




# Algoritmos codiciosos

IIC2133



Los algoritmos codiciosos se usan para resolver problemas que en su mayoría tienen las siguientes características –**paradigma de subconjuntos**:

- hay  $n$  inputs
- queremos obtener un subconjunto que satisfaga ciertas restricciones → una *solución factible*
- ... y, en particular, queremos una solución factible que, ya sea, maximice o minimice una *función objetivo* → una **solución óptima**

## P.ej., el problema de la mochila con objetos fraccionables

Tenemos  $n$  objetos y una mochila:

- el objeto  $k$  pesa  $w_k$
- la capacidad, en peso, de la mochila es  $m$
- si incluimos en la mochila una fracción  $x_k$  del objeto  $k$ , entonces obtenemos una ganancia  $p_k x_k$

$n$  inputs

El problema es cómo llenar la mochila cumpliendo tres condiciones:

i) maximizar la ganancia total obtenida

$$\sum_{k=1}^n p_k x_k$$

función  
objetivo

ii) sin exceder la capacidad de la mochila


$$\sum_{k=1}^n w_k x_k \leq m$$

restricciones

iii)  $0 \leq x_k \leq 1, 1 \leq k \leq n$

Una *solución factible* es un conjunto  $\{x_1, \dots, x_n\}$  que cumple ii) y iii)

Una *solución óptima* es una solución factible que cumple i)




Los algoritmos codiciosos trabajan en etapas, considerando un input a la vez:

- en cada etapa, se toma una decisión con respecto a si el input considerado pertenece, o no, a la solución óptima
- la idea es que una vez que la decisión es tomada, es final
  - ninguna de las decisiones que se tomen más adelante va a hacer que esta decisión cambie


En el problema de la mochila, se trata de seleccionar un subconjunto de objetos

... y determinar, para cada uno, la fracción  $x_k$ ,  $0 \leq x_k \leq 1$ , que debe ir en la mochila



Para esto, los inputs van siendo considerados en orden según un *procedimiento de selección*:

- si la inclusión del próximo input en la solución óptima parcial produce una solución infactible, entonces el input se descarta
- ... en otro caso, el input es agregado a la solución



El procedimiento de selección está basado en alguna *medida de optimización* o *estrategia codiciosa*:


- seleccionamos un input de forma localmente óptima  
... y esperamos que esta selección nos lleve a una solución globalmente óptima



Posibles medidas de optimización o estrategias codiciosas para el problema de la mochila:

- incluir a continuación el objeto con mayor ganancia
- incluir a continuación el objeto con menor peso
- incluir a continuación el objeto con mayor cociente ganancia/peso





En general, varias medidas de optimización diferentes pueden ser plausibles para un problema dado

... pero normalmente la mayoría de ellas va a producir soluciones subóptimas:

- los algoritmos codiciosos *no siempre* producen soluciones óptimas

P.ej.,  $n = 3$

$$w_1, w_2, w_3 = 18, 15, 10$$

$m = 20$

$$p_1, p_2, p_3 = 25, 24, 15$$


Algunas soluciones factibles:

$x_1, x_2, x_3$	$\sum_{k=1}^n w_k x_k$	$\sum_{k=1}^n p_k x_k$
a) 1, 2/15, 0	20	28.2
b) 0, 2/3, 1	20	31
c) 0, 1, 1/2	20	31.5



## Posibles medidas de optimización para el problema de la mochila:

- incluir a continuación el objeto con mayor ganancia → solución a), que no es óptima
- incluir a continuación el objeto con menor peso → solución b), que no es óptima
- **incluir a continuación el objeto con mayor cociente ganancia/peso → solución c), que es óptima**



En general, varias medidas de optimización diferentes pueden ser plausibles para un problema dado

... pero normalmente la mayoría de ellas va a producir soluciones subóptimas:

- los algoritmos codiciosos *no siempre* producen soluciones óptimas

**Esto implica que para estar seguros de que una estrategia codiciosa efectivamente produce una solución óptima es necesario demostrar que es así**

## Algoritmo abstracto del método codicioso para el paradigma de subconjuntos

contiene los  $n$  inputs

selecciona y saca un input de  $a$

```
greedy(a[], n):  
    solution = empty  
    for i = 1, ..., n:  
        x = select(a)  
        if feasible(solution, x):  
            solution = union(solution, x)  
    return solution
```

agrega x a la solución

función booleana que determina  
si x puede ser incluida en la solución

## Programación de charlas en una misma sala

Tenemos  $n$  charlas,

... cada una con una hora de inicio  $s_i$  y una hora de fin  $f_i$

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

Para dar las charlas tenemos una única sala

... en la que sólo se puede dar una charla a la vez

- si los intervalos de tiempo de dos charlas se traslapan, entonces sólo se puede dar una de ellas

El propósito es maximizar el número de charlas dadas

## Programación de charlas en una misma sala

Tenemos  $n$  charlas,

$n$  inputs

... cada una con una hora de inicio  $s_i$  y una hora de fin  $f_i$

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

Para dar las charlas tenemos una única sala

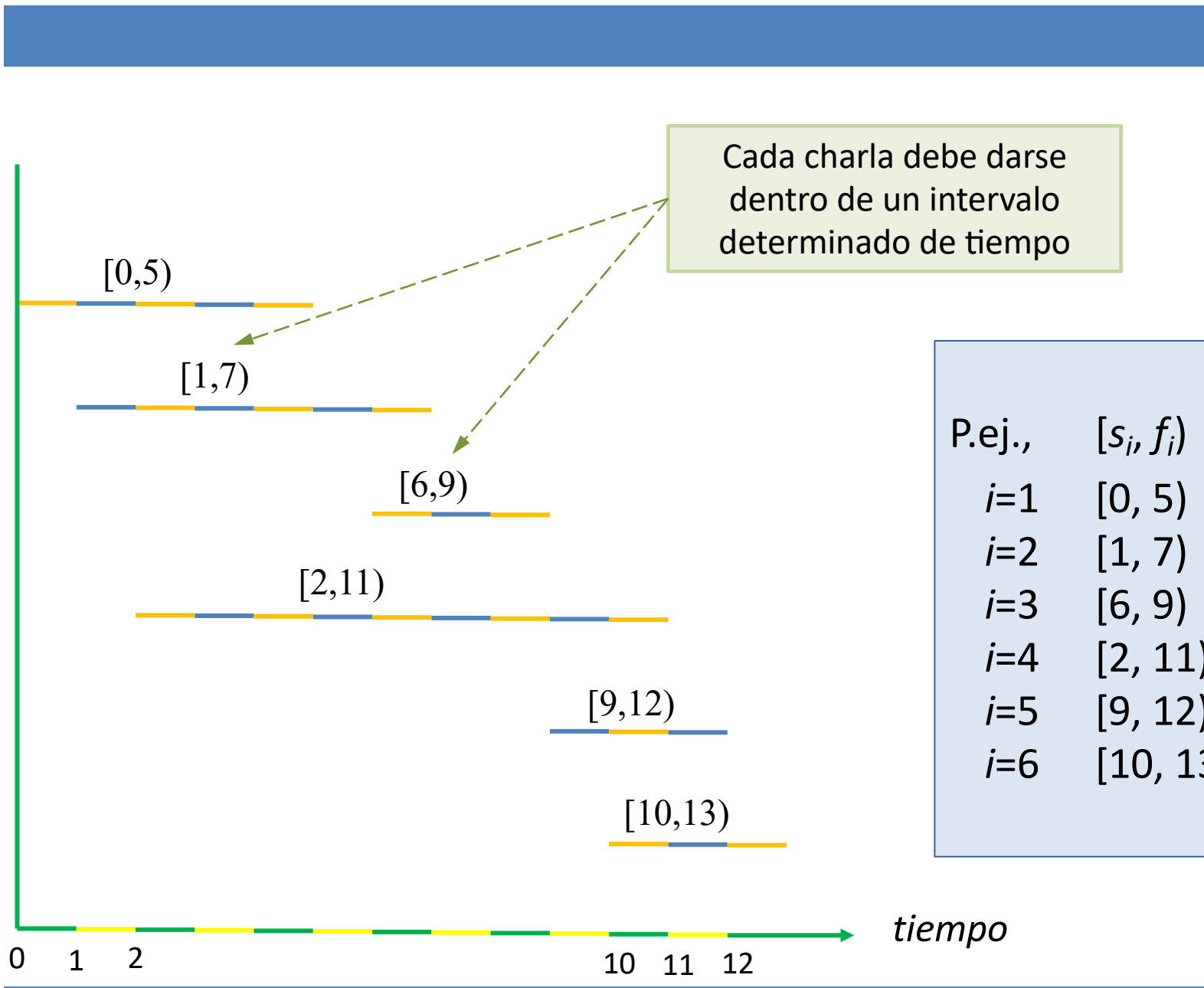
... en la que sólo se puede dar una charla a la vez

restricciones

- si los intervalos de tiempo de dos charlas se traslapan, entonces sólo se puede dar una de ellas

El propósito es maximizar el número de charlas dadas

función  
objetivo





Tres estrategias codiciosas que, en general, **no** producen una solución óptima

a) elegir primero la charla que *empieza más temprano*




b) elegir primero la charla *más corta*



Intervalo de tiempo de una charla; el tiempo transcurre de izquierda a derecha

c) elegir primero la charla que *tiene menos incompatibilidades* con otras charlas



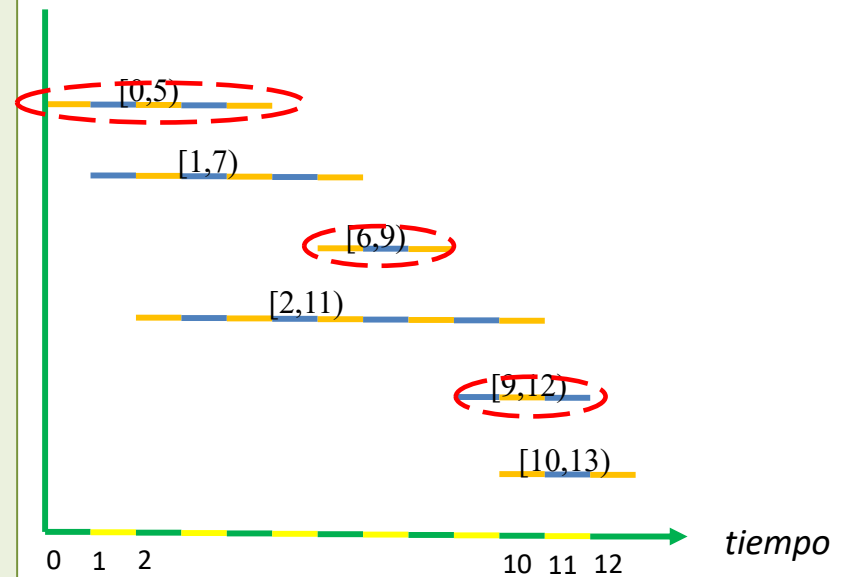


Sin embargo, el problema sí puede resolverse mediante una estrategia codiciosa:

**elegir primero la charla que termina más temprano**

En el ej., las charlas elegidas son 1, 3 y 5, y el número de charlas dadas es 3:

- la charla 1 es la que termina más temprano, en  $t = 5$
- ( la charla 2 es la segunda charla en terminar, pero es incompatible con la charla 1  $\Rightarrow$  la descartamos )
- la charla 3 es la tercera en terminar, en  $t = 9$ , y es compatible con la charla 1
- ( la charla 4 es incompatible con las charlas 1 y 3 )
- la charla 5 es compatible con la 1 y la 3
- ( la charla 6 es incompatible con la 5 )



## Selección de tareas con plazos y ganancias

Se acerca el final del semestre y tengo que hacer  $n$  tareas:


- tareas propiamente tales, estudiar para una l o examen, terminar un proyecto, preparar una presentación, ...

Cada tarea  $i$  tiene dos propiedades:

- un plazo  $d_i$  —un día del mes, tal que el día 1 es mañana
- una ganancia  $p_i$  —lo que hacer bien la tarea significa para mí— que la obtengo si y sólo si hago la tarea a tiempo (dentro del plazo)

Para hacer una tarea, tengo que dedicarle todo un día

→ sólo puedo hacer una tarea al día



Una solución factible es un subconjunto  $T$  de tareas tal que puedo hacer cada tarea en  $T$  a tiempo

El **valor** de  $T$  es la suma  $\sum_{k \in T} p_k$  de las ganancias de las tareas incluidas en  $T$

Una solución factible es *óptima* si su valor es máximo

P.ej., para  $n = 4$ , sean:

$$p_1, p_2, p_3, p_4 = 100, 10, 15, 27$$

$$d_1, d_2, d_3, d_4 = 2, 1, 2, 1$$

solución factible $T$	orden de procesamiento	valor
$\{1, 2\}$	2, 1	110
$\{1, 3\}$	1, 3 o 3, 1	115
$\{1, 4\}$	4, 1	127
$\{2, 3\}$	2, 3	25
$\{3, 4\}$	4, 3	42
$\{1\}$	1	100
$\{2\}$	2	10
$\{3\}$	3	15
$\{4\}$	4	27

Tomemos la propia función objetivo como medida de optimización (o estrategia codiciosa) para elegir la próxima tarea:

- la próxima tarea a considerar es la que aumenta más la suma  $\sum_{k \in T} p_k$ , sujeta a que el  $T$  resultante sea factible
- consideramos las tareas en orden decreciente de los  $p_i$ 's

La estrategia funciona en el ej.:

- inicialmente,  $T = \emptyset$  y  $\sum p_i = 0$  y consideramos las tareas en el orden 1, 4, 3 y 2
- agregamos la tarea 1 a  $T$  —tiene la mayor ganancia y  $T = \{1\}$  es factible
- agregamos la tarea 4 a  $T$  — $T = \{1, 4\}$  es factible
- consideramos la tarea 3, pero la descartamos — $T = \{1, 3, 4\}$  no es factible
- finalmente, consideramos la tarea 2 y la descartamos — $T = \{1, 2, 4\}$  no es factible