Apuntes de Algorítmica

Andrés Marzal
María José Castro
Pablo Aibar

Borrador
6 de febrero de 2007

Capítulo 8
PROGRAMACIÓN DINÁMICA

La estrategia algorítmica conocida como **programación dinámica** es una técnica de resolución que encuentra aplicación en numerosos problemas de optimización. El tipo de problemas abordables es, principalmente, aquél en el que las soluciones factibles se pueden descomponer en una secuencia de elementos (caminos en un grafo, series de decisiones, etc.) y cuya función objetivo satisface cierta condición de separabilidad y monotonía. No obstante, la programación dinámica también encuentra aplicación en la búsqueda de otras estructuras óptimas (árboles óptimos) y en la realización eficiente de ciertos cálculos de naturaleza recursiva.

Programación dinámica: Dynamic Programming.

La programación dinámica guarda relación con divide y vencerás. Divide y vencerás basa su eficiencia en la división de un problema en dos o más subproblemas *de igual o parecida talla* e inferior a la del original cuyas soluciones se combinan. La división de un problema en subproblemas de talla inferior conduce de forma natural a un planteamiento recursivo. Divide y vencerás permite diseñar algoritmos eficientes cuando la división proporciona instancias de talla similar (como en *mergesort*, donde el vector se parte en dos vectores de talla similar) y, además, no hay llamadas repetidas en el árbol de llamadas recursivas (recordemos el problema del cálculo de la potencia entera de un número).

El método de programación dinámica se aplica, precisamente, en problemas que adolecen de estos inconvenientes. Para ellos, la programación dinámica ofrece un repertorio de técnicas que permiten reducir la complejidad computacional propia de una resolución puramente recursiva. La fundamental consiste en evitar la repetición de cálculos en las llamadas recursivas mediante el almacenamiento de los resultados que ya han sido calculados para su posterior reutilización. Esta técnica se conoce como «**memorización**».

> **!** La palabra «programación» no guarda relación con la escritura de programas para computadores, sino con la tabulación de resultados (o sea, se usa en el mismo sentido que en «programación lineal»).

Memorización: Memoization.

Por otra parte, una transformación recursivo-iterativa permite, en ocasiones, reduccio-

nes adicionales de complejidad espacial: no sólo ahorra la memoria que ocupa la pila de llamadas a función, sino que permite reducir el propio tamaño de la tabla de soluciones precalculadas.

Empezaremos la exposición desarrollando los conceptos fundamentales al hilo de la resolución de algunos problemas paradigmáticos:

- el cálculo del camino de menor coste en un grafo acíclico estructurado y el cálculo del camino más probable sobre el mismo grafo;
- y el problema de la mochila discreta.

Tras exponer los conceptos teóricos fundamentales, resolveremos los siguientes problemas de optimización:

- El problema del cálculo del camino más corto en un grafo acíclico.
- El problema del cálculo del camino más corto formado por k aristas en un grafo (con o sin ciclos).
- El problema del cálculo del camino más corto en un grafo sin ciclos negativos.
- El problema del desglose óptimo de una cantidad de dinero (que resolveremos de dos formas distintas).
- El problema del desglose óptimo de una cantidad de dinero con limitación del número de monedas de cada valor disponibles.
- El problema de la asignación óptima de recursos.
- Segmentación de un texto en palabras (en ausencia de espacios blancos).
- Formateo estéticamente óptimo de un párrafo.
- La distancia de edición entre dos cadenas.
- La subsecuencia común más larga a dos cadenas.
- Alineamiento temporal no lineal.
- Carga máxima de un camión por carreteras con limitación de carga.

Todos estos problemas se ponen en correspondencia con la búsqueda de un camino óptimo en cierto grafo. Un problema de optimización que propone la búsqueda de una estructura más compleja (un árbol) y que resolveremos por programación dinámica es el del cálculo de la parentización óptima del producto de matrices.

Si bien la programación dinámica se aplica en muchos problema de optimización, su uso no se limita a éstos: algunos problemas de cálculo de naturaleza recursiva pueden beneficiarse de las técnicas propias de la programación dinámica. Para ilustrar este aspecto, resolveremos los siguientes problemas:

- Cálculo de los números de Fibonacci.
- Conteo del número de posibles desgloses en monedas.
- Análisis de una cadena con un autómata finito no determinista.
- Análisis de una cadena con una gramática incontextual en forma normal de Chomsky mediante el algoritmo de Cocke-Younger-Kasami.

8.1. Un par de problemas sobre el río Congo

A lo largo del río Congo hay E embarcaderos a los que nombramos con los números enteros 1, 2, ..., E. Es posible ir en canoa desde un embarcadero a cualquiera de los

dos siguientes en la dirección de la corriente. No se puede navegar contra corriente, ni tampoco ir más allá del segundo embarcadero sin efectuar escala alguna. Los problemas que nos interesa resolver consisten en encontrar trayectos óptimos (en los términos que definiremos más adelante) que parten del embarcadero 1 y finalizan en el embarcadero E. La figura 8.1 muestra un grafo que modela el río con 7 embarcaderos en el que los vértices origen y destino se marcan, respectivamente, con una flecha entrante y una flecha saliente.

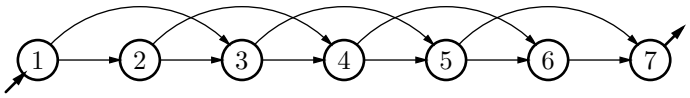


Figura 8.1: Grafo que modela el río Congo con 7 embarcaderos.

Vamos a plantear un par de problemas sobre el río Congo:

1. dada una función de ponderación que asigna un coste (positivo) a cada arco, calcular el camino de menor coste del primer al último embarcadero y su coste;
2. dada una función que asigna una probabilidad de transitar de cada embarcadero a cada uno de sus dos sucesores, calcular el camino más probable del primer al último embarcadero y su probabilidad.

El primer problema es, evidentemente, un caso particular del problema del camino más corto en un grafo y, por tanto, podríamos resolverlo mediante el algoritmo de Dijkstra. Como hay E vértices y $2E-3$ aristas en el grafo, el algoritmo de Dijkstra proporciona el coste del trayecto más barato en tiempo $O(E \lg E)$ y usa espacio $O(E)$. Si necesitamos conocer la secuencia de embarcaderos que corresponde a dicho trayecto, podemos recurrir a la técnica de recuperación del camino con punteros hacia atrás (véase la sección **??**) sin que ello suponga un aumento de la complejidad temporal o espacial.

Las técnicas de programación dinámica nos permitirán resolver el problema más eficientemente. La complejidad temporal se reducirá a $O(E)$ y la espacial pasará a ser $O(1)$ si sólo deseamos conocer el precio del trayecto más barato, y seguirá siendo $O(E)$ si deseamos, además, recuperar el camino óptimo.

El segundo problema no encaja exactamente en la definición del problema del camino más corto, pues no es un problema de minimización, sino de maximización, y no suma costes, sino que multiplica probabilidades. No obstante, veremos que es posible aplicar las mismas técnicas que en el primer problema y obtener la solución con el mismo coste temporal y espacial.

8.1.1. El problema del trayecto más barato en el río Congo

El coste que presenta ir en canoa del embarcadero i al $i+1$, para $1 \leq i \leq E-1$, se representa con $c(i, i+1)$, y el coste que presenta ir del embarcadero i al $i+2$, para $1 \leq i \leq E-2$, se representa con $c(i, i+2)$. Naturalmente, los costes son cantidades no negativas. La figura 8.1 muestra, sobre cada arco, el coste asociado en un río con 7 embarcaderos. Queremos ir del primer embarcadero al último. ¿Cuál es el camino de menor coste? ¿Qué coste presenta dicho camino?

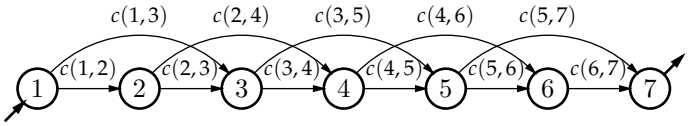


Figura 8.2: Grafo de la figura 8.1 con el coste asociado a cada arco.

Formalización

La dos preguntas que deseamos responder («¿Cuál es el camino de menor coste?» y «¿Qué coste presenta dicho camino?») corresponden a un problema de optimización: determinar el valor mínimo de una función objetivo (coste) que asigna un valor real a cada una de las soluciones factibles (secuencias válidas de embarcaderos). Empecemos por formalizar el problema para que se plantee en los términos precisos de un problema de optimización (véase la sección **??**).

Consideremos qué es un trayecto válido entre los vértices inicial y final: una secuencia

> **!** El proceso de formalización es muy similar en la mayor parte de los problemas de programación dinámica. Describimos qué entendemos por una solución: una serie de elementos simples que sigue ciertas reglas.

de embarcaderos $(e_1, e_2, \ldots, e_n)$ tal que

- empieza en el primer embarcadero: $e_1 = 1$;
- finaliza en el último embarcadero: $e_n = E$; y
- desde un embarcadero e_i se puede ir al embarcadero $e_i + 1$ o al $e_i + 2$, es decir, $1 \leq e_i - e_{i-1} \leq 2$ para $1 < i \leq n$.

El conjunto de los trayectos válidos es, pues,

$$X = \{(e_1, e_2, \ldots, e_n) \in [1..E]^+ \mid e_1 = 1; \quad e_n = E; \quad 1 \leq e_i - e_{i-1} \leq 2, \quad 1 < i \leq n\}.$$

En la terminología de los problemas de optimización X es el conjunto de las soluciones

> **!** Definimos así el conjunto de soluciones factibles.

factibles.

El coste de un trayecto se calcula como suma del coste de transitar directamente entre

> **!** A continuación definimos una función de coste o beneficio de las soluciones factibles en $\mathbb{R}$: la función objetivo.

cada par de embarcaderos contiguos en el trayecto:

$$C((e_1, e_2, \ldots, e_n)) = \sum_{1<i\leq n} c(e_{i-1}, e_i).$$

La función C es nuestra función objetivo.

Estamos interesados en encontrar el trayecto de menor coste (o uno cualquiera de

> **!** Nos proponemos encontrar una solución factible de menor coste o mayor beneficio, así como dicho coste o beneficio.

ellos si hay más de uno con coste mínimo):

$$(\hat{e}_1, \hat{e}_2, \ldots, \hat{e}_n) = \arg \min_{(e_1,e_2,\ldots,e_n)\in X} C((e_1, e_2, \ldots, e_n)).$$

Además, deseamos conocer el coste de dicho trayecto:

$$\min_{(e_1,e_2,\ldots,e_n)\in X} C((e_1, e_2, \ldots, e_n)).$$

EJERCICIOS

1 Una estrategia voraz de resolución consiste en empezar en el primer embarcadero y escoger siempre como siguiente embarcadero el que está unido al actual por el arco de menor coste. ¿Se encuentra la solución de menor coste siguiendo esta estrategia voraz?

Cálculo del menor coste: ecuación recursiva

De momento, vamos a dejar de lado el problema de determinar qué secuencia de embarcaderos permite efectuar el trayecto con ese coste mínimo. Empezaremos por plantear una ecuación que indique cómo calcular recursivamente el coste mínimo con el que podemos viajar del primer al último embarcadero:

$$\min_{(e_1,e_2,\ldots,e_n)\in X} \sum_{1<i\leq n} c(e_{i-1}, e_i) = \min_{\substack{(e_1,e_2,\ldots,e_n)|e_1=1;e_n=E;\\ 1\leq e_i-e_{i-1}\leq 2, 1<i\leq n}} \sum_{1<i\leq n} c(e_{i-1}, e_i).$$

Podemos sacar el último término del sumatorio:

> **!** Este paso es trivial y se basa en esta propiedad: $\sum_{1\leq i\leq n} x_i = \left(\sum_{1\leq i\leq n-1} x_i\right) + x_n$.

$$\min_{\substack{(e_1,e_2,\ldots,e_n)|e_1=1;e_n=E;\\ 1\leq e_i-e_{i-1}\leq 2, 1<i\leq n}} \sum_{1<i\leq n} c(e_{i-1}, e_i) = \min_{\substack{(e_1,e_2,\ldots,e_{n-1})|e_1=1;1\leq E-e_{n-1}\leq 2;\\ 1\leq e_i-e_{i-1}\leq 2, 1<i\leq n-1}} \left(\left(\sum_{1<i\leq n-1} c(e_{i-1}, e_i)\right) + c(e_{n-1}, E)\right).$$

Como el embarcadero e_{n-1} sólo puede ser $E-1$ o $E-2$, tenemos

> **!** El operador «min» se ha convertido en dos operadores «min» porque hemos separado la última decisión que podemos tomar. Este es un paso que siempre haremos.

$$\min_{\substack{(e_1,e_2,\ldots,e_n)|e_1=1;e_n=E;\\ 1\le e_i-e_{i-1}\le 2,1<i\le n}} \sum_{1<i\le n} c(e_{i-1},e_i) =$$

$$\min\left\{ \begin{array}{l} \min\limits_{\substack{(e_1,e_2,\ldots,e_{n-1})|e_1=1;e_{n-1}=E-2;\\ 1\le e_i-e_{i-1}\le 2,1<i\le n-1}} \left(\left(\sum\limits_{1<i\le n-1} c(e_{i-1},e_i)\right)+c(E-2,E)\right), \\ \min\limits_{\substack{(e_1,e_2,\ldots,e_{n-1})|e_1=1;e_{n-1}=E-1;\\ 1\le e_i-e_{i-1}\le 2,1<i\le n-1}} \left(\left(\sum\limits_{1<i\le n-1} c(e_{i-1},e_i)\right)+c(E-1,E)\right) \end{array} \right\}.$$

Podemos expresar la misma ecuación con una notación más compacta:

$$\min_{\substack{(e_1,e_2,\ldots,e_n)|e_1=1;e_n=E;\\ 1\le e_i-e_{i-1}\le 2,1<i\le n}} \sum_{1<i\le n} c(e_{i-1},e_i) =$$

$$\min_{j\in\{E-1,E-2\}} \left(\min_{\substack{(e_1,e_2,\ldots,e_{n-1})|e_1=1;e_{n-1}=j;\\ 1\le e_i-e_{i-1}\le 2,1<i\le n-1}} \left(\left(\sum_{1<i\le n-1} c(e_{i-1},e_i)\right)+c(j,E)\right)\right).$$

Para todo a, $a' \in \mathbb{R}$ tales que $a \le a'$ se cumple que $a+b \le a'+b$ para todo $b \in \mathbb{R}$. Así pues, para un valor fijo de $b \in \mathbb{R}$, se cumple

> **!** Esta propiedad es clave. Gracias a ella encontraremos una definición recursiva de $C(E)$.

$$\min_{a\in\mathbb{R}}(a+b) = \left(\min_{a\in\mathbb{R}} a\right)+b.$$

Esta propiedad es aplicable a la minimización interior, la que destacamos aquí con marco:

> **!** Equiparamos a con el sumatorio interior y b con $c(e_{n-1},E)$.

$$\min_{j\in\{E-1,E-2\}} \left(\boxed{\min_{\substack{(e_1,e_2,\ldots,e_{n-1})|e_1=1;e_{n-1}=j;\\ 1\le e_i-e_{i-1}\le 2,1<i\le n-1}} \left(\left(\sum_{1<i\le n-1} c(e_{i-1},e_i)\right)+c(j,E)\right)}\right) =$$

$$\min_{j\in\{E-1,E-2\}} \left(\left(\min_{\substack{(e_1,e_2,\ldots,e_{n-1})|e_1=1;e_{n-1}=j;\\ 1\le e_i-e_{i-1}\le 2,1<i\le n-1}} \sum_{1<i\le n-1} c(e_{i-1},e_i)\right)+c(j,E)\right).$$

Podemos concluir que

$$\min_{\substack{(e_1,e_2,\ldots,e_n)|e_1=1;e_n=E;\\ 1\le e_i-e_{i-1}\le 2,1<i\le n}} \sum_{1<i\le n} c(e_{i-1},e_i) = \min_{j\in\{E-1,E-2\}} \left(\left(\min_{\substack{(e_1,e_2,\ldots,e_{n-1})|e_1=1;e_{n-1}=j;\\ 1\le e_i-e_{i-1}\le 2,1<i\le n-1}} \sum_{1<i\le n-1} c(e_{i-1},e_i)\right)+c(j,E)\right).$$

Comparemos la expresión en la parte izquierda de la igualdad con la minimización interior de la parte derecha. Podemos advertir que son muy similares. De hecho, la parte izquierda se puede interpretar como «mínimo coste con el que podemos desplazarnos del embarcadero 1 al embarcadero E» y la minimización interior puede interpretarse como «mínimo coste con el que podemos desplazarnos del embarcadero 1 al embarcadero j», donde j es o bien el embarcadero $E-1$ o bien el embarcadero $E-2$. Abusando de la notación, denotemos con $C(j)$ al «mínimo coste con el que podemos desplazarnos del embarcadero 1 al embarcadero j»:

$$C(j) = \min_{\substack{(e_1,e_2,\ldots,e_n)|e_1=1;e_n=j;\\ 1\leq e_i-e_{i-1}\leq 2, 1<i\leq n}} \sum_{1<i\leq n} c(e_{i-1},e_i).$$

Ahora podemos reescribir la anterior ecuación así:

$$C(E) = \min_{j\in\{E-1,E-2\}} (C(j) + c(j,E)),$$

o sea,

$$C(E) = \min \left\{ \begin{array}{l} C(E-2) + c(E-2,E), \\ C(E-1) + c(E-1,E) \end{array} \right\}.$$

El razonamiento seguido es igualmente válido si queremos calcular el coste del camino más barato entre el primer embarcadero y un embarcadero i cualquiera tal que $2 < i \leq E$, así que esta ecuación también es válida:

$$C(i) = \min \left\{ \begin{array}{l} C(i-2) + c(i-2,i), \\ C(i-1) + c(i-1,i) \end{array} \right\}.$$

El embarcadero 2 es especial: sólo se puede llegar a él desde el primer embarcadero. El camino óptimo que finaliza en el segundo embarcadero tiene, pues, coste

$$C(2) = C(1) + c(1,2).$$

También el embarcadero 1 es especial. El único trayecto que permite ir del embarcadero 1 al mismo embarcadero 1 es el camino formado por un sólo elemento (él mismo) y ello no supone coste alguno. Podemos concluir que

$$C(1) = 0.$$

Y si revisamos ahora la fórmula de $C(2)$, tenemos que

$$C(2) = c(1,2).$$

Podemos agrupar las diferentes ecuaciones alcanzadas en ésta:

$$C(i) = \begin{cases} 0, & \text{si } i = 1; \\ c(1,2), & \text{si } i = 2; \\ \min(C(i-2) + c(i-2,i), C(i-1) + c(i-1,i)), & \text{si } i > 2, \end{cases} \tag{8.1}$$

donde, recordemos, $C(i)$ es coste del camino óptimo (el más barato) que parte del embarcadero 1 y llega al embarcadero i-ésimo. El coste del trayecto con que un turista viaja del primer al último embarcadero de la forma más económica es $C(E)$.

Algoritmo recursivo

Es fácil implementar un programa Python que cálcule $C(E)$ recursivamente. En este programa, representamos la función de coste del viaje directo entre embarcaderos, c, con un diccionario; o sea, $c[i\text{-}1, i]$ almacena el coste de viajar directamente de $i-1$ a i y $c[i\text{-}2, i]$ la de viajar directamente de $i-2$ a i:

> **!** Observa el estilo de programación usado, pues recurriremos a él más veces. La función *recursive_cheapest_price* calcula el precio del trayecto más barato y recibe un diccionario con el coste de cada trayecto directo entre dos embarcaderos conectados, c. Dentro se define la función C, que corresponde a la función recursiva (8.1). Las referencias al diccionario c en la función C son referencias al parámetro c de *recursive_cheapest_price*.

congo.py

```
1 def recursive_cheapest_price(E, c):
2     def C(i):
3         if i == 1:   return 0
4         elif i == 2: return c(1,2)
5         else:        return min(C(i-2) + c(i-2,i), C(i-1) + c(i-1,i))
6     return C(E)
```

El valor de $C(E)$ se obtiene efectuando la llamada *recursive_cheapest_price*(E, c). Veamos cómo funciona con el ejemplo de la figura 8.3:

test_congo.py

```
1 from congo import recursive_cheapest_price
2 from graph import WeightingFunction
3
4 c = WeightingFunction({(1,2): 4, (1,3): 6, (2,3): 3,
5                        (2,4): 7, (3,4): 1, (3,5): 4,
6                        (4,5): 2, (4,6): 7, (5,6): 4,
7                        (5,7): 5, (6,7): 9})
8
9 print 'Coste mínimo del embarcadero 1 al 7:', recursive_cheapest_price(7, c)
```

```
Coste mínimo del embarcadero 1 al 7: 14
```

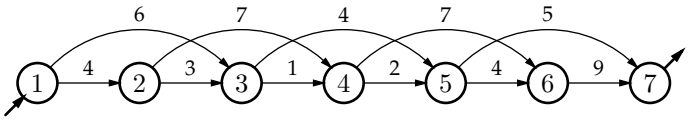


Figura 8.3: Instancia del problema del trayecto más barato en el río Congo.

Estudiemos el árbol de llamadas recursivas efectuadas al calcular $C(7)$ y que mostramos en la figura 8.4. Tiene interés observar cómo cada rama del árbol corresponde a un trayecto diferente. La rama de más a la izquierda, por ejemplo, corresponde al trayecto $(1, 3, 5, 7)$, y la de más a la derecha, al camino $(1, 2, 3, 4, 5, 6, 7)$. El árbol permite interpretar que el algoritmo explora recursivamente todo posible trayecto y «escoge» el de menor coste.

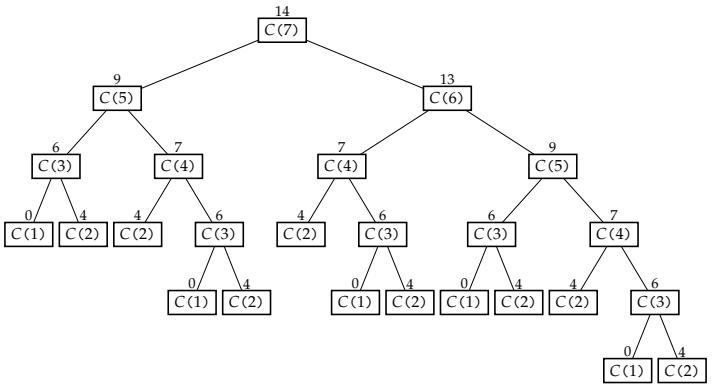


Figura 8.4: Árbol de llamadas para el cálculo recursivo de $C(7)$ de recursive_cheapest_price. Cada nodo se etiqueta con los argumentos de la correspondiente llamada a la función C. Sobre cada vértice indicamos el valor devuelto por la función.

El problema radica en que ciertas llamadas se ejecutan en más de una ocasión. La llamada a `C(5)`, por ejemplo, se ejecuta dos veces. Cada llamada a `C(5)` comporta nuevas llamadas a `C(3)` y `C(4)`, que también se efectúan al llamar a `C(6)`. Se produce un efecto en cadena que hace que el número de llamadas a `C(2)` y `C(1)`, hojas del árbol, acabe siendo de 8 y 5, respectivamente.

Podemos calcular la complejidad temporal del algoritmo a partir del número de llamadas, $N(E)$, efectuadas a la función C cuando calculamos $C(E)$. Esta ecuación recursiva nos proporciona dicho número:

$$N(E) = \begin{cases} c_0, & \text{si } i \leq 2; \\ N(E-2) + N(E-1) + c_1, & \text{si } i > 2; \end{cases}$$

donde c_0 y c_1 son enteros positivos. Es evidente que $N(E)$ es mayor que $F(E)$, el número de Fibonacci de índice E, así que podemos concluir que el número de llamadas crece, cuanto menos, exponencialmente con E. Pero, teóricamente, no deberían ser necesarias más de E llamadas recursivas, pues no hay más que E costes diferentes que calcular, uno por embarcadero.

Memorización de resultados intermedios

Una forma de resolver este problema consiste en memorizar los resultados que ya han sido calculados. Una simple tabla puede poner en correspondencia argumentos de la función y valores devueltos. Antes de realizar una llamada recursiva se puede comprobar si la tabla contiene o no el resultado buscado. Si es así, no hace falta efectuar la llamada.

congo.py *(cont.)*

```
8  def memoized_cheapest_price(E, c):
9      R = {} # Almacén de resultados intermedios.
10     def C(i):
11         if i == 1:    R[1] = 0
```

```
12        elif i == 2: R[2] = c(1,2)
13        else:
14            if i-2 not in R: C(i-2)
15            if i-1 not in R: C(i-1)
16            R[i] = min( R[i-1] + c(i-1,i), R[i-2] + c(i-2,i) )
17        return R[i]
18
19    return C(E)
```

El árbol de llamadas recursivas pasa a ser, ahora, el que se muestra en la figura 8.5. Con la nueva versión sólo ejecutamos una llamada a la función C por cada embarcadero. La complejidad temporal es, pues, $O(E)$. La complejidad espacial también es $O(E)$: ése es el orden del tamaño del almacén de resultados R y la profundidad de la pila de llamadas a función.

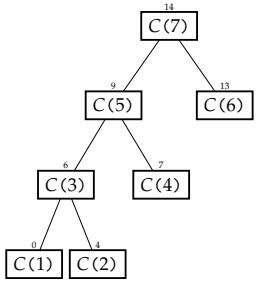


Figura 8.5: Árbol de llamadas de $C(7)$ en memoized_cheapest_price.

Una versión iterativa

El algoritmo con memorización paga un sobrecoste por las llamadas recursivas que efectúa. Es posible eliminarlo si diseñamos una solución iterativa. Analicemos las dependencias entre valores del almacén R o lo que es lo mismo, las dependencias entre llamadas recursivas de la función C. La figura 8.6 representa qué valores de R son calculados a partir de qué otros valores de R. Cada vértice del grafo de la figura es un **estado** y el grafo inducido por la ecuación recursiva es un **grafo de dependencias entre estados** o, simplemente, **grafo de dependencias**. Cada camino del grafo de dependencias se corresponde con un

> **!** Nótese que, aunque parecido, el grafo de dependencias entre resultados no es idéntico al grafo con el que representamos los embarcaderos y sus conexiones directas. La transformación recursivo-iterativa característica de la programación dinámica pasa por conocer el grafo de dependencias y encontrar un buen recorrido de sus vértices.

trayecto entre los embarcaderos 1 y E en el río, y viceversa.

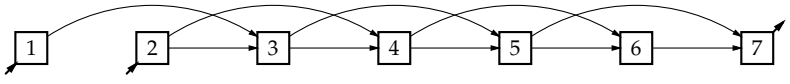


Figura 8.6: Grafo de dependencias entre valores de la tabla de resultados intermedios en el problema del río Congo.

Si efectuamos un recorrido de los vértices del grafo de dependencias en un orden topológico, al calcular cada valor de $R[i]$ tendremos ya calculados los valores que necesitamos, que son $R[i-1]$ y $R[i-2]$. Un recorrido por valor creciente del índice es un orden topológico en el grafo de dependencias:

- Calculamos primero $R[1]$ como 0.
- Calculamos entonces $R[2]$ como $c[1,2]$.
- Calculamos $R[3]$ como el mínimo entre $R[2]$ más $c[2,3]$ y $R[1]$ más $c[1,3]$, pues ya conocemos el valor de $R[2]$ y de $R[1]$.
- Calculamos $R[4]$ como el mínimo entre $R[3]$ más $c[3,4]$ y $R[2]$ más $c[2,4]$, pues ya conocemos el valor de $R[3]$ y de $R[2]$.
- ...

Este programa Python procede de ese modo:

> ! Hemos renombrado el almacén R por C, que es el nombre de la función recursiva.

congo.py *(cont.)*

```
21 def iterative_cheapest_price1(E, c):
22     C = {}
23     C[1] = 0
24     C[2] = c(1,2)
25     for i in xrange(3, E+1):
26         C[i] = min( C[i-1] + c(i-1,i), C[i-2] + c(i-2,i) )
27     return C[E]
```

Una traza

Ilustremos el funcionamiento del algoritmo con una traza. Usaremos los mismos datos de las ejecuciones de pruebas, es decir, resolveremos paso a paso la instancia de la figura 8.3.

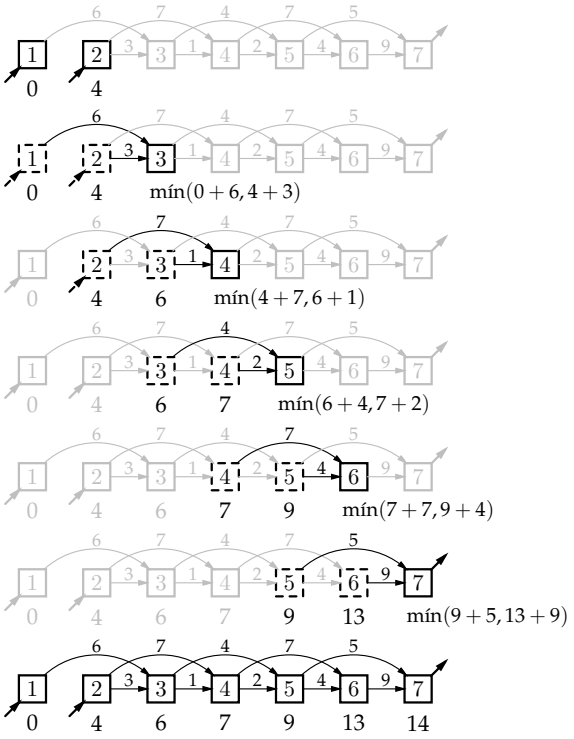


Figura 8.7: Traza del algoritmo iterativo de cálculo del coste del trayecto más barato en la instancia del río Congo de la figura 8.3. Se muestra el grafo de dependencias y, bajo cada estado, el valor asociado. De (a) a (f) se muestra el estado cuyo valor se calcula con trazo negro y continuo y los estados cuyos valores se usan en el cálculo se muestran con trazo negro y discontinuo.

En la figura 8.7 (a) se ilustra el estado de C (valores bajo los vértices) tras la inicialización de sus dos primeros componentes, los asociados a los embarcaderos 1 y 2. La figura 8.7 (b) muestra el cálculo del valor de $C[3]$, que es el menor valor entre $C[1]+c(1,3)$ y $C[2]+c(2,3)$, es decir, el mínimo entre 6 y 7. Se muestran en trazo negro los estados que, en cada instante, participan en el cálculo. La figura 8.7 (c) muestra el instante en el que

se calcula $C[4]$. El estado 4 se representa con un trazo negro continuo y los estados 2 y 3, cuyos valores asociados se consultan durante el cálculo, se muestran con trazo negro discontinuo. El primer estado, cuyo valor asociado ya no es necesario, se muestra en trazo gris. Las figuras 8.7 (d)-(f) muestran cada uno de los cálculos y la figura 8.7 (g) muestra el estado final de C.

Coste computacional

Hemos usado un diccionario para implementar el almacén de resultados intermedios,

> **!** En muchos de los problemas que resolveremos más adelante es posible también sustituir el diccionario C por un vector o matriz para garantizar accesos $O(1)$. Por regla general resulta trivial considerar cuándo es posible hacerlo. No obstante, nosotros no lo haremos cada vez en aras de la brevedad. Dejamos como ejercicio para el lector efectuar este refinamiento de cada programa cuando proceda.

pero un vector es suficiente, ya que los embarcaderos se identifican mediante enteros consecutivos:

congo.py *(cont.)*

```
29 from offsetarray import OffsetArray
30 def iterative_cheapest_price2(E, c):
31     C = OffsetArray([None]*E)
32     C[1] = 0
33     C[2] = c(1,2)
34     for i in xrange(3, E+1):
35         C[i] = min(C[i-1] + c(i-1,i), C[i-2] + c(i-2,i) )
36     return C[E]
```

El vector C contiene, al final de la ejecución de la función, un resultado por cada embarcadero. Es interesante notar que $C[i]$, para $1 \leq i \leq E$, es el coste del trayecto más barato entre el primer embarcadero y el embarcadero i. O sea, al resolver el problema del cálculo de coste del trayecto más barato entre los embarcaderos 1 y E resolvemos el mismo problema entre los embarcaderos 1 e i, para todo i entre 1 y E.

El coste temporal del algoritmo es, evidentemente, $O(E)$. Y el coste espacial de este algoritmo también es $O(E)$: usamos una celda de memoria para cada resultado intermedio (para cada embarcadero). No hemos ganado nada (asintóticamente) con respecto al algoritmo recursivo con memorización, aunque puede notarse cierta aceleración en la práctica al eliminar el sobrecoste propio de las llamadas recursivas.

Reducción de complejidad espacial

Si sólo estamos interesados en el precio del trayecto más barato al viajar entre los embarcaderos 1 y E, podemos ahorrar memoria y reducir el coste espacial a $O(1)$. La idea clave radica en considerar que cuando estamos resolviendo el cálculo del mínimo precio de un trayecto entre los embarcaderos 1 e i, sólo necesitamos conocer el precio del trayecto más barato entre los embarcaderos 1 e $i-1$ por una parte, y 1 e $i-2$ por otra: son los que se

muestran con trazo negro y discontinuo en la figura 8.7. Cualquier otro precio puede ser «olvidado» sin problema alguno: son los que corresponden a estados con trazo gris a la izquierda del estado actual en la figura 8.7.

Esta nuevo algoritmo sólo necesita tres variables para almacenar resultados intermedios, así que presenta coste espacial $O(1)$.

congo.py *(cont.)*

```
38 def iterative_cheapest_price(E, c):
39     C2 = 0
40     C1 = c(1,2)
41     for i in xrange(3, E+1):
42         C3 = min( C1 + c(i-1,i), C2 + c(i-2,i) )
43         C2, C1 = C1, C3
44     return C3
```

EJERCICIOS

2 Reduce el número de variables necesarias para los resultados intermedios a sólo dos.

Es posible (y bastante común) expresar una solución como la anterior con un vector circular de tres componentes:

congo.py *(cont.)*

```
46 def iterative_cheapest_price3(E, c):
47     C = [None, 0, c(1,2)]
48     for i in xrange(3, E+1):
49         C[i%3] = min( C[(i-1)%3] + c(i-1,i), C[(i-2)%3] + c(i-2,i) )
50     return C[E%3]
```

La secuencia de embarcaderos

Hemos resuelto eficientemente el problema de calcular el coste del camino más barato entre los embarcaderos primero y último, pero no sabemos qué secuencia de embarcaderos forma ese trayecto. Podemos aplicar la técnica «estándar» de punteros hacia atrás (véase la sección **??**) para asociar a cada vértice i su vértice predecesor en el camino óptimo del embarcadero 1 al i.

congo.py *(cont.)*

```
52 def cheapest_path(E, c):
53     C = OffsetArray([None] * E)
54     B = OffsetArray([None] * E)
55     C[1] = 0
56     C[2] = c(1,2)
57     B[2] = 1
58     for i in xrange(3, E+1):
59         if C[i-1] + c(i-1,i) < C[i-2] + c(i-2,i):
60             C[i] = C[i-1] + c(i-1,i)
61             B[i] = i-1
62         else:
63             C[i] = C[i-2] + c(i-2,i)
64             B[i] = i-2
```

```
65      path = []
66      i = E
67      while i != None:
68          path.append(i)
69          i = B[i]
70      path.reverse()
71      return path
```

test_cheapest_path.py

```
1  from graph import WeightingFunction
2  from congo import cheapest_path
3
4  c = WeightingFunction({(1,2): 4, (1,3): 6,  (2,3): 3,
5                         (2,4): 7,  (3,4): 1, (3,5): 4,
6                         (4,5): 2, (4,6): 7,  (5,6): 4,
7                         (5,7): 5,  (6,7): 9})
8
9  print 'Camino más barato del embarcadero 1 al 7:', cheapest_path(7, c)
```

```
Camino más barato del embarcadero 1 al 7: [1, 3, 4, 5, 7]
```

En la figura 8.8 mostramos, paso a paso, una traza del algoritmo con recuperación del camino. Las aristas de trazo discontinuo corresponden a los valores almacenados en el vector B. De cada vértice j parte un arco con trazo discontinuo que apunta al vértice i cuyo valor $C[i]$, combinado con el precio del viaje directo (i, j), es más barato. En la figura 8.8 (g) se muestra la secuencia de punteros hacia atrás que se recorre en la última fase del algoritmo y que permite conocer el camino óptimo. Es destacable que el camino óptimo entre 1 y j se pueda descomponer en un camino que también es óptimo al ir del embarcadero 1 a otro i tal que $i < j$ y una arista adicional (i, j). O sea, cualquier prefijo de un camino óptimo es también solución óptima para un subproblema. Decimos que el problema presenta **subestructura óptima** en sus soluciones.

El algoritmo resultante se ejecuta en tiempo $O(E)$. Obsérvese que recuperar el camino ha obligado a usar $O(E)$ celdas de memoria para los punteros hacia atrás. El vector C puede suprimirse y sustituirse por tres variables, pero no podemos suprimir el vector B.

EJERCICIOS

3 Modifica el programa desarrollado para que, en lugar de usar el vector C, use tres variables para resultados intermedios.

Lo cierto es que podemos prescindir de los punteros hacia atrás si usamos el contenido del vector C para recuperar el camino óptimo en una fase de postproceso:

congo.py *(cont.)*

```
73  def cheapest_path2(E, c):
74      C = OffsetArray([None] * E)
75      C[1] = 0
76      C[2] = c[1,2]
77      for i in xrange(3, E+1):
78          C[i] = min(C[i-1] + c[i-1,i], C[i-2] + c[i-2,i] )
```

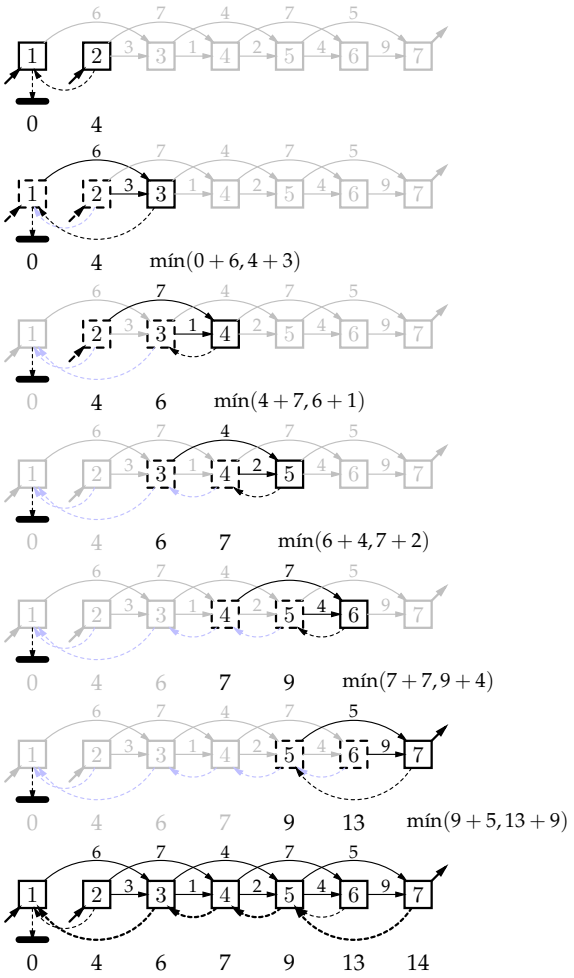


Figura 8.8: Traza del algoritmo iterativo de cálculo del coste del trayecto más barato en la instancia del río Congo de la figura 8.3. En la zona inferior del grafo se muestran los punteros hacia atrás (B) a partir de los cuales se recupera el camino de distancia mínima. En la figura (g) se muestra, en trazo grueso, la serie de punteros hacia atrás que representa al camino más barato.

```
79      path = [E]
80      i = E
81      while i != 1:
82          if C[i] == C[i-1] + c[i-1,i]: i = i - 1
83          else: i = i - 2
84          path.append(i)
85      path.reverse()
86      return path
```

Aunque no necesitamos el vector B, sí necesitamos todos los valores de C, así que la complejidad espacial también es $O(E)$.

8.1.2. El problema del trayecto más probable en el río Congo

Estamos realizando un estudio para el ministerio de turismo y hemos registrado las decisiones que toman los viajeros que recorren el río cuando deciden ir de un embarcadero a uno de sus posibles sucesores. La probabilidad de que un turista que se encuentra en el

embarcadero i-ésimo decida ir directamente al j-ésimo, independientemente de los embarcaderos recorridos hasta el momento, se denota con $p(j|i)$ y es un valor que hemos estimado con las frecuencias observadas. Naturalmente, $p(i+1|i) + p(i+2|i) = 1$ para todo i entre 1 y $E-2$ y $p(E|E-1) = 1$. ¿Qué trayecto sigue, con mayor probabilidad, un turista? ¿Con qué probabilidad?

La figura 8.9 modela el problema con 7 embarcaderos. Sobre cada arco se muestra la probabilidad asociada.

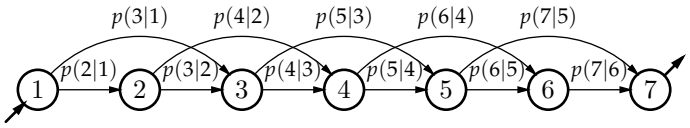


Figura 8.9: Grafo que representa el problema del camino más probable en el río Congo con 7 embarcaderos.

Nuevamente hemos de calcular un camino óptimo, pero no podemos aplicar (al menos no directamente) el algoritmo de Dijkstra. Vamos a reproducir los pasos seguidos en el problema del camino más barato en el río Congo y descubriremos que las técnicas de programación dinámica son aplicables también en este caso.

Formalización

El conjunto de soluciones factibles es el mismo que el problema anterior:

$$X = \{(e_1, e_2, \ldots, e_n) \in [1..E]^+ \mid e_1 = 1; \quad e_n = E; \quad 1 \leq e_i - e_{i-1} \leq 2, \quad 1 < i \leq n\}.$$

La función objetivo es diferente:

$$P((e_1, e_2, \ldots, e_n)) = \prod_{1<i\leq n} p(e_i|e_{i-1}).$$

Deseamos calcular el trayecto más probable:

$$(\hat{e}_1, \hat{e}_2, \ldots, \hat{e}_n) = \arg \max_{(e_1,e_2,\ldots,e_n)\in X} P((e_1, e_2, \ldots, e_n)).$$

Además, deseamos conocer la probabilidad de dicho trayecto:

$$\max_{(e_1,e_2,\ldots,e_n)\in X} P((e_1, e_2, \ldots, e_n)).$$

Cálculo de la mayor probabilidad: ecuación recursiva

Empezamos por calcular la probabilidad del camino más probable:

$$\max_{(e_1,e_2,\ldots,e_n)\in X} \prod_{1<i\leq n} p(e_i|e_{i-1}) = \max_{\substack{(e_1,e_2,\ldots,e_n)|e_1=1;e_n=E;\\ 1\leq e_i-e_{i-1}\leq 2, 1<i\leq n}} \prod_{1<i\leq n} p(e_i|e_{i-1}).$$

El último término del productorio puede salir fuera del mismo:

$$\max_{\substack{(e_1,e_2,\ldots,e_n)|e_1=1;e_n=E;\\ 1\leq e_i-e_{i-1}\leq 2, 1<i\leq n}} \prod_{1<i\leq n} p(e_i|e_{i-1}) = \max_{\substack{(e_1,e_2,\ldots,e_{n-1})|e_1=1;1\leq E-e_{n-1}\leq 2;\\ 1\leq e_i-e_{i-1}\leq 2, 1<i\leq n-1}} \left(\left(\prod_{1<i\leq n-1} p(e_i|e_{i-1}) \right) \cdot p(E|e_{n-1}) \right).$$

La maximización sobre todo posible camino puede descomponerse en dos maximizaciones, una sobre el penúltimo embarcadero del trayecto y otra sobre el resto:

$$\max_{\substack{(e_1,e_2,\ldots,e_n)|e_1=1;e_n=E;\\ 1\le e_i-e_{i-1}\le 2,1<i\le n}} \prod_{1<i\le n} p(e_i|e_{i-1}) = \max_{j\in\{E-1,E-2\}} \left(\max_{\substack{(e_1,e_2,\ldots,e_{n-1})|e_1=1;e_{n-1}=j;\\ 1\le e_i-e_{i-1}\le 2,1<i\le n-1}} \left(\left(\prod_{1<i\le n-1} p(e_i|e_{i-1}) \right) \cdot p(E|j) \right) \right).$$

Hasta aquí, el desarrollo es análogo al que seguimos al resolver el problema anterior. En este punto aplicamos cierta propiedad de la minimización y la suma sobre los reales. ¿Qué propiedad análoga hay para maximización, productos y valores reales entre 0 y 1? Ésta: para todo a, $a' \in [0,1]$ tales que $a \ge a'$ se cumple $a \cdot b \ge a' \cdot b$ para todo $b \in [0,1]$, y, en consecuencia,

$$\max_{a\in[0,1]} (a \cdot b) = \left(\max_{a\in[0,1]} a \right) \cdot b.$$

Podemos seguir:

$$\max_{j\in\{E-1,E-2\}} \left(\max_{\substack{(e_1,e_2,\ldots,e_{n-1})|e_1=1;e_{n-1}=j;\\ 1\le e_i-e_{i-1}\le 2,1<i\le n-1}} \left(\left(\prod_{1<i\le n-1} p(e_i|e_{i-1}) \right) \cdot p(E|j) \right) \right) = \max_{j\in\{E-1,E-2\}} \left(\left(\max_{\substack{(e_1,e_2,\ldots,e_{n-1})|e_1=1;e_{n-1}=j;\\ 1\le e_i-e_{i-1}\le 2,1<i\le n-1}} \prod_{1<i\le n-1} p(e_i|e_{i-1}) \right) \cdot p(E|j) \right).$$

La conclusión es, en este caso, que

$$\max_{\substack{(e_1,e_2,\ldots,e_n)|e_1=1;e_n=E;\\ 1\le e_i-e_{i-1}\le 2,1<i\le n}} \prod_{1<i\le n} p(e_i|e_{i-1}) = \max_{j\in\{E-1,E-2\}} \left(\left(\max_{\substack{(e_1,e_2,\ldots,e_{n-1})|e_1=1;e_{n-1}=j;\\ 1\le e_i-e_{i-1}\le 2,1<i\le n-1}} \prod_{1<i\le n-1} p(e_i|e_{i-1}) \right) \cdot p(E|j) \right).$$

Nuevamente se relaciona un problema referido al último embarcadero con problemas referidos a los dos embarcaderos previos. Si denotamos con $P(i)$ a la probabilidad del camino más probable que parte del primer embarcadero y finaliza en el embarcadero i, podemos plantear una ecuación recursiva análoga a (8.1):

$$P(i) = \begin{cases} 1, & \text{si } i = 1; \\ p(2|1), & \text{si } i = 2; \\ \max(P(i-2)\cdot p(i|i-2), P(i-1)\cdot p(i|i-1)), & \text{si } i > 2. \end{cases} \qquad (8.2)$$

La probabilidad del camino más probable se calcula evaluando $P(E)$.

Algoritmo recursivo

Llegados a este punto no hace falta entrar en detalles: la implementación de una función Python que resuelva la ecuación recursiva es trivial.

congo.py *(cont.)*

```
88 def recursive_maximum_probability(E, p):
89     def P(i):
90         if i == 1:   return 1
91         elif i == 2: return p(2,1)
92         else:        return max(P(i-2) * p(i,i-2),  P(i-1) * p(i,i-1))
93     return P(E)
```

test_congo1.py

```
1 from graph import WeightingFunction
2 from congo import recursive_maximum_probability
3
4 p = WeightingFunction({(2,1): 0.6, (3,1): 0.4, (3,2): 0.7,
5                        (4,2): 0.3,  (4,3): 0.7, (5,3): 0.3,
6                        (5,4): 0.6, (6,4): 0.4, (6,5): 0.6,
7                        (7,5): 0.4,  (7,6): 1.0})
8 print 'Probabilidad del camino más probable entre 1 y 7:', \
9       recursive_maximum_probability(7, p)
```

```
Probabilidad del camino más probable entre 1 y 7: 0.1176
```

Memorización, versión iterativa, reducción de la complejidad espacial y recuperación del camino

Sería ocioso seguir con una exposición tan detallada como la que seguimos al resolver el anterior problema: todos los pasos que podemos seguir son análogos a aquellos.

- La recursión es ineficiente porque efectúa cálculos repetidos. La técnica de memorización de resultados intermedios evita esta repetición de cálculos. El coste temporal y espacial pasa a ser $O(E)$.
- Si estudiamos el grafo de dependencias, que es el mismo de la figura 8.6, comprobaremos que un orden topológico de los estados permite diseñar una solución iterativa. La solución iterativa tiene coste temporal $O(E)$ y espacial $O(E)$.
- Si sólo deseamos conocer el valor de la probabilidad del trayecto más probable, podemos reducir la complejidad espacial a $O(1)$ almacenando los resultados intermedios en sólo tres variables.
- Pero si deseamos conocer también el camino, hemos de usar un almacén con $O(E)$ valores (los punteros hacia atrás), lo que devuelve el coste espacial al de la primera versión iterativa.

Los pasos que hemos considerado suelen seguirse (con alguna excepción) al resolver un problema de programación dinámica. Vale la pena ir familiarizándose son ellos.

EJERCICIOS

4 ¿Puedes deducir una ecuación recursiva similar a las obtenidas, pero para calcular la probabilidad del camino menos probable y el coste del camino más caro? Asegúrate de que todos los pasos son matemáticamente válidos.

8.2. El problema de la mochila discreta

El problema de la mochila discreta se enuncia así:

> Tenemos una mochila con capacidad para cargar W unidades de peso y N objetos que podemos cargar en ella. Cada objeto tiene un peso $w_i \in \mathbb{Z}^{>0}$ y un valor $v_i \in \mathbb{R}$, para $1 \leq i \leq N$. El beneficio que aporta cargar el objeto i es su valor v_i. ¿Qué objetos seleccionamos para cargar la mochila sin exceder su capacidad y de modo que el beneficio sea máximo?

Nótese que es un problema diferente del de la mochila con fraccionamiento (véase la sección **??**) en tanto que no permite considerar la carga de parte de un objeto. La capacidad de elección se limita a decidir si cada uno de los objetos se carga o no en la mochila. La estrategia voraz no funciona en este caso, a diferencia de lo que ocurría con el problema de la mochila con fraccionamiento. Es fácil encontrar un ejemplo que lo demuestre. Consideremos una mochila con capacidad 10 y tres objetos de pesos 6, 5 y 10 y valores respectivos 3, 2 y 4. El primer objeto tiene la relación valor/peso más alta (0.5), pero cargarlo en la mochila impide cargar ningún otro y obtenemos un beneficio total de 3. El último objeto, sin embargo, ofrece una relación valor/peso menor (0.4), pero su carga en la mochila produce un beneficio de 4.

8.2.1. Formalización

Formalicemos el problema en términos de optimización. Las soluciones del problema

> **!** Nuevamente empezamos por identificar el conjunto de soluciones factibles, la función objetivo y determinamos si estamos minimizando o maximizando su valor sobre las soluciones factibles.

pueden describirse con listas de valores que indican qué productos hemos cargado en la mochila.

Una primera posibilidad consiste en describir una selección de objetos con una lista que contenga sus índices. El conjunto de soluciones factibles sería entonces

$$X = \left\{ (x_1, x_2, \ldots, x_n) \in [1..N]^* \;\middle|\; x_i \neq x_j, 1 \leq i \neq j \leq n; \sum_{1 \leq i \leq n} w_{x_i} \leq W \right\}.$$

Es posible desarrollar una solución a partir de este modelado de las soluciones factibles, pero resulta más conveniente utilizar una representación distinta. Una selección de objetos puede modelarse con un vector x cuyas celdas valen 0 o 1: x_i valdrá 1 si decidimos cargar el objeto i en la mochila y 0 en caso contrario. El vector debe satisfacer la restricción

de que la mochila pueda soportar la carga de todos los objetos que representa:

$$X = \left\{ (x_1, x_2, \ldots, x_N) \in \{0,1\}^N \middle| \sum_{1 \leq i \leq N} x_i w_i \leq W \right\}.$$

Buscamos un elemento de X que haga máximo el beneficio. Nuestra función objetivo

> **!** Nótese que las soluciones factibles son secuencias de elementos simples con ciertas restricciones. En este caso, las restricciones afectan a la longitud de la secuencia (hay N elementos) y obligan a que cierto sumatorio sea menor que un valor.

es, pues,

$$f((x_1, x_2, \ldots, x_N)) = \sum_{1 \leq i \leq N} x_i v_i,$$

y deseamos calcular el vector que hace máximo el valor de f:

$$(\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_N) = \arg \max_{(x_1, x_2, \ldots, x_N) \in X} f((x_1, x_2, \ldots, x_N)).$$

8.2.2. Ecuación recursiva

Para abordar el problema por programación dinámica empezamos por plantearnos la resolución del problema del cálculo de mayor beneficio que podemos obtener.

$$\max_{(x_1, x_2, \ldots, x_N) \in X} \sum_{1 \leq i \leq N} x_i v_i = \max_{\substack{(x_1, x_2, \ldots, x_N) \in \{0,1\}^N: \\ \sum_{1 \leq i \leq N} x_i w_i \leq W}} \sum_{1 \leq i \leq N} x_i v_i.$$

El último término del sumatorio puede extraerse del mismo:

$$\max_{\substack{(x_1, x_2, \ldots, x_N) \in \{0,1\}^N: \\ \sum_{1 \leq i \leq N} x_i w_i \leq W}} \sum_{1 \leq i \leq N} x_i v_i = \max_{\substack{(x_1, x_2, \ldots, x_N) \in \{0,1\}^N: \\ \sum_{1 \leq i \leq N} x_i w_i \leq W}} \left(\left(\sum_{1 \leq i \leq N-1} x_i v_i \right) + x_N v_N \right).$$

Buscamos una secuencia de N ceros y unos. El último valor de la secuencia sólo puede ser un cero o un uno, así que conviene que hagamos explícita esa posibilidad: un uno

> **!** Observa que tanto en este caso como en el del río Congo nos planteamos qué posibilidades tenemos al final de la secuencia que forma la solución buscada.

significa que incluimos el objeto N-ésimo, pero eso sólo es posible si su peso es inferior o igual a la capacidad de carga de la mochila; por otra parte, un cero significa que ese

objeto no se carga, cosa que siempre podemos hacer. Así pues, queremos calcular

$$\begin{aligned}\max_{\substack{(x_1,x_2,\ldots,x_N)\in\{0,1\}^N:\\ \sum_{1\le i\le N} x_i w_i\le W}} \sum_{1\le i\le N} x_i v_i &= \begin{cases} \max\left\{ \begin{array}{l} \displaystyle\max_{\substack{(x_1,x_2,\ldots,x_{N-1})\in\{0,1\}^{N-1}:\\ \sum_{1\le i\le N-1} x_i w_i\le W}} \left(\sum_{1\le i\le N-1} x_i v_i\right), \\ \displaystyle\max_{\substack{(x_1,x_2,\ldots,x_{N-1})\in\{0,1\}^{N-1}:\\ \sum_{1\le i\le N-1} x_i w_i\le W-w_N}} \left(\left(\sum_{1\le i\le N-1} x_i v_i\right)+v_N\right) \end{array} \right\}, & \text{si } w_N\le W; \\ \displaystyle\max_{\substack{(x_1,x_2,\ldots,x_{N-1})\in\{0,1\}^{N-1}:\\ \sum_{1\le i\le N-1} x_i w_i\le W}} \left(\sum_{1\le i\le N-1} x_i v_i\right), & \text{si } w_N> W; \end{cases} \\ &= \max_{\substack{x_N\in\{0,1\}:\\ x_N w_N\le W}} \left(\max_{\substack{(x_1,x_2,\ldots,x_{N-1})\in\{0,1\}^{N-1}:\\ \sum_{1\le i\le N-1} x_i w_i\le W-x_N w_N}} \left(\left(\sum_{1\le i\le N-1} x_i v_i\right)+x_N v_N\right)\right).\end{aligned}$$

Para todo a, $a' \in \mathbb{R}$ tales que $a \ge a'$, se cumple que $a+b \ge a'+b$ para todo $b \in \mathbb{R}$. Así pues, para un valor fijo de $b \in \mathbb{R}$, se cumple

> **!** Esta propiedad es análoga a la que usamos en un paso similar al derivar la ecuación recursiva del problema del trayecto más barato en el río Congo: allí minimizábamos y aquí maximizamos.

$$\max_{a\in\mathbb{R}}(a+b) = \left(\max_{a\in\mathbb{R}} a\right)+b.$$

Tenemos, pues,

$$\begin{aligned}\max_{\substack{(x_1,x_2,\ldots,x_N)\in\{0,1\}^N:\\ \sum_{1\le i\le N} x_i w_i\le W}} \sum_{1\le i\le N} x_i v_i &= \max_{\substack{x_N\in\{0,1\}:\\ x_N w_N\le W}} \left(\max_{\substack{(x_1,x_2,\ldots,x_{N-1})\in\{0,1\}^{N-1}:\\ \sum_{1\le i\le N-1} x_i w_i\le W-x_N w_N}} \left(\left(\sum_{1\le i\le N-1} x_i v_i\right)+x_N v_N\right)\right) \\ &= \max_{\substack{x_N\in\{0,1\}:\\ x_N w_N\le W}} \left(\left(\max_{\substack{(x_1,x_2,\ldots,x_{N-1})\in\{0,1\}^{N-1}:\\ \sum_{1\le i\le N-1} x_i w_i\le W-x_N w_N}} \sum_{1\le i\le N-1} x_i v_i\right)+x_N v_N\right).\end{aligned}$$

Podemos advertir que la maximización interior define una instancia del problema de la mochila: cuál es el mayor beneficio que podemos obtener cargando una selección de los $N-1$ primeros objetos en una mochila de capacidad $W - x_N w_N$. Si denominamos $V(i,c)$ al máximo beneficio que podemos obtener con una selección de los primeros i objetos en una mochila con capacidad c, tenemos

$$\boxed{V(N,W)\quad \max_{\substack{(x_1,x_2,\ldots,x_N)\in\{0,1\}^N:\\ \sum_{1\le i\le N} x_i v_i\le W}} \sum_{1\le i\le N} x_i v_i} = \max_{\substack{x_N\in\{0,1\}:\\ x_N w_N\le W}} \left(\left(\boxed{V(N-1,W-x_N w_N)\quad \max_{\substack{(x_1,x_2,\ldots,x_{N-1})\in\{0,1\}^{N-1}:\\ \sum_{1\le i\le N-1} x_i w_i\le W-x_N w_N}} \sum_{1\le i\le N-1} x_i v_i}\right)+x_N v_N\right).$$

Es decir,

$$\begin{aligned} V(N,W) &= \max_{\substack{x_N\in\{0,1\}:\\ x_N w_N \le W}} \Big(V(N-1, W - x_N w_N) + x_N v_N \Big) \\ &= \begin{cases} \max(V(N-1,W), V(N-1,W-w_N)+v_N), & \text{si } w_N \le W; \\ V(N-1,W), & \text{si } w_N > W. \end{cases} \end{aligned}$$

Ya hemos encontrado una recurrencia que expresa la solución de un problema con N objetos y capacidad W en función de otros problemas, uno con $N-1$ objetos y capacidad W y otro con $N-1$ objetos y capacidad $W - x_N w_N$. El razonamiento que hemos seguido con respecto del par (N, W) puede reproducirse para cualquier par (i, c) donde $1 \le i \le N$ y $0 \le c \le W$:

$$\begin{aligned} V(i,c) &= \max_{\substack{x_i\in\{0,1\}:\\ x_i w_i \le c}} \Big(V(i-1, c - x_i w_i) + x_i v_i \Big) \\ &= \begin{cases} \max(V(i-1,c), V(i-1,c-w_i)+v_i), & \text{si } w_i \le c; \\ V(i-1,c), & \text{si } w_i > c. \end{cases} \end{aligned}$$

Identifiquemos casos base. Podemos considerar $V(0, c)$, para todo c entre 0 y W, casos base con beneficio nulo: si no hay objetos que cargar, no hay beneficio posible.

La ecuación recursiva es, pues,

$$V(i,c) = \begin{cases} 0, & \text{si } i = 0; \\ \max(V(i-1,c), V(i-1,c-w_i)+v_i), & \text{si } i > 0 \text{ y } w_i \le c; \\ V(i-1,c), & \text{si } i > 0 \text{ y } w_i > c. \end{cases} \tag{8.3}$$

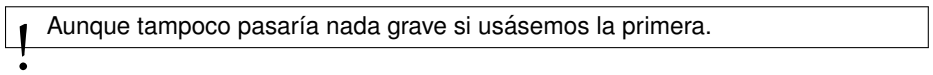
Podemos añadir un caso base adicional: $V(i, 0)$ es también 0 para todo i entre 0 y N, ya que es imposible obtener beneficio con una mochila en la que no se puede cargar nada.

$$V(i,c) = \begin{cases} 0, & \text{si } i = 0 \text{ o } c = 0; \\ \max(V(i-1,c), V(i-1,c-w_i)+v_i), & \text{si } i > 0,\ c > 0 \text{ y } w_i \le c; \\ V(i-1,c), & \text{si } i > 0,\ c > 0 \text{ y } w_i > c. \end{cases} \tag{8.4}$$

¿Qué ecuación recursiva es «la correcta»? Nótese que si consideramos $V(i, 0)$, con $i > 0$, en la ecuación (8.3), tenemos $V(i, 0) = V(i-1, 0)$, pues $w_i > 0$ por definición. Y podemos aplicar el mismo razonamiento reiteradamente:

$$V(i,0) = V(i-1,0) = V(i-2,0) = \cdots = V(0,0) = 0.$$

La ecuación recursiva (8.4) presenta una (ligera) ventaja frente a la (8.3), pues proporciona el valor de $V(i, 0)$ directamente, sin necesidad de definición recursiva alguna. Usaremos, por tanto, la ecuación (8.4).



8.2.3. Algoritmo recursivo

Un resolución recursiva es sencilla de implementar en Python:

knapsack.py

```
def recursive_knapsack_profit(W, v, w):
    def V(i, c):
        if i==0 or c == 0: return 0
        elif c-w[i] >= 0: return max(V(i-1, c), V(i-1, c-w[i]) + v[i])
        else:              return V(i-1, c)
    return V(len(v), W)
```

test_knapsack1.py

```
from knapsack import recursive_knapsack_profit
from offsetarray import OffsetArray

W = 6
v = OffsetArray([90, 75, 60, 20, 10])
w = OffsetArray([ 4,  3,  3,  2,  2])
print "Beneficio máximo al seleccionar objetos de valores %s y pesos %s para" % (v, w),
print "cargar en una mochila con capacidad %d: %d." % (W, recursive_knapsack_profit(W,v,w))
```

```
Beneficio máximo al seleccionar objetos de valores [90, 75, 60, 20, 10] y pesos
[4, 3, 3, 2, 2] para cargar en una mochila con capacidad 6: 135.
```

Este método resolutivo resulta muy ineficiente. Podemos visualizar la razón de la ineficiencia si recurrimos al árbol de llamadas a función que mostramos en la figura 8.10 para los datos de prueba del algoritmo. Se puede apreciar en él que se efectúan numerosas llamadas con los mismos parámetros: hay dos llamadas de la forma $V(3, 4)$, dos llamadas de la forma $V(2, 4)$, dos de la forma $V(2, 1)$, dos de la forma $V(1, 4)$, dos de la forma $V(1, 3)$, cuatro de la forma $V(1, 1)$, dos de la forma $V(0, 4)$, dos de la forma $V(0, 3)$, dos de la forma $V(0, 2)$, cuatro de la forma $V(0,1)$ y dos de la forma $V(0, 0)$.

8.2.4. Versión con memorización

La ineficiencia derivada de la repetición de llamadas puede evitarse si aplicamos la técnica de la memorización de resultados intermedios:

knapsack.py *(cont.)*

```
def memoized_knapsack_profit(W, v, w):
    R = {}
    def V(i, c):
        if i==0 or c == 0:
            R[i, c] = 0
        elif c-w[i] >= 0:
            if (i-1, c) not in R: V(i-1, c)
```

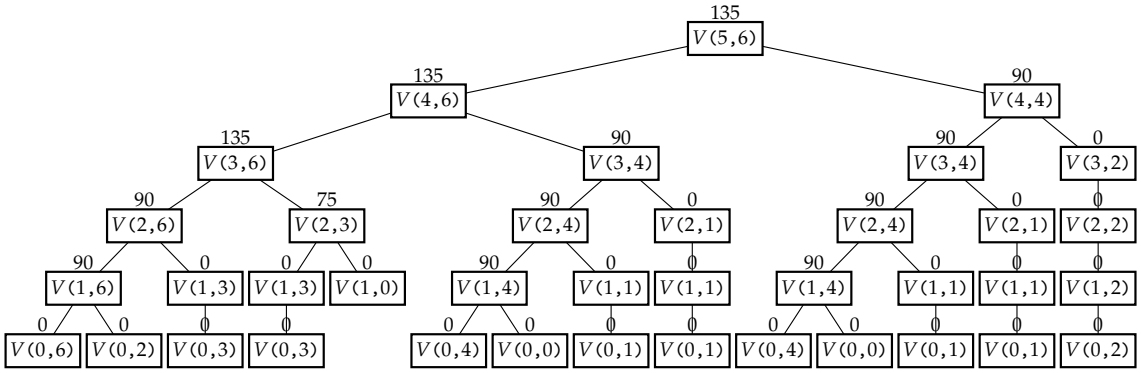


Figura 8.10: Árbol de llamadas para el cálculo recursivo de V (len (v) , W) en el ejemplo, con una mochila de capacidad 6 y objetos de valor $v_1 = 90$, $v_2 = 75$, $v_3 = 60$, $v_4 = 20$ y $v_5 = 10$ y pesos $w_1 = 4$, $w_2 = 3$, $w_3 = 3$, $w_4 = 2$ y $w_5 = 2$.

```
15            if (i-1, c-w[i]) not in R: V(i-1, c-w[i])
16            R[i, c] = max(R[i-1,c], R[i-1, c-w[i]] + v[i])
17        else:
18            if (i-1, c) not in R: R[i-1, c] = V(i-1, c)
19            R[i, c] = R[i-1, c]
20        return R[i, c]
21
22    return V(len(v), W)
```

La figura 8.11 muestra el árbol de llamadas recursivas en un cálculo con memorización equivalente al del árbol mostrado en la figura 8.10. Se puede apreciar un descenso notable en el número de llamadas: de 38 a 23.

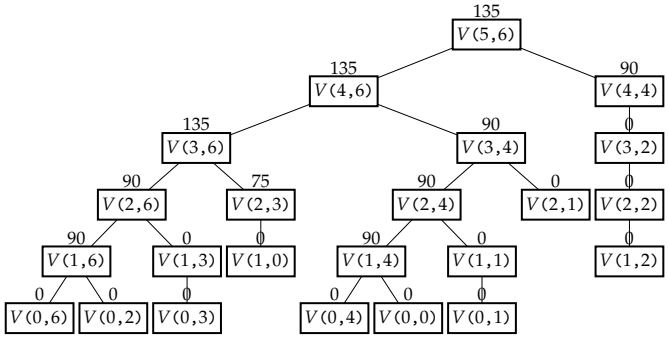


Figura 8.11: Árbol de llamadas para el cálculo recursivo con memorización del problema de la mochila con capacidad 6 y objetos de valor $v_1 = 90$, $v_2 = 75$, $v_3 = 60$, $v_4 = 20$ y $v_5 = 10$ y pesos $w_1 = 4$, $w_2 = 3$, $w_3 = 3$, $w_4 = 2$ y $w_5 = 2$.

8.2.5. Solución iterativa

Recordemos (del río Congo) que cada configuración de parámetros en las llamadas recursivas recibe el nombre de «estado». En este problema, un estado es un par de la forma (i, c) y corresponde a una instancia del problema con i objetos seleccionables y una mochila de capacidad c. Para transformar un algoritmo recursivo en iterativo, hemos de

fijarnos en el grafo que inducen los estados y las relaciones entre ellos que resultan de la ecuación recursiva. En la figura 8.12 se muestra el denominado grafo de dependencias para el cálculo recursivo cuyo árbol de llamadas se representó en la figura 8.10.

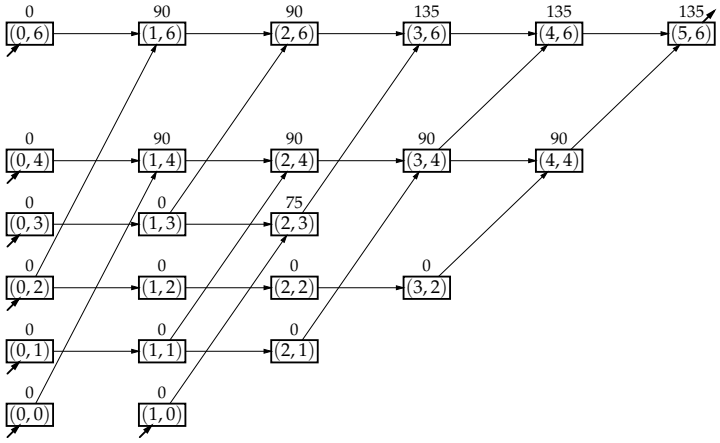


Figura 8.12: Grafo de dependencias de la instancia del problema de la mochila con capacidad 6 y objetos de valores $v_1 = 90$, $v_2 = 75$, $v_3 = 60$, $v_4 = 20$ y $v_5 = 10$ y pesos $w_1 = 4$, $w_2 = 3$, $w_3 = 3$, $w_4 = 2$ y $w_5 = 2$. Los estados iniciales y el estado final se muestra con flechas de entrada y salida, respectivamente.

En el grafo de dependencias, *cada camino entre un estado inicial y un estado final describe una solución factible y cada solución factible se representa por un camino del grafo*. Tomemos por caso el camino $((0,0),(1,4),(2,4),(3,4),(4,4),(5,6))$: describe la solución factible que carga en la mochila el primer objeto (peso 4 y valor 90) y el último objeto (peso 2 y valor 10). Cada columna de arcos se asocia a un objeto que podemos cargar o no en la mochila. Cargarlo o no hace que pasemos de un estado a otro. Los arcos horizontales suponen la no inclusión del objeto asociado a la columna de arcos correspondiente, mientras que los arcos no horizontales indican que el objeto correspondiente sí se incluye en la mochila (véase la figura 8.13). Los arcos están ponderados por el valor de los objetos. Buscamos un camino de dicho grafo: el que parte de un estado inicial, llega al estado final y hace máxima la suma de pesos de sus arcos.

Nuevamente hemos planteado un problema de cálculo del camino óptimo (en esta caso, el de máximo peso). Y nuevamente lo resolveremos con un método más eficiente que mediante una modificación del algoritmo de Dijkstra. Vamos a recorrer los estados para «resolverlos»: a cada estado (i,c) le asociaremos el valor de $V(i,\ c)$ (en la figura 8.12 se muestran los valores de V sobre sus correspondientes estados). Con este objetivo, hemos de encontrar un buen «orden de recorrido» de estados (i,c), de modo que al tratar de calcular $V(i,\ c)$ ya se hayan calculado los valores de los que depende, esto es, $V(i\text{-}1,\ c)$ y, posiblemente, $V(i\text{-}1,\ c\text{-}w[i])$. Hay, al menos, dos órdenes de recorrido válidos:

- por columnas, de izquierda a derecha y sin que importe el orden dentro de la

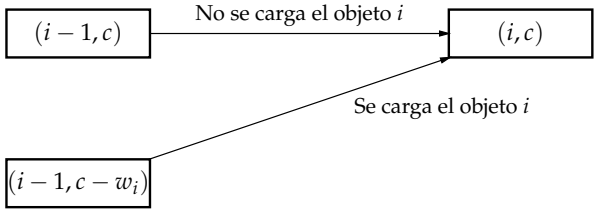


Figura 8.13: Interpretación de las aristas del grafo de dependencias en términos de la inclusión de objetos en la mochila.

> ! Ambos recorridos son órdenes topológicos del grafo de dependencias.

columna (véase la figura 8.14 (a)),

- y por filas, de abajo arriba y recorriendo los estados de cada fila de izquierda a derecha (véase la figura 8.14 (b)).

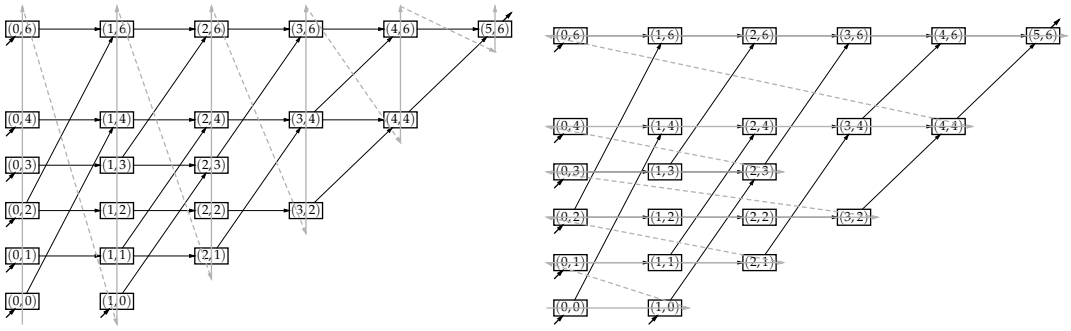


Figura 8.14: Dos órdenes de recorrido válidos (órdenes topológicos) para el grafo de dependencias de la figura 8.12: (a) por columnas y (b) por filas.

Hay un problema a la hora de diseñar un programa que efectúe uno de los dos recorridos: los estados que hemos de visitar se deducen a partir del estado final, cuando es este estado el último que hemos de visitar. Actuar en dos fases, una que deduzca el conjunto de estados que hemos de visitar y otra que los visite no resulta interesante, pues no aporta ventaja alguna sobre la técnica de recursión con memorización y sí un mayor sobrecoste.

Una solución comúnmente adoptada consiste en recorrer más estados que los estrictamente necesarios. Para entender esto, fijémonos en el grafo de la figura 8.15. En ese grafo hay estados que no aparecen en los anteriores. No obstante, comparte una propiedad con el anterior grafo de dependencias: *cada camino entre un estado inicial y un estado final describe una solución factible.*

El nuevo grafo presenta una gran ventaja: tanto el recorrido por filas como el recorrido por columnas se pueden implementar con un par de bucles **for**. Esta función iterativa, por ejemplo, resuelve el problema recorriendo el grafo por columnas:

knapsack.py *(cont.)*

```
24 def iterative_knapsack_profit1(W, v, w):
25     V = {}
26     for c in xrange(W+1):
27         V[0,c] = 0
28     for i in xrange(1, len(v)+1):
29         V[i,0] = 0
30         for c in xrange(1, w[i]):
31             V[i,c] = V[i-1,c]
32         for c in xrange(w[i], W+1):
```

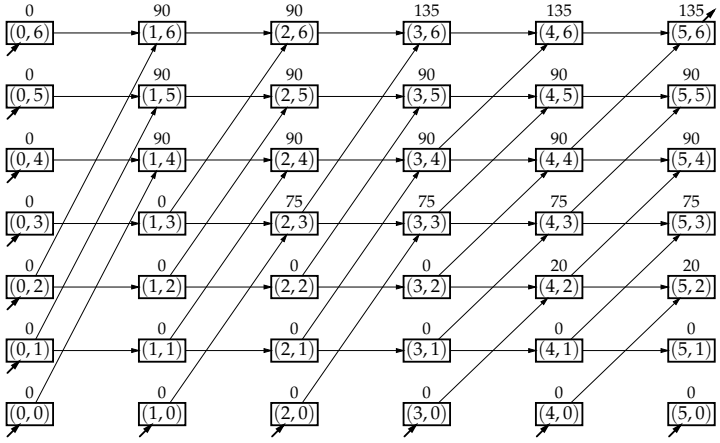


Figura 8.15: Grafo de dependencias extendido a partir del que se muestra en la figura 8.12. En el grafo se consideran estados innecesarios, pero que simplifican la escritura de un algoritmo de recorrido de los estados en orden topológico.

```
33            V[i,c] = max(V[i-1,c], V[i-1,c-w[i]] + v[i])
34    return V[len(v), W]
```

La complejidad temporal del algoritmo es $\Theta(NW)$. La complejidad espacial es, también, $\Theta(NW)$.

EJERCICIOS

5 Diseña una solución iterativa basada en un recorrido de los estados por filas.

8.2.6. Reducción de la complejidad espacial

Si sólo estamos interesados en conocer el beneficio máximo, podemos reducir la complejidad espacial. En el recorrido por columnas, cuando estamos visitando los estados de la forma (i, c) sólo hacemos referencia a estados de la forma $(i-1, c)$, es decir, al procesar la columna i necesitamos conocer valores propios de la columna $i-1$, pero no de otras columnas anteriores. Podemos limitar el almacén de resultados a sólo dos columnas: la «actual» y la «anterior».

knapsack.py *(cont.)*

```
36 def iterative_knapsack_profit2(W, v, w):
37     V = [[0] * (W+1), [0] * (W+1)]
38     curr, prev = 1, 0
39     for i in xrange(1, len(v)+1):
40         V[curr][0] = 0
41         for c in xrange(1, w[i]):
42             V[curr][c] = V[prev][c]
43         for c in xrange(w[i], W+1):
44             V[curr][c] = max(V[prev][c], V[prev][c-w[i]] + v[i])
45         curr, prev = prev, curr
46     return V[prev][W]
```

La complejidad espacial de este algoritmo es $O(W)$, en lugar de $O(NW)$.

Aún podemos efectuar una reducción de memoria más agresiva limitando el número de columnas a una sola: basta con recorrer los estados de cada columna «de abajo a arriba».

knapsack.py *(cont.)*

```
48 def iterative_knapsack_profit(W, v, w):
49     V = [0] * (W+1)
50     for i in xrange(1, len(v)+1):
51         V[0] = 0
52         for c in xrange(W, w[i]-1, -1):
53             V[c] = max(V[c], V[c-w[i]] + v[i])
54         for c in xrange(w[i]-1, 0, -1):
55             V[c] = V[c]
56     return V[W]
```

Aunque reducimos así el consumo de memoria a la mitad, no obtenemos una reducción del coste espacial asintótico con respecto a la versión anterior. Por ello, y para facilitar la legibilidad de los algoritmos, normalmente no apuraremos hasta este extremo las reducciones de complejidad espacial.

EJERCICIOS

6 ¿Es posible reducir la complejidad espacial si se efectúa un recorrido por filas?

8.2.7. Obtención de la carga óptima de la mochila

La secuencia de estados que conduce al máximo beneficio es fácil de obtener con la técnica de punteros hacia atrás sobre el grafo de dependencias. Podemos conocer la carga óptima interpretando adecuadamente la secuencia de estados. Recordemos que, como se muestra en la figura 8.13,

- las aristas horizontales, de la forma $((i-1,c),(i,c))$, indican que el objeto i no se carga en la mochila;
- y las aristas no horizontales, de la forma $((i-1,c-w_i),(i,c))$, indican que el objeto i sí se carga en la mochila.

knapsack.py *(cont.)*

```
58 from offsetarray import OffsetArray
59
60 def knapsack(W, v, w):
61     V, B = {}, {}
62     for c in xrange(W+1):
63         V[0,c], B[0,c] = 0, None
64     for i in xrange(1, len(v)+1):
65         V[i,0], B[i,0] = 0, (i-1, 0)
66         for c in xrange(1, w[i]):
67             V[i,c], B[i,c] = V[i-1,c], (i-1, c)
68         for c in xrange(w[i], W+1):
69             if V[i-1,c] > V[i-1,c-w[i]] + v[i]:
70                 V[i,c], B[i,c] = V[i-1,c], (i-1, c)
71             else:
72                 V[i,c], B[i,c] = V[i-1,c]+v[i], (i-1, c-w[i])
73     path = OffsetArray([])
74     (i, c) = (len(v), W)
75     while B[i,c] != None:
```

```
76          if B[i,c] != (i-1, c): path.append(1)
77          else: path.append(0)
78          (i, c) = B[i,c]
79      path.reverse()
80      return path
```

test_knapsack2.py

```
1  from knapsack import knapsack
2  from offsetarray import OffsetArray
3
4  W = 6
5  v = OffsetArray([90, 75, 60, 20, 10])
6  w = OffsetArray([ 4,  3,  3,  2,  2])
7  print "Selección de objetos de valores %s y pesos %s para obtener máximo" % (v, w),
8  print "beneficio al cargar en una mochila con capacidad %d: %s." % (W, knapsack(W,v,w))
```

```
Selección de objetos de valores [90, 75, 60, 20, 10] y pesos [4, 3, 3, 2, 2] par
a obtener máximo beneficio al cargar en una mochila con capacidad 6: [0, 1, 1, 0
, 0].
```

En la figura 8.16 se muestran los punteros hacia atrás sobre el grafo de dependencias de la instancia del problema de la mochila resuelto en el programa.

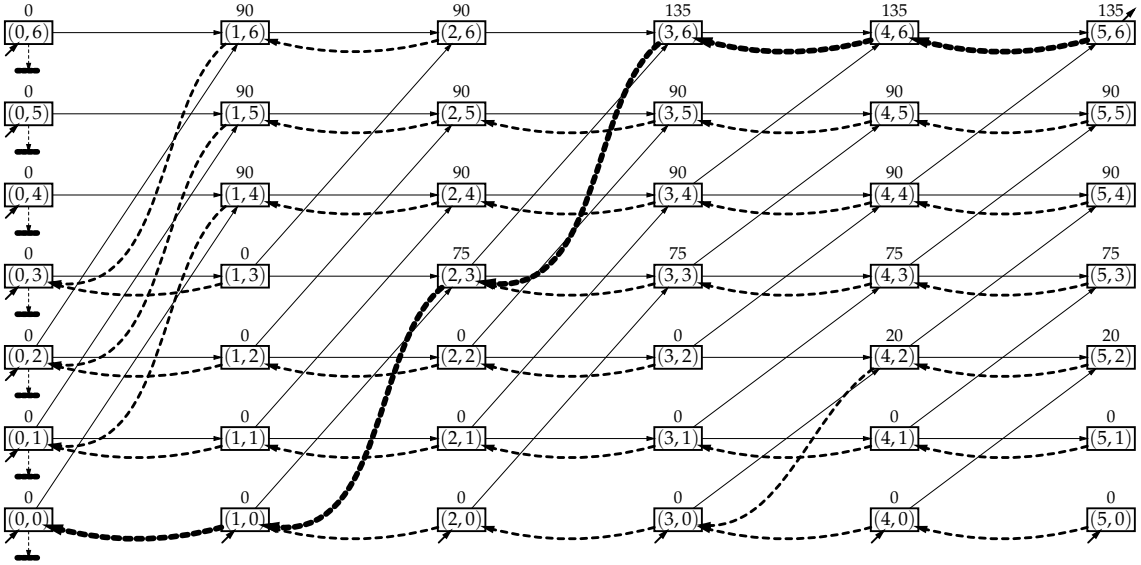


Figura 8.16: En trazo discontinuo, punteros hacia atrás en el grafo de dependencias de la figura 8.15. Los punteros hacia atrás de trazo grueso son los que corresponden a la secuencia óptima de decisiones.

El almacén de punteros hacia atrás ocupa $\Theta(NW)$ celdas, así que no es posible efectuar una reducción de complejidad espacial del estilo de la obtenida cuando no interesaba conocer la selección de objetos óptima. Podríamos, eso sí, reducir el tamaño de almacén

V, que pasaría de $\Theta(NW)$ a $\Theta(W)$, pero sin afectar al coste espacial asintótico del algoritmo.

EJERCICIOS

7 Diseña una solución para el problema de la mochila discreta cuando deseamos obtener el mayor beneficio posible (sin exceder la capacidad de carga) cargando exactamente M objetos, con $M \leq N$.

8 Diseña una solución para el problema de la mochila discreta cuando deseamos obtener el mayor beneficio posible (sin exceder la capacidad de carga) y disponemos de una cantidad ilimitada de objetos de valor v_i y peso w_i, para i entre 1 y N. ¿Funcionaría en este caso una aproximación voraz?

9 Deseamos conocer, si lo hay, el subconjunto de objetos cuyos pesos sumen exactamente W y nos proporcionen beneficio máximo. Diseña un algoritmo que resuelva este problema y analiza sus costes temporal y espacial.

10 Diseña un algoritmo que permita recuperar la secuencia de decisiones óptima sin almacenar punteros hacia atrás.

8.3. Esquema algorítmico de la programación dinámica

Los ejemplos desarrollados son ilustrativos del proceso que se sigue frecuentemente al solucionar un problema de programación dinámica:

1. **Formalización.** Planteamiento del problema en términos formales de optimización sobre un espacio de soluciones factibles, que son secuencias de elementos de cierto conjunto.
2. **Ecuación recursiva.** Derivación de una ecuación recursiva que soluciona el problema de obtención del valor óptimo de la función objetivo.
3. **Algoritmo recursivo.** Diseño de un algoritmo que soluciona recursivamente el problema.
4. **Memorización.** Detección de problemas de repetición de cálculos en el algoritmo recursivo y resolución mediante memorización de resultados en una tabla indexada por argumentos de la función recursiva (estados).
5. **Transformación recursivo-iterativa.** Obtención de un grafo de dependencias entre estados. Diseño de un algoritmo iterativo a partir de un recorrido de los vértices del grafo en orden topológico y el almacenamiento de resultados asociados a los vértices en una tabla. Si el grafo de dependencias está estructurado, puede que no sea necesario mantener en memoria la tabla completa de resultados, con el consiguiente ahorro de espacio.
6. **Cálculo de la solución factible óptima.** La solución factible óptima está compuesta por una serie de estados o decisiones simples. Los estados son los vértices de un camino en el grafo de dependencias (y las decisiones son sus aristas). La técnica de punteros hacia atrás o un proceso sobre el almacén de resultados permite recuperar los elementos que constituyen la solución óptima.

Detallemos cada uno de estos pasos y tratemos de ofrecer un marco general con el que facilitar el tratamiento de futuros problemas.

8.3.1. Formalización

Conjunto de estados y conjunto de decisiones

El esquema de programación dinámica es de aplicación en problemas de optimización cuyas soluciones factibles son secuencias de elementos simples. Consideremos en primer lugar el conjunto de soluciones factibles en los problemas planteados sobre el río Congo, donde las soluciones factibles eran secuencias de embarcaderos que satisfacían ciertas restricciones:

$$X = \{(e_1, e_2, \ldots, e_n) \in [1..E]^+ \mid e_1 = 1; \quad e_n = E; \quad 1 \leq e_i - e_{i-1} \leq 2, \quad 1 < i \leq n\}.$$

Estos elementos simples, cuya concatenación forma soluciones factibles, reciben el nombre de **estados**. Denotaremos con S al **conjunto de estados** de un problema. Parece claro que $X \subseteq S^*$. La recurrencia que resolvía el problema del cálculo del camino de

> **!** S^* es la clausura de S bajo una operación de concatenación, así que denota al conjunto de cadenas (secuencias) formadas con elementos de S.

menor coste (o el del camino más probable) tenía estados como argumentos.

Las soluciones factibles del problema de la mochila también eran secuencias de elementos simples (ceros y unos):

$$X = \left\{ (x_1, x_2, \ldots, x_N) \in \{0,1\}^N \;\middle|\; \sum_{1 \leq i \leq N} x_i w_i \leq W \right\}.$$

Recordemos que la recurrencia que permitía calcular la carga más beneficiosa se formulaba sobre pares de la forma (i, c), donde $0 \leq i \leq N$ y $0 \leq c \leq W$. Dichos pares son los estados, así que una solución factible no se expresa, en este caso, como una secuencia de estados. Pero bien podríamos haber definido el conjunto de soluciones factibles así:

$$X = \left\{ ((0, c_0), (1, c_1), (2, c_2), \ldots, (N, c_N)) \in ([0..N] \times [0..W])^{N+1} \;\middle|\; c_i - c_{i-1} \in \{0, w_i\}, 0 < i \leq N;\ c_N = W;\ c_0 \geq 0 \right\}.$$

Cada solución factible del primer tipo (secuencia de ceros y unos) se corresponde con una del segundo tipo (secuencia de estados). Es posible derivar la ecuación recursiva (8.4) a partir de la nueva formulación del conjunto de soluciones.

En el problema de la mochila discreta, el modelado original de las soluciones factibles describe una solución factible como una *secuencia de transiciones entre estados* o **secuencia de decisiones** (por ejemplo, 0 significa la decisión de no incluir un objeto y 1 la decisión de incluirlo). Notaremos con D al conjunto de decisiones. Las soluciones factibles son, en el problema de la mochila discreta, un subconjunto de D^*.

En los problemas que abordaremos, las soluciones factibles pueden modelarse como **secuencias de estados** o como **secuencias de decisiones**. En los problemas del río Congo, por ejemplo, podemos plantear las soluciones factibles como secuencias de decisiones así:

$$X = \left\{ (d_1, d_2, \ldots, d_n) \in \{1,2\}^+ \;\middle|\; \sum_{1 \le i \le n} d_i = E - 1 \right\}.$$

En ciertos problemas resultará más natural un modelado con secuencias de estados y en otros con secuencias de decisiones. Cuando modelemos las soluciones factibles con secuencias de decisiones, X será un subconjunto de D^* y cuando lo hagamos con secuencias de estados, X será un subconjunto de S^*. En ciertos problemas resultará más natural un modelado que el otro, pero ambos son equivalentes desde el punto de vista de las soluciones algorítmicas alcanzables.

En el desarrollo teórico que presentamos a continuación supondremos que las soluciones son secuencias que alternan estados y decisiones, es decir, una solución factible será una secuencia de la forma $(s_0, d_1, s_1, \ldots, d_n, s_n)$, donde $s_i \in S$, para $0 \le i \le n$, y $d_j \in D$, para $1 \le j \le n$. Aunque el desarrollo resulta un tanto más farragoso, presenta la ventaja de ser fácilmente adaptable a los dos tipos de modelado: basta con proyectar las soluciones sobre S^* o D^*, según corresponda.

Función objetivo

La función objetivo f proporciona un valor real para cada secuencia de estados o decisiones. Buscamos

a) la solución factible que proporciona el valor óptimo de la función objetivo (y en el caso de que dos o más soluciones factibles proporcionen el valor óptimo de la función objetivo, una cualquiera de ellas):

$$(\hat{s}_0, \hat{d}_1, \hat{s}_1, \ldots, \hat{d}_n, \hat{s}_n) = \arg \operatorname*{opt}_{(s_0, d_1, s_1, \ldots, d_n, s_n) \in X} f((s_0, d_1, s_1, \ldots, d_n, s_n)),$$

b) y/o dicho valor:

$$\operatorname*{opt}_{(s_0, d_1, s_1, \ldots, d_n, s_n) \in X} f((s_0, d_1, s_1, \ldots, d_n, s_n)).$$

Aunque nuestro propósito también sea resolver el primer problema, conviene empezar diseñando una ecuación recursiva para el segundo. Resolver esa ecuación recursiva proporciona, mediante técnicas estándar, la solución del primero.

En el problema del trayecto más barato en el río Congo, la función objetivo, que notamos con C, era la suma de precios de trayectos sencillos:

$$C((e_1, e_2, \ldots, e_n)) = \sum_{1 < i \le n} c(e_{i-1}, e_i).$$

En el problema del trayecto más probable, la función objetivo, P, era un producto de probabilidades

$$P((e_1, e_2, \ldots, e_n)) = \prod_{1 < i \le n} p(e_i | e_{i-1}).$$

Y en el problema de la mochila, la función objetivo era una suma de beneficios:

$$f((x_1, x_2, \ldots, x_N)) = \sum_{1 \leq i \leq N} x_i v_i.$$

Función de descomposición

Las secuencias válidas se pueden definir a partir de una **función de descomposición de estados** $\delta : S \to \mathcal{P}(S \times D)$:

$$X = \{(s_0, d_1, s_1, \ldots, d_n, s_n) \mid (s_{i-1}, d_i) \in \delta(s_i), \quad 1 \leq i \leq n\}.$$

La ecuación recursiva se plantea el cálculo del valor óptimo considerando que la solución óptima está formada por un prefijo $(\hat{s}_0, \hat{d}_1, \hat{s}_1, \ldots, \hat{d}_{n-1}, \hat{s}_{n-1})$ y un último par $(\hat{d}_n, \hat{s}_n)$. El conjunto de prefijos y pares finales posibles se obtiene con δ.

En los dos problemas planteados sobre el río Congo la función de descomposición es

$$\delta(e) = \{(e-1, 1), (e-2, 2)\}.$$

Si sólo consideramos la secuencia de estados, sin las decisiones asociadas, la función es

$$\delta(e) = \{e-1, e-2\}.$$

En el problema de la mochila, la función de descomposición es

$$\delta(i, c) = \begin{cases} \{(i-1, c)\}, & \text{si } w_i > c; \\ \{(i-1, c), (i-1, c-w_i)\}, & \text{si } w_i \leq c. \end{cases}$$

Estados finales e iniciales

Los problemas que hemos estudiado presentan un único estado fijo al final de cualquier solución factible. En el río Congo, por ejemplo, el último estado era E, el último embarcadero; en el problema de la mochila discreta, el último estado era (N, W), que corresponde a seleccionar objetos de entre N disponibles y no exceder una capacidad de carga W. En algunos problemas, sin embargo, no todas las soluciones factibles finalizan en un estado fijo.

Sea $\mathcal{F}$ el conjunto de valores que puede adoptar s_n, el último estado de una secuencia de X, es decir,

$$\mathcal{F} = \{s_n \mid (s_0, d_1, s_1, \ldots, d_n, s_n) \in X\}.$$

Si denotamos con $X(s)$, para cualquier elemento $s \in S$, al conjunto

$$X(s) = \{(s_0, d_1, s_1, \ldots, d_j, s_j) \mid s_j = s; \quad (s_{i-1}, d_i) \in \delta(s_i), \quad 1 \leq i \leq n\},$$

tenemos

$$X = \bigcup_{s \in \mathcal{F}} X(s).$$

El conjunto $\mathcal{F}$ es el **conjunto de estados finales**.

Si ahora denotamos con $\mathcal{I}$ al conjunto de valores posibles para s_0 en cualquier secuencia factible, podemos plantear la siguiente ecuación recursiva:

$$X(s) = \begin{cases} \{(s)\}, & \text{si } s \in \mathcal{I}; \\ \{(s_0, d_1, s_1, \ldots, d_j, s_j) \mid s_j = s; \quad (s_0, d_1, s_1, \ldots, d_{j-1}, s_{j-1}) \in X(s'), \quad (s', d_j) \in \delta(s_j)\}, & \text{si } s \notin \mathcal{I}. \end{cases}$$

El conjunto $\mathcal{I}$ es el **conjunto de estados iniciales** y determina los casos base en la definición recursiva del conjunto X.

En el río Congo consideramos que $\mathcal{I} = \{1, 2\}$ aunque buscásemos un trayecto que parte del primer embarcadero. El conjunto de estados finales es $\mathcal{F} = \{E\}$.

En el problema de la mochila con N objetos y capacidad W, el conjunto de estados iniciales es $\mathcal{I} = \{(0, c) : 0 \leq c \leq W\} \cup \{(i, 0) : 0 \leq i \leq N\}$. El conjunto de estados finales es $\mathcal{F} = \{(N, W)\}$.

Subestructura óptima: el principio de optimalidad

Hemos de tener en cuenta que no toda función objetivo es susceptible de conducir a una ecuación recursiva, así que hemos de tener claro bajo qué condiciones es posible abordar el problema con programación dinámica.

> **!** El principio de optimalidad fue enunciado así por Richard Bellman: «An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.» Es fácil de entender si recurrimos a un ejemplo sobre un mapa de carreteras. Si el camino más corto de València a Tarragona pasa por Sagunto y Castelló de la Plana, entonces el camino más corto de València a Castelló de la Plana pasa también por Sagunto. De hecho, el camino más corto de València a Castelló de la Plana es un prefijo del camino más corto de València a Tarragona.

Una clave es la observación de la denominada **subestructura óptima** o **principio de optimalidad**. Este principio puede formularse así: «en la solución óptima del problema, $(\hat{s}_0, \hat{d}_1, \hat{s}_1, \ldots, \hat{d}_n, \hat{s}_n)$, las secuencias de la forma $(\hat{s}_0, \hat{d}_1, \hat{s}_1, \ldots, \hat{d}_j, \hat{s}_j)$ para $j < n$ son soluciones óptimas de problemas de la misma naturaleza, pero diferente talla», o de forma más sencilla, «los prefijos de una solución óptima son, a su vez, óptimos».

Principio de optimalidad: Optimality Principle.

Tanto en los dos problemas planteados sobre el río Congo como en el de la mochila discreta se observa la existencia de subestructura óptima.

Una visión global

Hemos formulado el marco general de los problemas de programación dinámica como una generalización del problema del camino óptimo entre dos conjuntos de vértices de un (multi)grafo (el grafo de dependencias):

- Los estados son vértices y las decisiones son aristas;
- las soluciones factibles son caminos en el grafo;

- la función de descomposición, δ, proporciona una descripción del conjunto de aristas que conectan un estado con sus estados predecesores;
- los conjuntos de estados iniciales indican puntos iniciales válidos para los caminos y los estados finales, puntos de llegada válidos;
- y la función objetivo asigna un peso a cada camino.

La programación dinámica propone la resolución eficiente de una ecuación recursiva sobre los estados del grafo. Para que el método de programación dinámica sea aplicable, la función objetivo deberá satisfacer ciertas propiedades que presentamos a continuación.

8.3.2. Ecuación recursiva

Separabilidad de la función objetivo

La función objetivo, f, debe ser **separable**: si $(s_0, d_1, s_1, \ldots, d_n, s_n)$ es una solución factible, la función objetivo debe poder expresarse como

$$f((s_0, d_1, s_1, \ldots, d_n, s_n)) = \begin{cases} h(s_0), & \text{si } n = 0; \\ f((s_0, d_1, s_1, \ldots, d_{n-1}, s_{n-1})) \oplus g(d_n, s_n), & \text{si } n > 0; \end{cases}$$

donde g y h son funciones auxiliares, $g : D \times S \to \mathbb{R}$ y $h : S \to \mathbb{R}$, y $\oplus$ es un operador $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$ al que denominamos **operador de combinación de resultados**.

En el problema del trayecto más barato sobre el río Congo, el operador de combinación de resultados es la suma (el precio de un trayecto es el precio que cuesta llegar al penúltimo embarcadero más el del ir directamente de éste al último) y el del trayecto más probable, el operador de combinación de resultados es el producto (por un razonamiento análogo). En el problema del trayecto más barato, la función de inicialización era

$$h(e) = \begin{cases} 0, & \text{si } e = 1; \\ c(1,2), & \text{si } e = 2; \end{cases}$$

y en el del trayecto más probable,

$$h(e) = \begin{cases} 1, & \text{si } e = 1; \\ p(2|1), & \text{si } e = 2. \end{cases}$$

En el problema de la mochila discreta, el operador de combinación de resultados es nuevamente la suma. La función de inicialización era $h(i, c) = 0$ para todo estado (i, c) inicial.

Monotonía del operador de combinación de resultados

Para seguir con la derivación de la ecuación recursiva es preciso que el operador de combinación de resultados satisfaga una determinada condición de monotonía. Para expresarla, necesitamos distinguir problemas de minimización de problemas de maximización:

- Si el problema es de **minimización**, el operador $\oplus$ debe ser **monótono no decreciente por la izquierda**:

$$a \leq a' \quad \Rightarrow \quad a \oplus b \leq a' \oplus b,$$

para todo $a, a', b \in \mathbb{R}$. Con ello tenemos

$$\min_{a \in \mathbb{R}} (a \oplus b) = \left(\min_{a \in \mathbb{R}} a \right) \oplus b.$$

- Si el problema es de **maximización**, el operador $\oplus$ debe ser **monótono no creciente por la izquierda**:

$$a \geq a' \quad \Rightarrow \quad a \oplus b \geq a' \oplus b,$$

para todo $a, a', b \in \mathbb{R}$. Con ello tenemos

$$\max_{a \in \mathbb{R}} (a \oplus b) = \left(\max_{a \in \mathbb{R}} a \right) \oplus b.$$

Podemos volver al planteamiento general y afirmar que si el operador de combinación de resultados observa la condición de monotonía apropiada, entonces

$$\operatorname*{opt}_{(s_{n-1},d_n) \in \delta(s_n)} \left(\operatorname*{opt}_{(s_0,d_1,s_1,\ldots,d_{n-1},s_{n-1}) \in X(s_{n-1})} \Big(f((s_0,d_1,s_1,\ldots,d_{n-1},s_{n-1})) \oplus g(d_n,s_n) \Big) \right) =$$
$$\operatorname*{opt}_{(s_{n-1},d_n) \in \delta(s_n)} \left(\left(\operatorname*{opt}_{(s_0,d_1,s_1,\ldots,d_{n-1},s_{n-1}) \in X(s_{n-1})} f((s_0,d_1,s_1,\ldots,d_{n-1},s_{n-1})) \right) \oplus g(d_n,s_n) \right).$$

Notemos con $F(s)$ a $\operatorname{opt}_{(s_0,d_1,s_1,\ldots,d_n,s_n) \in X(s)} f((s_0,d_1,s_1,\ldots,d_n,s_n))$. Tenemos

$$F(s) = \begin{cases} h(s), & \text{si } s \in \mathcal{I}; \\ \operatorname*{opt}_{(s',d) \in \delta(s)} \Big(F(s') \oplus g(d,s) \Big), & \text{si } s \notin \mathcal{I}; \end{cases} \tag{8.5}$$

y buscamos

$$\operatorname*{opt}_{(s_0,d_1,s_1,\ldots,d_n,s_n) \in X} f((s_0,d_1,s_1,\ldots,d_n,s_n)) = \operatorname*{opt}_{s \in \mathcal{F}} F(s). \tag{8.6}$$

La suma y el producto son operadores de combinación de resultados que aparecen en numerosos problemas de interés y ambos satisfacen la condición de monotonía para la minimización y la maximización.

Esquema recursivo

El esquema algorítmico de resolución de la ecuación recursiva se muestra en la figura 8.17. Es una simple transcripción de las dos últimas ecuaciones.

scheme RecursiveDynamicProgramming($\mathcal{I}$: $\mathcal{P}(S)$, $\mathcal{F}$: $\mathcal{P}(S)$, δ: $S \to \mathcal{P}(S \times D)$, g: $D \times S \to \mathbb{R}$, h: $S \to \mathbb{R}$)
function F(s: S)
    if $s \in \mathcal{I}$:
        return h(s)
    else:
        return $\text{opt}_{(s',d)\in\delta(s)} \left(F(s') \oplus g(d,s)\right)$
    return $\text{opt}_{s\in\mathcal{F}} F(s)$

Figura 8.17: Esquema recursivo de programación dinámica.

8.3.3. Memorización de resultados intermedios

Una resolución directa de la ecuación recursiva suele plantear problemas serios de ineficiencia debidos a la repetición de llamadas recursivas a la función F con los mismos argumentos. La técnica de memorización evita la repetición de cálculos: el almacenamiento de los resultados conocidos en una tabla indexada por los argumentos de la función en cada llamada. El esquema recursivo con memorización se muestra en la figura 8.18.

scheme MemoizedDynamicProgramming($\mathcal{I}$: $\mathcal{P}(S)$, $\mathcal{F}$: $\mathcal{P}(S)$, δ: $S \to \mathcal{P}(S \times D)$, g: $D \times S \to \mathbb{R}$, h: $S \to \mathbb{R}$)
var
    R: table of $\mathbb{R}$ indexed by S
function F(s: S)
    if $s \in \mathcal{I}$:
        R[s] $\leftarrow$ h(s)
    else:
        for $(s',d) \in \delta(s)$:
            if $s' \notin R$: $F(s')$
        R[s] $\leftarrow \text{opt}_{(s',d)\in\delta(s)} \left(R[s'] \oplus g(d,s)\right)$
    return R[s]
return $\text{opt}_{s\in\mathcal{F}} F(s)$

Figura 8.18: Esquema recursivo de programación dinámica con memorización.

El coste temporal de los algoritmos que se derivan de este esquema es proporcional al número de aristas del grafo de dependencias. El coste espacial es proporcional al número de vértices.

8.3.4. Transformación recursivo-iterativa

La ecuación recursiva induce cierto grafo al que denominamos grafo de dependencias. Hay una correspondencia entre soluciones factibles y caminos del grafo que parten de vértices iniciales, $\mathcal{I}$, y llegan a vértices finales, $\mathcal{F}$. En el problema del río Congo, por

ejemplo, cada solución era un camino en el grafo de dependencias entre los estados 1 o 2 y el estado E. Dicho camino se expresaba como una secuencia de estados.

El grafo de dependencias debe ser acíclico para permitir un recorrido de sus vértices en orden topológico. Resolver $F(s)$ para los diferentes valores de s en el orden topológico conduce a una transformación recursivo-iterativa de la ecuación. Una versión iterativa resulta de interés en tanto que elimina el sobrecoste de las llamadas recursivas. El coste temporal de la ordenación topológica es proporcional a la suma del número de vértices y aristas del grafo de dependencias. Así pues, el coste temporal de los algoritmos que se derivan de este esquema también es proporcional a la suma del número de aristas y vértices del grafo de dependencias. El coste espacial es proporcional al número de vértices.

scheme DynamicProgramming($\mathcal{I}$: $\mathcal{P}(S)$, $\mathcal{F}$: $\mathcal{P}(S)$, δ: $S \to \mathcal{P}(S \times D)$, g: $D \times S \to \mathbb{R}$, h: $S \to \mathbb{R}$)
var
    R: table of $\mathbb{R}$ indexed by D
for s in topsort(S, δ):
        if s in $\mathcal{I}$:
            R[s] $\leftarrow$ h(s)
    else:
        R[s] $\leftarrow \text{opt}_{(s',d)\in\delta(s)} \left(R[s'] \oplus g(d,s) \right)$
return $\text{opt}_{s\in\mathcal{F}} R[s]$

Figura 8.19: Esquema iterativo de programación dinámica.

8.3.5. Reducción del coste espacial

Si el grafo de dependencias está estructurado, puede resultar factible reducir la complejidad espacial del algoritmo. Si el grafo es multietapa, por ejemplo, basta con almacenar los resultados asociados a dos etapas: la actual y la previa (como en el problema de la mochila). Grafos con una estructura muy simple (como el del río Congo) pueden conducir a algoritmos con un consumo de memoria $O(1)$.

8.3.6. Cálculo de la solución factible óptima

Calcular el valor óptimo de la función objetivo puede resultar meramente instrumental para alcanzar nuestro verdadero propósito: calcular la secuencia que hace óptimo el valor de la función objetivo, y que es

$$\begin{aligned}(\hat{s}_0, \hat{d}_1, \hat{s}_1, \ldots, \hat{d}_n, \hat{s}_n) &= \arg \operatorname*{opt}_{(s_0,d_1,s_1,\ldots,d_n,s_n)\in X} f((s_0, d_1, s_1, \ldots, d_n, s_n)) \\ &= \arg \operatorname*{opt}_{s\in\mathcal{F}} F(s).\end{aligned}$$

Podemos conocer la secuencia óptima de estados calculando secuencialmente estos valores:

$$\begin{aligned}
\hat{s}_n &= \arg \operatorname*{opt}_{s \in \mathcal{F}} F(s), \\
(\hat{s}_{n-1}, \hat{d}_n) &= \arg \operatorname*{opt}_{(s',d) \in \delta(\hat{s}_n)} \left(F(s') \oplus g(d, \hat{s}_n) \right), \\
(\hat{s}_{n-2}, \hat{d}_{n-1}) &= \arg \operatorname*{opt}_{(s',d) \in \delta(\hat{s}_{n-1})} \left(F(s') \oplus g(d, \hat{s}_{n-1}) \right), \\
&\vdots \\
(\hat{s}_0, \hat{d}_1) &= \arg \operatorname*{opt}_{(s',d) \in \delta(\hat{s}_1)} \left(F(s') \oplus g(d, \hat{s}_1) \right).
\end{aligned}$$

Las técnicas de recuperación del camino basadas en el almacenamiento de punteros atrás hacen sencilla la recuperación del camino en el grafo de dependencias. La figura 8.20 muestra un esquema que aplica esta técnica.

scheme BacktracingDynamicProgramming($\mathcal{I}$: $\mathcal{P}(S)$, $\mathcal{F}$: $\mathcal{P}(S)$, δ: $S \to \mathcal{P}(S \times D)$, g: $D \times S \to \mathbb{R}$, h: $S \to \mathbb{R}$)
var
    R: table of $\mathbb{R}$ indexed by S
    B: table of D indexed by S
    P: array of $S \cup D$
    for s in topsort(S, δ):
        if s in $\mathcal{I}$:
            R[s] $\leftarrow$ h(s)
            B[s] $\leftarrow$ None
        else:
            R[s] $\leftarrow \operatorname{opt}_{(s',d) \in \delta(s)} \left(R[s'] \oplus g(d, s) \right)$
            B[s] $\leftarrow \arg \operatorname{opt}_{(s',d) \in \delta(s)} \left(R[s'] \oplus g(d, s) \right)$
    $s \leftarrow \arg \operatorname{opt}_{s \in \mathcal{F}} R[s]$
    $P \leftarrow [s]$
    while B[s] $\neq$ None:
        $P \leftarrow B[s]$
        P. append(s)
    P. reverse()
    return P

Figura 8.20: Esquema iterativo de programación dinámica con recuperación de la solución óptima mediante punteros hacia atrás.

Por regla general, el espacio que requiere esta otra versión para almacenar los punteros hacia atrás elimina la eventual reducción de complejidad espacial alcanzada en el algoritmo anterior, es decir, requiere memoria orden del número de vértices del grafo de dependencias.

EJERCICIOS

11 Resuelve el problema del cálculo del trayecto más barato en un río en el que es posible ir directamente de un embarcadero a cualquier otro siguiendo la dirección de la corriente. Plantea una ecuación recursiva para el cálculo del precio del trayecto más barato y diseña dos algoritmos recursivos para su resolución: uno sin y otro con memorización de resultados intermedios. Dibuja el grafo de dependencias y diseña un algoritmo iterativo para el cálculo del precio del trayecto más barato. Analiza la complejidad computacional y, si es posible, reduce la complejidad espacial del algoritmo iterativo. Diseña, finalmente, un algoritmo que permita conocer el trayecto más barato.

8.4. El problema del camino más corto en un grafo acíclico

Dado un grafo acíclico ponderado $G = (V, E, d)$ y dos vértices s y t, a los que denominamos vértice inicial o fuente y vértice final u objetivo, respectivamente, encuéntrese un camino que parta de s y finalice en t tal que ningún otro camino entre ambos vértices presenta menor distancia.

Formalmente, buscamos un camino $(v_1, v_2, \ldots, v_n)$ tal que $v_1 = s$ y $v_n = t$ con distancia mínima.

$$(\hat{v}_1, \hat{v}_2, \ldots, \hat{v}_n) = \arg \min_{(v_1, v_2, \ldots, v_n) \in P(t)} D(v_1, v_2, \ldots, v_n).$$

donde $P(t)$ es el conjunto de caminos entre s y t y se define así:

$$P(t) = \{(v_1, v_2, \ldots, v_n) \mid v_1 = s; \quad v_n = t; \quad (v_i, v_{i+1}) \in E, 1 \leq i < n\},$$

y $D(v_1, v_2, \ldots, v_n)$ es la función de ponderación del camino:

$$D(v_1, v_2, \ldots, v_n) = \sum_{0 < i \leq n} d(v_{i-1}, v_i).$$

Por otra parte, deseamos conocer también la distancia recorrida por el camino más corto:

$$\min_{(v_1, v_2, \ldots, v_n) \in P(t)} D(v_1, v_2, \ldots, v_n).$$

8.4.1. Ecuación recursiva

Como es habitual, empezamos por resolver el segundo problema: el del cálculo de la distancia mínima. Abusando de la notación, denotemos con $D(t)$ la distancia del camino más corto entre s y t:

$$D(t) = \min_{(v_1, v_2, \ldots, v_n) \in P(t)} D(v_1, v_2, \ldots, v_n) = \min_{(v_1, v_2, \ldots, v_n) \in P(t)} \left(\sum_{1 \leq i < n} d(v_i, v_{i+1}) \right).$$

El conjunto de caminos $P(t)$ puede definirse recursivamente a partir de $P(u)$, para todo u predecesor de t:

$$P(t) = \{(v_1, v_2, \ldots, v_n) \mid v_1 = s; \quad v_n = t; \quad (v_1, v_2, \ldots, v_{n-1}) \in P(u), \quad (u, t) \in E\}.$$

El caso base de la recursión es

$$P(s) = \{(s)\}.$$

Podemos sustituir $P(t)$ por su expresión recursiva y considerar en primer lugar la minimización que afecta a la última arista que puede formar parte de un camino que finaliza en t:

$$D(t) = \min_{(u,t)\in E} \left(\min_{(v_1,v_2,\ldots,v_{n-1})\in P(u)} \left(\left(\sum_{1\leq i<n-1} d(v_i, v_{i+1}) \right) + d(u,t) \right) \right).$$

Y, ahora, extraer de la minimización interior el último término del sumatorio, pues la suma es un operador monótono no decreciente por la izquierda y estamos minimizando:

$$D(t) = \min_{(u,t)\in E} \left(\left(\min_{(v_1,v_2,\ldots,v_{n-1})\in P(u)} \left(\sum_{1\leq i<n-1} d(v_i, v_{i+1}) \right) \right) + d(u,t) \right).$$

Podemos reescribir el sumatorio interior, que no es más que la distancia de un camino:

$$D(t) = \min_{(u,t)\in E} \left(\left(\min_{(v_1,v_2,\ldots,v_{n-1})\in P(u)} D(v_1, v_2, \ldots, v_{n-1}) \right) + d(u,t) \right).$$

Es evidente que hemos llegado a una relación recursiva:

$$D(t) = \min_{(u,t)\in E} \left(D(u) + d(u,t) \right).$$

No hay nada especial en t, así que podríamos haber deducido la recursión para un vértice v cualquiera:

$$D(v) = \min_{(u,v)\in E} \left(D(u) + d(u,v) \right).$$

Hay un caso que podemos considerar base de la recursión: cuando $v = s$ se propone la búsqueda de la distancia óptima de s a s. Dicha distancia es, por definición, nula:

$$D(s) = 0.$$

Otro caso especial es el de los vértices sin predecesores. La minimización se propone entonces sobre un conjunto vacío de elementos. Definimos $\min(\emptyset)$ como $+\infty$.

$$D(v) = \begin{cases} 0, & \text{si } v = s; \\ +\infty, & \text{si } v \neq s \text{ y } \nexists (u,v) \in E; \\ \min_{(u,v)\in E} \left(D(u) + d(u,v) \right), & \text{en otro caso.} \end{cases} \tag{8.7}$$

El valor que deseamos calcular es $D(t)$.

8.4.2. Un algoritmo recursivo

Podemos «traducir» la ecuación recursiva a un programa Python y obtener así la implementación de un algoritmo recursivo.

dag_shortest_path.py

```
1  def recursive_dag_shortest_distance(G, d, s, v, infinity= 3.4e+38):
2      def D(v):
3      if v == s:                    return 0
4      elif G.in_degree(v) == 0:     return infinity
5      else:                         return min(D(u) + d(u,v) for u in G.preds(v))
6      return D(v)
```

El coste temporal de este algoritmo es, salvo en grafos acíclicos particulares, exponencial con el número de vértices. El coste espacial es $O(|V|)$.

8.4.3. Un algoritmo con memorización

Podemos evitar la repetición de cálculos almacenando en un diccionario el resultado asociado a cada estado y no efectuando una llamada recursiva sobre los estados cuyo valor asociado se conoce:

dag_shortest_path.py

```
1  def memoized_dag_shortest_distance(G, d, s, v, infinity= 3.4e+38):
2      R = {}
3      def D(v):
4          if v == s:
5              R[v] = 0
6          elif G.in_degree(v) == 0:
7              R[v] = infinity
8          else:
9              for u in G.preds(v):
10                 if u not in R: D(u)
11             R[v] = min(R[u] + d(u,v) for u in G.preds(v))
12         return R[v]
13
14     return D(t)
```

El coste temporal de este algoritmo es $O(|V| + |E|)$ y el coste espacial es $O(|V|)$.

8.4.4. Un algoritmo iterativo

Si al calcular $D(v)$ se produce una «llamada» recursiva a $D(u)$, es porque (u, v) es una arista. O sea, u es un vértice anterior a v en cualquier orden topológico de los vértices del grafo (recordemos que es un grafo acíclico). Podemos diseñar una versión iterativa basada en un recorrido de los vértices del grafo en orden topológico:

dag_shortest_path.py

```
1  from topsort import topsort
2
3  def dag_shortest_distance(G, d, s, t, infinity=3.4e+38):
4      D = {}
5      for v in topsort(G):
6          if v == s:                    D[v] = 0
7          elif G.in_degree(v) == 0:     D[v] = infinity
8          else:                         D[v] = min(D[u] + d(u,v) for u in G.preds(v))
9          if v == t: break
10     return D[t]
```

Probemos el programa. Diseñemos un programa que calcule el camino más corto entre los vértices 0 y 5 del grafo acíclico y ponderado de la figura 8.21.

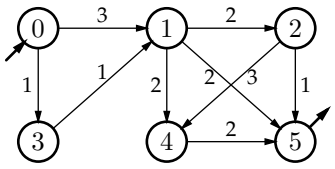


Figura 8.21: Grafo acíclico y ponderado con un vértice inicial y un vértice final.

test_shortest_path_a.py

```
1  from graph import Graph, WeightingFunction
2  from dag_shortest_path import dag_shortest_distance
3
4  G = Graph(V=range(6), E=[(0,1), (0,3), (1,2), (1,4), (1,5), (2,4), (2,5), (3,1), (4,5)])
5  d = WeightingFunction({(0,1):3, (0,3):1, (1,2):2, (1,4):2, (1,5):2,
6                         (2,4): 3, (2,5):1, (3,1):1, (4,5):2})
7  print 'Distancia entre los vértices 0 y 5:', dag_shortest_distance(G, d, 0, 5)
```

```
Distancia entre los vértices 0 y 5: 4
```

Hagamos una traza paso a paso. La función empieza ordenando topológicamente los vértices (figura 8.22 (a)). Se procede entonces a visitar los vértices en ese orden. Se empieza por visitar el vértice 0. Como es el vértice inicial, el valor de $D[0]$ es 0. Mostramos el valor de cada celda de D bajo el correspondiente vértice (figura 8.22 (b)). Pasamos ahora al vértice 3. Sólo hay un predecesor (el vértice 0). El valor de $D[0]$ es 0 y el peso de la arista $(0,1)$ es 1. El valor de $D[3]$ es el mínimo de un conjunto formado por un solo elemento de valor 1, o sea, es 1 (figura 8.22 (c)). Pasamos al vértice 1. Se puede llegar a este vértice desde dos vértices: el vértice 0 y el vértice 3. Si venimos del primero, hemos de considerar el valor $D[0]$ sumado al valor $d(0,1)$. Si venimos del segundo, hemos de considerar el valor $D[3]$ sumado al valor $d(3,1)$. La primera suma da 3 y la segunda, 2. El menor de ambos valores es 2, así que lo asignamos a $D[1]$ (figura 8.22 (d)). Pasamos al vértice 2. Sólo se puede llegar desde el vértice 1. Asignamos a $D[2]$ el valor que resulta de sumar $D[1]$ a $d(1,2)$ (figura 8.22 (e)). Pasamos al vértice 4. Calculamos el valor de $D[4]$ como el menor entre $D[1]$ más $d(1,4)$ y $D[2]$ más $d(2,4)$ (figura 8.22 (f)). Finalmente, calculamos $D[5]$. En este caso tenemos tres predecesores (figura 8.22 (g)). El resultado se muestra en la figura 8.22 (h).

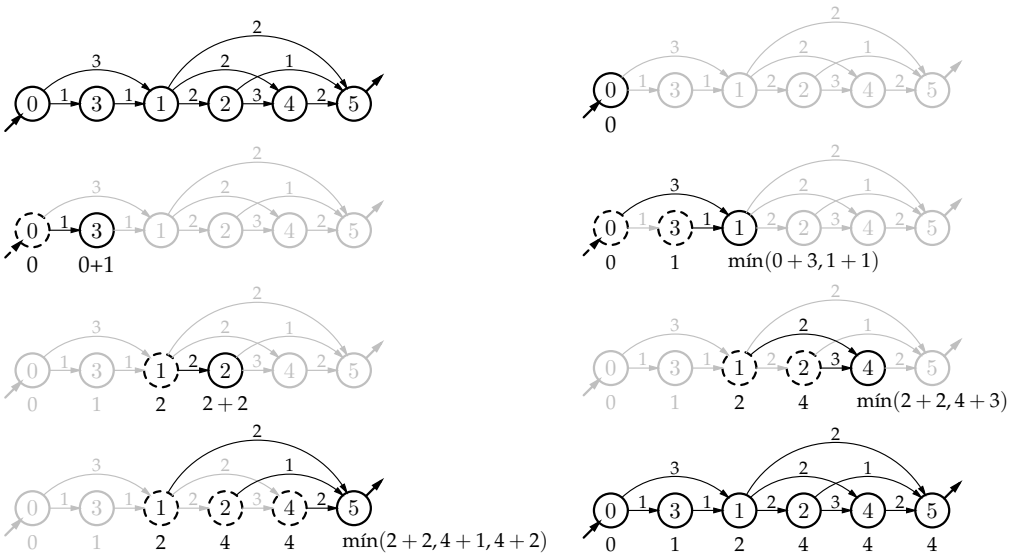


Figura 8.22: Traza del algoritmo iterativo de cálculo del coste del camino más corto en el grafo acíclico ponderado de la figura 8.21.

El coste temporal del algoritmo es $O(|V| + |E|)$ y el espacial, $O(|V|)$.

8.4.5. Reducción de la complejidad espacial

No es posible reducir la complejidad espacial en el caso de un grafo acíclico general. Ciertos grafos estructurados (como el del río Congo o los grafos multietapa) permiten reducir la complejidad espacial.

8.4.6. Cálculo del camino más corto

Los punteros hacia atrás permiten recuperar el camino óptimo:

dag_shortest_path.py

```
1  from topsort import topsort
2
3  def dag_shortest_path(G, d, s, t, infinity=3.4e+38):
4      D = {}
5      B = {}
6      for v in topsort(G):
7          if v == s:
8              D[v] = 0
9              B[v] = None
10         elif G.in_degree(v) == 0:
11             D[v] = infinity
12             B[v] = None
13         else:
```

```
            D[v] = infinity
            for u in G.preds(v):
                if D[u] + d(u,v) < D[v]:
                    D[v] = D[u] + d(u,v)
                    B[v] = u
        if v == t: break
    path = [t]
    while B[path[-1]] != None:
        path.append( B[path[-1]] )
    path.reverse()
    return D[t], path
```

Hagamos una traza paso a paso centrándonos en B, como ilustra la figura 8.23. Tras ordenar topológicamente los vértices del grafo, empezamos con el vértice 0. Como es el vértice inicial, su puntero atrás, $B[s]$ se inicializa a *None* (figura 8.23 (a)). Sólo hay un predecesor del vértice 3, el vértice 0, así que $B[3]$ apunta a él (figura 8.23 (b)). Pasamos al vértice 1. Se puede llegar a este vértice desde dos vértices. El camino óptimo viene del vértice 3, así que $B[1]$ apunta al vértice 3 (figura 8.23 (c)). Pasamos al vértice 2. Sólo se puede llegar desde el vértice 1 (figura 8.23 (d)). Pasamos al vértice 4. El camino óptimo viene del vértice 1 (figura 8.23 (e)). Ahora calculamos $D[5]$ y $B[5]$. Este último apunta al vértice 1 (figura 8.23 (f)). El camino óptimo del vértice 0 al vértice 5 se destaca ahora en trazo más grueso (figura 8.23 (h)). Recorriendo hacia atrás los punteros de B desde el vértice 5 y hasta llegar al valor *None* vamos formando el camino de distancia mínima. El recorrido obtiene el camino en orden inverso: `[5, 1, 3,`

```
0]
```

`0]`. Es necesario, pues, invertir el resultado para obtener el camino óptimo: `[0, 3, 1, 5]`.

Podemos codificar el procedimiento anterior de forma más compacta:

dag_shortest_path.py

```
from topsort import topsort

def dag_shortest_path(G, d, s, t, infinity=3.4e+38):
    # Devuelve la distancia del camino mínimo y la lista de vértices que lo forman.
    D = {}
    B = {}
    for v in topsort(G):
        if v == s:
            D[v], B[v] = 0, None
        elif G.in_degree(v) == 0:
            D[v], B[v] = infinity, None
        else:
            D[v], B[v] = min( (D[u] + d(u,v), u) for u in G.preds(v) )
        if v == t: break
    path = [t]
    while B[path[-1]] != None:
        path.append( B[path[-1]] )
    path.reverse()
    return D[t], path
```

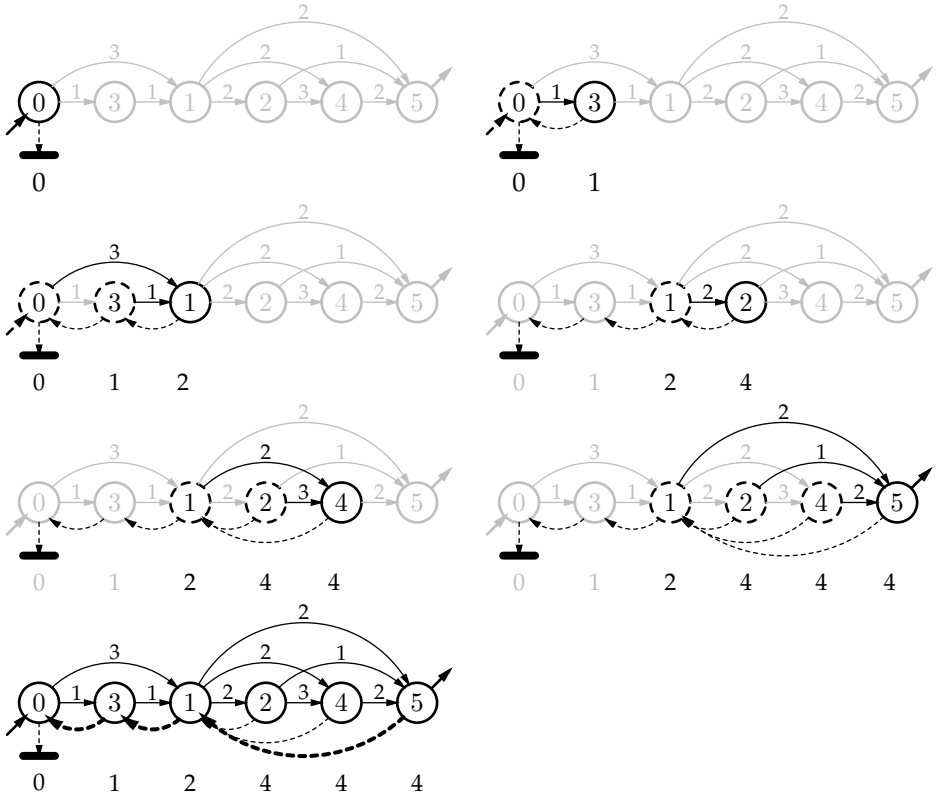


Figura 8.23: Traza del cálculo del camino más corto en el grafo acíclico de la figura 8.21. En la zona inferior del grafo se muestran los punteros hacia atrás (B) a partir de los cuales se recupera el camino de distancia mínima.

Hemos introducido una mejora: en una misma línea calculamos el valor de $D[v]$ y el de $B[v]$. El caso más curioso es la expresión

$$D[v],B[v] = min(\ (D[u] + d(u,v),\ u)\ \textbf{for}\ u\ \textbf{in}\ G.preds(v)\ ).$$

Estúdiala con detenimiento para entender cómo funciona.

Pongamos a prueba nuestro programa:

test_shortest_path_b.py

```
1 from graph import Graph, WeightingFunction
2 from dag_shortest_path import dag_shortest_path
3
4 G = Graph(V=range(6), E=[(0,1), (0,3), (1,2), (1,4), (1,5), (2,4), (2,5), (3,1), (4,5)])
5 d = WeightingFunction({(0,1):3, (0,3):1, (1,2):2, (1,4):2, (1,5):2,
6                        (2,4): 3, (2,5):1, (3,1):1, (4,5):2})
7 distance, path = dag_shortest_path(G, d, 0, 5)
8 print 'Distancia entre los vértices 0 y 5:', distance
9 print 'Camino de distancia mínima entre los vértices 0 y 5:', path
```

```
Distancia entre los vértices 0 y 5: 4
Camino de distancia mínima entre los vértices 0 y 5: [0, 3, 1, 5]
```

Análisis de complejidad computacional

El coste temporal de cualquiera de las versiones del método presentado (recursiva con memorización o iterativas) es $O(|V| + |E|)$ si utilizamos cualquier implementación de los grafos excepto la matriz de adyacencia. En tal caso, el coste temporal se elevaría a $\Theta(|V|^2)$.

La ordenación topológica de los vértices requiere tiempo $O(|V| + |E|)$. Una vez ordenados los vértices, se recorren uno a uno y para cada uno se consulta una vez el valor de cada uno de los arcos que inciden en él. El coste total es, pues, $O(|V| + |E|)$. La recuperación del camino no afecta, en principio, al coste temporal: recorrer los punteros hacia atrás, formar la lista con los vértices atravesados e invertirla al final requiere tiempo que podemos acotar superiormente con una función $O(|V|)$.

Si el grafo fuese ponderado positivo, también podríamos resolver el problema del camino más corto con el algoritmo de Dijkstra, pero su coste temporal, que es $O(|E| + |V| \lg |V|)$, resulta superior al de este método.

8.5. El problema del camino más corto formado por k aristas

Dado un grafo (no necesariamente acíclico) ponderado $G = (V, E, d)$ y dados dos vértices s y t, a los que denominamos vértice inicial o fuente y vértice final u objetivo, respectivamente, encuéntrese un camino que parta de s, finalice en t, esté formado por exactamente k aristas y tal que ningún otro camino entre dichos vértices y formado por k aristas presenta menor distancia.

Formalmente, buscamos un camino $(v_1, v_2, \ldots, v_{k+1})$ tal que $v_1 = s$ y $v_{k+1} = t$ con distancia mínima.

$$(\hat{v}_1, \hat{v}_2, \ldots, \hat{v}_{k+1}) = \arg \min_{(v_1, v_2, \ldots, v_{k+1}) \in P(t,k)} D(v_1, v_2, \ldots, v_{k+1}).$$

donde $D(v_1, v_2, \ldots, v_n)$ es la función de ponderación de caminos,

$$D(v_1, v_2, \ldots, v_n) = \sum_{0<i\leq n} d(v_{i-1}, v_i),$$

y donde $P(t, k)$ es el conjunto de caminos entre s y t formados por k aristas. Podemos definir este conjunto recursivamente:

$$P(t,k) = \{(v_1, v_2, \ldots, v_{k+1}) \mid v_1 = s; v_{k+1} = t; (v_1, v_2, \ldots, v_k) \in P(u, k-1); (u, t) \in E\}.$$

El caso base es $P(s, 0) = \{(s)\}$ y $P(v, 0) = \emptyset$ si $v \neq s$.

Deseamos conocer el camino más corto,

$$\arg \min_{(v_1, v_2, \ldots, v_{k+1}) \in P(t,k)} D(v_1, v_2, \ldots, v_{k+1})$$

y, por otra parte, la distancia recorrida por dicho el camino:

$$\min_{(v_1, v_2, \ldots, v_{k+1}) \in P(t,k)} D(v_1, v_2, \ldots, v_{k+1}).$$

8.5.1. Ecuación recursiva

Nos conviene definir ahora $D(t,k)$ como el peso del camino con k aristas más corto entre s y t:

$$D(t,k) = \min_{(v_1,v_2,\ldots,v_{k+1})\in P(t,k)} D(v_1,v_2,\ldots,v_{k+1}) = \min_{(v_1,v_2,\ldots,v_{k+1})\in P(t,k)} \left(\sum_{1\leq i<k+1} d(v_i,v_{i+1}) \right).$$

Sustituimos ahora $P(t,k)$ en la definición de $D(t,k)$ por su expresión recursiva:

$$D(t,k) = \min_{(u,t)\in E} \left(\min_{(v_1,v_2,\ldots,v_k)\in P(u,k-1)} \left(\left(\sum_{1\leq i<k} d(v_i,v_{i+1}) \right) + d(u,t) \right) \right).$$

Extraemos de la minimización interior el último término del sumatorio:

$$D(t,k) = \min_{(u,t)\in E} \left(\left(\min_{(v_1,v_2,\ldots,v_k)\in P(u,k-1)} \left(\sum_{1\leq i<k} d(v_i,v_{i+1}) \right) \right) + d(u,t) \right).$$

Podemos reescribir el sumatorio interior, que no es más que la distancia de un camino:

$$D(t,k) = \min_{(u,t)\in E} \left(\left(\min_{(v_1,v_2,\ldots,v_k)\in P(u,k-1)} D(v_1,v_2,\ldots,v_k) \right) + d(u,t) \right).$$

Hemos llegado a una relación recursiva:

$$D(t,k) = \min_{(u,t)\in E} \left(D(u,k-1) + d(u,t) \right).$$

Y como no hay nada especial en t, podríamos haber deducido la recursión para un vértice v cualquiera:

$$D(v,k) = \min_{(u,v)\in E} \left(D(u,k-1) + d(u,v) \right).$$

Hay un caso que podemos considerar base de la recursión: cuando $v = s$ y $k = 0$ se propone la búsqueda de la distancia óptima de s a s por un camino que no tiene arista alguna. Dicha distancia es, por definición, nula:

$$D(s,0) = 0.$$

Otro caso base tiene lugar si $v \neq s$ y $k = 0$, ya que todo camino válido debe empezar en s:

$$D(v,0) = +\infty.$$

Un caso especial es el de los vértices sin predecesores. La minimización se propone entonces sobre un conjunto vacío de elementos.

$$D(v,k) = \begin{cases} 0, & \text{si } v = s \text{ y } k = 0; \\ +\infty, & \text{si } v \neq s \text{ y } k = 0; \\ +\infty, & \text{si } \nexists (u,v) \in E \text{ y } k > 0; \\ \min\limits_{(u,v)\in E} \left(D(u,k-1) + d(u,v) \right), & \text{en otro caso.} \end{cases} \tag{8.8}$$

8.5.2. Grafo de dependencias y algoritmo iterativo

$D(t,k)$ es la distancia del camino más corto que lleva de s a t formado por exactamente k aristas. Hay una interpretación interesante de esta ecuación recursiva que se ilustra en la figura 8.24. Si buscamos la distancia del camino con 3 aristas más corto entre los vértices 0 y 5 del grafo de la figura 8.24 (a), por ejemplo, buscamos la distancia del camino más corto entre los vértices $(0,0)$ y $(5,3)$ del grafo de la figura 8.24 (b). Se trata de un grafo multietapa que se puede formar a partir del primero disponiendo en cada una de sus $k+1$ etapas un «copia» de los vértices del grafo original y disponiendo, por cada arista entre dos vértices u y v del grafo original, una arista entre los vértices correspondientes de dos etapas consecutivas.

Dado un grafo $G = (V,E)$, un valor positivo k y una función de ponderación d, su grafo multietapa asociado, $G' = (V',E')$, es

$$V' = \{(v,i) \mid v \in V, 0 \leq i \leq k\},$$
$$E' = \{((u,i-1),(v,i)) \mid (u,v) \in E, 0 < i \leq k\};$$

y su función de ponderación asociada d' se define así a partir de d:

$$d'((u,i-1),(v,i)) = d(u,v).$$

Nótese que todo camino $(v_1,v_2,\ldots,v_{k+1})$ en G con k aristas tiene un camino equivalente (con el mismo peso) en G': $((v_1,0),(v_2,1),\ldots,(v_{k+1},k))$. Y viceversa: todo camino en G' tiene un camino equivalente en G. Así pues, resulta evidente que calcular la distancia del camino más corto en G' entre $(s,0)$ y (t,k) es un problema equivalente a calcular la distancia del camino entre s y t de k aristas más corto en G.

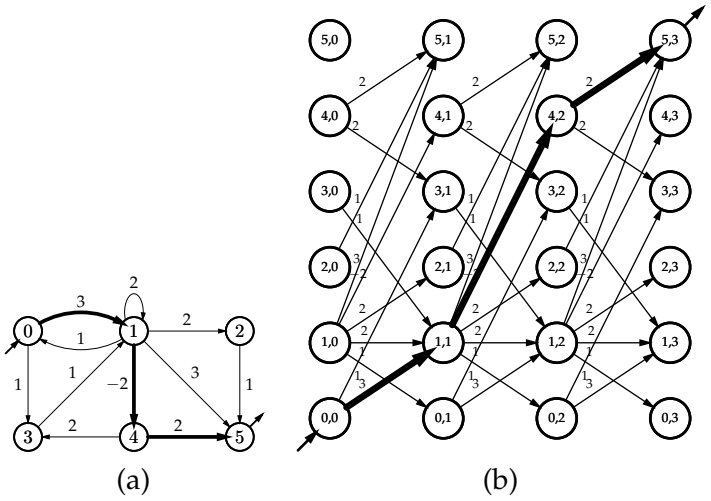


Figura 8.24: (a) Grafo ponderado (no acíclico). En trazo grueso se muestran las aristas del camino $(0,3,1,5)$, que es el más corto. (b) Grafo multietapa asociado al anterior sobre el que se efectúa la búsqueda del camino más corto entre 0 y 5 con 3 aristas. En trazo grueso se muestra el camino equivalente en este grafo al que se destacó en el grafo original de la izquierda.

No es necesario construir explícitamente G' para calcular su camino más corto entre $(s,0)$ y (t,k). He aquí una función iterativa implementada en Python:

k_edges.py

```
def k_edges_shortest_distance(G, d, s, t, k, infinity=3.4e+38):
    D = dict( ((v, 0), infinity) for v in G.V )
    D[s, 0] = 0
    for i in xrange(1, k+1):
        for v in G.V:
            if G.in_degree(v) == 0:
                D[v, i] = infinity
            else:
                D[v, i] = min( [ D[u, i-1] + d(u,v) for u in G.preds(v) ] )
    return D[t, k]
```

Probemos el algoritmo con el grafo de la figura 8.24 (a).

test_k_edges.py

```
from graph import Graph, WeightingFunction
from k_edges import k_edges_shortest_distance

G = Graph(V=range(6), E=[(0,1), (0,3), (1,0), (1,1), (1,4), (1,5), (2,5), (3,1), (4,3), (4,5)])
d = WeightingFunction({(0,1):3, (0,3):1, (1,0):1, (1,1):2, (1,4):-2, (1,5):3, (2,5):1,
                       (3,1):1, (4,3):2, (4,5):2})
for i in xrange(6):
    print 'Mínima distancia de 0 a 5 con %d aristas:' %i, k_edges_shortest_distance(G, d, 0, 5, i)
```

```
Mínima distancia de 0 a 5 con 0 aristas: 3.4e+38
Mínima distancia de 0 a 5 con 1 aristas: 3.4e+38
Mínima distancia de 0 a 5 con 2 aristas: 6
Mínima distancia de 0 a 5 con 3 aristas: 3
Mínima distancia de 0 a 5 con 4 aristas: 2
Mínima distancia de 0 a 5 con 5 aristas: 4
```

Observemos que el grafo de la figura 8.24 (b) es un grafo multietapa y, por tanto, acíclico. Podemos encontrar el camino más corto de $(s, 0)$ a (t, k) usando el algoritmo de cálculo del camino más corto en un grafo cíclico en tiempo proporcional a su número de aristas y consumiendo espacio proporcional a su número de vértices. El grafo tiene $(k+1)|V|$ vértices y $k|E|$ aristas. El coste espacial puede reducirse a sólo $O(|V|)$ si únicamente estamos interesados en la distancia del camino más corto: sólo se precisa almacenar valores intermedios para dos etapas: la actual y la anterior.

Cabe destacar que este algoritmo no es de aplicación exclusiva en grafos con ciclos: si deseamos calcular el camino más corto con k aristas en un grafo acíclico, hemos de proceder del mismo modo. Si deseamos, por ejemplo, calcular el camino más barato con 4 aristas en una instancia del río Congo con 7 embarcaderos (véase la figura 8.1), tendríamos que efectuar la búsqueda en un grafo multietapa como el que se muestra en la figura 8.25.

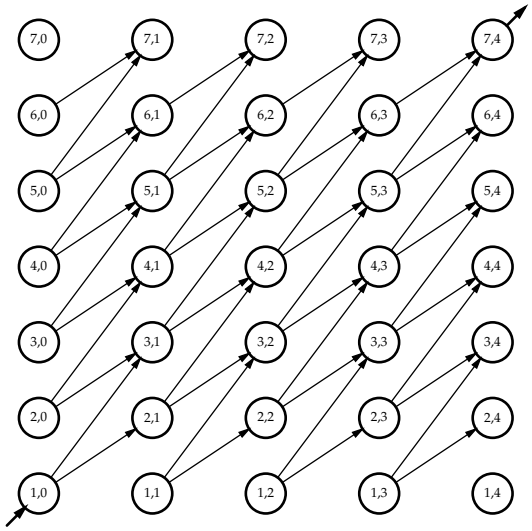


Figura 8.25: (a) Grafo ponderado sobre el que se efectúa la búsqueda del trayecto más barato con 4 aristas en una instancia del río Congo con 7 embarcaderos. Los arcos de la forma $((i,j),(i',j+1)$ tienen peso $c(i,i')$.

8.6. El problema del camino más corto en grafos sin ciclos negativos: el algoritmo de Bellman-Ford

Vamos a plantear ahora el problema del cálculo del camino más corto en un grafo ponderado y con ciclos. Sólo impondremos una condición: que el peso de un ciclo no sea negativo. Si lo fuera, el problema estaría mal definido, pues el camino óptimo tendría un número infinito de aristas. Debe tenerse presente que si la función de ponderación de aristas es no negativa, conviene usar el algoritmo de Dijkstra, pues presenta un coste computacional inferior al del algoritmo que vamos a desarrollar.

El problema que deseamos resolver se plantea así: Dado un grafo ponderado $G = (V, E, d)$ con ciclos no negativos y dados dos vértices s y t, a los que denominamos vértice inicial o fuente y vértice final u objetivo, respectivamente, encuéntrese un camino que parta de s y finalice en t tal que ningún otro camino entre dichos vértices presente menor distancia.

Formalmente, buscamos un camino $(v_1, v_2, \ldots, v_n)$ tal que $v_1 = s$ y $v_n = t$ con distancia mínima.

$$(\hat{v}_1, \hat{v}_2, \ldots, \hat{v}_n) = \arg \min_{(v_1,v_2,\ldots,v_n)\in P(t)} D(v_1, v_2, \ldots, v_n).$$

donde $P(t)$ es el conjunto de caminos entre s y t y se define así:

$$P(t) = \{(v_1, v_2, \ldots, v_n) \mid v_1 = s, v_n = t; (v_i, v_{i+1}) \in E, 1 \leq i < n\},$$

y $D(v_1, v_2, \ldots, v_n)$ es la función de ponderación del camino:

$$D(v_1, v_2, \ldots, v_n) = \sum_{0<i\leq n} d(v_{i-1}, v_i).$$

Por otra parte, deseamos conocer también la distancia recorrida por el camino más corto:

$$\min_{(v_1,v_2,\ldots,v_n)\in P(t)} D(v_1, v_2, \ldots, v_n).$$

8.6.1. Ecuación recursiva

Al derivar la ecuación recursiva para el problema del camino más corto en un grafo acíclico obtuvimos (ecuación 8.7)

$$D(v) = \begin{cases} 0, & \text{si } v = s; \\ +\infty, & \text{si } v \neq s \text{ y } \nexists (u,v) \in E; \\ \text{mín}_{(u,v)\in E} \left(D(u) + d(u,v) \right), & \text{en otro caso.} \end{cases}$$

Podemos derivar esta misma ecuación para el problema que nos ocupa ahora. Pero si el grafo presenta ciclos, no es posible encontrar un orden topológico en el grafo de dependencias (que coincide con el propio grafo G) con el que resolver la ecuación recursiva. ¿Qué hacer entonces?

El camino más corto entre s y t no puede tener más de $|V| - 1$ aristas. Si tuviera más, contendría al menos un ciclo. Como no admitimos ciclos negativos, siempre sería posible mejorar el peso del camino eliminado ese hipotético ciclo. Así pues,

$$D(t) = \min_{0 \leq k < |V|} D(t,k).$$

Es decir, el peso del camino más corto entre s y t es el menor de entre los pesos de los camino más cortos de s a t con k aristas, para k inferior a $|V|$. Por tanto, una posible forma de obtener dicha distancia sería efectuando $|V|$ llamadas a la función *k_edges_shortest_distance*. Sin embargo, es posible efectuar el cálculo completo con el coste de una sola llamada a la función *k_edges_shortest_distance*. Si calculamos el camino más corto con $|V| - 1$ aristas podemos obtener, como subproducto, el valor de $D(t,k)$ para todo $k < |V|$.

Recordemos que $D(v,k)$ se define así (ecuación (8.8)):

$$D(v,k) = \begin{cases} 0, & \text{si } v = s \text{ y } k = 0; \\ +\infty, & \text{si } v = s \text{ y } k > 0; \\ +\infty, & \text{si } v \neq s \text{ y } \nexists (u,v) \in E; \\ \text{mín}_{(u,v)\in E} \left(D(u,k-1) + d(u,v) \right), & \text{en otro caso.} \end{cases}$$

Hemos de calcular el valor de la expresión $\text{mín}_{0 \leq k < |V|} D(t,k)$ para conocer el peso del camino más corto con cualquier número de aristas.

EJERCICIOS

12 Deriva paso a paso la ecuación recursiva.

8.6.2. Grafo de dependencias y algoritmo iterativo

En la figura 8.26 se muestra el grafo que hubiésemos construido implícitamente al calcular el camino más corto entre 0 y 5 con 5 aristas en el grafo de la figura 8.24 (a). En ese grafo, hemos considerado que todos los vértices de la forma (t,i) son finales, para $0 \leq i < |V|$. La distancia mínima de entre las calculadas para cada uno de esos vértices es $\text{mín}_{0 \leq i < 6} D(5,i) = D(5,4) = 2$. Se trata de un grafo acíclico, así que el algoritmo desarrollado en el apartado anterior encuentra aplicación en este nuevo problema.

El algoritmo diseñado se conoce por algoritmo de Bellman-Ford:

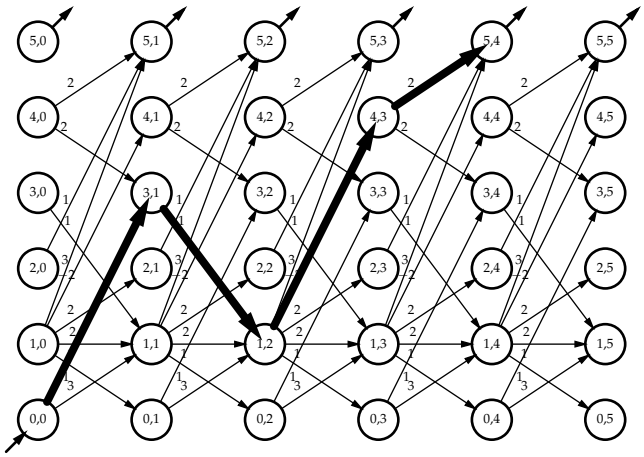


Figura 8.26: Grafo multietapa asociado al grafo de la figura 8.24 (a) sobre el que se efectúa implícitamente la búsqueda del camino más corto entre 0 y 5 con cualquier número de aristas. En trazo grueso se muestra el camino óptimo.

bellman_ford.py

```
def shortest_distance(G, d, s, t, infinity=3.4e+38):
    D = dict( ((v,0), infinity) for v in G.V )
    D[s, 0] = 0
    for i in xrange(1, len(G.V)):
        for v in G.V:
            if G.in_degree(v) == 0:
                D[v, i] = infinity
            else:
                D[v, i] = min( [ D[u, i-1] + d(u,v) for u in G.preds(v) ] )
    return min(D[t, i] for i in xrange(len(G.V)))
```

Ponemos a prueba el programa con el grafo 8.24 (a):

test_bellman_ford_b.py

```
from graph import Graph, WeightingFunction
from bellman_ford import shortest_distance

G = Graph(V=range(6), E=[(0,1), (0,3), (1,0), (1,1), (1,4), (1,5), (2,5), (3,1), (4,3), (4,5)])
d = WeightingFunction({(0,1):3, (0,3):1, (1,0):1, (1,1):2, (1,4):-2, (1,5):3, (2,5):1,
                       (3,1):1, (4,3):2, (4,5):2})
print 'Distancia de 0 a 5:', shortest_distance(G, d, 0, 5)
```

```
Distancia de 0 a 5: 2
```

8.6.3. El camino más corto

A la vista de que hemos reducido el cálculo del camino más corto en un grafo con ciclos (no negativos) al del camino más corto en un grafo acíclico obtenido a partir del grafo original con ciclos, resulta inmediato aplicar la técnica de los punteros hacia atrás para recuperar el camino óptimo.

bellman_ford.py

```
def shortest_path(G, d, s, t, infinity=3.4e+38):
    D = dict( ((v,0), infinity) for v in G.V )
    B = dict( ((v,0), None) for v in G.V )
    D[s, 0] = 0
    for i in xrange(1, len(G.V)):
        for v in G.V:
            if G.in_degree(v) == 0:
                D[v, i], B[v, i] = infinity, None
            else:
                D[v, i], B[v, i] = min((D[u, i-1] + d(u,v), u) for u in G.preds(v))
    mindist, k = min( (D[t, i], i) for i in xrange(len(G.V)))
    path = [t]
    while B[path[-1], k] != None:
        path.append( B[path[-1], k] )
        k -= 1
    path.reverse()
    return mindist, path
```

test_bellman_ford_c.py

```
from graph import Graph, WeightingFunction
from bellman_ford import shortest_path

G = Graph(V=range(6), E=[(0,1), (0,3), (1,0), (1,1), (1,4), (1,5), (2,5), (3,1), (4,3), (4,5)])
d = WeightingFunction({(0,1):3, (0,3):1, (1,0):1, (1,1):2, (1,4):-2, (1,5):3, (2,5):1,
                       (3,1):1, (4,3):2, (4,5):2})
mindist, path = shortest_path(G, d, 0, 5)
print 'Distancia de 0 a 5:', mindist
print 'Camino de distancia mínima de 0 a 5:', path
```

```
Distancia de 0 a 5: 2
Camino de distancia mínima de 0 a 5: [0, 3, 1, 4, 5]
```

8.6.4. Complejidad computacional

Más alla de un análisis directo de la complejidad, interesa reflexionar brevemente acerca de esta técnica de resolución. El problema de calcular el camino más corto en un grafo con ciclos no negativos se ha reducido a calcular el camino más corto en un grafo acíclico y multietapa. Hay una correspondencia biunívoca entre caminos con menos de $|V|$ vértices en el grafo original y caminos en el nuevo grafo. Hemos podido, de este modo, aplicar una técnica ya conocida para resolver el problema: encontramos el camino más corto en el nuevo grafo y devolvemos el camino asociado en el grafo original. El coste tiene, por tanto, tres componentes diferentes:

- la construcción del grafo multietapa,
- el cálculo del camino más corto en él,
- y la obtención del camino asociado en el grafo original.

El primer paso no tiene coste alguno porque no se efectúa explícitamente. Si $G = (V, E)$ es el grafo original, el grafo acíclico sobre el que efectuamos el cálculo del camino más corto presenta $|V|^2$ vértices y $(|V| - 1) \cdot |E|$ aristas. Como el cálculo del camino más corto en un grafo acíclico presenta una complejidad temporal $O(|V'| + |E'|)$, ésa es la complejidad temporal de este paso. La obtención del camino asociado sólo requiere eliminar la información relativa a las etapas del nuevo grafo y puede realizarse en tiempo proporcional a la longitud del camino, es decir, en tiempo $O(|V|)$.

El coste temporal del algoritmo es, pues, $O(|V||E|)$. El coste espacial es $O(|V|^2)$, pues ése es el número de vértices del grafo multietapa.

EJERCICIOS

13 ¿Se puede reducir la complejidad espacial siempre si sólo interesa el valor de la distancia mínima?

14 Si efectuamos el cálculo del camino más corto con el algoritmo de Bellman-Ford en un grafo no acíclico y ponderado *positivo*, ¿es necesario llegar hasta la etapa $|V|$? ¿Puede detenerse antes el algoritmo con la plena garantía de que se calcula correctamente el camino más corto?

8.7. El problema del desglose óptimo de una cantidad de dinero

Ya estudiamos el problema del desglose óptimo de una cantidad de dinero en la sección **??**. Vimos que los algoritmos voraces no siempre eran capaces de proporcionar una solución óptima y, a veces, ni tan siquiera una solución factible, aun cuando la hubiera. El algoritmo que desarrollamos en esta sección encuentra la solución óptima si hay alguna forma factible de desglose; si no la hay, el algoritmo puede utilizarse para detectar la imposibilidad de efectuar tal desglose.

Recordemos brevemente el planteamiento del problema. Disponemos de un sistema monetario con N monedas de valores respectivos $(v_1, v_2, v_3, \ldots, v_N) \in (\mathbb{Z}^{>0})^N$. Queremos desglosar una cantidad Q de modo que el número de monedas usado sea mínimo. Disponemos de una cantidad ilimitada de monedas de cada valor. (Estudiaremos el problema con limitación de monedas de cada valor en la sección 8.9.)

Se trata de un problema de optimización. El espacio de soluciones factibles puede expresarse como este conjunto de tuplas de números naturales con N componentes:

$$X = \left\{ (x_1, x_2, \ldots, x_N) \in \mathbb{N}^N \mid \sum_{1 \le i \le N} x_i v_i = Q \right\},$$

donde x_i es el número de monedas de valor v_i.

La función objetivo es

$$f((x_1, x_2, \ldots, x_N)) = \sum_{1 \le i \le N} x_i.$$

Deseamos encontrar el vector $(\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_N)$ que minimiza el valor de $f(x)$ para todo x en X:

$$(\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_N) = \arg \min_{(x_1, x_2, \ldots, x_N) \in X} \sum_{1 \le i \le N} x_i.$$

Asociado a este problema tenemos el de calcular el menor valor de $f(x)$ para todo x en X,

$$\min_{(x_1,x_2,\ldots,x_N)\in X} \sum_{1\leq i\leq N} x_i.$$

Como es habitual, empezamos por resolver el segundo problema.

8.7.1. Ecuación recursiva

Queremos formar una solución de la forma $(x_1, x_2, \ldots, x_N)$ que desglose la cantidad Q. Hemos modelado las soluciones como secuencia de decisiones: cada número x_i indica la cantidad de monedas de valor v_i que consideramos. Deseamos calcular

$$\min_{(x_1,x_2,\ldots,x_N)\in X} \sum_{1\leq i\leq N} x_i = \min_{\substack{(x_1,x_2,\ldots,x_N):\\ \sum_{1\leq i\leq N} x_i v_i = Q}} \sum_{1\leq i\leq N} x_i.$$

¿Qué puede valer x_N, el último componente del vector que hace mínimo el sumatorio? Es decir, ¿cuántas monedas de valor v_N podemos usar al desglosar la cantidad Q? Cualquier número natural entre 0 y $\lfloor Q/v_N \rfloor$. Sea cual sea, habremos generado un nuevo subproblema: encontrar un vector con una componente menos, $(x_1, x_2, \ldots, x_{N-1})$, para desglosar la cantidad $Q - x_N v_N$ con un juego de monedas de valores $(v_1, v_2, \ldots, v_{N-1})$.

$$\min_{\substack{(x_1,x_2,\ldots,x_N):\\ \sum_{1\leq i\leq N} x_i v_i = Q}} \sum_{1\leq i\leq N} x_i = \min_{0\leq x_N\leq \lfloor Q/v_N\rfloor} \left(\min_{\substack{(x_1,x_2,\ldots,x_{N-1}):\\ \sum_{1\leq i\leq N-1} x_i v_i = Q - x_N v_N}} \left(\left(\sum_{1\leq i\leq N-1} x_i \right) + x_N \right) \right).$$

O sea, para desglosar la cantidad Q con N tipos de monedas, hemos de resolver antes una serie de problemas relacionados: desglosar una cantidad Q', para diferentes valores de Q', con $N-1$ tipos de moneda. Los valores de Q' son Q, $Q - v_N$, $Q - 2v_N$, $Q - 3v_N$, $\ldots$, $Q - \lfloor Q/v_N \rfloor v_N$.

Como el operador suma sobre los naturales es monótono no decreciente por la izquierda y estamos minimizando,

$$\min_{\substack{(x_1,x_2,\ldots,x_N):\\ \sum_{1\leq i\leq N} x_i v_i = Q}} \sum_{1\leq i\leq N} x_i = \min_{0\leq x_N\leq \lfloor Q/v_N\rfloor} \left(\left(\min_{\substack{(x_1,x_2,\ldots,x_{N-1}):\\ \sum_{1\leq i\leq N-1} x_i v_i = Q - x_N v_N}} \sum_{1\leq i\leq N-1} x_i \right) + x_N \right).$$

Nótese que la minimización a mano izquierda de la igualdad y la minimización interior de la parte derecha guardan una gran semejanza: la primera propone el cálculo del desglose óptimo de la cantidad Q con N valores de moneda y la segunda propone el desglose óptimo de la cantidad $Q - x_N v_N$ con $N-1$ valores de moneda (donde x_N puede ser cualquier natural en cierto rango que empieza en cero). Podemos denominar $M(i, q)$ al número mínimo de monedas con que podemos desglosar la cantidad q usando el juego de monedas $(v_1, v_2, \ldots, v_i)$. Con esta notación, la parte izquierda es $M(N, Q)$ y la minimización interior es $M(N-1, Q - x_N v_N)$:

$$M(N, Q) = \min_{0\leq x_N\leq \lfloor Q/v_N\rfloor} \Big(M(N-1, Q - x_N v_N) + x_N \Big).$$

Hemos encontrado una relación recursiva entre problemas de la misma naturaleza pero diferente talla. ¿Qué caso o casos base encontramos? Podemos considerar qué ocurre cuando nos solicitan desglosar la cantidad 0 con 0 valores diferentes de moneda: la solución es 0. Si nos piden desglosar una cantidad mayor que cero con el mismo juego vacío de valores, no hay solución válida. Como estamos minimizando, podemos «marcar» este caso extraordinario con una penalización alta: un valor infinito.

La ecuación recursiva es

$$M(i,q) = \begin{cases} 0, & \text{si } i = 0 \text{ y } q = 0; \\ +\infty, & \text{si } i = 0 \text{ y } q > 0; \\ \min\limits_{0 \le x_i \le \lfloor q/v_i \rfloor} \Big(M(i-1, q - x_i v_i) + x_i \Big), & \text{en otro caso.} \end{cases} \tag{8.9}$$

Y deseamos calcular $M(N, Q)$.

8.7.2. Implementación directa de la ecuación recursiva

Es fácil codificar en Python la ecuación recursiva (8.9):

dynprog_money_change.py

```
1 def recursive_number_of_coins(Q, v, infinity=2**31):
2   def M(i, q):
3     if i==0:
4       if q==0: return 0
5       else:    return infinity
6     else:
7       return min(M(i-1, q-x*v[i]) + x for x in xrange(0,q/v[i]+1))
8   return M(len(v), Q)
```

test_dynprog_money_change.py

```
1 from dynprog_money_change import recursive_number_of_coins
2 from offsetarray import OffsetArray
3
4 Q = 24
5 v = OffsetArray([1, 2, 5, 10, 20, 50])
6 print "Monedas usadas al desglosar la cantidad %d con %s:" % (Q, v),
7 print recursive_number_of_coins(Q, v)
```

```
Monedas usadas al desglosar la cantidad 24 con [1, 2, 5, 10, 20, 50]: 3
```

Esta implementación presenta un grave problema: repite muchas llamadas a la función M con los mismos argumentos. En la figura 8.27 se muestra parte del árbol de llamadas asociado a la ejecución de $M(6, 24)$ con monedas de valores 1, 2, 5, 10, 20 y 50 (por problemas de espacio lo mostramos completo, pero sin etiquetar, en la figura 8.28). En el árbol parcial se puede apreciar, por ejemplo, que la llamada $M(2, 4)$ se efectúa en 4 ocasiones.

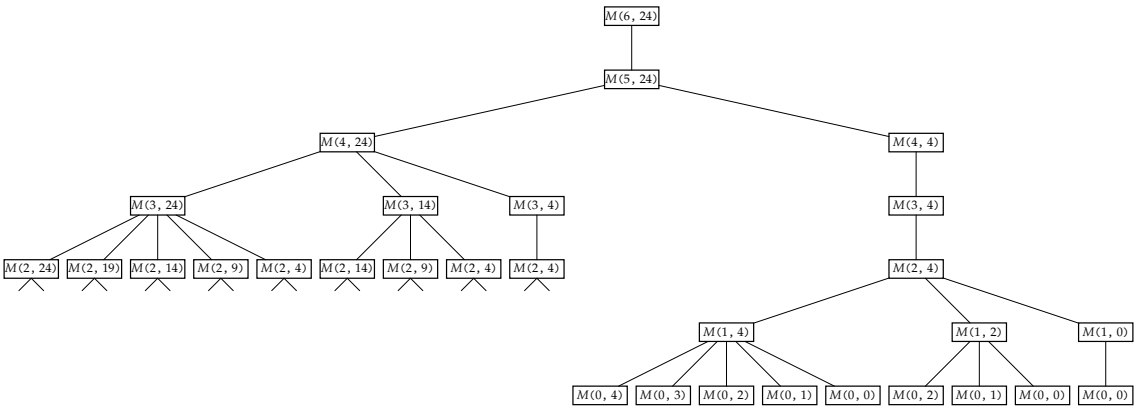


Figura 8.27: Parte del árbol de llamadas generado al invocar M(6, 24) en recursive_number_of_coins con monedas de valores 1, 2, 5, 10, 20 y 50.

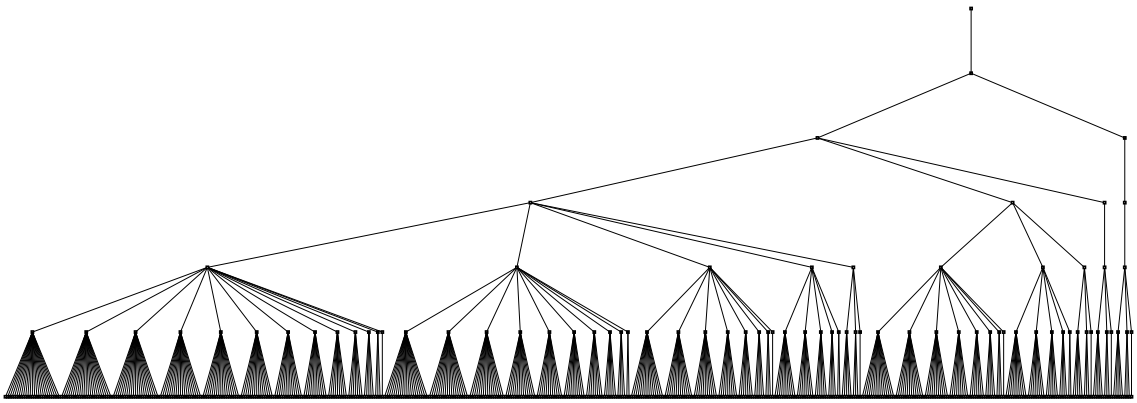


Figura 8.28: Representación completa (sin etiquetar) del árbol de llamadas que se muestra parcialmente en la figura 8.27.

Es evidente que la técnica de memorización de resultados sería de ayuda para evitar cálculos repetidos. Un almacén matricial con $(N+1) \times (Q+1)$ celdas es suficiente para almacenar resultados asociados a las llamadas recursivas.

La figura 8.29 muestra cómo se relacionan entre sí las diferentes celdas mediante un grafo de dependencias entre resultados de llamadas recursivas. Un vértice de la forma (i, q) representa el resultado de una llamada de la forma $M(i, q)$. Una arista que une $(i-1, q')$ con (i, q) indica que el resultado de $M(i-1, q')$ se usa en el cálculo del resultado de $M(i, q)$; en particular, supone el empleo de $x_i = (q - q')/v_i$ monedas de valor v_i. En la figura 8.30 anotamos junto a una arista del grafo de dependencias su significado.

El grafo de la figura 8.29 presenta una estructura clara: es un grafo multietapa con $N+1$ etapas. Un camino entre cualquier vértice inicial y el vértice final puede verse como un desglose en el que cada arista detalla el número de monedas de cierto valor que forman parte de una solución factible. Si el camino no empieza en (0,0), describe un desglose incompleto y, por tanto, no se admite como solución factible. El camino escogido

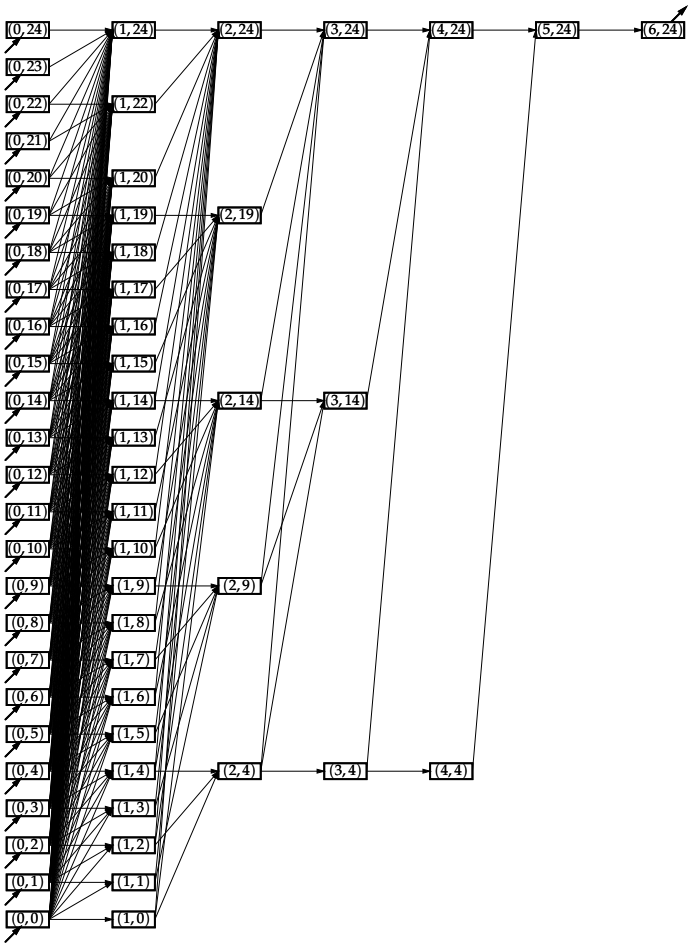


Figura 8.29: Grafo de dependencias entre resultados de la evaluación de cada llamada recursiva del árbol de la figura 8.28.

por el algoritmo sólo puede empezar en un vértice diferente de $(0, 0)$ si no existe solución factible alguna.

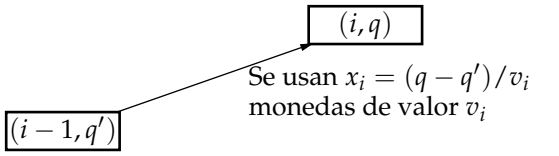


Figura 8.30: Significado de una arista en el grafo de dependencias de la figura 8.29.

He aquí una implementación del algoritmo con memorización:

dynprog_money_change.py

```
1 def memoized_number_of_coins(Q, v, infinity=2**31):
2     R = {}
3     def M(i, q):
4         if i==0:
5             if q == 0: R[0,0] = 0
6             else:      R[0,q] = infinity
7         else:
```

```
8            for x in xrange(0, q/v[i]+1):
9                if (i-1, q-x*v[i]) not in R: M(i-1, q-x*v[i])
10           R[i,q] = min(R[i-1,q-x*v[i]] + x for x in xrange(0, q/v[i]+1))
11       return R[i, q]
12
13   return M(len(v), Q)
```

test_dynprog_money_change2.py

```
1  from dynprog_money_change import memoized_number_of_coins
2  from offsetarray import OffsetArray
3
4  Q = 24
5  v = OffsetArray([1, 2, 5, 10, 20, 50])
6  print "Monedas usadas al desglosar la cantidad %d con %s:" % (Q, v),
7  print memoized_number_of_coins(Q, v)
```

```
Monedas usadas al desglosar la cantidad 24 con [1, 2, 5, 10, 20, 50]: 3
```

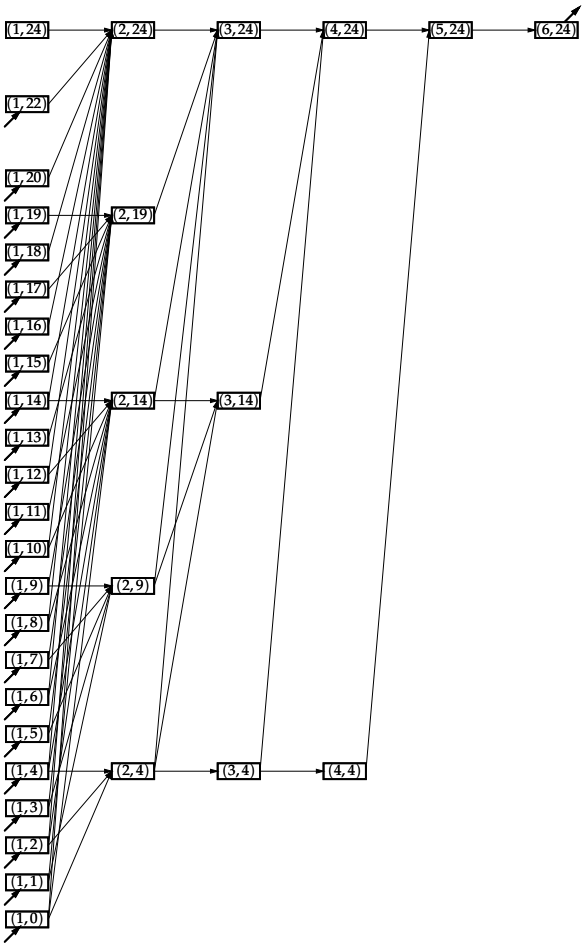
El coste temporal del algoritmo es $O(NQ^2)$ aunque, a la vista del grafo de dependencias, podemos apreciar que el número de aristas es sensiblemente inferior a NQ^2. La complejidad espacial es $O(NQ)$: aunque la profundidad de la pila de llamadas recursivas es $O(N)$, el almacén de resultados asociados a cada llamada recursiva requiere espacio $O(NQ)$.

EJERCICIOS

15 Hay un elevado número de vértices en la primera etapa y un número aún mayor de aristas (cuadrático con Q) entre la primera etapa y la segunda del grafo de la figura 8.29. Todos los vértices de la primera columna, excepto el etiquetado con $(0,0)$, corresponden a desgloses inválidos, pues no proporcionan la cantidad exacta y el valor asociado es infinito. Una ecuación recursiva alternativa prescinde de los vértices de la primera etapa:

$$M(i,q) = \begin{cases} q/v_1, & \text{si } i = 1 \text{ y } (q \bmod v_1) = 0; \\ +\infty, & \text{si } i = 1 \text{ y } (q \bmod v_1) \neq 0; \\ \min\limits_{0 \leq x_i \leq \lfloor q/v_i \rfloor} \Big(M(i-1, q - x_i v_i) + x_i \Big), & \text{si } i > 1. \end{cases} \tag{8.10}$$

Éste es el grafo de dependencias inducido por esta nueva ecuación recursiva al resolver $M(6,24)$ con monedas de valores 1, 2, 5, 10, 20 y 50.



Diseña un algoritmo recursivo con memorización para esta nueva ecuación recursiva. Analiza su complejidad computacional.

8.7.3. Transformación recursivo-iterativa

Podemos plantear la evaluación del valor asociado al vértice (N, Q) si recorremos los vértices del grafo de dependencias entre resultados siguiendo un orden topológico, ya que el grafo es acíclico. Uno particularmente fácil de describir (y efectuar) es un recorrido por «columnas» o etapas.

Puede resultar difícil tratar de visitar únicamente las celdas de la matriz de resultados que se muestran en el grafo de la figura 8.29. Esta versión recorre los vértices del grafo de la figura 8.31, es decir, toda posible combinación de cantidades pendientes de desglosar y número de tipos de valor con el que efectuar el desglose. Los valores asociados a los vértices pueden almacenarse en un matriz de dimensión $(N + 1) \times (Q + 1)$. Este algoritmo, que sigue el segundo orden de recorrido propuesto, resuelve el problema iterativamente:

dynprog_money_change.py *(cont.)*

```
15 def number_of_coins1(Q, v, infinity=2**31):
16     M = dict(((0,q), infinity) for q in xrange(1, Q+1))
17     M[0,0] = 0
18     for i in xrange(1, len(v)+1):
19         for q in xrange(Q+1):
20             M[i,q] = min(M[i-1,q-x*v[i]]+x for x in xrange(0, q/v[i]+1))
21     return M[len(v),Q]
```

test_dynprog_money_change3.py

```
1 from dynprog_money_change import number_of_coins1
2 from offsetarray import OffsetArray
3
4 Q = 24
5 v = OffsetArray([1, 2, 5, 10, 20, 50])
6 print "Monedas usadas al desglosar la cantidad %d con %s:" % (Q, v),
7 print number_of_coins1(Q, v)
```

```
Monedas usadas al desglosar la cantidad 24 con [1, 2, 5, 10, 20, 50]: 3
```

La complejidad espacial es $\Theta(NQ)$. La complejidad temporal es $O(NQ^2)$, idéntica a la del algoritmo anterior, aunque a la vista del grafo de dependencias extendido (figura 8.31) resulta evidente que estamos haciendo un sobreesfuerzo considerable al calcular un valor para $M[i,q]$ en pares (i,q) que no se recogen en el grafo de dependencias original y que se muestra en la figura 8.29.

EJERCICIOS

16 Diseña un algoritmo que calcule el número mínimo de monedas y que consuma espacio $O(Q)$.

8.7.4. El desglose óptimo

Ya estamos en condiciones de plantear el cálculo de la cantidad de monedas de cada valor que corresponde al desglose óptimo. Basta con establecer una analogía entre el cálculo efectuado y la resolución del problema del camino más corto en un grafo. Buscamos un camino de coste mínimo entre los vértices $(0,0)$ y (N,Q) en el grafo de dependencias. El peso de la arista $((i-1,q'),(i,q))$ es el número de monedas de valor v_i utilizadas para «convertir» la cantidad q' en la cantidad q. Si usamos punteros hacia atrás, podemos recuperar el camino óptimo e interpretar en los términos descritos cada arista de dicho camino.

dynprog_money_change.py

```
1 def optimal_change(Q, v, infinity=2**31):
2     B = dict(((n,q),None) for q in xrange(Q+1) for n in xrange(len(v)+1))
3     M = dict(((0,q), infinity) for q in xrange(1, Q+1))
4     M[0,0] = 0
5     for i in xrange(1, len(v)+1):
6         for q in xrange(Q+1):
```

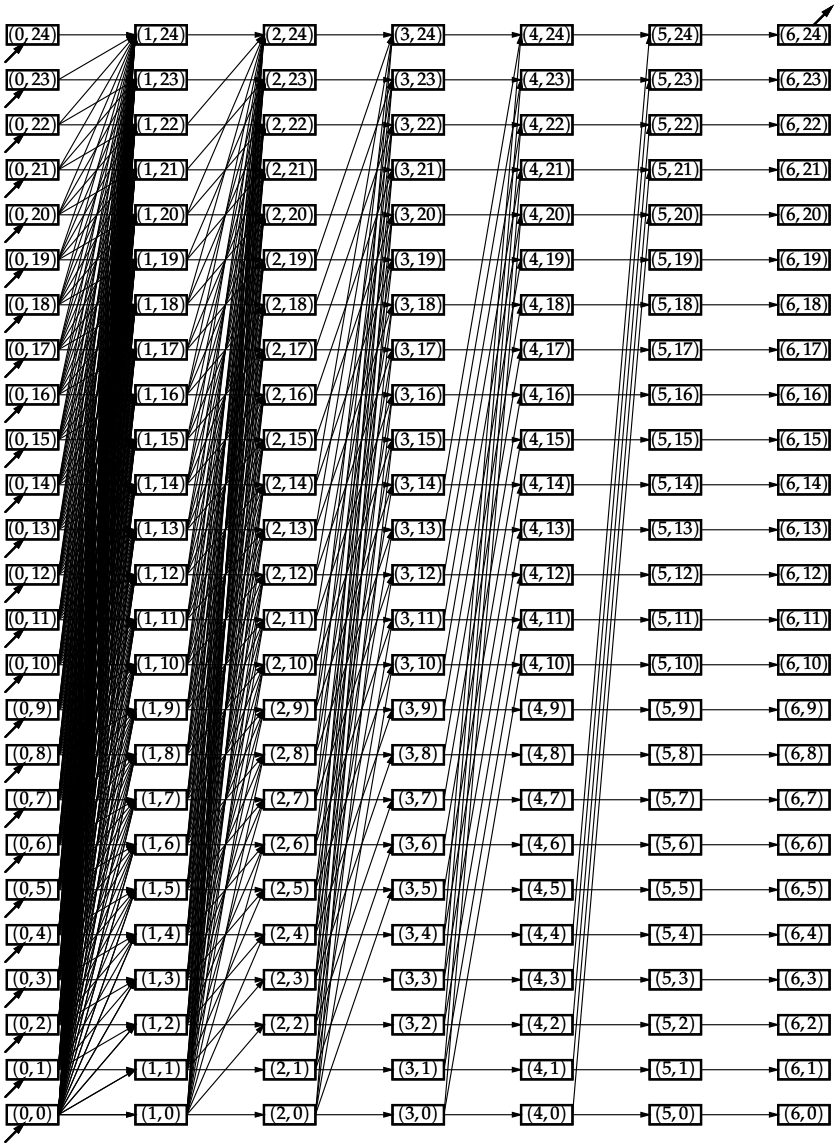


Figura 8.31: Grafo de dependencias extendido cuyos vértices se visitan durante el cálculo iterativo. El grafo de dependencias asociado a la ecuación recursiva (8.10), que se muestra en la figura 8.29, es un subgrafo de éste.

```
            M[i,q] = infinity
            for x in xrange(0, q/v[i]+1):
                if M[i-1,q-x*v[i]] + x < M[i,q]:
                    M[i,q] = M[i-1,q-x*v[i]] + x
                    B[i,q] = (i-1,q-x*v[i])
    change = {}
    (i,q) = (len(v),Q)
    while B[i,q] != None:
        (i',q') = B[i,q]
        change[i] = (q-q')/v[i]
        (i,q) = (i',q')
    return change
```

test_dynprog_money_change5.py

```
from dynprog_money_change import optimal_change
from offsetarray import OffsetArray

Q = 24
v = OffsetArray([1, 2, 5, 10, 20, 50])
change = optimal_change(Q, v)
for i in change:
    if change[i] == 1: print '%d moneda de valor %d' % (change[i], v[i])
    elif change[i] > 1: print '%d monedas de valor %d' % (change[i], v[i])
```

```
2 monedas de valor 2
1 moneda de valor 20
```

El coste espacial es $O(NQ)$. La reducción de memoria del ejercicio 7-16 «se pierde» al tener que almacenar los punteros hacia atrás para recuperar el camino óptimo.

EJERCICIOS

17 Diseña una versión del algoritmo recursivo con memorización que permita recuperar el desglose óptimo.

18 Diseña una versión del algoritmo con recuperación del desglose óptimo que no utilice punteros hacia atrás.

8.8. Una aproximación diferente al problema del desglose de una cantidad de dinero

Hemos formulado una ecuación recursiva y, a partir de ella, hemos derivado diferentes algoritmos de una forma relativamente mecánica. La ecuación recursiva (8.9) no es la única con la que podemos expresar la solución. Vamos a plantear ahora una ecuación diferente.

Hemos considerado que una solución es una tupla de tamaño fijo, $(x_1, x_2, \ldots, x_N)$ en la que cada elemento x_i indica el número de monedas de valor v_i considerado. Podemos formular las soluciones con un vector de monedas de talla arbitraria, digamos n. Cada elemento de una tupla $(x_1, x_2, \ldots, x_n)$ será el valor de una moneda y admitiremos valores

repetidos. La forma óptima de descomponer $Q = 24$ en un sistema con monedas de valor 1, 2, 5, 10, 20 y 50 consiste en dar una moneda de valor 20 y dos de valor 2. Con la notación adoptada se puede representar así: $(20, 2, 2)$.

Formalicemos esta nueva forma de plantear el problema. El conjunto de soluciones factibles será ahora una secuencia de valores de moneda:

$$X = \left\{ (x_1, x_2, \ldots, x_n) \in \{v_1, v_2, \ldots, v_N\}^* \mid \sum_{1 \leq i \leq n} x_i = Q \right\}.$$

La función objetivo, que deseamos minimizar, nos proporciona la talla de la secuencia:

$$f((x_1, x_2, \ldots, x_n)) = n.$$

8.8.1. Ecuación recursiva

Ahora que hemos modelado toda solución factible como una secuencia de valores, tratemos de encontrar una formulación recursiva en el cálculo del valor óptimo de la función objetivo.

Deseamos calcular

$$\min_{\substack{(x_1,x_2,\ldots,x_n)\in\{v_1,v_2,\ldots,v_N\}^*: \\ \sum_{1\leq i\leq n} x_i = Q}} n.$$

El último valor de moneda considerado en el desglose óptimo, $\hat{x}_n$, puede ser cualquiera de los del conjunto $\{v_1, v_2, \ldots, v_N\}$, pero teniendo en cuenta que la cantidad pendiente de desglosar, $Q - \hat{x}_n$, no sea negativa.

$$\begin{aligned} \min_{\substack{(x_1,x_2,\ldots,x_n)\in\{v_1,v_2,\ldots,v_N\}^*: \\ \sum_{1\leq i\leq n} x_i = Q}} n &= \min_{\substack{a\in\{v_1,v_2,\ldots,v_N\}: \\ Q-a\geq 0}} \left(\min_{\substack{(x_1,x_2,\ldots,x_{n-1})\in\{v_1,v_2,\ldots,v_N\}^*: \\ \sum_{1\leq i\leq n-1} x_i = Q-a}} n \right) \\ &= \min_{\substack{a\in\{v_1,v_2,\ldots,v_N\}: \\ Q-a\geq 0}} \left(\min_{\substack{(x_1,x_2,\ldots,x_{n-1})\in\{v_1,v_2,\ldots,v_N\}^*: \\ \sum_{1\leq i\leq n-1} x_i = Q-a}} n-1 \right) + 1. \end{aligned}$$

La igualdad entre estas expresiones se observa porque el operador suma sobre los naturales es monótono no decreciente por la izquierda y estamos minimizando.

Sea $M(Q)$ el menor número de monedas con que podemos desglosar la cantidad Q. Tenemos

$$M(Q) = \min_{\substack{a\in\{v_1,v_2,\ldots,v_N\}: \\ Q-a\geq 0}} M(Q-a) + 1.$$

Ya tenemos una expresión recursiva para el caso general. Necesitamos considerar un caso base. Resulta trivial calcular la descomposición óptima cuando la cantidad que queremos desglosar, Q, vale 0: tiene 0 monedas. Así pues, la ecuación recursiva es

> **!** Recuerda que asumimos que el mínimo de un conjunto vacío, $\min \emptyset$, vale infinito. Cuando q es mayor que cero y no hay ningún valor de moneda que restado de q dé una cantidad no negativa, el resultado de la minimización es infinito.

$$M(q) = \begin{cases} 0, & \text{si } q = 0; \\ \min\limits_{\substack{1 \le i \le N: \\ q - v_i \ge 0}} M(q - v_i) + 1, & \text{si } q > 0; \end{cases} \tag{8.11}$$

y deseamos calcular $M(Q)$.

8.8.2. Algoritmo recursivo

Resulta inmediato codificar en Python el cálculo recursivo con el que resolvemos la ecuación recursiva:

dynprog_money_change.py *(cont.)*

```
20 def recursive_number_of_coins2(Q, v):
21     def M(q):
22         if q==0: return 0
23         else:    return min( M(q-a)+1 for a in v if q-a >= 0 )
24     return M(Q)
```

test_dynprog_money_change6.py

```
1 from dynprog_money_change import recursive_number_of_coins2
2 from offsetarray import OffsetArray
3
4 Q = 24
5 v = OffsetArray([1, 2, 5, 10, 20, 50])
6 print "Monedas usadas al desglosar la cantidad %d con %s:" % (Q, v),
7 print recursive_number_of_coins2(Q, v)
```

```
Monedas usadas al desglosar la cantidad 24 con [1, 2, 5, 10, 20, 50]: 3
```

Como anteriores algoritmos recursivos, éste también efectúa numerosas llamadas a la función con los mismos argumentos. El tamaño del árbol de llamadas es enorme. Para ilustrar el crecimiento del número de nodos de los árboles completos, mostramos un árbol parcial con las llamadas recursivas efectuadas al calcular $M(24)$ en la figura 8.32. La figura 8.33 muestra los árboles completos correspondientes a las llamadas $M(5)$ y $M(10)$.

Una versión con memorización evita este problema. Obviamos la versión recursiva con memorización y pasamos directamente a una versión iterativa.

EJERCICIOS

19 Diseña un algoritmo con memorización para el cálculo de $M(Q)$.

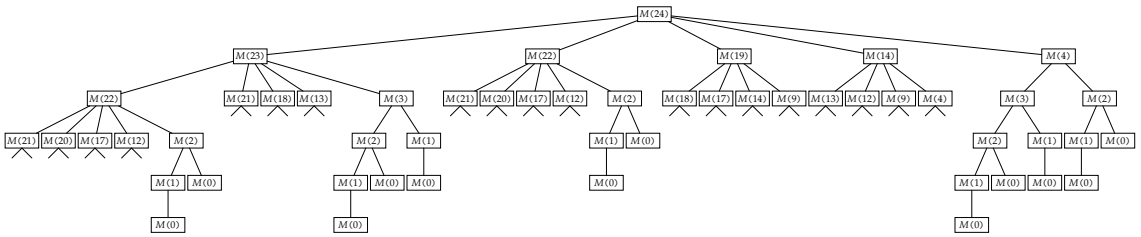


Figura 8.32: Árbol parcial de llamadas recursivas a M(24) en recursive_number_of_coins2 con monedas de valores 1, 2, 5, 10, 20 y 50.

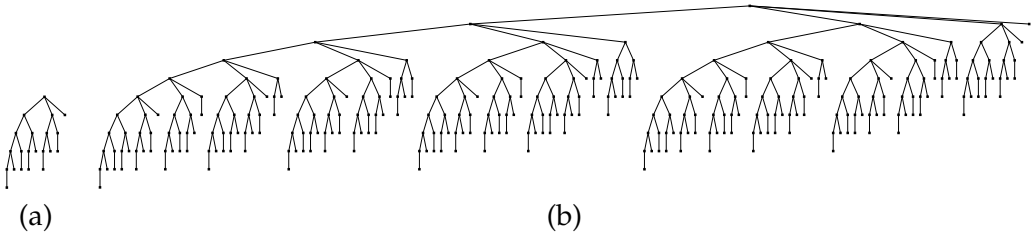


Figura 8.33: Árboles completos de llamadas recursivas para (a) M(5) y (b) M(10) con monedas de valores 1, 2, 5, 10, 20 y 50.

8.8.3. Transformación recursivo-iterativa

Para diseñar el algoritmo recursivo hemos de estudiar el grafo de dependencias entre resultados de llamadas a función. En la figura 8.34 se muestra dicho grafo de dependencias para $M(24)$ con v igual a `[1, 2, 5, 10, 20, 50]`. Cada arista de la forma (q', q) comporta el uso de una moneda de valor $q - q'$, como se muestra en la figura 8.35. El peso asociado para todas las aristas es 1, pues cada una representa el hecho de usar una moneda.

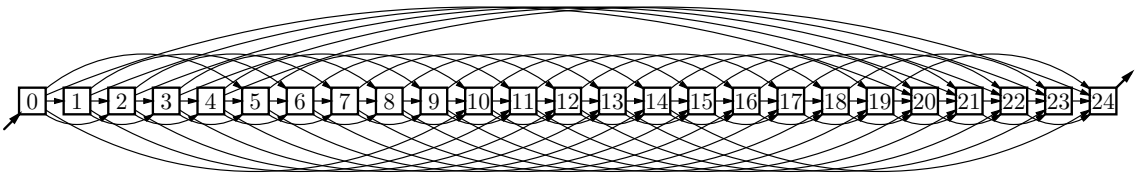
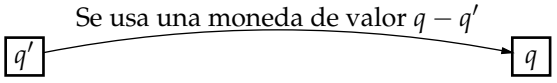


Figura 8.34: Grafo de dependencias asociado al cálculo de M(24) con un sistema con monedas de valores 1, 2, 5, 10, 20 y 50.

Figura 8.35: Significado de una arista en el grafo de dependencias de la figura 8.34.



Es evidente que podemos efectuar el cálculo de $M[Q]$ recorriendo el grafo de izquierda a derecha, es decir, usando una secuencia creciente de índices:

dynprog_money_change.py
```
1 def number_of_coins2(Q, v):
2   M = [None] * (Q+1)
3   M[0] = 0
4   for q in xrange(1, Q+1):
5     M[q] = min( M[q-a] + 1 for a in v if q-a>=0 )
6   return M[Q]
```

test_dynprog_money_change7.py
```
1 from dynprog_money_change import number_of_coins2
2 from offsetarray import OffsetArray
3
4 Q = 24
5 v = OffsetArray([1, 2, 5, 10, 20, 50])
6 print "Monedas usadas al desglosar la cantidad %d con %s:" % (Q, v),
7 print number_of_coins2(Q, v)
```

```
Monedas usadas al desglosar la cantidad 24 con [1, 2, 5, 10, 20, 50]: 3
```

El coste temporal del algoritmo es $O(NQ)$ y el coste espacial es $O(Q)$. La estructura del grafo nos permite diseñar una versión con consumo espacial $O(\text{máx}_{1\leq i\leq N} v_i)$.

Nótese que este algoritmo es más eficiente temporal y espacialmente que el que diseñamos a partir de la ecuación recursiva (8.10). O sea, el modelado del problema en términos de la resolución de una ecuación recursiva juega un papel crucial en el diseño de un buen algoritmo.

También resulta interesante destacar que el método resolutivo diseñado ha resultado ser equivalente al del cálculo del peso del camino más corto entre los vértices 0 y Q en un grafo acíclico cuyas aristas tienen todas peso 1. Recuerda que el recorrido por primero en anchura es la base para un algoritmo de cálculo del camino más corto que no necesariamente visita todos los vértices, así que puede conducir a una solución más eficiente que la estudiada en este apartado.

EJERCICIOS

20 Estudia la posibilidad de usar la exploración por primero en anchura para encontrar el desglose óptimo. ¿Depende el tiempo de ejecución del número de monedas devueltas? ¿Cómo?

8.8.4. El desglose óptimo

Como es ya habitual, usaremos la técnica de uso de punteros hacia atrás para recuperar el camino. Tan sólo hemos de tener la precaución de no devolver la lista de vértices, sino la lista de diferencias entre valores de vértices consecutivos, que es la lista de valores de monedas.

dynprog_money_change.py
```
1 def optimal_change2(Q, v):
2     B = [None] * (Q+1)
3     M = [None] * (Q+1)
4     M[0] = 0
```

```
    for q in xrange(1, Q+1):
        M[q] = Q+1 # Nos vale como infinito (es mayor que el máximo número de monedas).
        for a in v:
            if q-a>=0 and M[q-a] + 1 < M[q]:
                M[q] = M[q-a] + 1
                B[q] = q-a
    change = []
    q = Q
    while B[q] != None:
        change.append(q - B[q])
        q = B[q]
    return change
```

test_dynprog_money_change8.py

```
from dynprog_money_change import optimal_change2
from offsetarray import OffsetArray

Q = 24
v = OffsetArray([1, 2, 5, 10, 20, 50])
print 'Monedas:', optimal_change2(Q, v)
```

```
Monedas: [2, 2, 20]
```

La solución `[2, 20]` se lee como «una moneda de valor 2 y una moneda de valor 20». El algoritmo presenta un coste temporal $O(NQ)$ y espacial $O(Q)$.

EJERCICIOS

21 Un ayuntamiento tiene un presupuesto P para construir jardines y J posibles emplazamientos. El coste de construir un jardín en el emplazamiento j es c_j para $1 \leq j \leq J$. Diseña un algoritmo de programación dinámica que produzca como salida el mayor gasto que es posible realizar para construir jardines con el presupuesto P, además de los emplazamientos afortunados.

22 Diseña una solución para el problema de la mochila discreta considerando que los pesos de los objetos son valores reales positivos. Analiza el coste del algoritmo resultante.

8.9. Desglose de una cantidad de dinero con limitación del número de monedas de cada valor

Al estudiar el problema del desglose de una cantidad Q con un juego de monedas de valores $v_1, v_2, \ldots, v_N$ en la sección 8.7 consideramos que había una cantidad ilimitada de monedas de cada valor. No es la situación con que nos encontramos en un cajero automático o al dar cambio con una máquina expendedora.

Reformulemos el problema suponiendo que sólo disponemos de m_i monedas de valor v_i, para $1 \leq i \leq N$. La única diferencia con respecto a lo expuesto en el apartado 8.7 es el conjunto de soluciones, que ahora incorpora una restricción adicional:

$$X = \left\{ (x_1, x_2, \ldots, x_N) \in \mathbb{N}^N \mid 0 \leq x_i \leq m_i,\ 1 \leq i \leq N;\quad \sum_{1 \leq i \leq N} x_i v_i = Q \right\}.$$

8.9.1. Ecuación recursiva

No resulta difícil adaptar la ecuación recursiva (8.9) (véase la página 58) para que tenga en cuenta la nueva restricción:

$$M(i,q) = \begin{cases} 0, & \text{si } i = 0 \text{ y } q = 0; \\ +\infty, & \text{si } i = 0 \text{ y } q > 0; \\ \min\limits_{0 \le x_i \le \min(m_i, \lfloor q/v_i \rfloor)} \Big(M(i-1, q - x_i v_i) + x_i \Big), & \text{en otro caso.} \end{cases} \tag{8.12}$$

Deseamos calcular $M(N,Q)$.

El grafo de dependencias es similar al de la ecuación original, con la única diferencia de que el número de aristas incidentes en cualquier vértice de la columna i es igual o inferior a m_i. La figura **??** muestra el grafo de dependencias para una instancia concreta del problema. Si se compara con el grafo 8.29, que corresponde a la misma instancia sin limitación del número de monedas de cada tipo, se puede apreciar que hay menos vértices y muchas menos aristas.

8.9.2. Algoritmo iterativo

Se puede efectuar un recorrido de los vértices en el mismo orden que seguimos al considerar que la cantidad de monedas es ilimitada.

dynprog_limited_money_change.py

```
1 def number_of_coins(Q, v, m, infinity=2**31):
2     M = dict( ((0,q), infinity) for q in xrange(1, Q+1) )
3     M[0,0] = 0
4     for i in xrange(1, len(v)+1):
5         for q in range(Q+1):
6             M[i,q] = min(M[i-1,q-x*v[i]]+x for x in range(0, min(q/v[i]+1, m[i]+1)))
7     return M[len(v),Q]
```

test_dynprog_limited_money_change.py

```
1 from dynprog_limited_money_change import number_of_coins
2 from offsetarray import OffsetArray
3
4 Q = 24
5 v = OffsetArray([1, 2, 5, 10, 20, 50])
6 m = OffsetArray([3, 1, 4, 1, 2, 1])
7 print "Monedas usadas al desglosar la cantidad %d con %s" % (Q, v),
8 print "y limitación de monedas de cada valor a %s: %d" % (m, number_of_coins(Q, v, m))
```

```
Monedas usadas al desglosar la cantidad 24 con [1, 2, 5, 10, 20, 50] y limitació
n de monedas de cada valor a [3, 1, 4, 1, 2, 1]: 4
```

El coste temporal del algoritmo es $O(NQ^2)$. Si consideramos que la cantidad disponible de monedas de cada valor forma parte de la talla del problema, podemos expresar el coste temporal como una función $O(NQ\,\text{mín}(Q, \text{máx}(m_1, m_2, \ldots, m_N)))$. El coste espacial de una versión que almacene todo resultado intermedio es $O(NQ)$. Si sólo almacenamos dos columnas, la cantidad de memoria necesaria es $O(Q)$.

EJERCICIOS

23 Diseña un algoritmo que devuelva el desglose de monedas correspondiente a la solución óptima.

24 La ecuación (8.9) conducía a un algoritmo de coste temporal $O(NQ^2)$, cuando la ecuación (8.11) (véase la página 67) nos permitía diseñar un algoritmo $O(NQ)$, más eficiente. ¿No habría sido más práctico modificar la ecuación (8.11) para tener en cuenta la limitación de monedas disponibles? Trata de modificar la ecuación recursiva (8.11) y diseñar el correspondiente algoritmo iterativo.

25 Diseña un programa que simule un cajero automático. Inicialmente se dispone de cierta cantidad de cada tipo de billete. Cada vez que un usuario extrae una cantidad de dinero, el programa actualiza la cantidad de billetes disponibles. Necesitarás, pues, recuperar la secuencia óptima para saber cuántos billetes de cada tipo se entregan con cada cantidad de dinero.

8.10. El problema de la asignación óptima de recursos

Disponemos de U unidades de un recurso y deseamos asignar cierta cantidad del mismo a cada una de N actividades distintas. Una función $v : \mathbb{N} \times \mathbb{N} \to \mathbb{R}$ nos indica el beneficio que obtenemos al asignar u unidades del recurso a la actividad i con $v(i, u)$. Supondremos que no asignar ningún recurso a una actividad produce un beneficio nulo, es decir, que $v(i, 0) = 0$ para todo i entre 1 y N. El número de unidades del recurso que podemos asignar a una actividad i debe ser inferior o igual a m_i. Deseamos obtener la asignación de recursos a actividades que maximiza el beneficio total obtenido, es decir, la suma de beneficios que proporciona cada asignación individual.

> **!** Puede extrañar que no sea necesario asignar las U unidades del recurso. ¿No cabe esperar más beneficio cuánto más recurso se asigna a una actividad? No necesariamente. Imaginemos un huerto al que asignamos una cantidad de agua para riego: un exceso puede arruinar la cosecha. La función v no es monótona creciente con su segundo parámetro, así que el planteamiento del problema puede modelar este tipo de situaciones.

8.10.1. Formalización

Se trata, evidentemente, de un problema de optimización. ¿Podemos plantear toda solución factible como una secuencia de elementos simples? Un vector de enteros de talla N, $(x_1, x_2, \ldots, x_N)$, resulta adecuado. La componente i-ésima, x_i, es el número de unidades del recurso que asignamos a la actividad i. El vector debe satisfacer estas restricciones:

- la suma de sus componentes no puede ser superior a U;
- la componente i-ésima, para $1 \leq i \leq N$, debe estar comprendida entre 0 y m_i.

O sea,

$$X = \left\{ (x_1, x_2, \ldots, x_N) \in \mathbb{N}^N \;\middle|\; 0 \leq x_i \leq m_i, 1 \leq i \leq N; \quad \sum_{1 \leq i \leq N} x_i \leq U \right\}.$$

La función objetivo es

$$f((x_1, x_2, \ldots, x_N)) = \sum_{1 \leq i \leq N} v(i, x_i).$$

Deseamos obtener el vector que maximiza la función objetivo y conocer el valor máximo de la función.

8.10.2. Ecuación recursiva

Hemos de estudiar la función de descomposición de un problema en subproblemas. Consideremos qué posibilidades tenemos en la asignación de recursos a la última actividad: cualquier entero u entre 0 y m_N (ambos inclusive). Si disponíamos de U unidades, tras esa asignación nos quedarán $U - u$ unidades pendientes de asignación a las $N - 1$ primeras tareas.

Notemos con $P(i, u)$ el máximo beneficio que podemos obtener asignando u unidades del recurso a las i primeras actividades. La función objetivo es separable (es un sumatorio). Nos enfrentamos a una maximización y el operador de combinación de resultados es monótono no decreciente. Es fácil comprobar que

$$P(i,u) = \begin{cases} 0, & \text{si } i = 0 \text{ o } u = 0; \\ \max\limits_{0 \leq u' \leq \min(u, m_i)} P(i-1, u-u') + v(i,u'), & \text{si } i > 0 \text{ y } u > 0. \end{cases} \tag{8.13}$$

EJERCICIOS

26 ¿Es válida la ecuación recursiva si se puede obtener un beneficio de no asignar nada a una actividad? Si la respuesta es negativa, ¿puedes diseñar una ecuación recursiva válida?

8.10.3. Algoritmo iterativo

Obviaremos la implementación de los algoritmos recursivos sin y con memorización e iremos directamente al diseño de un algoritmo iterativo. El grafo de dependencias entre resultados es un grafo multietapa (véase la figura 8.36): hay $N + 1$ etapas, cada una de las cuales consta de $U + 1$ vértices a lo sumo, uno por cada posible valor del número de recursos pendientes de asignación. Podemos etiquetar los vértices con pares (i, u). Habrá una arista de la forma $((i - 1, u - u'), (i, u))$ para todo i entre 1 y N y para todo $0 \leq u' \leq m_i$. Esa arista representa la asignación de u' unidades del recurso a la actividad i con un beneficio $v(i, u')$ (véase la figura 8.37).

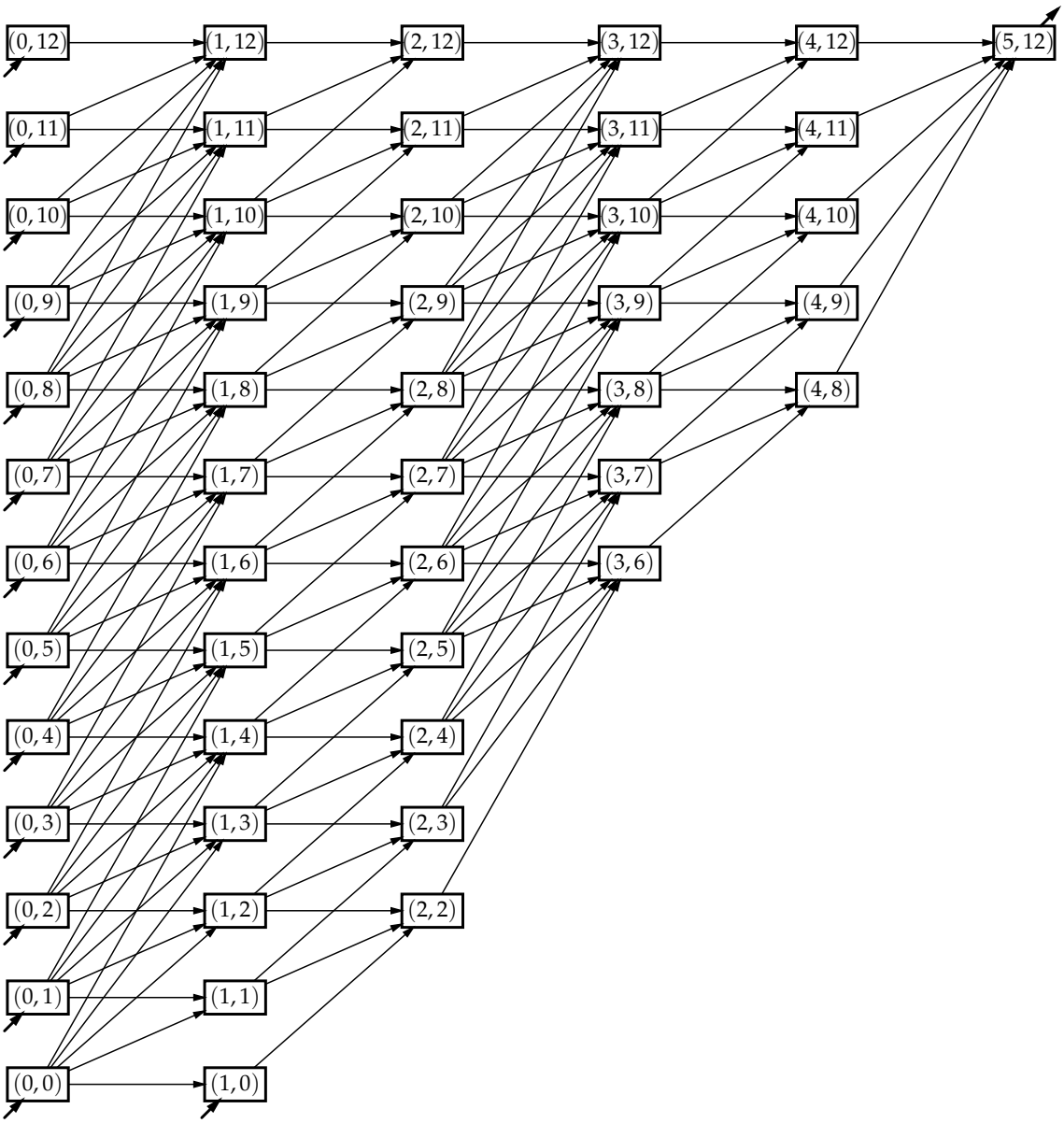


Figura 8.36: Grafo de dependencias para una instancia del problema de asignación de 12 unidades de un recurso a 5 actividades. Las actividades de índice par pueden recibir hasta 2 unidades del recurso, y las de índice impar, hasta 4.

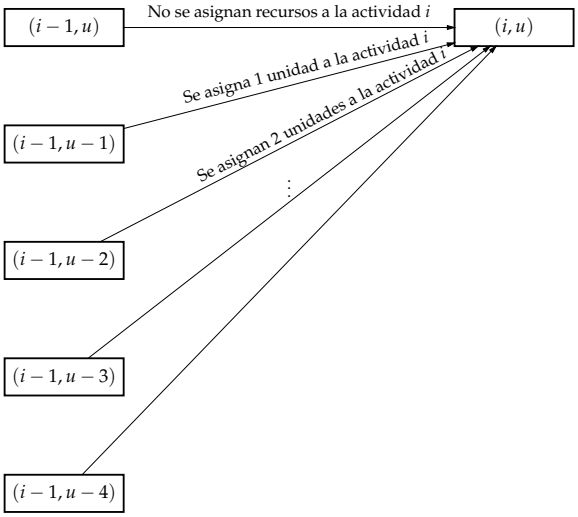


Figura 8.37: Interpretación de las aristas del grafo de dependencias en términos de la asignación de recursos a una actividad.

El grafo de dependencias para el algoritmo iterativo es, por simplicidad, un grafo extendido con $(N+1) \times (U+1)$ vértices. En esta implementación suministramos, además de los parámetros U y N, un vector con el mayor número de recursos asignable a una actividad y una función con el beneficio que obtenemos en una asignación concreta de recursos a una determinada actividad.

resource_allocation.py

```
1  def resource_allocation_profit(U, m, v):
2      P = {}
3      for u in xrange(U+1):
4          P[0, u] = 0
5      for i in xrange(1, len(m)+1):
6          P[i, 0] = 0
7          for u in xrange(1, U+1):
8              P[i, u] = max( P[i-1, u-u'] + v[i, u'] for u' in xrange(0, min(u, m[i])+1))
9      return P[len(m), U]
```

test_resource_allocation.py

```
1  from resource_allocation import resource_allocation_profit
2  from offsetarray import OffsetArray
3  from random import seed, randrange
4  seed(0)
5
6  U = 12
7  m = OffsetArray([2, 4, 2, 4, 2])
8  v = dict( ((i,u), randrange(100)) for i in xrange(1, len(m)+1) for u in xrange(0, U+1) )
9  print 'Beneficio:', resource_allocation_profit(U, m, v)
```

```
Beneficio: 409
```

El coste temporal del algoritmo es $O(NU^2)$. El coste espacial es $O(NU)$.

EJERCICIOS

27 Reduce la complejidad espacial a $O(U)$.

28 Diseña un algoritmo que indique cuántos recursos se asignan a cada actividad para obtener el beneficio máximo.

29 Diseña un algoritmo que resuelva el problema fijando un límite inferior y otro superior al número de recursos que podemos asignar a cada actividad.

30 Se quiere abonar C campos con M metros cúbicos de agua y N sacos de abono. La utilización de m metros cúbicos de agua y n sacos de abono en el campo c permite obtener una cosecha cuyo valor es $v(c, m, n)$. Desarrolla un algoritmo de programación dinámica que determine cuántos sacos de abono y metros cúbicos de agua hay que invertir en cada campo para obtener una cosecha de valor máximo. Analiza el algoritmo diseñado y calcula sus costes temporal y espacial. ¿Es posible reducir la complejidad espacial si sólo se desea conocer el beneficio que produce la mejor cosecha posible?

8.11. Segmentación de un texto en palabras

Dadas N palabras con ciertas probabilidades de aparición en un texto y dada una secuencia t de caracteres sin espacios en blanco, deseamos segmentar la secuencia de caracteres en la secuencia de palabras más probable.

Un ejemplo ayudará a entender lo solicitado. Supongamos que las palabras tienen una probabilidad de aparecer en el texto según se indica en esta tabla:

palabra	**probabilidad**	**palabra**	**probabilidad**
`a`	0.05	`aca`	0.01
`acas`	0.01	`ad`	0.01
`as`	0.03	`asa`	0.02
`ca`	0.01	`cadena`	0.06
`cas`	0.05	`casa`	0.08
`casaca`	0.05	`da`	0.05
`de`	0.05	`den`	0.02
`e`	0.02	`en`	0.11
`la`	0.12	`laca`	0.05
`lacas`	0.05	`na`	0.01
`nada`	0.07	`saca`	0.05
`sacad`	0.02	*cualquier otra*	0.00

Queremos segmentar la cadena `lacasacadenada` en una secuencia de palabras que formen una frase. Una posible segmentación es `la casaca de nada`. Su probabilidad es $\Pr(\texttt{la}) \cdot \Pr(\texttt{casaca}) \cdot \Pr(\texttt{de}) \cdot \Pr(\texttt{nada}) = 0.12 \cdot 0.05 \cdot 0.05 \cdot 0.07 = 0.000021$. Hay segmentaciones, como `l aca saca de nada` o `lac asa cadena da`, que tienen probabilidad 0 por contener «palabras» no incluidas en nuestro diccionario. La cadena `lacasacadenada` permite efectuar 43 segmentaciones con probabilidad no nula:

frase	**probabilidad**	**frase**	**probabilidad**
`la casa cadena da`	0.000028800000	`la cas a ca de nada`	0.000000010500
`la casaca de nada`	0.000021000000	`laca sacad e na da`	0.000000010000
`laca saca de nada`	0.000008750000	`la ca sacad en a da`	0.000000006600
`lacas a cadena da`	0.000007500000	`lacas aca den ad a`	0.000000005000
`lacas aca de nada`	0.000001750000	`la casa ca den a da`	0.000000004800
`laca sacad e nada`	0.000001400000	`la cas aca den a da`	0.000000003000
`la cas a cadena da`	0.000000900000	`la ca saca den a da`	0.000000003000
`la casa ca de nada`	0.000000336000	`la casa ca de na da`	0.000000002400
`la casaca den a da`	0.000000300000	`la cas aca de na da`	0.000000001500
`laca sacad en a da`	0.000000275000	`la ca saca de na da`	0.000000001500
`la cas aca de nada`	0.000000210000	`la ca sacad en ad a`	0.000000001320
`la ca saca de nada`	0.000000210000	`lacas a ca den a da`	0.000000001250
`la casaca de na da`	0.000000150000	`la casa ca den ad a`	0.000000000960
`laca saca den a da`	0.000000125000	`lacas a ca de na da`	0.000000000625
`lacas a ca de nada`	0.000000087500	`la cas aca den ad a`	0.000000000600
`laca saca de na da`	0.000000062500	`la ca saca den ad a`	0.000000000600
`la casaca den ad a`	0.000000060000	`lacas a ca den ad a`	0.000000000250
`laca sacad en ad a`	0.000000055000	`la ca sacad e na da`	0.000000000240
`la ca sacad e nada`	0.000000033600	`la cas a ca den a da`	0.000000000150
`lacas aca den a da`	0.000000025000	`la cas a ca de na da`	0.000000000075
`laca saca den ad a`	0.000000025000	`la cas a ca den ad a`	0.000000000030
`la cas a ca den ad a`	0.000000000030		

La segmentación `la casa cadena da` es la más probable de todas.

Deseamos calcular la segmentación más probable y conocer el valor de su probabilidad.

8.11.1. Formalización

Podemos representar una segmentación como una secuencia de marcas numéricas que indican donde empieza/acaba cada palabra. En la figura 8.38 se muestra una segmentación de $t =$`lacasacadenada` que puede representarse mediante la secuencia de enteros $(0, 2, 8, 10, 14)$.

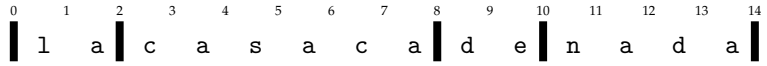


Figura 8.38: La segmentación de `lacasacadenada` *en* `la casaca de nada` *puede representarse con la secuencia de posiciones de marcas* $(0, 2, 8, 10, 14)$.

Toda segmentación es una secuencia de valores crecientes entre 0 y $|t|$ y viceversa. Así pues, el conjunto de soluciones factibles es

$$X = \left\{(s_0, s_1, \ldots, s_n) \in [0..|t|]^+ \mid s_0 = 0; \quad s_n = |t|; \quad s_{i-1} < s_i, \quad 1 \leq i \leq n\right\}.$$

La función objetivo, cuyo valor deseamos maximizar, es

$$f((s_0, s_1, \ldots, s_n)) = \prod_{1 \leq i \leq n} \Pr(t_{s_{i-1}:s_i}),$$

> **!** Este problema presenta interés, por ejemplo, en reconocimiento automático del habla. Al hablar, no pronunciamos pausas entre palabras. Un reconocedor puede devolver la secuencia de fonemas pronunciadas como una cadena de texto, sin indicación alguna de pausas que separen palabras. Una etapa del reconocedor debe segmentar correctamente la cadena en palabras, pero ocurre que hay pronunciaciones con más de una segmentación válida. Se hace necesario, pues, encontrar la idónea. Otro campo de aplicación es la corrección automática de textos en software ofimático u obtenidos con una herramienta de OCR.

donde $t_{i:j}$ es la cadena $t_i t_{i+1} \dots t_{j-1}$ y asumimos que el primer carácter de una cadena tiene índice 0.

8.11.2. Ecuación recursiva

Denotemos con $P(|t|)$ a la probabilidad máxima con que podemos segmentar la cadena $t_{0:|t|}$. Tenemos

$$P(|t|) = \max_{(s_0,s_1,\dots,s_n)\in X} \prod_{1\le i\le n} \Pr(t_{s_{i-1}:s_i}).$$

La última marca tiene un valor fijo, $s_n = |t|$. Nos planteamos, pues, qué valor puede adoptar la penúltima marca. En principio puede tomar cualquier valor entre 0 y $|t|-1$, es decir,

$$\begin{aligned} P(|t|) &= \max_{0\le a\le |t|-1} \left(\max_{\substack{(s_0,s_1,\dots,s_{n-1}):\\ s_0=0; s_{n-1}=a; s_{i-1}<s_i, 1\le i<n}} \left(\left(\prod_{1\le i\le n-1} \Pr(t_{s_{i-1}:s_i}) \right) \cdot \Pr(t_{a:|t|}) \right) \right) \\ &= \max_{0\le a\le |t|-1} \left(\left(\max_{\substack{(s_0,s_1,\dots,s_{n-1}):\\ s_0=0; s_{n-1}=a; s_{i-1}<s_i, 1\le i<n}} \left(\prod_{1\le i\le n-1} \Pr(t_{s_{i-1}:s_i}) \right) \right) \cdot \Pr(t_{a:|t|}) \right) \\ &= \max_{0\le a\le |t|-1} (P(a) \cdot \Pr(t_{a:|t|})). \end{aligned}$$

Podemos considerar $P(0)$ como un caso base con valor 1.

La ecuación recursiva es, pues,

$$P(j) = \begin{cases} 1, & \text{si } j = 0; \\ \max\limits_{0\le i\le j-1} (P(i) \cdot \Pr(t_{i:j})), & \text{si } j > 0. \end{cases} \tag{8.14}$$

El valor buscado es $P(|t|)$.

El grafo de dependencias entre resultados de llamadas recursivas para $t =$ `lacasacadenada` se muestra en la figura 8.39. La figura 8.40 muestra una arista del grafo y la probabilidad que le corresponde en términos del segmento que describe.

8.11.3. Algoritmo iterativo

Podemos calcular la probabilidad asociada a la mejor segmentación así:

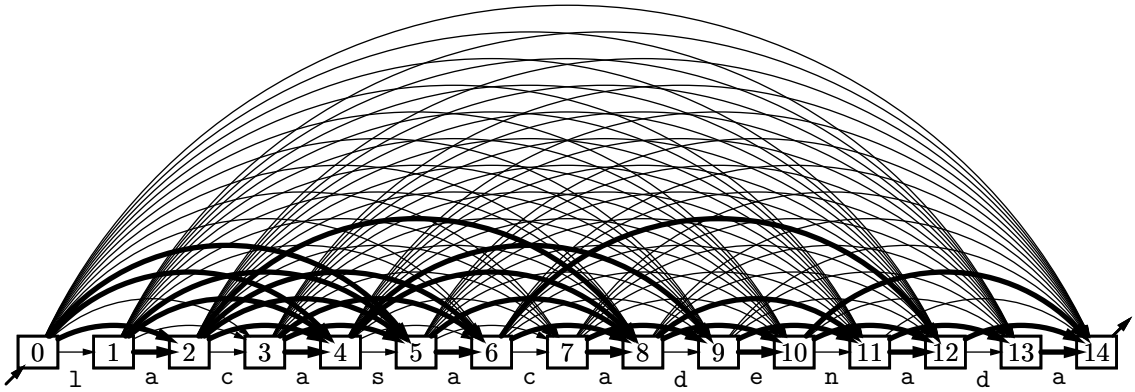


Figura 8.39: Grafo de dependencias para el problema de la segmentación de la cadena `lacasacadenada`*. Los arcos con probabilidad no nula se muestran con trazo grueso.*

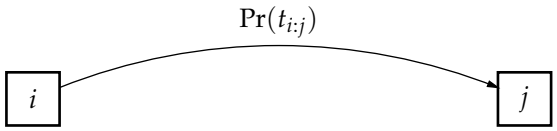


Figura 8.40: Interpretación de una arista del grafo de dependencias en términos del segmento de texto que describe.

> **!** Recuerda que el método *get* de un diccionario permite acceder al valor asociado a una clave si ésta existe; si no existe, se devuelve el valor del segundo parámetro, que actúa como valor por defecto.

segmentate.py

```
1  def probability(t, Pr):
2      P = [0] * (len(t)+1)
3      P[0] = 1
4      for j in xrange(1,len(t)+1):
5          P[j] = max(P[i] * Pr.get(t[i:j], 0) for i in xrange(j))
6      return P[len(t)]
```

test_segmentate_a.py

```
1  from segmentate import probability
2
3  Pr = {'a': 0.05,     'aca': 0.01,    'acas': 0.01,  'ad': 0.01,
4        'asa': 0.02,   'as': 0.03,     'la': 0.12,    'laca': 0.05,
5        'lacas': 0.05, 'ca': 0.01,     'cadena': 0.06,'cas': 0.05,
6        'casa': 0.08,  'casaca': 0.05,'da': 0.05,    'de': 0.05,
7        'den': 0.02,   'e': 0.02,      'en': 0.11,    'na': 0.01,
8        'nada': 0.07,  'saca': 0.05,   'sacad': 0.02}
9  t = 'lacasacadenada'
10 print 'Probabilidad de la mejor segmentación de %s: %10.8f.' % (t, probability(t, Pr))
```

```
Probabilidad de la mejor segmentación de lacasacadenada: 0.00002880.
```

El coste del algoritmo es $O(|t|^2)$. No hay forma de reducir la complejidad espacial, que es $O(|t|)$.

8.11.4. Recuperación de la segmentación óptima

Podemos plantearnos ya la recuperación de la segmentación óptima. En lugar de devolver una lista con los enteros que indica dónde se efectúa la separación en palabras, devolveremos una cadena con las palabras separadas por blancos:

segmentate.py

```
def segmentation(t, Pr):
    P = [0] * (len(t)+1)
    P[0] = 1
    B = [None] * (len(t)+1)
    for j in xrange(1,len(t)+1):
        P[j], B[j] = max( (P[i] * Pr.get(t[i:j], 0), i) for i in xrange(j) )
    sentence = []
    j = len(t)
    while B[j] != None:
        sentence.append( t[B[j]:j] )
        j = B[j]
    sentence.reverse()
    return ' '.join(sentence)
```

EJERCICIOS

31 Si tenemos en cuenta que la palabra más larga del diccionario está formada por W caracteres, podemos diseñar un algoritmo más eficiente. Modifica la ecuación recursiva para que tenga en cuenta esta información, diseña un nuevo algoritmo y analiza su complejidad computacional.

32 Dados los números naturales N y K, una descomposición de N en K sumandos positivos es una secuencia $n_1, n_2, \ldots, n_K$, tal que

$$N = \sum_{1 \le k \le K} n_k .$$

Se desea obtener la descomposición de N en K sumandos de modo que sea máximo su producto

$$\prod_{1 \le k \le K} n_k .$$

Por ejemplo, si $N = 6$ y $K = 3$ tenemos las siguientes descomposiciones posibles: $1+1+4$, $1+2+3$ y $2+2+2$. El producto de los sumandos es, respectivamente, $1 \cdot 1 \cdot 4 = 4$, $1 \cdot 2 \cdot 3 = 6$ y $2 \cdot 2 \cdot 2 = 8$. La descomposición óptima es, pues, $2+2+2$.

¿Es necesario recurrir a la programación dinámica para efectuar el cálculo de la descomposición óptima?

33 Dados N puntos de la recta real $x_1 < x_2 < \cdots < x_N$, se quiere encontrar una agrupación en M grupos *disjuntos* de puntos *consecutivos* $C_1, C_2, \ldots, C_M$ (de al menos un punto por grupo) tal que $\{x_1, x_2, \ldots, x_n\} = C_1 \cup C_2 \cup \ldots \cup C_M$ y se maximice la función de bondad del agrupamiento

$$\sum_{m=1}^{M} \Phi(i_m, j_m)$$

donde $\Phi(i_m, j_m)$ es una cierta medida de bondad del m-ésimo grupo, siendo i_m y j_m los extremos de dicho grupo ($1 \leq i_m \leq j_m \leq N$).

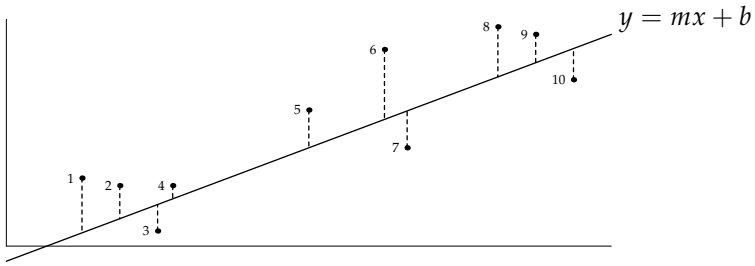
Se pide:

a) Diseñar un algoritmo que calcule la función de bondad de la mejor agrupación de los N puntos en M grupos.

b) Diseñar un algoritmo que obtenga (los extremos de) cada uno de los grupos de la mejor agrupación.

c) Análisis de complejidad de los algoritmos obtenidos, suponiendo que $\Phi(i_m, j_m)$ se puede calcular en tiempo $O(1)$.

34 Dada una serie de puntos en el plano (x_1, y_1), (x_2, y_2), $\ldots$, (x_n, y_n) podemos encontrar una recta $y = ax + b$ cuya suma de los cuadrados de las distancias verticales de los puntos a la recta sea mínima.

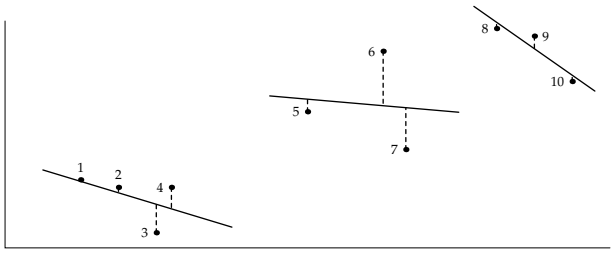


La recta $y = ax + b$ tiene estos parámetros:

$$a = \frac{\sum_{1\leq i\leq n} x_i y_i - \sum_{1\leq i\leq n} x_i \sum_{1\leq i\leq n} y_i}{n\sum_{1\leq i\leq n} x_i^2 - (\sum_{1\leq i\leq n} x_i)^2}, \qquad b = \frac{\sum_{1\leq i\leq n} y_i - a\sum_{1\leq i\leq n} x_i}{n}.$$

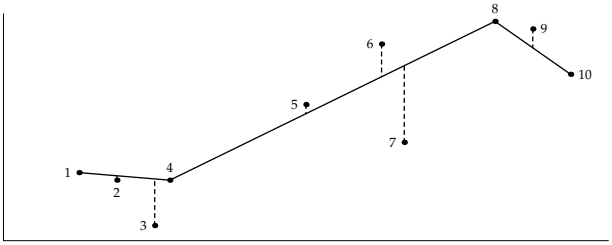
Dados N puntos en el plano real, (x_1, y_1), (x_2, y_2), $\ldots$, (x_N, y_N), tales que $x_1 < x_2 < \cdots < x_N$, deseamos encontrar la segmentación óptima en M grupos disjuntos de puntos contiguos de modo que la suma de los cuadrados de las distancias verticales a sus respectivas rectas de ajuste sean mínimas. Cada grupo debe contener al menos dos puntos.

He aquí un ejemplo de segmentación de 10 puntos en 3 grupos cuyos respectivos conjuntos de índices son $\{1,2,3,4\}$, $\{5,6,7\}$ y $\{8,9,10\}$:



Se pide un algoritmo tan eficiente como sea posible para calcular la segmentación óptima.

35 Dados N puntos en el plano real, (x_1, y_1), (x_2, y_2), $\ldots$, (x_N, y_N), tales que $x_1 < x_2 < \cdots < x_N$, deseamos encontrar la segmentación óptima en M grupos de puntos contiguos, cada uno de los cuales debe tener al menos 2 puntos. El último punto de cada grupo es el primero del siguiente grupo. Deseamos obtener la segmentación en M grupos como los descritos que hace mínimo el promedio del cuadrado de la distancia vertical a la línea que une los puntos inicial y final de cada grupo. He aquí un ejemplo de segmentación de 10 puntos en 3 grupos cuyos respectivos conjuntos de índices son $\{1,2,3,4\}$, $\{4,5,6,7,8\}$ y $\{8,9,10\}$:



Se pide un algoritmo tan eficiente como sea posible para calcular la segmentación óptima.

36 Modifica la solución dada al ejercicio anterior para que se minimice la suma de cuadrados de las distancias de cada punto a su correspondiente recta. La distancia de un punto (x', y') a la recta $ax + by + c = 0$ es $|ax' + by' + c|/\sqrt{a^2 + b^2}$.

37 Disponemos de una serie de recursos, $r_1, r_2, \ldots, r_R$, que podemos asignar a $d_1, d_2, \ldots, d_D$ actividades, con $R \geq D$. Podemos asignar una serie de recursos contiguos a una misma actividad. Si asignamos los recursos $r_i, r_{i+1}, \ldots, r_j$ a la actividad d_l, asignaremos los recursos $r_{j+1}, r_{j+2}, \ldots, r_k$ a la actividad d_{l+1}, donde $i \leq j < k$. No podemos dejar recurso alguno sin asignar a alguna actividad y cada actividad ha de recibir al menos uno de los recursos. La asignación de $r_i, r_{i+1}, \ldots, r_j$ a la actividad d_l ofrece un beneficio $v(i, j, l)$. Deseamos encontrar la asignación de máximo beneficio.

38 Disponemos de una serie de recursos, $r_1, r_2, \ldots, r_R$, que podemos asignar a $d_1, d_2, \ldots, d_D$ actividades, con $R \geq D$. Podemos asignar una serie de recursos contiguos a una misma actividad. Si asignamos los recursos $r_i, r_{i+1}, \ldots, r_j$ a la actividad d_l, podemos asignar los recursos $r_{i'}, r_{i'+1}, \ldots, r_{j'}$ a la actividad d_{l+1}, donde $i \leq j < i' < j'$. Nótese que es posible no asignar algunos recursos a actividad alguna, pero a toda actividad debe asignársele al menos un recurso. La asignación de $r_i, r_{i+1}, \ldots, r_j$ a la actividad d_l ofrece un beneficio $v(i, j, l)$. Deseamos encontrar la asignación de máximo beneficio.

8.12. Formateo de párrafos

En un procesador de textos que estamos diseñando deseamos obtener párrafos estéticamente aceptables. Cada párrafo se nos proporciona como una secuencia de N palabras, $w_0, w_1, \ldots, w_{N-1}$, de longitudes respectivas $|w_0|, |w_1|, \ldots, |w_{N-1}|$ (en caracteres). Deseamos justificar el párrafo limitando el ancho de todas las líneas a L caracteres.

He aquí un ejemplo de justificación a 40 columnas de un párrafo con el texto «ésta es la frase de ejemplo que contiene veintiocho palabras y usamos al ilustrar el resultado de un algoritmo de división en líneas justificadas y de anchura fija».

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
é s t a   e s   l a   f r a s e   d e   e j e m p l o   q u e   c o n t i e n e
v e i n t i o c h o   p a l a b r a s   y   u s a m o s   a l   i l u s t r a r
e l   r e s u l t a d o   d e   u n   a l g o r i t m o   d e   d i v i s i ó n
e n   l í n e a s   j u s t i f i c a d a s   y   d e   a n c h u r a   f i j a
```

Hemos tenido suerte y un solo espacio entre cada par de palabras ha permitido obtener una justificación perfecta. Pero no siempre ocurre así. Si en un línea disponemos las palabras $w_i, w_{i+1}, \ldots, w_{j-1}$, hemos de incluir al menos $j - i - 1$ espacios en blanco para

separarlas visualmente. El mínimo número de espacios que ocupan las palabras w_i, w_{i+1}, …, w_{j-1} se denotará con $l(i,j) = j - i - 1 + \sum_{i \le k < j} |w_k|$. Si $l(i,j)$ es una cantidad estrictamente inferior a L, la justificación no será perfecta y tendremos que añadir $L - l(i,j)$ espacios en blanco a la línea. Esos espacios en blanco, que afean la línea, tendrán que distribuirse a partes (tan) iguales (como sea posible) entre cada par de palabras.

Si, por ejemplo, formateamos el mismo texto del ejemplo con un ancho de línea de 30 caracteres, podemos obtener este resultado:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
é s t a                 e s                 l a                 f r a s e             d e
e j e m p l o                               q u e                       c o n t i e n e
v e i n t i o c h o           p a l a b r a s       y   u s a m o s
a l   i l u s t r a r       e l   r e s u l t a d o   d e   u n
a l g o r i t m o           d e             d i v i s i ó n             e n
l í n e a s               j u s t i f i c a d a s               y           d e
a n c h u r a                                                       f i j a
```

Una gran cantidad de espacios en blanco en una línea afea considerablemente el texto.

> **!** El texto que estás leyendo está formateado con LaTeX, un paquete de macros para el sistema TeX. El sistema de composición tipográfica TeX, diseñado por Donald E. Knuth, utiliza una versión elaborada del algoritmo que presentamos en este apartado, así que cada uno de los párrafos que ves ha sido procesado por un algoritmo de programación dinámica. El problema que resuelve TeX es más complicado: considera letras de anchura variable y espacios elásticos. Es destacable la definición de heurísticos para conseguir que el texto formateado sea estéticamente aceptable. El capítulo 14 del libro "The TeXbook" se dedica a este problema.

La última línea usa 19 espacios para separar dos palabras. Podemos definir una función $p(b)$ que proporcione una medida de la «penalización estética» asociada al uso de b blancos «extra» para separar n palabras. La penalización de un párrafo es la suma de penalizaciones de sus líneas. A título de ilustración, supongamos una función de penalización $p(b)$ definida así

$$p(b) = \begin{cases} +\infty, & \text{si } b < 0; \\ b^3, & \text{en otro caso.} \end{cases}$$

El párrafo anterior tendría una penalización $11^3 + 10^3 + 2^3 + 0^3 + 6^3 + 6^3 + 18^3 = 8603$.

Consideremos ahora este otro formateo del mismo párrafo con idéntica anchura de línea:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
é s t a       e s         l a         f r a s e   d e     e j e m p l o
q u e                   c o n t i e n e       v e i n t i o c h o
p a l a b r a s       y   u s a m o s   a l     i l u s t r a r
e l     r e s u l t a d o       d e     u n   a l g o r i t m o
d e       d i v i s i ó n             e n             l í n e a s
j u s t i f i c a d a s   y   d e     a n c h u r a     f i j a
```

La penalización de este párrafo es $3^3 + 7^3 + 1^3 + 2^3 + 9^3 + 0^3 = 1108$, menor que la del anterior párrafo.

Deseamos diseñar un algoritmo que encuentre la asignación óptima de palabras a líneas en el sentido de obtener la más baja penalización posible.

8.12.1. Formalización

Se trata de un problema de segmentación, similar al que hemos resuelto en el apartado anterior. Queremos segmentar la secuencia de palabras en una serie de grupos de modo que cierta función objetivo alcance un valor mínimo. La figura 8.41 ilustra la segmentación de un texto en seis partes.

$|_0$ `ésta` $_1$ `es` $_2$ `la` $_3$ `frase` $_4$ `de` $_5$ `ejemplo` $|_6$ `que` $_7$ `contiene` $_8$ `veintiocho` $|_9$ `palabras` $_{10}$ `y` $_{11}$ `usamos` $_{12}$ `al` $_{13}$ `ilustrar` $|_{14}$ `el` $_{15}$ `resultado` $_{16}$ `de` $_{17}$ `un` $_{18}$ `algoritmo` $|_{19}$ `de` $_{20}$ `división` $_{21}$ `en` $_{22}$ `líneas` $|_{23}$ `justificadas` $_{24}$ `y` $_{25}$ `de` $_{26}$ `anchura` $_{27}$ `fija` $|_{28}$

Figura 8.41: Representación de la segmentación $(0, 6, 9, 14, 19, 23, 28)$ *de la frase de ejemplo, que se muestra formateada como párrafo con anchura de 30 caracteres al final del último apartado.*

Si hay N palabras, buscamos una secuencia de enteros $(s_0, s_1, s_2, \ldots, s_n)$ tal que $s_0 = 0$, $s_n = N$ y, además, $s_{i-1} < s_i$ y $l(s_{i-1}, s_i) \leq L$ para todo i entre 1 y n. Las palabras de índice comprendido entre s_{i-1} y $s_i - 1$ forman parte de la línea i-ésima.

La función objetivo es

$$f((s_0, s_1, s_2, \ldots, s_n)) = \sum_{1 \leq i \leq n} p(L - l(s_{i-1}, s_i)).$$

8.12.2. Ecuación recursiva

El valor de s_n es siempre N. Hemos de plantearnos qué valores puede tomar s_{n-1}: cualquier valor entre 0 y $N - 1$ tal que $l(s_{n-1}, s_n)$ sea menor o igual que L.

La ecuación recursiva que resuelve el problema es ésta:

$$P(j) = \begin{cases} 0, & \text{si } j = 0; \\ \min\limits_{\substack{0 \leq i \leq j-1: \\ l(i,j) \leq L}} (P(i) + p(L - l(i,j))), & \text{en otro caso.} \end{cases} \tag{8.15}$$

La penalización mínima es $P(N)$.

EJERCICIOS

39 Demuestra la validez de la ecuación recursiva propuesta.

40 Dibuja el grafo de dependencias para formatear la frase «`un ejemplo más para formatear con el algoritmo de justificación de párrafos`» con un ancho de línea de 20 caracteres.

8.12.3. Algoritmo iterativo

El grafo de dependencias cuya representación se solicita en el último ejercicio revela que basta con un recorrido de menor a mayor valor del parámetro *j*, el índice con el que recorremos las palabras del texto:

paragraph.py

```
def p(b):
    if b < 0: return 2**31
    else:     return b**3

def minimum_penalty(w, L, p=p):
    P = [0] * (len(w)+1)
    for j in xrange(1, len(w)+1):
        chars = len(w[j-1])
        P[j] = P[j-1] + p(L-chars)
        for i in xrange(j-2, -1, -1):
            chars += 1+len(w[i])
            if chars > L: break
            P[j] = min(P[j], P[i] + p(L-chars))
    return P[len(w)]
```

test_paragraph_a.py

```
from paragraph import minimum_penalty

L = 30
t = "ésta es la frase de ejemplo que contiene veintiocho palabras y usamos al ilustrar"+\
    " el resultado de un algoritmo de división en líneas justificadas y de anchura fija"
print 'Penalización mínima para "%s" y %d columnas: %d' % (t, L, minimum_penalty(t.split(), L))
```

```
Penalización mínima para "ésta es la frase de ejemplo que contiene veintiocho pa
labras y usamos al ilustrar el resultado de un algoritmo de división en líneas j
ustificadas y de anchura fija" y 30 columnas: 1108
```

8.12.4. Recuperación de la segmentación óptima

Esta otra versión recupera la segmentación óptima en líneas y muestra el párrafo formateado resultante:

paragraph.py *(cont.)*

```
def justify_paragraph(w, L, p=p):
    P = [0] * (len(w)+1)
    B = [None] * (len(w)+1)
    for j in xrange(1, len(w)+1):
        chars = len(w[j-1])
        P[j], B[j] = P[j-1] + p(L-chars), j-1
        for i in xrange(j-2, -1, -1):
            chars += 1+len(w[i])
```

```
24              if chars > L: break
25              P[j], B[j] = min( (P[j],B[j]), (P[i] + p(L-chars), i) )
26      par = []
27      j = len(w)
28      while B[j] != None:
29          par.append( w[B[j]:j] )
30          j = B[j]
31      par.reverse()
32      fmt_lines = []
33      for line in par:
34          linechars = len(''.join(line))
35          blanks = L-linechars
36          if len(line) > 1: sep, extra =  blanks / (len(line)-1), blanks % (len(line)-1)
37          else: sep, extra = 1, 0
38          if extra:
39              fmt_lines.append((' '*(sep+1)).join(line[:extra])+' '*(sep+1)+(' '*sep).join(line[extra:]))
40          else:
41              fmt_lines.append( (' '*sep).join(line[extra:]) )
42      return '\n'.join(fmt_lines)
```

test_paragraph_b.py

```
1  from paragraph import justify_paragraph, minimum_penalty
2
3  L = 30
4  t = "ésta es la frase de ejemplo que contiene veintiocho palabras y usamos al ilustrar"+\
5      " el resultado de un algoritmo de división en líneas justificadas y de anchura fija"
6  for L in [40, 30, 20]:
7      print 'Formateado con %d columnas. Penalización %d.' % (L, minimum_penalty(t.split(), L))
8      print '-' * L
9      print justify_paragraph(t.split(), L)
10     print
```

```
Formateado con 40 columnas. Penalización 0.
----------------------------------------
ésta es la frase de ejemplo que contiene
veintiocho palabras y usamos al ilustrar
el resultado de un algoritmo de división
en líneas justificadas y de anchura fija

Formateado con 30 columnas. Penalización 1108.
------------------------------
ésta  es  la  frase de ejemplo
que     contiene    veintiocho
palabras  y usamos al ilustrar
el  resultado  de un algoritmo
de    división    en    líneas
justificadas y de anchura fija

Formateado con 20 columnas. Penalización 601.
--------------------
ésta  es la frase de
```

```
ejemplo que contiene
veintiocho  palabras
y usamos al ilustrar
el    resultado   de
un    algoritmo   de
división  en  líneas
justificadas       y
de    anchura   fija
```

El coste temporal del algoritmo es $O(NL)$ y el coste espacial es $O(N)$.

. . . . . EJERCICIOS . . . . .

41 También podemos justificar el texto con un algoritmo voraz: en una línea caben tantas palabras como sea posible. El último de los ejemplos hubiera sido formateado así con un algoritmo voraz:

```
Formateado con 20 columnas
--------------------
ésta  es la frase de
ejemplo que contiene
veintiocho  palabras
y usamos al ilustrar
el  resultado  de un
algoritmo         de
división  en  líneas
justificadas   y  de
anchura         fija
```

Escribe un programa que evalúe, para diferentes valores del número de columnas, la calidad estética (con la fórmula que estamos usando) de la solución encontrada por el algoritmo voraz y compare su valor con el que proporciona la justificación basada en programación dinámica.

42 Reduce la complejidad espacial para el caso en que sólo estemos interesados en conocer el valor de la menor penalización que podemos obtener.

43 Un verdadero algoritmo de justificación de párrafos dejaría sin justificar la última línea. El párrafo que venimos usando como ejemplo podría formatearse así:

```
Formateado con 20 columnas
--------------------
ésta  es la frase de
ejemplo que contiene
veintiocho  palabras
y usamos al ilustrar
el  resultado  de un
algoritmo         de
división  en  líneas
justificadas   y  de
anchura fija
```

Modifica el algoritmo para que obtenga la justificación del párrafo de penalización mínima sin justificar la última línea.

44 Dados dos enteros L_1 y L_2, queremos saber cuál es el número de columnas L comprendido entre L_1 y L_2 con el que podemos obtener el formateo de mínima penalización. Analiza el coste del algoritmo que toma esta decisión.

45 Modifica el algoritmo de justificación óptima de párrafos para que pueda indicarse explícitamente el número de líneas que deben formar el párrafo. Si el número de líneas es demasiado pequeño o demasiado grande como para que sea posible efectuar la justificación, el algoritmo lo indicará de algún modo.

46 Puede resultar mejor dividir un párrafo en líneas partiendo la última palabra y poniendo un guión. Consideremos nuevamente la frase de ejemplo:

```
Formateado con 20 columnas
--------------------
ésta  es la frase de
ejemplo que contiene
veintiocho  palabras
y usamos al ilustrar
el  resultado  de un
algoritmo  de  divi-
sión en  líneas jus-
tificadas  y  de an-
chura fija
```

Lenguas como el castellano, con reglas ortográficas muy regulares, facilitan enormemente la

> **!** El algoritmo de partición de palabras se presenta en el apéndice H del libro "The TEXbook", de Donald E. Knuth y es obra de Frank M. Liang, quien lo presentó en 1983 en su tesis doctoral "Word Hy-phen-a-tion by Com-pu-ter".

determinación de los puntos donde es lícito partir una palabra. TEX usa un algoritmo de partición de palabras que produce buenos resultados. El algoritmo utiliza un fichero en el que se indican las particiones válidas en secuencias de letras concretas. Un valor par entre dos letras indica que no es posible efectuar una partición y un valor impar, que sí. Cualquier secuencia de letras no recogida en la tabla implica un valor nulo entre cualquiera de sus pares de letras consecutivas. He aquí un resumen de la tabla para el castellano (puedes acceder al fichero con su contenido completo):

> **!** La tabla ha sido adaptada de la que creó José A. Mañas y que presentó en el artículo "On Word Division in Spanish", publicado 1987 en *Communications of the ACM*.

La primera entrada de la tabla, por ejemplo, se interpreta como $_1$`b`$_0$`a`$_0$ e indica que no es

> **!** Un fichero análogo a éste para el inglés contiene cerca de 4 500 reglas, cuando el fichero para el español no llega a 500.

posible poner un guión entre la `b` y la `a` pero, en principio, sí antes de la `b`. Las últimas líneas del fichero indican la prohibición de partir palabras de modo que el final de la línea actual o el principio de la siguiente línea corresponda a una palabra malsonante.

Desarrollemos el resto de la exposición con un ejemplo concreto. Tomemos por ejemplo la palabra «`palabra`». El método añade marcas de inicio y final (puntos) para obtener la cadena `.palabra.` y genera toda posible subsecuencia de dos o más caracteres: `.p`, `pa`, `al`, `la`, `ab`, `br`, `ra`, `a.`, `.pa`, `pal`, `ala`, `lab`, `abr`, `bra`, `ra.`, `.pal`, `pala`, `alab`, `labr`, `abra`, `bra.`, `.pala`, `palab`, `alabr`,

`labra`, `abra.`, `.palab`, `palabr`, `alabra`, `labra.`, `.palabr`, `palabra`, `alabra.`, `.palabra`, `palabra.` y `.palabra.`.

A continuación, busca cada subcadena en la tabla. En el caso de `palabra`, encuentra las subcadenas `pa`, `la`, `bra` y `ra`. Cada una de ellas incorpora una secuencia de números: $_1$`p`$_0$`a`$_0$, $_1$`l`$_0$`a`$_0$, $_1$`b`$_2$`r`$_0$`a`$_0$, $_1$`r`$_0$`a`$_0$. Finalmente, integra todas las subsecuencias numeradas en una sola para numerar la palabra original. Si en un punto confluyen dos o más valores diferentes, sólo se tiene en cuenta el mayor de ellos. El resultado de numerar la palabra original es $_1$`p`$_0$`a`$_1$`l`$_0$`a`$_1$`b`$_2$`r`$_0$`a`$_0$. Los números impares indican lugares por los que es posible partir la palabra. Podemos, pues, partir `palabra` donde hemos dispuesto estos guiones: `pa-la-bra`.

Te proporcionamos el algoritmo de partición ya implementado:

`hyphenator.py`

```
def load_table(filename):
    table = {1:{}, 2: {}, 3:{}, 4:{}, 5:{}, 6:{}, 7:{}, 8:{}}
    for line in file(filename):
        if line.strip():
            tokens = line.split()
            table[len(tokens[0])][tokens[0]] = [int(n) for n in tokens[1:]]
    return table

def hyphenate(word, table):
    marked = '.%s.' % word
    values = [0] * (len(word)+2)
    for i in xrange(len(marked)+1):
        for j in xrange(i+2, len(marked)+1):
            if j-i in table and marked[i:j] in table[j-i]:
                for v in range(len(table[j-i][marked[i:j]])):
                    values[i-1+v] = max(values[i-1+v], table[j-i][marked[i:j]][v])
    hyphenated = ''
    for i in xrange(1, len(values)-1):
        hyphenated += marked[i]
        if values[i] %2 == 1: hyphenated += '-'
    return hyphenated
```

`test_hyphenator.py`

```
from hyphenator import load_table, hyphenate

table = load_table('spanish_hyphenation_table')
for word in 'ejemplo', 'supercalifragilísticoespialidoso',\
            'algorítmica', 'disputa', 'no', 'pie', 'arriba',\
            'país', 'aislado', 'penséis', 'guión', 'tenía',\
            'cárcel':
    print hyphenate(word, table)
```

```
e-jem-plo
su-per-ca-li-fra-gi-lís-ti-coes-pia-li-do-so
al-go-rít-mi-ca
dispu-ta
no
pie
a-rri-ba
```

```
país
ais-la-do
pen-séis
guión
te-nía
cár-cel
```

Adapta el algoritmo de justificación de párrafos para que haga uso de la posibilidad de particionar una línea teniendo en cuenta que las palabras pueden partirse cuando están al final de una línea.

47 No te hemos contado todo acerca del algoritmo de partición en palabras. Hay algunas palabras que no se parten correctamente con las reglas descritas. Recogemos algunas en este fichero (con su correcta partición):

Modifica el programa anterior para que considere estas excepciones a las reglas.

48 Diseña un algoritmo que, dados un texto (secuencia de palabras) y una lista de longitudes de línea, ajuste óptimamente el texto a dichas líneas.

He aquí un ejemplo ilustrativo en el que se ajusta un texto a líneas cuyas longitudes se especifican mediante la lista `[10, 12, 14, 16, 18, 20, 18, 16, 14, 12, 10]`:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
F o r m a t o   d e
p á r r a f o s   c o n
f o r m a l   o s   r a r i t a s
q u e   l o s   u s u a r i o s
p u e d e n   d e f i n i r   c o n
n ú m e r o s   q u e   s e   p a s a n
c o n   u n a   l i s t a   d e
P y t h o n ,   c o m o   v e s
e n   e s t e   c a s o
q u e   i l u s t r a
l a   i d e a .
```

Analiza el coste temporal y espacial del algoritmo.

8.13. Distancia de edición entre cadenas de caracteres

Dados un alfabeto Σ y dos cadenas, $x = x_1x_2 \ldots x_{|x|}$ e $y = y_1y_2 \ldots y_{|y|}$ de Σ^*, deseamos obtener una medida que nos indique cuán semejantes o diferentes son. La medida debe tener en cuenta el número de pasos elementales, a los que denominamos «operaciones de edición», que permiten transformar una cadena en otra.

Consideraremos tres posibles operaciones de edición:

- *inserción* de un símbolo $a \in \Sigma$, que notaremos con $\lambda \to a$ (la cadena vacía se denota con λ);
- *borrado* de un símbolo $b \in \Sigma$, que notaremos con $b \to \lambda$; y
- *sustitución* de un símbolo $a \in \Sigma$ por otro diferente $b \in \Sigma$, que notamos con $a \to b$.

Las operaciones de edición son elementos de $((\Sigma \cup \{\lambda\}) \times (\Sigma \cup \{\lambda\})) - \{(\lambda, \lambda)\}$ que modelan errores típicos, por ejemplo, en la transmisión de datos o la escritura mecanográfica. Si un escritor intenta teclear la palabra `masa` y escribe `asa`, comete un error de borrado ($\mathtt{m} \to \lambda$); si al tratar de escribir `masa` escribe `masai`, el error es una inserción ($\lambda \to \mathtt{i}$); y si al intentar teclear la misma palabra escribe `pasa`, el error es una sustitución ($\mathtt{m} \to \mathtt{p}$).

Una secuencia de operaciones de edición es una *transformación de edición*. Una cadena puede convertirse en cualquier otra mediante una transformación de edición. La palabra `ejemplo` se puede convertir, por ejemplo, en la palabra `campos` con la transformación de edición $\mathtt{e} \to \lambda$, $\mathtt{j} \to \lambda$, $\lambda \to \mathtt{c}$, $\mathtt{e} \to \mathtt{a}$, $\mathtt{l} \to \lambda$ y $\lambda \to \mathtt{s}$:

$$x = \texttt{ejemplo} \overset{\mathtt{e}\to\lambda}{\Rightarrow} \texttt{jemplo} \overset{\mathtt{j}\to\lambda}{\Rightarrow} \texttt{emplo} \overset{\lambda\to\mathtt{c}}{\Rightarrow} \texttt{cemplo} \overset{\mathtt{e}\to\mathtt{a}}{\Rightarrow} \texttt{camplo} \overset{\mathtt{l}\to\lambda}{\Rightarrow} \texttt{campo} \overset{\lambda\to\mathtt{s}}{\Rightarrow} \texttt{campos} = y$$

Levenshtein definió una medida de disimilitud entre cadenas basada en las trans-

> **!** La distancia de Levenshtein puede aplicarse, por ejemplo, a la corrección automática o asistida de textos. Si una palabra no se encuentra en una lista de palabras válidas, podemos calcular la distancia de Levenshtein de la palabra desconocida a todas las válidas para hallar la (o las) más próxima(s). Esta palabra puede reemplazar a la no encontrada o sugerir al usuario que la acepte como corrección de una posible falta de ortografía. Las aplicaciones de la distancia de Levenshtein (y toda la familia de distancias relacionadas con ella) van más allá de la corrección de textos: clasificación de patrones, reconocimiento del habla, visión artificial, recuperación de errores en la transmisión de información, detección de similitudes entre secuencias de genoma, ...

formaciones de edición y sobre la que podemos definir una métrica. La denominada «**distancia de Levenshtein**» entre dos cadenas x e y es el menor número de operaciones de edición con la que podemos transformar x en y, y la denotaremos con $D(x, y)$.

La distancia de Levenshtein entre `ejemplo` y `campos` es 5. Esta transformación de edición lo prueba:

$$x = \texttt{ejemplo} \overset{\mathtt{e}\to\mathtt{c}}{\Rightarrow} \texttt{cjemplo} \overset{\mathtt{j}\to\mathtt{a}}{\Rightarrow} \texttt{caemplo} \overset{\mathtt{e}\to\lambda}{\Rightarrow} \texttt{camplo} \overset{\mathtt{l}\to\lambda}{\Rightarrow} \texttt{campo} \overset{\lambda\to\mathtt{s}}{\Rightarrow} \texttt{campos} = y$$

Nótese que, a efectos del cálculo de la distancia de Levenshtein, sólo nos interesan transformaciones de edición en las que cada carácter de x y cada carácter de y participa, a lo sumo, en una operación de edición. Si deseamos transformar una `a` en una `b`, por ejemplo, sólo tiene interés la transformación directa $\mathtt{a} \to \mathtt{b}$. Otras, como la transformación $\mathtt{a} \to \mathtt{c}$, $\mathtt{c} \to \mathtt{b}$, o la transformación $\mathtt{a} \to \lambda$, $\lambda \to \mathtt{b}$ no deben tenerse en cuenta: requieren mayor número de operaciones de edición que la operación que efectúa la misma transformación directamente.

A efectos prácticos, vamos a enriquecer el juego de operaciones de edición con la operación de *concordancia*: la sustitución de un símbolo por sí mismo. El objetivo es hacer que todo símbolo de x y todo símbolo de y, incluso los que no se modifican, participen en una operación de edición.

Podemos extender el ejemplo anterior con el nuevo tipo de operación de edición:

$$x = \texttt{ejemplo} \overset{\texttt{e}\to\texttt{c}}{\Rightarrow} \texttt{cjemplo} \overset{\texttt{j}\to\texttt{a}}{\Rightarrow} \texttt{caemplo} \overset{\texttt{e}\to\lambda}{\Rightarrow} \texttt{camplo} \overset{\texttt{m}\to\texttt{m}}{\Rightarrow} \texttt{camplo} \overset{\texttt{p}\to\texttt{p}}{\Rightarrow} \texttt{camplo} \overset{\texttt{l}\to\lambda}{\Rightarrow} \texttt{campo} \overset{\texttt{o}\to\texttt{o}}{\Rightarrow} \texttt{campo} \overset{\lambda\to\texttt{s}}{\Rightarrow} \texttt{campos} = y$$

Nótese que todo carácter de x e y participa en una (y sólo una) operación de edición.

Naturalmente, no tiene sentido considerar que una concordancia sea un error. Definimos una función de ponderación de operaciones de edición así:

$$\gamma(a \to b) = \begin{cases} 0, & \text{si } a = b; \\ 1, & \text{en otro caso;} \end{cases}$$

donde a y b son símbolos de $\Sigma \cup \{\lambda\}$, pero sin ser ambos λ.

El peso de una transformación de edición $(e_1, e_2, \ldots, e_n)$ es

$$d((e_1, e_2, \ldots, e_n)) = \sum_{1 \le i \le n} \gamma(e_i).$$

Si $E_{x,y}$ es el conjunto de secuencias de edición entre x e y tal que todo carácter de x y todo carácter de y participa en una operación de edición, buscamos el menor peso de cualquiera de las secuencias en E:

$$D(x, y) = \min_{(e_1, e_2, \ldots, e_n) \in E_{x,y}} \sum_{1 \le i \le n} \gamma(e_i).$$

No explicitamos el conjunto de soluciones factibles. Si bien se puede formular en términos de las secuencias de edición válidas, llegaremos a una definición alternativa y recursiva que conduce a un algoritmo eficiente.

8.13.1. Ecuación recursiva

Notemos con $x_{1:i}$ a la subcadena $x_1 x_2 \ldots x_i$. Si i es cero, entonces $x_{1:i}$ es λ. Hemos de

> **!** En problemas anteriores hemos supuesto que el primer carácter de una cadena tiene índice 0 y que los cortes excluyen al carácter cuyo índice coincide con su segundo parámetro. En este problema asumiremos, por conveniencia, que el primer carácter tiene índice 1 y que los cortes incluyen al carácter cuyo índice es igual al segundo parámetro (salvo, naturalmente, en los programas Python).

encontrar la transformación de edición que haga mínimo el valor de la función objetivo. ¿Qué operación de edición puede aparecer en la última posición cuando transformamos $x_{1:i}$ en $y_{1:j}$? Recordemos que, al tener que participar todo carácter en alguna operación de edición, esta operación afectará a x_i, a y_j o a ambos. Tenemos tres posibilidades:

- que sea un borrado: $x_i \to \lambda$;
- que sea una inserción: $\lambda \to y_j$;
- o que sea una sustitución/concordancia: $x_i \to y_j$.

Estudiemos qué ocurre en cada caso:

- Si es un borrado, $x_i \to \lambda$, aún hemos de transformar $x_{1:i-1}$ en $y_{1:j}$.

- Si es una inserción, $\lambda \to y_j$, aún hemos de transformar $x_{1:i}$ en $y_{1:j-1}$.
- Si es una sustitución/concordancia, $x_i \to y_j$, aún hemos de transformar $x_{1:i-1}$ en $y_{1:j-1}$.

De entre las tres posibilidades, nos interesa la que conduce a un menor peso de la secuencia de edición completa de la que ella es el último elemento. Ya se apunta una formulación recursiva del problema. Derivémosla formalmente. Queremos calcular

$$D(x_{1:i}, y_{1:j}) = \min_{(e_1,e_2,\ldots,e_n)\in E_{x_{1:i},y_{1:j}}} \sum_{1\le k\le n} \gamma(e_k),$$

y e_n puede tomar tres valores, $x_i \to \lambda$, $\lambda \to y_j$ o $x_i \to y_j$:

$$D(x_{1:i}, y_{1:j}) = \min \left\{ \begin{array}{l} \displaystyle\min_{(e_1,e_2,\ldots,e_{n-1})\in E_{x_{1:i-1},y_{1:j}}} \left(\sum_{1\le k\le n-1} \gamma(e_k) + \gamma(x_i \to \lambda) \right), \\ \displaystyle\min_{(e_1,e_2,\ldots,e_{n-1})\in E_{x_{1:i},y_{1:j-1}}} \left(\sum_{1\le k\le n-1} \gamma(e_k) + \gamma(\lambda \to y_j) \right), \\ \displaystyle\min_{(e_1,e_2,\ldots,e_{n-1})\in E_{x_{1:i-1},y_{1:j-1}}} \left(\sum_{1\le k\le n-1} \gamma(e_k) + \gamma(x_i \to y_j) \right) \end{array} \right\}.$$

Estamos minimizando y el operación de combinación es la suma, así que

$$\begin{aligned} D(x_{1:i}, y_{1:j}) &= \min \left\{ \begin{array}{l} \left(\displaystyle\min_{(e_1,e_2,\ldots,e_{n-1})\in E_{x_{1:i-1},y_{1:j}}} \sum_{1\le k\le n-1} \gamma(e_k) \right) + \gamma(x_i \to \lambda), \\ \left(\displaystyle\min_{(e_1,e_2,\ldots,e_{n-1})\in E_{x_{1:i},y_{1:j-1}}} \sum_{1\le k\le n-1} \gamma(e_k) \right) + \gamma(\lambda \to y_j), \\ \left(\displaystyle\min_{(e_1,e_2,\ldots,e_{n-1})\in E_{x_{1:i-1},y_{1:j-1}}} \sum_{1\le k\le n-1} \gamma(e_k) \right) + \gamma(x_i \to y_j) \end{array} \right\} \\ &= \min \left\{ \begin{array}{l} D(x_{1:i-1}, y_{1:j}) + \gamma(x_i \to \lambda), \\ D(x_{1:i}, y_{1:j-1}) + \gamma(\lambda \to y_j), \\ D(x_{1:i-1}, y_{1:j-1}) + \gamma(x_i \to y_j) \end{array} \right\}. \end{aligned}$$

Podemos simplificar la notación si consideramos que x e y están implícitos:

$$D(i,j) = \min \left\{ \begin{array}{l} D(i-1,j) + \gamma(x_i \to \lambda), \\ D(i,j-1) + \gamma(\lambda \to y_j), \\ D(i-1,j-1) + \gamma(x_i \to y_j) \end{array} \right\}.$$

¿Qué casos base podemos considerar? Uno evidente es la transformación de la cadena vacía en la cadena vacía, es decir, no hacer nada y, por tanto, tener coste de edición nulo.

$$D(0,0) = 0.$$

Otro caso base consiste en transformar una cadena $x_{1:i}$, para $i > 0$, en la cadena vacía. La última operación sólo puede ser un borrado:

$$D(i,0) = D(i-1,0) + \gamma(x_i \to \lambda).$$

Un razonamiento análogo conduce a que

$$D(0,j) = D(0,j-1) + \gamma(\lambda \to y_j).$$

La ecuación recursiva es, pues,

$$D(i,j) = \begin{cases} 0, & \text{si } i = 0 \text{ y } j = 0; \\ D(i-1,0) + \gamma(x_i \to \lambda), & \text{si } i > 0 \text{ y } j = 0; \\ D(0,j-1) + \gamma(\lambda \to y_j), & \text{si } i = 0 \text{ y } j > 0; \\ \text{mín} \left\{ \begin{array}{l} D(i-1,j) + \gamma(x_i \to \lambda), \\ D(i,j-1) + \gamma(\lambda \to y_j), \\ D(i-1,j-1) + \gamma(x_i \to y_j) \end{array} \right\}, & \text{si } i > 0 \text{ y } j > 0. \end{cases}$$

Podemos eliminar la función γ a costa de complicar ligeramente la ecuación:

$$D(i,j) = \begin{cases} 0, & \text{si } i = 0 \text{ y } j = 0; \\ D(i-1,0) + 1, & \text{si } i > 0 \text{ y } j = 0; \\ D(0,j-1) + 1, & \text{si } i = 0 \text{ y } j > 0; \\ \text{mín} \left\{ \begin{array}{l} D(i-1,j) + 1, \\ D(i,j-1) + 1, \\ \begin{cases} D(i-1,j-1), & \text{si } x_i = y_j; \\ D(i-1,j-1) + 1, & \text{si } x_i \neq y_j; \end{cases} \end{array} \right\}, & \text{si } i > 0 \text{ y } j > 0. \end{cases} \tag{8.16}$$

La distancia de Levenshtein entre dos cadenas x e y es $D(|x|,|y|)$.

8.13.2. Una aproximación distinta

No es ésta la única forma de derivar una ecuación recursiva. Podemos plantear una definición alternativa de las soluciones a partir de la observación de que las transformaciones de edición, sin más, presentan cierta ambigüedad. ¿Dónde se «aplica» una inserción como $\lambda \to$ `a`? Si x tiene más de una `e`, ¿dónde aplicamos una sustitución cómo `e` $\to$ `a`? Para eliminar esta ambigüedad hemos de acompañar cada operación de edición con la posición de x y/o y en que tiene efecto. Es interesante considerar representaciones alternativas de las transformaciones de edición, como las que se muestran en la figura 8.42.

Con estas representaciones gráficas es evidente que podemos expresar una secuencia de operaciones de edición mediante la secuencia de índices emparejados. La secuencia $((1,1),(2,2),(4,3),(5,4),(7,5))$ representa a la transformación de edición óptima que se muestra gráficamente en la figura 8.42 (a). Si explicitamos las operaciones de borrado e inserción, $((1,1),(2,2),(3,2),(4,3),(5,4),(6,4),(7,5),(7,6))$ expresa la misma transformación. Esta representación alternativa, que se muestra gráficamente en la figura 8.42 (b), recibe el nombre de **traza de edición**.

Cada dos pares consecutivos expresan una operación de edición. Los pares (i_{k-1}, j_{k-1}) e (i_k, j_k) corresponden a la operación de edición $x_{i_{k-1}+1:i_k} \to y_{j_{k-1}+1:j_k}$, donde $x_{a:b}$ es la

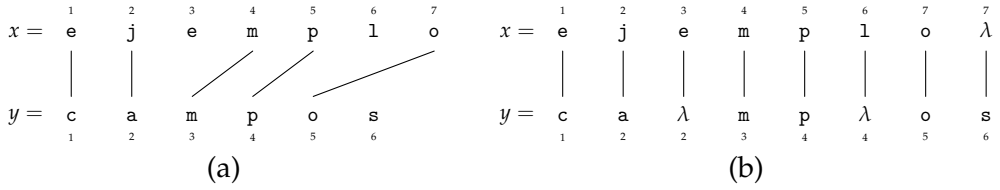


Figura 8.42: (a) Una representación de la transformación de edición óptima entre $x =$ `ejemplo` e $y =$ `campos`. Los caracteres de x sin emparejar con uno de y corresponden a una operación de borrado. (b) Una representación alternativa de la misma transformación de edición que explicita las operaciones de borrado.

subcadena $x_a, x_{a+1}, \ldots, x_b$ si $b \geq a$, y λ si $b < a$. Un ejemplo será de ayuda. En las cadena $x =$ `bien` e $y =$ `mal`, los pares $(3,2)$ y $(4,3)$ corresponden a la operación $x_{4:4} \rightarrow y_{3:3}$, es decir, `n` $\rightarrow$ `l`; y los pares $(3,2)$ y $(3,3)$ corresponden a la operación $x_{4:3} \rightarrow y_{3:3}$, es decir, $\lambda \rightarrow$ `l`.

Para evitar problemas con la interpretación del primer emparejamiento, nos conviene disponer de un elemento adicional, $(i_0, j_0) = (0,0)$. La figura 8.45 muestra la relación existente entre operaciones de edición y aristas en el grafo de dependencias. Una arista vertical de la forma $((i, j-1), (i, j))$ corresponde a la inserción del símbolo y_j. Una horizontal de la forma $((i-1, j), (i, j))$ corresponde a un borrado de x_i. Y una arista diagonal, de la forma $((i-1, j-1), (i, j))$, está asociada a la sustitución/concordancia de x_i e y_j.

Podemos reformular el conjunto de soluciones factibles en términos de este último tipo de representaciones. El conjunto de trazas de edición entre x e y, $T_{x,y}$, es el conjunto de series de pares de enteros, $((i_0, j_0), (i_1, j_1), (i_2, j_2), \ldots, (i_n, j_n))$, tales que

- $i_0 = 0$, $j_0 = 0$;
- $i_n = |x|$, $j_n = |y|$;
- $0 \leq i_k - i_{k-1} \leq 1$, $0 \leq j_k - j_{k-1} \leq 1$, para todo k entre 1 y n.

EJERCICIOS

49 Deriva la ecuación recursiva propia del cálculo de la distancia de Levenshtein entre dos cadenas x e y a partir del cálculo de

$$\min_{((i_0,j_0),(i_1,j_1),\ldots,(i_n,j_n))\in T_{x,y}} \sum_{1\leq k\leq n} \gamma(x_{i_{k-1}+1:i_k} \rightarrow y_{j_{k-1}+1:j_k}).$$

8.13.3. Algoritmo iterativo

Obviamos el desarrollo de un algoritmo recursivo y pasamos a desarrollar uno iterativo. El grafo de dependencias entre resultados de las llamadas recursiva es un grafo mallado. La figura 8.43 muestra el grafo de dependencias asociado al cálculo de la distancia de Levenshtein para las cadenas $x =$ `ejemplo` e $y =$ `campo`.

El vértice de la columna i y fila j corresponde a la transformación óptima (en el sentido de Levenshtein) de la cadena $x_{1:i}$ en la cadena $y_{1:j}$. Si dicho vértice se denota con (i, j), las aristas serán pares de pares de la forma $((i', j'), (i, j))$.

Podemos recorrer los vértices en cualquier orden de recorrido que preserve una ordenación topológica. En la figura 8.44 se muestran cuatro posibles órdenes de recorrido. Optaremos por un recorrido por filas.

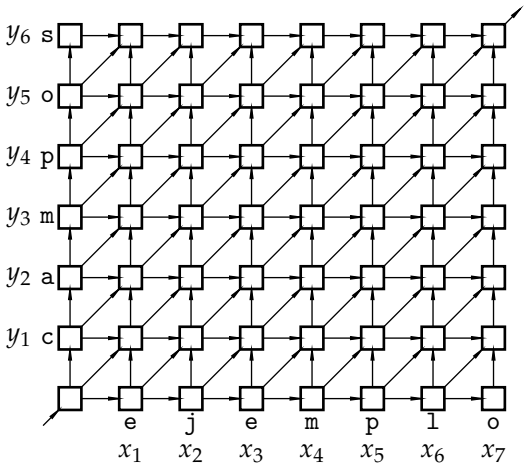


Figura 8.43: Grafo de dependencia para el cálculo de la distancia de Levenshtein entre las cadenas $x =$ `ejemplo` *e $y =$* `campos`*.*

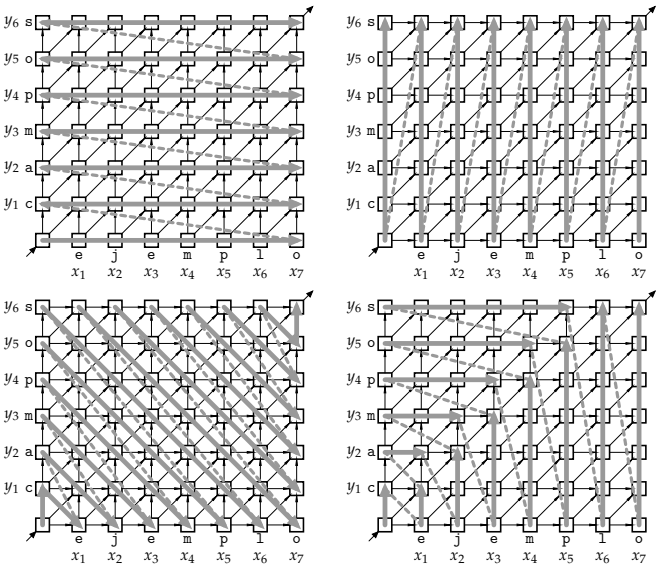


Figura 8.44: Cuatro órdenes de recorrido de los vértices del grafo de edición que respetan el orden topológico: arriba a la izquierda, por filas; arriba a la derecha, por columnas; abajo a la izquierda, por diagonales; abajo a la derecha, en cuña.

levenshtein_distance.py

```
1  def levenshtein_distance(x, y):
2      D = {}
3      D[0,0] = 0
4      for i in xrange(1, len(x)+1):
5          D[i,0] = D[i-1,0] + 1
6      for j in xrange(1, len(y)+1):
7          D[0,j] = D[0,j-1] + 1
8          for i in xrange(1, len(x)+1):
9              D[i,j] = min(D[i-1,j] + 1, D[i,j-1] + 1, D[i-1,j-1] + (x[i-1] != y[j-1]))
10     return D[len(x),len(y)]
```

test_levenshtein_distance1.py

```
from levenshtein_distance import levenshtein_distance
x = "ejemplo"
y = "campos"
print "Distancia de Levenshtein entre %s y %s: %d" % (x, y, levenshtein_distance(x, y))
```

```
Distancia de Levenshtein entre ejemplo y campos: 5
```

El coste temporal y espacial del algoritmo es $O(|x||y|)$.

EJERCICIOS

50 ¿Es válida esta otra ecuación recursiva para calcular la distancia de Levenshtein entre x e y con $D(|x|, |y|)$?

$$D(i,j) = \begin{cases} 0, & \text{si } i = 0 \text{ y } j = 0; \\ i, & \text{si } i > 0 \text{ y } j = 0; \\ j, & \text{si } i = 0 \text{ y } j > 0; \\ \text{mín} \left\{ \begin{array}{l} D(i-1,j)+1, \\ D(i,j-1)+1, \\ \begin{cases} D(i-1,j-1) & \text{si } x_i = y_j, \\ D(i-1,j-1)+1, & \text{si } x_i \neq y_j, \end{cases} \end{array} \right. & \text{si } i > 0 \text{ y } j > 0. \end{cases}$$

Si es válida, diseña un algoritmo iterativo para ella.

8.13.4. Una reducción de complejidad espacial

Si estudiamos el grafo veremos que al calcular valores asociados a vértices de una fila sólo recurrimos a valores de la fila anterior. Podemos reducir la complejidad espacial si almacenamos únicamente dos filas.

levenshtein_distance.py

```
def levenshtein_distance2(x, y):
    current_row = [None] * (1+len(x))
    previous_row = [None] * (1+len(x))
    current_row[0] = 0
    for i in xrange(1, len(x)+1):
        current_row[i] = current_row[i-1] + 1
    for j in xrange(1, len(y)+1):
        previous_row, current_row = current_row, previous_row
        current_row[0] = previous_row[0] + 1
        for i in xrange(1, len(x)+1):
            current_row[i] = min(current_row[i-1] + 1,
                                 previous_row[i] + 1,
                                 previous_row[i-1] + (x[i-1] != y[j-1]))
    return current_row[len(x)]
```

test_levenshtein_distance.py

```
1 from levenshtein_distance import levenshtein_distance2
2 x = "ejemplo"
3 y = "campos"
4 print "Distancia de Levenshtein entre %s y %s: %d" % (x, y, levenshtein_distance2(x, y))
```

```
Distancia de Levenshtein entre ejemplo y campos: 5
```

La complejidad espacial de la nueva versión es $O(|x|)$.

EJERCICIOS

51 Diseña una versión del programa cuyo coste espacial sea $O(\text{mín}(|x|, |y|))$.

52 Diseña una versión del programa que use un sólo vector de talla $1 + |x|$ para el almacenamiento de resultados asociados a una fila.

8.13.5. Recuperación de la transformación de edición óptima

Se puede ver fácilmente que una traza de edición es un camino en el grafo de dependencias.

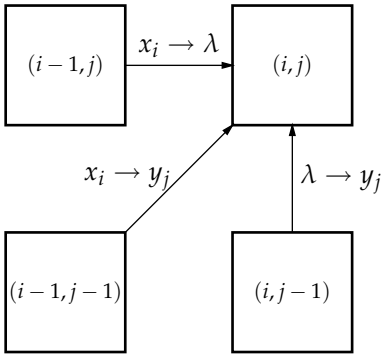


Figura 8.45: Interpretación de las aristas del grafo en términos de operaciones de edición.

Hemos etiquetado cada arista del grafo de dependencias (figura 8.46) con el peso de la operación de edición asociada (véase la figura 8.45). Nótese que todas las aristas horizontales y verticales pesan 1. Las aristas diagonales $((i-1, j-1), (i, j))$ pesan 1 cuando $x_i \neq y_j$ y 0 en caso contrario. Una traza de edición es un camino en el grafo que parte del vértice $(0, 0)$ y llega al vértice $(|x|, |y|)$. Su peso es la suma de pesos de las aristas visitadas. Un camino en el grafo es un «camino de edición» y está asociado a una transformación de edición. Los caminos de edición pueden representarse gráficamente con las denominadas trazas de edición o alineamientos. En la misma figura 8.46 se muestra el alineamiento correspondiente a un camino en el grafo de edición.

EJERCICIOS

53 Diseña un algoritmo que devuelva la traza de edición óptima para transformar x en y.

54 Diseña un algoritmo que devuelva una lista con el menor número de operaciones de edición necesarias para transformar x en y. Si, por ejemplo, suministramos las cadenas `ejemplo` y `campos`, la función devolverá una lista con este contenido:

`['sustituir e por c', 'sustituir j por a', 'borrar e', 'borrar l', 'insertar s']`.

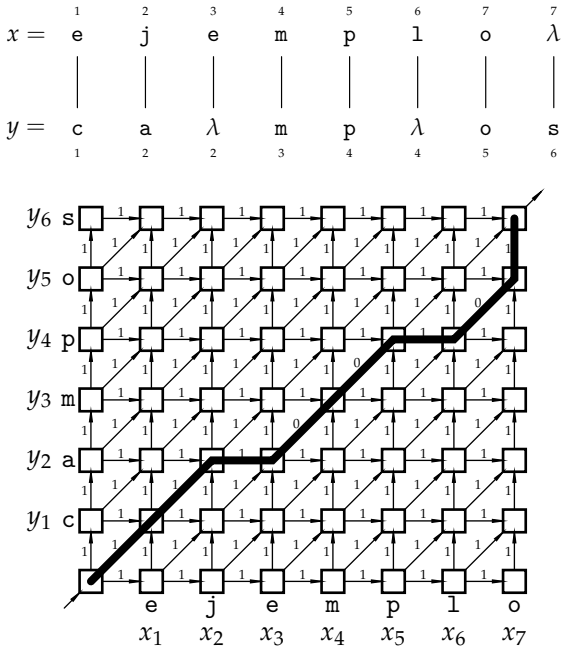


Figura 8.46: Arriba se muestra la representación gráfica de la traza de edición asociada al camino de edición destacado en trazo grueso sobre el grafo de edición para las cadenas $x =$ `ejemplo` e $y =$ `campo` que se muestra abajo.

55 Deseamos calcular la distancia de edición entre dos palabras con un sistema de penalizaciones para las operaciones de sustitución que depende de la disposición de los caracteres en el teclado «QWERTY» castellano. Una función $\gamma(\texttt{a} \to \texttt{b})$ devuelve el número de teclas del camino más corto que las une (menos uno). Por ejemplo, $\gamma(\texttt{a} \to \texttt{s})$ es 1 y $\gamma(\texttt{a} \to \texttt{e})$ vale 2 (la `s` se interpone entre ambas letras). Diseña un programa que calcule la distancia de edición considerando esta forma de penalizar las sustituciones.

56 Al comparar dos cadenas queremos incluir la posibilidad de que el carácter que ocupa cierta posición concuerde con cualquier otro. Marcaremos dicha posición con un carácter especial `&` al que denominamos «carácter de concordancia universal». La distancia de edición entre las cadenas `po&ible` y `potable` es 1: el carácter especial concuerda con la `t` y la `i` que le sucede se sustituye por la `a`. Diseña un algoritmo de programación dinámica que permita calcular la distancia de edición con marcas de concordancia universal.

57 En algunas aplicaciones prácticas sólo nos interesa conocer el valor de la distancia de Levenshtein si ésta no supera cierto valor k. ¿Puedes modificar el algoritmo y reducir su complejidad temporal a $O(\text{mín}(|x|, |y|)k)$ bajo esta nueva restricción?

8.14. El problema de la subsecuencia común más larga

Dadas dos cadenas $x = x_1x_2 \ldots x_{|x|}$ e $y = y_1y_2 \ldots y_{|y|}$ sobre cierto alfabeto Σ, se dice que $z = z_1z_2 \ldots z_n$ es una *subsecuencia común* a x e y si todos los caracteres que componen z forman parte de x e y y, además, aparecen en el mismo orden en que lo hacen en z (aunque no necesariamente en posiciones consecutivas).

Por ejemplo, si $x =$ `comparsa` e $y =$ `causarte`, $z =$ `asa` es una secuencia común a x e y (`comparsa`, `causarte`); también lo es $z' =$ `casa` (`comparsa`, `causarte`) y presenta mayor longitud.

Subsecuencia común más larga: Longest Common Subsequence.

Dadas dos cadenas x e y, el problema de la **subsecuencia común más larga** consiste en encontrar una subsecuencia común a x e y cuya longitud sea máxima.

8.14.1. Formalización

Una subsecuencia común a dos cadenas x e y puede representarse como una secuencia de pares de índices $((i_1, j_1), (i_2, j_2), \ldots, (i_n, j_n))$. Los índices $i_1, i_2, \ldots, i_n$ indican los caracteres de x que forman parte de la subsecuencia y los índices $j_1, j_2, \ldots, j_n$ hacen lo propio con y. La subsecuencia debe observar

- $1 \leq i_k \leq |x|$ y $1 \leq j_k \leq |y|$, para todo $1 \leq k \leq n$,
- $i_k < i_{k+1}$ y $j_k < j_{k+1}$ para todo $1 \leq k < n$ y
- $x_{i_k} = y_{j_k}$ para todo $1 \leq k \leq n$.

Podemos representar gráficamente una subsecuencia como un alineamiento entre caracteres idénticos, como se muestra en la figura 8.47.

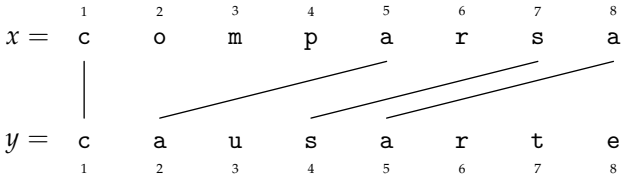


Figura 8.47: Representación gráfica del alineamiento entre $x =$ `comparsa` e $y =$ `causante` que define la subsecuencia $((1,1),(5,2),(7,4),(8,5))$ común a ambas (en particular, la más larga).

Las condiciones impuestas a los alineamientos impiden emparejamientos «cruzados» como el que se muestra en la figura 8.48.

> **!** Este problema ecuentra aplicación, por ejemplo, en análisis genético: dadas dos secuencias de ADN —cadenas formadas con las bases A (adenina), G (guanina), C (citosina) y T (timina)— o ARN —cadenas formadas con las bases A (adenina), G (guanina), C (citosina) y U (uracilo)—, se desea saber el fragmento más largo que presentan en común, con la posibilidad de que haya diferencias puntuales entre ambas.

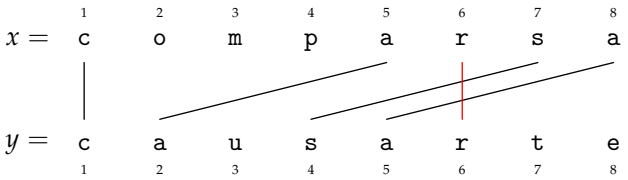


Figura 8.48: Representación gráfica de un alineamiento incorrecto: los cruces entre líneas están prohibidos.

El conjunto de subsecuencias es nuestro conjunto de soluciones factibles. La función objetivo, cuyo valor deseamos maximizar, es la talla de la subsecuencia.

8.14.2. Ecuación recursiva

Nótese que la secuencia de índices define un emparejamiento entre caracteres idénticos de las dos cadenas. Consideremos qué ocurre al calcular la subsecuencia común más larga entre $x_{1:i}$ e $y_{1:j}$. Podemos plantearnos qué ocurre con el último carácter de cada una de las cadenas, x_i e y_j:

- si $x_i = y_j$, el último par es (i, j), así que hemos de calcular la subsecuencia común más larga entre $x_{1:i-1}$ e $y_{1:j-1}$ y sumarle 1.
- si $x_i \neq y_j$, el par (i, j) no puede formar parte de la secuencia, así que hemos de seleccionar la subsecuencia más larga entre
  - la subsecuencia común más larga entre $x_{1:i-1}$ e $y_{1:j}$
  - y la subsecuencia común más larga entre $x_{1:i}$ e $y_{1:j-1}$.

  El último par de la más larga de ellas es el último par de la subsecuencia. Si alguna de las cadenas (o las dos) es la cadena vacía, la longitud de la subsecuencia común más larga es nula.

La ecuación recursiva es

$$\mathrm{LCS}(i,j) = \begin{cases} 0, & \text{si } i = 0 \text{ o } j = 0; \\ \max(\mathrm{LCS}(i-1,j), \mathrm{LCS}(i,j-1)), & \text{si } i > 0, j > 0 \text{ y } x_i \neq y_j; \\ \mathrm{LCS}(i-1,j-1) + 1, & \text{si } i > 0, j > 0 \text{ y } x_i = y_j. \end{cases} \tag{8.17}$$

Hemos considerado caso base con valor nulo el cálculo de la subsecuencia común más larga entre una cadena cualquiera y la cadena vacía.

El tamaño de la subsecuencia común más larga entre x e y es $\mathrm{LCS}(|x|, |y|)$.

> **!** Wagner y Fischer, los autores de los métodos de cálculo de distancias de edición y de la subsecuencia de edición más larga que hemos presentado, pusieron de manifiesto que el método de cálculo de la distancia de edición puede modificarse para proporcionar la longitud de la subsecuencia común más larga. Si $d(x, y)$ es la distancia de edición entre x e y considerando que una sustitución penaliza 2 y una inserción o borrado penaliza 1 y $l(x, y)$ es la longitud de la subsecuencia común más larga, ambos valores están relacionados de un modo curioso: $d(x, y) = |x| + |y| - 2l(x, y)$.

8.14.3. Versión iterativa del algoritmo

El grafo de dependencias inducido por la ecuación recursiva es similar al propio del cálculo de la distancia de Levenshtein, si bien presenta un conjunto de aristas diferente (véase la figura 8.49).

Buscamos el camino de máximo peso entre cualquier vértice de la primera fila o primera columna y el vértice de la esquina superior derecha. Podemos recorrer el grafo por filas o columnas (entre otros órdenes posibles). Este algoritmo efectúa un recorrido por filas:

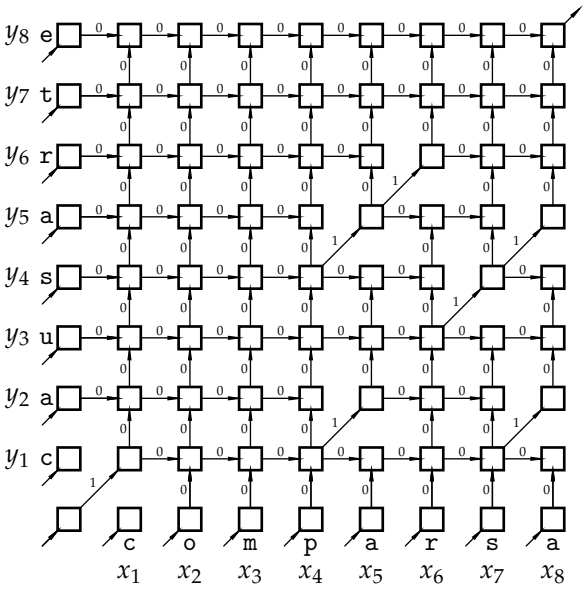


Figura 8.49: Grafo de dependencias en el cálculo de la subsecuencia común más larga entre $x =$ `comparsa` *e* $y =$ `causarte`.

lcs.py

```
1  def LCS_length(x, y):
2      LCS = {}
3      for i in xrange(0, len(x)+1):
4          LCS[i,0] = 0
5      for j in xrange(1, len(y)+1):
6          LCS[0,j] = 0
7          for i in xrange(1, len(x)+1):
8              if x[i-1] != y[j-1]: LCS[i,j] = max(LCS[i-1,j], LCS[i,j-1])
9              else:                LCS[i,j] = LCS[i-1,j-1] + 1
10     return LCS[len(x),len(y)]
```

test_longest_common_subsequence.py

```
1  from lcs import LCS_length
2  x = "comparsa"
3  y = "causarte"
4  print "Longitud de la LCS entre %s y %s: %d" % (x, y, LCS_length(x, y))
```

```
Longitud de la LCS entre comparsa y causarte: 4
```

El

> **!** He aquí un alineamiento de las bases que componen la subsecuencia común más larga entre un fragmento de RNA de Escherichia coli (izquierda) y otro de un humano (derecha).

algoritmo presenta una complejidad temporal y espacial a $O(|x||y|)$. Es posible reducir la complejidad espacial $O(|x|)$ u $O(|y|)$.

EJERCICIOS

58 Diseña un algoritmo para el cálculo de la longitud de la subsecuencia común más larga que use una cantidad de memoria $O(\text{mín}(|x|, |y|))$.

8.14.4. Recuperación de la subsecuencia común más larga

La técnica de punteros hacia atrás nos permite recuperar fácilmente el camino. En realidad no nos interesa el camino completo: sólo la información representada por las aristas «diagonales» del camino.

lcs.py

```
def LCS(x, y):
    LCS = {}
    B = {}
    for i in range(0, len(x)+1):
        LCS[i,0] = 0
        B[i,0] = None
    for j in range(1, len(y)+1):
        LCS[0,j] = 0
        B[0,j] = None
        for i in range(1, len(x)+1):
            if x[i-1] == y[j-1]:
                LCS[i,j] = LCS[i-1,j-1] + 1
                B[i,j] = (i-1, j-1)
            elif LCS[i,j-1] > LCS[i-1,j]:
                LCS[i,j] = LCS[i,j-1]
                B[i,j] = (i,j-1)
            else:
                LCS[i,j] = LCS[i-1,j]
                B[i,j] = (i-1,j)
    sequence = []
    i, j = len(x), len(y)
    while B[i,j] != None:
        if B[i,j] == (i-1,j-1):
            sequence.append(x[i-1])
        i, j = B[i,j]
    sequence.reverse()
    return ''.join(sequence)
```

test_longest_common_subsequence_b.py

```
from lcs import LCS
x = "comparsa"
y = "causarte"
print "Subsecuencia común más larga entre %s y %s: %s" % (x, y, LCS(x, y))
```

```
Subsecuencia común más larga entre comparsa y causarte: casa
```

El coste temporal y espacial de este algoritmo es $O(|x||y|)$.

EJERCICIOS

59 Un estudiante cree que es posible obtener la subsecuencia común más larga como subproducto del cálculo de la distancia de Levenshtein. Su razonamiento es que, a fin de cuentas, la distancia de Levenshtein busca un alineamiento con la menor suma de penalizaciones posibles, y lo único que no penaliza es una concordancia. Naturalmente, piensa, la menor penalización corresponde siempre al mayor número posible de concordancias, así que el alineamiento de edición óptimo contiene la subsecuencia más larga: basta con considerar las aristas diagonales que unen caracteres idénticos. ¿Está equivocado o acertado en su suposición?

60 Diseña un algoritmo de programación dinámica que encuentre la subsecuencia creciente más larga de una secuencia de N enteros. Por ejemplo, dada la secuencia 3, 5, 1, 7, 2, 9, 4, 6, una posible solución es 1, 2, 4, 6 y otra, 3, 5, 7, 9. Calcula los costes temporal y espacial del algoritmo iterativo.

8.15. Alineamiento temporal no lineal

Reconocimiento de formas: Pattern Recognition.

Podemos definir una métrica entre secuencias de valores que encuentra aplicación, entre otros campos, en el **reconocimiento de formas**. Consideremos un problema de reconocimiento en línea de escritura manuscrita. El usuario escribe en pantalla letras o dígitos de un juego de caracteres preestablecido. El sistema debe comparar cada trazo del usuario con los que tiene almacenados «de fábrica» y decidir a cuál se parece más. La figura 8.50 ilustra el problema.

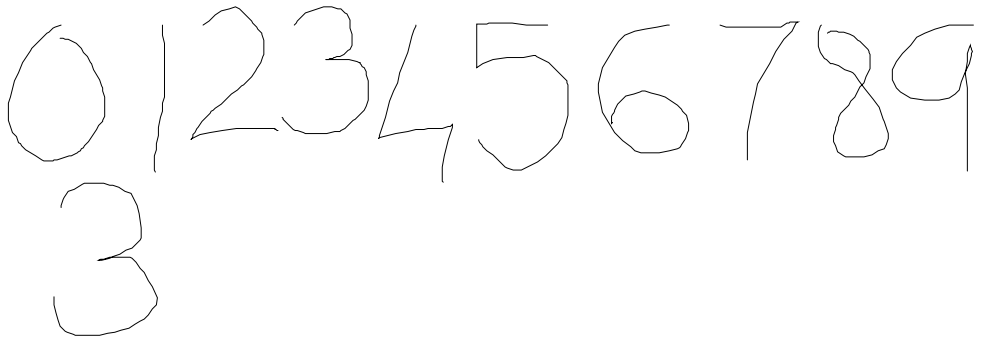


Figura 8.50: En la línea superior, trazos almacenados en la base de datos «de fábrica» para comparar con los trazos del usuario. En la línea inferior, un trazo introducido por el usuario que el sistema debe clasificar comparándolo con los de la línea superior.

Cada trazo es una secuencia de puntos en el plano de longitud arbitraria. La figura 8.51 muestra un trazo y la secuencia de puntos con que se representa.

Pongamos que comparamos dos trazos diferentes del dígito 3. Podemos hacerlo midiendo la distancia euclídea entre pares de puntos «equivalentes» en ambas sucesiones de puntos. El problema surge a la hora de definir «equivalentes». Podemos decidir que el i-ésimo punto de un trazo es equivalente al i-ésimo punto del otro. La figura 8.52 ilustra este tipo de alineamiento entre dos trazos del dígito 3. Pero hay problemas: las dos secuencias pueden tener muy diferente número de puntos y, aun en el caso de que tuvieran idéntica cantidad, un alineamiento punto a punto no pone en correspondencia partes similares de cada trazo.

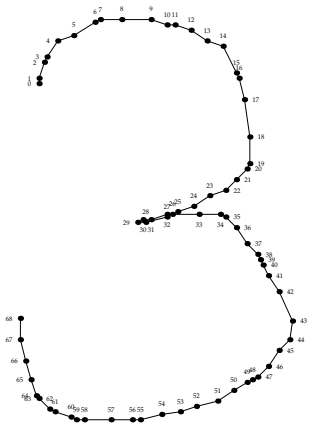


Figura 8.51: Cada trazo es una secuencia de puntos en el plano.

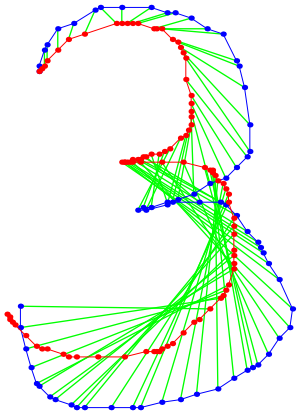


Figura 8.52: Alineamiento punto a punto entre dos trazos del dígito 3. Quedan puntos sin alinear y el alineamiento empareja puntos de zonas «no equivalentes».

Interesaría idear algún método que «supiera» alinear las partes similares de dos trazos. Si dispusiésemos de un método para calcular alineamientos sería sencillo definir una medida de disimilitud entre trazos: la suma de distancias entre puntos alineados. La figura 8.53 ilustra por qué cabe esperar que esta técnica ofrezca buenos resultados: a mano izquierda se muestra un alineamiento óptimo en cierto sentido entre dos instancias de un 3 y, a mano derecha, un alineamiento óptimo entre una instancia de un 3 y otra de un 4. El alineamiento se visualiza gráficamente con líneas que unen los puntos emparejados. La suma de distancias entre pares de puntos alineados es claramente menor al alinear dos instancias de un mismo dígito que al hacer lo mismo con instancias de dígitos distintos.

Si seguimos esta aproximación hemos de definir qué es un alineamiento óptimo. Pues bien, la definición guarda relación con el propio concepto de disimilitud: el alineamiento óptimo es aquel que hace mínima la suma de distancias entre puntos alineados.

Podemos advertir que los alineamientos válidos permiten que un punto de uno de los trazos se ponga en correspondencia con más de un punto del otro. Esta prohibido, eso sí, que un punto de cualquier trazo quede sin alinear (si se permitiera, el alineamiento óptimo, con coste nulo, no pondría a ningún punto en correspondencia con ningún otro). La figura 8.54 ilustra algunos alineamientos válidos y otros inválidos.

Definamos formalmente un alineamiento válido. Dadas dos secuencias de puntos $a =$

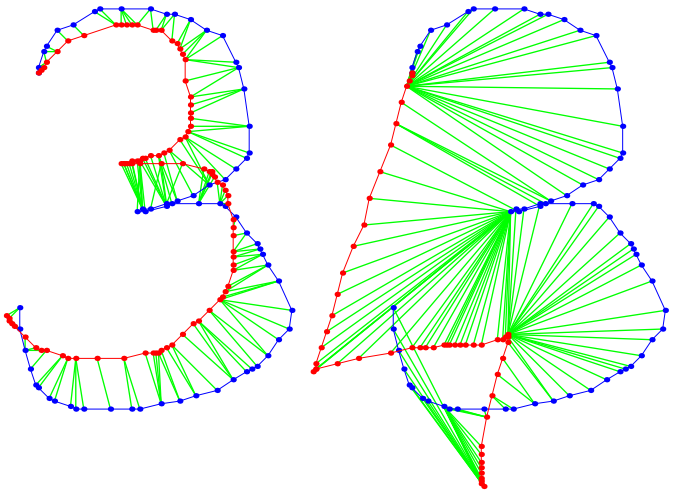


Figura 8.53: A la izquierda, alineamiento óptimo entre dos trazos del dígito 3. A la derecha, alineamiento óptimo entre el trazo de un 4 y el de un 3. Los puntos alineados en cada caso se muestran unidos por una línea recta. La longitud de dicha línea es, naturalmente, la distancia euclídea entre los puntos. Es evidente que la suma de longitudes entre puntos alineados (así como la longitud media) es menor al comparar dos instancias de un mismo dígito que al comparar instancias de dígitos diferentes.

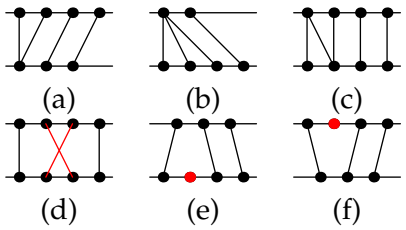


Figura 8.54: Algunos alineamientos correctos e incorrectos. Es posible alinear un punto de un trazo con más de un punto del otro, como se muestra en (a) y (b). También pueden darse situaciones como la mostrada en (c), en la que se ponen en correspondencia dos puntos de ambos trazos que ya se habían alineado previamente con otros puntos. Los alineamientos no pueden cruzar las líneas, como en (d), ni dejar puntos sin emparejar, como en (e) y (f).

> **!** Hay definiciones alternativas de alineamiento. En aras de la brevedad y simplicidad de exposición, no entraremos a considerarlas.

$(a_1, a_2, \ldots, a_m)$ y $b = (b_1, b_2, \ldots, b_n) \in (\mathbb{R} \times \mathbb{R})^+$, definimos un alineamiento entre a y b como una secuencia de pares $((i_1, j_1), (i_2, j_2), \ldots, (i_p, j_p))$ tal que:

- $(i_1, j_1) = (1, 1)$;
- $(i_p, j_p) = (m, n)$;

Para evitar cruces entre puntos alineados, debe observarse:

- $(i_{k-1}, j_{k-1}) \neq (i_k, j_k)$, para $1 < k \leq p$;
- $i_{k-1} \leq i_k$, para $1 < k \leq p$;
- $j_{k-1} \leq j_k$, para $1 < k \leq p$.

Si no desemos que el alineamiento deje «huecos» (pares de puntos sin alinear), hemos de añadir una restricción adicional

- $0 \leq i_k - i_{k-1} \leq 1$, para $1 < k \leq p$;
- $0 \leq j_k - j_{k-1} \leq 1$, para $1 < k \leq p$.

Sea $X(a, b)$ el conjunto de alineamientos entre a y b. Definimos la **distancia de alineamiento temporal no lineal** (DTW, por el inglés «Dynamic Time Warping») entre a y

Alineamiento temporal no lineal: Dynamic Time Warping.

b como

$$\mathrm{DTW}(a,b) = \min_{((i_1,j_1),(i_2,j_2),\ldots,(i_p,j_p))\in X(a,b)} \sum_{1\le k\le p} d(a_{i_k}, b_{j_k}),$$

donde $d(a_{i_k}, b_{j_k})$ es la distancia entre dos puntos (y que nosotros asumiremos que es la distancia euclídea).

8.15.1. Ecuación recursiva

Ya hemos formalizado el problema en términos de cálculo del valor mínimo de una función sobre cierto conjunto de secuencias. Derivemos la ecuación recursiva que permite calcular la DTW entre a y b. Pensemos en qué opciones tenemos para el último par de puntos alineados: sólo una. Por definición, los últimos puntos alineados son a_m y b_n. Pensemos, pues, en qué opciones tenemos para el penúltimo par:

- puede que a_{m-1} se alinee con b_n,
- o que b_{n-1} se alinee con a_m,
- o que a_{m-1} se alinee con b_{n-1}.

En el primer caso se produce un nuevo subproblema: alinear $a_{1:m-1}$ con $b_{1:n}$; en el segundo, alinear $a_{1:m}$ con $b_{1:n-1}$; y en el tercer caso, alinear $a_{1:m-1}$ con $b_{1:n-1}$. Ya podemos adivinar la recursión. Pero no nos precipitemos. Lleguemos a ella paso a paso:

$$\begin{aligned}
\mathrm{DTW}(a_{1:m}, b_{1:n}) &= \min_{((i_1,j_1),(i_2,j_2),\ldots,(i_p,j_p))\in X(a_{1:p},b_{i:p})} \sum_{1\le k\le p} d(a_{i_k}, b_{j_k}) \\
&= \min_{(i_{p-1},j_{p-1})\in\{(m-1,n),(m,n-1),(m-1,n-1)\}} \\
&\qquad \left(\min_{((i_1,j_1),(i_2,j_2),\ldots,(i_{p-1},j_{p-1}))\in X(a_{1:i_{p-1}},b_{1:j_{p-1}})} \left(\sum_{1\le k<p} d(a_{i_k}, b_{j_k}) \right) + d(a_m, b_n) \right) \\
&= \min_{(i_{p-1},j_{p-1}\in\{(m-1,n),(m,n-1),(m-1,n-1)\}} (\mathrm{DTW}(a_{1:i_{p-1}}, b_{1:j_{p-1}}) + d(a_m, b_n)) \\
&= \min \left\{ \begin{array}{l} \mathrm{DTW}(a_{1:m-1}, b_{1:n}) + d(a_m, b_n), \\ \mathrm{DTW}(a_{1:m}, b_{1:n-1}) + d(a_m, b_n), \\ \mathrm{DTW}(a_{1:m-1}, b_{1:n-1}) + d(a_m, b_n) \end{array} \right\}.
\end{aligned}$$

Son particularmente sencillos los casos en que nos piden alinear una secuencia formada con un sólo punto con otra de uno o más puntos:

$$\begin{aligned}
\mathrm{DTW}(a_1, b_{1:j}) &= \mathrm{DTW}(a_1, b_{1:j-1}) + d(a_1, b_j) = \sum_{1\le k\le j} d(a_1, b_k), \\
\mathrm{DTW}(a_{1:i}, b_1) &= \mathrm{DTW}(a_{1:i-1}, b_1) + d(a_i, b_1) = \sum_{1\le k\le i} d(a_k, b_1).
\end{aligned}$$

Y resulta trivial alinear una secuencia de un solo punto con otra de un solo punto:

$$\mathrm{DTW}(a_1, b_1) = d(a_1, b_1).$$

Simplificando la notación, mostramos la ecuación recursiva con $\mathrm{DTW}(i,j)$ para hacer referencia a $\mathrm{DTW}(a_{1:i}, b_{1:j})$:

$$\mathrm{DTW}(i,j) = \begin{cases} d(a_1,b_1), & \text{si } i=1 \text{ y } j=1; \\ \mathrm{DTW}(1,j-1)+d(a_1,b_j), & \text{si } i=1 \text{ y } j>1; \\ \mathrm{DTW}(i-1,1)+d(a_i,b_1), & \text{si } i>1 \text{ y } j=1; \\ \text{mín}\left\{\begin{array}{l}\mathrm{DTW}(i-1,j)+d(a_i,b_j),\\ \mathrm{DTW}(i,j-1)+d(a_i,b_j),\\ \mathrm{DTW}(i-1,j-1)+d(a_i,b_j)\end{array}\right\}, & \text{si } i>1 \text{ y } j>1. \end{cases}$$

Podemos refinar la ecuación para evitar la realización de tres cálculos de distancia euclídea en la minimización cuando una es suficiente:

$$\mathrm{DTW}(i,j) = \begin{cases} d(a_1,b_1), & \text{si } i=1 \text{ y } j=1; \\ \mathrm{DTW}(1,j-1)+d(a_1,b_j), & \text{si } i=1 \text{ y } j>1; \\ \mathrm{DTW}(i-1,1)+d(a_i,b_1), & \text{si } i>1 \text{ y } j=1; \\ \text{mín}\left\{\begin{array}{l}\mathrm{DTW}(i-1,j),\\ \mathrm{DTW}(i,j-1),\\ \mathrm{DTW}(i-1,j-1)\end{array}\right\}+d(a_i,b_j), & \text{si } i>1 \text{ y } j>1. \end{cases} \tag{8.18}$$

8.15.2. Algoritmo iterativo

En la figura 8.55 se muestra el grafo de dependencias inducido por la ecuación recursiva. Es un grafo mallado similar al que subyace al cálculo de la distancia de Levenshtein. Podemos efectuar un recorrido por filas, columnas, diagonales o en cuña.

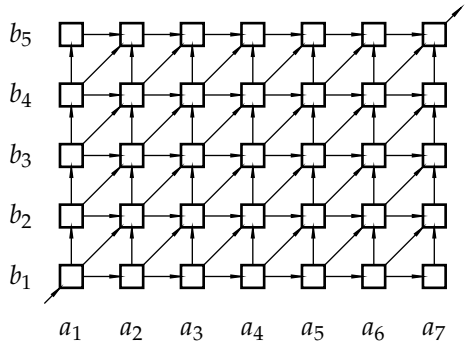


Figura 8.55: Grafo de dependencias en el cálculo de la distancia de alineamiento temporal no lineal entre dos secuencias de puntos de tallas 7 y 5.

He aquí el algoritmo:

dtw.py

```
1 def d(v, w): # distancia euclídea entre dos puntos.
2     return sqrt((v[0]-w[0])**2 + (v[1]-w[1])**2)
3
4 def DTW(a, b):
```

```
5    D = {}
6    D[0,0] = d(a[0], b[0])
7    for i in xrange(1, len(a)):
8        D[i,0] = D[i-1,0] + d(a[i], b[0])
9    for j in xrange(1, len(b)):
10       D[0,j] = D[0,j-1] + d(a[0], b[j])
11       for i in xrange(1, len(a)):
12           D[i,j] = min(D[i-1,j], D[i,j-1], D[i-1,j-1]) + d(a[i],b[j])
13   return D[len(a)-1,len(b)-1]
```

El coste temporal y espacial de esta versión es $O(mn)$.

EJERCICIOS

61 Reduce la complejidad espacial a $O(\text{mín}(m,n))$.

8.15.3. Normalización del resultado

La función DTW, tal cual ha sido definida, presenta un problema: tiende a producir valores más altos cuando se le suministran secuencias largas aunque parecidas que cuando se le presentan secuencias cortas y muy diferentes. Es un efecto producido porque en el primer caso participan más sumandos en el sumatorio. Para que la medida de DTW sea inmune a este efecto es necesario trabajar con la distancia promedio entre pares alineados. Ésta es una definición de la distancia de alineamiento temporal no lineal que incorpora esta idea del promedio:

$$\widehat{\text{DTW}}(a,b) = \min_{((i_1,j_1),(i_2,j_2),\ldots,(i_p,j_p))\in X(a,b)} \frac{\sum_{1\le k\le p} d(a_{i_k}, b_{j_k})}{p},$$

es decir, dividir la distancia entre el número de emparejamientos entre pares de puntos que hace el alineamiento óptimo. Pero hay un problema con esta definición: conduce a un algoritmo de programación dinámica que requiere tiempo $O(m^2 n)$, pues el número de emparejamientos varía entre $\text{máx}(m,n)$ y $m+n$ (véase la figura 8.56) y se ha de encontrar el alineamiento óptimo para cada posible valor.

> **!** Aunque hay técnicas más eficientes basadas en la denominada programación fraccional.

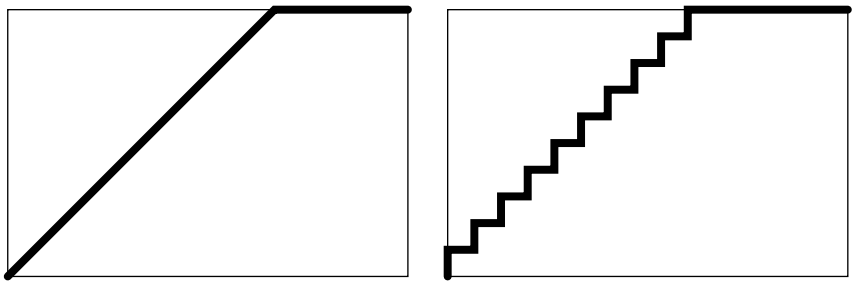


Figura 8.56: Representación esquemática de (a) el camino con el menor número de emparejamientos, $\text{máx}(m,n)$*, y (b) un camino con el mayor número posible de emparejamientos,* $m+n$*.*

Una definición alternativa es ésta otra

$$\widehat{\mathrm{DTW}}(a,b) = \min_{((i_1,j_1),(i_2,j_2),\ldots,(i_p,j_p))\in X(a,b)} \frac{\sum_{1\le k\le p} d(a_{i_k}, b_{j_k})}{m+n}.$$

Esta definición cuenta con la ventaja de que $m+n$ es un valor constante (no depende del alineamiento óptimo) y, por tanto,

$$\widehat{\mathrm{DTW}}(a,b) = \frac{\min_{((i_1,j_1),(i_2,j_2),\ldots,(i_p,j_p))\in X(a,b)} \sum_{1\le k\le p} d(a_{i_k}, b_{j_k})}{m+n} = \frac{\mathrm{DTW}(a,b)}{m+n}.$$

Sin embargo, el valor finalmente calculado no es un promedio de distancias entre pares de puntos: el número de pares de puntos que forman un alineamiento entre a y b está comprendido entre $\max(m,n)$ y $m+n$, y siempre dividimos por $m+n$.

Una técnica frecuentemente utilizada para obtener un verdadero promedio consiste en hacer que los alineamientos entre pares de puntos que corresponden a aristas «diagonales» en el grafo de dependencias pesen el doble que las otras. Es decir, se plantea la resolución de esta otra ecuación recursiva:

$$\mathrm{DTW}(i,j) = \begin{cases} 2d(a_1,b_1), & \text{si } i=1 \text{ y } j=1; \\ \mathrm{DTW}(1,j-1) + d(a_1,b_j), & \text{si } i=1 \text{ y } j>1; \\ \mathrm{DTW}(i-1,1) + d(a_i,b_1), & \text{si } i>1 \text{ y } j=1; \\ \min \left\{ \begin{array}{l} \mathrm{DTW}(i-1,j) + d(a_i,b_j), \\ \mathrm{DTW}(i,j-1) + d(a_i,b_j), \\ \mathrm{DTW}(i-1,j-1) + 2d(a_i,b_j) \end{array} \right\}, & \text{si } i>1 \text{ y } j>1; \end{cases} \tag{8.19}$$

y devolver como distancia promedio entre a y b el valor $\mathrm{DTW}(a,b)/(m+n)$. Se observa en la práctica que esta distancia normalizada produce mejores resultados que la que no tiene en cuenta la normalización.

EJERCICIOS

62 Este programa Python implementa un sencillo programa de reconocimiento «en línea» de dígitos escritos en pantalla con el ratón (a falta de un dispositivo más adecuado, como un lápiz óptico).

> **!** Las agendas electrónicas de diseño relativamente reciente integran software de reconocimiento de escritura manuscrita con funcionalidad similar a éste.

handwritten_digits.py

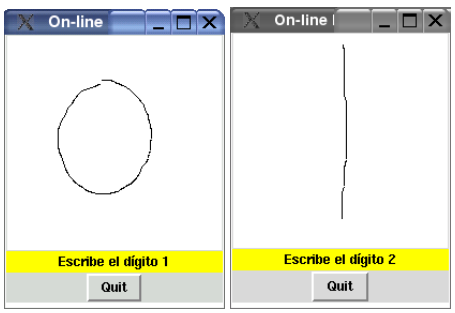
```
1 from Tkinter import *
2 from math import sqrt
3
4 class OnLineHandwritting(Frame):
5     def __init__(self, master=None):
6         Frame.__init__(self, master)
7         self.grid()
```

```
8         self.createWidgets()
9         self.strokeLines = []
10        self.mode = 'learning'
11        self.digits = []
12        self.askForDigit(0)
13
14    def createWidgets(self):
15        self.drawingArea = Canvas(self, width=200, height=200, bg='white')
16        self.drawingArea.grid()
17        self.messageArea = Label(self, bg='yellow')
18        self.messageArea.grid(sticky=E+W)
19        self.quitButton = Button ( self, text="Quit", command=self.quit )
20        self.quitButton.grid()
21        self.drawingArea.bind('<Button-1>', self.onButton)
22        self.drawingArea.bind('<ButtonRelease-1>', self.onReleaseButton)
23
24    def askForDigit(self, i):
25        self.messageArea.configure(text='Escribe el dígito %d' % i)
26
27    def reportDigit(self, i):
28        self.messageArea.configure(text='Has escrito un %d' % i)
29
30    def onButton(self, event):
31        self.x0, self.y0 = event.x, event.y
32        self.stroke = [(0, 0)]
33        for line in self.strokeLines: self.drawingArea.delete(line)
34        self.strokeLines = []
35        self.drawingArea.bind('<Motion>', self.onMotion)
36
37  def onReleaseButton(self, event):
38      self.drawingArea.unbind('<Motion>')
39      if self.mode == 'learning':
40          self.digits.append(self.stroke)
41          self.askForDigit(len(self.digits))
42          if len(self.digits) == 10:
43              self.mode = 'recognizing'
44              self.messageArea.configure(text='Escriba dígitos')
45          elif self.mode == 'recognizing':
46              classifiedAs = classify(self.stroke, self.digits)
47              self.reportDigit(classifiedAs)
48
49  def onMotion(self, event):
50      x1, y1 = self.stroke[-1]
51      self.stroke.append( (event.x-self.x0, event.y-self.y0) )
52      self.strokeLines.append(
53          self.drawingArea.create_line(event.x, event.y, x1+self.x0, y1+self.y0))
54
55 def classify(stroke, digits):
56     distances = [ DTW(stroke, digit) for digit in digits ]
```

```
57      return distances.index(min(distances))
58
59  def d(v, w):
60      return sqrt((v[0]-w[0])**2 + (v[1]-w[1])**2)
61
62  def DTW(x, y):
63      D = {}
64      D[0,0] = 2*d(x[0], y[0])
65      for i in xrange(1, len(x)):
66          D[i,0] = D[i-1,0] + d(x[i], y[0])
67      for j in xrange(1, len(y)):
68          D[0,j] = D[0,j-1] + d(x[0], y[j])
69          for i in xrange(1, len(x)):
70              dxy = d(x[i],y[j])
71              D[i,j] = min(D[i-1,j], D[i,j-1], D[i-1,j-1] + dxy) + dxy
72      return D[len(x)-1,len(y)-1]/float(len(x)+len(y))
73
74  app = OnLineHandwritting()
75  app.master.title("On-line handwritting")
76  app.mainloop()
```

Al iniciar el programa aparece en pantalla una ventana como ésta:

El usuario debe dibujar, con el ratón pulsado, un cero (de un solo trazo) en la zona blanca de la ventana. Una vez lo haya hecho, el sistema le solicitará que trace un uno. Y así sucesivamente hasta que el usuario haya trazado un nueve:



...

Entonces, el programa solicita reiteradamente al usuario que trace cualquiera de los dígitos y le informa de cuál cree que es. Para ello, el sistema calcula la distancia DTW entre el nuevo dígito y los diez que se introdujeron inicialmente y conjetura que el nuevo pertenece a la misma «clase» que el más cercano.

Cada trazo se representa como una sucesión de pares (x_0, y_0), (x_1, y_1),... (x_n, y_n) donde cada par (x_i, y_i) representa, en un instante dado, las coordenadas del ratón relativas al punto en el que pulsó el botón izquierdo. Con esta representación se consigue independencia del punto de inicio (invarianza a la translación). Modifica la representación o los cálculos efectuados durante la clasificación para conseguir:

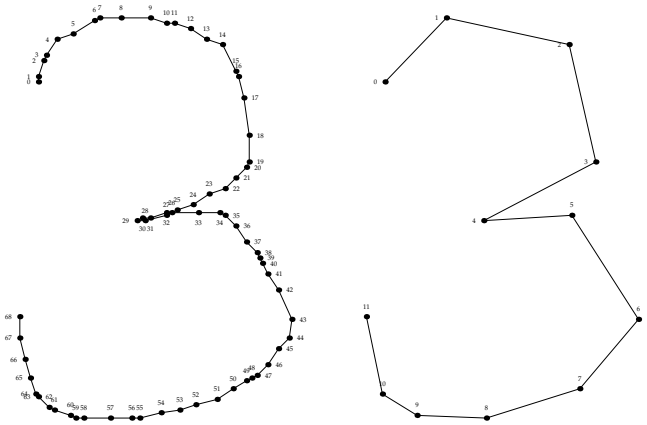
- invarianza a la rotación (aunque esto, sin más, te traerá problemas de confusión entre el 6 y el 9),
- invarianza a la dirección del trazo (o sea, que admita como idénticos un 2 cuyo trazo empieza por la zona superior izquierda y otro cuyo trazo empieza en la zona inferior derecha o un 0 trazado en sentido horario y otro trazado en sentido antihorario),
- e invarianza a la escala.

Modifica también el programa para que lea los dígitos de entrenamiento de un fichero y evite así tener que introducirlos cada vez.

63 ¿Cómo podríamos hacer que el sistema descrito en el ejercicio anterior admitiera dígitos dibujados con varios trazos?

64 Nos gustaría reducir significativamente el número de puntos que describen un trazo. Queremos segmentar la secuencia de N puntos en una cantidad K de segmentos. La «calidad» de un segmento formado por los puntos (x_i, y_i), (x_{i+1}, y_{i+1}), ..., (x_j, y_j), con $i < j$, es la suma promediada de distancias de cada uno de ellos a la recta que une los puntos (x_i, y_i) e (x_j, y_j).

Aquí tienes un trazo original del número tres, que tiene 68 segmentos, y una descripción suya con sólo 11 segmentos:



Diseña un algoritmo que obtenga la segmentación óptima, analiza su complejidad computacional y usa la nueva segmentación en el programa reconocedor del ejercicio anterior.

8.16. La carga máxima del camión

A la hora de establecer la ruta de un camión hemos de tener en cuenta la carga máxima que pueden soportar los puentes y carreteras por las que circulará. Disponemos de un mapa de carreteras en el que consta esta información. Deseamos encontrar el camino que une dos ciudades y nos permite la mayor carga posible. La carga máxima del camión que debe seguir una ruta es la carga del tramo entre ciudades que menor carga soporta.

Dado un grafo dirigido $G = (V, E)$ que modela un mapa de carreteras, una función de ponderación $c : E \to \mathbb{R}$ que indica la carga que podemos transportar por la carretera representada por cada arista, y dados dos vértices s y t, deseamos conocer la ruta entre s y t que permite que el camión circule con la máxima carga posible.

8.16.1. Formalización

Se trata de un problema que guarda gran similitud con el del camino más corto en un grafo cualquiera (sin ciclos negativos). Pero no es el mismo problema:

- deseamos obtener un valor máximo,
- y el «peso» de un camino $(v_1, v_2, \ldots, v_n)$ es $\min_{1<i\leq n} c(v_{i-1}, v_i)$.

El conjunto de caminos es el conjunto de secuencias de vértices $(v_1, v_2, \ldots, v_n)$ tal que $v_1 = s$, $v_n = t$ y $(v_{i-1}, v_i) \in E$ para todo i entre 2 y n.

La función objetivo es

$$f((v_1, v_2, \ldots, v_n)) = \min_{1<i\leq n} c(v_{i-1}, v_i).$$

Deseamos conocer el camino de s a t que maximiza su valor, es decir, calcular

$$(\hat{v}_1, \hat{v}_2, \ldots, \hat{v}_n) = \arg \max_{\substack{(v_1,v_2,\ldots,v_n):v_1=s;\\ v_n=t;(v_i,v_{i+1})\in E,1\leq i<n}} \left(\min_{1<i\leq n} c(v_{i-1}, v_i) \right).$$

La función objetivo es separable:

$$\min_{1<i\leq n} c(v_{i-1}, v_i) = \min \left(\min_{1<i\leq n-1} \big(c(v_{i-1}, v_i)\big), c(v_{n-1}, v_n) \right).$$

El operador de combinación de resultados es ahora «mín», aunque se nos presente habitualmente con notación de función, y no de operador. Pero no hay problema alguno: mín es, al igual que el operador $\oplus$, una función con perfil $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$. Nos hemos de preguntar si el operador satisface o no la propiedad de monotonía no creciente cuando maximizamos. Es decir, hemos de demostrar que

$$a \geq a' \Rightarrow \min(a, b) \geq \min(a', b).$$

Estudiemos las diferentes posibilidades:

- Si $\min(a, b)$ es a, $\min(a', b)$ será a', pues $a \geq a'$; en ese caso tendremos $\min(a, b) = a \geq a' = \min(a', b)$.

- Si, por el contrario, $\min(a,b) = b$, entonces $\min(a',b)$ puede ser a' o b. En cualquier caso, si $\min(a,b) = b$, entonces $a \geq b$.
  - Si $\min(a',b) = a'$, entonces $b \geq a'$. Por tanto, $\min(a,b) = b \geq a' = \min(a',b)$.
  - Si $\min(a',b) = b$, entonces $a' \geq b$. Como $a \geq a' \geq b$, tenemos $\min(a,b) = b = \min(a',b)$.

Tenemos, pues, que

$$\max_{a\in\mathbb{R}} \Big(\min(a,b) \Big) \geq \min \left(\Big(\max_{a\in\mathbb{R}} a \Big), b \right).$$

8.16.2. Ecuación recursiva

Procedamos a derivar la ecuación recursiva. Sea $M(t)$ la capacidad máxima de una ruta entre s y t:

$$\begin{aligned} M(t) &= \max_{\substack{(v_1,v_2,\ldots,v_n):v_1=s;\\ v_n=t;(v_{i-1},v_i)\in E,1<i\leq n}} \min_{1<i\leq n} c(v_{i-1},v_i) \\ &= \max_{\substack{(v_1,v_2,\ldots,v_n):v_1=s;\\ v_n=t;(v_{i-1},v_i)\in E,1<i\leq n}} \min \left(\min_{1<i\leq n-1} \left(c(v_{i-1},v_i) \right), c(v_{n-1},v_n) \right). \end{aligned}$$

El valor de v_n está fijo a t. El de v_{n-1} puede ser cualquier vértice predecesor de t, es decir, tal que el par (v_{n-1},t) es una arista del grafo:

$$\begin{aligned} M(t) &= \max_{(v_{n-1},t)\in E} \left(\max_{\substack{(v_1,v_2,\ldots,v_{n-1}):v_1=s;\\ (v_{i-1},v_i)\in E,1<i\leq n-1}} \min \left(\min_{1<i\leq n-1} \left(c(v_{i-1},v_i) \right), c(v_{n-1},t) \right) \right) \\ &= \max_{(v_{n-1},t)\in E} \min \left(\max_{\substack{(v_1,v_2,\ldots,v_{n-1}):v_1=s;\\ (v_{i-1},v_i)\in E,1<i\leq n-1}} \left(\min_{1<i\leq n-1} c(v_{i-1},v_i) \right), c(v_{n-1},t) \right) \\ &= \max_{(v_{n-1},t)\in E} \min \left(M(v_{n-1}), c(v_{n-1},t) \right). \end{aligned}$$

Nuevamente, da igual considerar t que cualquier otro vértice para alcanzar una expresión recursiva como ésta, así que

$$M(v) = \max_{(u,v)\in E} \min(M(u), c(u,v)).$$

Hemos de plantearnos los casos base. Un caso base evidente lo tenemos cuando $v = s$. La carga con que podemos desplazarnos de s a s es, en principio, infinita, pues no hay que transitar ninguna carretera con limitación de peso. Otro caso base lo determinan los vértices sin predecesores: su capacidad de carga es nula, pues no podemos ir de s a dichos nodos con carga alguna.

He aquí la ecuación recursiva completa:

> **!** Alternativamente al valor 0, podemos suponer una capacidad de carga $-\infty$ en los vértices sin predecesores. Cualquiera de las dos inicializaciones es razonable, pues ambas se rechazan finalmente en favor de cualquier camino que permita cargar el camión con algo.

$$M(v) = \begin{cases} +\infty, & \text{si } v = s; \\ 0, & \text{si } v \neq s \text{ y } \nexists (u,v) \in E; \\ \max\limits_{(u,v)\in E} \min(M(u), c(u,v)), & \text{en otro caso.} \end{cases} \tag{8.20}$$

Si el grafo es acíclico, ya tenemos la ecuación recursiva, pues induce un grafo de dependencias también acíclico. Pero si el grafo G presenta ciclos, ¡también los presentará el grafo de dependencias! No es posible, pues, abordar el problema por programación dinámica directamente.

EJERCICIOS

65 Diseña el algoritmo iterativo para grafos acíclicos. Analiza su coste computacional.

¿Qué podemos hacer? Un cambio de enfoque resulta de ayuda: podemos encontrar una ecuación recursiva cuyo grafo de dependencias no contenga ciclos si nos inspiramos en el algoritmo de Bellman-Ford. Allí optamos por recurrir al algoritmo que soluciona un problema diferente: el cálculo del camino más corto con k vértices. Haremos ahora algo similar: plantearemos el cálculo del camino con mayor capacidad de carga y que está formado por k vértices. Resolveremos este problema para cualquier número de vértices entre 1 y $|V|$ (no tiene sentido que el camino contenga bucles) y escogeremos el que mayor capacidad de carga ofrezca.

Si $M(t,k)$ es la capacidad de carga del camino con mayor capacidad de carga entre s y t con k vértices, tenemos:

$$M(t,k) = \max_{\substack{(v_1,v_2,\ldots,v_{k+1}):v_1=s;v_{k+1}=t;\\ (v_{i-1},v_i)\in E, 1<i\leq k+1}} \min_{1<j\leq k+1} c(v_{j-1},v_j).$$

Dejamos como ejercicio para el lector la derivación que nos permite llegar a esta ecuación recursiva:

$$M(v,k) = \begin{cases} +\infty, & \text{si } v = s \text{ y } k = 0; \\ 0, & \text{si } v \neq s \text{ y } k = 0; \\ 0, & \text{si } \nexists (u,v) \in E \text{ y } k = 0; \\ \max\limits_{(u,v)\in E} \min(M(u,k-1), c(u,v)), & \text{en otro caso.} \end{cases} \tag{8.21}$$

Queremos calcular

$$\max_{0\leq k\leq |V|} M(k,t).$$

EJERCICIOS

66 Diseña el algoritmo iterativo para el caso en el que el grafo tenga ciclos. Analiza su coste computacional.

67 Diseña un algoritmo que permita recuperar el camino óptimo en el problema de la carga máxima del camión.

68 Modifica el algoritmo para que, en caso de empate entre dos caminos, devuelva el que menor número de aristas recorre.

8.17. Parentización óptima del producto de matrices

Podemos multiplicar dos matrices A y B de dimensiones respectivas $p \times q$ y $q \times r$ para obtener una nueva matriz de dimensión $p \times r$. Si denominamos *flop* a una operación de producto y suma de números en coma flotante, podemos efectuar el producto matricial de A y B con pqr *flops*.

El producto matricial no es conmutativo, pero sí asociativo. Al efectuar el producto de tres matrices, A por B por C, podemos efectuar los cálculos de dos formas distintas: $(A \times B) \times C$ y $A \times (B \times C)$.

Consideremos cuatro matrices, M_0, M_1, M_2 y M_3, cuyas respectivas dimensiones son 10×4, 4×4, 4×3 y 3×5. Tenemos varias posibilidades a la hora de efectuar el cálculo de $M_0 \times M_1 \times M_2 \times M_3$:

- $((M_0 \times M_1) \times M_2) \times M_3$,
- $(M_0 \times M_1) \times (M_2 \times M_3)$ y
- $M_0 \times (M_1 \times (M_2 \times M_3))$.

Analicemos con detalle la primera de las posibilidades:

- Primero se efectúa el producto matricial $M_0 \times M_1$, que requiere realizar $10 \cdot 4 \cdot 4 = 160$ *flops*. La matriz resultante tiene dimensión 10×4.
- A continuación se efectúa el producto de la matriz resultante con M_2. Esta operación requiere efectuar $10 \cdot 4 \cdot 3 = 120$ *flops* y produce una matriz de dimensión 10×3.
- Y, finalmente, la matriz resultante se multiplica por M_3, de dimensión 10×5, para lo cual ejecuta $10 \cdot 3 \cdot 5 = 150$ *flops*.

El número total de *flops* es de $160 + 120 + 150 = 430$. Si lo hacemos de acuerdo con la parentización $(M_0 \times M_1) \times (M_2 \times M_3)$ se efectúan 420 *flops*. Y si operamos de acuerdo con la parentización $M_0 \times (M_1 \times (M_2 \times M_3))$, el número de *flops* es de sólo 340.

El problema que deseamos resolver se enuncia así: dadas N matrices, M_0, M_1, …, M_{N-1}, de dimensiones respectivas $p_0 \times q_0$, $p_1 \times q_1$, …, $p_{N-1} \times q_{N-1}$ tales que $q_i = p_{i+1}$ para $0 \leq i < N - 1$, deseamos calcular el producto $M_0 \times M_1 \times \cdots \times M_{N-1}$ ejecutando el menor número posible de *flops*.

8.17.1. Formalización

El conjunto de soluciones factibles es el conjunto de todas las posibles parentizaciones. La función objetivo, cuyo valor deseamos minimizar, es la suma de productos entre escalares.

Resulta difícil expresar con una notación sencilla el conjunto de las parentizaciones posibles al multiplicar N matrices. Cada parentización puede expresarse con un árbol

binario cuyas hojas son, de izquierda a derecha, las matrices M_0, M_1, ..., M_{N-1}. La figura 8.57 muestra el árbol correspondiente a cada una de las tres parentizaciones posibles del producto de cuatro matrices.

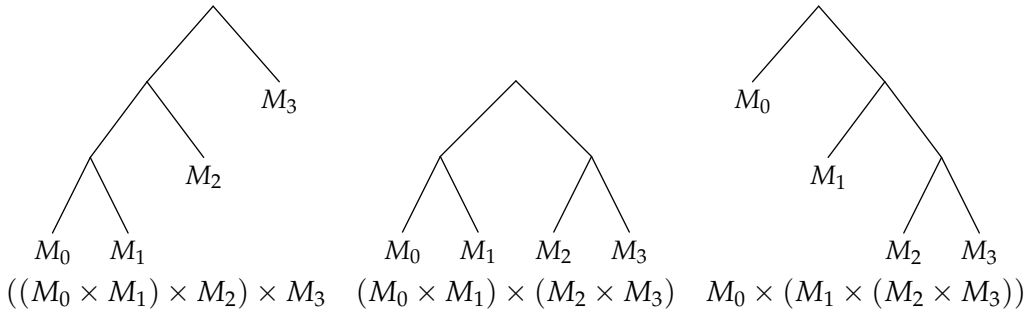


Figura 8.57: Árboles binarios que representan las tres parentizaciones posibles del producto $M_0 \times M_1 \times M_2 \times M_3$.

La optimización sobre el conjunto de parentizaciones puede plantearse como una optimización sobre un conjunto de árboles binarios. Buscamos aquél que minimiza el valor de cierta función. Pero definir el conjunto de árboles válidos resulta también bastante farragoso. Prescindiremos de introducir notación al efecto porque, como se verá a continuación, no resulta imprescindible.

8.17.2. Ecuación recursiva

Vamos a centrarnos únicamente en el último de los productos matriciales que vamos a efectuar, es decir, en la raíz del árbol. Ilustremos las posibilidades con un valor de N determinado, por ejemplo, $N = 6$. Tenemos 5 posibilidades y el último de sus productos requiere efectuar una cantidad conocida de multiplicaciones entre escalares, como se muestra en la tabla 8.1.

Último producto	***flops***
$(M_0) \times (M_1 \times M_2 \times M_3 \times M_4 \times M_5)$	$p_0q_0q_5$
$(M_0 \times M_1) \times (M_2 \times M_3 \times M_4 \times M_5)$	$p_0q_1q_5$
$(M_0 \times M_1 \times M_2) \times (M_3 \times M_4 \times M_5)$	$p_0q_2q_5$
$(M_0 \times M_1 \times M_2 \times M_3) \times (M_4 \times M_5)$	$p_0q_3q_5$
$(M_0 \times M_1 \times M_2 \times M_3 \times M_4) \times (M_5)$	$p_0q_4q_5$

Tabla 8.1: Posibilidades a la hora de efectuar la última multiplicación en el producto de seis matrices y el coste respectivo, en flops, de dicha multiplicación.

A las multiplicaciones entre escalares del último producto matricial efectuado hemos de sumar las que comporta la serie de multiplicaciones matriciales encerradas entre paréntesis en la izquierda y en la derecha. De entre todos esos valores, interesa el mínimo.

Representemos con $P(i,k)$ al menor número de multiplicaciones entre escalares con que podemos efectuar el cálculo $M_i \times M_{i+1} \times \cdots \times M_{k-1} \times M_k$:

$$P(i,k) = \min_{i \leq j < k} \Big(P(i,j) + p_i q_j q_k + P(j+1,k) \Big).$$

Si $i = k$, no hay que efectuar producto alguno, así que $P(i,i) = 0$, para $1 \leq i < n$.

La ecuación recursiva es

$$P(i,k) = \begin{cases} 0, & \text{si } i = k; \\ \min\limits_{i \leq j < k} \Big(P(i,j) + p_i q_j q_k + P(j+1,k) \Big), & \text{en otro caso.} \end{cases} \tag{8.22}$$

El valor que deseamos calcular es $P(0, N-1)$.

8.17.3. Algoritmo iterativo

La figura 8.58 muestra el grafo de dependencias para $N = 6$. Es importante apreciar que no hemos de calcular un *camino* de coste mínimo en el grafo, sino un *árbol* de coste mínimo (véase la figura 8.59): en cada vértice no escogemos *el* predecesor que minimiza cierto valor, sino *un par* de predecesores. La figura 8.60 ilustra esta idea mostrando los pares de aristas (y correspondientes pares de vértices) que podemos seleccionar al calcular el valor óptimo en el vértice $(0,5)$: cada par de aristas se ha dibujado usando el mismo tipo de trazo discontinuo. Los pares de vértices son $((0,0),(1,5))$, $((0,1),(2,5))$, $((0,2),(3,5))$, $((0,3),(4,5))$ y $((0,4),(5,5))$.

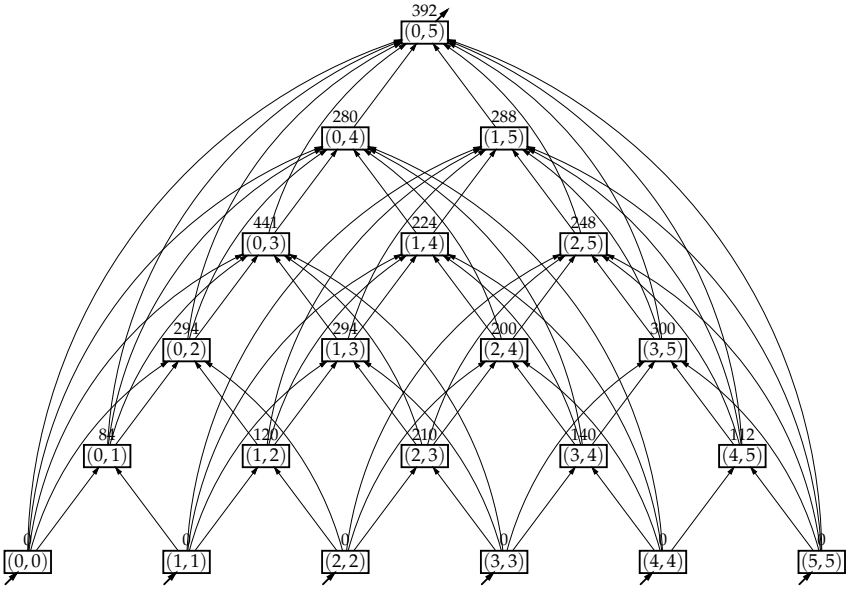


Figura 8.58: Grafo de dependencias para el problema del producto de matrices con $N = 6$.

Un recorrido de los nodos del árbol por niveles, de mayor a menor profundidad, es suficiente. Las hojas son los casos base y se inicializan en un primer paso.

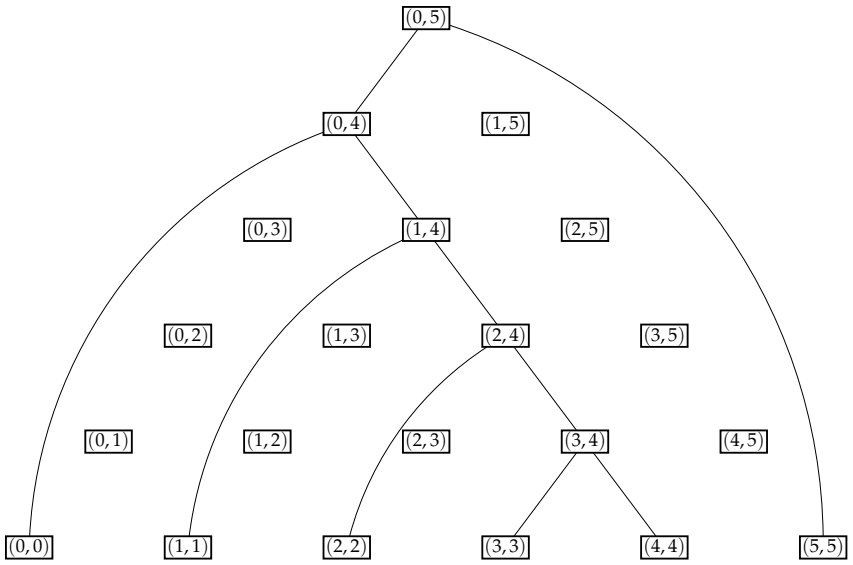


Figura 8.59: Representación de la solución $((M_0 \times (M_1 \times (M_2 \times (M_3 \times M_4)))) \times M_5)$ *al producto de 6 matrices en el grafo de dependencias. Nótese que la solución no es un camino, sino un árbol.*

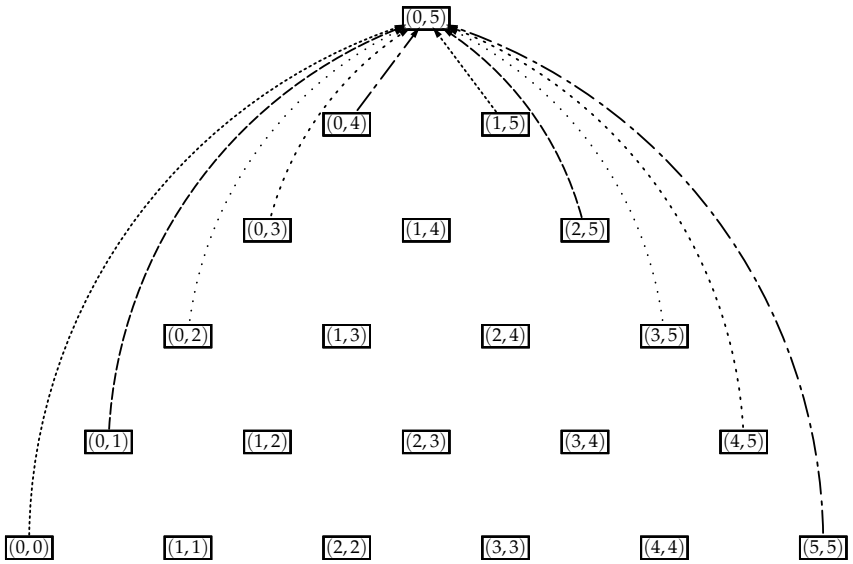


Figura 8.60: Pares de aristas relacionadas (dibujadas con el mismo trazo) e incidentes al vértice $(0,5)$ *en el grafo de dependencias.*

matrix_products_chain.py

```
def number_of_flops(dim):
    n = len(dim)
    P = dict( ((i,i), 0) for i in xrange(n) )
    for l in xrange(1, n):
        for i in xrange(0, n-l):
            P[i,i+l] = min(P[i,j] + dim[i][0]*dim[j][1]*dim[i+l][1] + P[j+1,i+l] for j in xrange(i,i+l))
    return P[0,n-1]
```

test_matrix_products_chain_a.py

```
from matrix_products_chain import number_of_flops
dim = [ (7,4), (4,3), (3,10), (10,7), (7,2), (2,8) ]
print 'Producto entre escalares:', number_of_flops(dim)
```

```
Producto entre escalares: 392
```

El coste temporal es $O(N^3)$ y el coste espacial $O(N^2)$.

EJERCICIOS

69 ¿Es posible reducir la complejidad espacial?

8.17.4. Recuperación de la parentización óptima

Podemos usar la técnica de los punteros hacia atrás para recuperar la parentización óptima. En principio parece que sean necesario dos punteros hacia atrás por cada vértice, pues la solución óptima con raíz en un vértice se forma con dos vértices antecesores. Pero no es necesario: basta con almacenar el valor de la segmentación óptima para ese vértice, pues con ella se pueden conocer los dos vértices implicados.

Por ejemplo, si la parentización de $M_0 \times M_1 \times M_2$ es $(M_0 \times M_1) \times M_2$ basta con almacenar el valor 1 en $B[0,2]$.

matrix_products_chain.py

```
def parenthesized_matrix_products_chain(dim):
    n = len(dim)
    P = {}
    B = {}
    for i in xrange(n):
        P[i,i], B[i,i] = 0, None
    for l in xrange(1, n):
        for i in xrange(0, n-l):
            P[i,i+l], B[i,i+l] = min( (P[i,j]+dim[i][0]*dim[j][1]*dim[i+l][1]+P[j+1,i+l], j)
                                                                    for j in xrange(i, i+l) )
    def backtrace(i, k):
        j = B[i,k]
        if j == None: return i
        else:         return [backtrace(i,j), backtrace(j+1,k)]
    return backtrace(0, n-1)
```

test_matrix_products_chain_b.py

```
from matrix_products_chain import parenthesized_matrix_products_chain
dim = [ (7,4), (4,3), (3,10), (10,7), (7,2), (2,8) ]
print 'Parentización óptima:', parenthesized_matrix_products_chain(dim)
```

```
Parentización óptima: [[0, [1, [2, [3, 4]]]], 5]
```

La solución de este ejemplo es el árbol de la figura 8.59.

8.18. Cálculo de los números de Fibonacci

Presentamos los números de Fibonacci en el apartado **??**. Recordemos su expresión recursiva:

$$F(n) = \begin{cases} 0, & \text{si } n = 0; \\ 1, & \text{si } n = 1; \\ F(n-2) + F(n-1), & \text{si } n > 1. \end{cases} \tag{8.23}$$

Es inmediato codificar esa expresión en Python:

fibonacci.py

```
def recursive_fibonacci(n):
    def F(n):
        if n == 0: return 0
        elif n == 1: return 1
        else: return F(n-2) + F(n-1)
    return F(n)
```

test_fibonacci1.py

```
from fibonacci import recursive_fibonacci

print "F(8) = %d " % recursive_fibonacci(8),
```

```
F(8) = 21
```

En la figura 8.61 se muestra el árbol de llamadas recursivas para el cálculo de $F(6)$. Un examen del árbol permite detectar que se efectúan muchas llamadas repetidas para un mismo valor del argumento: cinco llamadas a $F(0)$, ocho a $F(1)$, cinco a $F(2)$, tres a $F(3)$, dos a $F(4)$, una a $F(5)$ y una a $F(6)$. Estas repeticiones hacen que calcular $F(n)$ requiera tiempo que crece exponencialmente con n. El coste temporal puede expresarse en función de n con esta ecuación recursiva:

$$T(n) = \begin{cases} c_0, & \text{si } n \leq 1; \\ T(n-2) + T(n-1) + c_1, & \text{si } n > 1. \end{cases}$$

Es evidente que $T(n) \leq F(n)$, y $F(n)$, como vimos en el apartado **??**, crece exponencialmente con n. Por otra parte, el coste espacial es lineal con n, pues la máxima profundidad de la pila de llamadas es proporcional a n.

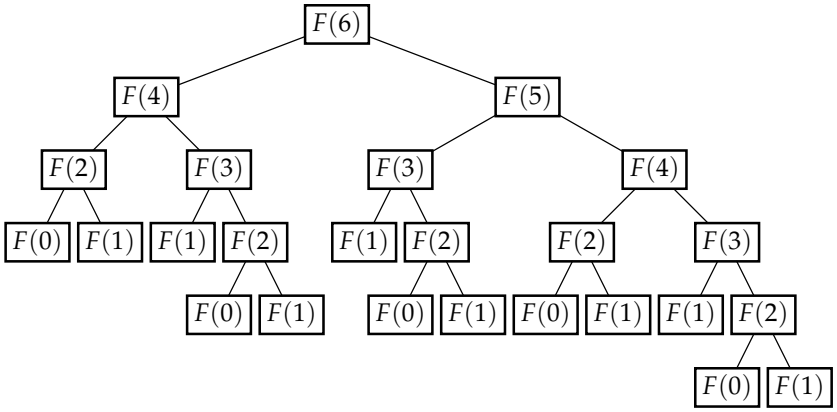


Figura 8.61: Árbol de llamadas recursivas para el cálculo de $F(6)$.

El problema no es de optimización, pero podemos aplicar las técnicas propias de la programación dinámica. Es absurdo repetir cálculos que van a producir el mismo resultado. Podríamos almacenar en una estructura de datos los valores que ya han sido calculados y evitar su cómputo cuando se solicite su valor.

fibonacci.py

```
1  def memoized_fibonacci(n):
2      R = {}
3      def F(n):
4          if n == 0:
5              R[n] = 0
6          elif n == 1:
7              R[n] = 1
8          else:
9              if n-2 not in R: F(n-2)
10             if n-1 not in R: F(n-1)
11             R[n] = R[n-2] + R[n-1]
12         return R[n]
13     return F(n)
```

test_fibonacci_with_memoization.py

```
1  from fibonacci import memoized_fibonacci
2
3  print "F(8) = %d " % memoized_fibonacci(8),
```

```
F(8) = 21
```

El árbol de llamadas recursivas para el cálculo de $F(6)$ se transforma ahora en el que se muestra en la figura 8.62.

La memorización ha reducido el coste temporal del algoritmo a $\Theta(n)$. El coste espacial sigue siendo $\Theta(n)$: sólo hemos reducido a la mitad la profundidad de la pila y usamos una estructura auxiliar con tamaño proporcional a n. Podemos tratar de reducir el coste espacial efectuando una transformación recursivo iterativa.

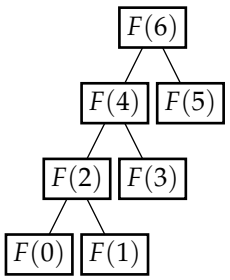


Figura 8.62: Árbol de llamadas recursivas para el cálculo de $F(6)$ con memorización.

Más que el árbol de llamadas recursivas, nos interesa considerar qué valores de la tabla R participan en el cálculo de qué valores de la misma tabla, o lo que es lo mismo, qué estados dependen de qué estados. La figura 8.63 muestra el grafo de dependencias inducido por la recurrencia.

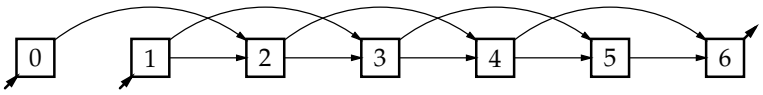


Figura 8.63: Grafo de dependencias entre estados al calcular $F(6)$.

El grafo inducido es acíclico, así que es posible encontrar un orden topológico entre los vértices. Es evidente que considerar los vértices por orden creciente del índice proporciona un orden topológico.

fibonacci.py

```
1 def fibonacci1(n):
2     F = [0] * (n+1)
3     if n >= 1: F[1] = 1
4     for i in xrange(2, n+1):
5         F[i] = F[i-2] + F[i-1]
6     return F[n]
```

test_iterative_fibonacci2.py

```
1 from fibonacci import fibonacci1
2
3 print "F(8) = %d " % fibonacci1(8),
```

```
F(8) = 21
```

No hemos conseguido, de momento, una reducción asintótica de la complejidad espacial o temporal. La complejidad espacial puede reducirse si observamos que, cuando estamos calculando $F[n]$ necesitamos los valores almacenados en $F[n-2]$ y $F[n-1]$, pero sólo esos. Es innecesario disponer de un vector para valores intermedios: basta con tres variables, una para el valor «actual», otra para el «anterior» y otra para el «anterior» a éste.

fibonacci.py

```
def fibonacci(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    anterior2 = 0
    anterior1 = 1
    for i in range(2, n+1):
        actual = anterior2 + anterior1
        anterior2 = anterior1
        anterior1 = actual
    return actual
```

test_iterative_fibonacci3.py

```
from fibonacci import fibonacci

print "F(8) = %d " % fibonacci(8),
```

```
F(8) = 21
```

El coste espacial de esta nueva versión es $O(1)$. La complejidad temporal sigue siendo $O(n)$.

EJERCICIOS

70 Deseamos calcular el número de combinaciones de m objetos tomados sobre un conjunto de n, notado como $\binom{n}{m}$, para $n \geq 1$ y $0 \leq m \leq n$, sabiendo que

$$\binom{n}{n} = \binom{n}{0} = 1,$$

y empleando cada una de las siguientes relaciones de recurrencia:

a)
$$\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1} \qquad \text{si } 0 < m < n,$$

b)
$$\binom{n+1}{m+1} = \sum_{0 \leq k \leq n} \binom{k}{m},$$

c)
$$\binom{r}{k} = \frac{r}{k}\binom{r-1}{k-1}, \qquad \text{si } k \neq 0,$$

d)
$$\binom{r}{k} = \frac{r}{r-k}\binom{r-1}{k}, \qquad \text{si } k \neq r.$$

Diseña algoritmos iterativos para el cómputo de $\binom{n}{m}$ a partir de cada una de las cuatro relaciones mostradas. Compara a continuación sus respectivas complejidades temporales y espaciales.

71 En una competición dos equipos A y B se enfrentan un máximo de $2n-1$ veces, ganando el primer equipo que acumula n victorias. Se supone que no hay partidos nulos y que el resultado de cada partido es independiente del resto. La probabilidad de que A gane un partido es p y de que lo gane B es $q = 1 - p$. ¿Cuál es la probabilidad de que A gane el campeonato?

8.19. Conteo del número de posibles desgloses en monedas

Ya hemos visto (sección 8.7) cómo calcular el desglose óptimo de una cantidad Q con un determinado juego de monedas. La ecuación recursiva

$$M(i,q) = \begin{cases} 0, & \text{si } i = 0 \text{ y } q = 0; \\ +\infty, & \text{si } i = 0 \text{ y } q > 0; \\ \min\limits_{0 \le x_i \le \lfloor q/v_i \rfloor} \Big(M(i-1, q - x_i v_i) + x_i \Big), & \text{en otro caso.} \end{cases}$$

nos indicaba el menor número de monedas con el que podíamos desglosar una cantidad q con monedas de valor $v_1, v_2, \ldots, v_i$. Nos preguntamos ahora cuántas formas hay de efectuar el desglose de la cantidad Q con N valores distintos de moneda.

$$P(i,q) = \begin{cases} 1, & \text{si } i = 0 \text{ y } q = 0; \\ 0, & \text{si } i = 0 \text{ y } q > 0; \\ \sum\limits_{0 \le x_i \le \lfloor q/v_i \rfloor} P(i-1, q - x_i v_i), & \text{en otro caso.} \end{cases} \tag{8.24}$$

El valor que deseamos conocer es $P(N, Q)$.

El grafo de dependencias inducido por esta recursión es básicamente el mismo que induce la anterior ecuación.

EJERCICIOS

72 Diseña un algoritmo de programación dinámica que calcula el número de posibles desgloses de moneda a partir de esta otra ecuación recursiva:

$$M(q) = \begin{cases} 0, & \text{si } q = 0; \\ \min\limits_{\substack{1 \le i \le N: \\ q - v_i \ge 0}} M(q - v_i) + 1, & \text{si } q > 0. \end{cases}$$

73 Diséñese un algoritmo de programación dinámica que compute el número de posibles parentizaciones con que es posible efectuar el producto matricial $M_0 \times M_1 \times \cdots \times M_{N-1}$.

74 Diseña un algoritmo que indique cuántos caminos hay que sean tan baratos como el más barato en el río Congo. ¿Hay alguna forma de conocerlos todos?

8.20. Análisis de una cadena con un autómata finito no determinista

Autómata finito no determinista: Non-deterministic finite automaton.

Un **autómata finito no determinista** (AFND) es una quintupla $A = (\Sigma, Q, q_0, E, F)$ donde

- Σ es un alfabeto de símbolo **terminales**;
- Q es un **conjunto** finito **de estados**;
- q_0 es un elemento de Q y se denomina **estado inicial**;
- $E : Q \times \Sigma \times Q$ es un **conjunto de transiciones entre estados**;
- y F es un subconjunto de F que recibe el nombre de **conjunto de estados finales**.

Los AFND pueden representarse mediante diagramas que representan los estados con vértices de un grafo y el conjunto de transiciones entre estados mediante aristas etiquetadas con símbolos de Σ. El estado inicial se señala con una flecha incidente y los estados finales con un doble círculo. La figura 8.64 muestra el diagrama correspondiente al autómata A definido con la quintupla (Σ, Q, q_0, E, F), donde

- $\Sigma = \{a, b\}$,
- $Q = \{0, 1, 2, 3, 4, 5\}$,
- $q_0 = 0$,
- $E = \{(0, a, 1), (0, b, 2), (1, a, 1), (1, a, 2), (1, b, 3), (2, a, 4), (3, a, 5), (4, b, 2), (4, a, 3), (4, b, 5), (5, a, 5)\}$,
- y $F = \{4, 5\}$.

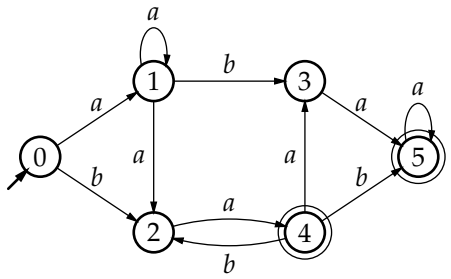


Figura 8.64: Representación de un autómata finito no determinista.

Una cadena $x \in \Sigma^*$ es **aceptada** por un autómata A si hay una secuencia de transiciones $(q_0, x_1, q_1), (q_1, x_2, q_2), \ldots, (q_{n-1}, x_n, q_n)$ tal que q_0 es el estado inicial y q_n es un estado final cualquiera. El lenguaje aceptado por un autómata A es el conjunto de cadenas que acepta y se denota con $L(A)$.

Dada una cadena x y un autómata A, deseamos saber si x pertenece a $L(A)$.

8.20.1. Ecuación recursiva

Denotemos con $L(q)$, para todo q de Q, al lenguaje $L((\Sigma, Q, q_0, E, \{q\}))$. El conjunto $L(q)$ puede definirse recursivamente:

$$L(q) = \bigcup_{(q', a, q) \in E} \{xa \mid x \in L(q')\}.$$

Como el autómata puede presentar ciclos de los que forme parte el estado inicial, no es posible encontrar una expresión cerrada para $L(q_0)$. Pero sí podemos afirmar que la cadena vacía, λ, pertenece a $L(q_0)$.

Sea $P(i, q)$ una función que vale «cierto» si $x_{1:i} \in L(q)$. A partir de la definición recur-

> **!** Estamos buscando un «camino» en el AFND que use $|x|$ aristas. La ecuación recursiva es reminiscente de la del cálculo del camino óptimo en un grafo (con o sin ciclos) que usa k aristas. Basta con que haya un camino, de ahí que se use la o-lógica: si hemos podido llegar con $x_{1:i-1}$ a algún estado q', es suficiente con que haya una transición (q', x_i, q) para que el estado q sea «alcanzable» con $x_{1:i}$.

siva de $L(q)$ llegamos a

$$P(i,q) = \begin{cases} q = q_0, & \text{si } i = 0; \\ \bigvee\limits_{(q',x_i,q)\in E} P(i-1,q'), & \text{si } i > 0; \end{cases}$$

donde $\bigvee$ calcula la o-lógica de varios valores.

La pertenencia de x a $L(A)$ se determina calculando

$$\bigvee_{q\in F} P(|x|,q).$$

8.20.2. Algoritmo iterativo

La ecuación recursiva induce un grafo de dependencias como el que se muestra, para un caso particular, en la figura 8.65.

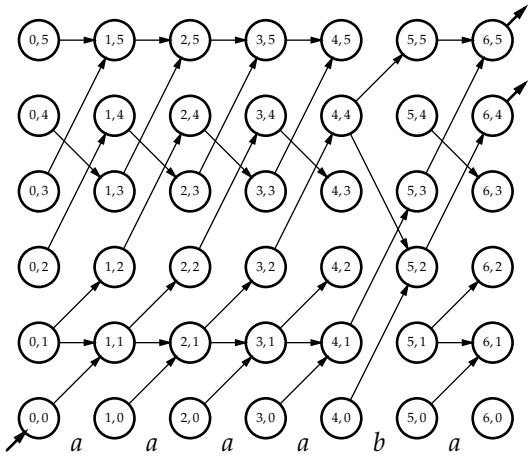


Figura 8.65: Grafo de dependencias al analizar la cadena aaaaba con el autómata de la figura 8.64.

afnd.py

```
1 def accepts(x, Q, q0, preds, F):
2     P = {}
3     for q in Q:
4         P[q,0] = False
5     P[q0,0] = True
6     for i in xrange(1, len(x)+1):
7         for q in Q:
```

```
            P[q,i] = False
            for (q1, c) in preds[q]:
                if c == x[i-1] and P[q1,i-1]:
                    P[q,i] = True
                    break
    accepted = False
    for q in F:
        if P[q,len(x)]:
            accepted = True
            break
    return accepted
```

test_afnd.py

```
from afnd import accepts

Q = range(6)
q0 = 0
F = [4,5]
preds = {0: [],                                  1: [(0,'a'), (1,'a')],
         2: [(0,'b'), (1,'b'), (4,'b')],         3: [(1,'b'), (4,'a')],
         4: [(2,'a')],                           5: [(3,'a'), (4,'b'), (5,'a')]}
for x in ['aaaaba', 'aa', 'aaababaa', 'abaaaaab']:
    print '¿Pertenece %s a L(A)?:' % x, accepts(x, Q, q0, preds, F)
```

```
¿Pertenece aaaaba a L(A)?: True
¿Pertenece aa a L(A)?: False
¿Pertenece aaababaa a L(A)?: True
¿Pertenece abaaaaab a L(A)?: False
```

El coste temporal del algoritmo es $O(|x||Q|^2)$ y el coste espacial es $O(|x||Q|)$.

8.20.3. ¿No es un problema de optimización?

Ciertamente no parece que hayamos resuelto un problema de optimización. Aunque esta visión del problema cambia si consideramos que los valores booleanos están ordenados y «falso» es menor que «cierto». En tal caso, el operador $\vee$ actúa sobre los booleanos como el operador «máx» sobre los reales, así que el problema sí puede considerarse de optimización.

De hecho, podemos convertir el problema en un problema de optimización si consideramos esta ecuación recursiva alternativa:

$$P(i,q) = \begin{cases} 0, & \text{si } i = 0 \text{ y } q \neq q_0; \\ 1, & \text{si } i = 0 \text{ y } q = q_0; \\ \max\limits_{(q',x_i,q)\in E} P(i-1,q'), & \text{si } i > 0. \end{cases} \tag{8.25}$$

La cadena $x_{1:i}$ pertenece a $L(q)$ si $P(i,q) = 1$. Se trata, pues, de un problema de optimización.

EJERCICIOS

75 Un AFND con λ-transiciones es un AFND cuyo conjunto de transiciones entre estados es un subconjunto de $Q \times (\Sigma \cup \{\lambda\}) \times Q$. Cuando, al analizar una cadena, se alcanza un estado q, todo estado q' tal que $(q, \lambda, q') \in E$ es alcanzado.

Diseña un algoritmo de programación dinámica que determine la pertenencia de una cadena x al lenguaje aceptado por un AFND con λ-transiciones.

76 Diseña un algoritmo de programación dinámica que determine el número de secuencias de estados que permiten analizar una cadena x con un AFND.

77 Sea $A = (\Sigma, Q, q_0, E, F, \Pr)$ un autómata finito *estocástico* donde

- Σ es un alfabeto,
- Q es un conjunto finito de estados,
- q_0 es el estado inicial,
- $E : Q \times \Sigma \times Q$ es el conjunto de transiciones entre estados;
- F es el conjunto de estados finales que, por simplicidad, supondremos $F = \{q_f\}$,
- $\Pr : E \mapsto \mathbb{R}$ es una función tal que $\Pr(q', a \mid q)$ es la probabilidad de que se emita el símbolo a y se transite al estado q' al encontrarse en el estado q. Esta función observa

$$\sum_{q' \in Q} \sum_{a \in \Sigma} \Pr(q', a \mid q) = 1$$

para todo $q \in Q$.

Sea $L(A)$ el lenguaje aceptado por el autómata $(\Sigma, Q, q_0, E, F, \Pr)$ y sea x una cadena de Σ^*. Diseña algoritmos que calculen:

a) La probabilidad de que una cadena x pertenezca a $L(A)$. (La probabilidad es la suma de probabilidades de toda secuencia de estados con la que es posible aceptar x.)

b) La secuencia de estados que con mayor probabilidad reconoce la cadena x, si la probabilidad de que x pertenezca a $L(A)$ es mayor que cero.

78 Es posible definir la distancia de una cadena a un AFND A como la distancia de Levenshtein de dicha cadena a la cadena de $L(A)$ más próxima. Diseña un algoritmo que permita obtener la distancia de una cadena a un AFND.

8.21. Análisis de una cadena con una gramática incontextual en forma normal de Chomsky: el algoritmo de Cocke-Younger-Kasami

Gramática incontextual: Context-Free Grammar.

Una **gramática incontextual en forma normal de Chomsky** es una cuadrupla $G = (\Sigma, N, S, P)$ donde

- Σ es un alfabeto de símbolos **terminales**;

Forma normal de Chomsky: Chomsky Normal Form.

- N es un alfabeto de símbolo **no terminales**;
- S es un símbolo de N al que denominamos **símbolo inicial**;
- P es un subconjunto de $(N \times (N \times N)) \cup (N \times \Sigma)$ a cuyos elementos denominamos **reglas** o **producciones**.

Símbolo inicial: Start symbol.

En adelante usaremos letras minúsculas para referirnos a símbolos de Σ, y mayúsculas para referirnos a símbolos de N. Las tuplas de P serán, pues, de la forma (A, B, C) o (A, a), si bien se denotarán, respectivamente, con $A \to BC$ y $A \to a$.

Reservamos la letras del alfabeto griego para representar cadenas de $(N \cup \Sigma)^+$. Dada una cadena $\gamma = \alpha A \eta$ decimos que **deriva directamente** $\alpha\beta\eta$, y lo denotamos con $\gamma \Rightarrow \alpha\beta\eta$, si $A \to \beta \in P$. Denominamos parte izquierda y parte derecha de una regla $A \to \beta$ a A y β, respectivamente.

Decimos que γ **deriva** η, y lo denotamos con $\gamma \Rightarrow^* \eta$, si existe una secuencia β_1, β_2, …, β_n de cadenas de $(N \cup \Sigma)^+$ tal que $\gamma = \beta_1$, $\beta_n = \eta$ y $\beta_1 \Rightarrow \beta_2$, $\beta_2 \Rightarrow \beta_3$, …, $\beta_{n-1} \Rightarrow \beta_n$. La secuencia $\beta_1, \beta_2, \ldots, \beta_n$ es una **derivación** de γ en η.

El lenguaje generado por una gramática G es $L(G) = \{x \in \Sigma^* \mid S \Rightarrow^* x\}$, es decir, el conjunto de cadenas de terminales que deriva de S.

Dada una cadena $x \in \Sigma^*$ deseamos saber si $x \in L(G)$ y, de ser así, conocer la derivación de S que la genera.

Conviene introducir un concepto adicional: el de **árbol de análisis**. Un árbol de análisis de una cadena $x \in \Sigma$ para una gramática incontextual en forma normal de Chomsky es un árbol

Árbol de análisis: Parse tree.

- cuyas hojas están etiquetadas con símbolos de Σ y, tomadas de izquierda a derecha, forman la cadena x;
- cuyos nodos internos están etiquetados con símbolos de N y tienen, cada uno, uno o dos hijos:
  - si son dos, sus etiquetas, que son símbolos de N, corresponden a la parte derecha de una regla cuya parte izquierda es la etiqueta del nodo padre;
  - y si es uno, éste es un nodo hoja cuya etiqueta es la parte derecha de una regla cuya parte izquierda es la etiqueta del nodo padre;
- y cuya raíz está etiquetada con el símbolo S.

Toda derivación de S en una cadena $x \in \Sigma^*$ puede representarse mediante un árbol de análisis.

Ilustremos los diferentes conceptos y el planteamiento del problema con un ejemplo. Sea G la gramática (Σ, N, S, P) donde $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, $N = \{S, A, B, C\}$ y P está formado por las siguientes producciones:

$$S \to AB, \qquad S \to AC, \qquad C \to SB, \qquad A \to \mathtt{a}, \qquad B \to \mathtt{b}.$$

La derivación directa $ASB \Rightarrow AABB$ forma parte de la derivación

$$S \Rightarrow AC \Rightarrow ASB \Rightarrow AABB \Rightarrow \mathtt{a}ABB \Rightarrow \mathtt{a}A\mathtt{b}B \Rightarrow \mathtt{aab}B \Rightarrow \mathtt{aabb}.$$

Esta derivación puede representarse con el árbol de análisis de la figura 8.66.

La cadena `aabb` es derivable desde S, así que forma parte del lenguaje $L(G)$.

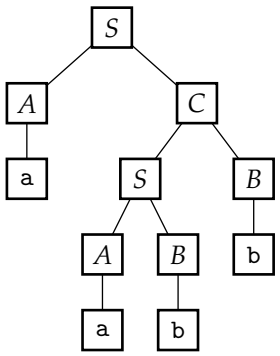


Figura 8.66: Árbol de análisis para la cadena `aabb`.

Podemos reformular, pues, el problema así: «Dada una cadena $x \in \Sigma^*$ deseamos saber si $x \in L(G)$ y, de ser así, conocer un árbol de análisis que la genere.» Nótese que decimos «un árbol de análisis» y no «el árbol de análisis». Una gramática incontextual es **ambigua** si una misma cadena puede derivarse de S con dos o más árboles de análisis diferentes.

El problema que deseamos resolver se plantea en estos términos: dada una cadena x, ¿pertenece a $L(G)$?; si es así, encuéntrese un árbol de análisis para la cadena.

8.21.1. Ecuación recursiva

Extendemos el concepto de lenguaje generado por G al de lenguaje generado por un no terminal A de G:

$$L(A) = \{x \in \Sigma^* \mid A \Rightarrow^* x\}.$$

Evidentemente, $L(G) = L(S)$.

Si la cadena x es de talla unitaria, resulta trivial decidir si pertenece o no a $L(A)$, para cualquier A en N: basta con comprobar si $A \to x$ es una producción de P.

Una cadena x de talla 2 o superior pertenece a $L(A)$ si y sólo si podemos partirla en dos fragmentos, $x_{1:k}$ y $x_{k+1:|x|}$ de modo que $x_{1:k} \in L(B)$, $x_{k+1:|x|} \in L(C)$, y $A \to BC \in P$.

Podemos definir una función $\Pi(1, |x|, A)$ que devuelve el valor booleano «cierto» si $x \in L(A)$ y «falso» en caso contrario. Dicha función puede definirse recursivamente:

$$\Pi(i,j,A) = \begin{cases} A \to x_i \in P, & \text{si } j - i = 1; \\ \bigvee\limits_{A \to BC} \bigvee\limits_{i<k<j} \big(\Pi(i,k,B) \wedge \Pi(k+1,j,C)\big), & \text{si } j - i > 1. \end{cases} \tag{8.26}$$

Nuestro objetivo es averiguar si $x \in L(S)$, es decir, calcular $\Pi(1, |x|, S)$.

8.21.2. Algoritmo iterativo

El grafo de dependencias se muestra en la figura 8.67. Al igual que ocurría con el cálculo de la parentización óptima del producto de matrices, no buscamos un camino óptimo en el grafo, sino un árbol (de hecho, un árbol de análisis). El grafo recuerda al de aquel problema, aunque con más vértices y una disposición de aristas algo más elaborada: es

como si se hubiera replicado aquel grafo una vez por cada no terminal y sólo se unieran vértices de la forma (i, k, A) con pares de vértices de la forma $((i, j, B), (j + 1, k, C))$ si existe una producción $A \to BC$.

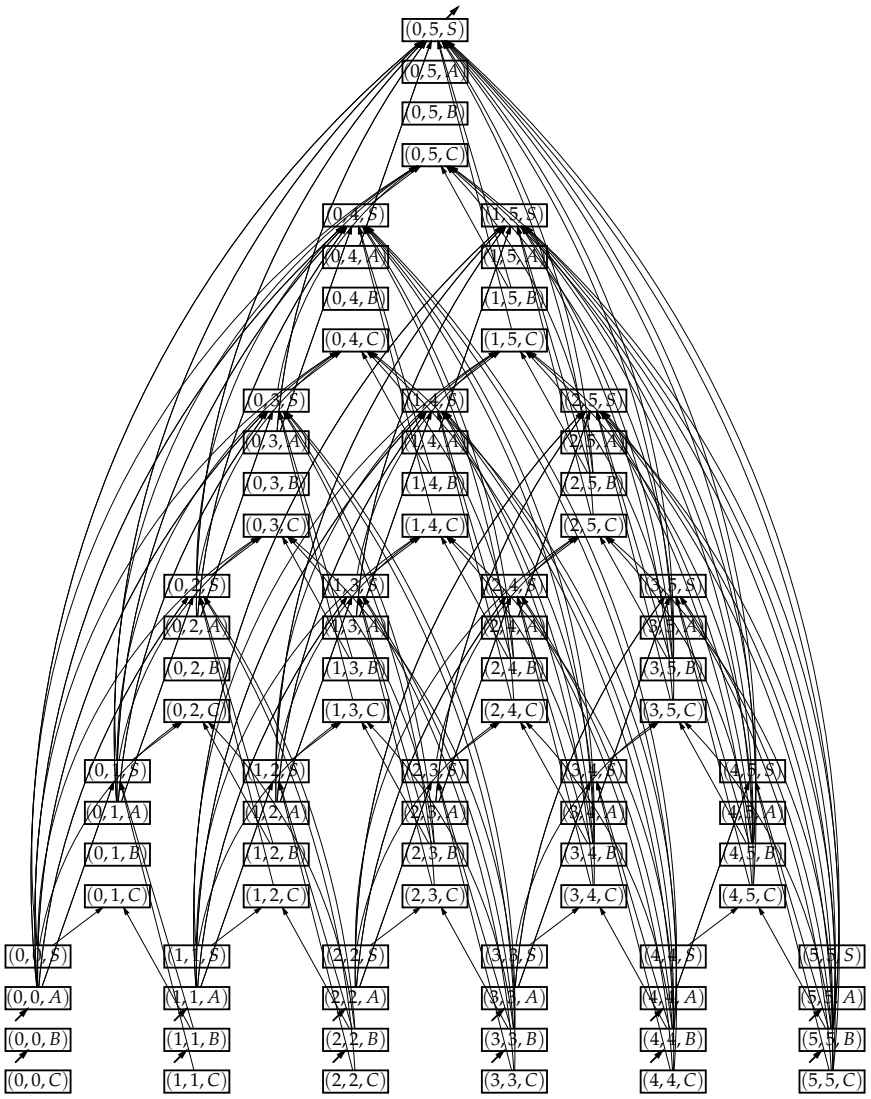


Figura 8.67: Grafo de dependencias para el algoritmo CYK.

A la vista del grafo de dependencias está claro que podemos recorrer los vértices por niveles, de abajo a arriba.

cyk.py

```
1 def CYK(x, P):
2     n = len(x)
3     R = {}
4     for i in xrange(len(x)):
5         for A in P:
6             R[i,i,A] = False
```

```
            for right in P[A]:
                if len(right) == 1 and right[0] == x[i]:
                    R[i,i,A] = True
                    break
    for l in xrange(1, n):
        for i in xrange(0, n-l):
            for A in P:
                R[i,i+l,A] = False
                for right in P[A]:
                    if len(right) == 2:
                        for k in xrange(i,i+l):
                            if R[i,k,right[0]] and R[k+1,i+l,right[1]]:
                                R[i,i+l,A] = True
                                break
    return R[0,n-1,'S']
```

test_cyk_a.py

```
from cyk import CYK

P = {'S': [['A','B'], ['A','C']],
     'C': [['S','B']],
     'A': [['a']],
     'B': [['b']]}
for x in 'aaabbb', 'aabbb', 'ab', 'aaaa':
    print '¿Pertenece %s a L(G)?' % x, CYK(x, P)
```

```
¿Pertenece aaabbb a L(G)? True
¿Pertenece aabbb a L(G)? False
¿Pertenece ab a L(G)? True
¿Pertenece aaaa a L(G)? False
```

El algoritmo presenta un coste temporal $O(|x^3||P|)$, o sea, cúbico con la talla de la cadena y proporcional al número de producciones (en realidad, al número mayor de producciones con una misma parte izquierda). El coste espacial es $O(|x|^2|N|)$.

EJERCICIOS

79 Reformula el problema en términos de un problema de optimización de modo similar a como hicimos en el apartado 8.20.3 cuando consideramos el problema de la pertenencia de una cadena al lenguaje aceptado por un AFND.

8.21.3. Recuperación del árbol óptimo

Podemos usar la técnica de punteros hacia atrás. Si en la celda de R asociada a un vértice se almacena el valor *True* es porque hay dos vértices ligados por un par de aristas con valor *True*. El puntero hacia atrás asociado a cada vértice debe almacenar suficiente información como para saber qué dos vértices le permitieron la asignación del valor *True*.

cyk.py *(cont.)*

```
def parse_tree(x, P):
    n = len(x)
    R = {}
    B = {}
    for i in xrange(len(x)):
        for A in P:
            R[i,i,A] = False
            B[i,i,A] = None
            for right in P[A]:
                if len(right) == 1 and right[0] == x[i]:
                    R[i,i,A] = True
                    B[i,i,A] = x[i]
                    break
    for l in xrange(1, n):
        for i in xrange(0, n-l):
            for A in P:
                R[i,i+l,A] = False
                B[i,i+l,A] = None
                for right in P[A]:
                    if len(right) == 2:
                        for k in xrange(i,i+l):
                            if R[i,k,right[0]] and R[k+1,i+l,right[1]]:
                                R[i,i+l,A] = True
                                B[i,i+l,A] = (k, right[0], right[1])
                                break
    def backtrace(i, k, A):
        if len(B[i,k,A]) == 1:
            return B[i,k,A]
        else:
            j, B', C' = B[i,k,A]
            return [A, backtrace(i,j,B'), backtrace(j+1,k,C')]
    return backtrace(0, n-1, 'S')
```

test_cyk_b.py

```
from cyk import CYK, parse_tree

P = {'S': [['A','B'], ['A','C']],
     'C': [['S','B']],
     'A': [['a']],
     'B': [['b']]}
for x in ['aaabbb', 'aabbb', 'ab', 'aaaa']:
    print '¿Pertenece %s a L(G)?' % x, CYK(x,P)
    if CYK(x,P): print 'Árbol de análisis:', parse_tree(x, P)
```

```
¿Pertenece aaabbb a L(G)? True
Árbol de análisis: ['S', 'a', ['C', ['S', 'a', ['C', ['S', 'a', 'b'], 'b']], 'b'
]]
¿Pertenece aabbb a L(G)? False
¿Pertenece ab a L(G)? True
```

```
Árbol de análisis: ['S', 'a', 'b']
¿Pertenece aaaa a L(G)? False
```

EJERCICIOS

80 Diseña un algoritmo que nos diga si una cadena x puede ser analizada con una gramática incontextual en forma normal de Chomsky con más de un árbol de análisis.

81 Dados un número natural L y N objetos de longitudes $l_1, l_2, \ldots, l_N$, donde $l_n \in \mathbb{N}^{>0}$ para todo n entre 1 y N, se desea saber si existe un subconjunto de los N objetos cuya suma de longitudes sea exactamente L.

a) Diseña la ecuación recursiva de un algoritmo de programación dinámica que devuelva «cierto» si existe tal subconjunto y «falso» en caso contrario.

b) Diseña un algoritmo iterativo para la ecuación del apartado anterior con un coste espacial de $O(L)$. Calcula el coste temporal.

c) Modifica el algoritmo iterativo anterior para que, adicionalmente, se obtenga el subconjunto de objetos (en caso de que exista). Calcula el coste temporal y espacial del algoritmo resultante.

82 Disponemos de una cisterna con una cantidad ilimitada de agua y deseamos llenar una cuba con exactamente M litros. Disponemos de un juego de jarras numeradas de 1 a J. La jarra número i tiene una capacidad de L_i litros, siendo dicha cantidad un número entero. Si sólo es posible trasvasar agua de la cisterna a la cuba y se debe llenar completamente cada jarra usada, ¿podemos lograr nuestro objetivo con exactamente K operaciones de trasvase?

Diseña un algoritmo de programación dinámica que solucione el problema y, caso de que sea posible efectuar el trasvase en K etapas, detalla las operaciones de trasvase.

83 Sean S_1 y S_2 dos casas de suministro de tornillos que sirven a n destinos, $D_1, D_2, \ldots, D_n$. Sea d_i la demanda de tornillos del destino D_i, y sea r_j el número de tornillos en inventario en la casa S_j. Supondremos que $r_1 + r_2$ es igual a $\sum_{1 \leq i \leq n} d_i$. Un envío de x tornillos de la casa S_j al destino D_i cuesta $c_{ij}(x)$ euros. Diséñese un algoritmo que permita ofrecer el suministro más barato posible a todos los destinos.

84 A lo largo del río Amazonas hay N poblados en los que podemos hacer escala. El poblado i está separado del poblado $i+1$ por una distancia en kilómetros que podemos conocer consultando el valor $d(i)$. Sabemos además que es imposible nadar más de K kilómetros sin parar a descansar en un poblado. (Suponemos $d(i) \leq K$, para $1 \leq i \leq N$.)

Encuéntrese un algoritmo que proporcione la lista de poblados que hay que visitar al ir del poblado 1 al N en cada uno de estos supuestos:

a) Deseamos realizar el recorrido visitando el menor número posible de poblados.

b) Si por cortesía hemos de estar en un pueblo i al menos tantos días como nos indica $c(i)$, el recorrido deseado es aquel que requiere el menor número posible de días.

c) Con el mismo supuesto del anterior apartado, deseamos conocer el camino más rápido que visita exactamente M poblados.

85 Sea $G = (V, E)$ un grafo dirigido y ponderado por $p : E \to \mathbb{R}^{\geq 0}$, donde $V = \{0, 1, 2, \ldots, N\}$; $E = \{(i, j) \in V^2 \mid a \leq j - i \leq b\}$; siendo a, b números naturales tales que $a < b < N$.

a) Diseña un algoritmo de complejidad temporal $O(N(b-a))$ para calcular el peso del camino mínimo de 0 a N.

b) Diseña un algoritmo que obtenga la secuencia de nodos visitados.

86 Dados dos enteros positivos N y K, una K-factorización de N es una serie de K enteros positivos, m_1, m_2,..., m_K, tal que $N = \prod_{1 \leq k \leq K} m_k$. Por ejemplo: si N vale 12 y K vale 3, los valores 1, 3 y 4 son una 3-factorización de 12. Diseña un algoritmo de programación dinámica que encuentre una K-factorización de N tal que el valor de $\sum_{1 \leq k \leq K} m_k$ sea mínimo. Estudia el coste espacial y temporal del algoritmo.

87 Se desea cargar un barco cuya capacidad máxima es de P kilos. La carga potencial consiste en un conjunto de n contenedores. Cada contenedor i, para $1 \leq i \leq n$, tiene un peso p_i. La suma total de los pesos de los n contenedores puede ser mayor que P.

Desarrollar un algoritmo de programación dinámica que determine cuál es el conjunto de contenedores que se puede cargar de forma que el peso cargado sea máximo. Plantea una ecuación recursiva que solucione el problema y estudia la complejidad computacional (espacial y temporal) del algoritmo iterativo resultante.

88 La universidad prepara la oferta académica para el próximo curso, titulación por titulación. La docencia de la titulación i-ésima del catálogo se debe cubrir con una cantidad de créditos comprendida entre c_i y C_i, para $1 \leq i \leq T$, siendo T el número total de titulaciones. La universidad ha estimado el número de alumnos matriculados que cabe esperar si asigna a cada titulación un determinado número de créditos. Así, $a(i, j)$ es el número esperado de alumnos que se matricularán al asignar j créditos a la titulación i. Diseña un algoritmo de programación dinámica que indique el número de alumnos máximo que se matricularán sabiendo que se tiene profesorado para impartir un total de K créditos.

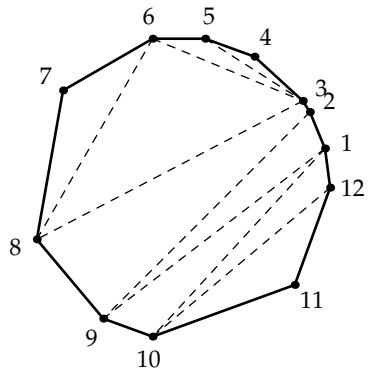
89 Disponemos de un conjunto de n claves (cadenas de caracteres) *ordenadas alfabéticamente*, cada una con un valor asociado. En la fase de explotación de cierta aplicación, la probabilidad de acceso a la información con la clave i-ésima es p_i. Deseamos organizar los pares clave/valor en un árbol binario de búsqueda de modo que se garantice el acceso más rápido posible a cada valor mediante su clave.

Nótese que la función objetivo que deseamos minimizar es, básicamente, la presentada al calcular el código Huffman, pero que no nos enfrentamos al mismo problema: en éste queremos mantener ordenado el conjunto de claves.

90 El ministerio de turismo no ha quedado satisfecho con nuestro estudio sobre el trayecto más probable en el río Congo. Sospecha que los turistas no deciden qué trayecto realizar sólo en función del último embarcadero visitado, sino de los dos últimos embarcaderos visitados. La probabilidad de ir al embarcadero k si se acaban de visitar los embarcaderos i y j, por ese orden, es $p(i \mid j, k)$. Diseña un algoritmo que calcule el trayecto que más probablemente realizará un turista con el nuevo modelo estadístico.

91 Sean x e y dos cadenas de caracteres sobre un alfabeto Σ. Se quiere transformar x en y utilizando las operaciones de edición convencionales y una adicional: la transposición de dos caracteres. Es una operación que denotamos con $ab \rightarrow ba$, para cualquier par de caracteres a, y b de Σ tales que $a \neq b$. Desarrolla un algoritmo de programación dinámica que encuentre el número mínimo de operaciones necesarias para transformar a en b. Calcula el coste computacional de tu propuesta.

92 Una triangulación de un polígono convexo es una descomposición del mismo en triángulos disjuntos cuyos vértices son también vértices del polígono.



Dado un polígono convexo descrito por sus n vértices ordenados en sentido antihorario, (x_1, y_1), (x_2, y_2), ..., (x_n, y_n), diseña un algoritmo de programación dinámica que calcule la triangulación cuya suma de perímetros de los triángulos resultantes sea mínima.

93 Disponemos de N objetos de pesos enteros $w_1, w_2, \ldots, w_N$. ¿Es posible disponerlos todos en los dos platos de una balanza de modo que esta esté en perfecto equilibrio? Diseña un algoritmo de programación dinámica y analiza su coste computacional.