Diseño de algoritmos

Programación Dinámica

Jesús Bermúdez de Andrés

Universidad del País Vasco/Euskal Herriko Unibertsitatea **(UPV/EHU)**

Curso 2008-09

1. Programación Dinámica
   - Definición de la técnica
   - Funciones con memoria
   - Optimización con Programación dinámica
   - Problema de la mochila con valores enteros
   - Problema de la subsecuencia común más larga
   - Problema de todos los caminos mínimos

Programación dinámica

La técnica de **Programación dinámica** fue inventada como un método general de optimización de procesos de decisión por etapas.

La técnica de programación dinámica es adecuada para resolver problemas cuya solución puede caracterizarse recursivamente (como con la técnica divide y vencerás) y en la que los subproblemas que aparecen en la recursión se solapan de algún modo, lo que significaría una repetición de cálculos inaceptable si se programara la solución recursiva de manera directa.

La aplicación de la técnica de programación dinámica evita la repetición de cálculos mediante la memorización de la solución de cada subproblema en una tabla, de manera que no haya que calcularlo más de una vez.

Fases para la aplicación de la técnica

La aplicación de la técnica de programación dinámica tiene dos fases fundamentales:

1. Definir recursivamente la solución del problema.
2. Definir la estructura de datos para memorizar las soluciones de los subproblemas y escribir el algoritmo que va calculando los valores de esa estructura de datos siguiendo la caracterización de la solución definida en la fase 1, pero sin repetir el cálculo de soluciones de subproblemas.

Combinatoria

¿Cuántos subconjuntos de n elementos pueden formarse con un conjunto de m elementos? ($n \leq m$)
Denominemos $S(m, n)$ a ese número:

- Si $n = 0$ entonces sólo puede formarse el conjunto vacío: $S(m, 0) = 1$
- Si $n > 0$, el número de subconjuntos queda caracterizado por el siguiente razonamiento: Tomemos un elemento x del conjunto de m elementos. Por un lado están todos los subconjuntos que contienen a x y, por otro lado, todos los que no.
  - En el primer caso, el número de subconjuntos es $S(m-1, n-1)$ ya que x es uno de los elementos elegidos y los otros $n-1$ han de ser elegidos de entre $m-1$ elementos.
  - En el segundo caso, hay que elegir n elementos de la colección de $m-1$ restantes (al quitar x), así que su número debe ser $S(m-1, n)$.

  Por lo tanto, $S(m, n) = S(m-1, n-1) + S(m-1, n)$ cuando $n > 0$

Es evidente que la programación recursiva directa de esta recurrencia produce multitud de cálculos repetidos.

Combinaciones(m, n)

Definimos una estructura de datos adecuada para memorizar las soluciones de los subproblemas: Una tabla $T(0..m, 0..n)$ tal que $T(i, j)$ guardará $S(i, j)$.
Un algoritmo que calcula esos valores, según dicta la recurrencia definida para $S(m, n)$, desde los casos básicos a los generales, es el siguiente:

```
func Combinaciones(m, n) return natural
    for i in 0 ... m loop
        T(i, 0) ← 1
        for j in 1 ... mín(i − 1, n) loop
            T(i, j) ← T(i − 1, j − 1) + T(i − 1, j)
        end loop
        if i ≤ n then T(i, i) ← 1
    end loop
    return T(m, n)
```

Análisis: Obviamente, $\Theta(mn)$ en tiempo y en espacio extra.

Funciones con memoria (1/2)

Podemos escribir un algoritmo recursivo, que siga el dictado de la recurrencia para $S(m, n)$, evitando la repetición de cálculos.

Basta con poner una condición antes de realizar cada llamada recursiva (la condición de que ese valor no haya sido calculado previamente). Definimos

la misma tabla $T(0..m, 0..n)$ tal que $T(i, j)$ guardará $S(i, j)$. Inicializamos la tabla con valores que representen que las casillas no han sido todavía calculadas.

Por ejemplo:
$T(i, j) \leftarrow 0 \quad \forall i : 0 \ldots m,\ j : 0 \ldots n$

Funciones con memoria (2/2)

func CombinacionesFM(m, n) **return** natural
  if $T(m,n) \neq 0$ **then return** $T(m,n)$
  if $n = 0 \vee n = m$ **then**
    $T(m,n) \leftarrow 1$
    return $T(m,n)$
  $\{n \neq 0 \wedge n \neq m \wedge T(m,n)$ no calculado $\}$
  $T(m,n) \leftarrow$ CombinacionesFM$(m-1,\ n-1) +$
    CombinacionesFM$(m-1,\ n)$
  return $T(m,n)$

Análisis: Básicamente, se rellena la misma tabla de valores que con Combinaciones(m, n) (algunos valores menos) y cada casilla se rellena con un coste de $O(1)$; así que $\Theta(mn)$ en tiempo y en espacio extra.

Programación dinámica y optimización

Un requisito para el uso de la programación dinámica en la resolución de problemas de optimización es que el problema en cuestión satisfaga el

Principio de optimalidad: En toda secuencia óptima de decisiones para resolver el problema, cualquier subsecuencia es óptima para resolver el subproblema correspondiente.

Programación dinámica y optimización: Fases (1/2)

El diseño de un algoritmo de optimización según la técnica de programación dinámica se realiza en cuatro fases:

1. Caracterizar la estructura de una solución optimal (asegurar que se satisface el *principio de optimalidad*).
2. Definir recursivamente la función de optimización.
3. Escribir el algoritmo que calcula la función de optimización sin repetir cálculos de soluciones de subproblemas.
4. Escribir el algoritmo que construye una solución optimal del problema, a partir de la información calculada en el paso anterior.

La fase 4 no se realiza cuando sólo se busca el valor de la función de optimización. Si hay que realizar la fase 4, en la fase 3 debe considerarse el cálculo de información adicional.

Programación dinámica y optimización: Fases (2/2)

En general, las soluciones a problemas de optimización obtenidas con la técnica de programación dinámica pueden imaginarse como compuestas de una serie de opciones tomadas en estados sucesivos.

Para abordar las fases 1 y 2, mencionadas antes, puede ayudar la táctica de suponer que se ha tomado una de esas opciones para llegar a la solución y analizar qué subproblemas quedarían todavía pendientes para ser resueltos y cómo se combinarían para solucionar el original de manera óptima.

Obsérvese que, seguramente, esa opción inicial no es única y que interesará estudiar las demás posibilidades, quedándonos con la mejor.

Problema de la mochila con valores enteros

Disponemos de n objetos de valores v_i $(i : 1 \ldots n)$ y pesos p_i $(i : 1 \ldots n)$, respectivamente. Queremos cargar una mochila con una colección de esos objetos que maximice la suma de los valores transportados. La capacidad máxima de la mochila es $M \geq 0$.

Fase 1: Estructura de una solución optimal

Este problema satisface el *principio de optimalidad*:

- Una carga optimal de la mochila contiene una colección de objetos S. Si retiramos un objeto de esa colección optimal S (supongamos que está numerado con el índice k), la colección restante $S - \{k\}$ es una forma óptima de cargar una mochila de capacidad $M - p_k$.

Podemos construir una solución optimal a partir de soluciones optimales de los n subproblemas siguientes (uno por cada objeto $i : 1 \ldots n$): Carga óptima de una mochila de capacidad $M - p_i$ considerando la colección de objetos inicial sin el objeto número i.

Una solución optimal del problema original es la que más valor acumule tras sumar v_i $(i : 1 \ldots n)$ al valor de la correspondiente solución del subproblema i.

Fase 2: Definición recursiva de la función de optimización

Primer intento

$B(C) =$ Máximo beneficio que puede llevarse en la mochila de capacidad C (contando con los objetos dados)

$$B(C) \quad = \quad \textbf{max}\{v_i + B(C - p_i) \mid \quad (1 \leq i \leq n) \wedge p_i \leq C\}$$

Inconveniente: En $B(C - p_i)$ no se refleja que el objeto i no puede tomar parte en la solución.

Fase 2: Definición recursiva de la función de optimización

Segundo intento

- Establecemos un orden entre los elementos (según su numeración $\{1, \ldots, n\}$).
- Añadimos un parámetro a la función, que representa una etapa en la resolución del problema.

$B(C, k) =$ Máximo beneficio que puede llevarse en la mochila de capacidad C cuando puedo poner objetos del conjunto $\{1, \ldots, k\}$.
$k = 0$ significa que el conjunto de elementos es vacío.
Nos interesa el valor $B(C, n)$

$$
\begin{aligned}
B(0,k) &= 0 \quad \forall k \\
B(C,0) &= 0 \quad \forall C \\
B(C,k) &= \mathbf{max}\{B(C,k-1),\ v_k + B(C-p_k, k-1)\} \\
&\quad\ \ si\ k > 0 \wedge p_k \leq C \\
B(C,k) &= B(C,k-1) \quad si\ k > 0 \wedge p_k > C
\end{aligned}
$$

Fase 3: Algoritmo que calcula la función de optimización

Definimos una tabla $T(0..M, 0..n)$ tal que cada casilla $T(i,j)$ guardará el valor de $B(i,j)$.

```
proc Mochila (T(0..M, 0..n), p(1..n), v(1..n))
    for k in 0..n loop T(0, k) ← 0 end loop
    for C in 0..M loop T(C, 0) ← 0 end loop
    for k in 1..n loop
        for C in 1..M loop
            if p(k) ≤ C then
                if T(C, k − 1) < v(k) + T(C − p(k), k − 1) then
                    T(C, k) ← v(k)+T(C-p(k), k-1)
                else
                    T(C, k) ← T(C, k-1)
                end if
            else
                T(C, k) ← T(C, k-1)
```

Análisis: $\Theta(Mn)$ en tiempo y en espacio extra

Fase 4: Construcción de una solución optimal (1/2)

Añadimos unas "marcas" en el algoritmo, que nos indicarán la opción que se tomó en tal estado.

Típicamente, la estructura para las marcas es una estructura paralela a la que guarda los resultados.

$$\text{objetos}(C, k) = \text{true} \quad \Leftrightarrow \quad T(C, k-1) < v_k + T(C - p_k, k-1)$$

Significa que en el estado (C, k) interesa tomar el objeto k

Fase 4: Construcción de una solución optimal (2/2)

```
proc MOCHILA (T(0..M, 0..n), p(1..n), v(1..n))
    for k in 0..n loop T(0, k) ← 0 end loop
    for C in 0..M loop T(C, 0) ← 0 end loop
    for k in 1..n loop
        for C in 1..M loop
            if p(k) ≤ C then
                if T(C, k − 1) < v(k) + T(C − p(k), k − 1) then
                    T(C, k) ← v(k)+T(C-p(k), k-1)
                    objetos(C, k) = true
                else
                    T(C, k) ← T(C, k-1)
                    objetos(C, k) = false
                end if
            else
                T(C, k) ← T(C, k-1)
                objetos(C, k) = false
```

Construcción de solución óptima

La solución está caracterizada por la siguiente función:

$$
\begin{aligned}
\mathit{Cons\_Mochila}(C, 0) &= \{\} \\
\mathit{Cons\_Mochila}(0, k) &= \{\} \\
\mathit{Cons\_Mochila}(C, k) &= \mathit{Cons\_Mochila}(C, k-1) \\
&\quad \mathit{si}\ \mathit{objetos}(C, k) = \mathit{false} \\
\mathit{Cons\_Mochila}(C, k) &= \{k\} \cup \mathit{Cons\_Mochila}(C - p_k, k-1) \\
&\quad \mathit{si}\ \mathit{objetos}(C, k) = \mathit{true}
\end{aligned}
$$

func CONS_MOCHILA(C, k) **return** conjunto
if $k = 0 \vee C = 0$ **then**
    return $\{\ \}$
else
    if objetos(C, k) = false **then**
        return CONS_MOCHILA(C, $k-1$)
    else
        return $\{k\} \cup$ CONS_MOCHILA($C - p_k$, $k-1$)

Nos interesa el valor de CONS_MOCHILA(M, n). Análisis: Θ(n)

Problema de la subsecuencia común más larga

Una *subsecuencia* de una secuencia S se obtiene retirando de S cualquier número de elementos de cualesquiera posiciones.
Por ejemplo: `sedante` es una subsecuencia de `estupendamente`.

El problema que queremos resolver es el de calcular una subsecuencia común, lo más larga posible, de dos secuencias dadas R y S.

Por ejemplo:

- Si $R =$ `aturdido` y $S =$ `tranquilo`, la solución es `trio`
- Si $R =$ `si` y $S =$ `no`, la solución es ε (secuencia vacía)

Fase 1: Estructura de una solución optimal

La siguiente función recursiva $f(R, S)$ define (por casos) una *subsecuencia común de longitud maximal* de dos secuencias R y S:

$$
\begin{array}{rcl}
f(R, \varepsilon) & = & \varepsilon \\
f(\varepsilon, S) & = & \varepsilon \\
f(a \cdot R, a \cdot S) & = & a \cdot f(R, S) \\
f(a \cdot R, b \cdot S) & = & \textit{la de mayor longitud entre } \; f(R, b \cdot S) \; \textit{ y } \; f(a \cdot R, S) \\
 & & \textit{si } \; a \neq b
\end{array}
$$

Fase 2: Definición recursiva de la función de optimización

Dadas las secuencias $R(1..m)$ y $S(1..n)$

$L(i,j) =$ *longitud de la subsecuencia más larga de* $R(i..m)$ *y* $S(j..n)$

$$
\begin{aligned}
L(m+1,j) &= 0 \ \forall j \\
L(i,n+1) &= 0 \ \forall i \\
L(i,j) &= 1 + L(i+1,j+1) \\
&\quad si \ \ R(i) = S(j) \wedge i \leq m \wedge j \leq n \\
L(i,j) &= max\{L(i+1,j), L(i,j+1)\} \\
&\quad si \ \ R(i) \neq S(j) \wedge i \leq m \wedge j \leq n
\end{aligned}
$$

Fase 3: Algoritmo que calcula la función de optimización

Utilizaremos una tabla de dimensiones $(m+1) \times (n+1)$ en la que cada posición (i, j) almacenará la solución del subproblema $L(i, j)$

```
proc SubsecMasLarga (R(1..m), S(1..n), L(1..m+1, 1..n+1))
    for i ←1...m+1 loop
        L(i, n+1) ←0
    end loop
    for j ←1...n+1 loop
        L(m+1, j) ←0
    end loop
    for i ←m downto 1 loop
        for j ←n downto 1 loop
            if R(i)=S(j) then
                L(i, j) ←1+L(i+1, j+1)
            else
                L(i, j) ←max { L(i+1, j), L(i, j+1) }
        end loop
    end loop
```

Análisis: $\Theta(mn)$ en tiempo y en uso de espacio extra.

Fase 4: Construcción de una solución optimal

Por último, para calcular la subsecuencia común, que corresponde a la longitud calculada por SubsecMasLarga(R(1..m), S(1..n), L(1..m+1, 1..n+1)), completaremos ese algoritmo para dejar las *marcas* siguientes:

$$Marca(i,j) = \begin{cases} X & \text{si } R(i) = S(j) \\ 1 & \text{si } R(i) \neq S(j) \wedge L(i+1,j) > L(i,j+1) \\ 2 & \text{si } R(i) \neq S(j) \wedge L(i+1,j) < L(i,j+1) \end{cases}$$

significando que la subsecuencia de longitud L(i, j) es:

$$Subsec(i,j) = \begin{cases} R(i) \cdot Subsec(i+1,j+1) & \text{si } Marca(i,j) = X \\ Subsec(i+1,j) & \text{si } Marca(i,j) = 1 \\ Subsec(i,j+1) & \text{si } Marca(i,j) = 2 \end{cases}$$

Ahora sólo queda implementar el algoritmo que calcula *Subsec(i, j)* e invocarlo con los parámetros $i = 1$ y $j = 1$. Es fácil conseguir una implementación de coste $O(m+n) = O(max(m,n))$ en tiempo.

Problema de todos los caminos mínimos

Considérese un grafo dirigido con pesos no negativos asociados a las aristas. ¿Cual es la distancia mínima entre cada par de nodos?

Fase 1: Estructura de una solución optimal

Para cada par de nodos a, b, un camino mínimo está compuesto de una sucesión de aristas.

La longitud del camino puede ser 1 (la arista que va del nodo origen a al nodo destino b) o mayor que 1.

En este segundo caso, habrá que pasar por algún nodo intermedio k. Evidentemente, el trozo de camino que va del origen a hasta k debe ser un camino mínimo, y lo mismo para el trozo de camino que va de k al destino b.

Fase 2: Definición recursiva de la función de optimización

Asumimos que los nodos están numerados $1 \ldots n$. Disponemos de la matriz de adyacencia del grafo $P(1..n, 1..n)$:

- $P(i,j) \geq 0$ *si* $i \neq j$
- $P(i,i) = 0 \; \forall i$
- $P(i,i) = \infty$ si no existe la arista (i,j)

Primer intento:
$D(i,j) =$ Distancia mínima del nodo i al nodo j
$D(i,j) = \mathbf{min}\{P(i,j),\ D(i,k) + D(k,j) \mid 1 \leq k \leq n \wedge k \neq i \wedge k \neq j\}$

$D(i,j)$ está mal definida: en su definición aparece $D(i,k)$, que a su vez necesitaría $D(i,j)$

Segundo intento

Añadimos un parámetro más a la función, que representa una noción de etapas en las que subdividimos el problema.

$D(i, j, k) =$ Distancia mínima de i a j cuando podemos pasar por los nodos $\{1 \ldots k\}$.

Si $k = 0$ significa que no podemos pasar por ningún nodo intermedio.

Cuando podemos pasar por los nodos $\{1..k\}$, hay dos opciones:

- No pasar por k, y por lo tanto utilizar sólo los restantes $\{1..k-1\}$ nodos.
- Pasar por k, y por lo tanto realizar el camino de i a k de forma mínima usando sólo los restantes $\{1..k-1\}$ nodos, y análogamente de k a j.

$$
\begin{aligned}
D(i, j, 0) &= P(i, j) \\
D(i, j, k) &= \textbf{min}\{D(i, j, k-1), D(i, k, k-1) + D(k, j, k-1)\}
\end{aligned}
$$

Gestión de la estructura de memorización

Necesitamos los valores $D(i, j, n)\ \forall i, j$

Podríamos calcular los valores de n matrices $D(i, j, p)$ para $p = 1..n$, y quedarnos con los resultados de la n-ésima.

Hay que observar que, en la etapa k, se satisface que

$$\begin{aligned} D(i, k, k) &= D(i, k, k-1) \\ D(k, j, k) &= D(k, j, k-1) \end{aligned}$$

La fila y columna k no se modifica en la etapa k.

Por lo tanto nos basta una sola matriz $D(i, j)$ que se va actualizando en cada etapa según la recurrencia definida.

Fase 3: Algoritmo que calcula la función de optimización

```
func FLOYD (P(1..n, 1..n)) return matriz
    D(1..n, 1..n) ← P(1..n, 1..n)
    for k in 1..n loop
        for i in 1..n loop
            for j in 1..n loop
                D(i, j) ← min(D(i, j), D(i, k)+D(k, j))
    return D(1..n, 1..n)
```

Análisis: $\Theta(n^3)$ en tiempo y $\Theta(n^2)$ en espacio extra

Fase 4: Construcción de una solución optimal

Para poder construir los caminos mínimos entre cada par de nodos, añadimos una matriz de marcas:
$M(i, j) = k$, significará que para ir de i a j por un camino mínimo es preciso pasar por el nodo k.
Como siempre, las marcas se añaden en la parte del código donde se toman decisiones.

```
func FLOYD (P(1..n, 1..n)) return matrices
    D(1..n, 1..n) ← P(1..n, 1..n)
    for k in 1..n loop
        for i in 1..n loop
            for j in 1..n loop
                if D(i, k)+D(k, j) < D(i, j) then
                    D(i, j) ← D(i, k)+D(k, j)
                    M(i, j) ← k
    return D(1..n, 1..n) y M(1..n, 1..n)
```

Construcción de una solución óptima

Algoritmo que construye la lista de nodos intermedios para ir del nodo i al nodo j por camino mínimo.
$M(1..n, 1..n)$ ha sido construida por el algoritmo FLOYD.

func CAMINO_FLOYD (M, i, j) **return** lista de nodos
    if M(i, j) $= 0$ **then**
        return [ ] {secuencia vacía}
    else
        k $\leftarrow$ M(i, j)
        return CAMINO_FLOYD (M, i, k)•[k]•CAMINO_FLOYD (M, k, j)

(• simboliza la operación de concatenación de listas.)

Análisis: O(n), siendo n el número de nodos del grafo. Obsérvese que es un algoritmo lineal en la longitud del camino mínimo de i a j calculado por FLOYD.