

Ejemplos de programación dinámica

Problema de los intereses de banco

Se tiene una cantidad M , que se debe invertir en n bancos. Cada banco cuenta con una función de interés $f_1, f_2, f_3, \dots, f_n$, donde $f_i(0) = 0, (i = 1, \dots, n)$. De qué forma se debe repartir el capital de tal forma que la suma de todos los intereses generados por los bancos $(f_1(x_1), f_2(x_2), f_3(x_3), \dots, f_n(x_n))$ sea máxima?

Solución

Sea f_i un vector que almacena el interés del banco i ($1 \leq i \leq n$) para una inversión de 1, 2, 3, ..., M pesos. Esto es, $f_i(j)$ indicará el interés que ofrece el banco i para j pesos, con $0 < i \leq n, 0 < j \leq M$.

Para poder plantear el problema como una sucesión de decisiones, llamaremos $I_n(M)$ al interés máximo al invertir M pesos en n bancos,

$$I_n(M) = f_1(x_1) + f_2(x_2) + \dots + f_n(x_n)$$

que es la función a maximizar, sujeta a la restricción $x_1 + x_2 + \dots + x_n = M$.

Veamos cómo aplicar el principio de óptimo. Si $I_n(M)$ es el resultado de una secuencia de decisiones y resulta ser óptima para el problema de invertir una cantidad M en n bancos, cualquiera de sus subsecuencias de decisiones ha de ser también óptima y así la cantidad

$$I_{n-1}(M - x_n) = f_1(x_1) + f_2(x_2) + \dots + f_{n-1}(x_{n-1})$$

será también óptima para el subproblema de invertir $(M - x_n)$ pesos en $n - 1$ bancos. Y por tanto el principio de óptimo nos lleva a plantear la siguiente relación en recurrencia:

$$I_n(x) = \begin{cases} f_1(x) & \text{si } n = 1 \\ \text{Max}_{0 \leq t \leq x} \{I_{n-1}(x-t) + f_n(t)\} & \text{en otro caso.} \end{cases}$$

Para resolverla y calcular $I_n(M)$, vamos a utilizar una matriz I de dimensión $n \times M$ en donde iremos almacenando los resultados parciales y así eliminar la repetición de los cálculos. El valor de $I[i, j]$ va a representar el interés de j pesos cuando se dispone de i bancos, por tanto la solución buscada se encontrará en $I[n, M]$. Para guardar los datos iniciales del problema vamos a utilizar otra matriz F , de la misma dimensión, y donde $F[i, j]$ representa el interés del banco i para j pesos.

En consecuencia, para calcular el valor pedido de $I[n, M]$ rellenaremos la tabla por filas, empezando por los valores iniciales de la ecuación en recurrencia, y según el siguiente algoritmo:

```

CONST n = ...; (* numero de bancos *)
M = ...; (* cantidad a invertir *)
TYPE MATRIZ = ARRAY [1..n],[0..M] OF CARDINAL;
PROCEDURE Intereses(VAR F:MATRIZ;VAR I:MATRIZ):CARDINAL;
VAR i,j: CARDINAL;
BEGIN
    FOR i:=1 TO n DO I[i,0]:=0 END;
    FOR j:=1 TO M DO I[1,j]:=F[1,j] END;
    FOR i:=2 TO n DO
        FOR j:=1 TO M DO
            I[i,j]:=Max(I,F,i,j)
        END
    END;
RETURN I[n,M]
END Intereses;

```

La función *Max* es la que calcula el máximo

```

PROCEDURE Max(VAR I,F:MATRIZ;i,j:CARDINAL):CARDINAL;
VAR max,t:CARDINAL;
BEGIN
    max:= I[i-1,j] + F[i,0];
    FOR t:=1 TO j DO
        max:=Max2(max,I[i-1,j-t]+F[i,t])
    END;
RETURN max
END Max;

```

La función *Max2* es la que calcula el máximo de dos números naturales. La complejidad del algoritmo completo es de orden $O(nM^2)$, puesto que la complejidad de *Max* es $O(j)$ y se invoca dentro de dos bucles anidados que se desarrollan desde 1 hasta M . Es importante hacer notar el uso de parámetros por referencia en lugar de por valor para evitar la copia de las matrices en la pila de ejecución del programa. Por otro lado, la complejidad espacial del algoritmo es del orden $O(nM)$, pues de este orden son las dos matrices que se utilizan para almacenar los resultados intermedios. En este ejemplo queda de manifiesto la efectividad del uso de estructuras en los algoritmos de Programación Dinámica para conseguir obtener tiempos de ejecución de orden polinómico, frente a los tiempos exponenciales de los algoritmos recursivos iniciales.

El viaje más barato por río

Sobre el río Guadalhorce hay n embarcaderos. En cada uno de ellos se puede alquilar un bote que permite ir a cualquier otro embarcadero río abajo (es imposible ir río arriba). Existe una tabla de tarifas que indica el coste del viaje del embarcadero i al j

para cualquier embarcadero de partida i y cualquier embarcadero de llegada j más abajo en el río ($i < j$). Puede suceder que un viaje de i a j sea más caro que una sucesión de viajes más cortos, en cuyo caso se tomaría un primer bote hasta un embarcadero k y un segundo bote para continuar a partir de k . No hay coste adicional por cambiar de bote.

Nuestro problema consiste en diseñar un algoritmo eficiente que determine el coste mínimo para cada par de puntos i, j ($i < j$) y determinar, en función de n , el tiempo empleado por el algoritmo.

Solución

Llamaremos $T[i, j]$ a la tarifa para ir del embarcadero i al j (directo). Estos valores se almacenarán en una matriz triangular superior de orden n , siendo n el número de embarcaderos.

El problema puede resolverse mediante Programación Dinámica ya que para calcular el coste óptimo para ir del embarcadero i al j podemos hacerlo de forma recurrente, suponiendo que la primera parada la realizamos en un embarcadero intermedio

k ($i < k \leq j$):

$$C(i, j) = T(i, k) + C(k, j).$$

En esta ecuación se contempla el viaje directo, que corresponde al caso en el que k coincide con j . Esta ecuación verifica también que la solución buscada $C(i, j)$ satisface el principio del óptimo, pues el coste $C(k, j)$, que forma parte de la solución, ha de ser, a su vez, óptimo. Podemos plantear entonces la siguiente expresión de la solución:

$$C(i, j) = \begin{cases} 0 & \text{si } i = j \\ \text{Min}_{i < k \leq j} \{T(i, k) + C(k, j)\} & \text{si } i < j \end{cases}$$

La idea de esta segunda expresión surge al observar que en cualquiera de los trayectos siempre existe un primer salto inicial óptimo. Para resolverla según la técnica de Programación Dinámica, hace falta utilizar una estructura para almacenar resultados intermedios y evitar la repetición de los cálculos. La estructura que usaremos es una matriz triangular de costes $C[i, j]$, que iremos rellenando por diagonales mediante el procedimiento que hemos denominado *Costes*. La solución al problema es la propia tabla, y sus valores $C[i, j]$ indican el coste óptimo para ir del embarcadero i al j .

```
CONST MAXEMBARCADEROS = ...;
TYPE MATRIZ=ARRAY[1..MAXEMBARCADEROS],[1..MAXEMBARCADEROS] OF
CARDINAL;
PROCEDURE Costes(VAR C:MATRIZ;n:CARDINAL);
VAR i, diagonal:CARDINAL;
BEGIN
    FOR i:=1 TO n DO C[i,i]:=0 END; (* condiciones iniciales *)
```

```

    FOR diagonal:=1 TO n-1 DO
      FOR i:=1 TO n-diagonal DO
        C[i,i+diagonal]:=Min(C,i,i+diagonal)
      END
    END
  END
END Costes;

```

Dicho procedimiento utiliza la siguiente función, que permite calcular la expresión del mínimo

```

PROCEDURE Min(VAR C:MATRIZ; i,j:CARDINAL):CARDINAL;
VAR k,min:CARDINAL;
BEGIN
  min:=MAX(CARDINAL);
  FOR k:=i+1 TO j DO
    min:=Min2(min,T[i,k] + C[k,j])
  END;
RETURN min
END Min;

```

La función *Min2* es la que calcula el mínimo de dos números naturales. Es importante observar que esta función, por la forma en que se va rellenando la matriz *C*, sólo hace uso de los elementos calculados hasta el momento.

La complejidad del algoritmo es de orden $O(n^3)$, pues está compuesto por dos bucles anidados de tamaño n , que contienen la llamada a una función de orden $O(n)$, la que calcula el mínimo.

Transformación de cadenas

Sean u y v dos cadenas de caracteres. Se desea transformar u en v con el mínimo número de operaciones básicas del tipo siguiente: eliminar un carácter, añadir un carácter, y cambiar un carácter. Por ejemplo, podemos pasar de *abbac* a *abcbc* en tres pasos:

```

abbac → abac (eliminamos b en la posición 3)
      → ababc (añadimos b en la posición 4)
      → abcbc (cambiamos a en la posición 3 por c)

```

Sin embargo, esta transformación no es óptima. Lo que queremos en este caso es diseñar un algoritmo que calcule el número mínimo de operaciones, de esos tres tipos, necesarias para transformar u en v y cuáles son esas operaciones, estudiando su complejidad en función de las longitudes de u y v .

En primer lugar, la transformación mostrada arriba no es óptima ya que podemos pasar de *abbac* a *abcbc* en sólo dos pasos:

```

abbac → abcac (cambiamos b en la posición 3 por c)
      → abcbc (cambiamos a en la posición 4 por c)

```

Llamaremos m a la longitud de la cadena u , n a la longitud de la cadena v , y $OB(m,n)$ indicará el número de operaciones básicas mínimo para transformar una cadena u de longitud m en otra cadena v de longitud n .

Para resolver el problema utilizando Programación Dinámica es necesario plantearlo como una sucesión de decisiones que satisfaga el principio de óptimo. Para plantearla, vamos a fijarnos en el último elemento de cada una de las cadenas. Si los dos son iguales, entonces tendremos que calcular el número de operaciones básicas necesarias para obtener de la primera cadena menos el último elemento, y la segunda cadena también sin el último elemento, es decir,

$$OB(m,n) = OB(m-1,n-1) \text{ si } u_m = v_n.$$

Pero si los últimos elementos fueran distintos habría que escoger la situación más beneficiosa de entre tres posibles: (i) considerar la primera cadena y la segunda pero sin el último elemento, o bien (ii) la primera cadena menos el último elemento y la segunda cadena, o bien (iii) las dos cadenas sin el último elemento.

Esto da lugar a la siguiente relación en recurrencia para $OB(m,n)$ para este caso:

$$OB(m,n) = 1 + \text{Min}\{OB(m,n-1), OB(m-1,n), OB(m-1,n-1)\} \text{ si } m > 0, n > 0 \text{ y } u_m \neq v_n.$$

En cuanto a las condiciones iniciales, tenemos las tres siguientes:

$$OB(0,0) = 0, OB(m,0) = m \text{ y } OB(0,n) = n.$$

Una vez disponemos de la ecuación en recurrencia necesitamos resolverla utilizando alguna estructura que nos permita reutilizar resultados intermedios, como mostramos a continuación

```

CONST MAXCARACTERES = ...;
TYPE CADENA=ARRAY[1..MAXCARACTERES] OF CHAR;
TABLA=ARRAY[0..MAXCARACTERES],[0..MAXCARACTERES] OF CARDINAL;

PROCEDURE Cadena(VAR OB:TABLA;u,v:CADENA;n,m:CARDINAL):CARDINAL;
VAR i,j: CARDINAL;
BEGIN
    FOR i:=0 TO m DO OB[i,0]:=i; END;
    FOR j:=0 TO n DO OB[0,j]:=j; END;
    FOR i:=1 TO m DO
        FOR j:=1 TO n DO
            IF u[i]=v[j] THEN OB[i,j]:=OB[i-1,j-1]
            ELSE OB[i,j]:=Min3(OB[i,j-1],OB[i-1,j],OB[i-1,j-1])+1;
        END
    END
END;
RETURN OB[m,n]
END Cadena;

```

El procedimiento *Cadena* va a permitir la creación de la tabla *OB* que calcula el número mínimo de operaciones básicas. La solución se encuentra en $OB[m,n]$, y la tabla se construye fila a fila (a partir de los valores que definen las condiciones iniciales) para poder ir reutilizando los valores calculados previamente. La función *Min3* es la que calcula el mínimo de tres enteros.

Como el algoritmo se limita a dos bucles anidados que sólo incluyen operaciones constantes la complejidad de este algoritmo es de orden $O(mn)$.