



# Programación dinámica

IIC2133

---

## Selección de tareas con ganancias

Tenemos  $n$  tareas,

... cada una con hora de inicio  $s_i$  y hora de término  $t_i$

- que definen el intervalo de tiempo  $[s_i, f_i)$  de la tarea

... y cada una produce una ganancia  $v_i$  si es realizada

Para realizar las tareas tenemos una única máquina

... que sólo puede realizar una tarea a la vez

- es decir, si los intervalos de tiempo de dos tareas se traslapan, entonces solo se puede realizar una de ellas

¿Cuáles tareas realizar de manera que **la suma de las ganancias de las tareas realizadas sea máxima?**

- no importa el número de tareas realizadas



Suponemos que las tareas están ordenadas por hora de término:

- $f_1 \leq f_2 \leq \dots \leq f_n$

Para cada tarea  $j$  definimos  $b(j)$  como la tarea  $i$  que termina más tarde antes del inicio de  $j$

- $f_i \leq s_j$  tal que para todo  $k > i$ ,  $f_k > s_j$
- $b(j) = 0$  si ninguna tarea  $i < j$  satisface la condición anterior

Supongamos que tenemos una solución óptima  $\Omega$ ; obviamente:

- la tarea  $n$  pertenece a  $\Omega$
- ... o bien la tarea  $n$  no pertenece a  $\Omega$

Si la tarea  $n$  **pertenece** a  $\Omega$ ,

... entonces ninguna tarea  $k$ ,  $b(n) < k < n$ , puede pertenecer a  $\Omega$

... y  $\Omega$  debe incluir una solución óptima para las tareas  $1, \dots, b(n)$

Si la tarea  $n$  **no pertenece** a  $\Omega$ ,

... entonces  $\Omega$  es igual a la solución óptima para las tareas  $1, \dots, n-1$

La solución óptima para las tareas  $1, \dots, n$  involucra encontrar las soluciones óptimas a problemas más pequeños del mismo tipo

Si  $\Omega_j$  es la solución óptima al problema de las tareas 1, ...,  $j$

... y  $\text{opt}(j)$  es su valor

... entonces buscamos  $\Omega_n$  con valor  $\text{opt}(n)$

Generalizando de la diap. anterior:

- $j$  pertenece a  $\Omega_j$ , en cuyo caso  $\text{opt}(j) = v_j + \text{opt}(b(j))$
- $j$  no pertenece a  $\Omega_j$ , en cuyo caso  $\text{opt}(j) = \text{opt}(j-1)$

Por lo tanto,

$$(*) \text{opt}(j) = \max\{ v_j + \text{opt}(b(j)) , \text{opt}(j-1) \}$$

Un algoritmo recursivo para calcular  $\text{opt}(n)$ :

```
opt(j):  
    if  $j = 0$ :  
        return 0  
    else:  
        return  $\max\{ v_j + \text{opt}(b(j)) , \text{opt}(j-1) \}$ 
```

- supone que las tareas están ordenadas por hora de término y que tenemos calculados los  $b(j)$  para cada  $j$
- $\text{opt}(0) = 0$  basado en la convención de que este es el óptimo para un conjunto vacío de tareas

La corrección del algoritmo se puede demostrar por inducción



El problema de **opt** es su tiempo de ejecución en el peor caso:

- cada llamada a **opt** da origen a otras dos llamadas a **opt**
- esto es, tiempo exponencial

Pero solo está resolviendo  $n+1$  subproblemas diferentes:

- **opt**(0), **opt**(1), ..., **opt**( $n$ )
- la razón por la que toma tiempo exponencial es el número de veces que resuelve cada subproblema

Podemos guardar el valor de **opt**( $j$ ) en un arreglo global la primera vez que lo calculamos

... y luego usar este valor ya calculado en lugar de todas las futuras llamadas recursivas

```
rec-opt(j):
  if j = 0:
    return 0
  else:
    if m[j] no está vacía:
      return m[j]
    else:
      m[j] = max{  $v_j + \text{rec-opt}(j)$  ,  $\text{rec-opt}(j-1)$  }
      return m[j]
```

**rec-opt**(n) es  $O(n)$ :

- ¿por qué?

Por supuesto, además de calcular el valor de la solución óptima,  
... necesitamos saber cuál es esta solución





La clave es el arreglo  $\mathbf{m}$ :

- usamos el valor de soluciones óptimas a los subproblemas sobre las tareas  $1, 2, \dots, j$  para cada  $j$
- ... y usa (\*) para definir el valor de  $\mathbf{m}[j]$  basado en los valores que aparecen antes (en índices menores que  $j$ ) en  $\mathbf{m}$

Cuando llenamos  $\mathbf{m}$ , el problema está resuelto:

- $\mathbf{m}[n]$  contiene el valor de la solución óptima
- ... y podemos usar  $\mathbf{m}$  para recrear la solución propiamente tal

**Pero también podemos calcular los valores en  $m$  iterativamente:**

- $m[0] = 0$  y vamos incrementando  $j$
- cada vez que necesitamos calcular un valor  $m[j]$ , usamos (\*)

**it-opt:**

$m[0] = 0$

for  $j = 1, 2, \dots, n$  :

$m[j] = \max\{ v_j + m[b(j)] , m[j-1] \}$

Podemos demostrar por inducción que **it-opt** asigna a  $m[j]$  el valor  $\text{opt}(j)$

**it-opt** es claramente  $O(n)$



Para desarrollar un algoritmo de programación dinámica,

... necesitamos una colección de subproblemas, derivados del problema original, que satisfagan algunas propiedades:

- i) el número de subproblemas es (idealmente) polinomial
- ii) la solución al problema original puede calcularse a partir de las soluciones a los subproblemas
- iii) hay un orden natural de los subproblemas, desde “el más pequeño” hasta “el más grande”

... y una recurrencia fácil (ojalá) de calcular (tal como \*)

... que permite calcular la solución a un subproblema a partir de las soluciones a subproblemas más pequeños

## La mochila con objetos 0/1

Tenemos  $n$  objetos no fraccionables, cada uno con un valor  $v_k$  y un peso  $w_k$ ,


... y queremos ponerlos en una mochila con una capacidad total  $W$  ( $< \sum w_k$ , es decir, no podemos incluirlos todos)

... de manera de maximizar la suma de los valores pero sin exceder la capacidad de la mochila

Si  $knap(p, q, \omega)$  representa el problema de maximizar  $\sum v_k x_k$

... sujeto a  $\sum w_k x_k \leq \omega$  y  $x_k = \{0, 1\}$

... entonces el problema es  $knap(1, n, W)$



Sea  $y_1, y_2, \dots, y_n$  una selección óptima de valores 0/1 para  $x_1, x_2, \dots, x_n$

$y_1$  puede ser 0 o 1

Si  $y_1 = 0$  (es decir, el objeto 1 no está en la solución),

... entonces  $y_2, y_3, \dots, y_n$  debe ser una selección óptima para  $\text{knap}(2, n, W)$ :

- de lo contrario, no sería una selección óptima para  $\text{knap}(1, n, W)$

Si  $y_1 = 1$ ,

... entonces  $y_2, y_3, \dots, y_n$  debe ser una selección óptima para  $\text{knap}(2, n, W - w_1)$ :

- de lo contrario, habría otra selección  $z_2, z_3, \dots, z_n$  de valores 0/1 tal que  
...  $\sum w_k z_k \leq W - w_1$  y  $\sum v_k z_k > \sum v_k y_k, 2 \leq k \leq n$   
... y la selección  $y_1, z_2, z_3, \dots, z_n$  sería una selección para  $\text{knap}(1, n, W)$  con mayor valor

Es decir, **el problema se puede resolver a partir de las soluciones óptimas a subproblemas del mismo tipo**

Sea  $g_k(\omega)$  el valor de una solución óptima a  $\text{knap}(k+1, n, \omega)$ :

- $g_0(W)$  es el valor de una solución óptima a  $\text{knap}(1, n, W)$  —el problema original
- las decisiones posibles para  $x_1$  son 0 y 1
- de las diapos. anteriores se deduce que

$$g_0(W) = \max\{ g_1(W) , g_1(W-w_1) + v_1 \}$$

Más aún,

... si  $y_1, y_2, \dots, y_n$  es una solución óptima a  $\text{knap}(1, n, W)$ ,

... entonces para cada  $j, 1 \leq j \leq n$

$$y_1, \dots, y_j, y_{j+1}, \dots, y_n$$

... deben ser soluciones óptimas a<sup>1</sup>

$$\text{knap}(1, j, \sum w_k y_k), 1 \leq k \leq j$$


$$\text{knap}(j+1, n, W - \sum w_k y_k), 1 \leq k \leq j$$

Por lo tanto<sup>2</sup>,

$$g_k(\omega) = \max\{ g_{k+1}(\omega), g_{k+1}(\omega - w_{k+1}) + v_{k+1} \}$$

... en que  $g_n(\omega) = 0$  si  $\omega = 0$  y  $g_n(\omega) = -\infty$  si  $\omega < 0$





<sup>1</sup> significa que la solución a un subproblema puede calcularse a partir de las soluciones a subproblemas más pequeños

<sup>2</sup> significa que hay una recurrencia (fácil) de calcular

P.ej., si  $n = 3$ ,  $(w_1, w_2, w_3) = (2, 3, 4)$ ,  $(v_1, v_2, v_3) = (1, 2, 5)$ , y  $W = 6$

... tenemos que calcular

$$g_0(6) = \max\{ g_1(6), g_1(4)+1 \}$$

$$g_1(6) = \max\{ g_2(6), g_2(3)+2 \} = \max\{5, 2\} = 5, \text{ ya que}$$

$$g_2(6) = \max\{ g_3(6), g_3(2)+5 \} = \max\{0, 5\} = 5$$

$$g_2(3) = \max\{ g_3(3), g_3(-1)+5 \} = \max\{0, -\infty\} = 0$$

$$g_1(4) = \max\{ g_2(4), g_2(1)+2 \} = \max\{5, 2\} = 5, \text{ ya que}$$

$$g_2(4) = \max\{ g_3(4), g_3(0)+5 \} = \max\{0, 5\} = 5$$

$$g_2(1) = \max\{ g_3(1), g_3(-3)+5 \} = \max\{0, -\infty\} = 0$$

$$\text{Luego, } g_0(6) = \max\{5, 5 + 1\} = 6$$

## Árbol de búsqueda binario óptimo

Tenemos una lista de palabras  $w_1, w_2, \dots, w_n$  y sus respectivas frecuencias de ocurrencia  $f_1, f_2, \dots, f_n$

Queremos almacenar las palabras en un árbol de búsqueda binario

... de manera de minimizar el tiempo de acceso total esperado

El número de comparaciones necesarias para llegar a un nodo a profundidad  $p$  es  $p+1$

... de modo que si la palabra  $w_i$  está a profundidad  $p_i$ ,

... queremos minimizar el costo  $\sum f_i (1 + p_i)$  del árbol

Supongamos que para las palabras  $w_{izq}$  hasta  $w_{der}$ , el árbol óptimo tiene como raíz a  $w_k$

Entonces

- el subárbol izquierdo contiene las palabras  $w_{izq}$  hasta  $w_{k-1}$   
... y supongamos que su costo es  $C_{izq,k-1}$
- el subárbol derecho contiene las palabras  $w_{k+1}$  hasta  $w_{der}$   
... y supongamos que su costo es  $C_{k+1,der}$
- ambos subárboles deben ser óptimos

Y nuestro árbol óptimo (de arriba) tiene costo

$$\begin{aligned} C_{izq,der} &= \min_k \{ f_k + C_{izq,k-1} + C_{k+1,der} + \sum_{izq,k-1} f_j + \sum_{k+1,der} f_j \} \\ &= \min_k \{ C_{izq,k-1} + C_{k+1,der} + \sum_{izq,der} f_j \} \quad , izq \leq k \leq der \end{aligned}$$

Supongamos  $n = 7$  y que las palabras y sus frecuencias son

$w = (a, am, and, egg, if, the, two)$

$f = (0.22, 0.18, 0.20, 0.05, 0.25, 0.02, 0.08)$

En la primera iteración encontramos los árboles óptimos para los siguientes 7 rangos de palabras (de una palabra c/u):

a...a, am...am, and...and, egg...egg, if...if, the...the, two...two

En la segunda, encontramos los árboles óptimos para los 6 rangos de dos palabras (la raíz en **negrita**):


**a**...am(.58), am...**and**(.56), **and**...egg(.30), egg...**if**(.35), **if**...the(.29), the...**two**(.12)

En la tercera, los árboles óptimos para 5 rangos de tres palabras:

a...and(**am**,1.02), am...egg(**and**,.66), and...if(**if**,.80), egg...the(**if**,.39), if...two(**if**,.47)

... Y así hasta la séptima iteración:

un rango de 7 palabras con raíz **and** y costo 2.12



Vimos cómo aplicar PD a un problema de selección de un subconjunto —mochila 0/1

Veamos ahora un problema de selección de una permutación

Los problemas de selección de permutaciones son más difíciles que los de selección de subconjuntos:

... hay  $n!$  permutaciones diferentes de  $n$  objetos

... hay sólo  $2^n$  subconjuntos diferentes de  $n$  objetos

...  $n! > 2^n$ ,  $n \geq 4$

## El Vendedor Viajero

Sea  $G = (V, E)$  un grafo direccional cuyas aristas tienen costos  $c_{ij} > 0$ ,  $\forall (i, j) \in E$ :

$$c_{ij} = \infty \quad \text{si } (i, j) \notin E \qquad |V| = n > 1$$

Un tour de  $G$  es un ciclo simple direccional que incluye a todos los vértices de  $V$ :

- su costo es la suma de los costos de las aristas que lo forman

El problema del vendedor viajero consiste en encontrar un tour de costo mínimo



Suponemos que un tour empieza y termina en el vértice 1

Todo tour tiene dos tramos:

- una arista  $(1, k)$  para algún  $k \in V - \{1\}$
- una ruta desde el vértice  $k$  al vértice 1 que pasa una vez por cada vértice en  $V - \{1, k\}$

Si el tour es óptimo, entonces la ruta de  $k$  a 1 debe ser una ruta más corta de  $k$  a 1 que pasa por todos los vértices en  $V - \{1, k\}$ :

- el problema exhibe subestructura óptima (satisface el principio de optimalidad)



Sea  $g(i, S)$  la longitud de la ruta más corta que comienza en el vértice  $i$ , pasa por todos los vértices en  $S$  una vez, y termina en el vértice 1:

$g(1, V - \{1\})$  es la longitud de un tour óptimo

del principio de optimalidad, se deduce que

$$g(1, V - \{1\}) = \min_{2 \leq k \leq n} \{ c_{1k} + g(k, V - \{1, k\}) \}$$

generalizando,

$$g(i, S) = \min_{j \in S} \{ c_{ij} + g(j, S - \{j\}) \}$$

Podemos saber el valor de  $g(1, V - \{1\})$  si conocemos los valores de  $g(k, V - \{1, k\})$  para todos los valores posibles de  $k$ .

Partiendo de la última ecuación:

$$g(i, \emptyset) = c_{i1}, 1 \leq i \leq n$$

luego, obtenemos  $g(i, S)$  para todos los  $S$  de tamaño 1

luego, obtenemos  $g(i, S)$  para todos los  $S$  de tamaño 2

etc.

Podemos construir el tour óptimo si mantenemos para cada  $g(i, S)$  el valor de  $j$  que minimiza el lado derecho de la ecuación

Ejemplo:

Determinar el tour óptimo y su costo para el grafo direccional de 4 vértices representado por la siguiente matriz de adyacencias

	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

$$g(2, \emptyset) = c_{21} = 5$$

$$g(3, \emptyset) = c_{31} = 6$$

$$g(4, \emptyset) = c_{41} = 8$$

$$g(2, \{3\}) = c_{23} + g(3, \emptyset) = 15 \quad g(2, \{4\}) = 18$$

$$g(3, \{2\}) = 18 \quad g(3, \{4\}) = 20$$

$$g(4, \{2\}) = 13 \quad g(4, \{3\}) = 15$$

$$g(2, \{3, 4\}) = \min\{c_{23} + g(3, \{4\}), c_{24} + g(4, \{3\})\} = 25$$

$$g(3, \{2, 4\}) = \min\{c_{32} + g(2, \{4\}), c_{34} + g(4, \{2\})\} = 25$$

$$g(4, \{2, 3\}) = \min\{c_{42} + g(2, \{3\}), c_{43} + g(3, \{2\})\} = 23$$

$$g(1, \{2, 3, 4\}) = \min\{c_{12} + g(2, \{3, 4\}), c_{13} + g(3, \{2, 4\}), c_{14} + g(4, \{2, 3\})\}$$

$$= \min\{35, 40, 43\} = 35$$

Así, el costo de un tour óptimo para el grafo del ejemplo es 35.



¿Cuál es el tour óptimo?

- el tour empieza en el vértice 1
- de la evaluación de  $g(1, \{2, 3, 4\})$ , determinamos que el siguiente vértice es el 2
- luego, de la evaluación de  $g(2, \{3, 4\})$  determinamos que el próximo vértice es el 4

Por lo tanto, el tour óptimo es  $\langle 1, 2, 4, 3, 1 \rangle$