



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

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

Cada una produce una ganancia v_i si es realizada

¿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 Ω

Obviamente:

- la tarea n pertenece a Ω
- ... o bien la tarea n no pertenece a Ω

Si la tarea n **no pertenece** a Ω ,

... entonces Ω es igual a la solución óptima para las tareas 1, ..., $n-1$ (un problema del mismo tipo, pero más pequeño)

...

...

En cambio, si la tarea n **pertenece** a Ω ,

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

... y Ω debe incluir, además de la tarea n , una solución óptima para las tareas $1, \dots, b(n)$ (un problema del mismo tipo, pero más pequeño)

Es decir, en ambos casos, la solución óptima para las tareas 1, ..., n involucra encontrar las soluciones óptimas a problemas más pequeños del mismo tipo

Si Ω_j es la solución óptima al problema de las tareas 1, ..., j

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

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

Generalizando de las diaps. anteriores:

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

Por lo tanto,

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

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:

- **$\text{opt}(0)$, $\text{opt}(1)$, ..., $\text{opt}(n)$**
- 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(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 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 $m[j]$ basado en los valores que aparecen antes (en índices menores que j) en m

Cuando llenamos m , el problema está resuelto:

- $m[n]$ contiene el valor de la solución óptima
- ... y podemos usar m para reconstruir la solución propiamente tal

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

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

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:

- próx. diapo.

- 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 * en diap. #10)
- ... 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

$$\dots \sum w_k z_k \leq W-w_1 \text{ y } \sum v_k z_k > \sum v_k y_k, 2 \leq k \leq n$$

... 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 a¹

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

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

Por lo tanto²,

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

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

¹ significa que la solución a un subproblema puede calcularse a partir de las soluciones a subproblemas del mismo tipo más pequeños

² 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

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

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

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

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

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

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

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

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

Rutas más cortas entre todos los pares de vértices

Podemos ejecutar $|V|$ veces un algoritmo para rutas más cortas desde un vértice, una vez para cada vértice en el rol de s :

- si los costos de las aristas son no negativos, podemos usar el algoritmo de Dijkstra
... el tiempo de ejecución sería $O(VE \log V)$

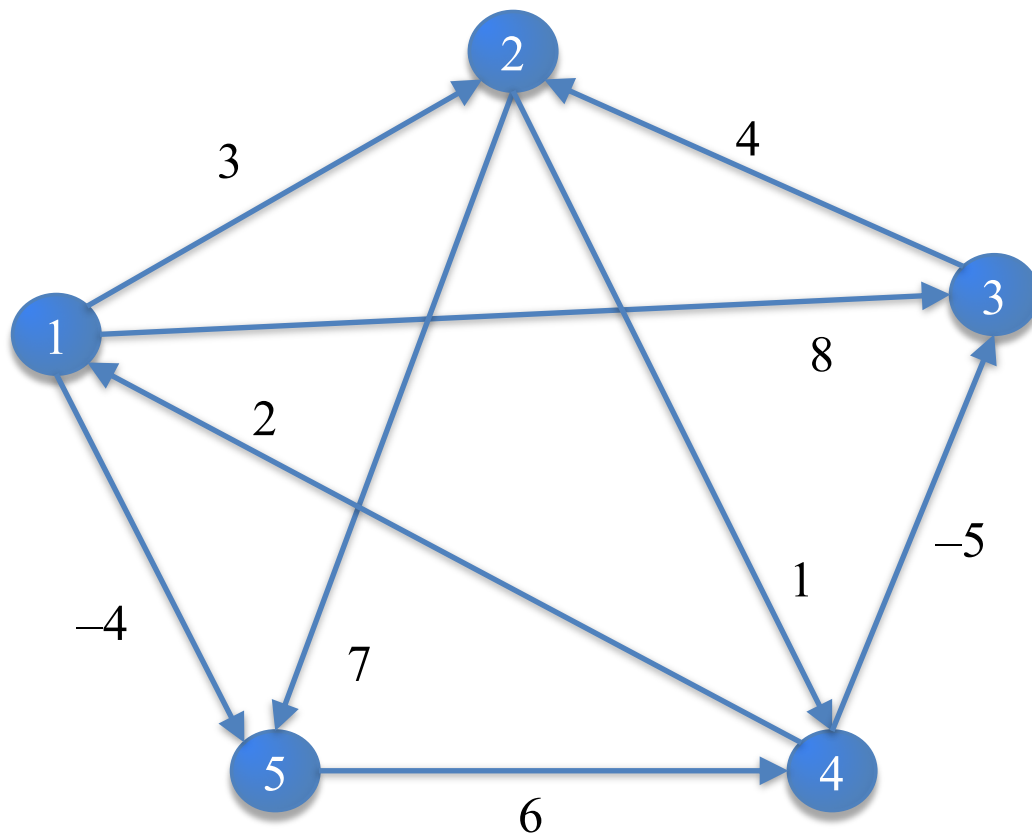
- si las aristas pueden tener costos negativos, debemos usar el algoritmo de Bellman-Ford
... el tiempo de ejecución sería $O(V^2E)$, que para grafos densos es $O(V^4)$

Podemos mejorar este último desempeño

Representaremos G por su *matriz de adyacencias* (en vez de las listas de adyacencias, que hemos usado mayoritariamente)

Si los vértices están numerados $1, 2, \dots, n$ (o sea, $|V| = n$),

... el input es una matriz W que representa los costos de las aristas



$W =$

0	3	8	∞	-4
∞	0	∞	1	7
∞	4	0	∞	∞
2	∞	-5	0	∞
∞	∞	∞	6	0

$W = (\omega_{ij})$, en que

$$\omega_{ij} = 0 \quad \text{si } i = j$$

= costo de la arista direccional (i, j) si $i \neq j$ y $(i, j) \in E$

$$= \infty \quad \text{si } i \neq j \text{ y } (i, j) \notin E$$

Suponemos que G **no contiene ciclos de costo negativo**

El algoritmo de Floyd-Warshall

El algoritmo considera los vértices intermedios de una ruta más corta

Si los vértices de G son $V = \{1, 2, \dots, n\}$, consideremos el subconjunto $\{1, 2, \dots, k\}$, para algún k



Para cualquier par de vértices $i, j \in V$,

... consideremos todas las rutas de i a j cuyos vértices intermedios están todos tomados del conjunto $\{1, 2, \dots, k\}$

... y sea p una ruta más corta entre ellas

k puede ser o no un vértice (intermedio) de p

Si k **no es** un vértice de p ,

... entonces todos los vértices (intermedios) de p están en el conjunto $\{1, 2, \dots, k-1\}$

\Rightarrow una ruta más corta de i a j con todos los vértices intermedios en $\{1, 2, \dots, k-1\}$

... es también una ruta más corta de i a j con todos los vértices intermedios en $\{1, 2, \dots, k\}$

Si k es un vértice de p , entonces podemos dividir p en dos tramos:

el tramo p_1 de i a k

... y el tramo p_2 de k a j

⇒ por el principio de optimalidad, p_1 es una ruta más corta de i a k con todos los vértices intermedios en $\{1, 2, \dots, k-1\}$

... y p_2 es una ruta más corta de k a j con todos los vértices intermedios en $\{1, 2, \dots, k-1\}$

Sea $d_{ij}^{(k)}$ el costo de una ruta más corta de i a j , tal que todos los vértices intermedios están en el conjunto $\{1, 2, \dots, k\}$

Cuando $k = 0$,

... una ruta de i a j sin vértices intermedios con número mayor que 0

... simplemente no tiene vértices intermedios,

... y por lo tanto tiene a lo más una arista $\Rightarrow d_{ij}^{(0)} = \omega_{ij}$

Definimos $d_{ij}^{(k)}$ recursivamente por

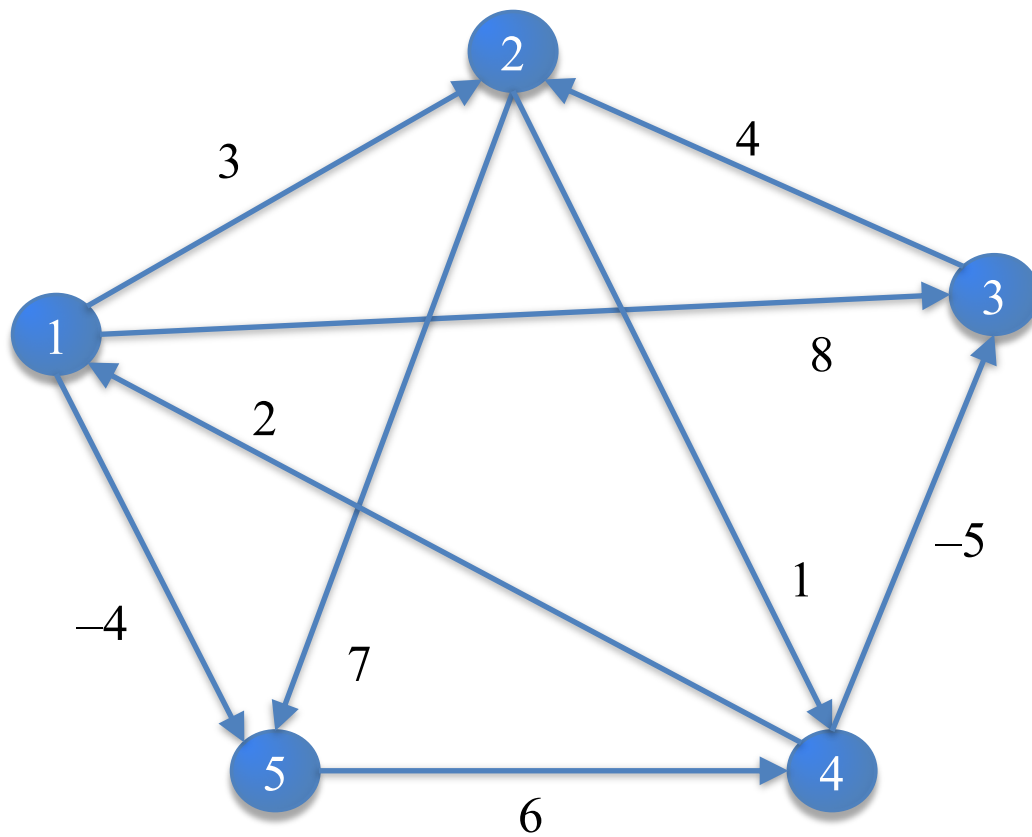
$$\begin{aligned} d_{ij}^{(k)} &= \omega_{ij} && \text{si } k = 0 \\ &= \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) && \text{si } k \geq 1 \end{aligned}$$

La matriz $D^{(n)} = (d_{ij}^{(n)})$ da la respuesta final:

$$d_{ij}^{(n)} = \delta(i, j) \text{ para todo } i, j \in V$$

El algoritmo de Floyd-Warshall, *bottom-up*, toma tiempo $O(V^3)$

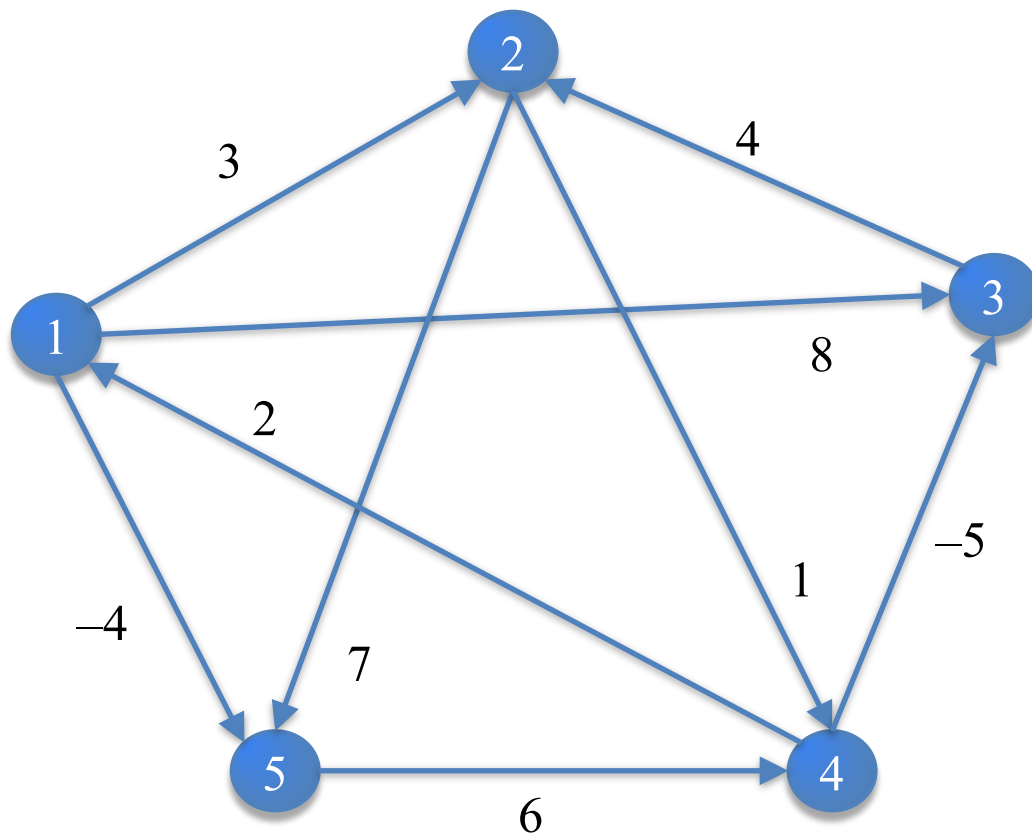
```
D(0) = W
for k = 1 ... n:
    sea D(k) = (dij(k)) una nueva matriz
    for i = 1 ... n:
        for j = 1 ... n:
            dij(k) = min(dij(k-1), dik(k-1) + dkj(k-1))
return D(n)
```



$$D^{(0)} = \begin{matrix} & 0 & 3 & 8 & \infty & -4 \\ & \infty & 0 & \infty & 1 & 7 \\ & \infty & 4 & 0 & \infty & \infty \\ 2 & 2 & \infty & -5 & 0 & \infty \\ & \infty & \infty & \infty & 6 & 0 \end{matrix}$$

$$D^{(1)} = \begin{matrix} & 0 & 3 & 8 & \infty & -4 \\ & \infty & 0 & \infty & 1 & 7 \\ & \infty & 4 & 0 & \infty & \infty \\ 2 & 2 & 5 & -5 & 0 & -2 \\ & \infty & \infty & \infty & 6 & 0 \end{matrix}$$

$$D^{(2)} = \begin{matrix} & 0 & 3 & 8 & 4 & -4 \\ & \infty & 0 & \infty & 1 & 7 \\ & \infty & 4 & 0 & 5 & 11 \\ 2 & 2 & 5 & -5 & 0 & -2 \\ & \infty & \infty & \infty & 6 & 0 \end{matrix}$$



$$D^{(3)} = \begin{matrix} & 0 & 3 & 8 & 4 & -4 \\ & \infty & 0 & \infty & 1 & 7 \\ & \infty & 4 & 0 & 5 & 11 \\ 2 & 2 & -1 & -5 & 0 & -2 \\ & \infty & \infty & \infty & 6 & 0 \end{matrix}$$

$$D^{(4)} = \begin{matrix} & 0 & 3 & -1 & 4 & -4 \\ & 3 & 0 & -4 & 1 & -1 \\ & 7 & 4 & 0 & 5 & 3 \\ 2 & 2 & -1 & -5 & 0 & -2 \\ & 8 & 5 & 1 & 6 & 0 \end{matrix}$$

$$D^{(5)} = \begin{matrix} & 0 & 1 & -3 & 2 & -4 \\ & 3 & 0 & -4 & 1 & -1 \\ & 7 & 4 & 0 & 5 & 3 \\ 2 & 2 & -1 & -5 & 0 & -2 \\ & 8 & 5 & 1 & 6 & 0 \end{matrix}$$