

Si no hay espacio para r en L , es necesario pedir al sistema de archivos un bloque nuevo L' para pasar la segunda mitad de los registros de L a L' , insertando r en el lugar adecuado en L o L' .[†] Sea el nodo P el padre de L ; P es conocido, ya que el procedimiento de búsqueda siguió un camino desde la raíz hasta L , pasando por P . Se aplica ahora el procedimiento de inserción recursivamente para colocar en P una clave k' y un apuntador l' a L' ; k' y l' se insertan inmediatamente después de la clave y y del apuntador de L . El valor de k' es el de la clave más pequeña de L' .

Si P ya tiene m apuntadores, la inserción de k' y l' dentro de P hará que P se dividiera y requiera la inserción de una clave y un apuntador en el padre de P . Los efectos de esta inserción pueden transmitirse a los antecesores del nodo L en dirección a la raíz, por el camino que se siguió con el procedimiento de búsqueda original. Incluso puede ser necesario dividir la raíz, en cuyo caso se creará una nueva raíz con las dos mitades de la raíz anterior como sus dos hijos. Esta es la única situación en la que un nodo puede tener menos de $m/2$ hijos.

Eliminación

Para eliminar un registro r con una clave x , primero se encuentra la hoja L que contiene a r . Despues, se elimina r de L , si existe. Si r es el primer registro en L , es necesario ir a P , el padre de L , para ajustar el valor de la clave en la entrada de P para que L sea el nuevo valor de la primera clave de L . Sin embargo, si L es el primer hijo de P , la primera clave de L no se registra en P , sino que aparece en alguno de

[†] Esta estrategia es la más sencilla de las varias respuestas aplicables a la situación en que ha de dividirse un bloque. En los ejercicios se mencionan algunas otras opciones que proporcionan una mayor ocupación media de los bloques con una inserción más laboriosa.

los antecesores de P , de forma específica, en el menor antecesor A tal que L no sea el descendiente más a la izquierda de A . Así pues, se debe propagar el cambio en la menor clave de L de vuelta por la trayectoria de la raíz a L .

Si L queda vacío después de la eliminación, se devuelve al sistema de archivos †; después se ajustan las claves y los apuntadores de P para reflejar la eliminación de L . Si el número de hijos de P ahora es menor que $m/2$, se examina el nodo P' situado inmediatamente a la izquierda (o a la derecha) de P en el mismo nivel del árbol. Si P' tiene por lo menos $[m/2] + 1$ hijos, se distribuyen las claves y los apuntadores de P y P' en forma equitativa entre ambos, cuidando, por supuesto, el orden de clasificación, de manera que ambos nodos tengan por lo menos $[m/2]$ hijos. Después, se modifican los valores de las claves para P y P' en el padre de P y, si fuera necesario, se propagan recursivamente los efectos de este cambio a todos los antecesores de P que se vean afectados.

Si P' tiene exactamente $[m/2]$ hijos, es bueno combinar P y P' en un solo nodo con $2[m/2] - 1$ hijos (esto es, m hijos a lo sumo). Despues, se deben quitar en el padre la clave y el apuntador correspondientes a P' . Esta eliminación puede hacerse con una aplicación recursiva del procedimiento de eliminación.

Si los efectos de la eliminación se propagan hasta la raíz, puede requerirse la combinación de los dos únicos hijos de ésta. En ese caso, el nodo combinado resultante quedará como nueva raíz, y la raíz anterior puede devolverse al sistema de archivos. La altura del árbol B se verá reducida en uno.

Ejemplo 11.5. Considérese el árbol B de orden 5 de la figura 11.10. Insertar el registro con clave 23 en este árbol produce el árbol B de la figura 11.11. Para insertar 23, se debe partir el bloque que contiene 22, 23, 24 y 26, ya que se ha supuesto que un bloqué se llena con tres registros. Los dos registros más pequeños permanecen en el mismo bloqué y los otros dos se colocan en un bloqué nuevo. Un par apuntador-clave del nodo nuevo debe insertarse en el padre, que entonces se divide, porque no puede contener seis apuntadores. La raíz recibe el par apuntador-clave para el nodo nuevo, pero no se divide, porque tiene capacidad en exceso.

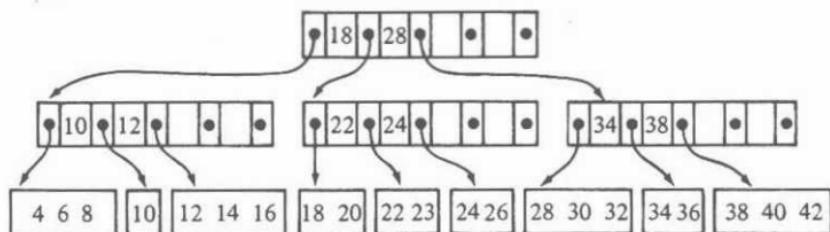


Fig. 11.11. Árbol B después de la inserción.

† Se pueden utilizar estrategias para evitar que los bloques hoja queden totalmente vacíos. Como ejemplo, a continuación se describe un esquema que impide que los nodos interiores queden ocupados en menos de la mitad de su capacidad, y esta técnica se puede aplicar también a las hojas, con un valor de m igual al mayor número de registros que caben en un bloqué.

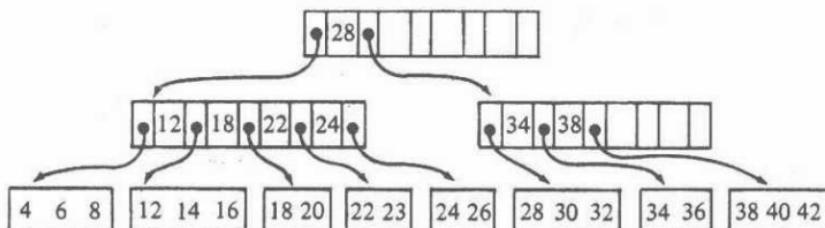


Fig. 11.12. Árbol B después de la eliminación.

La eliminación del registro 10 del árbol B de la figura 11.11 produce el árbol B de la figura 11.12. Aquí, el bloque que contiene a 10 se descarta; su padre queda con sólo dos hijos, y el hermano derecho tiene el número mínimo, tres; así, se combina el padre con su hermano, haciendo un nodo con cinco hijos. □

Análisis del tiempo de las operaciones con árboles B

Supóngase que se tiene un archivo con n registros organizado en un árbol B de orden m . Si cada hoja contiene en promedio b registros, el árbol tiene cerca de $[n/b]$ hojas. Los caminos más largos posibles se producen si cada nodo interior tiene la menor cantidad posible de hijos, esto es, $m/2$. En este caso, habrá unos $2[n/b]/m$ padres de hojas, $4[n/b]/m^2$ padres de padres de hojas, y así sucesivamente.

Si hay j nodos en el camino que va de la raíz a una hoja, entonces $2^{j-1}[n/b]/m^{j-1} \geq 1$, pues si no habría menos de un nodo en el nivel de la raíz. Por tanto, $[n/b] \geq (m/2)^{j-1}$, y $j \leq 1 + \log_{m/2} [n/b]$. Por ejemplo, si $n = 10^6$, $b = 10$ y $m = 100$, entonces $j \leq 3.5$. Obsérvese que b no es el número máximo de registros que se pueden poner en un bloque, sino un promedio, o número esperado. Sin embargo, redistribuyendo los registros entre bloques vecinos cuando uno se vacía hasta menos de la mitad, se puede asegurar que b es por lo menos la mitad del valor máximo. Obsérvese también que se ha supuesto que cada nodo interior tiene el mínimo número posible de hijos; en la práctica, el nodo interior promedio tendrá más que el mínimo, por lo que el análisis anterior resulta conservador.

Para una inserción o eliminación, se necesitan j accesos a bloques para localizar la hoja apropiada. El número exacto de accesos adicionales a bloques necesario para culminar la inserción o la eliminación, y distribuir sus efectos por el árbol, es difícil de calcular. La mayor parte de las veces sólo es necesario escribir de nuevo un bloque, la hoja que contiene al bloque de interés. Así, $2 + \log_{m/2} [n/b]$ puede tomarse como el número aproximado de accesos a bloques para inserción o eliminación.

Comparación de métodos

Se ha estudiado la función de dispersión, los índices dispersos y los árboles B como métodos posibles de organización de archivos externos. Es interesante comparar en

cada método el número de accesos a bloques relacionados con una operación con archivos.

La función de dispersión suele ser el método más rápido de los tres, y requiere en promedio dos accesos a bloques por cada operación (excluyendo los accesos requeridos para buscar la tabla de cubetas), si el número de cubetas es lo bastante grande para que la cubeta típica use sólo un bloque. Sin embargo, con la función de dispersión no es fácil acceder a los registros en el orden de clasificación.

Un índice disperso en un archivo de n registros permite que las operaciones sobre archivos se realicen en unos $2 + \log(n/bb')$ accesos a bloques mediante la búsqueda binaria; aquí, b es el número de registros que caben en un bloque y b' es el número de pares apuntador-clave que caben en un bloque del archivo índice. Los árboles B permiten operaciones con archivos en $2 + \log_{m/2} [n/b]$ accesos a bloques, donde m , el grado máximo de los nodos interiores, es aproximadamente b' . Los índices dispersos y los árboles B permiten el acceso a los registros en el orden de clasificación.

Todos estos métodos son muy buenos comparados con la búsqueda secuencial obvia en un archivo. Las diferencias de tiempo entre ellos, no obstante, son pequeñas y difíciles de determinar analíticamente, especialmente considerando que los parámetros importantes, como la longitud esperada del archivo y la razón de ocupación de bloques, son difíciles de predecir.

Parece ser que los árboles B se están popularizando con rapidez como medio de acceso a archivos en sistemas de bases de datos. En parte, se debe a su capacidad para manejar consultas de registros con claves en un intervalo determinado (lo que aprovecha el hecho de que los registros aparecen ordenados en el archivo principal). El índice disperso también maneja con eficiencia dichas consultas, pero es casi seguro que es menos eficiente que los árboles B. Intuitivamente, la razón de que los árboles B sean superiores a los índices dispersos es que se puede considerar un árbol B como un índice disperso sobre un índice disperso sobre un índice disperso, y así sucesivamente (aunque rara vez se necesitan más de tres niveles de índices).

Los árboles B también funcionan relativamente bien cuando se usan como índices secundarios, donde las «claves» no definen únicamente un registro. Aunque los registros con un valor dado en los campos designados de un índice secundario se extienden sobre muchos bloques, se pueden leer todos con un número de accesos a bloque igual al número de bloques que contienen esos registros, más el número de sus antecesores en el árbol B. En comparación, si esos registros, más otro grupo de tamaño similar tienen la misma función de dispersión y llegan a la misma cubeta, entonces la recuperación de cualquier grupo de una tabla de dispersión puede requerir un número de accesos a bloques cercano al doble del número de bloques en los cuales puede caber cada grupo. Es factible que haya otras razones en favor de los árboles B, como su rendimiento cuando varios procesos tienen acceso simultáneamente a la estructura, pero eso está fuera del alcance de este libro.

Ejercicios

- 11.1** Escribase un programa *concatena* que tome una secuencia de nombres de archivos como argumentos y escriba el contenido de los mismos en la salida estándar, concatenando así los archivos.
- 11.2** Escribase un programa *incluye* que copie su entrada en su salida, excepto cuando encuentre una línea de la forma `#incluye archivo`, en cuyo caso reemplaza esta línea por el contenido del archivo mencionado. Obsérvese que los archivos incluidos también pueden contener proposiciones `#incluye`.
- 11.3** ¿Cómo se comporta el programa del ejercicio 11.2 cuando un archivo se incluye a sí mismo?
- 11.4** Escribase un programa *compara* que compare dos archivos, registro por registro, para determinar si son iguales.
- *11.5** Reescribase el programa de comparación de archivos del ejercicio 11.4 con el algoritmo SCL de la sección 5.6 para encontrar la subsecuencia común más larga de registros en ambos archivos.
- 11.6** Escribase un programa *encuentra* que tome dos argumentos que consten de una cadena y un nombre de archivo, e imprima todas las líneas del archivo que contengan la cadena como una subcadena. Por ejemplo, si la cadena es «*ado*» y el archivo es una lista de palabras, entonces *encuentra* imprime todas las palabras que contienen el trígrama «*ado*».
- 11.7** Escribase un programa que lea un archivo y escriba en su salida estándar los registros del archivo clasificados.
- 11.8** ¿Cuáles son las primitivas que Pascal ofrece para tratar con archivos externos? ¿Cómo pueden mejorarse?
- *11.9** Supóngase que se maneja una clasificación en varias fases con tres archivos, y en la i -ésima fase se crea un archivo con r_i fragmentos de longitud l_i . En la n -ésima fase, se desea un fragmento en uno de los archivos y ninguno en los otros dos. Explíquese por qué cada uno de los siguientes enunciados debe ser cierto.
- $l_i = l_{i-1} + l_{i-2}$ para $i \geq 1$, donde l_0 y l_{-1} se consideran las longitudes de los fragmentos en los dos archivos ocupados inicialmente.
 - $r_i = r_{i-2} - r_{i-1}$ (o, en forma equivalente, $r_{i-2} = r_{i-1} + r_i$ para $i \geq 1$), donde r_0 y r_{-1} son el número de fragmentos en los dos archivos iniciales.
 - $r_n = r_{n-1} = 1$ y, por tanto, r_n, r_{n-1}, \dots, r_1 , forma una sucesión de Fibonacci.
- *11.10** ¿Qué condición debe agregarse a las del ejercicio 11.9 para hacer posible una clasificación en varias fases,
- con fragmentos iniciales de longitud uno (esto es, $l_0 = l_{-1} = 1$)?
 - la ejecución para k fases, pero con fragmentos iniciales diferentes a uno permitido?

Sugerencia. Considérense unos cuantos ejemplos, como $l_n = 50$, $l_{n-1} = 31$ o $l_n = 50$, $l_{n-1} = 32$.

****11.11** Generalíicense los ejercicios 11.9 y 11.10 a clasificaciones de varias fases con más de tres archivos.

****11.12** Muéstrese que:

- a) Cualquier algoritmo de clasificación externa que utilice sólo una cinta como almacenamiento externo debe requerir un tiempo $\Omega(n^2)$ para clasificar n registros.
- b) Un tiempo $O(n \log n)$ es suficiente si hay dos cintas para almacenamiento externo.

11.13 Supóngase que se tiene un archivo externo de arcos dirigidos $x \rightarrow y$ que forma un grafo acíclico, y que no hay suficiente espacio en la memoria interna para contener el conjunto completo de vértices o aristas al mismo tiempo.

- a) Escribábase un programa de clasificación topológica externa que escriba un ordenamiento lineal de vértices, de modo que si $x \rightarrow y$ es un arco dirigido, el vértice x aparezca antes de y en el ordenamiento lineal.
- b) ¿Cuál es la complejidad de tiempo y espacio de este programa como una función del número de accesos a bloques?
- c) ¿Qué haría este programa si el grafo dirigido fuera cíclico?
- **d) ¿Cuál es el número mínimo de accesos a bloque necesarios para clasificar topológicamente un grafo dirigido acíclico almacenado externamente?

11.14 Supóngase que se tiene un archivo de un millón de registros, donde cada registro ocupa 100 bytes. Los bloques son de 1000 bytes de longitud, y un apuntador a un bloque ocupa 4 bytes. Diséñese una organización con función de dispersión para este archivo. ¿Cuántos bloques se necesitan para la tabla de cubetas y las cubetas?

11.15 Diséñese una organización con árboles B para el archivo del ejercicio 11.14.

11.16 Escribanse programas para implantar las operaciones RECUPERA, INSERTA, SUPRIME y MODIFICA en

- a) archivos con función de dispersión,
- b) archivos indizados,
- c) archivos con árboles B.

11.17 Escribábase un programa para encontrar el k -ésimo elemento más grande en

- a) un archivo con índice disperso
- b) un archivo con árbol B

- *11.18 Supóngase que se necesitan $a + bm$ milisegundos para leer un bloque que contiene un nodo de un árbol m -ario de búsqueda, y $c + d \log_2 m$ milisegundos para procesar cada nodo en memoria interna. Si hay n nodos en el árbol, es necesario leer cerca de $\log_m n$ nodos para localizar un registro dado. Por tanto, el tiempo total requerido para encontrar un registro dado en el árbol es

$$(\log_m n)(a + bm + c + d \log_2 m) = (\log_2 n)((a + c + bm)/\log_2 m) + d$$

milisegundos. Háganse estimaciones razonables para los valores de a , b , c y d , y represéntese gráficamente esta cantidad como una función de m . ¿Para qué valor de m se obtiene el mínimo?

- *11.19 Un árbol B^* es un árbol B en el que cada nodo interno está lleno en dos terceras partes por lo menos (en vez de la mitad). Diséñese un esquema de inserción para árboles B^* que retrase la división de nodos internos hasta que dos nodos hermanos estén llenos. Los dos nodos hermanos pueden entonces dividirse en tres, cada uno lleno en dos terceras partes. ¿Qué ventajas y desventajas tienen los árboles B^* en relación con los árboles B ?
- *11.20 Cuando la clave de un registro es una cadena de caracteres, se puede ahorrar espacio almacenando sólo un prefijo de la clave como separador clave de cada nodo interno de un árbol B . Por ejemplo, «gato» y «perro», pueden separarse por el prefijo «g» o «ga» de «gato». Diséñese un algoritmo de inserción en árboles B que utilice prefijos de claves como separadores que sean siempre lo más cortos posible.
- *11.21 Supóngase que en cierto archivo las operaciones de inserción y eliminación se efectúan en una fracción p del tiempo, y en el tiempo $1-p$ restante se realizan recuperaciones donde se especifica exactamente un campo. Hay k campos en los registros, y una recuperación especifica el i -ésimo campo con probabilidad q_i . Supóngase también que una recuperación requiere a milisegundos si no hay índice secundario para el campo especificado, y b milisegundos, si lo hay, y que una inserción o eliminación requiere $c + sd$ milisegundos, donde s es el número de índices secundarios. Determínese, como una función de a , b , c , d , p y las q_i , qué índices secundarios deben crearse para el archivo, de manera que el tiempo promedio por operación sea minimice.
- **11.22 Supóngase que las claves son de un tipo que puede ordenarse linealmente, como los números reales, y que se conoce la distribución de probabilidades con que aparecerán las claves de valores dados en el archivo. Se puede aprovechar este conocimiento para mejorar una búsqueda binaria cuando se busca una clave en un índice disperso. Un esquema, llamado *búsqueda por interpolación*, usa esta información estadística para predecir dónde es más probable que se encuentre una clave x , en el intervalo de bloques de índices B_1, \dots, B_p , a los cuales se ha limitado la búsqueda. Proporcíonese

- a) un algoritmo para aprovechar el conocimiento estadístico en esta forma, y
 b) una demostración de que $O(\log \log n)$ accesos a bloques son suficientes, en promedio, para encontrar una clave.
- 11.23** Supóngase que se tiene un archivo externo de registros, y que cada registro consta de una arista de un grafo G y un costo asociado a esa arista.
- Escríbase un programa para construir un árbol abarcador de costo mínimo para G , suponiendo que existe suficiente memoria para almacenar todos los vértices de G , pero no todas las aristas.
 - ¿Cuál es la complejidad de tiempo de ese programa como una función del número de vértices y aristas?
- Sugerencia.* Un enfoque posible de este problema es mantener en memoria un bosque con los componentes conexos actuales. Cada arista es leída y procesada como sigue: si la siguiente arista tiene extremos en dos componentes distintos, se agrega y se mezclan los componentes. Si la arista crea un ciclo en un componente ya existente, se agrega y se elimina la arista de mayor costo del ciclo (que puede ser la arista actual). Este enfoque es similar al algoritmo de Kruskal, pero no requiere clasificar las aristas, lo cual es un aspecto importante de este problema.
- 11.24** Supóngase que se tiene un archivo que contiene una secuencia de números positivos y negativos a_1, a_2, \dots, a_n . Escríbase un programa $O(n)$ para encontrar una subsecuencia contigua a_i, a_{i+1}, \dots, a_j que tenga la suma mayor $a_i + a_{i+1} + \dots + a_j$ de cualquier subsecuencia.

Notas bibliográficas

Como material adicional sobre clasificación externa, véase Knuth [1973]. Material posterior sobre estructuras de datos externas y su uso en sistemas de bases de datos puede encontrarse en esa obra y en Ullman [1982] y Wiederhold [1982]. La clasificación en varias fases se estudia en Shell [1971]. El esquema de intercalación con seis buffers de la sección 11.2 es de Friend [1956], y el de cuatro, de Knuth [1973].

La selección de índices secundarios, de la cual el ejercicio 11.21 es una simplificación, se analiza en Lum y Ling [1970] y Schkolnick [1975]. Los árboles B se presentaron originalmente en Bayer y McCreight [1972]. En Comer [1979], se examinan muchas variantes, y en Gudes y Tsur [1980] se evalúa el rendimiento en la práctica.

La información acerca del ejercicio 11.12, clasificación con una y dos cintas, puede encontrarse en Floyd y Smith [1973]. El ejercicio 11.22 sobre búsqueda por in-

terpolación se analiza con detalle en Yao y Yao [1976], y Perl, Itai y Avni [1978].

Una implantación excelente del enfoque sugerido en el ejercicio 11.23 para el problema del árbol abarcador externo de costo mínimo fue estudiada por V. A. Vyssotsky, alrededor de 1960 (sin publicar). El ejercicio 11.24 se debe a M. I. Shamos.

12

Administración de memoria

En este capítulo se analizan las estrategias básicas para reutilizar el espacio en memoria o compartirlo entre objetos distintos que crecen y se contraen de manera arbitraria. Por ejemplo, se estudiarán los métodos que mantienen listas enlazadas de espacio disponible, y técnicas de «recolección de basura», que sirven para conocer la disponibilidad de espacio sólo cuando parece que se ha terminado el espacio disponible.

12.1 Aspectos de la administración de memoria

En la operación de sistemas de cómputo existen muchas situaciones en las que se administra un recurso limitado de memoria, es decir, se comparte entre varios «competidores». Un programador que no se ocupe de la realización de programas del sistema (compiladores, sistemas operativos, etcétera) tal vez no perciba dichas actividades, debido a que suelen realizarse «entre bastidores». Como ejemplo, los programadores de Pascal saben que el procedimiento *new(p)* hará que el apuntador *p* señale hacia un objeto nuevo del tipo correcto; pero, ¿de dónde procede el espacio para el objeto? El procedimiento *new* tiene acceso a una región grande de memoria, conocida como «estructura dinámica» (*heap*), que las variables del programa no usan. De esa región, se selecciona un bloque no utilizado de bytes consecutivos suficiente para contener un objeto del tipo al que apunta *p*, y se hace que *p* contenga la dirección del primer byte de ese bloque. Pero ¿cómo sabe el procedimiento *new* qué bytes de la memoria están «desocupados»? La sección 12.4 sugiere la respuesta.

Aún más misterioso es lo que sucede si se modifica el valor de *p*, bien por una asignación o por otra llamada a *new(p)*. El bloque de memoria al que apunta *p* puede ser ahora *inaccesible*, en el sentido de que no hay forma de llegar a él mediante las estructuras de datos del programa, y se puede reutilizar su espacio. Por otro lado, antes de cambiar *p*, su valor pudo haberse copiado en otra variable. En ese caso, el bloque de memoria será parte todavía de las estructuras de datos del programa. ¿Cómo se puede saber si un bloque de la región de memoria utilizada por el procedimiento *new* ya no es requerido por el programa?

El tipo de administración de memoria que se efectúa en Pascal sólo es uno más. Por ejemplo, en algunas situaciones, como Pascal, objetos de tamaños distintos comparten el mismo espacio de memoria; en otras situaciones, todos los objetos que comparten el espacio son del mismo tamaño. Esta distinción con respecto a los tamaños

de los objetos es una forma de clasificar las clases de problemas de administración de memoria a que uno debe enfrentarse. A continuación se presentan algunos ejemplos más.

1. En el lenguaje de programación LISP, el espacio de memoria se divide en *celdas* que, en esencia, son registros que constan de dos campos; cada campo puede contener un *átomo* (un objeto del tipo elemental, como puede ser un entero) o un apuntador a una celda. Los átomos y apuntadores son del mismo tamaño, así que todas las celdas requieren el mismo número de bytes. Todas las estructuras de datos conocidas pueden construirse a partir de esas celdas. Por ejemplo, las listas enlazadas de átomos pueden usar los primeros campos de las celdas para contener átomos, y los segundos campos, para contener apuntadores a las siguientes celdas de la lista. Los árboles binarios pueden representarse utilizando el primer campo de cada celda para apuntar al hijo izquierdo, y el segundo, para apuntar al hijo derecho. Al ejecutar un programa en LISP, el espacio de memoria empleado para contener una celda puede ser a la vez parte de estructuras distintas en momentos diferentes, ya sea porque una celda se mueve entre estructuras, o porque se separa de todas las estructuras y su espacio se utiliza de nuevo.
2. Un sistema de archivos, en general, divide los dispositivos de almacenamiento secundario, como los discos, en bloques de longitud fija. Por ejemplo, UNIX siempre usa bloques de 512 bytes. Los archivos se almacenan en una secuencia de bloques (que no son necesariamente consecutivos). Conforme se crean y destruyen archivos, los bloques del almacenamiento secundario quedan disponibles para ser utilizados de nuevo.
3. Un sistema operativo de multiprogramación típico permite que varios programas comparten la memoria principal al mismo tiempo. Cada programa requiere una cantidad determinada de memoria, la cual es conocida por el sistema operativo, y este requisito es parte de la solicitud de servicio emitida cuando se desea ejecutar el programa. Mientras en los ejemplos (1) y (2) los objetos que comparten memoria (celdas y bloques, respectivamente) eran todos del mismo tamaño, distintos programas requieren cantidades diferentes de memoria. Así, cuando termina un programa que usa 100K bytes, puede reemplazarse por otros dos que manejen 50K cada uno, o uno 20K y otro 70K (con 10K no utilizados). Como solución distinta, los 100K bytes liberados a la terminación del programa pueden combinarse con los 50K adyacentes que estén sin utilizar, y entonces puede ejecutarse un programa que requiera hasta 150K. Otra posibilidad es que ningún programa nuevo quepa en el espacio liberado, y que 100K bytes quede momentáneamente sin utilizar.
4. Hay muchos lenguajes de programación, como SNOBOL, APL o SETL, que asignan espacio a objetos de tamaño arbitrario. A esos objetos, que son valores asignados a variables, se les asigna un bloque de espacio de un gran bloque de memoria, que suele denominarse *estructura dinámica (heap)*. Cuando cambia el valor de una variable, al valor nuevo se le asigna un espacio en la estructura dinámica, y un apuntador para que la variable apunte al nuevo valor. Es posible que el valor anterior de la variable quede ahora sin utilizar, y su espacio se pueda volver a utilizar. Sin embargo, los lenguajes como SNOBOL o SETL realizan asig-

naciones como $A = B$ haciendo que el apuntador de A apunte al mismo objeto que el apuntador de B ; si A o B fueran reasignados, el objeto anterior no se liberaría y su espacio no podría solicitarse.

Ejemplo 12.1. En la figura 12.1(a) se observa la estructura dinámica que puede manejar en un programa en SNOBOL con las variables A , B y C . El valor de cualquier variable en SNOBOL es una cadena de caracteres y, en este caso, el valor de A y B es «BUENOS DIAS» y el valor de C es «PASA LA SAL».

Se ha elegido la representación de cadenas de caracteres por medio de apunadores a bloques de memoria en la estructura dinámica. Esos bloques tienen sus dos primeros bytes (el número 2 es un valor típico que podría cambiarse) dedicados a un entero que da la longitud de la cadena; por ejemplo, «BUENOS DIAS» tiene longitud 11, contando el espacio entre palabras, así que el valor de A (y de B) ocupa 13 bytes.

Si el valor de B se cambia por «BUENAS NOCHES», se puede encontrar un bloque vacío en el montón con 15 bytes para almacenar el nuevo valor de B , incluyendo los dos bytes de la longitud. Se hace que el apuntador de B apunte al nuevo valor, como se muestra en la figura 12.1(b). El bloque que contiene el entero 11 y «BUENOS DIAS» aún es útil, pues A continúa apuntando a él. Si el valor de A cambia, ese bloque quedará sin uso y puede volver a utilizarse. La forma de saber que no hay apunadores a tales bloques es un tema importante en este capítulo □

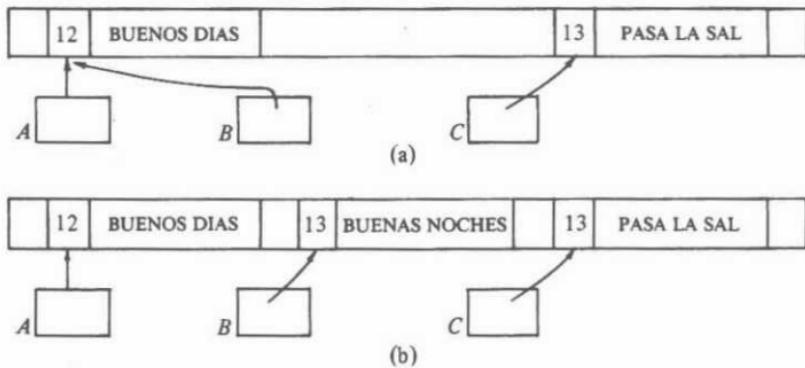


Fig. 12.1. Variables de cadenas en una estructura dinámica.

En los cuatro ejemplos anteriores, se pueden observar diferencias a lo largo, por lo menos, de dos «dimensiones» ortogonales. El primer aspecto es si los objetos que se encuentran compartiendo el espacio son de la misma longitud o no. En los dos primeros ejemplos, los programas en LISP y el almacenamiento de archivos, los objetos, celdas tipo LISP en un caso, y bloques con partes de archivos, en el otro, son del mismo tamaño. Este hecho permite ciertas simplificaciones del problema de la administración de memoria. Por ejemplo, en la realización en LISP, una región de memoria se divide en espacios, cada uno de los cuales puede contener exactamente una celda. El problema de la administración es encontrar espacios vacíos para ubi-

car las celdas recién creadas y nunca se necesita almacenar una celda en una posición en que se translapen dos espacios. Igualmente, en el segundo ejemplo, un disco se divide en bloques de tamaño idéntico, y a cada bloque se le asigna parte de un archivo; nunca se emplea un bloque para almacenar partes de dos o más archivos, aunque un archivo termine en mitad de un bloque.

En contraste, el tercero y cuarto ejemplos cubren la asignación de memoria para un sistema de multiprogramación y administración de una estructura dinámica para los lenguajes que tratan con variables cuyos valores son objetos «grandes», y aquí se habla de asignación de espacio en bloques de tamaño diferente. Este requisito presenta ciertos problemas que no se presentan en el caso de longitudes fijas. Por ejemplo, se teme la *fragmentación*, una situación en la cual hay mucho espacio sin utilizar, pero está distribuido en fragmentos tan pequeños que no puede encontrarse espacio para un objeto grande. Se profundizará más acerca de la administración de estructuras dinámicas en las secciones 12.4 y 12.5.

El segundo aspecto importante es si la *recolección de basura*, un término muy descriptivo para la recuperación de espacio no utilizado, se efectúa explícita o implícitamente, esto es, mediante un mandato del programa o sólo como respuesta a una petición de espacio que no puede satisfacerse de otra manera. En el caso de la administración de archivos, cuando se elimina un archivo, los bloques que se utilizaron para contenerlo son conocidos por el sistema de archivos. Por ejemplo, el sistema de archivos puede grabar la dirección de uno o más «bloques maestros» para cada archivo existente; los bloques maestros listan las direcciones de todos los bloques utilizados por el archivo. Así, cuando se elimina un archivo, el sistema de archivos puede hacerlo explícitamente disponible para reutilizar todos los bloques eliminados por ese archivo.

En contraste, las celdas de LISP continúan ocupando su espacio en memoria cuando se apartan de las estructuras de datos del programa. Debido a la posibilidad de que haya varios apuntadores a una celda, no se puede decir cuándo una celda está separada por completo y, por tanto, tampoco recoger en forma explícita las celdas, como se hizo con los bloques de un archivo eliminado. Tarde o temprano, todos los espacios en memoria corresponderán a celdas útiles o inútiles, y la siguiente solicitud de espacio para otra celda activará implícitamente una «recolección de basura»; en ese momento, el intérprete de LISP marca todas las celdas útiles, por medio de un algoritmo similar al que se presentará en la sección 12.3, y después enlazará todos los bloques que contengan celdas inútiles en una lista de espacio disponible, para poderlas reutilizar.

La figura 12.2 ilustra las cuatro clases de administración de memoria y da un ejemplo de cada una. De la figura 12.2 ya se han comentado los ejemplos de bloques de tamaño fijo. La administración de la memoria principal en un sistema de programación múltiple es un ejemplo de petición explícita de bloques con longitud variable. Esto es, cuando un programa termina, el sistema operativo, sabiendo qué área de memoria se otorgó al programa y sabiendo que ningún otro programa puede usar ese espacio, hace que el espacio quede inmediatamente disponible para cualquier otro programa.

La administración de una estructura dinámica en SNOBOL o en muchos otros lenguajes es un ejemplo de bloques con longitud variable y recolección de basura.

		recuperación del espacio no utilizado	
		explícita	recolección de basura
tamaño del bloque	fijo	sistema de archivos	LISP
	variable	sistema de multiprogramación	SNOBOL

Fig. 12.2. Ejemplos de las cuatro estrategias de administración de memoria.

Como en LISP, un intérprete típico de SNOBOL no intenta recuperar bloques de memoria hasta que se agota el espacio. En ese momento, el intérprete realiza una recolección de basura igual que lo hace un intérprete de LISP, pero con la posibilidad adicional de que las cadenas se muevan en torno a la estructura dinámica para reducir la fragmentación, y que los bloques libres adyacentes se combinen para formar bloques más grandes. Obsérvese que los dos últimos pasos no tienen sentido en un ambiente LISP.

12.2 Administración de bloques de igual tamaño

Supóngase que se tiene un programa que maneja celdas con un par de campos cada una; cada campo puede ser un apuntador a una celda o puede contener un «átomo». Por supuesto, la situación es igual a la de un programa escrito en LISP, pero el programa puede escribirse casi en cualquier lenguaje, incluso en Pascal, si se definen las celdas del tipo registro variante. Las celdas vacías que se encuentran disponibles para su incorporación a una estructura de datos se colocan en una lista de espacio disponible, y cada variable del programa se representa por un apuntador a una celda; la celda apuntada puede pertenecer a una gran estructura de datos.

Ejemplo 12.2. En la figura 12.3, se observa una estructura de datos posible. *A*, *B* y *C* son variables, y las letras minúsculas representan átomos. Obsérvense algunos fenómenos interesantes: la celda que contiene el átomo *a* está apuntada por la variable *A* y por otra celda; la celda que contiene el átomo *c* está apuntada por dos celdas distintas; las celdas que contienen *q* y *h* son un caso especial, porque aunque se apuntan mutuamente, no son accesibles desde las variables *A*, *B* o *C*, ni están en la lista de espacio disponible. □

Supóngase que cuando se ejecuta el programa, se pueden quitar celdas nuevas de la lista de espacio disponible; por ejemplo, puede ser conveniente sustituir el apuntador nulo de la celda con el átomo *c* de la figura 12.3 por un apuntador a una celda nueva que contenga el átomo *i* y un apuntador nulo. Esta celda se eliminará de la

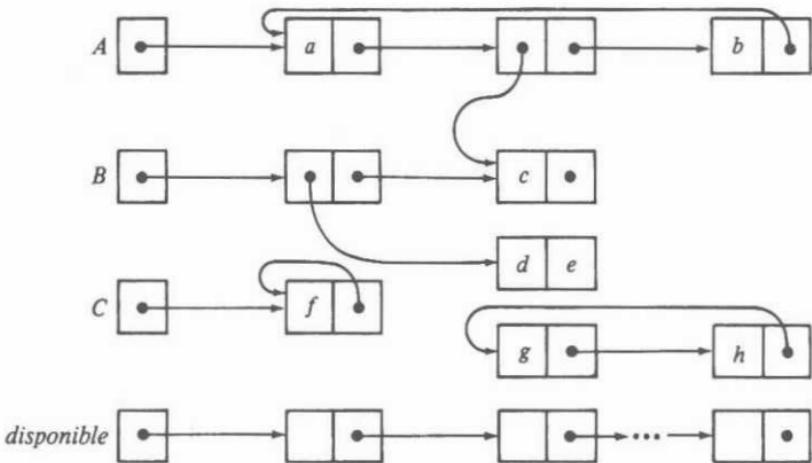


Fig. 12.3. Red de celdas.

cabeza de la lista de espacio disponible. También es posible que de vez en cuando los apuntadores cambien, de forma que las celdas se separen de las variables del programa, como sucedió con las celdas *g* y *h* de la figura 12.3. Por ejemplo, la celda en que se encuentra el átomo *c*, pudo haber apuntado alguna vez a la celda que contiene a *g*. Como otro ejemplo, el valor de la variable *B* puede cambiar en un momento dado, lo cual si nada más ha cambiado, aparta la celda apuntada por *B* en la figura 12.3, al igual que la celda que contiene a *d* y *e* (pero no la celda que contiene a *c*, pues ésta es accesible desde *A*). Las celdas que no pueden alcanzarse desde ninguna variable y que no están en la lista de espacio disponible son conocidas como celdas *inaccesibles*.

Cuando se separan las celdas, y no las vuelve a necesitar el programa, sería deseable que volvieran a la lista de espacio disponible, para poderlas usar de nuevo. Si no se reclaman dichas celdas, con el tiempo se llegará a la situación inaceptable de que el programa no esté empleando todas las celdas, aunque requiera una celda nueva, y la lista de espacio disponible esté vacía. Es entonces cuando debe realizarse una recolección de basura, lo que consume tiempo. Este paso de recolección de basura es «implícito», en el sentido de que no lo llamó explícitamente la petición de espacio.

Cuentas de referencia

Un enfoque atractivo para la detección de celdas inaccesibles consiste en incluir en cada celda una *cuenta de referencia*, esto es, un campo de tipo entero cuyo valor sea igual al número de apuntadores a la celda. Es fácil mantener las cuentas de referencia; cuando se hace que un apuntador apunte a una celda, se agrega un uno a la cuenta de referencia de esa celda, y cuando se reasigna un apuntador no nulo, primero

disminuye en uno la cuenta de referencia de la celda apuntada. Si una cuenta de referencia llega a valer cero, la celda será inaccesible y se podrá devolver a la lista de disponibles.

Lamentablemente, las cuentas de referencia no siempre funcionan. Las celdas con g y h de la figura 12.3 son inaccesibles y enlazadas en un ciclo. Sus cuentas de referencia valen 1, de manera que no se pueden devolver a la lista de disponibles. Se puede intentar detectar ciclos de celdas inaccesibles de distintas formas, pero tal vez no valga la pena hacerlo. Las cuentas de referencia son útiles para estructuras que no tienen ciclos de apuntadores. Un ejemplo de estructura sin posibilidades de ciclos es una colección de variables apuntando a bloques que contienen datos, como en la figura 12.1. De esta forma, se puede hacer la recolección de basura de forma explícita, recogiendo un bloque cuando su cuenta de referencia ha alcanzado el valor cero. Sin embargo, cuando las estructuras de datos permiten ciclos de apuntadores, la estrategia basada en cuentas de referencia por lo general es inferior a otro enfoque que se analizará en la siguiente sección, tanto en función del espacio requerido por las celdas como del tiempo relacionado con el caso de las celdas inaccesibles.

12.3 Algoritmos de recolección de basura para bloques de igual tamaño

Ahora se presenta un algoritmo para detectar qué celdas de una colección de los tipos sugeridos en la figura 12.3 son accesibles desde las variables del programa. Se definirá con precisión el problema definiendo un tipo de celda en Pascal que es una variante de tipo registro; las cuatro variantes, que se llamarán PP , PA , AP , AA , se determinan según cuáles de los dos campos de datos son apuntadores y cuáles son átomos. Por ejemplo, PA significa que el campo izquierdo es un apuntador y el campo derecho es un átomo. Un campo booleano adicional en las celdas, llamado *marca*, indica si la celda es accesible. Es decir, poniendo *marca* en verdadero al hacer la recolección de basura, se «marca» la celda, indicando que es accesible. En la figura 12.4 se muestran las definiciones de los tipos.

```

type
  tipo_átomo = { algún tipo apropiado, de preferencia,
    del mismo tamaño que los apuntadores }
  patrones = (PP, PA, AP, AA);
  tipo_celda = record
    marca: boolean;
    case patrón: patrones of
      PP: (izquierda: ↑ tipo_celda; derecha: ↑ tipo_celda);
      PA: (izquierda: ↑ tipo_celda; derecha: tipo_átomo);
      AP: (izquierda: tipo_átomo; derecha: ↑ tipo_celda);
      AA: (izquierda: tipo_átomo; derecha: tipo_átomo);
  end;

```

Fig. 12.4. Definición del tipo de las celdas.

Se supone que hay un arreglo de celdas, que ocupa casi toda la memoria, y una colección de variables, que son apuntadores a las celdas. Por simplicidad, se da por hecho que hay sólo una variable, llamada *fuente*, apuntando a una celda, pero la extensión a muchas variables no es difícil †. Esto es, se declara

```
var
    fuente: ↑ tipo_celda;
    memoria: array [1..tamaño_memoria] of tipo_celda;
```

Para marcar las celdas accesibles desde *fuente*, primero se «desmarcan» todas las celdas, sean o no accesibles, recorriendo todo el arreglo *memoria* y poniendo el campo *marca* en falso. Después, se realiza una búsqueda primera de profundidad en el grafo que emana de *fuente*, marcando todas las celdas visitadas. Las celdas visitadas son exactamente aquellas que son accesibles. Después, se recorre el arreglo *memoria* para agregar a la lista de espacio disponible todas las celdas no marcadas. La figura 12.5 muestra un procedimiento *bpf* para realizar la búsqueda en profundidad; *bpf* se llama con el procedimiento *recoge* que desmarca todas las celdas, y marca las celdas accesibles llamando a *bpf*. No se presenta el código para enlazar la lista de espacio disponible debido a las peculiaridades de Pascal. Por ejemplo, aunque se pueden enlazar las celdas disponibles con todas las celdas izquierdas o todas las derechas, ya que se ha supuesto que los apuntadores y los átomos tienen el mismo tamaño, no está permitido reemplazar átomos por apuntadores en las celdas que sean del tipo variante *AA*.

```
(1) procedure bpf( celda_actual: ↑ tipo_celda );
    { Si se marcó la celda actual no se hace nada; en caso contrario,
      debe marcarse y llamar a bpf en cualquier celda apuntada por
      la celda actual }

begin
(2)   with celda_actual ↑ do
(3)     if marca = false then begin
(4)       marca := true;
(5)       if (patrón = PP) or (patrón = PA) then
(6)         if izquierda <> nil then
(7)           bpf(izquierda);
(8)         if (patrón = PP) or (patrón = AP) then
(9)           if derecha <> nil then
(10)             bpf(derecha);

end
end; { bpf }

(11) procedure recoge;
var
    i: = integer;
```

† Cada lenguaje de programación debe proporcionar un método propio de representación del conjunto de variables actual, y cualquiera de los métodos estudiados en los capítulos 4 y 5 es adecuado. Por ejemplo, la mayor parte de las aplicaciones usan una tabla de dispersión para guardar las variables.

```

begin
(12)    for i := 1 to tamaño_memoria do { "desmarcar" todas las celdas }
(13)        memoria[i].marca := false;
(14)        bpf(fuente); { marcar las celdas accesibles }
(15)        { aquí va el código para la recogida }
end; { recoge }

```

Fig. 12.5. Algoritmo para marcar celdas accesibles.

Recolección en el mismo sitio

El algoritmo de la figura 12.5 tiene un defecto sutil; en un ambiente de programación donde la memoria es limitada, puede suceder que no haya espacio disponible para almacenar la pila requerida para las llamadas recursivas a *bpf*. Como se ilustró en la sección 2.6, cada vez que *bpf* se llama a sí mismo, Pascal (o cualquier otro lenguaje que permita la recursión) crea un «registro de activación» para esa llamada particular a *bpf*. En general, un registro de activación contiene espacio para parámetros y variables locales al procedimiento, del cual cada llamada necesita su propia copia. Algo que también es necesario en cada registro de activación es una «dirección de retorno», el lugar al cual debe regresar el control cuando termina esta llamada recursiva al procedimiento.

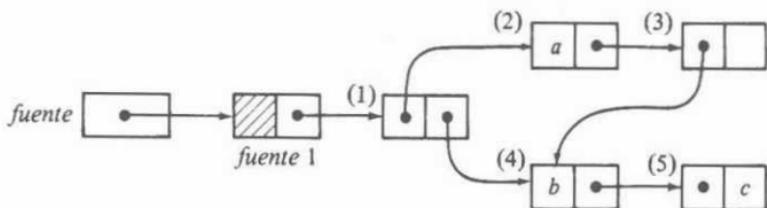
En el caso particular de *bpf*, sólo se necesita espacio para el parámetro y la dirección de retorno [esto es, fue llamado desde la línea (14) de *recoge*, desde la línea (7) o la (10) de otra invocación de *bpf*]. No obstante, esta solicitud de espacio es suficiente para que, si toda la memoria se enlaza en una sola cadena extendida desde *fuente* (por lo que el número de llamadas activas a *bpf* puede ser en algún momento igual a la longitud de esta cadena), se requerirá bastante más espacio para la pila de registros de activación que el asignado para la memoria. Si ese espacio quizás no estuviera disponible, sería imposible realizar la marcación.

Por fortuna, hay un ingenioso algoritmo, conocido como *algoritmo de Deutsch-Schorr-Waite*, para hacer la marcación en el mismo sitio. Es necesario convencerse de que la secuencia de celdas sobre la cual se ha realizado una llamada a *bpf*, sin haber terminado, en realidad forma un camino desde *fuente* hasta la celda en la cual se hizo la llamada actual a *bpf*. Así, se puede usar una versión no recursiva de *bpf*, y en vez de una pila de registros de activación para registrar el camino de celdas desde *fuente* hasta la celda que se está examinando en ese momento, se pueden usar los campos apuntadores del camino para contener el camino mismo. Esto es, cada celda del camino, excepto la última, contiene en el campo *derecha* o *izquierda* un apuntador a su *predecesor*, la celda más cercana a *fuente*. Se describirá el algoritmo de Deutsch-Schorr-Waite con un campo extra de un solo bit, llamado *atrás*, que es de un tipo enumerado (*I,D*), e informa si el campo izquierdo o el derecho apunta al predecesor. Más tarde, se evaluará la forma de almacenar la información contenida en *atrás* en el campo *patrón*, sin necesidad de espacio adicional en las celdas.

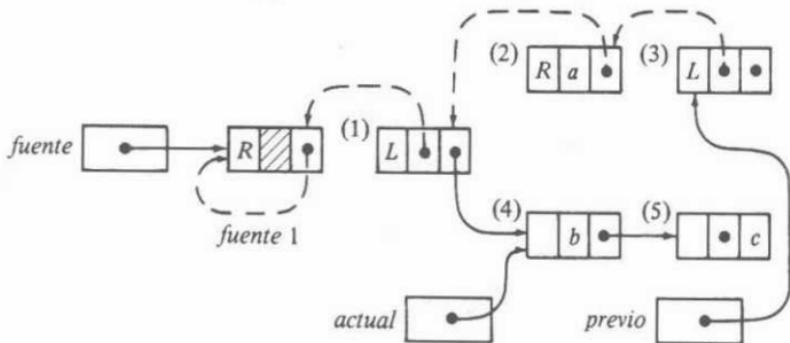
El nuevo procedimiento para búsqueda en profundidad no recursiva, que se llama *bpfnr*, se vale de un apuntador, *actual*, a la celda actual, y un apuntador, *previo*, al predecesor de la celda actual. La variable *fuente* apunta a una celda *fuentel* que

tiene un apuntador sólo en su campo derecho \dagger . Antes de marcar se asigna valor inicial a *fuentel* para tener $atrás = D$, y su campo derecho apuntando a sí mismo. A la celda que suele apuntar *fuentel*, apunta ahora *actual*, y a *fuentel* ahora *previo*. Se detiene la operación de marcado ocurre cuando $actual = previo$, lo cual sólo ocurre cuando ambos apuntan a *fuentel*, y se ha revisado por completo la estructura.

Ejemplo 12.3. La figura 12.6(b) muestra una posible estructura que emana de *fuente*. Si hace una búsqueda primera de profundidad en la estructura, se visitan (1), (2), (3) y (4), en ese orden. La figura 12.6(b) muestra las modificaciones hechas a los apuntadores cuando la celda actual es (4). Se muestra el valor del campo *atrás*, aunque no sucede lo mismo con los campos *marca* y *patrón*. El camino actual va de (4)



(a) Estructura de una celda



(b) Situación en la que la celda (4) es la actual

Fig. 12.6. Uso de apuntadores para representar el camino de regreso a *fuente*.

a (3) a (2) y a (1), de vuelta a *fuentel*; está representado por líneas (apuntadores) de puntos. Por ejemplo, la celda (1) tiene $atrás = I$, ya que el campo *izquierda* de (1), que en la figura 12.6 contiene un apuntador a (2), está apuntando hacia atrás, en vez de hacia adelante a lo largo del camino; sin embargo, se restaurará ese apuntador cuando la búsqueda en profundidad por fin regrese de la celda (2) a la (1). Análogamente, en la celda (2), $atrás = D$, y el campo derecho de (2) apunta hacia atrás en el camino hacia (1), en vez de dirigirse a (3), como lo hacía en la figura 12.6(a). □

\dagger Esta incomodidad es necesaria debido a las peculiaridades de Pascal.

Son tres los pasos básicos para realizar la búsqueda en profundidad:

1. *Avanzar.* Si se determina que la celda actual tiene uno o más apuntadores no nulos, se avanza al primero de ellos, esto es, se sigue el apuntador en *izquierda*, pero si no lo hay, se sigue el apuntador en *derecha*. Por «avanzar» se entiende hacer que la celda apuntada se convierta en la celda actual y, la celda actual, en la anterior. Para ayudar a encontrar el camino de regreso, se hace que el apuntador recién seguido apunte a la celda anterior. Esos cambios se muestran en la figura 12.7(a), en el supuesto de que se siga al apuntador izquierdo. En esa figura, los apuntadores anteriores se representan con líneas de trazo continuo, y los nuevos, con líneas de puntos.
2. *Comutador.* Si se determina que las celdas siguientes a la actual ya se han revisado (por ejemplo, la celda actual puede tener sólo átomos, puede estar marcada, o se pudo haber «replegado» a la celda actual desde la apuntada por el campo *derecha* de la actual), se consulta el campo *atrás* de la celda anterior. Si ese valor es *I*, y el campo *derecha* de la celda previa contiene un apuntador no nulo a alguna celda *C*, se hace que *C* se convierta en la celda actual, y la identidad de la celda anterior no se cambia. Sin embargo, el valor de *atrás* en la celda anterior se hace *D*, y el apuntador izquierdo en esa celda recibe su valor correcto; esto es, se hace que apunte a la celda actual anterior. Para mantener el camino de regreso a *fuente* desde la celda anterior, se hace que el apuntador a *C* en el campo *derecha* apunte hacia donde apuntaba izquierdo. La figura 12.7(b) muestra esos cambios.
3. *Replegar.* Si se determina, como en (2), que las celdas que parten de la actual se han recorrido, pero el campo *atrás* de la celda anterior es *D*, o es *I*, pero el campo derecho contiene un átomo o un apuntador nulo, entonces se han revisado todas las celdas que parten de la anterior. Se efectúa el repliegue haciendo que la celda anterior sea la actual y que la celda siguiente en el camino de la celda anterior a *fuente* sea la nueva celda anterior. Esos cambios se muestran en la figura 12.7(c), en el supuesto de que *atrás = D* en la celda anterior.

Una coincidencia fortuita es que cada paso de la figura 12.7 se puede considerar como la rotación simultánea de tres apuntadores. Por ejemplo, en la figura 12.7(a), se reemplazan simultáneamente (*previo*, *actual*, *actual izquierda*) por (*actual*, *actual izquierda*, *previo*), respectivamente. Debe subrayarse la simultaneidad: la ubicación de *actual izquierda* no cambia al asignar un valor nuevo a *actual*. Para realizar esas modificaciones a los apuntadores es útil contar con un procedimiento *rotar*, que se muestra en la figura 12.8. Obsérvese en especial que el uso de parámetros por referencia asegura que las ubicaciones de los apuntadores se establezcan antes de cambiar ningún valor.

Ahora se considera el diseño del procedimiento no recursivo *bpfnr* para hacer el marcado. Este procedimiento es uno de esos raros procesos que son más fáciles de entender cuando se escriben con etiquetas y proposiciones *goto*. En especial, existen dos «estados» del procedimiento, «avance» representado por la etiqueta 1, y «repliegue», representado por la etiqueta 2. Se introduce inicialmente el primer estado, y también siempre que se pasa a una celda nueva, por un paso de avance o uno de

conmutación. En este estado, se intenta otro paso de avance, y sólo se efectúa un repliegue o una conmutación si existe un bloqueo, que se puede deber a dos razones: 1) la celda recién alcanzada ya está marcada, o 2) no hay apuntadores no nulos en la celda. Cuando hay un bloqueo, se cambia al segundo estado o «replegado».

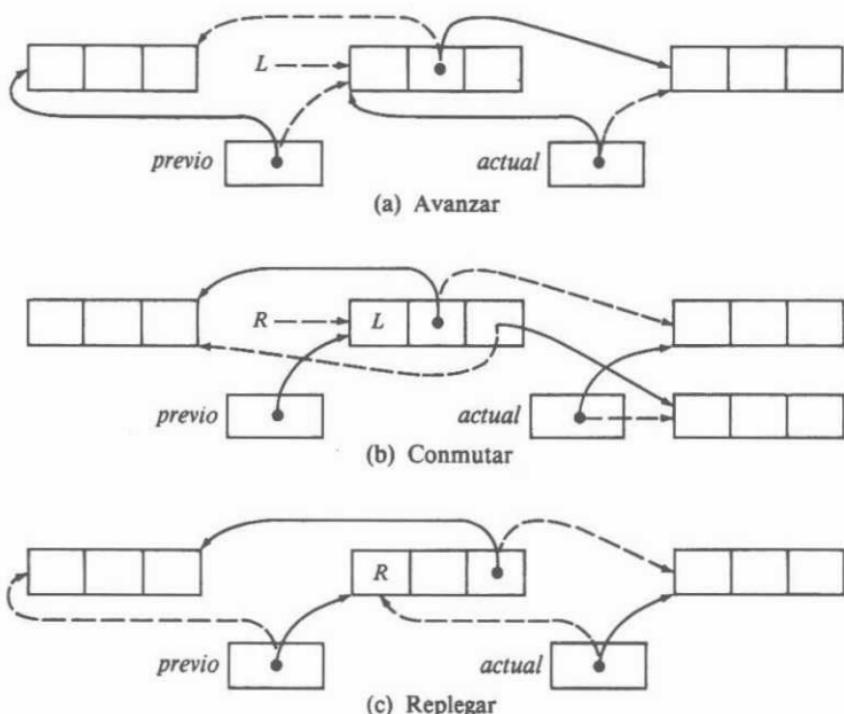


Fig. 12.7. Tres pasos básicos.

```
procedure rota ( var p1, p2, p3: ↑ tipo_celda );
  var
    temp: ↑ tipo_celda;
  begin
    temp := p1;
    p1 := p2;
    p2 := p3;
    p3 := temp
  end; { rota }
```

Fig. 12.8. Procedimiento de modificación de apuntadores.

El segundo estado se alcanzará cuando ocurra un repliegue o cuando no sea posible permanecer en estado de avance porque hay un bloqueo. En el estado de repliegue, se prueba si ya se ha replegado hasta la celda ficticia *fuentel*. Como se analizó antes, se reconocerá esta situación porque *previo = actual*, en cuyo caso se pasa al estado 3. De otra forma, se decide un repliegue para permanecer en ese estado o se pasa al estado de avance. El código de *bpfnr* se muestra en la figura 12.9; este código utiliza las funciones *bloque_izq*, *bloque_der* y *bloque*, que prueban si los campos izquierdo o derecho de una celda, o ambos, tienen un átomo o un apuntador nulo, *bloque* también prueba si hay una celda marcada.

```

function bloque_izquierdo ( celda: tipo_celda): boolean;
{ prueba si el campo izquierda es un átomo o un apuntador nulo }
begin
  with celda do
    if (patrón = PP) or (patrón = PA) then
      if izquierda <> nil then return (false);
    return (true)
  end; { bloque_izquierdo }

function bloque_derecho ( celda : tipo_celda ): boolean;
{ prueba si el campo derecha es un átomo o un apuntador nulo }
begin
  with celda do
    if (patrón = PP) or (patrón = AP) then
      if derecha <> nil then
        return (false);
    return (true)
  end; { bloque_derecho }

function bloque ( celda : tipo_celda ): boolean;
{ prueba si la celda está marcada o no contiene un apuntador no nulo s }
begin
  if (celda.marca = true) or bloque_izquierdo(celda) and bloque_derecho
    (celda) then
    return (true)
  else
    return (false)
end; { bloque }

procedure bpfnr; { marca las celdas accesibles desde fuente }
var
  actual, previo: †tipo_celda;
begin { asignación de valor inicial }
  actual := fuente1.derecha; { celda apuntada por fuente 1 }
  previo := fuente1; { previo apunta a fuente1 }
  fuente1.atrás := D;
  fuente1.derecha := fuente1; { fuente1 se apunta a sí mismo }
  fuente1.marca := true;

```

```

estado 1: { intenta avanzar }
    if bloque(actual↑) then begin { prepara el repliegue }
        actual↑.marca := true;
        goto estado2
    end
    else begin { marca y avanza }
        actual↑.marca := true;
        if bloque_izquierdo(actual↑) then begin { sigue al apuntador derecho }
            actual↑.atrás := D;
            rota(previo, actual, actual↑.derecha); {realiza los cambios
                de la figura 12.7(a), pero siguiendo al apuntador derecho }
            goto estado1
        end
        else begin { sigue al apuntador izquierdo }
            actual↑.atrás := I;
            rota(previo, actual, actual↑.izquierda);
            { realiza los cambios de la figura 12.7(a) }
            goto estado1
        end
    end
end;

estado2: { termina, repliega o conmuta }
    if previo = actual then { termina }
        goto estado3
    else if (previo↑.atrás = I) and
        not bloque_derecho(previo↑) then begin { conmuta }
        previo↑.atrás := D;
        rota(previo↑.izquierda, actual, previo↑.derecha);
        { realiza los cambios de la figura 12.7(b) }
        goto estado1
    end
    else if previo↑.atrás = D then { repliega }
        rota (previo, previo↑.derecha, actual)
        { realiza los cambios de la figura 12.7(c) }
    else { previo↑.atrás = I }
        rota(previo, previo↑.izquierda, actual);
        { realiza los cambios de la figura 12.7(c), pero con el campo
            izquierda de la celda previa comprendido en el camino }
    goto estado2
end;

estado3: { poner aquí el código para enlazar celdas no marcadas en la
            lista de espacio disponible }
end; { bpfnr }

```

Fig. 12.9. Algoritmo no recursivo para marcar celdas.

Algoritmo Deutsch-Schorr-Waite sin un bit extra para el campo *atrás*

Es posible, aunque poco probable, que el bit extra utilizado en las celdas por el campo *atrás* haga que las celdas necesiten un byte extra, o incluso de una palabra adicional. En tal caso, es bueno saber que en realidad no se necesita el bit extra, al menos si se programa en un lenguaje que, a diferencia de Pascal, permita utilizar los bits del campo *patrón* para propósitos distintos de los declarados: designadores del formato de registro variante. El «truco» consiste en observar que si se usa el campo *atrás*, como su celda está en el camino de vuelta a *fuente1*, los valores posibles del campo *patrón* están restringidos. Por ejemplo, si *atrás* = *I*, entonces se sabe que el *patrón* debe ser *PP* o *PA*, pues es obvio que el campo *izquierda* tiene un apuntador. Los mismo sucede cuando *atrás* = *D*. Así, si se dispone dos bits para representar *patrón* y (cuando sea necesario) *atrás*, se puede codificar la información necesaria como en la figura 12.10, por ejemplo.

Debe observarse que en el programa de la figura 12.9, siempre se sabe si se está usando *atrás*, para saber qué interpretación de la figura 12.10 es aplicable. Tan sólo cuando *actual* apunta a un registro, el campo *atrás* en ese registro no se usa; cuando *previo* lo apunta, entonces sí. Por supuesto, cuando se mueven esos apuntadores, es necesario ajustar los códigos; por ejemplo, si *actual* apunta a una celda con los bits 10, que se interpretan de acuerdo con la figura 12.10 como *patrón* = *AP*, y se decide avanzar, de modo que *previo* apuntará ahora a esta celda, se fija *atrás* = *D*, pues sólo el campo derecho contiene un apuntador, y los bits apropiados son 11. Obsérvese que si el *patrón* fuera *AA*, que no tiene representación en la columna central de la figura 12.10, no se querrá que *previo* apunte a la celda, pues no hay apuntadores que seguir en un movimiento de avance.

Código	En el camino hacia <i>fuente1</i>	Fuera del camino
00	<i>atrás</i> = <i>I</i> , <i>patrón</i> = <i>PP</i>	<i>patrón</i> = <i>PP</i>
01	<i>atrás</i> = <i>I</i> , <i>patrón</i> = <i>PA</i>	<i>patrón</i> = <i>PA</i>
10	<i>atrás</i> = <i>D</i> , <i>patrón</i> = <i>PP</i>	<i>patrón</i> = <i>AP</i>
11	<i>atrás</i> = <i>D</i> , <i>patrón</i> = <i>AP</i>	<i>patrón</i> = <i>AA</i>

Fig. 12.10. Interpretación de dos bits como *patrón* y *atrás*.

12.4 Asignación de almacenamiento para objetos de diferentes tamaños

Considérese ahora el manejo de una estructura dinámica, como lo presenta la figura 12.1, donde hay una colección de apuntadores a bloques asignados. Los bloques contienen datos de algún tipo. En la figura 12.1, por ejemplo, los datos son cadenas de caracteres. Aunque el tipo de los datos almacenados en la estructura dinámica no tiene por qué ser cadenas de caracteres, se supone que los datos no tienen apuntadores a localidades de la estructura dinámica.

El problema de la administración de estructuras dinámicas tiene aspectos que facilitan y también dificultan su mantenimiento en comparación con las estructuras

de listas de celdas de igual tamaño, tratadas en la sección anterior. El factor principal que facilita el problema es que el marcado de bloques usados no es un proceso recursivo; sólo se tienen que seguir los apuntadores externos de la estructura dinámica y marcar los bloques apuntados. No es necesaria una búsqueda en profundidad de una estructura enlazada ni de un algoritmo como el Deutsch-Schorr-Waite.

Por otro lado, la administración de la lista de espacio disponible no es tan simple como en la sección 12.3. Podría imaginarse que las regiones vacías [por ejemplo, en la Fig. 12.1(a) hay tres regiones vacías] están enlazadas como se sugiere en la figura 12.11. Ahí se observa una estructura dinámica de 3000 palabras dividida en cinco bloques. Dos bloques de 200 y 600 palabras, respectivamente, contienen los valores de X e Y . Los tres bloques restantes están vacíos, y están enlazados en una cadena que parte de *dispo*, el encabezado para el espacio disponible.

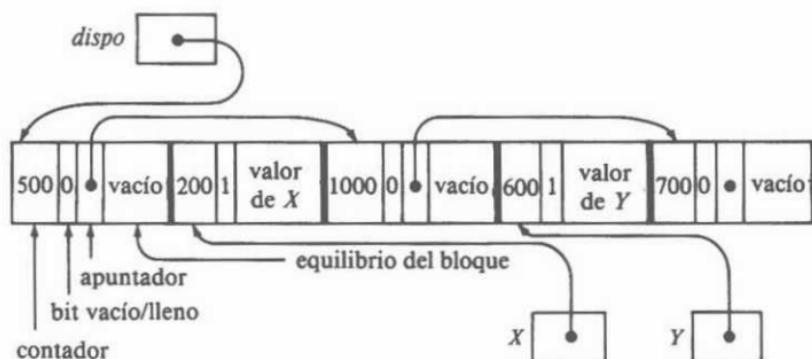


Fig. 12.11. Una estructura dinámica con lista de espacio disponible.

Para que los bloques vacíos se puedan encontrar siempre que se requiera almacenar datos nuevos, y pueda disponerse de los que contienen datos inútiles, en esta sección se hacen las siguientes suposiciones.

1. Cada bloque es bastante grande para contener
 - a) un *contador* que indique el tamaño (en bytes o palabras, de acuerdo con el computador) del bloque,
 - b) un apuntador (para enlazar el bloque al espacio disponible), más
 - c) un bit para indicar si el bloque está vacío o no; este bit se conoce como *bit vacío / lleno* o *usado / no usado*.
2. Un bloque vacío tiene, desde la izquierda (dirección más baja), un contador que indica su longitud, un bit vacío/lleno con un valor de cero, que indica que el bloque está vacío, un apuntador al siguiente bloque disponible, y el espacio libre.
3. Un bloque que contiene datos tiene, desde la izquierda, un contador, un bit vacío / lleno con valor 1 indicando que el bloque está en uso, y los datos †.

† Obsérvese que en la figura 12.1, en vez de un contador que indica la longitud del bloque, se emplea la longitud de los datos.

Una consecuencia interesante de las suposiciones anteriores es que los bloques deben ser capaces de almacenar datos, algunas veces (cuando están en uso), y apun-tadores, en otras (cuando están en desuso), exactamente en el mismo lugar, con lo que es imposible o muy incómodo escribir programas que manipulen bloques de esta clase en Pascal o cualquier otro lenguaje fuertemente orientado a tipos. Así, esta sección debe ser discursiva por necesidad; sólo pueden escribirse programas en seu-do-Pascal, nunca programas reales en Pascal. Sin embargo, no hay ningún problema para escribir programas que hagan las cosas descritas en lenguaje ensamblador o en la mayoría de los lenguajes de programación de sistemas, como C.

Fragmentación y compactación de bloques vacíos

Para ver uno de los problemas especiales que se presentan en el manejo de estruc-turas dinámicas, supóngase que la variable Y de la figura 12.11 cambia, de forma que el bloque que representa a Y debe devolverse al espacio disponible. Es más fácil insertar el bloque nuevo al principio de la lista de disponibles, como se sugiere en la figura 12.12. En esa figura se observa un caso de *fragmentación*, la tendencia a representar grandes áreas vacías en la lista de espacio disponible por medio de «frag-mentos», esto es, varios bloques pequeños formando el total. En el caso en cuestión, los últimos 2300 bytes de la estructura dinámica de la figura 12.12 están vacíos, pero el espacio está dividido en tres bloques de 1000, 600 y 700 bytes, y esos bloques no están siquiera, en orden consecutivo en la lista de disponibles. Sin alguna forma de recolección de basura, sería imposible satisfacer una petición de, por ejemplo, un blo-que de 2000 bytes.

Es obvio que cuando se devuelve un bloque a la lista de disponibles es conve-niente observar los bloques inmediatamente a la izquierda y a la derecha del bloque que se está haciendo disponible. Encontrar el bloque de la derecha es sencillo; si el blo-que que se está recuperando empieza en la posición p y el contador vale c , el blo-que de la derecha empieza en la posición $p + c$. Si se conoce p (por ejemplo, el apun-tador Y de la Fig. 12.11 contiene el valor p del bloque hecho disponible en la Fig. 12.12), basta con leer los bytes que empiezan en la posición p , tantos como se re-quieran para contener c , y obtener el valor c . Del byte $p + c$, se salta el campo del

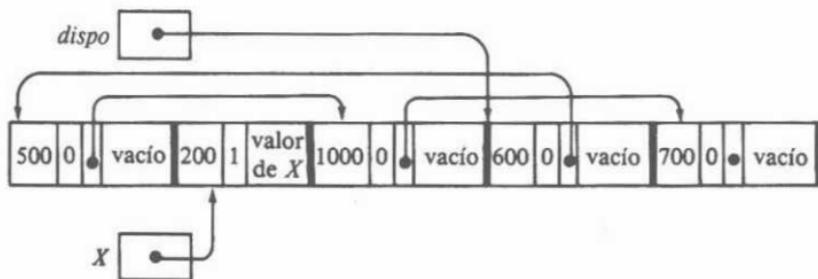


Fig. 12.12. Después de devolver el bloque de Y .

contador para encontrar el bit que indica si el bloque está vacío o no. Si está vacío, los bloques que empiezan en p y en $p + c$ pueden combinarse.

Ejemplo 12.4. Supóngase que la estructura dinámica de la figura 12.11 empieza en la posición 0. Entonces, el bloque de Y que se está devolviendo empieza en el byte 1700, así que $p = 1700$ y $c = 600$. El bloque que empieza en $p + c = 2300$ también está vacío, por lo que se pueden combinar en un solo bloque que empieza en 1700 con un contador de 1300, la suma de los contadores de los dos bloques. \square

Sin embargo, no es tan fácil fijar la lista de disponibles después de combinar los bloques. Se puede crear el bloque combinado agregando tan sólo el contador del segundo bloque a c . Sin embargo, el segundo bloque permanecerá enlazado en la lista de disponibles y deberá separarse, para lo cual es necesario encontrar el apuntador a ese bloque desde su predecesor en la lista de disponibles. Se presentan varias estrategias, pero ninguna se recomienda en especial.

1. Recorrer la lista hasta encontrar un apuntador con valor $p + c$. Este apuntador debe estar en el bloque anterior al que se ha combinado con su vecino. Reemplazar el apuntador encontrado por el apuntador del bloque de $p + c$. Esto, en efecto, elimina el bloque que empieza en la posición $p + c$ de la lista de disponibles. Su espacio queda disponible, por supuesto; es parte del bloque que empieza en p . En promedio, habrá que revisar la mitad de la lista de disponibles, así que el tiempo requerido es proporcional a esa longitud.
2. Usar una lista doblemente enlazada para el espacio disponible. Entonces, el bloque predecesor puede encontrarse con rapidez y el bloque en $p + c$ se puede eliminar de la lista. Este enfoque requiere un tiempo constante, independiente de la longitud de la lista de disponibles, pero requiere espacio adicional para otro apuntador en cada bloque vacío, incrementando así el tamaño mínimo de un bloque para contener un contador, un bit vacío/lleto, y dos apuntadores.
3. Conservar clasificada toda la lista de espacio disponible de acuerdo con la posición. Entonces, se sabe que el bloque de la posición p es el predecesor en la lista del bloque en $p + c$, y la manipulación del apuntador necesaria para eliminar el segundo bloque puede efectuarse en un tiempo constante. Sin embargo, la inserción de un bloque disponible nuevo requiere revisar en promedio la mitad de la lista, por lo que no es más eficiente que el método (1).

De los tres métodos, el primero y el tercero requieren un tiempo proporcional a la longitud de la lista de disponibles para devolver un bloque al espacio disponible y combinarlo con su vecino derecho, si está vacío. Este tiempo puede ser prohibitivo o no, dependiendo de lo larga que sea la lista y del tiempo total del programa se consume en la manipulación de la estructura dinámica. El segundo método —doble enlace para la lista de disponibles— tiene sólo el defecto de incrementar el tamaño mínimo de los bloques. Lamentablemente, cuando se considera cómo combinar un bloque devuelto con su vecino de la izquierda, como en los otros métodos, se observa que el doble enlace no ayuda a encontrar vecinos izquierdos en menos tiempo que el requerido para recorrer la lista de disponibles.

Encontrar un bloque inmediatamente a la izquierda del actual no es fácil. La po-

sición p de un bloque y de su contador c determinan la posición del bloque de la derecha, pero esto no sirve como guía para encontrar el principio del bloque izquierdo. Es necesario encontrar un bloque vacío que empiece en alguna posición p_1 y tenga un contador c_1 tal que $p_1 + c_1 = p$. Para esto hay tres posibles estrategias.

1. Revisar la lista de disponibles buscando un bloque en la posición p_1 y con contador c_1 donde $p_1 + c_1 = p$. Esta operación lleva un tiempo proporcional a la longitud de la lista de disponibles.
2. Conservar un apuntador en cada bloque (usado o no) indicando la posición del bloque de la izquierda. Este enfoque permite encontrar el bloque izquierdo en un tiempo constante; se prueba si está vacío, en cuyo caso se combina con el bloque en cuestión. Se puede encontrar el bloque en la posición $p + c$ y hacer que apunte al principio del bloque nuevo, para poder mantener esos apuntadores a la izquierda †.
3. Mantener clasificada la lista de disponibles de acuerdo con la posición. Entonces el bloque vacío de la izquierda se encuentra al insertar en la lista el bloque vaciado recientemente, y sólo es necesario revisar, usando la posición y el contador del bloque vacío anterior, que no se interponga ningún bloque no vacío.

Al igual que con la intercalación de bloques vacíos recientes con el bloque de la derecha, el primero y tercer enfoques para la búsqueda e intercalación con el bloque de la izquierda requieren un tiempo proporcional a la longitud de la lista de disponibles. El método (2) también requiere tiempo constante, pero tiene una desventaja sobre los problemas relativos a la lista de disponibles doblemente enlazada (ya sugerida en relación con la búsqueda de los bloques vecinos de la derecha). Mientras que el doble enlace de los bloques vacíos aumenta el tamaño mínimo de los bloques, no puede decirse que este enfoque desperdicie espacio, ya que sólo los bloques que no se usan para almacenar datos son los que se enlazan. Sin embargo, apuntar a los vecinos izquierdos requiere un apuntador en los bloques utilizados y en los no utilizados, y con justicia puede acusársele de consumir espacio. Si el tamaño promedio de los bloques es de cientos de bytes, el espacio extra para un apuntador puede ser despreciable. Por otro lado, el espacio adicional puede ser prohibitivo si el bloque típico tiene sólo 10 bytes de longitud.

Para resumir las implicaciones de estas exploraciones en cuanto a cómo combinar bloques vacíos recientes con vecinos vacíos, hay tres enfoques para tratar la fragmentación.

1. Usar uno de varios enfoques, como puede ser conservar clasificada la lista de disponibles, que requiere un tiempo proporcional a la longitud de la lista cada vez que un bloque queda sin utilizar, pero permite encontrar y combinar vecinos vacíos.
2. Usar una lista de espacio disponible doblemente enlazada cada vez que un bloque quede sin utilizar, y también utilizar apuntadores a los vecinos izquierdos de todos los bloques, estén o no disponibles, para combinar vecinos vacíos en un tiempo constante.

† Como ejercicio, se debe descubrir cómo mantener los apuntadores cuando un bloque se divide en dos; se toma una parte para un elemento de datos nuevo, mientras que la otra permanece vacía.

3. No hacer nada explícito para combinar vecinos vacíos. Cuando no sea posible encontrar un bloque tan grande como para contener datos nuevos, revisar los bloques de izquierda a derecha, combinando vecinos vacíos y después crear una lista nueva de disponibles. Un bosquejo del programa que hace esto se muestra en la figura 12.13.

```

(1)  procedure combina;
var
(2)    p, q: apuntadores a bloques;
        { p indica el extremo izquierdo del bloque vacío que se está
          acumulando; q indica un bloque a la derecha de p que se
          incorporará en el bloque p si está vacío }
begin
(3)    p:= bloque más a la izquierda de la estructura dinámica;
(4)    vaciar la lista de disponibles;
(5)    while p < extremo derecho de la estructura dinámica do
(6)      if p apunta a un bloque lleno con contador c then
(7)        p := p + c; { saltar los bloques llenos }
(8)      else begin { p apunta al principio de una secuencia de
        bloques vacíos; deben combinarse }
(9)        q := p + c; { asignar el valor inicial q al siguiente bloque }
(10)       while q apunta a un bloque vacío con un contador, d, y q <
            extremo derecho de la estructura dinámica do begin
            agregar d al contador del bloque apuntado por p;
            q := q + d
        end
(13)       inserta el bloque apuntado por p en la lista de disponibles;
(14)       p := q;
    end
end; { combina }

```

Fig. 12.13. Combinación de bloques vacíos adyacentes.

Ejemplo 12.5. Como ejemplo, considérese el programa de la figura 12.13, aplicado a la estructura dinámica de la figura 12.12. Supóngase que el byte de más a la izquierda de la estructura dinámica es cero, así que inicialmente, $p = 0$. Como $c = 500$ para el primer bloque, a q se le asigna un valor inicial $p + c = 500$. Cuando el bloque que empieza en 500 está lleno, el ciclo de las líneas (10) a (12) no se ejecuta y el bloque que consta de los bytes 0 a 499 se coloca en la lista de disponibles, haciendo que $dispo$ apunte al byte 0 y poniendo un apuntador nil en el lugar designado en ese bloque (después del contador y el bit vacío/lleno). Después, se asigna a p el valor 500 en la línea (14), y se incrementa a 700 en la línea (7). El apuntador q toma el valor 1700 en la línea (9), y después, 2300 y 3000 en la línea (12), al tiempo que 600 y 700 se agregan al contador 1000 en el bloque que empieza en 700. Cuando q excede del byte de más a la derecha, 2999, el bloque que empieza en 700, que ahora tiene

el contador 2300, se inserta en la lista de disponibles. Entonces, en la línea (14), p se ajusta en 3000 y el ciclo externo finaliza en la línea (5). \square

El número total de bloques y el número de bloques disponibles puede no ser muy diferente, y probablemente la frecuencia con que se puede encontrar que no hay bloques vacíos bastante grandes sea baja, se cree que el método (3), combinar bloques vacíos adyacentes sólo cuando se termine el espacio adecuado, es superior a (1) en una situación real. El método (2) es un posible competidor, pero considera los requisitos de espacio adicional y el hecho de que se requiere un tiempo extra cada vez que se inserta o elimina un bloque de la lista de disponibles, se cree que (2) será preferible a (3) en circunstancias muy raras, y tal vez se pueda olvidar.

Selección de bloques disponibles

Se ha tratado con detalle qué debe suceder cuando deja de necesitarse un bloque y se puede devolver al espacio disponible. Existe también el proceso inverso de proporcionar bloques para contener nuevos datos. Es evidente que se debe seleccionar algún bloque disponible y usarlo parcial o totalmente para contener los datos nuevos. Hay dos temas a tener en cuenta. Primero, ¿qué bloque vacío se selecciona? Segundo, si es necesario usar sólo parte de un bloque seleccionado, ¿qué parte se usa?

La segunda pregunta es fácil de aclarar. Si se utiliza un bloque con contador c y se requieren $d < c$ bytes de ese bloque, se escogen los últimos d bytes. De esta forma, sólo es necesario reemplazar c por $c-d$, y el bloque vacío restante puede permanecer en la lista de disponibles \dagger . \square

Ejemplo 12.6. Supóngase que se requieren 400 bytes para la variable W en la situación representada en la figura 12.12. Podría decidirse tomar los 400 bytes finales de los 600 que existen en el primer bloque de la lista de disponibles. Tal situación se muestra en la figura 12.14.

La selección de un bloque para colocar datos nuevos no es fácil, debido a que existen conflictos entre las metas de tales estrategias. Se desea, por un lado, poder tomar con rapidez un bloque vacío en el cual quepan los datos y, por otro, hacer una selección de un bloque vacío que reduzca la fragmentación. Hay dos estrategias que representan extremos en el espectro conocidas como «primer ajuste» y «mejor ajuste», y se describen a continuación.

1. *Primer ajuste.* Para seguir la estrategia de *primer ajuste*, cuando se necesita un bloque de tamaño d , se revisa la lista de disponibles desde el principio hasta llegar a un bloque de tamaño $c \geq d$. Utilíicense las últimas d palabras en ese bloque, como se describió antes.
2. *Mejor ajuste.* Para seguir la estrategia de *mejor ajuste*, cuando se necesita un bloque de tamaño d , se examina toda la lista de disponibles para encontrar el bloque de tamaño mínimo d que sea lo más cercano posible a d . Se toman las últimas d palabras de ese bloque.

\dagger Si $c - d$ es tan pequeño que un contador y un apuntador no caben, hay que utilizar el bloque completo y eliminarlo de la lista de disponibles.

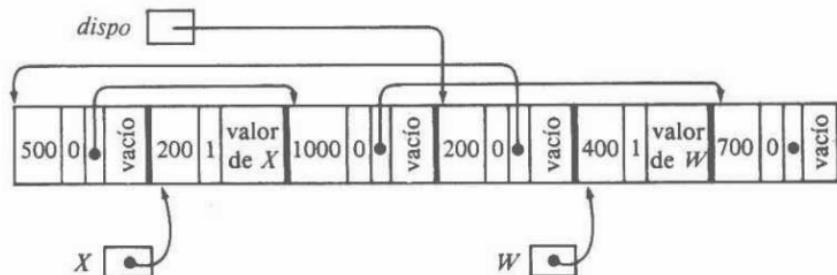


Fig. 12.14. Configuración de la memoria.

Pueden hacerse algunas observaciones acerca de estas estrategias. La estrategia de mejor ajuste es bastante más lenta que la de primer ajuste, ya que con esta última se puede esperar encontrar, en promedio, un bloque suficiente con rapidez, mientras que con la de mejor ajuste es necesario recorrer toda la lista de disponibles. La estrategia de mejor ajuste puede acelerarse si se mantienen listas de bloques disponibles de acuerdo con varias gamas de tamaños. Por ejemplo, se puede mantener una lista de bloques disponibles con longitud entre 1 y 16 bytes †, entre 17 y 32, entre 33 y 64, y así sucesivamente. Esta «mejora» no beneficia en forma apreciable la estrategia de primer ajuste, y, de hecho, puede alestarla si las estadísticas de los tamaños de bloques son malas. (Compárese la búsqueda del primer bloque de tamaño mayor que 32 en la lista de disponibles completa y en la lista de bloques de tamaño 17 a 32, por ejemplo.) Una última observación es que un espectro de estrategias se puede definir entre las dos planteadas aquí, buscando el mejor ajuste entre los primeros k bloques disponibles para algún tamaño fijo k .

La estrategia de mejor ajuste parece reducir la fragmentación en comparación con la de primer ajuste, en el sentido de que la primera tiende a producir «fragmentos» muy pequeños, es decir, bloques abandonados. Y aunque el número de esos fragmentos es casi el mismo que para el primer ajuste, tienden a ocupar un área menor. Sin embargo, el mejor ajuste no tiende a producir «fragmentos» de tamaño medio. En cambio, los bloques disponibles tienden a ser fragmentos muy pequeños o bloques devueltos al espacio disponible. Como consecuencia, hay secuencias de peticiones que la estrategia de primer ajuste puede satisfacer y la de mejor ajuste no, y viceversa.

Ejemplo 12.7. Supóngase, como en la figura 12.12, que la lista de disponibles consta de los bloques de tamaño 600, 500, 1000 y 700, en ese orden. Si se emplea la estrategia de primer ajuste y se hace una petición de un bloque de tamaño 400, se obtiene el bloque de tamaño 600 que es el primero de la lista en el cual cabe un bloque de tamaño 400. La lista de disponibles tiene ahora bloques de tamaño 200, 500, 1000 y 700. Así que es imposible satisfacer de inmediato tres peticiones de bloques de tamaño 600 (aunque se puede hacer después de combinar bloques vacíos adyacentes o al recorrer bloques utilizados sobre la estructura dinámica).

† En realidad, hay un tamaño de bloque mínimo mayor que 1, puesto que los bloques deben contener un apuntador, un contador y un bit vacío/lleno si se van a encadenar a una lista de disponibles.

No obstante, si se aplicara la estrategia de mejor ajuste con la lista de disponibles 600, 500, 1000 y 700, y llegara la petición de 400, podría colocarse donde ajusta mejor, esto es, en el bloque de 500, dejando una lista de bloques disponibles de 600, 100, 1000 y 700. En este caso, se pueden satisfacer tres peticiones de bloques de tamaño 600 sin necesidad de recurrir a la reorganización del almacenamiento.

Por otro lado, hay situaciones donde, al empezar con la lista 600, 500, 1000, 700 de nuevo, la estrategia de mejor ajuste puede fallar, mientras que la de primer ajuste puede funcionar sin necesidad de reorganización del almacenamiento. Dada la petición de 400 bytes, el mejor ajuste puede, igual que antes, dejar la lista en 600, 100, 1000 y 700, mientras que el primer ajuste la deja en 200, 500, 1000 y 700. Supóngase que las dos peticiones siguientes son de 1000 y 700, de manera que cualquier estrategia asignaría completamente los dos últimos bloques vacíos, dejando 600 y 100 en el caso del mejor ajuste, y 200 y 500, en el caso del primer ajuste. Ahora bien, el primer ajuste puede satisfacer peticiones de bloques de tamaño 200 y 500, mientras que el mejor ajuste obviamente no podrá. □

12.5 Sistemas de manejo de memoria por afinidades (*buddy systems*)

Hay una familia de estrategias para mantener una estructura dinámica que evite parcialmente los problemas de fragmentación y distribución difícil de bloques vacíos. Esas estrategias, llamadas «sistemas de manejo de memoria por afinidades», consumen poco tiempo al combinar bloques vacíos adyacentes. El inconveniente es que los bloques llegan en un surtido limitado de tamaños, así que se puede desperdiciar algún espacio colocando datos en un bloque más grande de lo necesario.

La idea central de todos los sistemas por afinidades es que los bloques son sólo de ciertos tamaños; por ejemplo, $s_1 < s_2 < s_3 < \dots < s_k$ son todos los tamaños entre los cuales pueden encontrarse los bloques. Algunas selecciones comunes para la secuencia s_1, s_2, \dots son 1, 2, 4, 8, ... (el sistema de afinidades exponencial) y 1, 2, 3, 5, 8, 13, ... (el sistema de afinidades de Fibonacci, donde $s_{i+1} = s_i + s_{i-1}$). Todos los bloques vacíos de tamaño s_i están enlazados en una lista, y hay un arreglo de encabezados de lista de disponibles, una para cada tamaño s_i permitido †. Si se requiere un bloque de tamaño d para un grupo de datos nuevo, se escoge un bloque disponible de tamaño s_i tal que $s_i \geq d$, pero $s_{i-1} < d$, esto es, el tamaño más pequeño permitido en el cual caben los datos nuevos.

Surgen algunas dificultades cuando no existen bloques vacíos del tamaño deseado s_i . En ese caso, es necesario encontrar un bloque de tamaño s_{i+1} y dividirlo en dos, uno de tamaño s_i y el otro de tamaño $s_{i+1} - s_i$ ††. El sistema de afinidades impone la restricción de que $s_{i+1} - s_i$ sea alguna s_j , para $j \leq i$. Ahora se ve la forma en

† Puesto que los bloques vacíos deben contener apuntadores (y, como se verá, otra información también), en realidad no se inicia la secuencia de tamaños permitidos en 1, sino en algún número apropiado más grande en la secuencia, como 8 bytes.

†† Por supuesto, si no hay bloques vacíos de tamaño s_{i+1} , se crea uno dividiendo un bloque de tamaño s_{i+2} , y así sucesivamente. Si no existen bloques de ningún tamaño mayor, en realidad no hay espacio y es necesario reorganizar la estructura dinámica como en la sección siguiente.

que se puede restringir la selección de valores para las s_i . Si se hace que $j = i - k$, para alguna $k \geq 0$, dado que $s_{i+1} - s_i = s_{i-k}$ se sigue que

$$s_{i+1} = s_i + s_{i-k} \quad (12.1)$$

La ecuación (12.1) es aplicable cuando $i > k$, y junto con los valores para s_1, s_2, \dots, s_k , determina completamente a s_{k+1}, s_{k+2}, \dots . Por ejemplo, si $k = 0$, (12.1) queda como

$$s_{i+1} = 2s_i \quad (12.2)$$

empezando con $s_1 = 1$ en (12.2), se obtiene la secuencia exponencial 1, 2, 4, 8, ... Por supuesto, no importa el valor inicial de s_1 ; las s crecerán exponencialmente en (12.2). Otro ejemplo, si $k = 1$, $s_1 = 1$ y $s_2 = 2$, (12.1) se transforma en

$$s_{i+1} = s_i + s_{i-1} \quad (12.3)$$

esta ecuación define la sucesión de Fibonacci: 1, 2, 3, 5, 8, 13, ...

Para cualquier valor de k escogido en (12.1), habrá un *sistema de afinidades de orden k-ésimo*. Para cualquier k , la secuencia de tamaños permitidos crece exponencialmente; esto es, la razón s_{i+1}/s_i se aproxima a alguna constante mayor que uno. Por ejemplo, para $k = 0$, s_{i+1}/s_i es exactamente 2. Para $k = 1$ la razón se aproxima a la «razón dorada» $(\sqrt{5}+1)/2 = 1.618$, y la razón decrece conforme k crece, pero nunca es menor que 1.

Distribución de bloques

En el sistema de afinidades de orden k -ésimo, cada bloque de tamaño s_{i+1} se puede considerar formado por un bloque de tamaño s_i y otro de tamaño s_{i-k} . Para ser más específicos, supóngase que el bloque de tamaño s_i está a la izquierda (en las posiciones numeradas menores) del bloque de tamaño s_{i-k} †. Si se considera la estructura dinámica como un solo bloque de tamaño s_n , para alguna n grande, entonces las posiciones en las que los bloques de tamaño s_i pueden empezar están completamente determinadas.

Las posiciones en el sistema exponencial, o de orden 0-ésimo, son fáciles de determinar. Si se supone que las posiciones en la estructura están numeradas a partir de 0, un bloque de tamaño s_i empieza en cualquier posición que comience con un múltiplo de 2^i , esto es, 0, $2^i, \dots$. Más aún, cada bloque de tamaño 2^{i+1} , que empieza, por ejemplo, en $j2^{i+1}$ está compuesto de dos «grupos afines» de tamaño 2^i , que empiezan en las posiciones $(2j)2^i$, que son $j2^{i+1}$, y $(2j+1)2^i$. Así, es fácil encontrar el grupo afín de un bloque de tamaño 2^i . Si empieza en algún múltiplo par de 2^i , como $(2j)2^i$, su afín está a la derecha, en la posición $(2j+1)2^i$. Si empieza en un múltiplo impar de 2^i , por ejemplo, en $(2j+1)2^i$, su afín está a la izquierda, en $(2j)2^i$.

Ejemplo 12.8. No es simple el comportamiento de los sistemas de afinidades de orden mayor que cero. La figura 12.15 muestra el sistema de afinidades de Fibonacci

† A veces es conveniente considerar que los bloques de tamaño s_i, s_{i-k} forman un bloque de tamaño s_{i+1} como «afines»; de ahí el término «sistema de afinidades».

utilizado en una estructura dinámica de tamaño 55, con bloques de tamaños s_i , s_2, \dots , $s_8 = 2, 3, 5, 8, 13, 21, 34$ y 55. Por ejemplo, el bloque de tamaño 3 que empieza en 26 es afín al bloque de tamaño 5 que empieza en 21; juntos, forman el bloque de tamaño 8 que empieza en 21 y que es afín al bloque de tamaño 5 que empieza en 29. Juntos, estos conforman el bloque de tamaño 13 que empieza en 21, y así sucesivamente. □

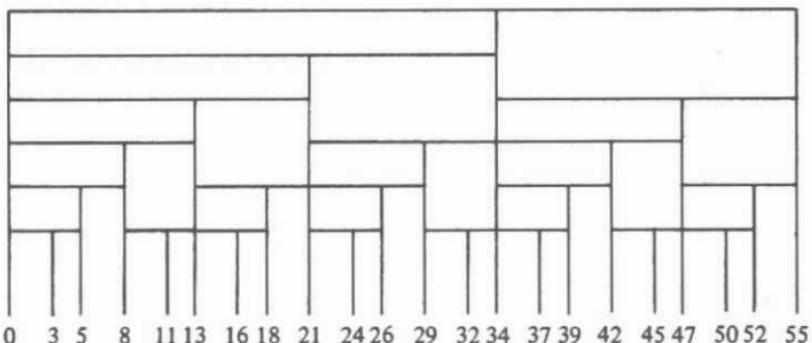


Fig. 12.15. División de una estructura dinámica según el sistema de afinidades de Fibonacci.

Asignación de bloques

Si se requiere un bloque de tamaño n , se escoge cualquiera de la lista de disponibles de tamaño s_i , donde $s_i \geq n$ e $i = 1$ o $s_{i-1} < n$, esto es, el bloque de mejor ajuste. En un sistema de afinidades de orden k -ésimo, si no hay bloques de tamaño s_i , es posible elegir un bloque de tamaño s_{i+1} o s_{i+k+1} para dividirlo, obteniendo en cualquier caso algún bloque de tamaño s_i . Si no existen bloques de esos tamaños, se crea uno aplicando la estrategia de división recursivamente para el tamaño s_{i+1} .

Sin embargo, hay una pequeña trampa. En un sistema de orden k -ésimo, puede ser que no se dividan los bloques de tamaño s_1, s_2, \dots, s_k , ya que de eso puede resultar un bloque de tamaño menor que s_1 . Se debe usar dicho bloque completo si no se encuentra disponible otro menor. Este problema no surge si $k = 0$, esto es, en el sistema exponencial. Puede reducirse en el sistema de Fibonacci si se empieza con $s = 1$, pero esa selección quizás no sea aceptable, puesto que los bloques de tamaño uno (byte o palabra, tal vez), pueden ser muy pequeños para contener un apuntador y un bit vacío/lleno.

Devolución de bloques al espacio disponible

Cuando un bloque queda listo para reutilizarse, se puede ver una de las ventajas de los sistemas de afinidades. Algunas veces se puede reducir la fragmentación combinando el bloque que quedó disponible recientemente con su afín, si éste está dispo-

nible también †. De hecho, si fuera el caso, se puede combinar el bloque resultante con su afín, si está vacío, y así sucesivamente. La combinación de grupos afines vacíos requiere sólo una cantidad constante de tiempo, lo que la convierte en una alternativa atractiva cuando se presentan combinaciones periódicas de bloques vacíos adyacentes, como se sugirió en la sección anterior, que requiere un tiempo proporcional al número de bloques vacíos.

El sistema exponencial hace que la localización de grupos afines sea especialmente fácil. Si se acaba de devolver el bloque de tamaño 2^i que empieza en $p2^i$, su afín está en $(p+1)2^i$ si p es par, y en $(p-1)2^i$ si p es impar.

Para un sistema de orden $k \geq 1$, la localización de afines no es tan simple. Para hacerla más fácil es necesario almacenar cierta información en cada bloque.

1. Un bit vacío/lleno, como tienen todos los bloques.
2. El *índice de tamaño*, un entero i tal que el bloque sea de tamaño s_i .
3. El *contador de afinidad izquierdo*, descrito a continuación.

En cada par de afines, uno (el *afín izquierdo*) está a la izquierda del otro (el *afín derecho*). Intuitivamente, el contador de afinidad izquierdo de un bloque dice cuántas veces consecutivas éste es todo o parte de un afín izquierdo. En un aspecto formal, toda la estructura dinámica, tratada como un bloque de tamaño s_n , tiene un contador de afinidad izquierdo igual a 0. Cuando se divide cualquier bloque de tamaño s_{i+1} , con el contador de afinidad izquierdo b , en bloques de tamaño s_i y s_{i-k} , que son los afines izquierdo y derecho, respectivamente, el afín izquierdo tiene un contador $b+1$, mientras que el derecho tiene un contador igual a 0, independiente de b . Por ejemplo, en la figura 12.15, el bloque de tamaño 3 que empieza en 0 tiene un contador de afinidad izquierdo igual a 6, y el bloque de tamaño 3 que empieza en 13 tiene un contador izquierdo igual a 2.

Además de la información anterior, los bloques vacíos, pero no los utilizados, tienen apunadores de avance y de retroceso para la lista de disponibles del tamaño adecuado. Los apunadores bidireccionales facilitan las combinaciones de afines, que deben eliminarse de la lista de disponibles.

La forma en que se maneja esta información es como sigue. Supóngase que k es el orden del sistema de afinidades. Cualquier bloque que empiece en la posición p con un contador de afinidad izquierdo igual a 0 es un afín derecho. Así, si tiene índice de tamaño j , su afín izquierdo es de tamaño s_{j+k} y empieza en la posición $p-s_{j+k}$. Si el contador de afinidad izquierdo es mayor que 0, entonces el bloque es un afín izquierdo de un bloque de tamaño s_{j-k} , el cual está localizado en la posición $p+s_j$.

Si se combina un afín izquierdo de tamaño s_i , teniendo un contador de afinidad izquierdo igual a b , con un afín derecho de tamaño s_{i-k} , el bloque resultante tendrá un índice de tamaño $i+1$, que empieza en la misma posición que el bloque de tamaño s_i , y tiene un contador de afinidad izquierdo $b-1$. Así, la información necesaria puede mantenerse con facilidad al combinar dos afines vacíos. Como ejercicio, se puede comprobar que la información se mantiene al dividir un bloque vacío de tamaño s_{i+1} en un bloque ocupado de tamaño s_i y uno vacío de tamaño s_{i-k} .

† Igual que en la sección anterior, se supone que un bit de cada bloque está reservado para indicar si el bloque está en uso o vacío.

Si se mantiene toda esta información, y se ligan las listas de disponibles en ambas direcciones, sólo se emplea una cantidad constante de tiempo en cada división de un bloque o se combinan los afines en un bloque más grande. Como el número de combinaciones nunca puede exceder del número de divisiones, la cantidad total de trabajo es proporcional a este número. No es difícil reconocer que la mayor parte de las solicitudes de un bloque no requieren divisiones, ya que hay disponible un bloque del tamaño correcto. Sin embargo, hay situaciones extremas en las que cada asignación requiere bastantes divisiones. El ejemplo extremo es donde se solicita repetidamente un bloque del tamaño más pequeño y después se devuelve. Si hay n tamaños diferentes, se requieren por lo menos n/k divisiones en un sistema de orden k -ésimo, las cuales estarán seguidas de n/k combinaciones cuando se devuelve el bloque.

12.6 Compactación del almacenamiento

Hay ocasiones en las que, aun después de combinar todos los bloques adyacentes, no es posible satisfacer una solicitud de un bloque nuevo. Esto puede deberse simplemente a que en el montón no hay espacio para formar un bloque del tamaño deseado. Pero es más típica una situación como la que se muestra en la figura 12.11, donde si bien hay 2200 bytes disponibles, no se puede satisfacer una solicitud de un bloque mayor de 1000. El problema es que el espacio disponible está dividido entre varios bloques no contiguos. Hay dos enfoques generales a este problema.

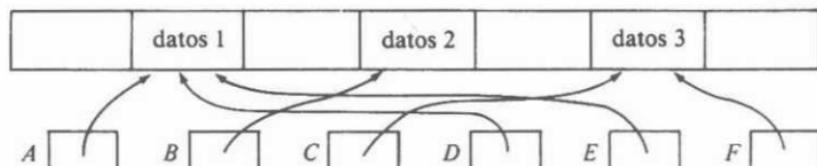
1. Lograr que el espacio disponible para el conjunto de datos pueda estar compuesto de varios bloques vacíos. Si ocurre así, también se puede requerir que todos los bloques sean del mismo tamaño y consistan en espacio para un apuntador y para los datos. En un bloque utilizado, el apuntador indica el siguiente bloque usado para los datos, y es nulo en el último bloque. Por ejemplo, si se estuvieran almacenando datos cuyo tamaño fuera casi siempre pequeño, se escogerían bloques de tamaño igual a 16 bytes, con 4 utilizados para un apuntador y 12 para los datos. Si los conjuntos de datos fueran en general más grandes, podrían escogerse bloques con varios cientos de bytes, asignando de nuevo cuatro para un apuntador y el resto para los datos.
2. Cuando falla la combinación de bloques vacíos adyacentes, y no se es capaz de proveer un bloque bastante grande, se mueven los datos por la estructura dinámica de manera que todos los bloques llenos queden en el extremo izquierdo (posición más baja), y se forme un bloque grande disponible a la derecha.

El método (1), que usa cadenas de bloques para un conjunto de datos, tiende a consumir mucho espacio. Si se escoge un tamaño de bloque pequeño, se emplea una fracción grande de espacio para «las cabeceras», los apuntadores necesarios para mantener las cadenas. Al utilizar bloques grandes, se ocupa menos espacio en las cabeceras, pero muchos bloques estarán casi desperdiciados, almacenando pocos datos. La única situación en que esta clase de enfoque es preferible, es cuando los conjuntos de datos típicos sean muy grandes. Por ejemplo, muchos sistemas de archi-

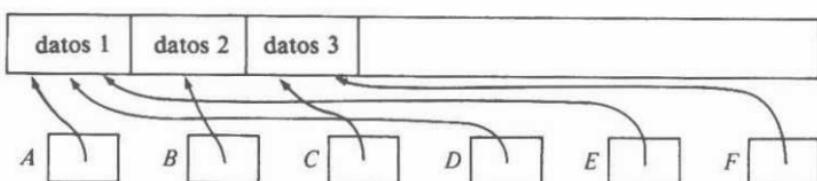
vos trabajan de esta forma, dividiendo la estructura dinámica, que por lo común es una unidad de disco, en bloques del mismo tamaño, por ejemplo 512 a 4096 bytes, dependiendo del sistema. Como muchos archivos son mucho más grandes que esos números, no se desperdicia demasiado el espacio, y los apuntadores a los bloques que conforman un archivo requieren relativamente poco espacio. La asignación de espacio bajo esta disciplina es casi directa, debido a lo que se ha aprendido en las secciones anteriores, por lo que no se analizará aquí la manera de realizarla.

Problema de la compactación

Un problema que debe enfrentarse con frecuencia es el de tomar una colección de bloques en uso, como la sugerida en la figura 12.16(a), donde cada bloque puede ser de tamaño diferente y estar apuntado por más de un apuntador, y correrlos hacia la izquierda hasta que todo el espacio disponible quede en el extremo derecho de la estructura dinámica, como se muestra en la figura 12.16(b). Como es natural, los apuntadores deben continuar apuntando a los mismos datos.



(a) Antes de la compactación



(b) Despues de la compactación

Fig. 12.16. Proceso de compactación del almacenamiento.

Existen soluciones simples a este problema asignando un poco de espacio adicional en cada bloque, y se analizará otro método más complicado, pero eficiente, que no requiere espacio extra en los bloques utilizados, sólo el que se ha usado hasta ahora en cualquiera de los esquemas de manejo del espacio estudiados, es decir, un bit vacío/lleto y un contador que indique el tamaño del bloque.

Un esquema simple para la compactación consiste en revisar primero todos los bloques desde la izquierda, sean llenos o vacíos, y calcular una «dirección de avance» para cada bloque lleno. La dirección de avance de un bloque es su posición actual menos la suma de todo el espacio vacío que se encuentra a su izquierda, esto es, la posición a la cual habrá de pasar finalmente. Es fácil calcular la dirección de avance. Al revisar los bloques desde la izquierda, se acumula la cantidad de espacio vacío que se encuentra para sustraer esta cantidad a la posición de cada bloque visto. El algoritmo se plantea en la figura 12.17.

```

(1)    var
        p: integer; { posición del bloque actual }
        hueco: integer; { cantidad total de espacio vacío hallado
                          hasta ahora }
    begin
(2)      p := extremo izquierdo de la estructura dinámica;
(4)      hueco := 0;
(5)      while p ≤ extremo derecho de la estructura dinámica do begin
            { hace que p apunte al bloque B }
(6)          if B está vacío then
                hueco := hueco + contador en el bloque B
            else { B está lleno }
                dirección de avance de B := p - hueco;
(8)                p := p + contador en el bloque B
    end
end;

```

Fig. 12.17. Cálculo de las direcciones de avance.

Una vez calculada la dirección de avance, se revisan todos los apuntadores al montón †. Se sigue cada apuntador hacia algún bloque *B* para reemplazar el apuntador por la dirección de avance encontrada en ese bloque. Por último, se mueven todos los bloques llenos hacia su dirección de avance. Este proceso es similar al de la figura 12.17, con la línea (8) reemplazada por

```

for i:= p to p-1 + contador en B do
    estructura dinámica [i - hueco] := estructura dinámica[i];

```

para pasar el bloque *B* a la izquierda en una cantidad igual a *hueco*. Obsérvese que el movimiento de bloques llenos, que requiere un tiempo proporcional a la cantidad de bloques en uso dentro de la estructura dinámica, probablemente dominará los demás costos de compactación.

† En el resto del texto se supone que la colección de dichos apuntadores está disponible. Por ejemplo, una realización normal de SNOBOL almacena pares que constan de un nombre de variable y un apuntador al valor de ese nombre en una tabla de dispersión, con la función de dispersión calculada a partir del nombre. Examinar la tabla completa permite visitar todos los apuntadores.

Algoritmo de Morris

F. L. Morris descubrió un método para la compactación de la estructura dinámica sin usar espacio en los bloques para la dirección de avance. No obstante, requiere un bit de marca asociado con cada apuntador y con cada bloque para indicar el final de una cadena de apuntadores. La idea esencial es crear una cadena de apuntadores que salen de una posición fija en cada bloque lleno y enlazados todos a ese bloque. Por ejemplo, en la figura 12.16(a) se observan tres apuntadores, A , D y E , apuntando al bloque lleno del extremo izquierdo. En la figura 12.18, se encuentra la cadena deseada de apuntadores. Una porción de los datos de tamaño igual a la de un apuntador se ha extraído del bloque y colocado al final de la cadena, donde se encontraba el apuntador A .

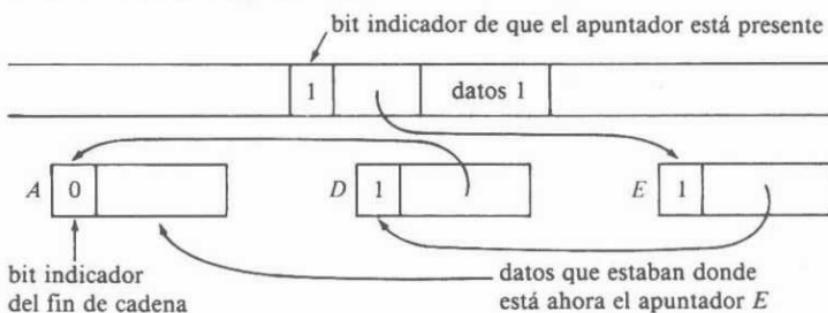


Fig. 12.18. Encadenamiento de apuntadores.

El método para crear dichas cadenas de apuntadores es como sigue. Se examinan todos los apuntadores en cualquier orden conveniente. Supóngase un apuntador p al bloque B . Si el bit de marca en el bloque B es cero, entonces p es el primer apuntador encontrado que apunta a B . Se colocan en p los contenidos de las posiciones de B utilizadas para la cadena de apuntadores, y se hace que esas posiciones de B apunten a p . Despues, se hace el bit de marca en B igual a 1, indicando que ahora tiene un apuntador, y el bit de marca en p igual a 0, indicando el fin de la cadena de apuntadores y la presencia de los datos desplazados.

Ahora bien, al considerar el apuntador p al bloque B , el bit de marca en B está en uno; entonces, B ya tiene la cabeza de la cadena de apuntadores. Se copia el apuntador de B en p , para que B apunte a p , y se hace el bit de marca en p igual a 1; de este modo, p queda, efectivamente, en la cabeza de la cadena.

Una vez que se tienen todos los apuntadores a cada bloque enlazados en una cadena que parte desde ese bloque, se pasan los bloques llenos tan a la izquierda como sea posible, igual que en el sencillo algoritmo analizado con anterioridad. Por último, se examina cada bloque en su nueva posición y se recorre su cadena de apuntadores. Cada apuntador localizado se actualiza para que apunte al bloque en su nueva posición. Al encontrar el final de la cadena se reinstalan los datos de B , contenidos en el último apuntador, a su lugar correcto en el bloque B y se fija el bit de marca del bloque igual a 0.

Ejercicios

- 12.1** Considérese la siguiente estructura dinámica de 1000 bytes, donde los bloques en blanco están en uso, y los bloques etiquetados están enlazados en una lista libre en orden alfabético. Los números indican la posición del primer byte de cada bloque.

0	100	200	400	500	575	700	850	900	999
<i>a</i>		<i>b</i>		<i>c</i>	<i>d</i>		<i>e</i>	<i>f</i>	

Supóngase que se realizan las siguientes peticiones:

- 1) asignar un bloque de 120 bytes,
- 2) asignar un bloque de 70 bytes,
- 3) devolver al frente de la lista de disponibles el bloque que se encuentra entre los bytes 700 y 849, y
- 4) asignar un bloque de 130 bytes.

Proporciónese la lista de espacio libre, en orden, después de ejecutar la secuencia de pasos anterior, suponiendo que los bloques libres se seleccionan de acuerdo con la estrategia de

- a) primer ajuste
- b) mejor ajuste.

- 12.2** Obsérvese la siguiente estructura dinámica, donde las regiones en blanco se encuentran en uso y las regiones etiquetadas están vacías.

0	100	200	300	500
	<i>a</i>		<i>b</i>	

Establézcase una secuencia de solicitudes que puedan satisfacerse al emplear

- a) primer ajuste, pero no mejor ajuste
- b) mejor ajuste, pero no primer ajuste

- *12.3** Supóngase que se utiliza un sistema de afinidades exponencial con tamaños 1, 2, 4, 8 y 16 en una estructura dinámica de tamaño 16. Si se solicita un bloque de tamaño n , para $1 \leq n \leq 16$, es necesario asignar un bloque de tamaño 2^i , donde $2^{i-1} < n \leq 2^i$. La porción no utilizada del bloque, si existe, no puede emplearse para satisfacer ninguna otra solicitud. Si se necesita un bloque de tamaño 2^i , $i < 4$, y no existe dicho bloque libre, primero se busca un bloque de tamaño 2^{i+1} y se divide en dos partes del mismo tamaño. Si no existe un bloque de tamaño 2^{i+1} , se busca y se parte un bloque de tamaño 2^{i+2} , y así sucesivamente. Si se llega a buscar un bloque libre de tamaño 32, el proceso falla y no puede satisfacerse la solicitud. A

efectos de este ejercicio, no se combinan bloques libres adyacentes en la estructura dinámica.

Existen secuencias de solicitud a_1, a_2, \dots, a_n cuya suma es menor que 16, tal que la última petición no puede satisfacerse. Por ejemplo, considérese la secuencia 5, 5, 5. La primera petición hace que el bloque inicial de tamaño 16 se parte en dos bloques de tamaño 8, uno de los cuales se usa para satisfacer la solicitud. El bloque libre restante de tamaño 8 satisface la segunda, y no queda espacio libre para satisfacer la tercera.

Encuéntrese una secuencia a_1, a_2, \dots, a_n de enteros entre 1 y 16 (no es necesario que sean idénticos), cuya suma sea lo más pequeña posible, tal que, tratada como una secuencia de solicitudes de bloques de tamaño a_1, a_2, \dots, a_n , la última petición no se satisface. Explíquese por qué esta secuencia de peticiones no se satisface, pero cualquier otra secuencia cuya suma sea más pequeña sí puede satisfacerse.

- 12.4** Considérese la compactación de memoria en la administración de bloques de igual tamaño. Supóngase que cada bloque consta de un campo para datos y otro para apuntador, y que se han marcado todos los bloques que se encuentran en uso actualmente. Los bloques están ubicados entre las localidades de memoria a y b . Se desea relocalizar todos los bloques activos de manera que ocupen memoria contigua partiendo de a . Recuérdese que, para relocalizar un bloque, el campo del apuntador de cualquier bloque que apunta al bloque relocalizado debe actualizarse. Diséñese un algoritmo para la compactación de bloques.
- 12.5** Se da un arreglo de tamaño n . Proporcionese un algoritmo para correr todos los elementos del arreglo k lugares en forma cíclica, en sentido contrario al de las manecillas del reloj, sólo con una cantidad constante de memoria adicional, independiente de k y n . *Sugerencia.* Téngase en cuenta qué sucede si se invierten los k primeros elementos, los $n-k$ últimos elementos y, por último, el arreglo completo.
- 12.6** Diséñese un algoritmo para sustituir una subcadena y de una cadena xyz por otra subcadena y' , con la menor cantidad posible de memoria adicional. ¿Cuál es la complejidad de tiempo y espacio de este algoritmo?
- 12.7** Escribase un programa para hacer una copia de una lista dada. ¿Cuál es la complejidad de tiempo y espacio del programa?
- 12.8** Escribase un programa para determinar si dos listas son idénticas. ¿Cuál es la complejidad de tiempo y espacio de este programa?
- 12.9** Obténgase el algoritmo de compactación de las estructuras dinámicas de Morris, mostrado en la sección 12.6.
- *12.10** Diséñese un esquema de asignación de almacenamiento para una situación en la que la memoria se asigna y libera en bloques de longitud 1 y 2. Proporcionense cotas sobre la eficiencia del algoritmo.

Notas bibliográficas

La administración eficiente almacenamiento es un tema central en muchos lenguajes de programación, incluyendo SNOBOL [Farber, Griswold y Polonsky (1964)], LISP [McCarthy (1965)], APL [Iverson (1962)] y SETL [Schwartz (1973)]. Nicholls [1975] y Pratt [1975] comentan las técnicas de administración de almacenamiento en el contexto de la compilación de lenguajes de programación.

El sistema de afinidades para asignación de almacenamiento fue publicado por primera vez por Knowlton [1965]. Los sistemas de afinidades de Fibonacci fueron estudiados por Hirschberg [1973].

El elegante algoritmo de marcado utilizado en la recolección de basura fue descubierto por Peter Deutsch (Deutsch y Bobrow [1966]) y por Schorr y Waite [1965]. El esquema de compactación de estructuras dinámicas de la sección 12.6 se debe a Morris [1978].

Robson [1971] y Robson [1974] analizan la cantidad de memoria requerida para algoritmos dinámicos de asignación de almacenamiento. Robson [1977] presenta un algoritmo acotado en espacio de trabajo para copiar estructuras cíclicas. Fletcher y Silver [1966] contiene otra solución al ejercicio 12.5 que utiliza poca memoria adicional.