

Tema 2: Programación dinámica

A. Salguero, F. Palomo, I. Medina

Universidad de Cádiz

Diseño de Algoritmos
Curso 2018/19

Introducción

La idea de la *programación dinámica* es evitar recalcular lo mismo, almacenando en memoria las subsoluciones encontradas.

Es aquí donde se aprecia mejor el «compromiso tiempo-espacio» en el diseño de algoritmos: se sacrifica espacio para ganar tiempo. La estructura de datos en la que se almacenan las subsoluciones se llama *tabla de subproblemas resueltos*.

La *programación dinámica* es una técnica *ascendente*, la mayor dificultad está en elegir el orden en el que se resuelven los subejemplares.

Hay que fijar el orden en que se debe rellenar la tabla: al calcular un subejemplar, aquéllos implicados ya deben estar almacenados.

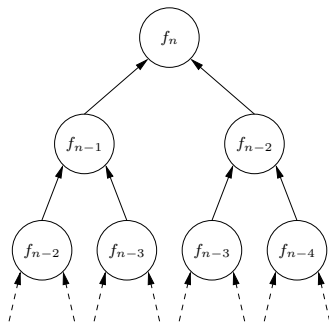
Ejemplo: sucesión de Fibonacci

$$f_n = \begin{cases} n, & n < 2 \\ f_{n-1} + f_{n-2}, & n \geq 2 \end{cases}$$

Si se calcula f_n a partir de su definición recursiva aparece un problema. El algoritmo realiza $\Theta(\phi^n)$ sumas de números de $O(n)$ dígitos.

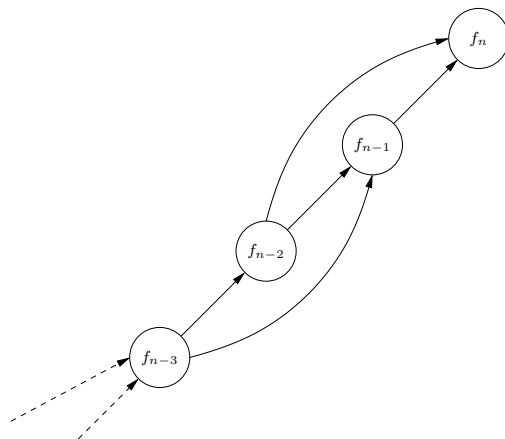
El número exacto de llamadas recursivas que hace el algoritmo es el doble del de sumas.

Ejemplo: sucesión de Fibonacci



La siguiente modificación evita la repetición de operaciones:
almacena los valores ya calculados para su empleo posterior.

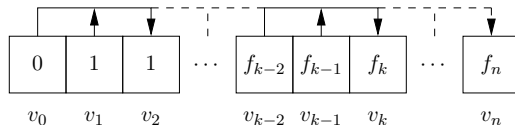
Ejemplo: sucesión de Fibonacci



Se dispone un vector de $n + 1$ elementos, uno por cada número a calcular. Inicialmente, los dos primeros son f_0 y f_1 . Para hallar f_n con $n > 1$, se rellenan los demás en orden creciente.

Ejemplo: sucesión de Fibonacci

Cuando se calcula f_k con $k > 1$, los valores de f_{k-1} y f_{k-2} ya han sido almacenados, con lo que se evita recalcularlos.



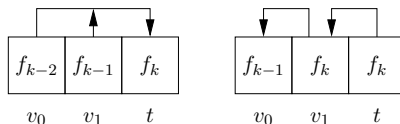
Con esta idea obtenemos un algoritmo que realiza $n - 1$ operaciones de suma para $n > 0$, es decir, $\Theta(n)$ sumas. Sólo se emplea un vector de $n + 1$ elementos, con lo que el algoritmo también es $\Theta(n)$ espacial.

Nota

En general, el elemento v_k almacena f_k , que tiene $\Theta(k)$ dígitos. La suma de f_{k-1} y f_{k-2} para obtener f_k emplea $\Theta(k)$ sumas elementales de dígitos. En realidad, la complejidad temporal y espacial del algoritmo es $\Theta(n^2)$.

Ejemplo: sucesión de Fibonacci

Pero aún podemos mejorar esto: con dos elementos y una variable auxiliar tenemos resuelto el problema.



$f : n \rightarrow r$

$v[0] \leftarrow 0$

$v[1] \leftarrow 1$

mientras $n > 1$

$t \leftarrow v[0] + v[1]$

$v[0] \leftarrow v[1]$

$v[1] \leftarrow t$

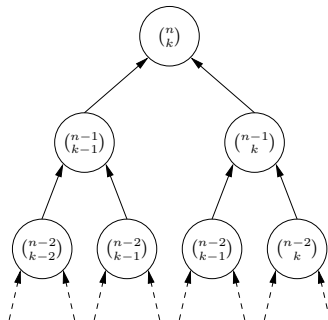
$n \leftarrow n - 1$

$r \leftarrow v[1]$

Ejemplo: números combinatorios

$$\binom{n}{k} = \begin{cases} 1, & k = 0 \vee k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k}, & 0 < k < n \end{cases}$$

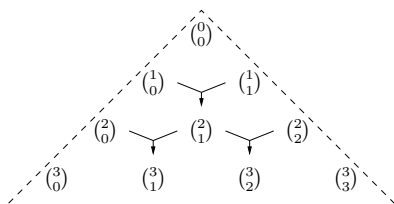
- Se repiten cálculos, como en Fibonacci.



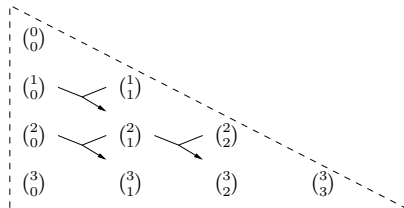
- El número de sumas es $\binom{n}{k} - 1 \in \Theta\left(\binom{n}{k}\right)$.
- El número de llamadas recursivas es el doble que el de sumas.

Ejemplo: números combinatorios

La idea de resolución por programación dinámica reside en el *triángulo de Pascal*:



(a) Aspecto tradicional



(b) Disposición tabular

Ejemplo: números combinatorios

Basta con una matriz en la que cada c_{ij} , con $0 \leq j \leq i \leq n$ almacene $\binom{i}{j}$.

$$c_{ij} = \begin{cases} \binom{i}{j}, & j \leq i \\ 0, & j > i \end{cases}$$

- Inicialmente, la primera fila tiene un 1 en su primer elemento y 0 en los demás. Calcular $\binom{n}{k}$ para un $n > 1$, implica rellenar las filas restantes con los valores apropiados.
- Este algoritmo realiza $\Theta(nk)$ sumas, empleando una matriz de $\Theta(nk)$ elementos.
- Cada fila se calcula empleando sólo la anterior, así, bastan dos vectores de longitud $k + 1$, es decir, $\Theta(k)$ elementos.

Ejemplo: números combinatorios

El espacio se puede reducir a un único vector:

$$C : n \times k \rightarrow r$$

$$c[0] \leftarrow 1$$

desde $j \leftarrow 1$ hasta k

$$c[j] \leftarrow 0$$

desde $i \leftarrow 1$ hasta n

desde $j \leftarrow k$ hasta 1

$$c[j] \leftarrow c[j] + c[j - 1]$$

$$r \leftarrow c[k]$$

Se puede disminuir el número de operaciones:

- Antes de cada paso del bucle j , c_j valdrá $\binom{i-1}{j}$ y, después, contendrá $\binom{i}{j}$. Por lo tanto, c_j no se modificará cuando $j > i$. El bucle interno puede comenzar en $\min\{i, k\}$.
- Una vez calculado c_k en el paso n del bucle i se puede terminar. Podemos parar en el paso $n - 1$ y calcular c_k fuera del bucle.

Ejemplo: números combinatorios

```

 $C : n \times k \rightarrow r$ 
si  $k = 0 \vee k = n$ 
   $r \leftarrow 1$ 
si no
  si  $k > n$ 
     $r \leftarrow 0$ 
  si no
     $c[0] \leftarrow 1$ 
    desde  $j \leftarrow 1$  hasta  $k$ 
       $c[j] \leftarrow 0$ 
    desde  $i \leftarrow 1$  hasta  $n - 1$ 
      desde  $j \leftarrow \min(i, k)$  hasta 1
         $c[j] \leftarrow c[j] + c[j - 1]$ 
     $r \leftarrow c[k] + c[k - 1]$ 

```

Existe simetría, ya que $\binom{n}{k} = \binom{n}{n-k}$ cuando $0 \leq k \leq n$. Así, si $\lfloor \frac{n}{2} \rfloor < k \leq n$ conviene calcular $\binom{n}{n-k}$, por ser $0 \leq n - k < \lfloor \frac{n}{2} \rfloor$.

Problemas de optimización

Muchos problemas pueden modelarse de forma que encontrar una solución consista en ejecutar una secuencia de decisiones.

Puede haber varias soluciones: la secuencia determinará una de ellas. Cada solución puede ser mejor o peor atendiendo a distintos criterios.

Cuando la solución obtenida debe ser óptima estamos ante un *problema de optimización*. Una *secuencia de decisiones óptima* es la que conduce a una *solución óptima*.

A veces se pueden resolver siguiendo una estrategia devoradora adecuada.

Esto en general no es posible. No siempre se puede resolver un problema empleando información local en la toma de decisiones.

El principio de optimalidad

Cualquier problema de optimización se puede resolver por fuerza bruta.

Si hay que tomar una secuencia de no menos de n decisiones y en cada una hay un mínimo de p posibilidades hay que evaluar $\Omega(p^n)$ secuencias (*explosión combinatoria*).

El principio de optimalidad permite reducir el número de secuencias posibles, eliminando las que descubre que no son óptimas:

En una secuencia óptima de decisiones, toda subsecuencia ha de ser también óptima.

Esto no es una ley que se cumpla en todos los problemas, pero en los que se cumple indica cuándo una decisión no es apropiada.

El problema de la mochila

Dados n objetos, cada uno con un valor v_i y un peso p_i , y una mochila con una capacidad, c , que limita el peso total que puede transportar, se desea hallar la composición de la mochila que maximiza el valor de la carga.

La solución se modela asociando a cada objeto un valor x_i que representa la fracción del mismo que queda en la mochila. En su versión discreta, los objetos son indivisibles, $x_i \in \{0, 1\}$.

La resolución del problema discreto es un proceso de toma de decisiones. Hay que decidir para cada objeto si se incluye o no. El principio de optimalidad se cumple para este problema.

El problema de la mochila

Aplicando el principio de optimalidad resulta $f(n, c)$, que representa el valor máximo que se puede obtener:

Capacidad de Mochila

n objetos

$$f(n, c) = \begin{cases} 0, & \rightarrow \text{No se puede llevar con objeto, no cabe en } n=1 \wedge c < p_1 \\ v_1, & \rightarrow \text{Si cabe, pero solo cabe el mínimo, } h = \text{objeto } n=1 \wedge c \geq p_1 \\ f(n-1, c), & \Rightarrow \text{Si } n \text{ no cabe, } h = \text{objetos anteriores } n > 1 \wedge c < p_n \\ \max\{f(n-1, c), f(n-1, c - p_n) + v_n\}, & n > 1 \wedge c \geq p_n \end{cases}$$

Si cabe \Rightarrow Se mide o no en la mochila, según el beneficio más

- Si sólo hay un objeto i puede ocurrir que:

- $c < p_i$: no se incluye, el valor máximo de la carga será 0.
- $c \geq p_i$: se incluye, el valor máximo de la carga será v_i .

① max h de una mochila c , considerando los anteriores, sin incluir n

② $v_n + f(n-1, c - p_n)$
 (valor objeto n) + beneficio de incluir o no el objeto introducido.

El problema de la mochila

- Si hay varios objetos y estamos con el objeto i puede ocurrir que:
 - 1 $c < p_i$: no se incluye, el valor máximo de la carga será el que se obtenga de considerar los restantes objetos.
 - 2 $c \geq p_i$: el valor máximo de la carga será el máximo de
 - 1 El que se obtiene al descartar el objeto y considerar los restantes para rellenar la mochila.
 - 2 El que se obtiene al incluir el objeto y considerar los restantes para rellenar la mochila.

El problema de la mochila

Para calcular $f(n, c)$ se emplea como tabla de subproblemas resueltos una matriz en la que el elemento f_{ij} almacenará el valor $f(i, j)$.

$mochila : v \times p \times n \times c \rightarrow r$

desde $j \leftarrow 0$ hasta c

si $j < p[1]$

$f[1, j] \leftarrow 0$

si no

$f[1, j] \leftarrow v[1]$

desde $i \leftarrow 2$ hasta n

desde $j \leftarrow 0$ hasta c

si $j < p[i]$

$f[i, j] \leftarrow f[i-1, j]$

si no

$f[i, j] \leftarrow \max(f[i-1, j],$
 $f[i-1, j - p[i]] + v[i])$

$r \leftarrow f[n, c]$

- Requiere información, se compara con el último objeto de análisis, si es menor \rightarrow lo he introducido, se resta ese objeto \leftarrow
- $f_{ij} \Rightarrow$ objeto
- $c \Rightarrow$ capacidad
- Complejidad espacial $\Rightarrow n \times c$ bits/bits

El algoritmo es $\Theta(nc)$ temporal y espacial.

El problema del cambio de moneda

Disponemos de n tipos de monedas de valor v_i y deseamos devolver un cambio de c unidades monetarias empleando el mínimo número posible de monedas de cada tipo.

Como simplificación supondremos que disponemos de una cantidad ilimitada de monedas de cada tipo. Esta versión del problema se conoce como *problema del cambio de moneda con suministro ilimitado*.

La resolución de este problema puede contemplarse como un proceso de toma de decisiones. Hay que decidir para cada tipo de moneda cuántas incluimos en el cambio.

El principio de optimalidad se cumple para este problema.

El problema del cambio de moneda

Aplicando el principio de optimalidad resulta $f(n, c)$, que representa el número mínimo de monedas con las que se puede devolver el cambio; cundo $c > 0$, es: \rightarrow la ecuación es similar a la mochila.

$$f(n, c) = \begin{cases} \infty, & \leftarrow \text{Valor más grande de la moneda que ha devuelto} \\ 1 + f(1, c - v_1), & \leftarrow \text{NÓ 2, ya que } 0=1 \\ f(n-1, c), & \\ \min\{f(n-1, c), 1 + f(n, c - v_n)\}, & \end{cases}$$

$n = 1 \wedge c < v_1$
 $n = 1 \wedge c \geq v_1$
 $n > 1 \wedge c < v_n$
 $n > 1 \wedge c \geq v_n$

$\rightarrow 1 + f(1, c - v_1) \Rightarrow$ cuando no puedes cambiar el tipo.
 \hookrightarrow Tipo de moneda $\rightarrow 1ct, 2ct, \dots$

El problema del cambio de moneda

- Si sólo se dispone de un tipo de moneda i puede ocurrir que:
 - 1 $c < v_i$, será imposible dar el cambio y devolvemos ∞ .
 - 2 $c \geq v_i$, habrá que emplear una moneda i más las mínimas que hagan falta para devolver la cantidad restante.
- Si disponemos de varios tipos de moneda y estamos considerando las monedas de tipo i puede ocurrir que:
 - 1 $c < v_i$, será imposible dar el cambio exacto con ese tipo de moneda y se emplearán las mínimas que hagan falta para devolver dicha cantidad con los restantes tipos de moneda.
 - 2 $c \geq v_i$, el número mínimo de monedas necesarias para devolver el cambio será el mínimo de los siguientes valores:
 - 1 El que se obtiene cuando se descartan las monedas i y se consideran sólo los restantes tipos de monedas para devolver el cambio.
 - 2 El que se obtiene cuando se incluye una moneda i más las mínimas que hagan falta para devolver el resto.

El problema del cambio de moneda

Para calcular $f(n, c)$ se emplea como tabla de subproblemas resueltos una matriz en la que el elemento f_{ij} almacenará el valor $f(i, j)$.

cambio : $v \times n \times c \rightarrow r$

$f[1, 0] \leftarrow 0$

desde $j \leftarrow 1$ hasta c

si $j < v[1]$

$f[1, j] \leftarrow \infty$

si no

$f[1, j] \leftarrow 1 + f[1, j - v[1]]$

desde $i \leftarrow 2$ hasta n

$f[i, 0] \leftarrow 0$

desde $j \leftarrow 1$ hasta c

si $j < v[i]$

$f[i, j] \leftarrow f[i - 1, j]$

si no

$f[i, j] \leftarrow \min(f[i - 1, j], 1 + f[i, j - v[i]])$

$r \leftarrow f[n, c]$

Caminos mínimos

Sea $G = \langle V, A \rangle$ un grafo ponderado con una función $p : V \times V \rightarrow \mathbb{R}^+ \cup \{0, \infty\}$. Se desea obtener, para cada pareja de vértices, el camino mínimo entre ambos.

Por ejemplo, los valores de p se pueden interpretar como «distancias» entre los vértices:

- Un vértice está siempre de sí mismo a una distancia nula.
- Si dos vértices no están conectados por una arista su distancia es infinita.

$$p(v_i, v_j) = \begin{cases} 0, & i = j \\ c_{ij}, & i \neq j \wedge \langle v_i, v_j \rangle \in A \\ \infty, & i \neq j \wedge \langle v_i, v_j \rangle \notin A \end{cases}$$

Caminos mínimos

El principio de optimalidad se cumple para este problema. Para resolverlo se parte de la matriz de pesos de G :

$$P = \begin{bmatrix} p_{11} & \cdots & p_{1n} \\ \vdots & \ddots & \vdots \\ p_{n1} & \cdots & p_{nn} \end{bmatrix}$$

donde $p_{ij} = p(v_i, v_j)$.

El algoritmo de Floyd recibe inicialmente la matriz de pesos y sobre ella va calculando los valores de los caminos mínimos.

Tras cada iteración k , $P^{[k]}$ contiene los valores de los caminos mínimos que únicamente emplean los vértices $\{v_1, \dots, v_k\}$:

$$\begin{aligned} p_{ij}^{[0]} &= p_{ij} \\ p_{ij}^{[k]} &= \min\{p_{ij}^{[k-1]}, p_{ik}^{[k-1]} + p_{kj}^{[k-1]}\}, \quad k > 0 \end{aligned}$$

Caminos mínimos

Nótese que:

- $P^{[k]}$ sólo depende de $P^{[k-1]}$.
- En la iteración k no se modifican ni la fila ni la columna k :

$$p_{kj}^{[k]} = \min\{p_{kj}^{[k-1]}, p_{kk}^{[k-1]} + p_{kj}^{[k-1]}\} = p_{kj}^{[k-1]}$$

$$p_{ik}^{[k]} = \min\{p_{ik}^{[k-1]}, p_{ik}^{[k-1]} + p_{kk}^{[k-1]}\} = p_{ik}^{[k-1]}$$

Basta emplear únicamente la matriz P :

Floyd : $P \rightarrow P$

desde $i \leftarrow 1$ hasta n

$p[i, i] \leftarrow 0$

desde $k \leftarrow 1$ hasta n

desde $i \leftarrow 1$ hasta n

desde $j \leftarrow 1$ hasta n

$p[i, j] \leftarrow \min(p[i, j], p[i, k] + p[k, j])$

La complejidad temporal de este algoritmo es de $\Theta(n^3)$ operaciones elementales.

Clausura reflexiva y transitiva

En lo que sigue, consideraremos dos conjuntos finitos,
 $V = \{v_1, \dots, v_n\}$ y $A \subseteq V \times V$.

Definición

Dado un grafo $G = \langle V, A \rangle$, se define su *clausura reflexiva y transitiva* como el grafo $G^* = \langle V, A^* \rangle$ que se obtiene a partir de G añadiendo aristas entre los vértices que están unidos por algún camino.

Definición

Dada una relación binaria A sobre V , se define su *clausura reflexiva y transitiva* como la menor relación A^* que extiende a A y es reflexiva y transitiva.

Nota

La clausura de una relación bajo una propiedad es la menor relación que la extiende y que cumple dicha propiedad.

Clausura reflexiva y transitiva

Definición

Dada una relación binaria A sobre V , se define su *matriz* como:

$$M_A = \begin{bmatrix} m_{11} & \cdots & m_{1n} \\ \vdots & \ddots & \vdots \\ m_{n1} & \cdots & m_{nn} \end{bmatrix}, \text{ donde : } m_{ij} \equiv v_i A v_j$$

Nota

M_A no es más que la *matriz de adyacencia* del grafo de la relación A .

- Relaciones binarias, matrices booleanas y grafos son conceptos muy relacionados.
- La clausura reflexiva y transitiva de la relación de adyacencia de un grafo produce la relación de accesibilidad.

Clausura reflexiva y transitiva

El algoritmo de Warshall recibe inicialmente la matriz de adyacencia del grafo y sobre ella va construyendo la matriz de accesibilidad. Sea A la matriz de adyacencia del grafo. Notando por $A^{[k]}$ el valor de A tras la iteración k se obtiene:

$$a_{ij}^{[0]} = \begin{cases} \top, & i = j \\ a_{ij}, & i \neq j \end{cases}$$
$$a_{ij}^{[k]} = a_{ij}^{[k-1]} \vee (a_{ik}^{[k-1]} \wedge a_{kj}^{[k-1]}), \quad k > 0$$

Este algoritmo es similar al de Floyd: no es casualidad, ambos problemas son casos particulares del cálculo de la clausura reflexiva y transitiva en un *semianillo cerrado*.

En realidad, ambos algoritmos son una especialización del algoritmo de Kleene.

Clausura reflexiva y transitiva

Warshall : $A \rightarrow A$

desde $i \leftarrow 1$ hasta n

$a[i, i] \leftarrow \top$

desde $k \leftarrow 1$ hasta n

desde $i \leftarrow 1$ hasta n

desde $j \leftarrow 1$ hasta n

$a[i, j] \leftarrow a[i, j] \vee (a[i, k] \wedge a[k, j])$

Este algoritmo ejecuta $\Theta(n^3)$ operaciones elementales.





Nota

En general, calcular G^* es equivalente a:

- 1 Construir la matriz A a partir de G .
- 2 Calcular la matriz A^* a partir de A .
- 3 Construir el grafo G^* a partir de A^* .

Calcular la clausura reflexiva y transitiva de un grafo se reduce así a calcular la de una matriz.

Referencias

-  Aho, Alfred; Hopcroft, John & Ullman, Jeffrey.
The Design and Analysis of Computer Algorithms.
Addison-Wesley. 1974.
-  Brassard, Gilles & Bratley, Paul.
Fundamentos de Algoritmia.
Prentice-Hall. 1997.
-  Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L. & Stein, Clifford.
Introduction to Algorithms.
MIT Press. 2001. 2ª ed.
-  Horowitz, Ellis & Sahni, Sartaj.
Fundamentals of Computer Algorithms.
Pitman. 1978.