

Algorítmica

Conrado Martínez
U. Politècnica Catalunya

ALG Q2-2021–2022



Temario

- Parte 1: Repaso de Conceptos Algorítmicos
- Parte 2: Búsqueda y Ordenación Digital
- Parte 3: Algoritmos Voraces
- Parte 4: Programación Dinámica
- Parte 5: Flujos sobre Redes y Programación Lineal

Parte I

Repaso de Conceptos Algorítmicos

Parte I

Repasso de Conceptos Algorítmicos

- Análisis de Algoritmos
- Divide y vencerás
- Grafos y Recorridos

- Eficiencia de un algoritmo = consumo de recursos de cómputo: tiempo de ejecución y espacio de memoria
- Análisis de algoritmos → Propiedades sobre la eficiencia de algoritmos
 - Comparar soluciones algorítmicas alternativas
 - Predecir los recursos que usará un algoritmo o ED
 - Mejorar los algoritmos o EDs existentes y guiar el diseño de nuevos algoritmos

En general, dado un algoritmo A cuyo conjunto de entradas es \mathcal{A} su **eficiencia** o **coste** (en tiempo, en espacio, en número de operaciones de E/S, etc.) es una función T de \mathcal{A} en \mathbb{N} (o \mathbb{Q} o \mathbb{R} , según el caso):

$$\begin{aligned} T : \mathcal{A} &\rightarrow \mathbb{N} \\ \alpha &\rightarrow T(\alpha) \end{aligned}$$

Ahora bien, caracterizar la función T puede ser muy complicado y además proporciona información inmanejable, difícilmente utilizable en la práctica.

Sea \mathcal{A}_n el conjunto de entradas de tamaño n y $T_n : \mathcal{A}_n \rightarrow \mathbb{N}$ la función T restringida a \mathcal{A}_n .

- *Coste en caso mejor:*

$$T_{\text{mejor}}(n) = \min\{T_n(\alpha) \mid \alpha \in \mathcal{A}_n\}.$$

- *Coste en caso peor:*

$$T_{\text{peor}}(n) = \max\{T_n(\alpha) \mid \alpha \in \mathcal{A}_n\}.$$

- *Coste promedio:*

$$\begin{aligned} T_{\text{avg}}(n) &= \sum_{\alpha \in \mathcal{A}_n} \Pr(\alpha) T_n(\alpha) \\ &= \sum_{k \geq 0} k \Pr(T_n = k). \end{aligned}$$

1 Para todo $n \geq 0$ y para cualquier $\alpha \in \mathcal{A}_n$

$$T_{\text{mejor}}(n) \leq T_n(\alpha) \leq T_{\text{peor}}(n).$$

2 Para todo $n \geq 0$

$$T_{\text{mejor}}(n) \leq T_{\text{avg}}(n) \leq T_{\text{peor}}(n).$$

Estudiaremos generalmente sólo el coste en caso peor:

- 1 Proporciona garantías sobre la eficiencia del algoritmo, el coste **nunca** excederá el coste en caso peor
- 2 Es más fácil de calcular que el coste promedio

Una característica esencial del coste (en caso peor, en caso mejor, promedio) es su **tasa de crecimiento**

Ejemplo

- 1 Funciones lineales:

$$f(n) = a \cdot n + b \Rightarrow f(2n) \approx 2 \cdot f(n)$$

- 2 Funciones cuadráticas:

$$q(n) = a \cdot n^2 + b \cdot n + c \Rightarrow q(2n) \approx 4 \cdot q(n)$$

Se dice que las funciones lineales y las cuadráticas tienen tasas de crecimiento distintas. También se dice que son de **órdenes de magnitud** distintos.

$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	262144
5	32	160	1024	32768	$6,87 \cdot 10^{10}$
6	64	384	4096	262144	$4,72 \cdot 10^{21}$
...					
ℓ	N	L	C	Q	E
$\ell + 1$	$2N$	$2(L + N)$	$4C$	$8Q$	E^2

Los factores constantes y los términos de orden inferior son irrelevantes desde el punto de vista de la tasa de crecimiento:
p.e. $30n^2 + \sqrt{n}$ tiene la misma tasa de crecimiento que
 $2n^2 + 10n \Rightarrow$ notación asintótica

Definición

Dada una función $f : \mathbb{R}^+ \rightarrow \mathbb{R}^+$ la clase $\mathcal{O}(f)$ (O-grande de f) es

$$\mathcal{O}(f) = \{g : \mathbb{R}^+ \rightarrow \mathbb{R}^+ \mid \exists n_0 \exists c \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$$

En palabras, una función g está en $\mathcal{O}(f)$ si existe una constante c tal que $g < c \cdot f$ para toda n a partir de un cierto punto (n_0).

Aunque $\mathcal{O}(f)$ es un conjunto de funciones por tradición se escribe a veces $g = \mathcal{O}(f)$ en vez de $g \in \mathcal{O}(f)$. Sin embargo, $\mathcal{O}(f) = g$ no tiene sentido.

Propiedades básicas de la notación \mathcal{O} :

- 1 Si $\lim_{n \rightarrow \infty} g(n)/f(n) < +\infty$ entonces $g \in \mathcal{O}(f)$
- 2 Es reflexiva: para toda función $f : \mathbb{R}^+ \rightarrow \mathbb{R}^+$, $f \in \mathcal{O}(f)$
- 3 Es transitiva: si $f \in \mathcal{O}(g)$ y $g \in \mathcal{O}(h)$ entonces $f \in \mathcal{O}(h)$
- 4 Para toda constante $c > 0$, $\mathcal{O}(f) = \mathcal{O}(c \cdot f)$

Los factores constantes no son relevantes en la notación asintótica y los omitiremos sistemáticamente: p.e. hablaremos de $\mathcal{O}(n)$ y no de $\mathcal{O}(4 \cdot n)$; no expresaremos la base de logaritmos ($\mathcal{O}(\log n)$), ya que podemos pasar de una base a otra multiplicando por el factor apropiado:

$$\log_c x = \frac{\log_b x}{\log_b c}$$

Otras notaciones asintóticas son Ω (omega), Θ (zeta), o (little-oh) y ω (little-omega). La primera define un conjunto de funciones acotada inferiormente por una dada:

$$\Omega(f) = \{g: \mathbb{R}^+ \rightarrow \mathbb{R}^+ \mid \exists n_0 \exists c > 0 \forall n \geq n_0: g(n) \geq c \cdot f(n)\}$$

La notación Ω es reflexiva y transitiva; si $\lim_{n \rightarrow \infty} g(n)/f(n) > 0$ entonces $g \in \Omega(f)$. Por otra parte, si $f \in \mathcal{O}(g)$ entonces $g \in \Omega(f)$ y viceversa.

Se dice que $\mathcal{O}(f)$ es la clase de las funciones que crecen no más rápido que f . Análogamente, $\Omega(f)$ es la clase de las funciones que crecen no más despacio que f .

La notación $o(f)$ designa el conjunto de funciones que crecen estrictamente más despacio que f :

$$g \in o(f) \iff g \in \mathcal{O}(f) \wedge f \notin \mathcal{O}(g).$$

Ej: $n \in o(n^2)$, $n^2 \notin o(n^2)$

Recíprocamente $\omega(f)$ es el conjunto de funciones que crecen estrictamente más rápido que f :

$$g \in \omega(f) \iff g \in \Omega(f) \wedge f \notin \Omega(g).$$

Ej: $n^2 \in \omega(n)$, $n \notin \omega(n)$

- Las notaciones o y ω son transitivas pero **no** reflexivas:
 $f \notin o(f)$, $f \notin \omega(f)$.
- Si $\lim_{n \rightarrow \infty} g(n)/f(n) = 0$ entonces $g \in o(f)$; si
 $\lim_{n \rightarrow \infty} g(n)/f(n) = \infty$ entonces $g \in \omega(f)$.
- La relación entre o y ω es análoga a la de \mathcal{O} y Ω : si
 $f \in o(g)$ entonces $g \in \omega(f)$ y viceversa.

Finalmente,

$$\Theta(f) = \Omega(f) \cap \mathcal{O}(f)$$

es la clase de las funciones con la misma tasa de crecimiento que f .

La notación Θ es reflexiva y transitiva, como las otras. Es además simétrica: $f \in \Theta(g)$ si y sólo si $g \in \Theta(f)$. Si $\lim_{n \rightarrow \infty} g(n)/f(n) = c$ donde $0 < c < \infty$ entonces $g \in \Theta(f)$.

Propiedades adicionales de las notaciones asintóticas (las inclusiones son estrictas):

- 1 Para cualesquiera constantes $\alpha < \beta$, si f es una función creciente entonces $\mathcal{O}(f^\alpha) \subset \mathcal{O}(f^\beta)$; $f^\alpha \in o(f^\beta)$
- 2 Para cualesquiera constantes a y $b > 0$, si f es creciente, $\mathcal{O}((\log f)^a) \subset \mathcal{O}(f^b)$; $(\log f)^a \in o(f^b)$
- 3 Para cualquier constante $a \geq 0$ y $c > 1$, si f es creciente, $\mathcal{O}(f^a) \subset \mathcal{O}(c^f)$; $f^a \in o(c^f)$

Los operadores convencionales (sumas, restas, divisiones, etc.) sobre clases de funciones definidas mediante una notación asintótica se extienden de la siguiente manera:

$$A \otimes B = \{h \mid \exists f \in A \wedge \exists g \in B : h = f \otimes g\},$$

donde A y B son conjuntos de funciones. Expresiones de la forma $f \otimes A$ donde f es una función se entenderá como $\{f\} \otimes A$.

Este convenio nos permite escribir de manera cómoda expresiones como $n + \mathcal{O}(\log n)$, $n^{\mathcal{O}(1)}$, ó $\Theta(1) + \mathcal{O}(1/n)$.

Regla de las sumas:

$$\Theta(f) + \Theta(g) = \Theta(f + g) = \Theta(\max\{f, g\}).$$

Regla de los productos:

$$\Theta(f) \cdot \Theta(g) = \Theta(f \cdot g).$$

Reglas similares se cumplen para las notaciones \mathcal{O} y Ω .

Análisis de algoritmos iterativos

- 1 El coste de una operación elemental es $\Theta(1)$.
- 2 Si el coste de un fragmento S_1 es f y el de S_2 es g entonces el coste de $S_1; S_2$ es $f + g$.
- 3 Si el coste de S_1 es f , el de S_2 es g y el coste de evaluar B es h entonces el coste en caso peor de

```
if  $B$  then  $S_1$ 
else  $S_2$ 
end if
```

es $\max\{f + h, g + h\}$.

- 4 Si el coste de S durante la i -ésima iteración es f_i , el coste de evaluar B es h_i y el número de iteraciones es g entonces el coste T de

while B **do**

S

end while

es

$$T(n) = \sum_{i=1}^{i=g(n)} f_i(n) + h_i(n).$$

Si $f = \max\{f_i + h_i\}$ entonces $T = \mathcal{O}(f \cdot g)$.

Análisis de algoritmos recursivos

El coste (en caso peor, medio, ...) de un algoritmo recursivo $T(n)$ satisface, dada la naturaleza del algoritmo, una ecuación **recurrente**: esto es, $T(n)$ dependerá del valor de T para tamaños menores. Frecuentemente, la recurrencia adopta una de las dos siguientes formas:

$$T(n) = a \cdot T(n - c) + g(n),$$

$$T(n) = a \cdot T(n/b) + g(n).$$

La primera corresponde a algoritmos que tiene una parte no recursiva con coste $g(n)$ y hacen a llamadas recursivas con subproblemas de tamaño $n - c$, donde c es una constante.

La segunda corresponde a algoritmos que tienen una parte no recursiva con coste $g(n)$ y hacen a llamadas recursivas con subproblemas de tamaño (aproximadamente) n/b , donde $b > 1$.

Teorema

Sea $T(n)$ el coste (en caso peor, en caso medio, ...) de un algoritmo recursivo que satisface la recurrencia

$$T(n) = \begin{cases} f(n) & \text{si } 0 \leq n < n_0 \\ a \cdot T(n - c) + g(n) & \text{si } n \geq n_0, \end{cases}$$

donde n_0 es una constante, $c \geq 1$, $f(n)$ es una función arbitraria y $g(n) = \Theta(n^k)$ para una cierta constante $k \geq 0$.

Entonces

$$T(n) = \begin{cases} \Theta(n^k) & \text{si } a < 1 \\ \Theta(n^{k+1}) & \text{si } a = 1 \\ \Theta(a^{n/c}) & \text{si } a > 1. \end{cases}$$

Teorema

Sea $T(n)$ el coste (en caso peor, en caso medio, ...) de un algoritmo recursivo que satisface la recurrencia

$$T(n) = \begin{cases} f(n) & \text{si } 0 \leq n < n_0 \\ a \cdot T(n/b) + g(n) & \text{si } n \geq n_0, \end{cases}$$

donde n_0 es una constante, $b > 1$, $f(n)$ es una función arbitraria y $g(n) = \Theta(n^k)$ para una cierta constante $k \geq 0$.

Sea $\alpha = \log_b a$. Entonces

$$T(n) = \begin{cases} \Theta(n^k) & \text{si } \alpha < k \\ \Theta(n^k \log n) & \text{si } \alpha = k \\ \Theta(n^\alpha) & \text{si } \alpha > k. \end{cases}$$

Parte I

Repaso de Conceptos Algorítmicos

- Análisis de Algoritmos
- Divide y vencerás
- Grafos y Recorridos

Introducción

El principio básico de **divide y vencerás** (en inglés, *divide and conquer*; en catalán, *dividir per conquerir*) es muy simple:

- 1 Si el ejemplar (instancia) del problema a resolver es suficientemente simple, se encuentra la solución mediante algún método directo.
- 2 En caso contrario, se *divide* o *fragmenta* el ejemplar dado en subejemplares x_1, \dots, x_k y se resuelve, independiente y recursivamente, el problema para cada uno de ellos.
- 3 Las soluciones obtenidas y_1, \dots, y_k se *combinan* para obtener la solución al ejemplar original.

- El esquema de divide y vencerás expresado en pseudocódigo tiene el siguiente aspecto:

```
procedure DIVIDE_Y_VENCERAS( $x$ )
  if  $x$  es simple then
    return SOLUCION_DIRECTA( $x$ )
  else
     $\langle x_1, x_2, \dots, x_k \rangle := \text{DIVIDE}(x)$ 
    for  $i := 1$  to  $k$  do
       $y_i := \text{DIVIDE\_Y\_VENCERAS}(x_i)$ 
    end for
    return COMBINA( $y_1, y_2, \dots, y_k$ )
  end if
end procedure
```

- Ocasionalmente, si $k = 1$, se habla del esquema de *reducción*.

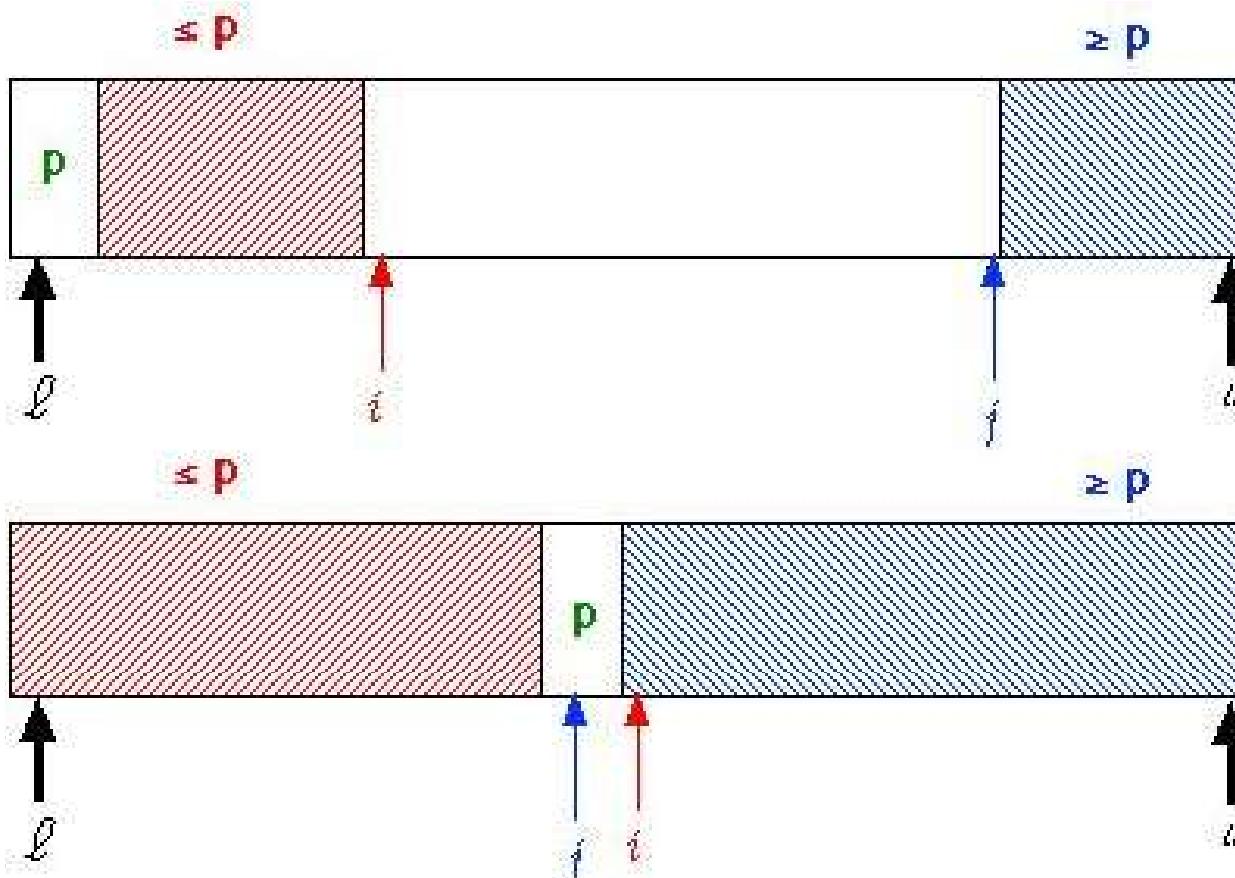
El esquema de divide y vencerás se caracteriza adicionalmente por las siguientes propiedades:

- No se resuelve más de una vez un mismo subproblema
- El tamaño de los subejemplares es, en promedio, una fracción del tamaño del ejemplar original, es decir, si x es de tamaño n , el tamaño esperado de un subejemplar cualquiera x_i es n/c_i donde $c_i > 1$. Con frecuencia, esta condición se cumplirá siempre, no sólo en promedio.

QuickSort

QUICKSORT (Hoare, 1962) es un algoritmo de ordenación que usa el principio de divide y vencerás, pero a diferencia de los ejemplos anteriores, no garantiza que cada subejemplar tendrá un tamaño que es fracción del tamaño original.

La base de *quicksort* es el procedimiento de PARTICIÓN: dado un elemento p denominado **pivote**, debe reorganizarse el segmento como ilustra la figura.



El procedimiento de partición sitúa a un elemento, el pivote, en su lugar apropiado. Luego no queda más que ordenar los segmentos que quedan a su izquierda y a su derecha. Mientras que en MERGESORT la división es simple y el trabajo se realiza durante la fase de combinación, en Quicksort sucede lo contrario.

Para ordenar el segmento $A[\ell..u]$ el algoritmo queda así

```
procedure QUICKSORT( $A, \ell, u$ )
Ensure: Ordena el subvector  $A[\ell..u]$ 
if  $u - \ell + 1 \leq M$  then
    usar un algoritmo de ordenación simple
else
    PARTICION( $A, \ell, u, k$ )
    ▷  $A[\ell..k - 1] \leq A[k] \leq A[k + 1..u]$ 
    QUICKSORT( $(A, \ell, k - 1)$ )
    QUICKSORT( $(A, k + 1, u)$ )
end if
end procedure
```

En vez de usar un algoritmo de ordenación simple (p.e. ordenación por inserción) con cada segmento de M o menos elementos, puede ordenarse mediante el algoritmo de inserción al final:

`QUICKSORT(A , 1, $A.\text{SIZE}()$)`

`INSERTSORT(A , 1, $A.\text{SIZE}()$)`

Puesto que el vector A está quasi-ordenado tras aplicar `QUICKSORT`, el último paso se hace en tiempo $\Theta(n)$, donde $n = A.\text{SIZE}()$.

Se estima que la elección óptima para el umbral o corte de recursión M oscila entre 20 y 25.

Existen muchas formas posibles de realizar la partición. En Bentley & McIlroy (1993) se discute un procedimiento de partición muy eficiente, incluso si hay elementos repetidos. Aquí examinamos un algoritmo básico, pero razonablemente eficaz.

Se mantienen dos índices i y j de tal modo que $A[\ell + 1..i - 1]$ contiene elementos menores o iguales que el pivote p , y $A[j + 1..u]$ contiene elementos mayores o iguales. Los índices barren el segmento (de izquierda a derecha, y de derecha a izquierda, respectivamente), hasta que $A[i] > p$ y $A[j] < p$ o se cruzan ($i = j + 1$).

```

procedure PARTICION( $A, \ell, u, k$ )
Require:  $\ell \leq u$ 
Ensure:  $A[\ell..k - 1] \leq A[k] \leq A[k + 1..u]$ 
     $i := \ell + 1; j := u; p := A[\ell]$ 
    while  $i < j + 1$  do
        while  $i < j + 1 \wedge A[i] \leq p$  do
             $i := i + 1$ 
        end while
        while  $i < j + 1 \wedge A[j] \geq p$  do
             $j := j - 1$ 
        end while
        if  $i < j + 1$  then
             $A[i] := A[j]$ 
        end if
    end while
     $A[\ell] := A[j]; k := j$ 
end procedure

```

El coste de Quicksort en caso peor es $\Theta(n^2)$ y por lo tanto poco atractivo en términos prácticos. Esto ocurre si en todos o la gran mayoría de los casos uno de los subsegmentos contiene muy pocos elementos y el otro casi todos, p.e. así sucede si el vector está ordenado creciente o decrecientemente. El coste de la partición es $\Theta(n)$ y entonces tenemos

$$\begin{aligned}
 Q(n) &= \Theta(n) + Q(n - 1) + Q(0) \\
 &= \Theta(n) + Q(n - 1) = \Theta(n) + \Theta(n - 1) + Q(n - 2) \\
 &= \dots = \sum_{i=0}^n \Theta(i) = \Theta\left(\sum_{0 \leq i \leq n} i\right) \\
 &= \Theta(n^2).
 \end{aligned}$$

Sin embargo, en promedio, el pivote quedará más o menos centrado hacia la mitad del segmento como sería deseable —justificando que *quicksort* sea considerado un algoritmo de divide y vencerás.

Para analizar el comportamiento de QUICKSORT sólo importa el orden relativo de los elementos. También podemos investigar exclusivamente el número de comparaciones entre elementos, ya que el coste total es proporcional a dicho número.

Supongamos que cualquiera de los $n!$ ordenes relativos posibles tiene idéntica probabilidad, y sea q_n el número medio de comparaciones.

$$\begin{aligned}
 q_n &= \sum_{1 \leq j \leq n} \mathbb{E}[\# \text{ compar.} \mid \text{pivote es } j\text{-ésimo}] \times \Pr\{\text{pivote es } j\text{-ésimo}\} \\
 &= \sum_{1 \leq j \leq n} (n - 1 + q_{j-1} + q_{n-j}) \times \frac{1}{n} \\
 &= n + \mathcal{O}(1) + \frac{1}{n} \sum_{1 \leq j \leq n} (q_{j-1} + q_{n-j}) \\
 &= n + \mathcal{O}(1) + \frac{2}{n} \sum_{0 \leq j < n} q_j
 \end{aligned}$$

Para resolver esta recurrencia emplearemos el denominado *continuous master theorem* (CMT).

El CMT considera recurrencias de divide y vencerás con el siguiente formato

$$F_n = t_n + \sum_{0 \leq j < n} \omega_{n,j} F_j, \quad n \geq n_0$$

para un entero positivo n_0 , una función t_n , denominada *función de peaje*, y unos pesos $\omega_{n,j} \geq 0$. Los pesos deben satisfacer dos condiciones adicionales:

- 1 $W_n = \sum_{0 \leq j < n} \omega_{n,j} \geq 1$
- 2 $Z_n = \sum_{0 \leq j < n} \frac{j}{n} \cdot \frac{\omega_{n,j}}{W_n} < 1.$

El paso fundamental es hallar una *función de forma* $\omega(z)$ que aproxima los pesos $\omega_{n,j}$.

Definición

Dado un conjunto de pesos $\omega_{n,j}$, $\omega(z)$ es una función de forma para el conjunto de pesos si

- 1 $\int_0^1 \omega(z) dz \geq 1$
- 2 existe una constante $\rho > 0$ tal que

$$\sum_{0 \leq j < n} \left| \omega_{n,j} - \int_{j/n}^{(j+1)/n} \omega(z) dz \right| = \mathcal{O}(n^{-\rho})$$

Un método simple y que funciona usualmente para calcular funciones de forma consiste en sustituir j por $z \cdot n$ en $\omega_{n,j}$, multiplicar por n y tomar el límite para $n \rightarrow \infty$.

$$\omega(z) = \lim_{n \rightarrow \infty} n \cdot \omega_{n,z \cdot n}$$

Las extensiones a los números reales de muchas funciones discretas son inmediatas, p.e. $j^2 \rightarrow z^2$.

Para los números binomiales se puede emplear la aproximación

$$\binom{z \cdot n}{k} \sim \frac{(z \cdot n)^k}{k!}.$$

La extensión de los factoriales a los reales viene dada por la función gamma de Euler $\Gamma(z)$ y la de los números armónicos es la función $\Psi(z) = \frac{d \ln \Gamma(z)}{dz}$.

Por ejemplo, en quicksort los pesos son todos iguales:

$\omega_{n,j} = \frac{2}{n}$. La función de forma correspondiente es $\omega(z) = \lim_{n \rightarrow \infty} n \cdot \omega_{n,z \cdot n} = 2$.

Teorema (Roura, 1997)

Sea F_n descrita por la recurrencia

$$F_n = t_n + \sum_{0 \leq j < n} \omega_{n,j} F_j,$$

sea $\omega(z)$ una función de forma correspondiente a los pesos $\omega_{n,j}$, y $t_n = \Theta(n^a(\log n)^b)$, para $a \geq 0$ y $b > -1$ constantes.

Sea $\mathcal{H} = 1 - \int_0^1 \omega(z)z^a dz$ y $\mathcal{H}' = -(b+1) \int_0^1 \omega(z)z^a \ln z dz$. Entonces

$$F_n = \begin{cases} \frac{t_n}{\mathcal{H}} + o(t_n) & \text{si } \mathcal{H} > 0, \\ \frac{t_n}{\mathcal{H}'} \ln n + o(t_n \log n) & \text{si } \mathcal{H} = 0 \text{ y } \mathcal{H}' \neq 0, \\ \Theta(n^\alpha) & \text{si } \mathcal{H} < 0, \end{cases}$$

donde $x = \alpha$ es la única solución real no negativa de la ecuación

$$1 - \int_0^1 \omega(z)z^x dz = 0.$$

Consideremos q_n . Ya hemos visto que los pesos son $\omega_{n,j} = 2/n$ y $t_n = n - 1$. Por tanto $\omega(z) = 2$, $a = 1$ y $b = 0$. Puede comprobarse fácilmente que se dan todas las condiciones necesarias para la aplicación del CMT. Calculamos

$$\mathcal{H} = 1 - \int_0^1 2z \, dz = 1 - z^2 \Big|_{z=0}^{z=1} = 0,$$

por lo que tendremos que aplicar el caso 2 del CMT y calcular \mathcal{H}'

$$\mathcal{H}' = - \int_0^1 2z \ln z \, dz = \frac{z^2}{2} - z^2 \ln z \Big|_{z=0}^{z=1} = \frac{1}{2}.$$

Por lo tanto,

$$\begin{aligned} q_n &= \frac{n \ln n}{1/2} + o(n \log n) = 2n \ln n + o(n \log n) \\ &= 1,386 \dots n \log_2 n + o(n \log n). \end{aligned}$$

QuickSelect

El problema de la selección consiste en hallar el j -ésimo de entre n elementos dados. En concreto, dado un vector A con n elementos y un rango j , $1 \leq j \leq n$, un algoritmo de selección debe hallar el j -ésimo elemento en orden ascendente. Si $j = 1$ entonces hay que encontrar el mínimo, si $j = n$ entonces hay que hallar el máximo, si $j = \lfloor n/2 \rfloor$ entonces debemos hallar la mediana, etc.

Es fácil resolver el problema con coste $\Theta(n \log n)$ ordenando previamente el vector y con coste $\Theta(j \cdot n)$, recorriendo el vector y manteniendo los j elementos menores de entre los ya examinados. Con las estructuras de datos apropiadas puede rebajarse el coste a $\Theta(n \log j)$, lo cual no supone una mejora sobre la primera alternativa si $j = \Theta(n)$.

QUICKSELECT (Hoare, 1962), también llamado FIND y *one-sided* QUICKSORT, es una variante del algoritmo QUICKSORT para la selección del j -ésimo de entre n elementos.

Supongamos que efectuamos una partición de un subvector $A[\ell..u]$, conteniendo los elementos ℓ -ésimo a u -ésimo de A , y tal que $\ell \leq j \leq u$, respecto a un pivote p . Una vez finalizada la partición, supongamos que el pivote acaba en la posición k . Por tanto, en $A[\ell..k - 1]$ están los elementos ℓ -ésimo a $(k - 1)$ -ésimo de A y en $A[k + 1..u]$ están los elementos $(k + 1)$ -ésimo a u -ésimo. Si $j = k$ hemos acabado ya que hemos encontrado el elemento solicitado. Si $j < k$ entonces procedemos recursivamente en el subvector de la izquierda $A[\ell..k - 1]$ y si $j > k$ entonces encontraremos el elemento buscado en el subvector $A[k + 1..u]$.

procedure QUICKSELECT(A, ℓ, j, u)

Ensure: Retorna el $(j + 1 - \ell)$ -ésimo menor elemento de $A[\ell..u]$, $\ell \leq j \leq u$

if $\ell = u$ **then**

return $A[\ell]$

end if

PARTICION(A, ℓ, u, k)

if $j = k$ **then**

return $A[k]$

end if

if $j < k$ **then**

return QUICKSELECT($A, \ell, j, k - 1$)

else

return QUICKSELECT($A, k + 1, j, u$)

end if

end procedure

Puesto que QUICKSELECT es recursiva final es muy simple obtener una versión iterativa eficiente que no necesita espacio auxiliar.

En caso peor, el coste de QUICKSELECT es $\Theta(n^2)$. Sin embargo, su coste promedio es $\Theta(n)$ donde la constante de proporcionalidad depende del cociente j/n . Knuth (1971) ha demostrado que $C_n^{(j)}$, el número medio de comparaciones necesarias para seleccionar el j -ésimo de entre n es:

$$\begin{aligned} C_n^{(j)} = & 2((n+1)H_n - (n+3-j)H_{n+1-j} \\ & - (j+2)H_j + n+3) \end{aligned}$$

El valor máximo se alcanza para $j = \lfloor n/2 \rfloor$; entonces $C_n^{(j)} = 2(\ln 2 + 1)n + o(n)$.

Consideremos ahora el análisis del coste promedio C_n , suponiendo que j adopta cualquier valor entre 1 y n con igual probabilidad.

$$C_n = n + \mathcal{O}(1) + \frac{1}{n} \sum_{1 \leq k \leq n} \mathbb{E}[\text{núm. de comp.} \mid \text{el pivote es el } k\text{-ésimo}] ,$$

puesto que el pivote es el k -ésimo con igual probabilidad para toda k .

La probabilidad de que $j = k$ es $1/n$ y entonces ya habremos acabado. La probabilidad de continuar a la izquierda es $(k - 1)/n$ y entonces se harán de hacer C_{k-1} comparaciones. Análogamente, con probabilidad $(n - k)/n$ se continuará en el subvector a la derecha y se harán C_{n-k} comparaciones.

$$\begin{aligned} C_n &= n + \mathcal{O}(1) + \frac{1}{n} \sum_{1 \leq k \leq n} \frac{k-1}{n} C_{k-1} + \frac{n-k}{n} C_{n-k} \\ &= n + \mathcal{O}(1) + \frac{2}{n} \sum_{0 \leq k < n} \frac{k}{n} C_k. \end{aligned}$$

Aplicando el CMT con la función de forma

$$\lim_{n \rightarrow \infty} n \cdot \frac{2}{n} \frac{z \cdot n}{n} = 2z$$

obtenemos $\mathcal{H} = 1 - \int_0^1 2z^2 dz = 1/3$ y $C_n = 3n + o(n)$.

Algoritmo de selección de Rivest y Floyd



Robert W. Floyd (1936–2001) Ronald L. Rivest (1947–)

En 1970 Floyd y Rivest diseñaron un algoritmo de selección con coste lineal en caso peor; solo resulta necesario garantizar que el pivote elegido en cada paso de quickselect divide el vector en dos subvectores cada uno de los cuales contiene una fracción del tamaño original n . Dicho pivote debe obtenerse en tiempo lineal. Entonces, en caso peor, el coste del algoritmo es

$$C(n) = \mathcal{O}(n) + C(p \cdot n),$$

para algún valor $p < 1$. Puesto $\log_{1/p} 1 = 0 < 1$ concluimos que $C(n) = \mathcal{O}(n)$. Por otro lado, obviamente $C(n) = \Omega(n)$, y por lo tanto $C(n) = \Theta(n)$.

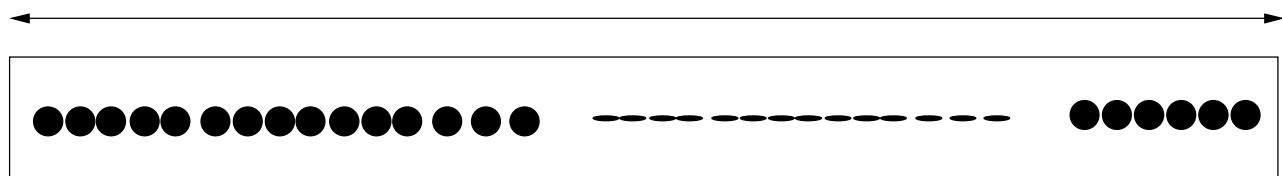
La única diferencia entre el algoritmo QUICKSELECT de Hoare y el algoritmo de Floyd y Rivest reside en la elección del pivote en cada fase recursiva.

¡Y el algoritmo de Floyd y Rivest obtiene un “buen” pivote usando el propio algoritmo recursivamente!

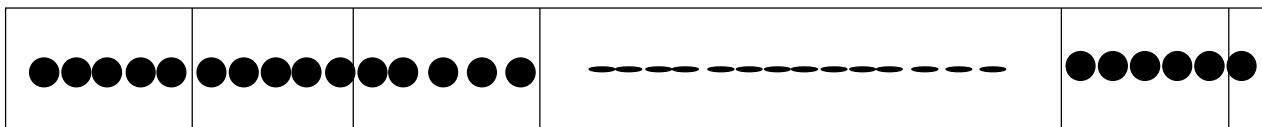
En concreto lo que hace el algoritmo es elegir la denominada **pseudo-mediana** del vector como pivote.

- 1 El subvector $A[\ell..u]$ de tamaño $n = u - \ell + 1$ se subdivide en bloques de q elementos (excepto posiblemente el último bloque, que puede contener $< q$ elementos), para alguna constante impar q . Para cada bloque obtenemos la mediana de sus q elementos.
- 2 Aplicamos el algoritmo recursivamente sobre las $\lceil n/q \rceil$ medianas obtenidas en el paso previo para hallar la mediana de las medianas. Este elemento no es, en general, la mediana del subvector original, por eso se le denomina *pseudo-mediana*.
- 3 La pseudo-mediana se utiliza como pivote para particionar el subvector original, y se hace una llamada recursiva en el subvector apropiado, a la izquierda o a la derecha del pivote, según el rango buscado—exactamente como en quickselect.

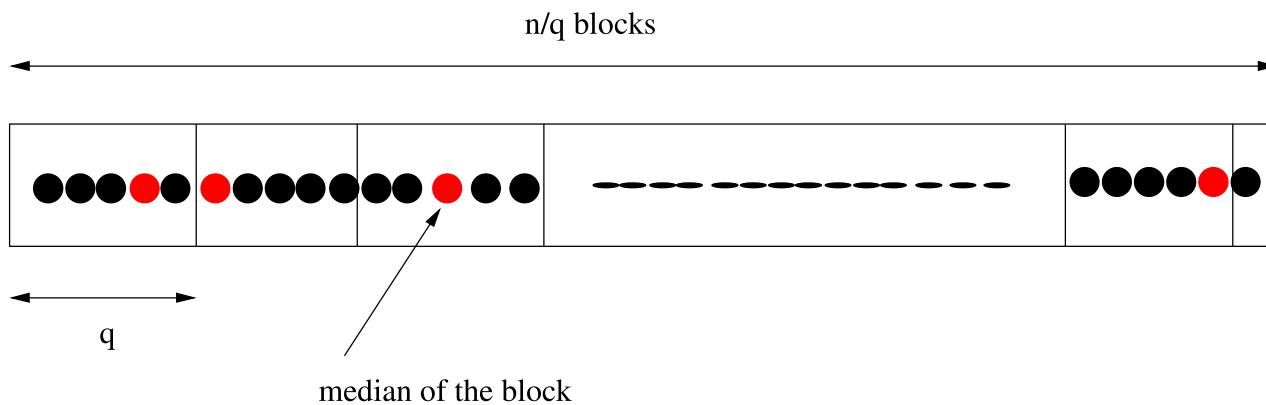
n elements

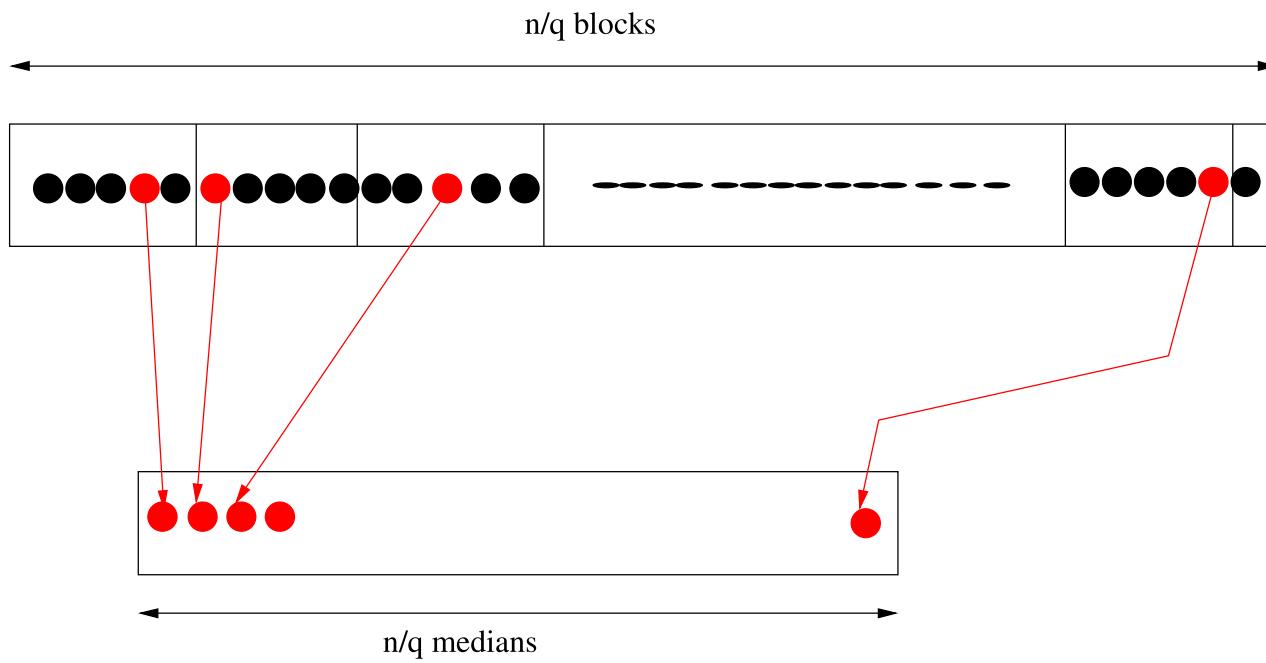


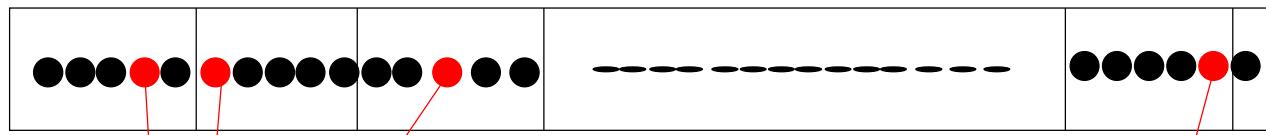
n/q blocks



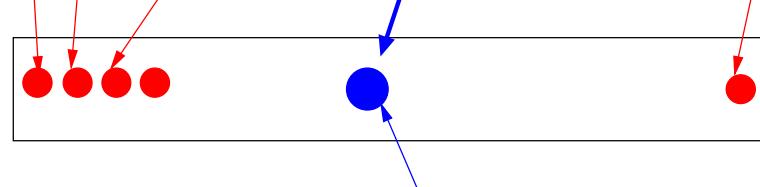
q







find the median RECURSIVELY



median of the n/q medians: $\sim n/2q$ are smaller, $\sim n/2q$ are larger

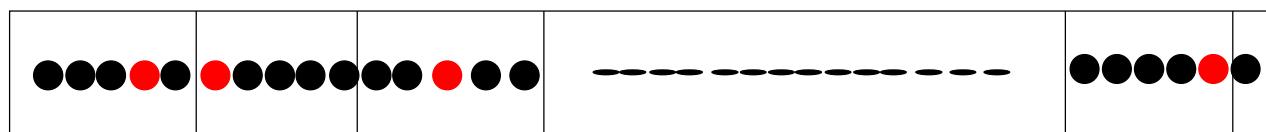


partition the array using

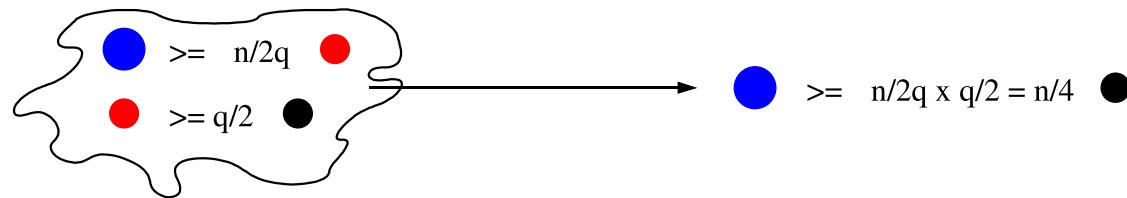


$$\text{blue circle} \geq n/2q \quad \text{red circle}$$

$$\text{red circle} \geq q/2 \quad \text{black circle}$$



partition the array using



- 1 El coste de la primera fase (calcular las medianas de los bloques) es $\Theta(n)$, ya que hallar la mediana de cada bloque es $\Theta(q) = \Theta(1)$ y hay $\lceil n/q \rceil$ bloques.
- 2 La llamada recursiva para encontrar la mediana de las medianas: $C(n/q)$, la entrada consiste en $\lceil n/q \rceil$ medianas.
- 3 Está garantizado que hay $\approx n/2q \cdot q/2 = n/4$ elementos menores que el pivote; análogamente, hay $\approx n/4$ elementos mayores que el pivote. En caso peor, después de la partición (coste $\Theta(n)$) proseguiremos en un subvector con $\leq 3n/4$ elementos.

Poniendo todo junto, el coste en caso peor satisface

$$C(n) = \mathcal{O}(n) + C(n/q) + C(3n/4),$$

donde el término $\mathcal{O}(n)$ incluye los costes de calcular las medianas de los bloques y de particionar el vector original. La solución de la recurrencia (se puede hacer con un *Discrete Master Theorem* no explicado en ALG) es $C(n) = \mathcal{O}(n)$ si

$$\frac{3}{4} + \frac{1}{q} < 1.$$

Por ejemplo, $q = 3$ no servirá, pero $q = 5$, el valor originalmente propuesto por Floyd y Rivest, sí.

Parte I

Repaso de Conceptos Algorítmicos

- Análisis de Algoritmos
- Divide y vencerás
- Grafos y Recorridos

Definición

Un **grafo (no dirigido)** es un par $G = \langle V, E \rangle$ donde V es un conjunto finito de **vértices** (también llamados **nodos**) y E es un conjunto de **aristas**; cada arista $e \in E$ es un par no ordenado $\{u, v\}$ donde u y v ($u \neq v$) son elementos de V .

Definición

Un **grafo dirigido o digrafo** es un par $G = \langle V, E \rangle$ donde V es un conjunto finito de **vértices** o **nodos** y E es un conjunto de **arcos**; cada arco $e \in E$ es un par (u, v) donde u y v ($u \neq v$) son elementos de V .

Si en vez de un conjunto de arcos o aristas tenemos multiconjuntos de arcos o aristas, y se permiten **bucles** (esto es, arcos de la forma (u, u) o aristas $\{u, u\}$) entonces tenemos **multigrafos** (dirigidos y no dirigidos, respectivamente).

Dado un arco $e = (u, v)$ se denomina **origen** a u y **destino** a v . Se dice que u es un **predecesor** de v y que v es un **sucesor** de u . Para una arista $e = \{u, v\}$ se dice que u y v son sus **extremos**, que la arista es **incidente** a u y v , que u y v son **adyacentes**.

Definición

Un **camino** $P = v_0, v_1, v_2, \dots, v_n$ de longitud n en un grafo $G = \langle V, E \rangle$ es una secuencia de vértices de G tal que para toda i , $0 \leq i < n$

$$\{v_i, v_{i+1}\} \in E$$

El vértice v_0 es el origen del camino P y v_n es su destino.

Un camino en un digrafo se define de manera análoga: para cada i , (v_i, v_{i+1}) es un arco del digrafo.

Se dice que un camino es **simple** si no se repite ningún vértice.

Un camino en el que $v_0 = v_n$ es un **ciclo**.

Definición

Un grafo $G = \langle V, E \rangle$ es **conexo** si y sólo si, para todo par de vértices u y v , existe un camino que va de u a v en G .

Para digrafos la definición es idéntica, sólo que entonces se dice que el digrafo es **fuertemente conexo**.

Definición

Dado un grafo $G = \langle V, E \rangle$, el grafo $H = \langle V', E' \rangle$ se dice que es un **subgrafo** de G si y sólo si $V' \subseteq V$, $E' \subseteq E$, y para toda arista $e' = (u', v') \in E'$ se cumple que u' y v' pertenecen a V' .

Si E' contiene todas las aristas de E que son incidentes a dos vértices de V' , se dice que H es el subgrafo **inducido** por V' .

La definición de subgrafo de un grafo dirigido es completamente análoga.

Definición

Una **componente conexa** C de un grafo G es un subgrafo inducido maximal conexo de G . Maximal quiere decir que si se añade cualquier vértice v a $V(C)$ entonces el subgrafo inducido correspondiente no es conexo—en particular, no existe camino entre v y los restantes vértices de C .

En el caso de digrafos, se habla de **componentes fuertemente conexas**: una componente fuertemente conexa del digrafo G es un subgrafo dirigido inducido maximal fuertemente conexo.

Definición

Un grafo conexo y sin ciclos se denomina **árbol (libre)**.

Si G es un grafo conexo, un subgrafo $T = \langle V, E' \rangle$ (es decir que contiene los mismos vértices que G) es un **árbol de expansión** si T es un árbol.

Los árboles libres no tienen raíz, y no existe orden entre los vértices adyacentes a un vértice dado del árbol.

Lema

Si $G = \langle V, E \rangle$ es un árbol entonces $|E| = |V| - 1$.

A menudo trabajaremos con grafos o digrafos con **etiquetas** en las aristas o arcos: el etiquetado de un grafo o digrafo es una función $\phi : E \rightarrow \mathcal{L}$ entre el conjunto de aristas o arcos de G y el conjunto (eventualmente infinito) de etiquetas \mathcal{L} .

Cuando las etiquetas son números (enteros, racionales, reales), se dice que el grafo o digrafo es **ponderado** y a la etiqueta $\phi(e)$ de una arista o arco e se le suele denominar **peso**.

Implementación

Los grafos se implementan habitualmente de una de las dos siguientes formas:

- 1 Mediante **matrices de adyacencia**: la componente $A[i, j]$ de la matriz nos dice si el arco entre los vértices i y j existe o no; eventualmente, $A[i, j]$ puede contener también el peso del arco entre los vértices i y j
- 2 Mediante **listas de adyacencia**: tenemos una tabla T de listas enlazadas; la lista $T[i]$ es la lista de sucesores del vértice i

La implementación mediante matrices de adyacencia es muy costosa en espacio: si $|V| = n$, se requiere espacio $\Theta(n^2)$ para representar el grafo, independientemente del número de aristas/arcos que haya en el grafo. Sólo es interesante si necesitamos poder decidir la existencia (o no) de una arista o arco con máxima eficiencia.

Por regla general, la implementación que se usará es la de listas de adyacencia. El espacio que se requiere es $\Theta(n + m)$, donde $m = |E|$ y $n = |V|$; el espacio utilizado es por lo tanto lineal respecto al tamaño del grafo.

```
typedef int vertex;
typedef pair<vertex, vertex> edge;
typedef list<edge>::iterator edge_iter;

class Graph {
// grafos no dirigidos con V = {0, ..., n-1}
public:
// crea un grafo vacío (sin vértices y sin aristas)
    Graph();
// crea un grafo con n vértices y sin aristas
    Graph(int n);
// añade un vértice al grafo; el nuevo vértice tendrá
// identificador n, donde n era el número de vértices
// del grafo antes de añadir el vértice
void add_vertex();
// añade la arista (u,v) o e = (u,v) al grafo
void add_edge(vertex u, vertex v);
void add_edge(edge e);

// consultoras del numero de vertices y de aristas
int nr_vertices() const;
int nr_edges() const;

// devuelve lista de adyacentes a un vertice
list<edge> adjacent(vertex u) const;
...
private:
    int n, m;
    vector<list<edge> > T;
    ...
}
```

Recorridos

Definición

Dado un grafo conexo $G = \langle V, E \rangle$, un **recorrido** del grafo es una secuencia que contiene todos y cada uno de los vértices en $V(G)$ exactamente una vez y tal que para cualquier vértice v en la secuencia se cumple que, o bien v es el primer vértice de la secuencia, o bien existe una arista $e \in E(G)$ que une al vértice v con otro vértice que le precede en la secuencia.

El recorrido de un grafo es una secuencia de los recorridos de las componentes conexas de G . No hay ningún vértice que aparezca en el recorrido sin que se haya completado el recorrido de las componentes conexas correspondientes a los vértices que le preceden en el recorrido.

En el caso de los digrafos, un recorrido que comienza en un cierto vértice v visitará todos y cada uno de los vértices accesibles desde v exactamente una vez; la visita de un vértice $w \neq v$ implica que existe algún otro vértice visitado previamente del cual w es sucesor.

Los recorridos nos permiten visitar todos los vértices de un grafo, con arreglo a la topología del grafo, pues se utilizan las aristas o arcos del grafo para ir haciendo las sucesivas visitas.

Mediante un recorrido (o una combinación de recorridos) podemos resolver eficientemente numerosos problemas sobre grafos. Por ejemplo

- Determinar si un grafo es conexo o no
- Hallar las componentes conexas de un grafo no dirigido
- Hallar las componentes fuertemente conexas de un grafo dirigido
- Determinar si un grafo contiene ciclos o no
- Decidir si un grafo es 2-coloreable, equivalentemente, si es bipartido
- Decidir si un grafo es biconexo o no (la eliminación de un vértice no desconecta al grafo)
- Hallar el camino más corto (con menor número de aristas/arcos) entre dos vértices dados
- etc.

Recorrido en profundidad (DFS)

En el **recorrido en profundidad** (ing: *Depth-First Search*, DFS) de un grafo G , se visita un vértice v y desde éste, recursivamente, se realiza el recorrido de cada uno de los vértices adyacentes/sucesores de v no visitados.

Cuando se visita por vez primera un vértice v se dice que se ha **abierto** el vértice; el vértice se **cierra** cuando se completa, recursivamente, el recorrido en profundidad de los vértices adyacentes/sucesores.

La numeración **directa** o **numeración DFS** de un vértice es el número de orden en el que se abre el vértice; así, el vértice en el que se inicia el recorrido tiene numeración directa 1.

La numeración **inversa** de un vértice es el número de orden en el que se cierra el vértice. El primer vértice del recorrido para el cual todos sus vecinos/sucesores ya han sido visitados tiene numeración inversa 1, y así sucesivamente.

▷ *visitado*, *ndfs*, *ninv*, *num_dfs*, *num_inv* son variables globales

```
procedure DFS( $G$ )
    for  $v \in V(G)$  do
        visitado[ $v$ ] := false
        ndfs[ $v$ ] := 0; ninv[ $v$ ] := 0
    end for
    num_dfs := 0; num_inv := 0
    for  $v \in V(G)$  do
        if  $\neg$ visitado[ $v$ ] then
            DFS-REC( $G, v, v$ )
        end if
    end for
end procedure
```

```

procedure DFS-REC( $G$ ,  $v$ ,  $padre$ )
    PRE-VISIT( $v$ )
     $visitado[v] := \text{true}$ 
     $num\_dfs := num\_dfs + 1$ ;  $ndfs[v] := num\_dfs$ 
    for  $w \in G.\text{ADJACENT}(v)$  do
        if  $\neg visitado[w]$  then
            PRE-VISIT-EDGE( $v, w$ )
            DFS-REC( $G, w, v$ )
            POST-VISIT-EDGE( $v, w$ )
        else
            ▷ si  $w \neq padre$  entonces hemos encontrado un ciclo
            end if
        end for
        POST-VISIT( $v$ )
         $num\_inv := num\_inv + 1$ ;  $ninv[v] := num\_inv$ 
end procedure

```

```
// Macros para escribir recorridos en un grafo
#define forall(v,G) for(vertex (v) = 0; (v) < (G).nr_vertices(); ++(v))

#define forall_adj(e, u, G) \
    for(edge_iter (e) = (G).adjacent((u)).begin(); \
        (e) != (G).adjacent((u))].end() ; ++(e))

#define target(eit) ((eit) -> second)
#define source(eit) ((eit) -> first)
```

```
void DFS(const Graph& G) {
    vector<bool> visitado(G.nr_vertices(), false);
    vector<int> ndfs(G.nr_vertices(), 0);
    vector<int> ninv(G.nr_vertices(), 0);
    int num_dfs = 0;
    int num_inv = 0;

    forall(v, G)
        if (not visitado[v])
            DFS(G, v, v, visitado, num_dfs, num_inv, ndfs, ninv);
}
```

```

void DFS(const Graph& G, vertex v, vertex padre,
          vector<bool>& visitado,
          int& num_dfs, int& num_inv,
          vector<int>& ndfs,
          vector<int>& ninv) {

    PRE-VISIT(v);
    visitado[v] = true;
    ++num_dfs; ndfs[v] = num_dfs;
    forall_adj(e, v, G) {
        vertex w = target(e);
        if (not visitado[w]) {
            PRE-VISIT-EDGE(v,w);
            DFS(G, w, v, visitado, num_dfs, num_inv, ndfs, ninv);
            POST-VISIT-EDGE(v,w);
        } else {
            // si w != padre entonces hemos encontrado un ciclo
        }
    }
    POST-VISIT(v);
    ++num_inv; ninv[v] = num_inv;
}

```

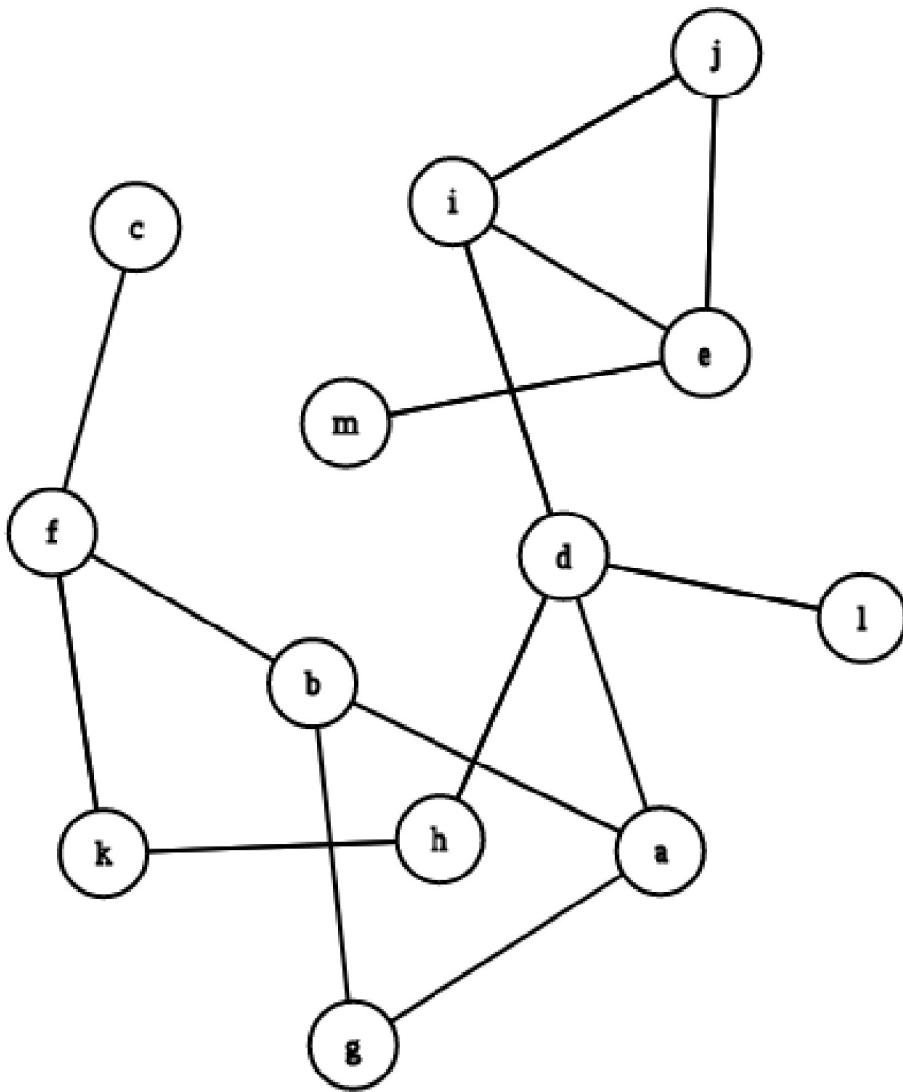
El recorrido en profundidad de una componente conexa induce un árbol de expansión, al que se denomina árbol del recorrido T_{DFS} .

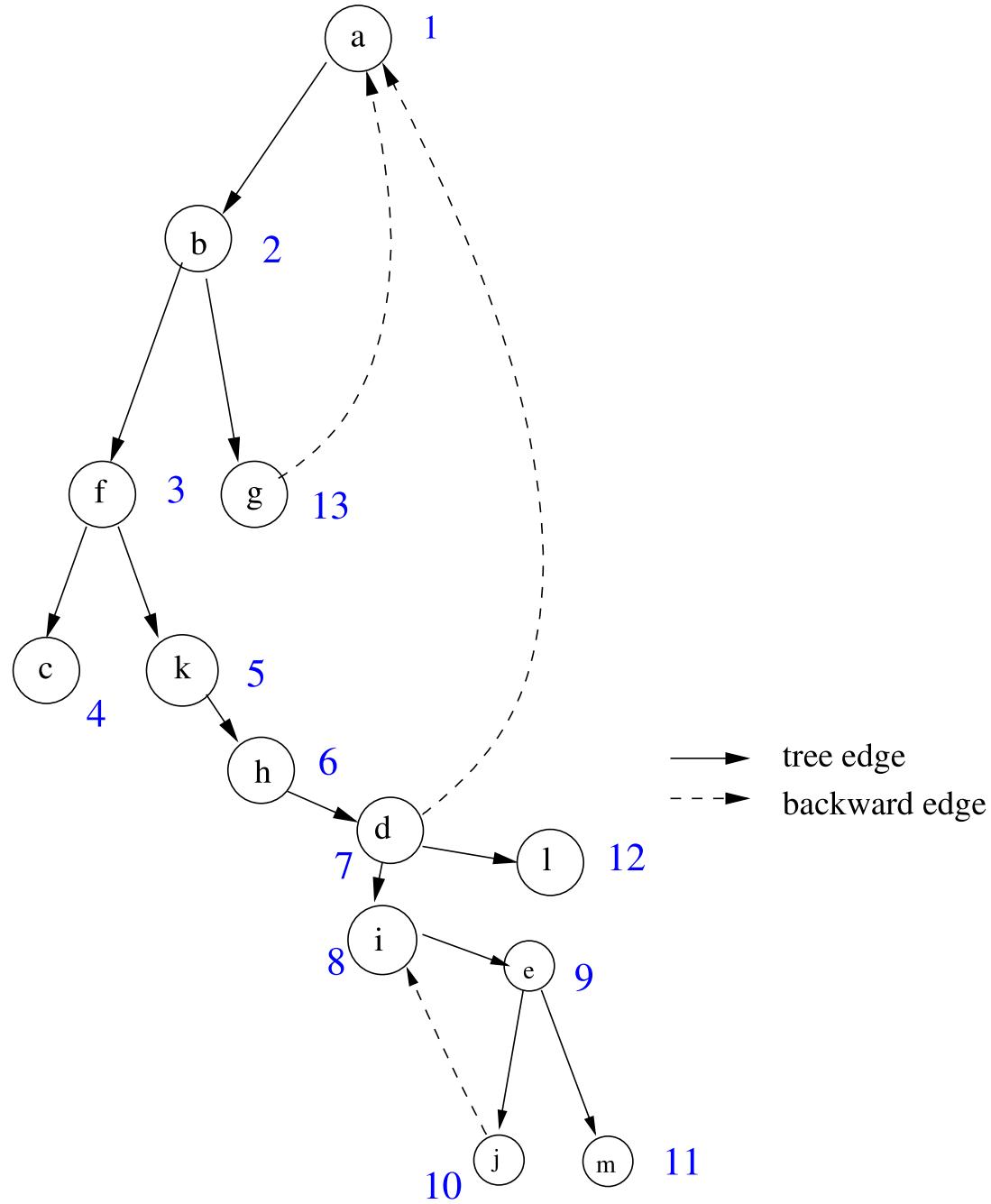
Las aristas de la componente conexa se clasifican entonces en **aristas de árbol** (*tree edges*) y **aristas de retroceso** (*backward edges*). Éstas últimas unen el vértice que se acaba de abrir con un vértice visitado (abierto o cerrado) y por tanto cierran ciclos. El recorrido de todo el grafo da lugar a un bosque de expansión.

Para cada vértice v , $CC[v]$ será el número de la componente conexa en la que está v . $TreeEdges[i]$ es el conjunto de aristas de árbol en la componente i , y $BackEdges[i]$ el conjunto de aristas de retroceso. El número de componentes conexas viene dado por ncc .

```
procedure DFS( $G$ )
    for  $v \in V(G)$  do
         $visitado[v] := \text{false}$ 
         $CC[v] := 0$ 
    end for
     $ncc := 0$ 
    for  $v \in V(G)$  do
        if  $\neg visitado[v]$  then
             $ncc := ncc + 1$ 
             $TreeEdges[ncc] := \emptyset$ 
             $BackEdges[ncc] := \emptyset$ 
            DFS-REC( $G, v, v$ )
        end if
    end for
end procedure
```

```
procedure DFS-REC( $G, v, padre$ )
     $visitado[v] := \text{true}$ 
     $CC[v] := ncc$ 
    for  $w \in G.\text{ADJACENT}(v)$  do
        if  $\neg visitado[w]$  then
             $TreeEdges[ncc] := TreeEdges[ncc] \cup \{(v, w)\}$ 
            DFS-REC( $G, w, v$ )
        else if  $w \neq padre$  then
             $BackEdges[ncc] := BackEdges[ncc] \cup \{(v, w)\}$ 
        end if
    end for
end procedure
```

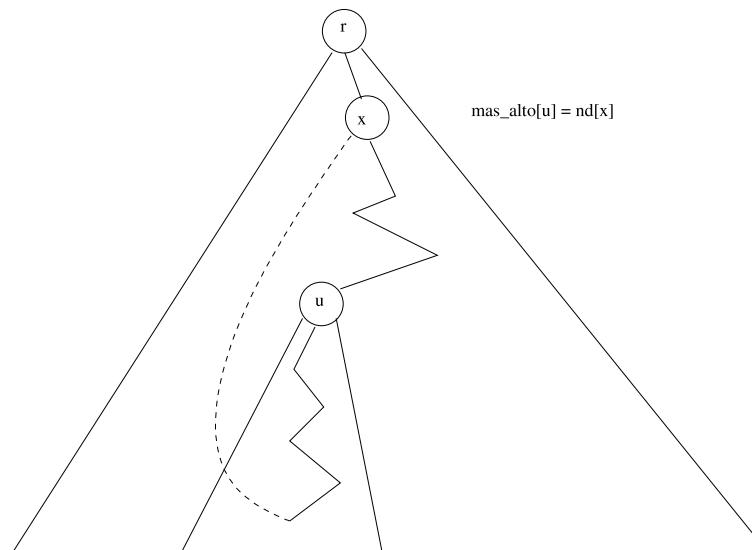




Aplicaciones del DFS: Grafos Biconexos

Un grafo conexo G se dice que es **biconexo** si al eliminar uno cualquiera de sus vértices sigue siendo conexo.

Supongamos que efectuamos un DFS empezando en el vértice r y que para cada vértice u determinamos el número DFS más bajo que es alcanzable siguiendo un camino desde u hasta uno de sus descendientes en el T_{DFS} y a continuación “subimos” con una arista de retroceso. Denominaremos $\text{mas_alto}[u]$ a dicho número.



Un vértice u es un **punto de articulación** de G si su eliminación desconecta el grafo. Si G no tiene puntos de articulación entonces G es biconexo. Para ver si un vértice es punto de articulación o no deberemos considerar varios casos:

- 1 u es una hoja del T_{DFS} (no tiene descendientes): entonces no es un punto de articulación ya que nunca se desconectará el grafo
- 2 $u = r$ es la raíz del T_{DFS} : es punto de articulación si y sólo si tiene más de un descendiente en el T_{DFS} , su eliminación desconectaría los subárboles, pero si sólo hay uno su eliminación no desconecta a los demás vértices.

- 3 u no es hoja y no es la raíz: entonces es punto de articulación si y sólo si alguno de los vértices adyacentes descendientes w de u cumple $\text{mas_alto}[w] \geq \text{ndfs}[u]$, es decir, si algún descendiente no puede “escapar” del subárbol enraizado en u sin pasar por u .

Por otro lado para cada vértice u , $\text{mas_alto}[u]$ es el mínimo entre: 1) $\text{ndfs}[u]$, 2) $\text{mas_alto}[v]$ para todo v descendiente directo de u en el T_{DFS} y 3) $\text{ndfs}[v]$ para todo v adyacente a u mediante una arista de retroceso.

```
procedure BICONEXO( $G$ )
```

Require: G es conexo

```
for  $v \in V(G)$  do
```

```
    visitado[ $v$ ] := false
```

```
    ndfs[ $v$ ] := 0
```

```
    punto_art[ $v$ ] := false
```

```
    num_desc[ $v$ ] := 0
```

```
end for
```

```
num_dfs := 0
```

```
es_biconexo := true
```

▷ Basta lanzar un DFS porque G es conexo

```
 $v$  := un vértice cualquiera de  $G$ 
```

```
BICONEXO-REC( $G, v, v, es\_biconexo, ndfs, \dots$ )
```

```
return  $es\_biconexo$ 
```

```
end procedure
```

```

procedure BICONEXO-REC( $G, v, \text{padre}, \dots$ )
     $\text{num\_dfs} := \text{num\_dfs} + 1; \text{ndfs}[v] := \text{num\_dfs}$ 
     $\text{visitado}[v] := \text{true}$ 
     $\text{mas\_alto}[v] := \text{ndfs}[v]$ 
    for  $w \in G.\text{ADJACENT}(v)$  do
        if  $\neg \text{visitado}[w]$  then
             $\text{num\_desc}[v] := \text{num\_desc}[v] + 1$ 
            BICONEXO-REC( $G, w, v, \dots$ )
             $\text{mas\_alto}[v] := \min(\text{mas\_alto}[v], \text{mas\_alto}[w])$ 
             $\text{punto\_art}[v] := \text{punto\_art}[v] \vee$ 
                 $(\text{mas\_alto}[w] \geq \text{ndfs}[v])$ 
        else if  $\text{padre} \neq w$  then
             $\text{mas\_alto}[v] := \min(\text{mas\_alto}[v], \text{ndfs}[w])$ 
        end if
    end for
    if  $v = \text{padre}$  then  $\triangleright v$  es la raíz del  $T_{\text{DFS}}$ 
         $\text{punto\_art}[v] := \text{num\_desc}[v] > 1$ 
    end if
     $\text{es\_biconexo} := \text{es\_biconexo} \wedge \neg \text{punto\_art}[v]$ 
end procedure

```

Análisis del DFS

Supongamos que el trabajo que se realiza en cada visita de un vértice (PRE- y POST-VISIT) y cada visita de una arista, sea del árbol o de retroceso, es $\Theta(1)$. Entonces el coste del DFS es

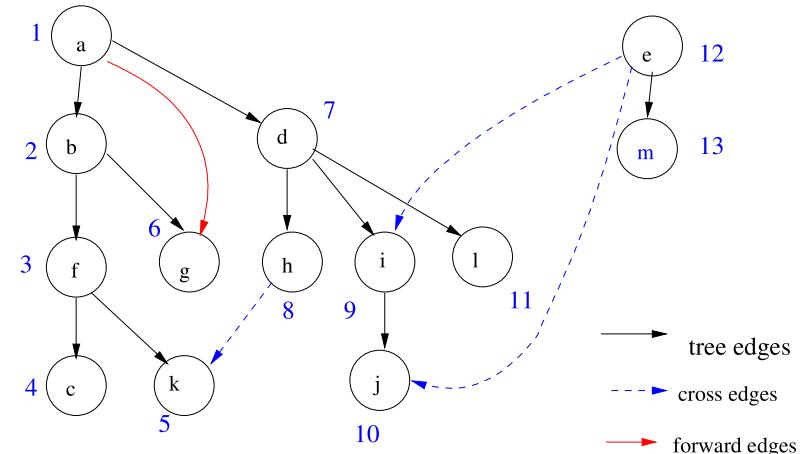
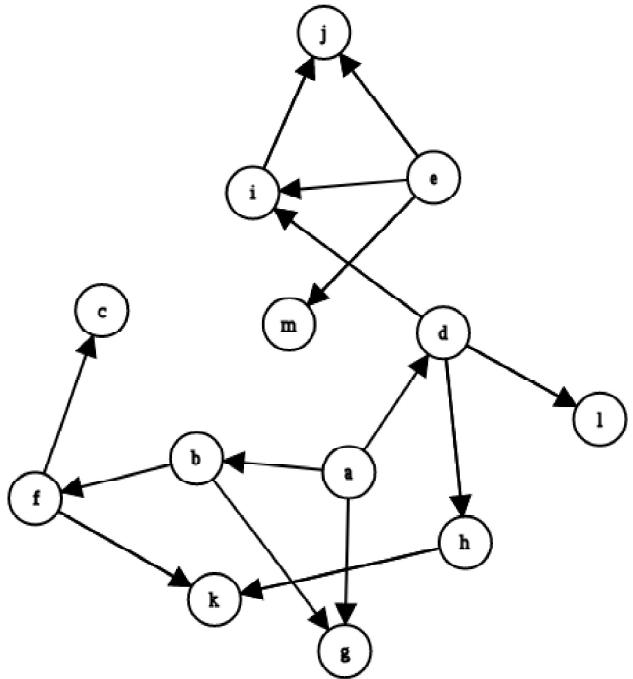
$$\begin{aligned} \sum_{v \in V} (\Theta(1) + \Theta(\text{grado}(v))) &= \Theta \left(\sum_{v \in V} (1 + \text{grado}(v)) \right) = \\ &\Theta \left(\sum_{v \in V} 1 + \sum_{v \in V} \text{grado}(v) \right) = \Theta(n + m) \end{aligned}$$

DFS en Digrafos

El recorrido en profundidad de un digrafo tiene propiedades algo distintas de las del DFS de un grafo no dirigido. Al lanzar un DFS desde un vértice v de un digrafo G se visitarán los vértices (no visitados) de la componente fuertemente conexa de v pero además todos los otros vértices **accesibles** desde v . El DFS de un digrafo induce sucesivos árboles de recorrido, enraizados en los vértices desde los cuales se lanzan los recorridos recursivos. Los conceptos de numeración DFS y de numeración inversa son igual que en el caso de grafos no dirigidos.

El DFS de un digrafo también induce una clasificación de los arcos, pero es algo diferente:

- 1 Arcos de árbol: van del vértice v en curso a un vértice no visitado w
- 2 Arcos de retroceso: van del vértice v en curso a un vértice antecesor w en el árbol T_{DFS} ; se cumple que $\text{ndfs}[w] < \text{ndfs}[v]$ y que $\text{ndfs}[w]$ está abierto
- 3 Arcos de avance (*forward edges*): van del vértice v en curso a un vértice descendiente pero previamente visitado w en el T_{DFS} ; se cumple que $\text{ndfs}[v] < \text{ndfs}[w]$
- 4 Arcos de cruce (*cross edges*): van del vértice v en curso a un vértice previamente visitado w en el mismo T_{DFS} o en un árbol diferente; se cumple que $\text{ndfs}[w] < \text{ndfs}[v]$, pero el vértice w ya está cerrado ($ninv[w] \neq 0$)

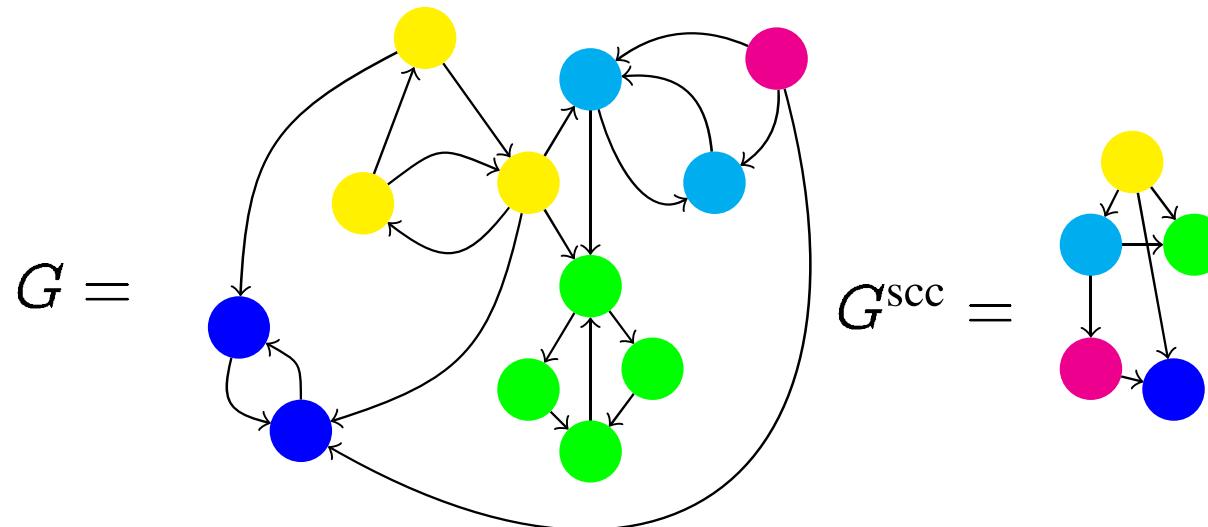


Aplicaciones del DFS: Componentes Fuertemente Conexas

Consideraremos aquí el algoritmo de Kosaraju-Sharir para calcular las componentes fuertemente conexas (SCC, *strongly connected components*) de un digrafo G . Para ello se emplean dos DFS sobre el digrafo G .

El algoritmo se fundamenta en la observación de que G y G^T (el digrafo que G transpuesto, es decir, el que se obtiene invirtiendo la orientación de todos los arcos de G) tienen idénticas SCCs.

Dado un digrafo $G = \langle V, E \rangle$ se denomina grafo de condensación o grafo de SCCs al digrafo acíclico dirigido $G^{\text{SCC}} = \langle V^{\text{SCC}}, E^{\text{SCC}} \rangle$ en el que hay un vértice $[v]$ por cada SCC de G y existe un arco $([u], [v])$ si existen vértices $x, y \in G$ tales que $(x, y) \in E$ y x pertenece a la SCC asociada a $[u]$ e y pertenece a la SCC asociada a $[v]$.



Por otro lado, es obvio que $(G^{\text{SCC}})^T = (G^T)^{\text{SCC}}$.

Supongamos que iniciamos un DFS en un vértice u que pertenece a un vértice hoja en $(G^{\text{scc}})^T$. Entonces visitaremos exactamente los vértices de la SCC de u (ej: un vértice amarillo en el ejemplo). Si visitamos los vértices de G^{scc} en orden topológico inverso iremos recobrando todas las SCCs. Por ejemplo, si ya se ha visitado la SCC amarilla y lanzamos un DFS desde un vértice verde visitaremos todos los vértices de la SCC verde (y nada más).
Pero en el algoritmo querremos evitar la construcción explícita de $(G^{\text{scc}})^T$.

Lanzamos sucesivos DFS sobre G y vamos guardando los vértices de G en una lista L en orden inverso de cierre (esto es, el primer vértice cerrado en el primer DFS es el último de L y así sucesivamente). Si un vértice x aparece en L antes que un vértice y entonces ambos pertenecen a una misma SCC o bien la SCC de x debe visitarse antes que la SCC de y en un orden topológico inverso de $(G^{\text{SCC}})^T$ porque en G es seguro que desde y no puede alcanzarse a x , salvo que estén en la misma SCC. Por lo tanto en G^T o bien x e y son de la misma SCC o desde x no puede alcanzarse a y .

El algoritmo resuelve el problema entonces mediante dos recorridos en profundidad (tres, si hay que calcular G^T o las listas de predecesores) y como todas las operaciones por vértice y por arco aplicadas son de coste constante, el coste del algoritmo es lineal respecto al tamaño del digrafo $\Theta(|V| + |E|)$.

Primera fase del algoritmo de Kosaraju-Sharir

- ▷ Post: Genera una lista L de los vértices de G
- ▷ en orden descendiente de número de cierre

procedure DFS(G, L)

for $v \in V(G)$ **do**

$visitado[v] := \text{false}$

end for

$L := \emptyset$

for $v \in V(G)$ **do**

if $\neg visitado[v]$ **then**

 DFS-REC($G, v, L, visitado$)

end if

end for

end procedure

```
procedure DFS-REC( $G, v, L, visitado$ )
     $visitado[v] := \text{true}$ 
    for  $w \in G.\text{SUCCESSOR}(v)$  do
        if  $\neg visitado[w]$  then
            DFS-REC( $G, w, L, visitado$ )
        end if
    end for
     $L := \{v\} + L$ 
end procedure
```

Segunda fase del algoritmo de Kosaraju-Sharir

procedure EXTRACTSCC(G)

EXTRACT-SCCG, L

for $v \in V(G)$ **do**

$SCC[v] := -1$

end for

$nscc := 0$

for $v \in L$ **do**

if $SCC[v] = -1$ **then**

$nscc := nscc + 1$

- ▷ Necesitaremos los predecesores
- ▷ de cada vértice de G

 EXTRACTSCC-REC($G, v, SCC, nscc$)

end if

end for

end procedure

```
procedure EXTRACTSCC-REC( $G, v, SCC, nscc$ )
     $SCC[v] := nscc$ 
    for  $w \in G.\text{PREDECESSOR}(v)$  do
        if  $SCC[w] = -1$  then
            EXTRACTSCC-REC( $G, v, SCC, nscc$ )
        end if
    end for
end procedure
```

Parte II

Búsqueda y Ordenación Digital

Parte II

Búsqueda y Ordenación Digital

- Tries
- Ordenación Digital

Tries

Las claves que identifican a los elementos de una colección están formadas por una secuencia de símbolos (p.e. caracteres, dígitos, bits), siendo esta descomposición más o menos “natural”, y dicha circunstancia puede aprovecharse ventajosamente para implementar las operaciones típicas de un diccionario de manera notablemente eficiente.

Adicionalmente, es frecuente que necesitemos ofertar operaciones del TAD basadas en esta descomposición de las claves en símbolos: por ejemplo, podemos querer una operación que dada una colección de palabras C y una palabra p nos devuelva la lista de todas las palabras de C que contienen a la palabra p como subcadena.

Consideremos un alfabeto finito $\Sigma = \{\sigma_1, \dots, \sigma_m\}$ de cardinalidad $m \geq 2$. Mediante Σ^* denotaremos, como es habitual en muchos contextos, el conjunto de las secuencias o cadenas formadas por símbolos de Σ . Dadas dos secuencias u y v denotaremos $u \cdot v$ la secuencia resultante de concatenar u y v . Para la secuencia de longitud 0, es decir, la secuencia vacía usaremos la notación λ .

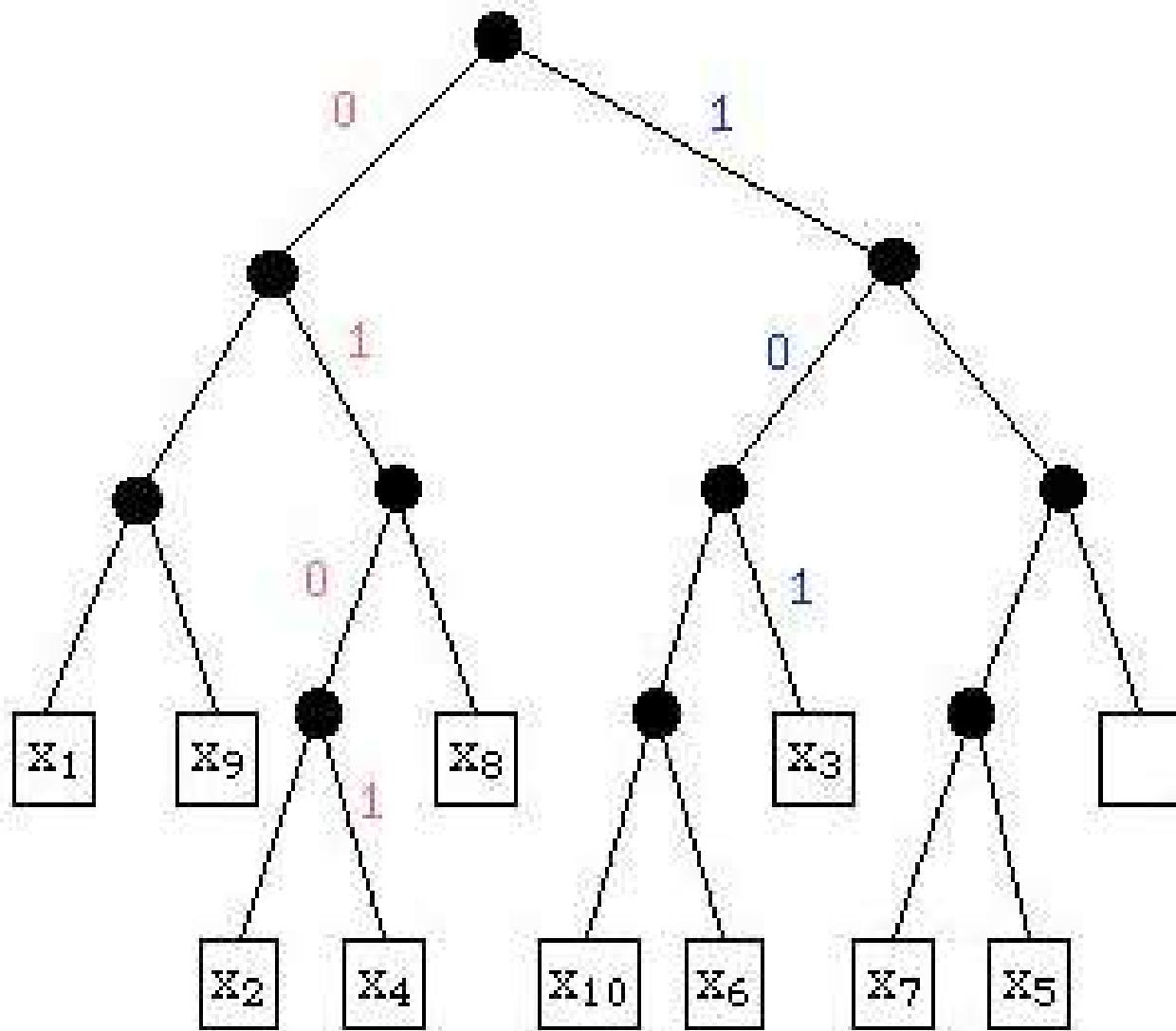
Definición

Dado un conjunto finito de secuencias $X \subset \Sigma^*$ de idéntica longitud, el *trie* T correspondiente a X es un árbol m -ario definido recursivamente de la siguiente manera:

- 1 Si X contiene un sólo elemento o ninguno entonces T es un árbol consistente en un único nodo que contiene al único elemento de X o está vacío.
- 2 Si $|X| \geq 2$, sea T_i el trie correspondiente a $X_i = \{y \mid x = \sigma_i \cdot y \in X \wedge \sigma_i \in \Sigma\}$. Entonces T es un árbol m -ario constituido por una raíz \circ y los m subárboles T_1, T_2, \dots, T_m .

2

X ₁	=	000101
X ₂	=	010001
X ₃	=	101000
X ₄	=	010101
X ₅	=	110101
X ₆	=	100111
X ₇	=	110001
X ₈	=	011111
X ₉	=	001110
X ₁₀	=	100001



Lema

Si las aristas del trie T correspondiente a un conjunto X se etiquetan mediante los símbolos de Σ de tal modo que la arista que une la raíz con su primer subárbol se etiqueta σ_1 , la que une la raíz con su subárbol se etiqueta σ_2 , etc. entonces las etiquetas del camino que nos llevan desde la raíz hasta una hoja no vacía que contiene a x constituyen el prefijo más corto que distingue unívocamente a x ; es decir, ningún otro elemento de X empieza con el mismo prefijo.

Lema

Sea p la etiqueta correspondiente a un camino que va de la raíz de un trie T hasta un cierto nodo (interno u hoja) de T . Entonces el subárbol enraizado en dicho nodo contiene todos los elementos de X que tienen en común al prefijo p (y no más elementos).

Lema

Dado un conjunto $X \subset \Sigma^$ de secuencias de igual longitud, su trie correspondiente es único. En particular T no depende del orden en que se “presenten” los elementos de X .*

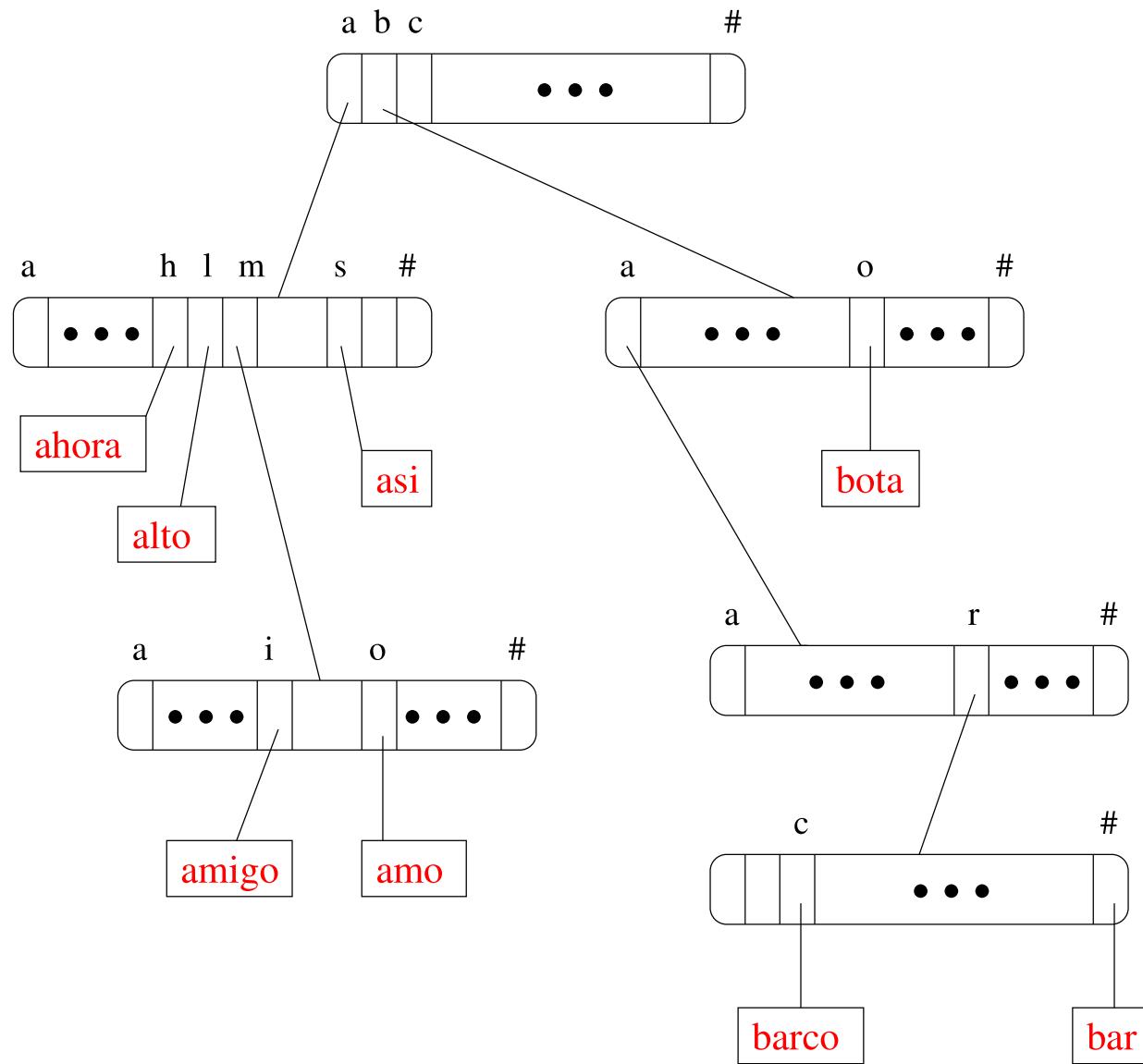
Lema

La altura de un trie T es igual a la longitud mínima de prefijo necesaria para distinguir cualesquiera dos elementos del conjunto al que corresponde el trie. En particular, si ℓ es la longitud de las secuencias en X , la altura de T será $\leq \ell$.

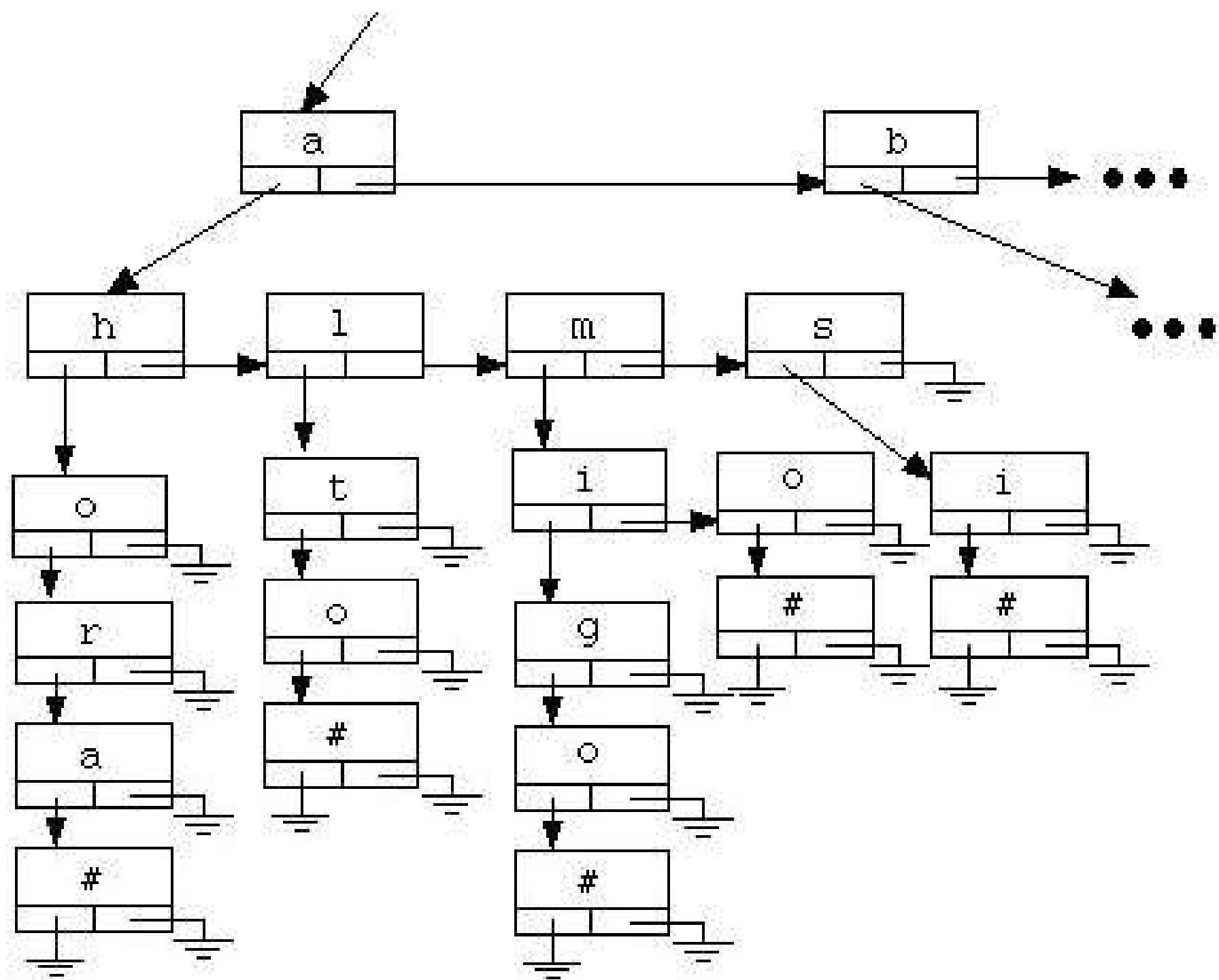
La definición de tries impone que todas las secuencias sean de igual longitud, lo cual es muy restrictivo. Pero si no exigimos esta condición entonces tenemos un problema que habremos de afrontar: si un elemento x es prefijo propio de otro elemento y , cómo podremos distinguirlo? Cómo diferenciamos las situaciones en la que x e y pertenecen ambos a X de las situaciones en la que sólo y está en X ?

Una solución habitual consiste en ampliar Σ con un símbolo especial (p.e. \sharp) de fin de secuencia, y marcar cada una de las secuencias en X con dicho símbolo. Ello garantiza que ninguna de las secuencias (marcadas) es prefijo propio de otra. El “precio” a pagar es que hay que trabajar con un alfabeto de $m + 1$ símbolos.

$$X = \{\text{ahora, alto, amigo, amo, asi, bar, barco, bota, ...}\}$$



Las técnicas de implementación de los tries son las convencionales para árboles. Si se utiliza un vector de apuntadores por nodo, los símbolos de Σ suelen poderse utilizar directamente como índices (eventualmente, se habrá de utilizar una función $ind : \Sigma \rightarrow \{1, \dots, m\}$). Las hojas que contienen los elementos de X pueden almacenar exclusivamente los sufijos restantes, ya que el prefijo está ímplicitamente codificado en el camino desde la raíz a la hoja. En el caso en que se utilice la representación primogénito-siguiente hermano, cada nodo almacena un símbolo y sendos apuntadores al primogénito y al siguiente hermano. Puesto que suele haber definido un orden sobre Σ la lista de hijos de cada nodo suele ordenarse de acuerdo a áquel.



Aunque es más costoso en espacio emplear nodos del trie para representar las palabras completas, resulta ventajoso evitar la necesidad de nodos de distinto tipo, apuntadores a nodos de diferentes tipos, o representaciones poco eficientes para los nodos (p.e. *unions*).

```
// La clase Clave debe soportar las siguientes
// operaciones:
// x.length() devuelve la longitud >= 0 de una clave x
template <typename Clave>
int length() throw();

// x[i] devuelve el i-ésimo símbolo de x,
// lanza un error si i < 0 o i >= x.length()
template <typename Simbolo, typename Clave>
Simbolo operator[](const Clave& x, int i) throw(error);

template <typename Simbolo, typename Clave, typename Valor>
class DiccDigital {
public:
    ...
private:
    struct nodo_trie {
        Simbolo _c;
        nodo_trie* _primg;
        nodo_trie* _sigher;
        Valor _v;
    };
    nodo_trie* raiz;
    ...
};
```

```

template <typename Simbolo,
           typename Clave,
           typename Valor>
void DiccDigital<Simbolo,Clave,Valor>::busca(
    const Clave& k, bool& esta, Valor& v) const
    throw(error) {

    nodo_trie* p = busca_en_trie(raiz, k, 0);
    if (p == nullptr)
        esta = false;
    else {
        esta = true;
        v = p -> _v;
    }
}

// Pre: p es la raíz del subárbol contenido
// las claves cuyos i-1 primeros símbolos
// coinciden con los i-1 símbolos iniciales de k
// Coste: Θ(longitud(k))
template <typename Simbolo, typename Clave,
           typename Valor>
DiccDigital<Simbolo,Clave,Valor>::nodo_trie*
DiccDigital<Simbolo,Clave,Valor>::busca_en_trie(
    nodo_trie* p, const Clave& k,
    int i) const throw() {

    if (p == nullptr)      return nullptr;
    if (i == k.length()) return p;
    if (p -> _c > k[i]) return nullptr;
    if (p -> _c < k[i])
        return busca_en_trie(p -> _sigher, k, i);
    // p -> _c == k[i]
    return busca_en_trie(p -> _primg, k, i+1);
}

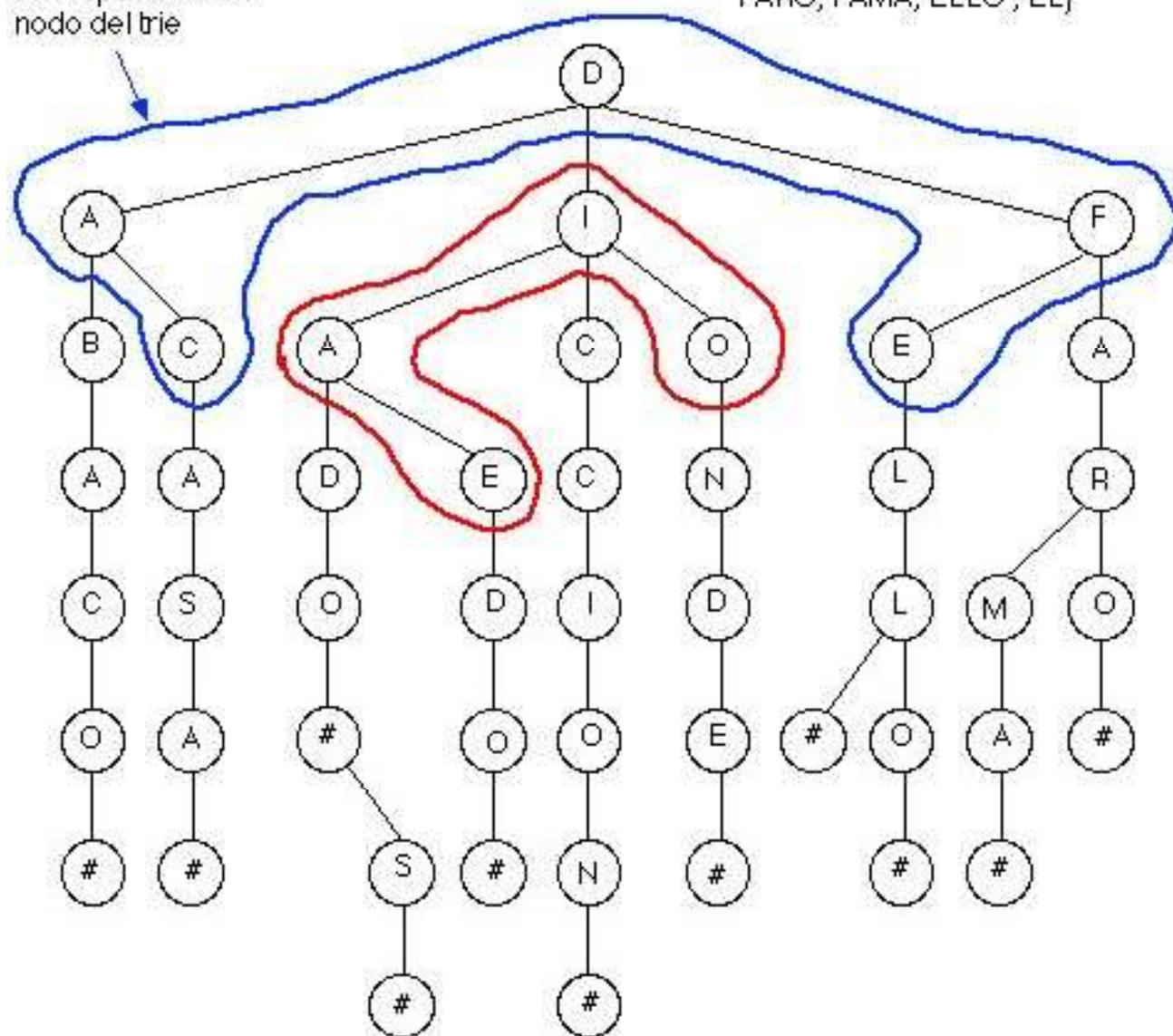
```

Una solución que intenta combinar eficiencia en el acceso a los subárboles y en espacio consiste en implementar cada “nodo” del trie como un BST. La estructura resultante se denomina *árbol ternario de búsqueda* ya que cada uno de sus nodos contiene tres apuntadores: dos apuntadores al hijo izquierdo y derecho, respectivamente, en el BST, y un apuntador a la raíz del subárbol al que da acceso el nodo.

```
template <typename Simbolo,
          typename Clave,
          typename Valor>
class DiccDigital {
public:
    ...
    void inserta(const Clave& k, const Valor& v)
        throw(error);
    ...
private:
    struct nodo_tst {
        Simbolo _c;
        nodo_tst* _izq;
        nodo_tst* _cen;
        nodo_tst* _der;
        Valor _v;
    };
    nodo_tst* raiz;
    ...
static nodo_tst* inserta_en_tst(nodo_tst* t,
                                int i, const Clave& k, const Valor& v)
    throw(error);
    ...
};
```

X = {DICCION, DADO, DADOS,
DEDO; DONDE, ABACO, CASA,
FARO, FAMA, ELLO, EL}

corresponde a un
nodo del trie



```
template <typename Simbolo,
          typename Clave,
          typename Valor>
void DiccDigital<Simbolo,Clave,Valor>::inserta(
    const Clave& k, const Valor& v) throw(error) {
    // Simbolo() es un simbolo nulo,
    // p.e. si Simbolo==char entonces
    // Simbolo() == '\0'
    k[k.length()] = Simbolo(); // añadir centinela
                            // al final de la clave
    raiz = inserta_en_tst(raiz, 0, k, v);
}
```

```

template <typename Simbolo,
          typename Clave,
          typename Valor>
DiccDigital<Simbolo,Clave,Valor>::nodo_tst*
DiccDigital<Simbolo,Clave,Valor>::inserta_en_tst (
    nodo_tst* t, int i,
    const Clave& k, const Valor& v)
throw(error) {

    if (t == nullptr) {
        t = new nodo_tst;
        t -> _izq = t -> _der = t -> cen = nullptr;
        t -> _c = k[i];
        if (i < k.length() - 1) {
            t -> _cen = inserta_en_tst(t -> _cen, i + 1, k, v);
        } else { // i == k.length() - 1; k[i] == Simbolo()
            t -> _v = v;
        }
    } else {
        if (t -> _c == k[i])
            t -> _cen = inserta_en_tst(t -> _cen, i + 1, k, v);
        if (k[i] < t -> _c)
            t -> _izq = inserta_en_tst(t -> _izq, i, k, v);
        if (t -> _c < k[i])
            t -> _der = inserta_en_tst(t -> _der, i, k, v);
    }
    return t;
}

```

Parte II

Búsqueda y Ordenación Digital

- Tries
- Ordenación Digital

Ordenación digital

La descomposición digital de las claves puede emplearse además de para la búsqueda para la ordenación. Los algoritmos de ordenación basados en estos principios se denominan de **ordenación digital**.

- *Counting sort*
- *Bucket sort*
- *Radix sort*

Estos algoritmos ordenan un vector de tamaño n que contienen enteros en un rango $[0..r]$, por ejemplo, con coste inferior a $\Theta(n \log n)$, a menudo $\Theta(n)$. También pueden usarse estas ideas para ordenar n strings de longitud ℓ cada uno y formados por símbolos de un cierto alfabeto finito $\Sigma = \{\sigma_1, \dots, \sigma_U\}$.

Counting sort

Supongamos que todos los elementos de A están en un cierto rango $[0..r]$, siendo r un valor razonablemente pequeño, $r = \Theta(n)$. Entonces el algoritmo simplemente cuenta, para cada $j \in [0..r]$ cuántos j 's hay en A ; sea c_j dicho número. Una vez hemos hecho esto, basta generar un vector con c_0 ceros, c_1 unos, etc.

El algoritmo usará un vector auxiliar B de tamaño n para la salida y otro vector auxiliar C de tamaño r para los conteos.

Counting sort

Require: $0 \leq A[i] \leq r$ para todo i , $0 \leq i < n$

procedure COUNTINGSORT($A[0..n - 1]$, r)

- for** $j := 0$ **to** r **do** $\triangleright O(r)$
 - $C[j] := 0$
- end for**
- for** $i := 0$ **to** $n - 1$ **do** $\triangleright O(n)$
 - $C[A[i]] := C[A[i]] + 1;$
- end for**
- $\triangleright C[j] = \text{número de } j\text{'s en } A, 0 \leq j \leq r$
- for** $j := 1$ **to** r **do** $\triangleright O(r)$
 - $C[j] := C[j] + C[j - 1]$
- end for**
- $\triangleright C[j] = \text{número de valores } \leq j\text{'s en } A, 0 \leq j \leq r$
- for** $i := n - 1$ **downto** 0 **do** $\triangleright O(n)$
 - $B[C[A[i]]] := A[i]; C[A[i]] := C[A[i]] - 1;$
- end for**

end procedure

Counting sort

El coste de counting sort es obviamente $\Theta(n + r)$. Si $r = \mathcal{O}(n)$ entonces el coste es $\Theta(n)$.

Dado que se necesita espacio auxiliar $\Theta(r)$ el algoritmo de counting sort es justamente interesante solo cuando $r = \mathcal{O}(n)$, de otro modo se incurre en un tiempo mayor que lineal y lo peor, espacio superlineal también.

Es importante destacar que la implementación dada hace que *counting sort* sea **estable**: en caso de empate, se respeta el orden original.

Bucket sort

El algoritmo de **bucket sort** se basa en un principio similar a *counting sort*: se usa una estructura de datos auxiliar con B cubetas (*buckets*) y los n objetos a ordenar se reparten entre las B cubetas. Por ejemplo una cubeta puede contener todos los strings que comienzan por una cierta letra. Todo objeto en la cubeta B_i ha de ser menor o igual que cualquier objeto en la cubeta B_{i+1} , con arreglo al orden que nos interesa.

Después se ordena cada cubeta utilizando un algoritmo de coste lineal, y la salida final se obtiene escribiendo en el vector de salida los contenidos (ya ordenados) de las sucesivas cubetas.

Bucket sort

El coste de la primera fase será $\Theta(n)$ asumiendo que determinar y colocar cada objeto $A[i]$ en la cubeta B_j que le corresponde se hace en tiempo constante.

Si b_j denota el número de objetos en la cubeta B_j , ordenar los objetos de b_j tendrá coste $\Theta(b_j)$. Y añadir los contenidos ordenados en la salida también es $\Theta(b_j)$.

En total:

$$\Theta(n) + \sum_{j=1}^B \Theta(b_j) = \Theta(n) + \Theta\left(\sum_{j=1}^B b_j\right) = \Theta(n).$$

LSD Radix sort

Consideremos un vector de n elementos cada uno de los cuales es una secuencia de ℓ símbolos.

Si ordenamos repetidamente el vector con arreglo a los sucesivos símbolos de derecha a izquierda (en caso de ser bits diríamos del bit de menor peso al bit de mayor peso) mediante un algoritmo de ordenación **estable** de coste lineal entonces el vector queda ordenado en tiempo $\Theta(n \cdot \ell)$.

Este el algoritmo de LSD radix sort (LSD=Least Significant Digit).

LSD Radix sort

```
procedure LSDRADIXSORT( $A$ ,  $\ell$ )
    for  $i := 0$  to  $\ell - 1$  do
        Ordenar  $A$  con un sort estable de coste
        lineal, de acuerdo los  $i$ -ésimos símbolos
    end for
end procedure
```

LSD Radix sort

Ejemplo

329	720	720	329
475	475	329	355
657	355	436	436
$\ell = 3$	839	436	838
	436	657	355
	720	329	657
	355	839	720
		475	839

LSD Radix sort

Teorema

LSD Radix sort ordena correctamente el vector de n elementos.

Demostración

Por inducción sobre ℓ .

Base: Si $\ell = 1$ se ordenan los n elementos de A con arreglo a su único símbolo.

Hipótesis de inducción: Los n elementos están correctamente ordenados si nos fijamos exclusivamente en sus últimos $\ell - 1$ símbolos. Sean $A[i]$ y $A[j]$ dos elementos cualesquiera del vector tras ordenar respecto a $\ell - 1$ símbolos.

LSD Radix sort

Demostración (continúa)

Si ordenamos con ordenación estable respecto al símbolo ℓ -ésimo (de mayor peso) y $A[i][\ell] = A[j][\ell]$ entonces la ordenación estable mantendrá el orden que tuvieran $A[i]$ y $A[j]$ ya que empatan respecto al símbolo ℓ -ésimo, lo cual es correcto.

Si $A[i][\ell] < A[j][\ell]$ entonces es seguro que $A[i] < A[j]$ y el algoritmo de ordenación estable sobre el símbolo ℓ -ésimo pondrá $A[i]$ precediendo a $A[j]$. □

LSD Radix sort

Supongamos que los elementos del vector son enteros y los “símbolos” que componen los elementos son los dígitos del elemento en una cierta base b , es decir, los símbolos son números en $\{0, \dots, b - 1\}$. Entonces si usamos *counting sort* cada iteración cuesta $\Theta(n + b)$ y el coste total de LSD radix sort es $R(n, b, \ell) = \Theta((n + b) \cdot \ell)$.

- Supongamos que cada entero en el vector es un número positivo $< f(n)$.
- Entonces $\ell = \lceil \log_b(f(n)) \rceil$ y el coste de LSD radix sort es $R(n, \ell, b) = \Theta((n + b) \log f(n))$.
- Si $f(n) = \omega(1)$ entonces el coste es $R(n, \ell, b) = \omega(n)$, mayor que lineal.

Podemos mejorar la eficiencia? Sí: cambiando la base b .

LSD Radix sort

Por ejemplo, si en vez de usar $b = 2$ tomamos $b' = 2^r$ entonces $\ell' = \log_{2^r} f(n) = \frac{\log_2 f(n)}{r}$ y el coste del LSD radix sort se rebaja en un factor $1/r$. En vez de trabajar con símbolos que son bits, nuestros símbolos pasan a ser bloques de r bits.

En general, pasaremos de ℓ bits a $\ell' = \lceil \ell/r \rceil$ dígitos en base 2^r . Entonces el coste de LSD radix sort es $\Theta((n + 2^r)\ell/r)$; tomando $r = c \log_2 n$ para alguna $c < 1$ conseguimos que el coste sea $\Theta(n)$

LSD Radix sort

No obstante, se necesita especial atención para mantener el coste de counting sort lineal, ya que en la nueva situación estaremos trabajando con números de $\Theta(\log n)$ bits y no podemos asumir que mover o copiar números de ese tamaño nos tome tiempo constante.

MSD Radix sort

Consideremos ahora un vector de n elementos cada uno de los cuales es una secuencia de ℓ bits. Dado un elemento x , $\text{bit}(x, i)$ denotará su i -ésimo bit.

Ordenamos el vector con relación al bit de mayor peso y obtenemos dos bloques. Los elementos cuyo bit de mayor peso es 0 y a continuación los elementos con bit de mayor peso 1. Los dos bloques se ordenan separada y recursivamente respecto al bit de segundo mayor peso, y así sucesivamente. Este es el algoritmo de MSD Radix sort (MSD=Most Significant Digit).

MSD Radix sort

Puede generalizarse fácilmente a elementos formados por ℓ símbolos cada uno, donde cada símbolo es un dígito entre 0 y $b - 1$ (o puede verse de esta forma, por ejemplo, letras con códigos ASCII de 0 a 255); cada etapa no recursiva subdivide el bloque en curso en $\leq b$ bloques, cada bloque correspondiendo a los elementos cuyo i -ésimo bit es un cierto dígito/símbolo.

MSD Radix sort

```
// Llamada inicial:  
// radix_sort(A, 0, A.size()-1, ℓ - 1);  
template <typename Elem, typename Symb>  
void radix_sort(vector<Elem>& A, int i, int j, int r) {  
  
    if (i < j and r >= 0) {  
        int k;  
        radix_split(A, i, j, r, k);  
        radix_sort(A, i, k, r - 1);  
        radix_sort(A, k + 1, j, r - 1);  
    }  
}
```

MSD Radix sort

```
// bit(x, r) devuelve el bit r-ésimo de x
// r == 0 => bit de menor peso
// r == ℓ - 1 => bit de mayor peso
template <typename Elem, typename Symb>
void radix_split(vector<Elem>& A, int i, int j,
                 int r, int& k) {

    int u = i; int v = j;
    while (u < v + 1) {
        while (u < v + 1 and bit(A[u], r) == 0) ++u;
        while (u < v + 1 and bit(A[v], r) == 1) --v;
        if (u < v + 1) swap(A[u], A[v]);
    }
    k = v;
}
```

MSD Radix sort

Cada etapa de *radix sort* tiene un coste no recursivo lineal; puesto que la profundidad del árbol de recursión es ℓ , el coste del algoritmo es $\Theta(n \cdot \ell)$. Otra forma de deducir el coste consiste en considerar el coste asociado a cada elemento de A : un elemento cualquiera de A es examinado (y eventualmente intercambiado con otro) a lo sumo ℓ veces, de ahí que el coste total sea $\Theta(n \cdot \ell)$.

MSD Radix sort

MSD radix sort está en correspondencia con los tries de forma análoga a cómo quicksort está en correspondencia con los árboles binarios de búsqueda.

En el análisis de MSD radix sort podemos aplicar consideraciones semejantes a las que hacíamos para LSD radix sort: cambiando la base b haremos hasta b bloques con cada `radix_split` y la profundidad de la recursión se reducirá en un factor $1/r$.

MSD Radix sort

El algoritmo que hace la partición en bloques `radix_split` **no** respeta el orden original y en consecuencia MSD radix sort **no es estable**, a diferencia de lo que sucede con LSD radix sort.

Por otro lado si aumentamos la base b (=número de símbolos distintos posibles) se nos complica enormemente el algoritmo de partición a no ser que se use espacio extra auxiliar—y una de las ventajas de MSD radix sort es **no** necesitar espacio auxiliar!

Quicksort for Strings

Una solución intermedia entre quicksort y MSD radix sort que funciona de manera muy eficiente en la práctica para ordenar n strings se inspira en las ideas que fundamentan los TSTs.

```
// Cada string está acabado con un carácter end-of-string
// menor que cualquier otro carácter.
void qsort(vector<string>& A, int i, int j, int r) {
    if (i < j) {
        int p, q;
        char c = A[i][r]; // usamos la letra r-ésima de A[i]
                            // como pivote
        3-way-partition(A, c, i, j, p, q);
        // A[i..p-1] = elementos cuya r-ésima letra es < c
        // A[p..q] = elementos cuya r-ésima letra es = c
        // A[q+1..j] = elementos cuya r-ésima letra es > c
        qsort(A, i, p-1, r);
        qsort(A, p, q, r+1);
        qsort(A, q+1, j, r);
    }
}
```

Parte III

Algoritmos Voraces

Parte III

Algoritmos Voraces

- Introducción a los Algoritmos Voraces
- Compresión de Datos: Códigos de Huffman
- Árboles de Expansión Mínimos: Algoritmos de Kruskal y de Prim
- Particiones

Algoritmos Voraces

- Los algoritmos voraces suelen emplearse para resolver **problemas de optimización combinatoria**: Dada una entrada, buscamos una solución óptima al problema de acuerdo a una cierta **función objetivo**. Las soluciones consisten en una secuencia de elementos.
- Por ejemplo, dado un grafo $G = \langle V, E \rangle$ y dos vértices $u, v \in G$ buscamos un camino de u a v con el mínimo número de aristas. La solución es la secuencia de vértices o aristas que conforman el camino.

Algoritmos Voraces

- Los algoritmos voraces suelen emplearse para resolver **problemas de optimización combinatoria**: Dada una entrada, buscamos una solución óptima al problema de acuerdo a una cierta **función objetivo**. Las soluciones consisten en una secuencia de elementos.
- Por ejemplo, dado un grafo $G = \langle V, E \rangle$ y dos vértices $u, v \in G$ buscamos un camino de u a v con el mínimo número de aristas. La solución es la secuencia de vértices o aristas que conforman el camino.

Algoritmos Voraces

Un algoritmo voraz obtiene una solución óptima tomando una serie de decisiones **irreversibles** (no hay *backtracking*).

- Las decisiones tomadas son localmente óptimas (**decisiones miopes**) con el fin de construir, incrementalmente, la solución globalmente óptima.
- Pero en muchas situaciones adoptar decisiones localmente óptimas **no** nos conduce a una solución globalmente óptima.
- A veces empleamos los algoritmos voraces para obtener soluciones aproximadas, subóptimas pero suficientemente cercanas a la optimalidad, y todo ello de manera mucho más eficiente

Algoritmos Voraces

Un algoritmo voraz obtiene una solución óptima tomando una serie de decisiones **irreversibles** (no hay *backtracking*).

- Las decisiones tomadas son localmente óptimas (**decisiones miopes**) con el fin de construir, incrementalmente, la solución globalmente óptima.
- Pero en muchas situaciones adoptar decisiones localmente óptimas **no** nos conduce a una solución globalmente óptima.
- A veces empleamos los algoritmos voraces para obtener soluciones aproximadas, subóptimas pero suficientemente cercanas a la optimalidad, y todo ello de manera mucho más eficiente

Algoritmos Voraces

Un algoritmo voraz obtiene una solución óptima tomando una serie de decisiones **irreversibles** (no hay *backtracking*).

- Las decisiones tomadas son localmente óptimas (**decisiones miopes**) con el fin de construir, incrementalmente, la solución globalmente óptima.
- Pero en muchas situaciones adoptar decisiones localmente óptimas **no** nos conduce a una solución globalmente óptima.
- A veces empleamos los algoritmos voraces para obtener soluciones aproximadas, subóptimas pero suficientemente cercanas a la optimalidad, y todo ello de manera mucho más eficiente

Algoritmos Voraces

- Los algoritmos voraces son por naturaleza iterativos; en cada iteración se toma una decisión miope e irreversible localmente óptima respecto a una “componente” de la solución, y a continuación se resuelve el subproblema resultante de tomar dicha decisión.
- Cada decisión puede depender de las decisiones anteriores, pero no de elecciones futuras. El esquema voraz también se emplea a menudo para diseñar algoritmos **on-line**, que deben tomar decisiones para una secuencia de peticiones, sin conocimiento de las decisiones futuras
- Un algoritmo voraz jamás revierte una decisión ya tomada (nunca hace *backtrack*).

Algoritmos Voraces

- Los algoritmos voraces son por naturaleza iterativos; en cada iteración se toma una decisión miope e irreversible localmente óptima respecto a una “componente” de la solución, y a continuación se resuelve el subproblema resultante de tomar dicha decisión.
- Cada decisión puede depender de las decisiones anteriores, pero no de elecciones futuras. El esquema voraz también se emplea a menudo para diseñar algoritmos **on-line**, que deben tomar decisiones para una secuencia de peticiones, sin conocimiento de las decisiones futuras
- Un algoritmo voraz jamás revierte una decisión ya tomada (nunca hace *backtrack*).

Algoritmos Voraces

- Los algoritmos voraces son por naturaleza iterativos; en cada iteración se toma una decisión miope e irreversible localmente óptima respecto a una “componente” de la solución, y a continuación se resuelve el subproblema resultante de tomar dicha decisión.
- Cada decisión puede depender de las decisiones anteriores, pero no de elecciones futuras. El esquema voraz también se emplea a menudo para diseñar algoritmos **on-line**, que deben tomar decisiones para una secuencia de peticiones, sin conocimiento de las decisiones futuras
- Un algoritmo voraz jamás revierte una decisión ya tomada (nunca hace *backtrack*).

Algoritmos Voraces

Para que una estrategia voraz funcione correctamente es necesario que el problema considerado reúna dos características:

- **Propiedad de la decisión voraz (*greedy choice property*):** un óptimo global es alcanzable tomando decisiones localmente óptimas.
- **Subestructura óptima:** una vez tomadas una serie de decisiones locales, existe una solución globalmente óptima del problema original que contiene la solución parcial representada por las decisiones ya tomadas.

Frecuentemente, el criterio de optimización local usado para tomar las decisiones voraces induce un orden global que se utilizará para ordenar la toma de decisiones del algoritmo voraz.

El problema de la Mochila Fraccionaria

Dado un conjunto de n objetos, cada uno con peso $w_i > 0$ y valor $v_i > 0$, y una capacidad o peso máximo admisible $W > 0$ de la mochila, el objetivo es determinar un conjunto de objetos o fracciones de objetos que maximicen el valor acumulado y cuyo peso combinado no exceda W .

En definitiva, queremos determinar los valores $x_i \in [0, 1]$, $1 \leq i \leq n$, tales que

$$\sum_{i=1}^n x_i v_i$$

es máximo y $\sum_{i=1}^n x_i w_i \leq W$. Asumiremos, sin pérdida de generalidad, que $\sum_{i=1}^n w_i > W$, de otro modo la solución buscada es trivialmente $x_i = 1$ para todo i .

El problema de la Mochila Fraccionaria

Ejemplo

$$n = 5, W = 100$$

Item	1	2	3	4	5
w	10	20	30	40	50
v	20	30	66	40	60



正面

El problema de la Mochila Fraccionaria

```
procedure GREEDYFKNAPSACK( $n, v, w, W$ )
     $O := \{1, \dots, n\}$ ;  $val := 0$ ;
    while  $W > 0$  do
        Sea  $i \in O$  un elemento con la propiedad P
        if  $w[i] \leq W$  then
             $x[i] := 1$ ;  $W := W - w[i]$ ;  $val := val + v[i]$ ;
        else
             $x[i] := W/w[i]$ ;  $val := val + v[i] * x[i]$ ;  $W := 0$ 
        end if
        Eliminar  $i$  de  $O$ 
    end while
    return  $x$ 
end procedure
```

El problema de la Mochila Fraccionaria

Criterio #1: el objeto más valioso

Ejemplo

$$n = 5, W = 100$$

Item	1	2	3	4	5
w	10	20	30	40	50
v	20	30	66	40	60
x	0	0	1	0.5	1

Valor total **146**

El problema de la Mochila Fraccionaria

Criterio #2: el objeto más ligero

Ejemplo

$$n = 5, W = 100$$

Item	1	2	3	4	5
w	10	20	30	40	50
v	20	30	66	40	60
x	1	1	1	1	0

Valor total 156

El criterio #1 (el objeto más valioso) no nos da la solución óptima!

El problema de la Mochila Fraccionaria

Criterio #3: el objeto con mayor valor por unidad de peso
(mayor v/w)

Ejemplo

$$n = 5, W = 100$$

Item	1	2	3	4	5
w	10	20	30	40	50
v	20	30	66	40	60
v/w	2.0	1.5	2.2	1.0	1.2
x	1	1	1	0	0.8

Valor total **164**

El criterio #2 ([el objeto más ligero](#)) tampoco nos da una solución óptima!

El problema de la Mochila Fraccionaria

Teorema

Si en GREEDYFKNAPSACK seleccionamos en cada iteración el objeto con mayor ratio valor/peso, obtendremos siempre una solución óptima del problema de la Mochila Fraccionaria.

Demostración

Supongamos que ordenamos (y reindexamos) los objetos de tal modo que

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \dots \geq \frac{v_n}{w_n}$$

El problema de la Mochila Fraccionaria

Demostración (continúa)

Sea $X = (x_1, \dots, x_n)$, donde $x_i \in [0, 1]$, la solución voraz. Bajo la suposición de que $\sum w_i > W$ al menos una $x_i < 1$, ya que no podremos colocar todos los objetos enteramente. Sea j el menor índice tal que $x_j < 1$. Por las propiedades del algoritmo tendremos:

- 1 $x_i = 1$, para toda $i < j$
- 2 $x_i = 0$, para toda $i > j$
- 3 Adicionalmente, toda la capacidad de la mochila es usada: $\sum_{i=1}^n x_i w_i = W$

El problema de la Mochila Fraccionaria

Demostración (continúa)

Sea $Y = (y_1, \dots, y_n)$, $y_i \in [0, 1]$, una solución **factible**, i.e.,

$$\sum_{i=1}^n y_i w_i \leq W$$

- Se cumple $\sum_{i=1}^n y_i w_i \leq W = \sum_{i=1}^n x_i w_i$
- Por lo tanto $0 \leq \sum_{i=1}^n x_i w_i - \sum_{i=1}^n y_i w_i = \sum_{i=1}^n (x_i - y_i) w_i$
- Y la diferencia de valor de las soluciones X e Y es

$$\begin{aligned} v(X) - v(Y) &= \sum_{i=1}^n x_i v_i - \sum_{i=1}^n y_i v_i = \sum_{i=1}^n (x_i - y_i) v_i \\ &= \sum_{i=1}^n (x_i - y_i) w_i \frac{v_i}{w_i} \end{aligned}$$

El problema de la Mochila Fraccionaria

Demostración (continúa)

Sea $Y = (y_1, \dots, y_n)$, $y_i \in [0, 1]$, una solución **factible**, i.e.,

$$\sum_{i=1}^n y_i w_i \leq W$$

- Se cumple $\sum_{i=1}^n y_i w_i \leq W = \sum_{i=1}^n x_i w_i$
- Por lo tanto
- $0 \leq \sum_{i=1}^n x_i w_i - \sum_{i=1}^n y_i w_i = \sum_{i=1}^n (x_i - y_i) w_i$
- Y la diferencia de valor de las soluciones X e Y es

$$\begin{aligned} v(X) - v(Y) &= \sum_{i=1}^n x_i v_i - \sum_{i=1}^n y_i v_i = \sum_{i=1}^n (x_i - y_i) v_i \\ &= \sum_{i=1}^n (x_i - y_i) w_i \frac{v_i}{w_i} \end{aligned}$$

El problema de la Mochila Fraccionaria

Demostración (continúa)

Sea $Y = (y_1, \dots, y_n)$, $y_i \in [0, 1]$, una solución **factible**, i.e.,

$$\sum_{i=1}^n y_i w_i \leq W$$

- Se cumple $\sum_{i=1}^n y_i w_i \leq W = \sum_{i=1}^n x_i w_i$
- Por lo tanto
$$0 \leq \sum_{i=1}^n x_i w_i - \sum_{i=1}^n y_i w_i = \sum_{i=1}^n (x_i - y_i) w_i$$
- Y la diferencia de valor de las soluciones X e Y es

$$\begin{aligned} v(X) - v(Y) &= \sum_{i=1}^n x_i v_i - \sum_{i=1}^n y_i v_i = \sum_{i=1}^n (x_i - y_i) v_i \\ &= \sum_{i=1}^n (x_i - y_i) w_i \frac{v_i}{w_i} \end{aligned}$$

El problema de la Mochila Fraccionaria

Demostración (continúa)

Vamos a acotar $v(X) - v(Y) = \sum_{i=1}^n (x_i - y_i)w_i \frac{v_i}{w_i}$.

- Si $i < j$ entonces $x_i = 1$, luego $x_i - y_i \geq 0$ y puesto que $\frac{v_i}{w_i} \geq \frac{v_j}{w_j}$, tendremos

$$(x_i - y_i) \frac{v_i}{w_i} \geq (x_i - y_i) \frac{v_j}{w_j}$$

- Si $i > j$, $x_i = 0$, y $x_i - y_i \leq 0$ pero puesto que $\frac{v_i}{w_i} \leq \frac{v_j}{w_j}$, ahora tenemos

$$(x_i - y_i) \frac{v_i}{w_i} \geq (x_i - y_i) \frac{v_j}{w_j}$$

- Tenemos la misma desigualdad tanto si $i < j$ como si $i > j$.

El problema de la Mochila Fraccionaria

Demostración (continúa)

Vamos a acotar $v(X) - v(Y) = \sum_{i=1}^n (x_i - y_i)w_i \frac{v_i}{w_i}$.

- Si $i < j$ entonces $x_i = 1$, luego $x_i - y_i \geq 0$ y puesto que $\frac{v_i}{w_i} \geq \frac{v_j}{w_j}$, tendremos

$$(x_i - y_i) \frac{v_i}{w_i} \geq (x_i - y_i) \frac{v_j}{w_j}$$

- Si $i > j$, $x_i = 0$, y $x_i - y_i \leq 0$ pero puesto que $\frac{v_i}{w_i} \leq \frac{v_j}{w_j}$, ahora tenemos

$$(x_i - y_i) \frac{v_i}{w_i} \geq (x_i - y_i) \frac{v_j}{w_j}$$

- Tenemos la misma desigualdad tanto si $i < j$ como si $i > j$.

El problema de la Mochila Fraccionaria

Demostración (continúa)

Vamos a acotar $v(X) - v(Y) = \sum_{i=1}^n (x_i - y_i)w_i \frac{v_i}{w_i}$.

- Si $i < j$ entonces $x_i = 1$, luego $x_i - y_i \geq 0$ y puesto que $\frac{v_i}{w_i} \geq \frac{v_j}{w_j}$, tendremos

$$(x_i - y_i) \frac{v_i}{w_i} \geq (x_i - y_i) \frac{v_j}{w_j}$$

- Si $i > j$, $x_i = 0$, y $x_i - y_i \leq 0$ pero puesto que $\frac{v_i}{w_i} \leq \frac{v_j}{w_j}$, ahora tenemos

$$(x_i - y_i) \frac{v_i}{w_i} \geq (x_i - y_i) \frac{v_j}{w_j}$$

- Tenemos la misma desigualdad tanto si $i < j$ como si $i > j$.

El problema de la Mochila Fraccionaria

Demostración (continúa) —

- Usando las desigualdades obtenidas

$$\begin{aligned}v(X) - v(Y) &= \sum_{i=1}^n (x_i - y_i) w_i \frac{v_i}{w_i} \\&\geq \sum_{i=1}^n (x_i - y_i) w_i \frac{v_j}{w_j} \geq \frac{v_j}{w_j} \sum_{i=1}^n (x_i - y_i) w_i \geq 0\end{aligned}$$

- Puesto que $v(X) - v(Y) \geq 0$ para cualquier solución factible Y concluimos que X es una solución óptima.



El problema de la Mochila Fraccionaria

Demostración (continúa) —

- Usando las desigualdades obtenidas

$$\begin{aligned}v(X) - v(Y) &= \sum_{i=1}^n (x_i - y_i) w_i \frac{v_i}{w_i} \\&\geq \sum_{i=1}^n (x_i - y_i) w_i \frac{v_j}{w_j} \geq \frac{v_j}{w_j} \sum_{i=1}^n (x_i - y_i) w_i \geq 0\end{aligned}$$

- Puesto que $v(X) - v(Y) \geq 0$ para cualquier solución factible Y concluimos que X es una solución óptima.



El problema de la Mochila Fraccionaria

```
procedure GREEDYFKNAPSACK( $n, v, w, W$ )
     $O := \{1, \dots, n\}$ ;  $val := 0$ ;
    while  $W > 0$  do
        Sea  $i \in O$  un elemento con  $v_i/w_i$  máximo
        if  $w[i] \leq W$  then
             $x[i] := 1$ ;  $W := W - w[i]$ ;  $val := val + v[i]$ ;
        else
             $x[i] := W/w[i]$ ;  $val := val + v[i] * x[i]$ ;  $W := 0$ 
        end if
        Eliminar  $i$  de  $O$ 
    end while
    return  $x$ 
end procedure
```

Coste? $\mathcal{O}(n^2)$

Podemos obtener un coste menor?

El problema de la Mochila Fraccionaria

```
procedure GREEDYFKNAPSACK( $n, v, w, W$ )
     $O := \{1, \dots, n\}$ ;  $val := 0$ ;  $i := 1$ ;
    Ordenar  $O$  por orden descendente de ratio  $v/w$ 
    while  $W > 0 \wedge i < n$  do
        if  $w[i] \leq W$  then
             $x[i] := 1$ ;  $W := W - w[i]$ ;  $val := val + v[i]$ ;
        else
             $x[i] := W/w[i]$ ;  $val := val + v[i] * x[i]$ ;  $W := 0$ 
        end if
         $i := i + 1$ 
    end while
    return  $x$ 
end procedure
```

Coste? $\mathcal{O}(n \log n)$

El problema de la Mochila Fraccionaria

Teorema

El problema de la Mochila Fraccionaria puede resolverse en tiempo $\mathcal{O}(n \log n)$.

Por contra, el problema de la Mochila Entera o 0-1 KNAPSACK, en el que cada objeto puede ser colocado entero o no colocado en absoluto en la mochila ($x_i \in \{0, 1\}$) es un problema NP-duro.

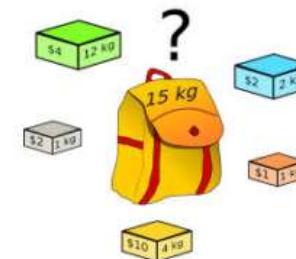
No obstante, la estrategia voraz es una buena heurística para obtener soluciones aproximadas (y también puede usarse para mejorar el rendimiento de algoritmos de backtracking y branch& bound, al permitir mejores podas al explorar espacio de soluciones).

El problema de la Mochila Fraccionaria

Ejemplo

$$n = 3, W = 50$$

Item	1	2	3
w	10	20	30
v	60	100	120
v/w	6	5	4



Con el criterio voraz tomaríamos $x = (1, 1, 0)$ y el valor total sería 160. No es una solución óptima. La solución $y = (0, 1, 1)$ alcanza un valor total de 220.

Planificación de Tareas y Actividades

Planteamiento general:

- Se nos dan n tareas o actividades, cada una de ellas con diferentes características (por ejemplo, duración, tiempo de inicio, coste de ejecución, . . .), que han de ser procesadas en un sistema con **un solo procesador/múltiples procesadores**.
- El objetivo es determinar una **planificación (schedule)**, es decir, cuándo y dónde ejecutar cada tarea, y qué tareas no podrán ser ejecutadas, eventualmente, todo ello con el fin de optimizar un cierto criterio (minimizar el tiempo total de ejecución, el coste, maximizar el *throughput*, . . .)

Problemas de planificación (con un único procesador)

- 1 INTERVAL SCHEDULING: Cada tarea tiene un tiempo de **inicio** y de **final**. En un instante de tiempo dado solo puede haber una tarea en ejecución. El objetivo es maximizar el número total de tareas ejecutadas.
- 2 WEIGHTED INTERVAL SCHEDULING: Como el anterior, pero cada tarea tiene un **beneficio**. El objetivo es maximizar el beneficio total obtenido con las tareas ejecutadas.
- 3 JOB SCHEDULING (minimización del retraso): Las tareas pueden comenzar a ejecutarse en cualquier momento, siempre que el procesador esté libre; cada tarea tiene una duración t_i y un tiempo límite d_i (**deadline**); se define el **retraso** ℓ_i como el tiempo que transcurre entre el deadline y el instante en que finaliza la ejecución de la tarea si dicho instante es posterior al deadline y 0 si es anterior ($\ell_i = \max(0, s_i + t_i - d_i)$). El objetivo es asignar tiempos de inicio s_i a las tareas de manera que en cada momento solo hay una tarea en ejecución y se minimiza el retraso máximo.

Interval scheduling

El problema de INTERVAL SCHEDULING (también conocido como de *selección de actividades*) consiste en determinar un subconjunto de tareas que pueden ser ejecutadas sin conflicto y de máxima cardinalidad.

- Cada tarea i , $1 \leq i \leq n$ tiene un **tiempo de inicio** s_i y un **tiempo de finalización** f_i , con $s_i < f_i$.
- Solo existe un procesador para ejecutar las tareas, en un instante cualquiera de tiempo dado solo puede estar ejecutando **una** tarea.
- Cada tarea debe ejecutarse en su totalidad, desde su tiempo de inicio a su tiempo de finalización, o no ser ejecutada en absoluto.

El objetivo es hallar un subconjunto de tareas **mutuamente compatibles** de **cardinalidad máxima**. Dos tareas i y j distintas son compatibles si $f_i \leq s_j$ o $f_j \leq s_i$.

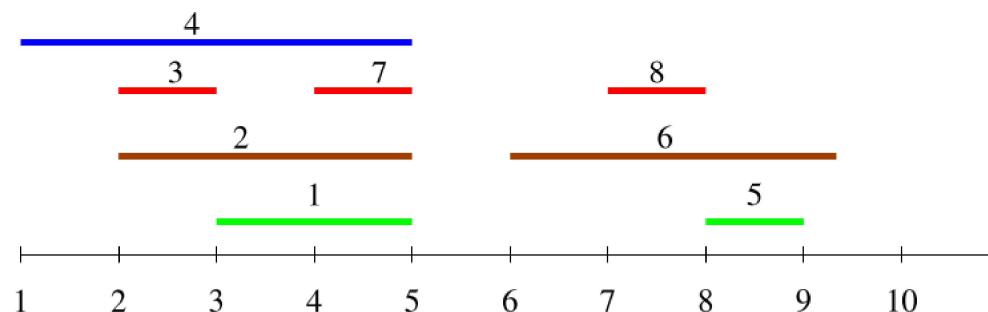
Interval scheduling

Ejemplo

Tarea : 1 2 3 4 5 6 7 8

Inicio (s_i): 3 2 2 1 8 6 4 7

Fin (f_i): 5 5 3 5 9 9 5 8



Interval scheduling

Criterio #1: tiempo de finalización más temprano (*earlier finish time*)

```
procedure INTERVALSCHEDULING( $A$ )
     $S := \emptyset$ ;  $T := \{1, \dots, n\}$ ;
    while  $T \neq \emptyset$  do
        Sea  $i$  la tarea con mínimo  $f_i$  en  $T$ 
         $S := S \cup \{i\}$ ;
        Eliminar  $i$  de  $T$ , y todas las tareas  $j \in T$  t.q.  $s_j < f_i$ 
    end while
    return  $S$ 
end procedure
```

Interval scheduling

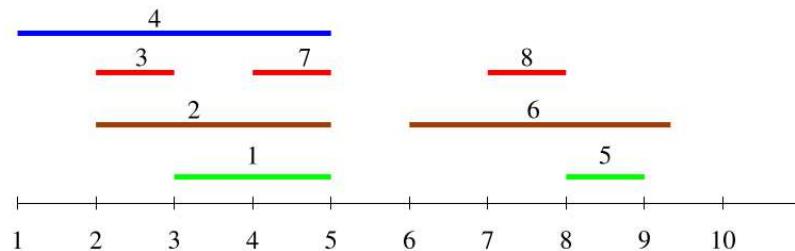
Criterio #1: tiempo de finalización más temprano (*earlier finish time*)

Tarea: 3 4 2 1 7 8 5 6

s_i : 3 1 2 3 4 7 8 6

f_i : 3 5 5 5 5 8 9 9

Sol: 3 1 8 5



Interval scheduling: Corrección

Teorema

El algoritmo voraz con el criterio de tiempo de finalización más temprano obtiene una solución óptima del problema INTERVAL SCHEDULING.

Demostración

Probaremos que existe una solución óptima S_{OPT} que contiene la tarea i con tiempo de finalización f_i mínimo en T . Todas las tareas con $s_j \leq f_i$ son incompatibles la tarea i .

En consecuencia, $S_{\text{OPT}} \setminus \{i\}$ tiene que ser necesariamente una solución óptima para el conjunto de tareas

$$T' = T \setminus (\{i\} \cup \{j \in T : s_j \leq f_i\}),$$

pues de otro modo tendríamos una contradicción \Rightarrow
subestructura óptima

Interval scheduling: Corrección

Demostración (continúa)

Demostraremos que al menos una solución óptima incluye una tarea con tiempo de finalización mínimo, y lo haremos por reducción al absurdo.

Sea i una tarea con mínimo f_i y supongamos que para cualquier solución óptima S_{OPT} , tenemos $i \notin S_{\text{OPT}}$. Consideremos una solución óptima S_{OPT} y sea k una tarea en S_{OPT} con tiempo de finalización mínimo. Por hipótesis, $f_k > f_i$.

Interval scheduling: Corrección

Demostración (continúa)

Toda otra tarea k' en S tiene que empezar después de que k termine, $f_k < s_{k'}$, pues en caso contrario no serían compatibles k y k' (tendríamos, sino, $s_{k'} \leq f_k < f_{k''}!!$). Esto significa que todas las tareas k' también son compatibles con la tarea i (en efecto: $f_i < f_k \leq s_{k'}$)

Por lo tanto $S' = S_{\text{OPT}} \cup \{i\} \setminus \{k\}$ es también una solución óptima ya que $|S'| = |S_{\text{OPT}}|$, pero contiene a la tarea i , en contradicción con la hipótesis inicial. □

Interval scheduling: Coste

```
procedure INTERVALSCHEDULING( $A$ )
     $S := \emptyset$ ;  $T := \{1, \dots, n\}$ ;
    while  $T \neq \emptyset$  do
        Sea  $i$  la tarea con menor  $f_i$  en  $T$ 
         $S := S \cup \{i\}$ ;
        Eliminar  $i$  y todas las tareas  $j \in T$  t.q.  $s_j < f_i$ 
    end while
    return  $S$ 
end procedure
```

En una implementación ingenua hallar la tarea i y su eliminación de T tiene coste $\Theta(n)$ y el coste total es $\Theta(n^2)$.

Interval scheduling: Coste

```
procedure INTERVALSCHEDULING2( $A$ )
```

 Ordenar A en orden ascendente de tiempo de finalización

 ▷ **Sea** $[a_1, \dots, a_n]$ la lista resultante

$S := \emptyset; j := 0; A[0].f := -\infty$

for $i := 1$ **to** n **do**

if $A[i].s \geq A[j].f$ **then**

$S := S \cup \{i\}; j := i$

end if

end for

return S

end procedure

El coste de INTERVALSCHEDULING2 es $\Theta(n \log n)$ correspondiente a la ordenación; el bucle **for** tiene coste $\Theta(n)$.

Interval scheduling: Coste

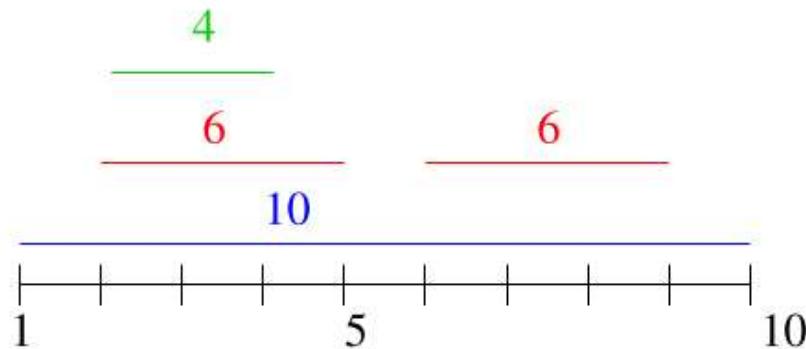
Las tareas se consideran por tiempo de finalización de menor a mayor (según el criterio voraz que conduce al óptimo); en todo momento j es el índice de la última tarea seleccionada y si la tarea $A[i]$, con $i > j$, cumple $A[i].s < A[j].f$ entonces i y j son incompatibles. En caso contrario, la tarea i es la tarea con tiempo de finalización mínimo compatible con las tareas ya seleccionadas.

Por lo tanto INTERVAL2SCHEDULING nos devolverá una solución óptima en tiempo $\Theta(n \log n)$.

Si los tiempos de finalización están dentro de un rango no muy grande podríamos emplear un método de ordenación digital para rebajar el coste del algoritmo a $\Theta(n)$.

Weighted Activity Selection

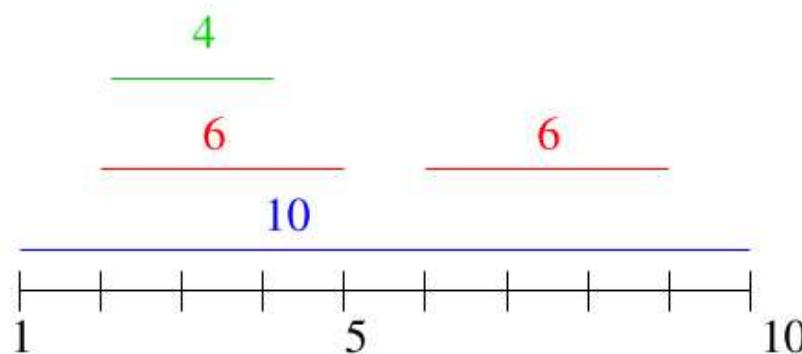
Generaliza el problema anterior: dadas n tareas con **tiempos de inicio** s_i , **tiempos de finalización** f_i y **pesos** w_i , el objetivo es hallar una selección de tareas mutuamente compatibles cuyo peso total es máximo.



INTERVALSCHEDULING2 seleccionaría la tarea en verde y la segunda tarea en rojo con peso total 10, lo cual **no** es óptimo.

Weighted Activity Selection

Si el criterio fuera escoger la tarea con mayor peso mutuamente compatible con las ya seleccionadas, la solución obtenida **tampoco es óptima**



El nuevo algoritmo voraz selecciona la tarea azul con peso 10; la solución óptima es seleccionar las dos tareas rojas con peso total 12.

En este problema fracasa todo criterio voraz. Existen no obstante soluciones “eficientes” con **Programación Dinámica** si los pesos no son muy grandes (tienen $\Theta(\log n)$ bits).

Lateness Minimization

Tenemos n tareas cuya ejecución puede comenzar en cualquier momento, siempre que el procesador esté libre; cada tarea tiene una duración t_i y un tiempo límite d_i (**deadline**). El objetivo de la planificación es hallar tiempos de inicio s_i para todas las tareas de tal modo que:

- 1 Cualesquiera tareas i y j , $i \neq j$, no solapan en su ejecución: $(s_i, f_i) \cap (s_j, f_j) = \emptyset$ donde $f_k = s_k + t_k$, $k = 1, \dots, n$
- 2 Se minimiza el retraso (*lateness*) máximo $L = \max_{1 \leq i \leq n} \ell_i$, donde el retraso de la tarea i es

$$\ell_i = \max(0, f_i - d_i),$$

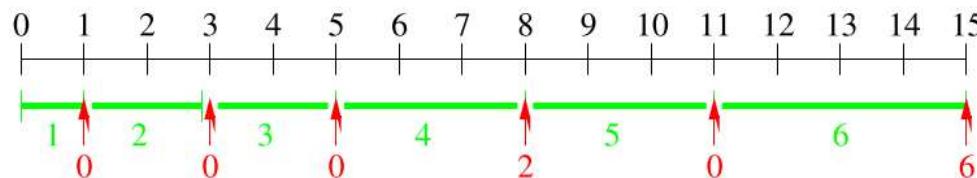
el tiempo en exceso más allá del *deadline* de la tarea.

Lateness Minimization

Ejemplo

t	1	2	2	3	3	4
d	9	8	15	6	14	9
s	0	1	3	5	8	11
ℓ	0	0	0	2	0	6

$$L = 6$$



Lateness Minimization

```
procedure GENERICLATENESS( $A$ )
    Ordenar  $A$  de acuerdo al criterio  $X$ 
     $S[1] := 0; t := A[1].t; L := \max(0, t - A[1].d);$ 
    for  $i := 2$  to  $n$  do
         $S[i] := t;$ 
         $t := t + A[i].t;$ 
         $L := L + \max(0, t - A[i].d));$ 
    end for
    return  $\langle S, L \rangle$ 
end procedure
```

Lateness Minimization

Criterio #1: tareas con menor duración primero

i	t_i	d_i
1	1	6
2	5	5

$s_1 = 0, s_2 = 1$ tiene retraso $L = 1$, pero
 $s_1 = 5, s_2 = 0$ tiene menor retraso $L = 0!!$

Lateness Minimization

Criterio #2: tareas con tiempo de inicio sin retraso más tardío
(equivale a $d_i - t_i$ mínimo)

i	t_i	d_i	$d_i - t_i$
1	1	3	2
2	9	10	1

$s_2 = 0, s_1 = 9$ no minimiza el retraso

Lateness Minimization

Criterio #3: tareas con *deadline* más temprano (=las tareas más urgentes primero)

Ordenar por d_i creciente

procedure LATENESSURGENT(A)

 Ordenar A por d_i creciente

$S[1] := 0; t := A[1].t;$

$L := \max(0, t - A[1].d);$

for $i := 2$ **to** n **do**

$S[i] := t; t := t + A[i].t;$

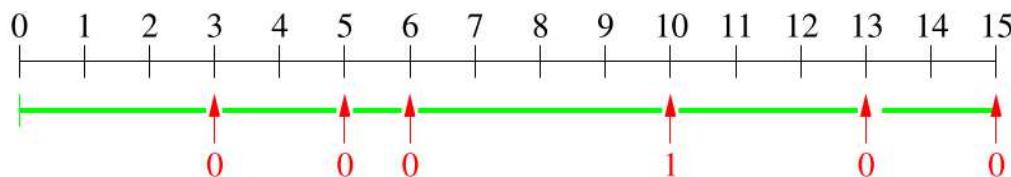
$L := \max(L, \max(0, t - A[i].d))$

end for

return $\langle S, L \rangle$

end procedure

i	t	d	nuevo i
1	1	9	3
2	2	8	2
3	2	15	6
4	3	6	1
5	3	14	5
6	4	9	4



Urgentes Primero: Coste

```
procedure LATENESSURGENT( $A$ )
    Ordenar  $A$  por  $d_i$  creciente
     $S[1] := 0; t := A[1].t; L := \max(0, t - A[0].d);$ 
    for  $i := 2$  to  $n$  do
         $s_i := t; t := t + A[i].t;$ 
         $L := \max(L, \max(0, t - A[i].d))$ 
    end for
    return  $\langle S, L \rangle$ 
end procedure
```

Coste del bucle: $\Theta(n)$

Coste total: $\Theta(n \log n)$

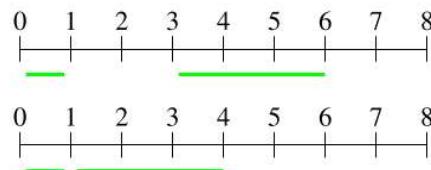
Urgentes Primero: Corrección

Lema

Existe una planificación óptima de las tareas de manera que en todo instante de tiempo hay alguna tarea en ejecución.

Demostración

Para cualquier planificación en la que haya momentos “ociosos” (ninguna tarea en ejecución) podemos encontrar una planificación alternativa eliminando los “huecos” de modo que no haya instantes de tiempo sin ninguna tarea en ejecución y el retraso total sea el mismo o menor.



N.B.: LATENESS_{URGENT} genera una planificación sin momentos ociosos.

Urgentes Primero: Corrección

Diremos que una planificación S tiene una inversión si $s_i < s_j$ pero $d_i > d_j$.

Si indexamos las tareas de manera que $d_0 \leq d_1 \leq \dots \leq d_{n-1}$ entonces la planificación generada por LATENESSURGENT no tiene inversiones. Dando por sentado que todas las tareas tienen duración no nula ($t_i > 0$) tendremos que en la planificación de LATENESSURGENT cumple

$$s_0 < s_1 < \dots < s_{n-1}.$$

Lema

Dada una planificación S que contiene inversiones, el intercambio de dos tareas consecutivas invertidas reduce el número de inversiones en una unidad pero no incrementa el retraso total.

Urgentes Primero: Corrección

Demostración

Consideremos que en S la tarea i se ejecuta justo antes que la tarea j (son las tareas k -ésima y $(k + 1)$ -ésima en S , respectivamente) y que $d_i > d_j$, esto es, forman una inversión.

Nota: Si indexamos por d creciente tendremos que la primera tarea ejecutada es la 1, a continuación la 2, etc. Si hay una inversión habrá alguna tarea $j \neq i$ que se ejecuta la i -ésima. Sea i el menor índice para el que eso ocurre. Entonces deducimos que $j > i$ y $s_j < s_i$ puesto que la tarea i se tendrá que ejecutar más tarde; pero $d_i < d_j$. Si S tiene inversiones siempre existirá al menos una inversión que involucra a dos tareas que se ejecutan consecutivamente.

Urgentes Primero: Corrección

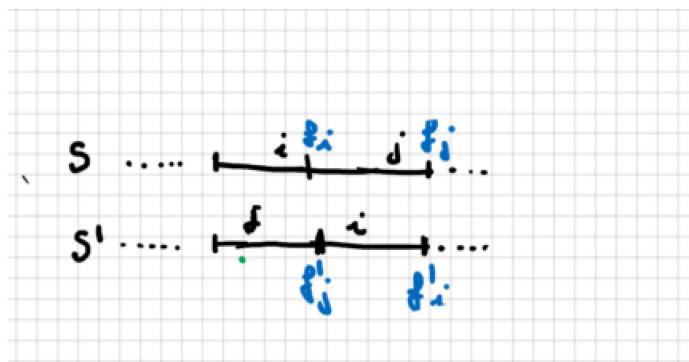
Demostración (continúa)

Si intercambiamos el orden de ejecución de las tareas i y j obtenemos una nueva planificación S' en la que $s'_m = s_m$ excepto si $m = i$ o $m = j$, y por tanto, los retrasos ℓ'_m en S' no cambian salvo que $m \notin \{i, j\}$, i.e., $\ell'_m = \ell_m$. Solo tendremos que ℓ'_i y ℓ'_j pueden cambiar y ser diferentes de ℓ_i y ℓ_j , respectivamente.

Urgentes Primero: Corrección

Demostración (continúa)

- Sean $f_i = s_i + t_i$, $f_j = s_j + t_j$ los tiempos de finalización en S y $f'_i = s'_i + t_i$, $f'_j = s'_j + t_j$ los tiempos de finalización en S' .
- $s_j = f_i$ y $f_i < f_j$; $s'_i = f'_j$ y $f'_j < f'_i$
- $s'_j = s_i$ y $f'_i = f_j$ ($\Rightarrow f'_j < f_j$)
- $f_j \leq d_j$ (recordemos: $d_j < d_i$)



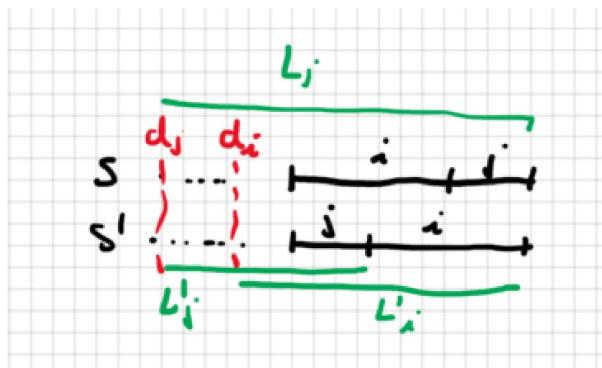
$$f'_j < f_j = f'_i \leq d_j < d_i \wedge f_i < f_j < d_i \implies \ell'_i = \ell'_j = \\ \ell_i = \ell_j = 0$$

S y S' tienen el mismo retraso máximo

Urgentes Primero: Corrección

Demostración (continúa)

- Sean $f_i = s_i + t_i$, $f_j = s_j + t_j$ los tiempos de finalización en S y $f'_i = s'_i + t_i$, $f'_j = s'_j + t_j$ los tiempos de finalización en S' .
- $s_j = f_i$ y $f_i < f_j$; $s'_i = f'_j$ y $f'_j < f'_i$
- $s'_j = s_i$ y $f'_i = f_j$ ($\Rightarrow f'_j < f_j$)
- Si $d_i < f_i$ (recordemos: $d_j < d_i$)



$$l'_j = \max(0, f'_j - d_j) \leq \max(0, f_j - d_j) = l_j$$

$$\begin{aligned} l'_i &= \max(0, f'_i - d_i) = \max(0, f_j - d_i) \leq \\ &\max(0, f_j - d_j) = l_j \end{aligned}$$

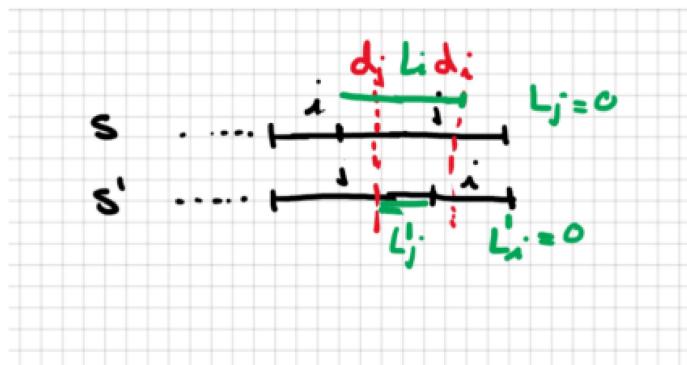
$$\max\{l'_i, l'_j\} \leq l_j \leq \max\{l_i, l_j\} \implies L' \leq L$$

S' tiene el mismo o menor retraso máximo que S

Urgentes Primero: Corrección

Demostración (continúa)

- Sean $f_i = s_i + t_i$, $f_j = s_j + t_j$ los tiempos de finalización en S y $f'_i = s'_i + t_i$, $f'_j = s'_j + t_j$ los tiempos de finalización en S' .
- $s_j = f_i$ y $f_i < f_j$; $s'_i = f'_j$ y $f'_j < f'_i$
- $s'_j = s_i$ y $f'_i = f_j$ ($\Rightarrow f'_j < f_j$)
- Si $f_i \leq d_i \leq f_j$ (recordemos: $d_j < d_i$)



$$\ell'_j = f'_j - d_j \leq f_j - d_j = \ell_j$$

$$\ell'_i = f'_i - d_i = f_j - d_i \leq f_j - d_j = \ell_j$$

$$\max\{\ell'_i, \ell'_j\} \leq \ell_j = \max\{\ell_i, \ell_j\} \Rightarrow L' \leq L$$

S' tiene el mismo o menor retraso máximo que S

Urgentes Primero: Corrección

Demostración (continúa)

En los tres casos posibles, el retraso total de S' es menor o igual que el de S .



Urgentes Primero: Conclusión

Teorema

El algoritmo LATENESSURGENT obtiene una planificación óptima con retraso total mínimo en tiempo $\mathcal{O}(n \log n)$

Demostración

Si consideramos una planificación S_{opt} que tiene inversiones y menor retraso total que LATENESSURGENT llegamos a una contradicción.

Podemos eliminar todo momento ocioso de la planificación (por el primer lema) e ir eliminando sucesivamente todas las inversiones manteniendo el retraso total óptimo (por el lema anterior); no podemos mejorarlo ya que asumimos que el retraso de S_{opt} es mínimo.

Urgentes Primero: Conclusión

Demostración (continúa)

Tras eliminar todas las inversiones habremos llegado a una planificación sin momentos ociosos y que no contiene ninguna inversión, es decir, que asigna los tiempos de inicio por orden creciente de deadline, esto es, la planificación que nos da LATENESS_{URGENT} y el retraso total es idéntico al de S_{opt} !



Parte III

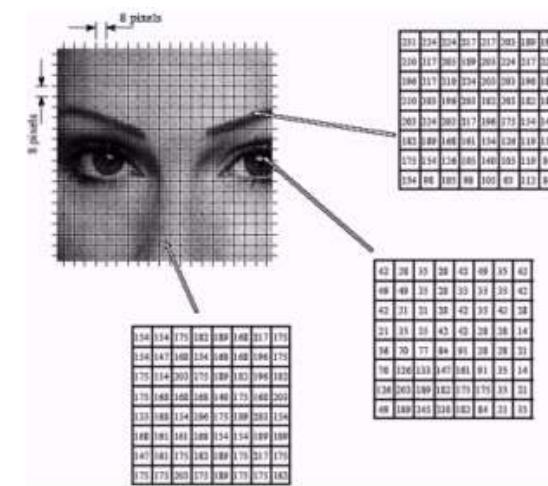
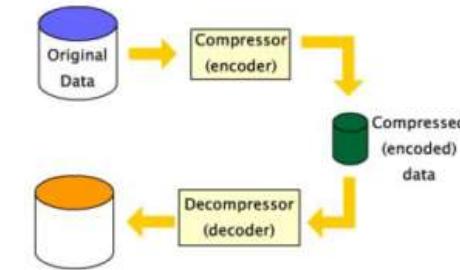
Algoritmos Voraces

- Introducción a los Algoritmos Voraces
- Compresión de Datos: Códigos de Huffman
- Árboles de Expansión Mínimos: Algoritmos de Kruskal y de Prim
- Particiones

Compresión de Datos

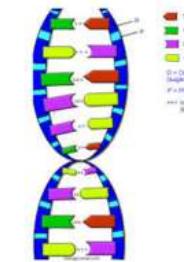
Nos dan un texto T que utiliza símbolos de un alfabeto finito Σ . Nuestro objetivo es representar T con el mínimo número de bits posible.

La **compresión de datos** (*data compression*) tiene como finalidad reducir el tiempo de transmisión de archivos grandes y/o reducir el espacio de almacenamiento. Si usamos una codificación de longitud variable necesitamos un sistema sencillo de codificación y decodificación \implies **códigos prefijos** (*prefix codes*), en los que ningún código es prefijo de otro código



Example.

AAACAGTTGCAT ... GGTCCCTAGG
130,000,000



- Codificación de longitud fija: $A = 00$, $C = 01$, $G = 10$ and $T = 11$. Se necesitan 260 millones de bits (≈ 260 MB)
- Codificación de longitud variable: supongamos que A aparece $70 \cdot 10^6$ veces, C aparece $3 \cdot 10^6$ veces, G aparece $20 \cdot 10^6$ y T aparece $37 \cdot 10^7$ veces, es mejor asignar un código corto a A y códigos más largos a C y G , p.e., $A = 0$, $T = 10$, $G = 110$, $C = 111$.

Códigos prefijos

Definición

Dado un alfabeto (conjunto de símbolos) Σ , un **código prefijo** ϕ asigna una cadena de bits a cada símbolo $x \in \Sigma$ y se cumple que, para cualesquiera símbolos distintos $x, y \in \Sigma$, $\phi(x)$ no es un prefijo de $\phi(y)$.

Ejemplo

- Si $\phi(A) = 1$ y $\phi(C) = 101 \implies \phi$ no es un código prefijo!
- Si $\phi(A) = 1$, $\phi(T) = 01$, $\phi(G) = 000$ y $\phi(C) = 001$ entonces ϕ es un código prefijo
- Los códigos prefijos son fáciles de decodificar, sin necesidad de *backtracking*, de forma única de izquierda a derecha:



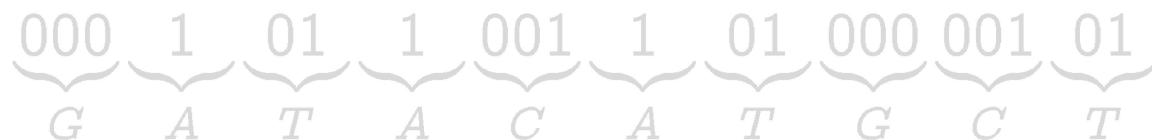
Códigos prefijos

Definición

Dado un alfabeto (conjunto de símbolos) Σ , un **código prefijo** ϕ asigna una cadena de bits a cada símbolo $x \in \Sigma$ y se cumple que, para cualesquiera símbolos distintos $x, y \in \Sigma$, $\phi(x)$ no es un prefijo de $\phi(y)$.

Ejemplo

- Si $\phi(A) = 1$ y $\phi(C) = 101 \implies \phi$ no es un código prefijo!
- Si $\phi(A) = 1$, $\phi(T) = 01$, $\phi(G) = 000$ y $\phi(C) = 001$ entonces ϕ es un código prefijo
- Los códigos prefijos son fáciles de decodificar, sin necesidad de *backtracking*, de forma única de izquierda a derecha:



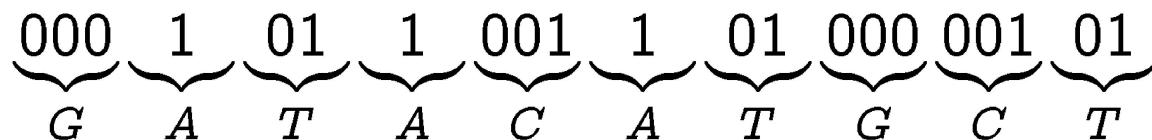
Códigos prefijos

Definición

Dado un alfabeto (conjunto de símbolos) Σ , un **código prefijo** ϕ asigna una cadena de bits a cada símbolo $x \in \Sigma$ y se cumple que, para cualesquiera símbolos distintos $x, y \in \Sigma$, $\phi(x)$ no es un prefijo de $\phi(y)$.

Ejemplo

- Si $\phi(A) = 1$ y $\phi(C) = 101 \implies \phi$ no es un código prefijo!
- Si $\phi(A) = 1$, $\phi(T) = 01$, $\phi(G) = 000$ y $\phi(C) = 001$ entonces ϕ es un código prefijo
- Los códigos prefijos son fáciles de decodificar, sin necesidad de *backtracking*, de forma única de izquierda a derecha:



Árbol de prefijos

Un código prefijo puede representarse mediante un árbol binario etiquetado (de hecho, un *trie*).

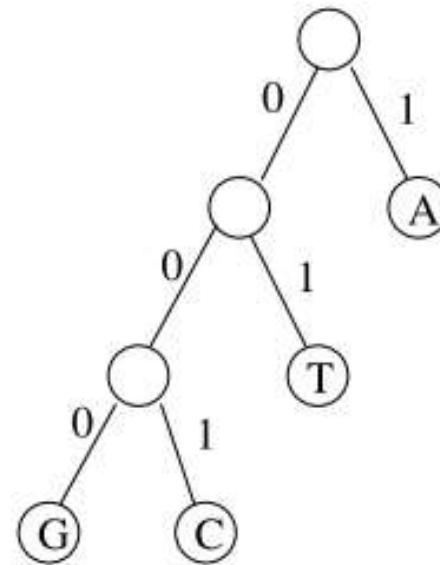
Definición

Un árbol de prefijos T (*prefix tree*) es un árbol binario que cumple las siguientes propiedades:

- Cada símbolo de Σ distinto se asocia a una hoja (nodo sin descendientes)
- La arista que une a un nodo con la raíz de su subárbol izquierdo se etiqueta 0, la arista al subárbol derecho se etiqueta 1
- Las etiquetas en el camino de la raíz a una hoja especifican el código asignado al símbolo asociado la hoja.

Árbol de prefijos

Σ	ϕ
A	1
T	01
G	000
C	001



Longitud de la codificación

- Dado un texto S de longitud $|S| = n$ escrito con símbolos de Σ y un código prefijo ϕ , denotaremos $B(S)$ a la longitud del texto codificado
- Sea $f(x, S)$ la frecuencia relativa de $x \in \Sigma$ en el texto S (Nota: $\sum_{x \in \Sigma} f(x, S) = 1$). Si queda claro por el contexto cuál es el texto S , escribiremos $f(x)$ en vez de $f(x, S)$.
- Entonces $f(x) \cdot n$ es la frecuencia absoluta de x en S y

$$B(S) = \sum_{x \in \Sigma} n \cdot f(x) \cdot |\phi(x)| = n \cdot \sum_{x \in \Sigma} f(x) \cdot |\phi(x)|$$

- $\alpha(S) = \sum_{x \in \Sigma} f(x) \cdot |\phi(x)|$ es la longitud media del código ϕ o número promedio de bits por símbolo, y también se le conoce como factor de compresión.

Longitud de la codificación

- Dado un texto S de longitud $|S| = n$ escrito con símbolos de Σ y un código prefijo ϕ , denotaremos $B(S)$ a la longitud del texto codificado
- Sea $f(x, S)$ la frecuencia relativa de $x \in \Sigma$ en el texto S (**Nota:** $\sum_{x \in \Sigma} f(x, S) = 1$). Si queda claro por el contexto cuál es el texto S , escribiremos $f(x)$ en vez de $f(x, S)$.
- Entonces $f(x) \cdot n$ es la frecuencia absoluta de x en S y

$$B(S) = \sum_{x \in \Sigma} n \cdot f(x) \cdot |\phi(x)| = n \cdot \sum_{x \in \Sigma} f(x) \cdot |\phi(x)|$$

- $\alpha(S) = \sum_{x \in \Sigma} f(x) \cdot |\phi(x)|$ es la **longitud media** del código ϕ o **número promedio de bits por símbolo**, y también se le conoce como **factor de compresión**.

Longitud de la codificación

- Dado un texto S de longitud $|S| = n$ escrito con símbolos de Σ y un código prefijo ϕ , denotaremos $B(S)$ a la longitud del texto codificado
- Sea $f(x, S)$ la frecuencia relativa de $x \in \Sigma$ en el texto S (**Nota:** $\sum_{x \in \Sigma} f(x, S) = 1$). Si queda claro por el contexto cuál es el texto S , escribiremos $f(x)$ en vez de $f(x, S)$.
- Entonces $f(x) \cdot n$ es la frecuencia absoluta de x en S y

$$B(S) = \sum_{x \in \Sigma} n \cdot f(x) \cdot |\phi(x)| = n \cdot \sum_{x \in \Sigma} f(x) \cdot |\phi(x)|$$

- $\alpha(S) = \sum_{x \in \Sigma} f(x) \cdot |\phi(x)|$ es la **longitud media** del código ϕ o **número promedio de bits por símbolo**, y también se le conoce como **factor de compresión**.

Longitud de la codificación

- Dado un texto S de longitud $|S| = n$ escrito con símbolos de Σ y un código prefijo ϕ , denotaremos $B(S)$ a la longitud del texto codificado
- Sea $f(x, S)$ la frecuencia relativa de $x \in \Sigma$ en el texto S (**Nota:** $\sum_{x \in \Sigma} f(x, S) = 1$). Si queda claro por el contexto cuál es el texto S , escribiremos $f(x)$ en vez de $f(x, S)$.
- Entonces $f(x) \cdot n$ es la frecuencia absoluta de x en S y

$$B(S) = \sum_{x \in \Sigma} n \cdot f(x) \cdot |\phi(x)| = n \cdot \sum_{x \in \Sigma} f(x) \cdot |\phi(x)|$$

- $\alpha(S) = \sum_{x \in \Sigma} f(x) \cdot |\phi(x)|$ es la **longitud media** del código ϕ o **número promedio de bits por símbolo**, y también se le conoce como **factor de compresión**.

Longitud de la codificación

- En términos del árbol de prefijos T de ϕ , la longitud $|\phi(x)|$ del código de x es la profundidad $d_T(x)$ de la hoja etiquetada con x en T
- Por lo tanto

$$\alpha(T) = \sum_{x \in \Sigma} f(x) \cdot d_T(x)$$

dicho de otro modo, $\alpha(T)$ es la profundidad promedio de las hojas no vacías (la “probabilidad” de una hoja no vacía es la frecuencia del símbolo asociado a la hoja)

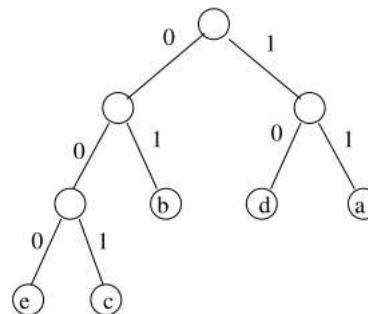
Códigos de longitud variable vs. de longitud fija

Ejemplo

- Sea $\Sigma = \{a, b, c, d, e\}$ y S un texto con alfabeto Σ con las siguientes frecuencias:

$$f(a) = 0,32, f(b) = 0,25, f(c) = 0,20, f(d) = 0,18, f(e) = 0,05$$

- Un código ϕ_0 de longitud fija necesita $\lceil \lg 5 \rceil = 3$ bits por código, de manera que $\alpha(\phi_0) = 3$
- Consideremos el código prefijo ϕ_1 :



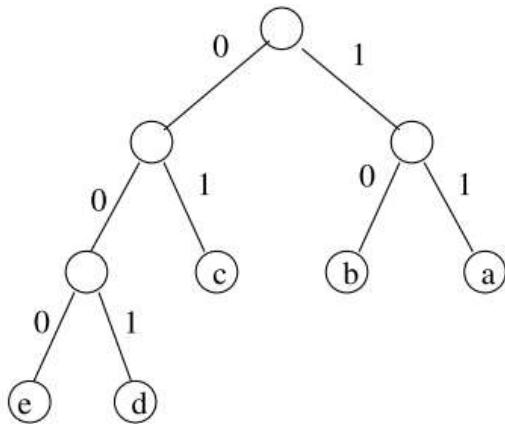
$$\alpha(\phi_1) = 0,32 \cdot 2 + 0,25 \cdot 2 + 0,20 \cdot 3 + 0,18 \cdot 2 + 0,05 \cdot 3 = 2,25$$

- En promedio, ϕ_1 necesita 2.25 bits/símbolo frente a los 3 bits/símbolo del código de longitud fija.

Códigos de longitud variable vs. de longitud fija

¿Es $\alpha = 2,25$ el menor factor de compresión posible? ¿Cuál es la máxima compresión alcanzable?

Consideremos el código prefijo ϕ_2 :



$$\alpha(\phi_2) = 0,32 \cdot 2 + 0,25 \cdot 2 + 0,20 \cdot 2 + 0,18 \cdot 3 + 0,05 \cdot 3 = 2,23$$

¡ $\alpha(\phi_2)$ es mejor que $\alpha(\phi_1)$!

¿Es $\alpha(\phi_2)$ la mínima posible? (\Rightarrow mejor compresión en promedio)

Hay que considerar todos los códigos prefijos posibles para el alfabeto Σ .

Códigos prefijos óptimos

Dado un texto, un **código prefijo óptimo** es un código prefijo ϕ con $\alpha(\phi)$ mínima.

Intuitivamente, en el árbol de prefijos de un código prefijo óptimo, los símbolos muy frecuentes están a poca profundidad, a poca distancia de la raíz (códigos cortos), mientras que los símbolos de muy baja frecuencia habrán de estar a mucha mayor profundidad.

Árboles de prefijos óptimos

Un árbol de prefijos T se dice **completo** si todas sus hojas tienen símbolos asociados.

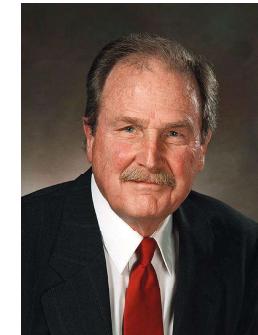
- Si $m = |\Sigma|$ un árbol completo contiene exactamente m hojas y $m - 1$ nodos internos ($2m - 1$ nodos en total). Cada una de las hojas tiene asociado uno de los m símbolos.
- Si un árbol de prefijos no es completo entonces tiene $> 2m - 1$ nodos y algunas de sus hojas **no** tienen asociado ningún símbolo de Σ .

Proposición

El árbol de prefijos que representa a un código prefijo óptimo es completo.

Códigos de Huffman

David Huffman (1925–99) propuso en 1952 un algoritmo voraz eficiente para construir códigos prefijos óptimos.



El algoritmo de Huffman genera un árbol de prefijos completo, con las hojas tan cerca de la raíz como resulte posible, con los símbolos de alta frecuencia a poca profundidad y los símbolos de baja frecuencia más alejados.

Códigos de Huffman

Se nos dan las frecuencias relativas $f(x)$ para todos los símbolos $x \in \Sigma$.

- El algoritmo mantiene una cola de prioridad Q cuyos elementos son nodos de un árbol prefijo y cuyas prioridades son la suma de las $f(x)$ en las hojas del subárbol enraizado en el nodo
- Se construye el árbol de prefijos (conocido como **árbol de Huffman**) de abajo a arriba como sigue:
 - Se insertan m hojas, cada una con un símbolo $x \in \Sigma$ con prioridad $f(x)$
 - Se extraen los dos elementos de prioridad mínima en Q y se crea un nodo cuyos hijos son los dos elementos extraídos y cuya prioridad es la suma de las prioridades. El elemento se inserta en Q .
 - Repetir el paso anterior hasta que solo queda un nodo (raíz del árbol de prefijos) en Q

El árbol de prefijos construído corresponde a un código prefijo óptimo (denominado **código de Huffman**)

Códigos de Huffman

```
procedure HUFFMAN( $\Sigma, f$ )
     $Q := \emptyset;$ 
    for  $x \in \Sigma$  do
        Crear una hoja  $\ell(x)$  con símbolo  $x$  y prioridad  $f(x)$ 
         $Q.\text{INSERT}(\ell(x), f(x));$ 
    end for
    while  $Q.\text{SIZE()} > 1$  do
         $x := Q.\text{EXTRACT-MIN}()$ 
         $y := Q.\text{EXTRACT-MIN}()$ 
        Crear un nuevo nodo  $z$ 
         $z.\text{left} := x; z.\text{right} := y;$ 
         $f(z) := f(x) + f(y)$ 
         $Q.\text{INSERT}(z, f(z))$ 
    end while
    return  $Q.\text{EXTRACT-MIN}()$ 
end procedure
```

Si Q es un heap el algoritmo tiene coste $\mathcal{O}(n \lg n)$.

Códigos de Huffman

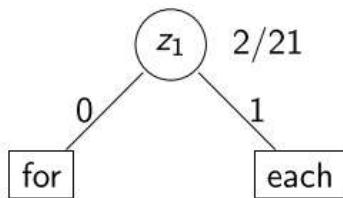
Ejemplo

Consideremos el texto: *for each rose, a rose is a rose, the rose* y el alfabeto $\Sigma = \{\text{for}, \text{each}, \text{rose}, \text{a}, \text{is}, \text{the}, ',', \flat\}$

Frecuencias: $f(\text{for}) = 1/21$, $f(\text{rose}) = 4/21$, $f(\text{is}) = 1/21$, $f(\text{a}) = 2/21$, $f(\text{each}) = 1/21$, $f(',') = 2/21$, $f(\text{the}) = 1/21$, $f(\flat) = 9/21$

Cola de prioridad:

$Q_0 = [(\text{for} : 1/21), (\text{each} : 1/21), (\text{is} : 1/21), (\text{a} : 2/21), (': 2/21), (\text{the} : 2/21), (\text{rose} : 4/21), (\flat : 9/21)]$

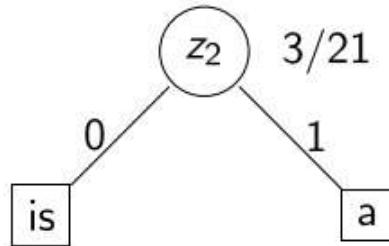


$Q_1 = [(\text{is} : 1/21), (\text{a} : 2/21), (': 2/21), (\text{the} : 2/21), (z_1 : 2/21), (\text{rose} : 4/21), (\flat : 9/21)]$

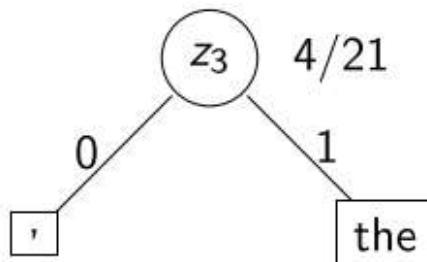
Códigos de Huffman

Ejemplo

$Q_1 = [(\text{is} : 1/21), (\text{a} : 2/21), (': 2/21), (\text{the} : 2/21), (z_1 : 2/21), (\text{rose} : 4/21), (\flat : 9/21)]$



$Q_2 = [(: 2/21), (\text{the} : 2/21), (z_1 : 3/21), (z_2 : 3/21), (\text{rose} : 4/21), (\flat : 9/21)]$

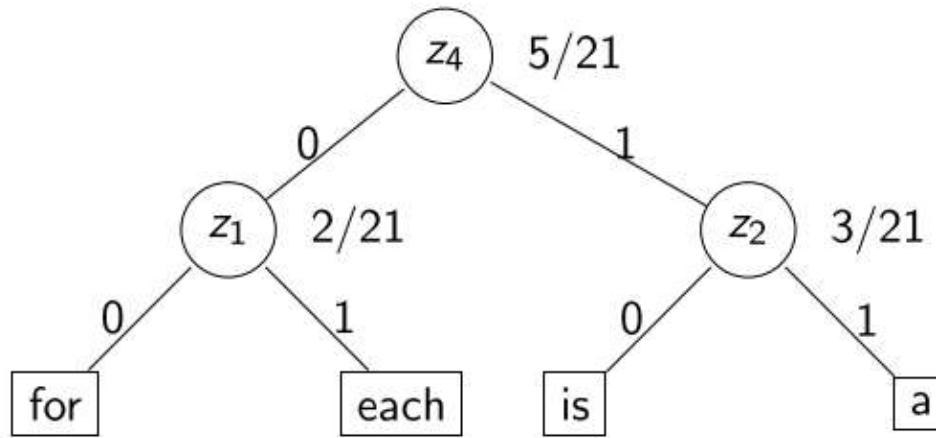


$Q_3 = [(z_1 : 2/21), (z_2 : 3/21), (\text{rose} : 4/21), (z_3 : 4/21), (\flat : 9/21)]$

Códigos de Huffman

Ejemplo

$$Q_3 = [(z_1 : 2/21), (z_2 : 3/21), (\text{rose} : 4/21), (z_3 : 4/21), (\flat : 9/21)]$$

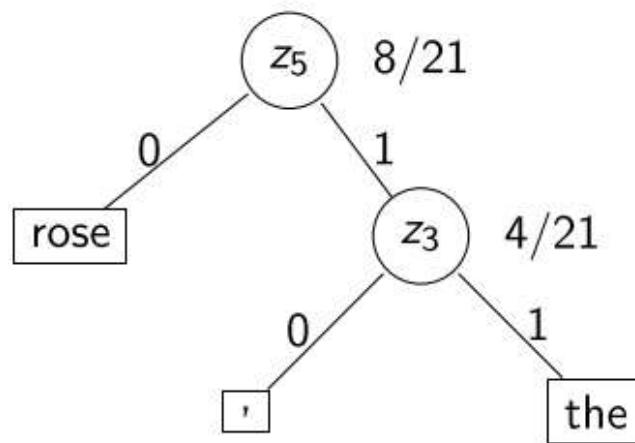


$$Q_4 = [(\text{rose} : 4/21), (z_3 : 4/21), (z_4 : 5/21), (\flat : 9/21)]$$

Códigos de Huffman

Ejemplo

$$Q_4 = [(\text{rose} : 4/21), (z_3 : 4/21), (z_4 : 5/21), (\flat : 9/21)]$$

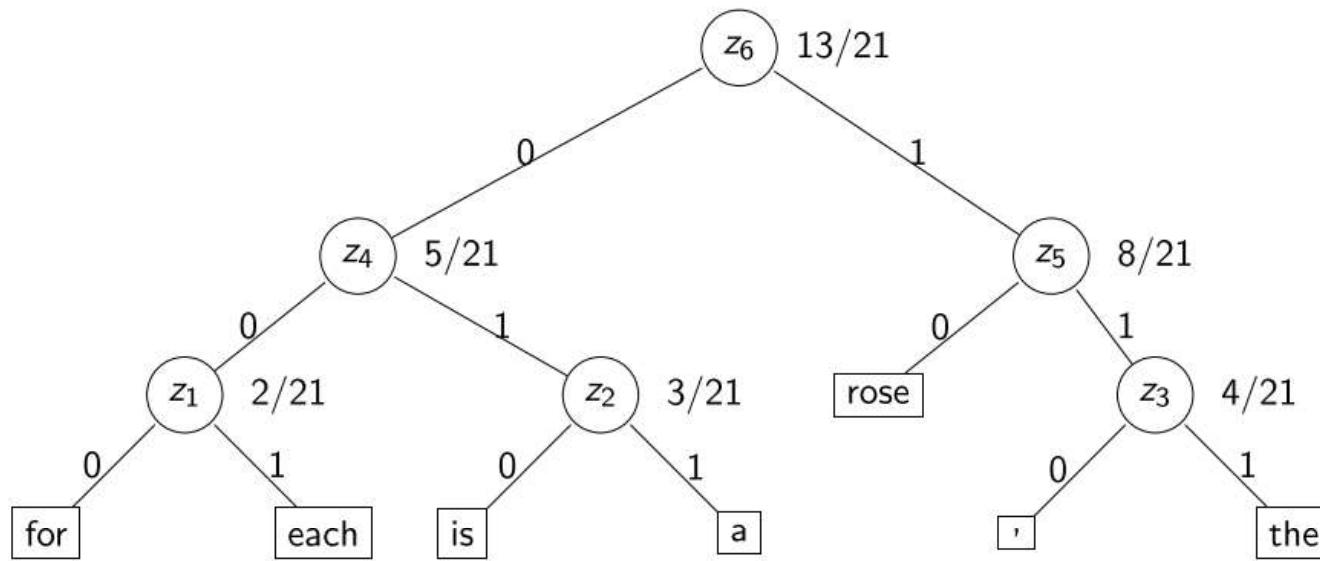


$$Q_5 = [(z_4 : 5/21), (z_5 : 8/21), (\flat : 9/21)]$$

Códigos de Huffman

Ejemplo

$$Q_5 = [(z_4 : 5/21), (z_5 : 8/21), (\text{b} : 9/21)]$$

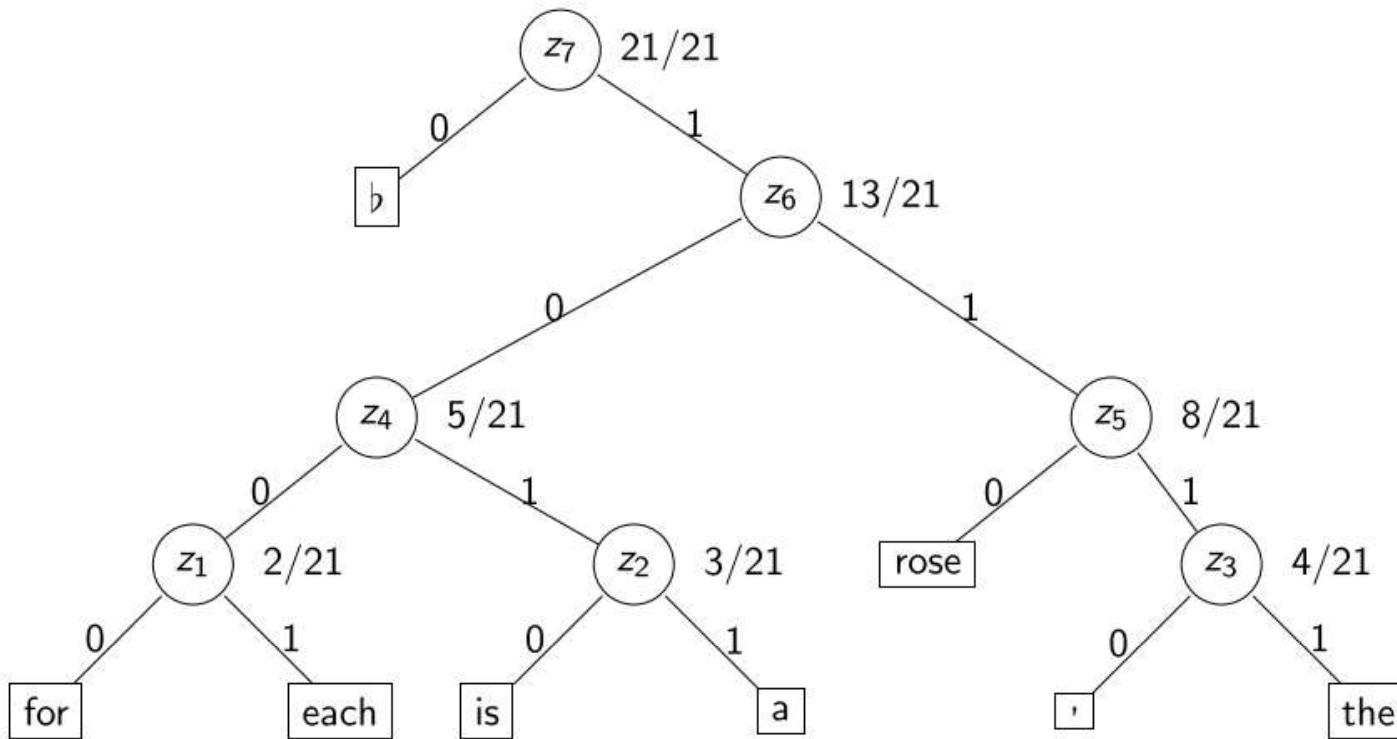


$$Q_6 = [(\text{b} : 9/21), (z_6 : 13/21)]$$

Códigos de Huffman

Ejemplo

$$Q_6 = [(\emptyset : 9/21), (z_6 : 13/21)]$$



$$Q_7 = [(z_7 : 21/21)]$$

Códigos de Huffman

Ejemplo

- El texto *for each rose, a rose is a rose, the rose* se codifica como

10000100101101110010110110010100101101101110011110110
- Existen otros códigos prefijos óptimos (p.e., asignar $z.left := x$ y $z.right := y$ es arbitrario! y hay decisiones arbitrarias a tomar cuando dos nodos x e y tienen igual frecuencia $f(x) = f(y)$)
- La longitud del texto codificado es 53, el factor de compresión es $53/21 = 2,523\dots$.
- Con un código de longitud fija se necesitan 4 bits por símbolo y 84 bits para codificar el texto (en vez de 53).

Códigos de Huffman: Corrección del algoritmo

Teorema (Propiedad de la decisión voraz)

Sea Σ un alfabeto finito y $x, y \in \Sigma$ los dos símbolos con las menores frecuencias. Existe un código prefijo óptimo ϕ tal que $|\phi(x)| = |\phi(y)|$ y $\phi(x)$ y $\phi(y)$ solo difieren en su último bit.

Demostración

Sea T el árbol de prefijo de ϕ , y supongamos que x e y no están a la misma profundidad ($|\phi(x)| \neq |\phi(y)|$). Puesto que T es completo habrá dos hojas con símbolos a y b que están a máxima profundidad y son descendientes de un mismo nodo interno (son hermanas).

Supondremos además, sin pérdida de generalidad, que $f(a) \leq f(b)$ y $f(x) \leq f(y)$.

Construimos un nuevo árbol T' intercambiando $a \leftrightarrow x$ y $b \leftrightarrow y$. Puesto que $f(x) \leq f(a)$ y $d_T(a) = d_{T'}(x) \leq d_T(x)$, y que $f(y) \leq f(b)$ y $d_T(b) = d_{T'}(y) \leq d_T(y)$ deducimos que $B(T') < B(T)$, por tanto T' es óptimo. Y

Códigos de Huffman: Corrección del algoritmo

Teorema (Estructura subóptima)

Sea T' un árbol de prefijos para un código óptimo correspondiente al alfabeto $\Sigma' = \Sigma \setminus \{x, y\} \cup \{z\}$ donde x e y son los símbolos con mínimas frecuencias en Σ y $z \notin \Sigma$ es un nuevo símbolo con frecuencia $f(x) + f(y)$. Si reemplazamos la hoja z en T' por un subárbol con un nodo $\langle z, f(z) \rangle$ que tiene x e y como hijos izquierdo y derecho, respectivamente, obtenemos un nuevo árbol T que es óptimo para Σ .

Demostración

Sea T_0 un árbol de prefijos cualquiera para el alfabeto Σ . Nuestro objetivo es probar que el árbol T descrito en el enunciado cumple $B(T) \leq B(T_0)$.

Códigos de Huffman: Corrección del algoritmo

Demostración (continúa)

Por la propiedad de la decisión voraz solo necesitamos considerar árboles de prefijos T_0 en los que x e y estén en hojas hermanas y su padre z común tiene frecuencia $f(x) + f(y)$.

Sea T'_0 el árbol que se obtiene reemplazando el subárbol enraizado en z por una hoja etiquetada z y con frecuencia $f(z) = f(x) + f(y)$. Entonces T'_0 es un árbol de prefijos para el alfabeto $\Sigma' = \Sigma \setminus \{x, y\} \cup \{z\}$ donde $f(z) = f(x) + f(y)$.

Como T' es el árbol de prefijos óptimo para Σ' sabemos que $B(T') \leq B(T'_0)$.

Códigos de Huffman: Corrección del algoritmo

Demostración (continúa)

Ahora comparamos $B(T_0)$ y $B(T'_0)$ y $B(T)$ con $B(T')$:

$$\begin{aligned}B(T_0) &= B(T'_0) + f(x) + f(y) \\B(T) &= B(T') + f(x) + f(y) \\&\leq B(T'_0) + f(x) + f(y) = B(T_0).\end{aligned}$$



Códigos de Huffman

Los códigos de Huffman son óptimos bajo una serie de supuestos:

- Solo consideramos **compresión sin pérdida** (*lossless compression*), el proceso de codificación y decodificación son únicos, al decodificar se recupera exactamente el texto original
- El alfabeto Σ debe ser conocido de antemano
- Las frecuencias $f(x)$ deben darse de antemano o recolectadas en un primer recorrido de los datos, previo a la construcción del código y la codificación (\Rightarrow puede ser infactible o demasiado lento en determinadas aplicaciones!)
- Un buen lugar para saber más y explorar extensiones de la comprensión de datos mediante códigos de Huffman es este artículo en Wikipedia

[https:](https://en.wikipedia.org/wiki/Huffman_coding)

//en.wikipedia.org/wiki/Huffman_coding

Parte III

Algoritmos Voraces

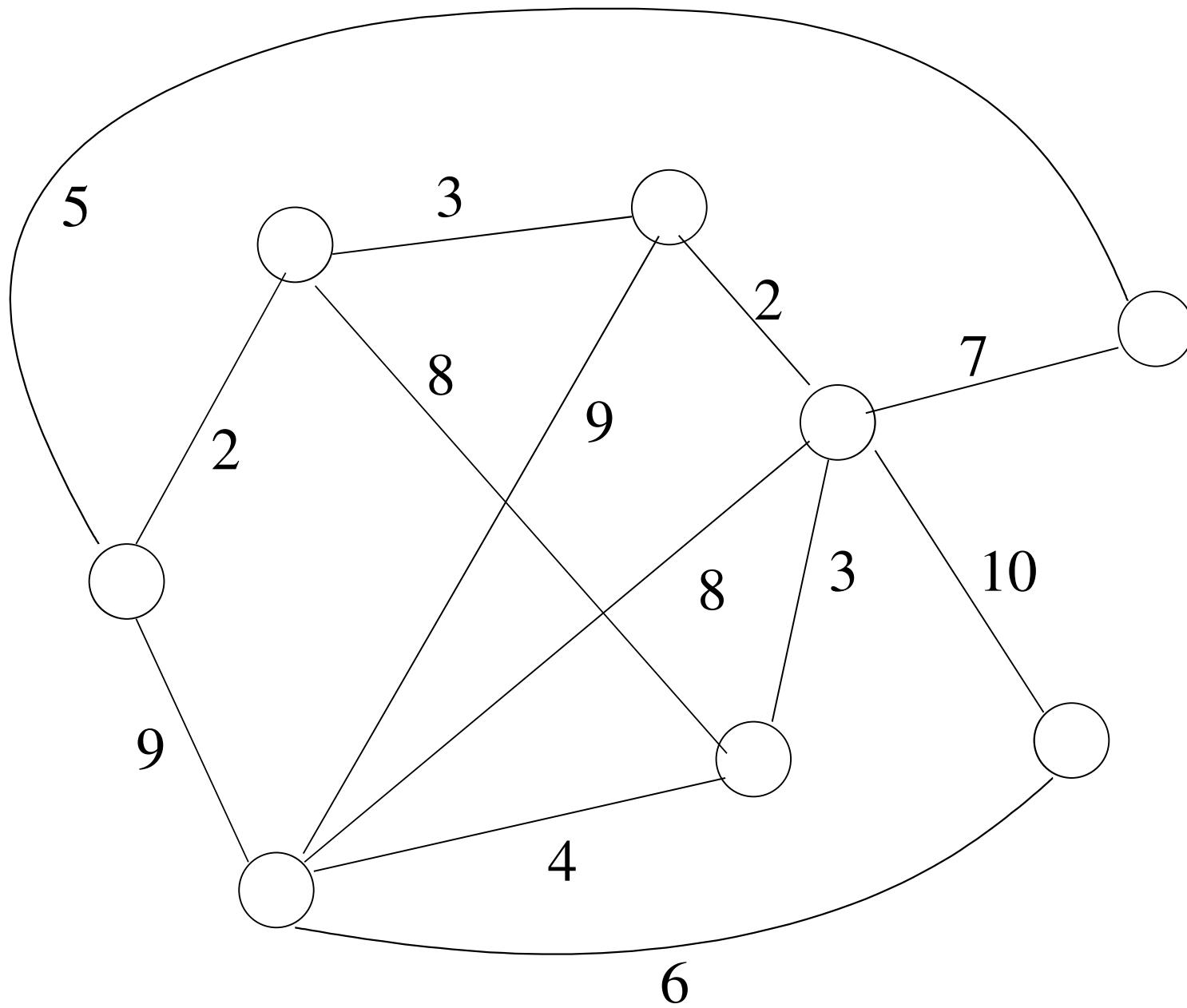
- Introducción a los Algoritmos Voraces
- Compresión de Datos: Códigos de Huffman
- Árboles de Expansión Mínimos: Algoritmos de Kruskal y de Prim
- Particiones

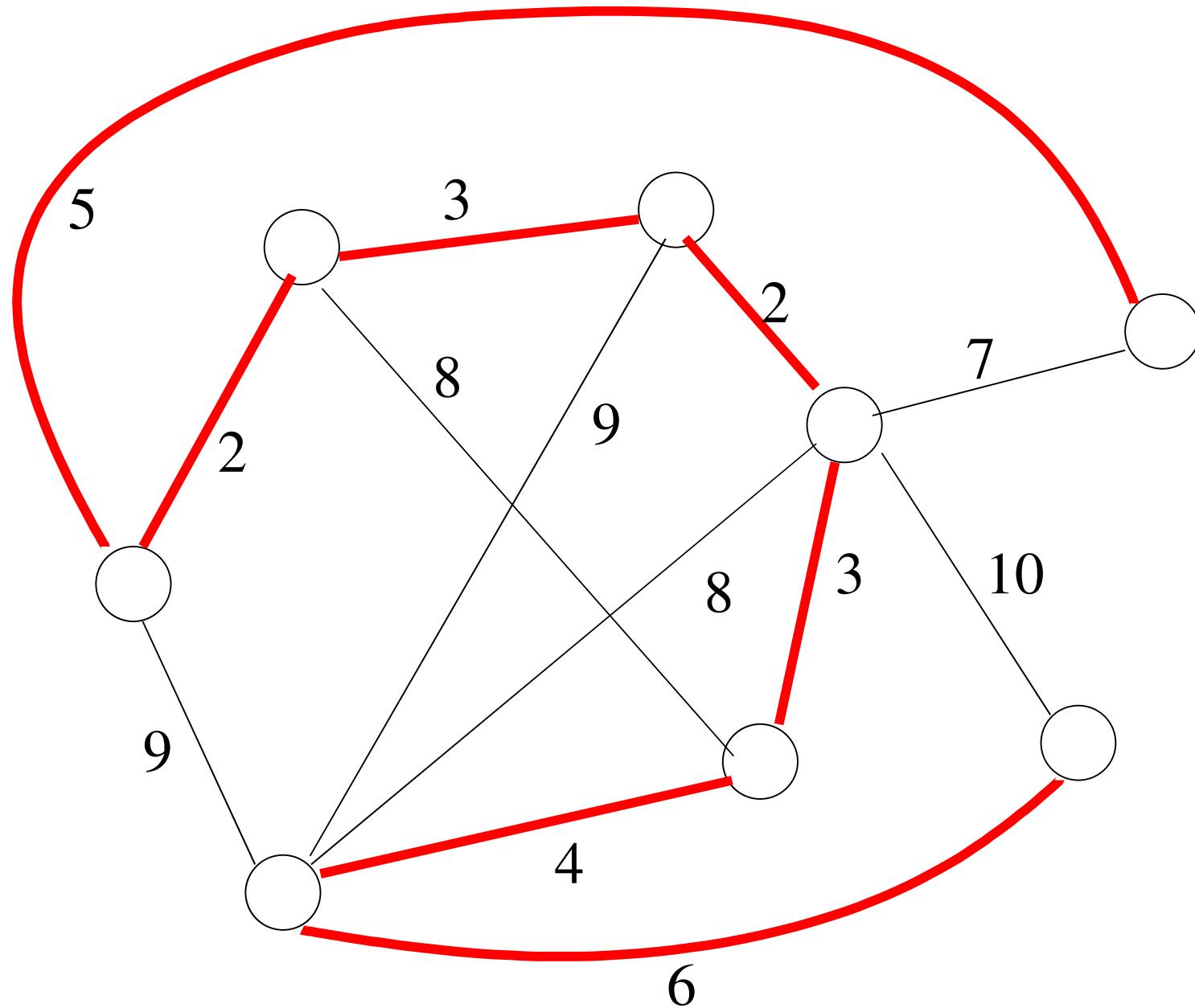
Árboles de Expansión Mínimos

Dado un grafo no dirigido y conexo $G = \langle V, E \rangle$ con pesos o costes en las aristas $\omega : E \rightarrow \mathbb{R}$, un **árbol de expansión mínimo** (*MST: minimum spanning tree*) $T = \langle V, A \rangle$ es un subgrafo de G tal que tiene el mismo conjunto de vértices ($V(T) = V(G)$), es un árbol (es decir, es conexo y acíclico) y su coste

$$\omega(T) = \sum_{e \in A} \omega(e)$$

es mínimo entre todos los posibles árboles de expansión de G .





Existen multitud de algoritmos para calcular un MST de un grafo. No obstante todos ellos siguen un esquema voraz:

```
 $A := \emptyset$ ; Candidatas :=  $E$ 
while  $|A| \neq |V(G)| - 1$  do
    Seleccionar una arista  $e \in \text{Candidatas}$  que
        no crea un ciclo en  $T$ 
     $A := A \cup \{e\}$ 
    Candidatas := Candidatas –  $\{e\}$ 
end while
```

Diremos que un conjunto de aristas $A \subset E(G)$ es **prometedor** si y sólo si:

- 1 A no contiene ciclos
- 2 A es un subconjunto de las aristas de un MST del grafo G

Un **corte** del grafo G es una partición de su conjunto de vértices en dos subconjuntos C y C' no vacíos y disjuntos:

$$V(G) = C \cup C'; \quad C \cap C' = \emptyset$$

Una arista e **respeta** un corte $\langle C, C' \rangle$ si sus dos extremos están ambos en C o ambos en C' , en caso contrario se dice que e **cruza** el corte.

Teorema

Sea A un conjunto prometedor de aristas que respeta un cierto corte $\langle C, C' \rangle$ del grafo G . Sea $e \in E(G)$ la arista de mínimo peso que cruza el corte $\langle C, C' \rangle$.

Entonces

$$A \cup \{e\}$$

es prometedor

El teorema anterior nos da la “receta” para diseñar algoritmos de cálculo de un MST: una vez que hayamos definido cuál es el corte que corresponde a cada iteración, la selección de la arista consistirá en localizar la arista e de mínimo peso que cruza el corte. Por definición, como A respeta el corte y e lo cruza, e no puede crear un ciclo en A .

Más aún: la corrección de los algoritmos que se fundamentan en esta idea queda automáticamente establecida.

Demostración:

Sea A' el conjunto de aristas de un MST T' tal que $A \subset A'$.

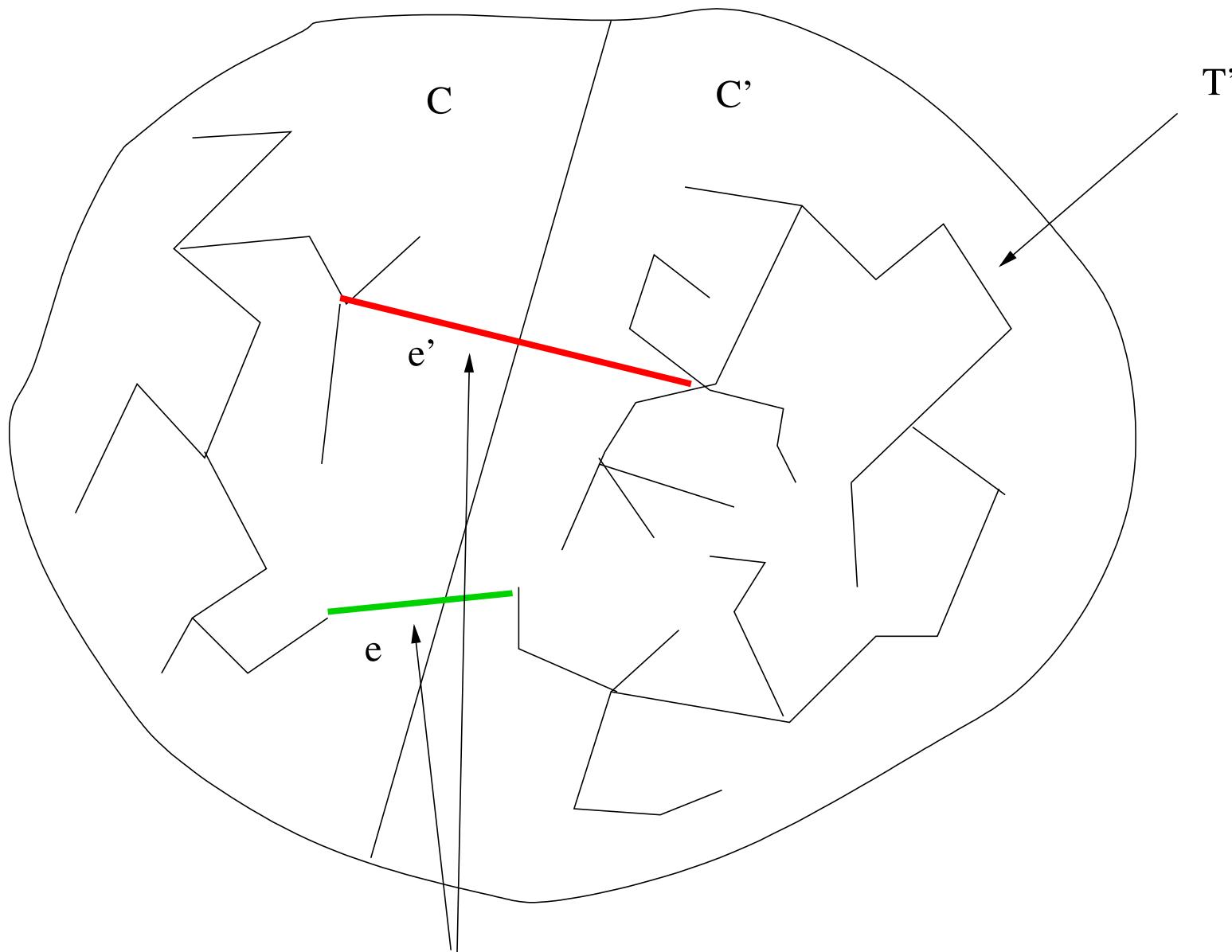
Puesto que A respeta el corte, al menos una arista de las que cruzan el corte tendría que pertenecer a A' , en caso contrario A' no sería conexo. Supongamos que una de dichas aristas es e' . Si e' es la arista de mínimo peso que cruza el corte, como $A \cup \{e'\}$ es prometedor, hemos demostrado el teorema. ¿Pero que ocurre si e' no es la arista de mínimo peso que cruza el corte?

Demostración (cont.):

El coste del MST T' incluirá el coste de las aristas de A , el coste $\omega(e')$ y el coste de otras aristas. Si agregásemos a T' la arista e de mínimo peso que cruza el corte, crearíamos un ciclo, porque T' es un árbol. Sea e' la arista de T' que cruza el corte forma parte de dicho ciclo. De manera que

$$T = T' \cup \{e\} - \{e'\}$$

también es un árbol de expansión. Y su coste es menor o igual que el de T' puesto que sólo cambiamos el coste de e' por el de e , que es el mínimo. Como T' es MST, llegamos a una contradicción a menos que e y e' tengan el mismo coste, y por tanto T y T' tendrían el mismo coste. El teorema queda demostrado, ya que $A \cup \{e\}$ es un subconjunto de las aristas de T , que es un MST.



we can replace e' in T' by e to obtain a new spanning tree with smaller cost!!

Algoritmo de Prim

En el algoritmo de Prim, se mantiene un conjunto de vértices $Vistos \subset V(G)$, y en cada etapa del algoritmo se selecciona la arista de mínimo peso que une a un vértice u de $Vistos$ con un vértice v no visto (i.e., en $V(G) - Vistos$).

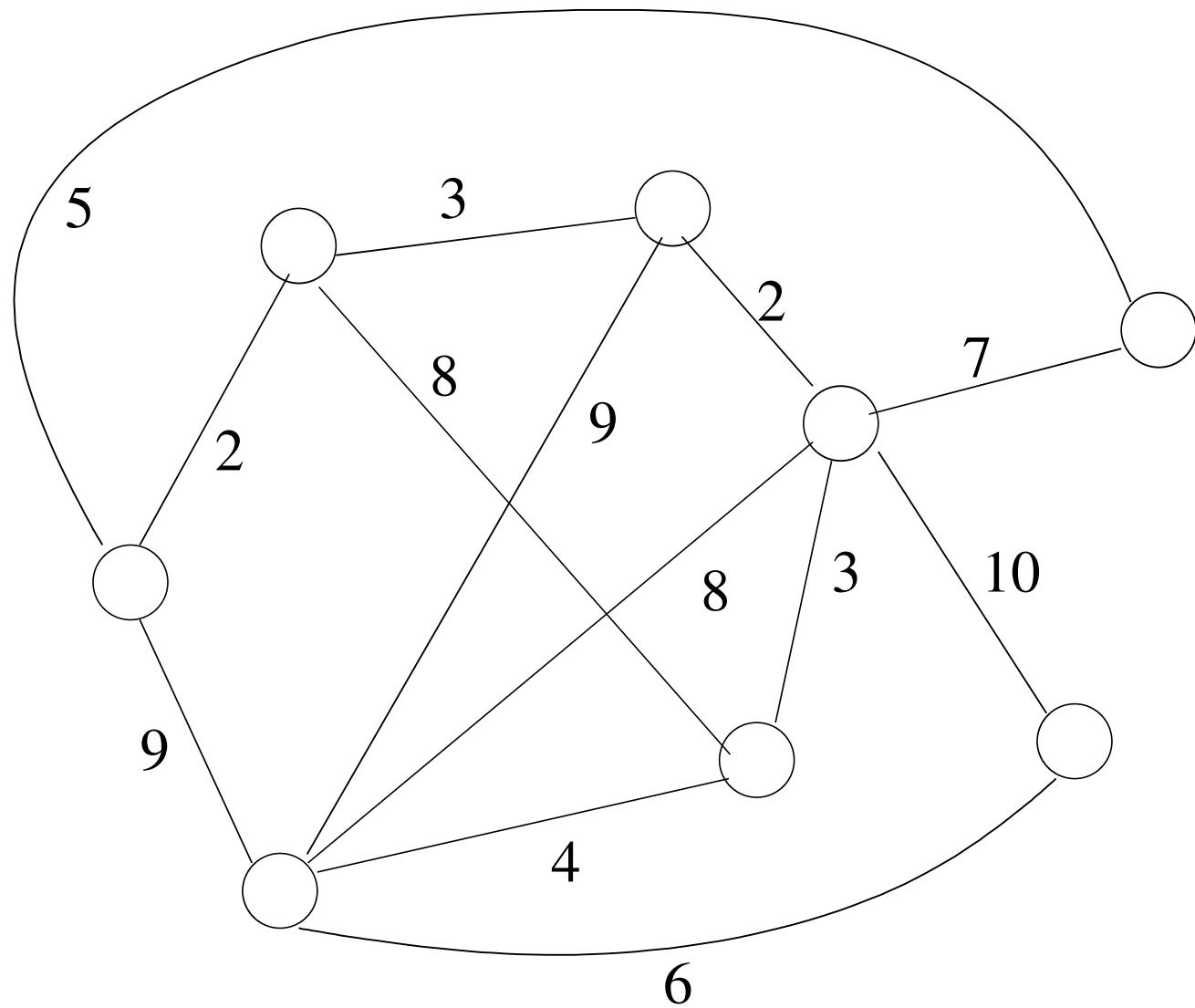
Dicha arista no puede crear un ciclo y se añade al conjunto A . Asimismo el vértice v pasa a ser de $Vistos$.

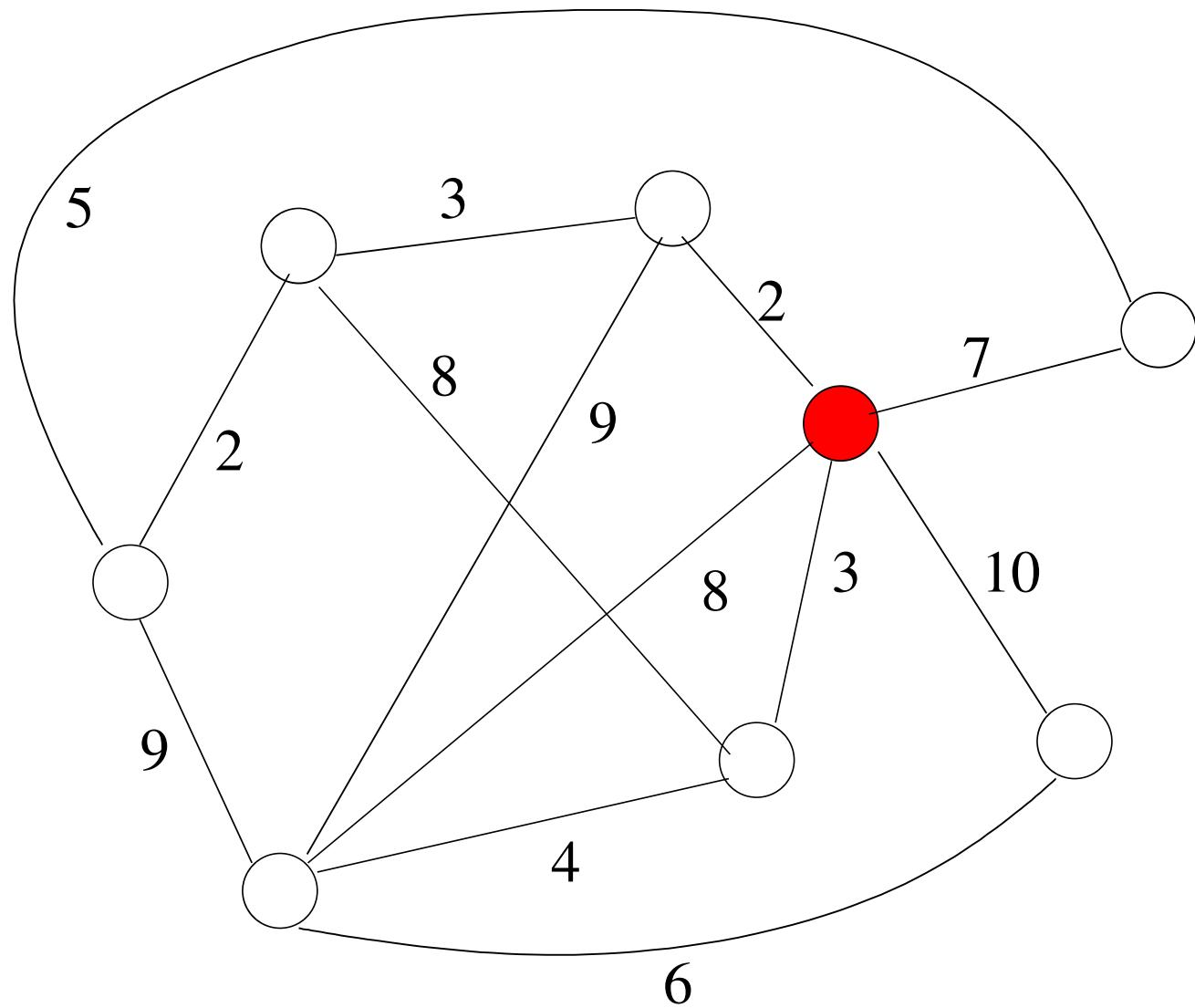
La corrección del algoritmo es inmediata; el conjunto A respeta el corte $\langle Vistos, V(G) - Vistos \rangle$ y seleccionamos la arista de mínimo peso que cruza el corte para agregarla a A .

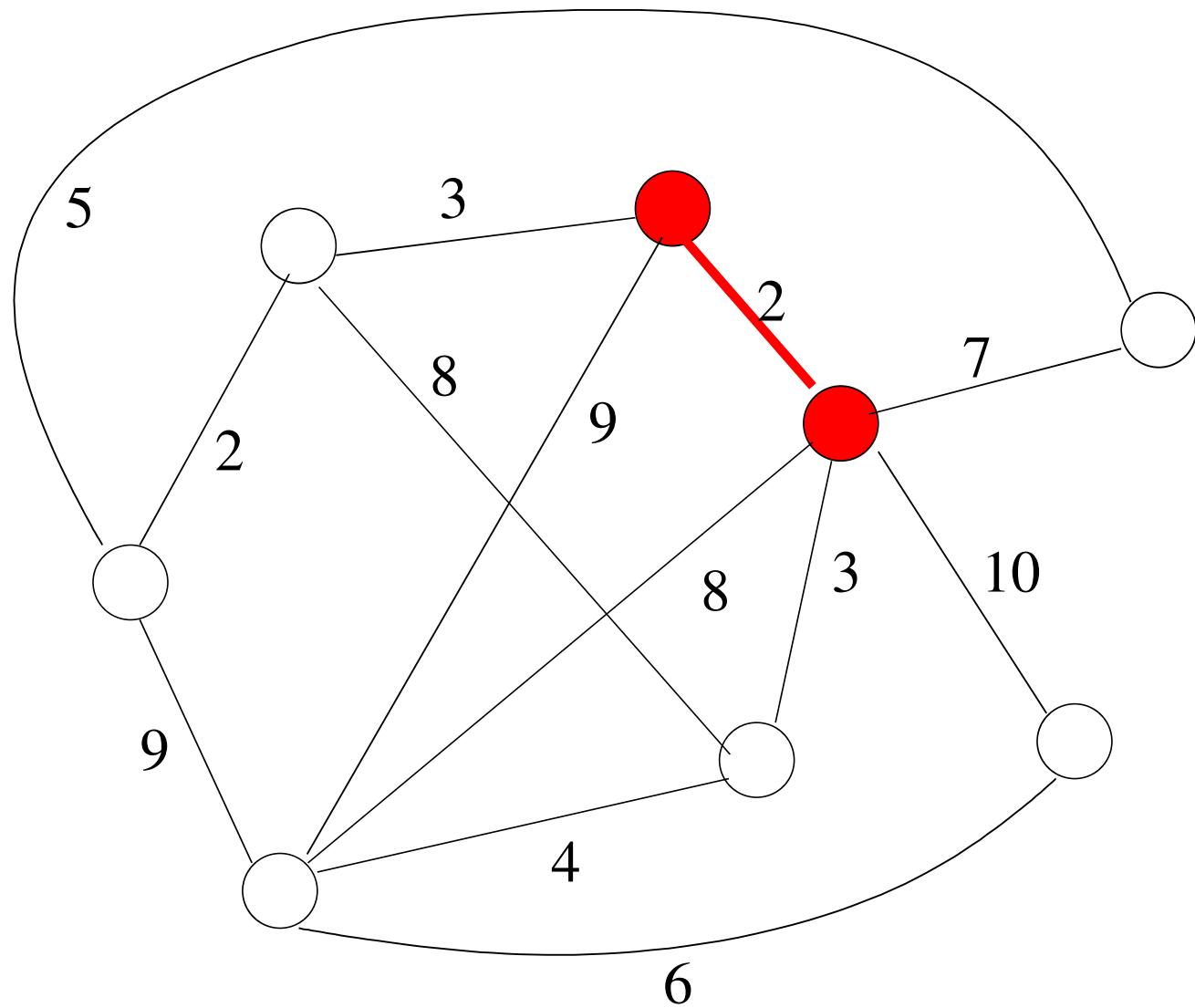
```

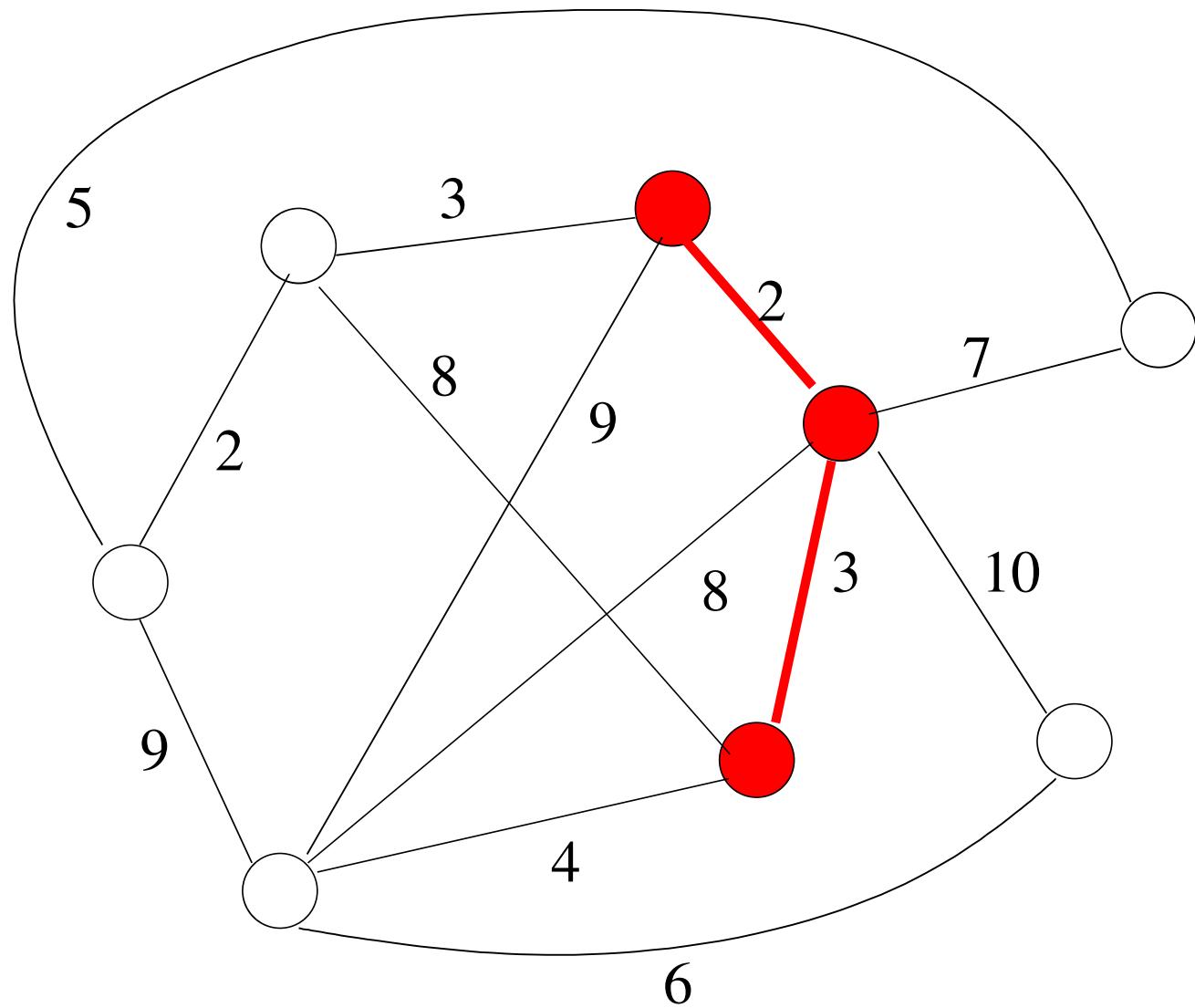
procedure PRIM( $G$ )
     $A := \emptyset$ ;  $Vistos := \{s\}$ 
     $Candidatas := \emptyset$ 
    for  $v \in G.\text{ADYACENTES}(s)$  do
         $Candidatas := Candidatas \cup \{(s, v)\}$ 
    end for
    while  $|A| \neq |V(G)| - 1$  do
        Seleccionar la arista  $e = (u, v)$ 
        de  $Candidatas$  de mínimo peso
         $A := A \cup \{e\}$ 
        Sea  $u$  el vértice en  $Vistos$ :
         $Vistos := Vistos \cup \{v\}$ 
        for  $w \in G.\text{ADYACENTES}(v)$  do
            if  $w \notin Vistos$  then
                 $Candidatas := Candidatas \cup \{(v, w)\}$ 
            end if
        end for
    end while
    return  $A$ 
end procedure

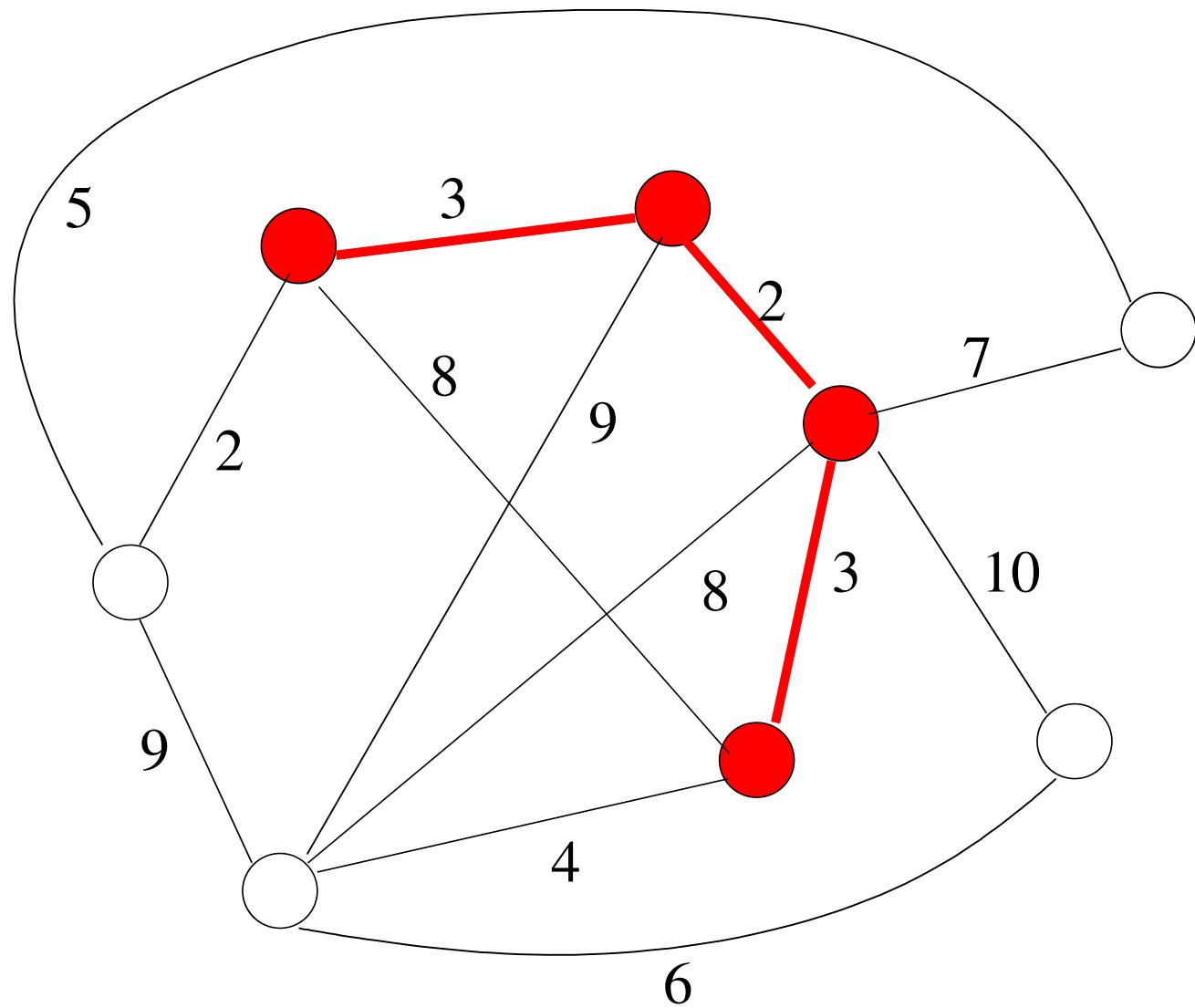
```

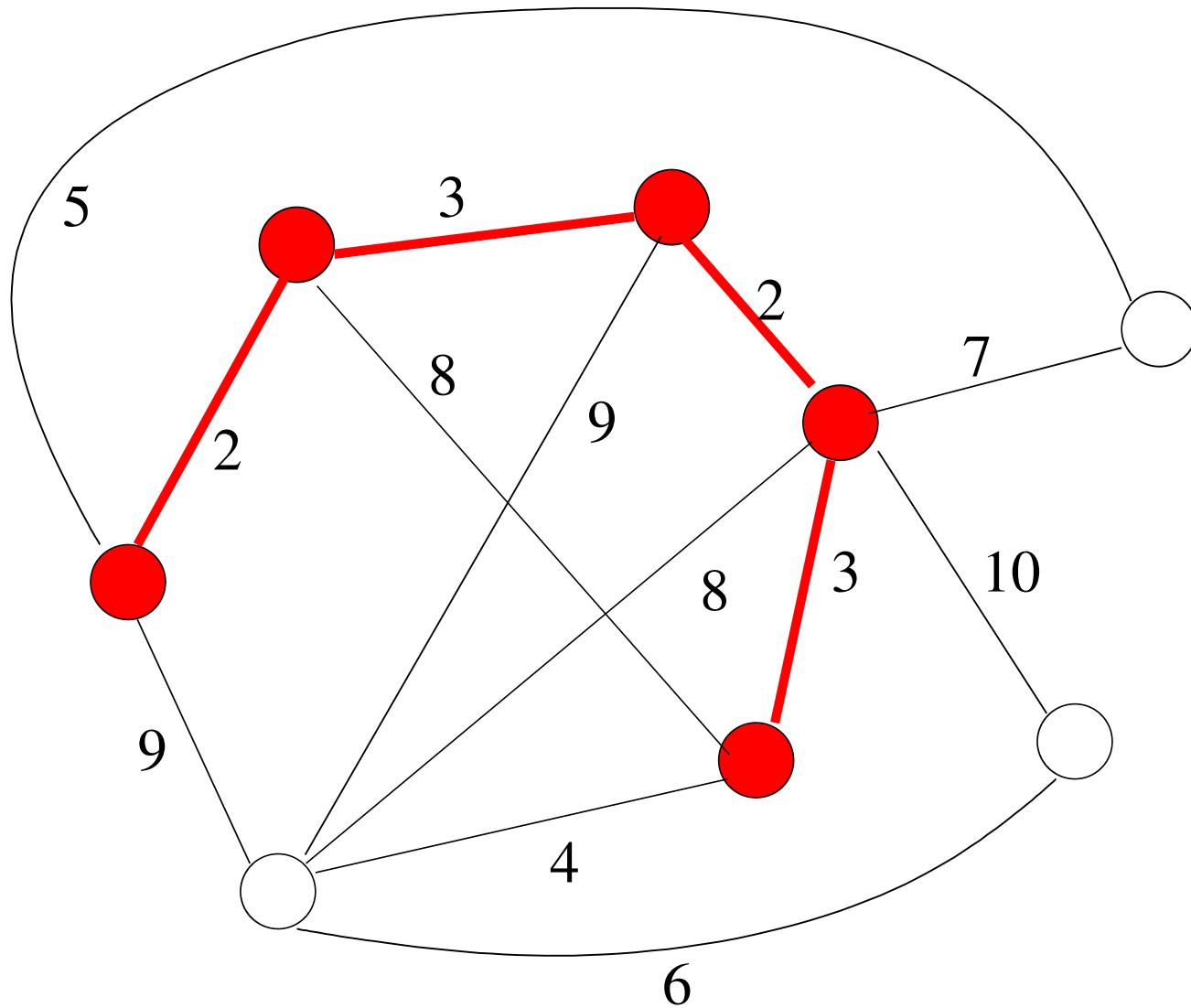


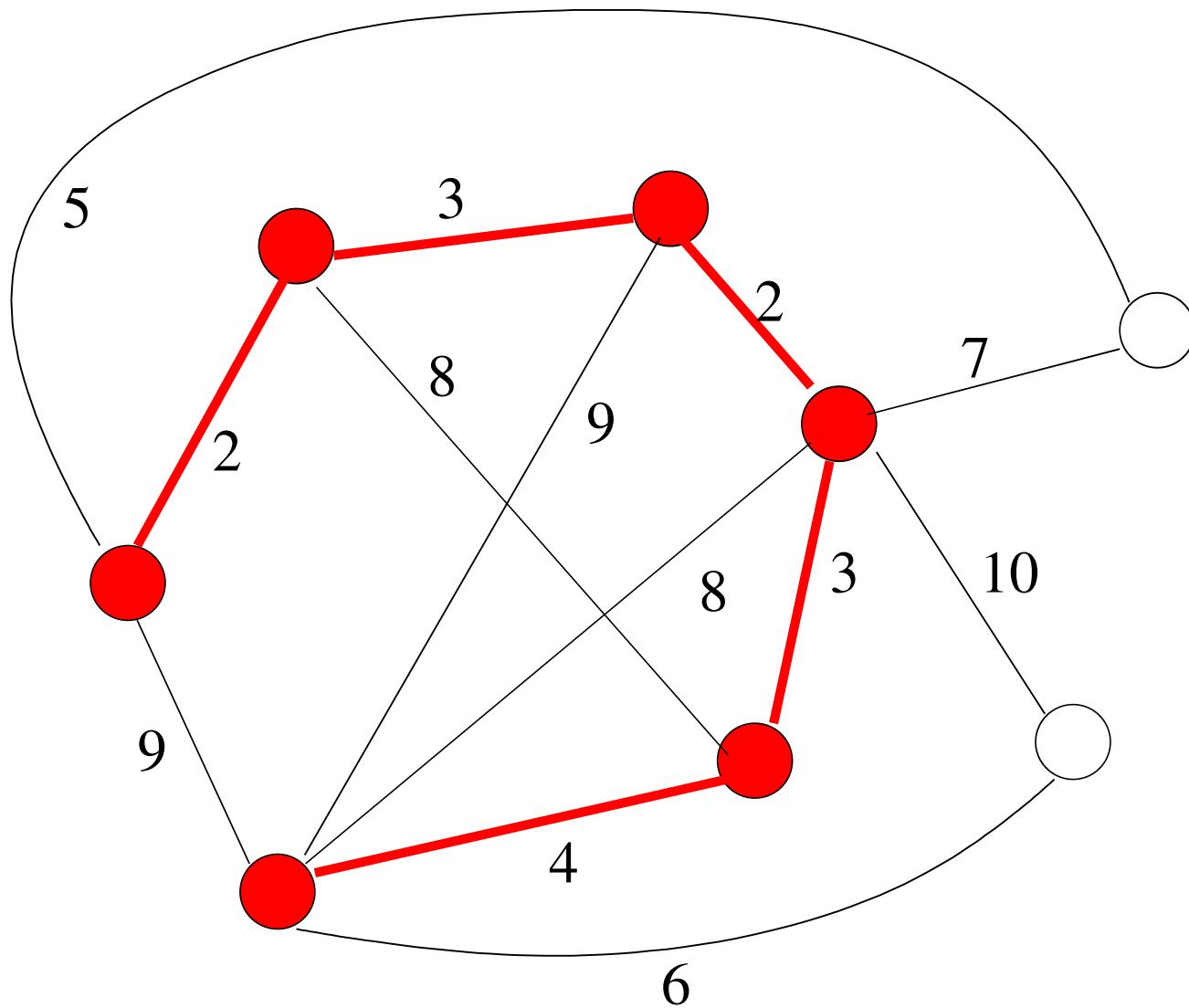












El coste del algoritmo de Prim dependerá de cómo implementemos la selección de la arista candidata. El bucle principal hace $n - 1$ iteraciones, pues en cada iteración añadimos una arista al árbol A , o equivalentemente, en cada iteración *vemos* un vértice nuevo y terminaremos cuando todos estén *vistos*.

En el interior del bucle principal se hacen dos tareas “costosas”: seleccionar la arista de mínimo peso en *Candidatas* y añadir ciertas aristas adyacentes al vértice recién visto al conjunto de *Candidatas*.

El coste de la segunda tarea es proporcional al grado del vértice v y en total aportará al coste

$$\sum_{v \in V(G)} \Theta(\text{grado}(v)) = \Theta\left(\sum_{v \in V(G)} \text{grado}(v)\right) = \Theta(m)$$

Pero si usamos una implementación ingenua para el conjunto de *Candidatas*, el coste de seleccionar una arista es $\mathcal{O}(m)$ y el coste total del algoritmo de Prim es $\mathcal{O}(n \cdot m)$.

Para conseguir mejorar sensiblemente el coste del algoritmo el conjunto de *Candidatas* debe implementarse como un min-heap, siendo la prioridad de cada arista su coste.

Entonces la selección (y eliminación) de la arista de coste mínimo en cada iteración pasa a ser $\mathcal{O}(\log m)$.

Por otro lado, el coste de ir añadiendo aristas a *Candidatas* lo tenemos que recalcular:

$$\begin{aligned}\sum_{v \in V(G)} \Theta(\text{grado}(v)(1 + \log m)) &= \Theta\left(\sum_{v \in V(G)} \text{grado}(v)(1 + \log m)\right) \\ &= \Theta(m \log m)\end{aligned}$$

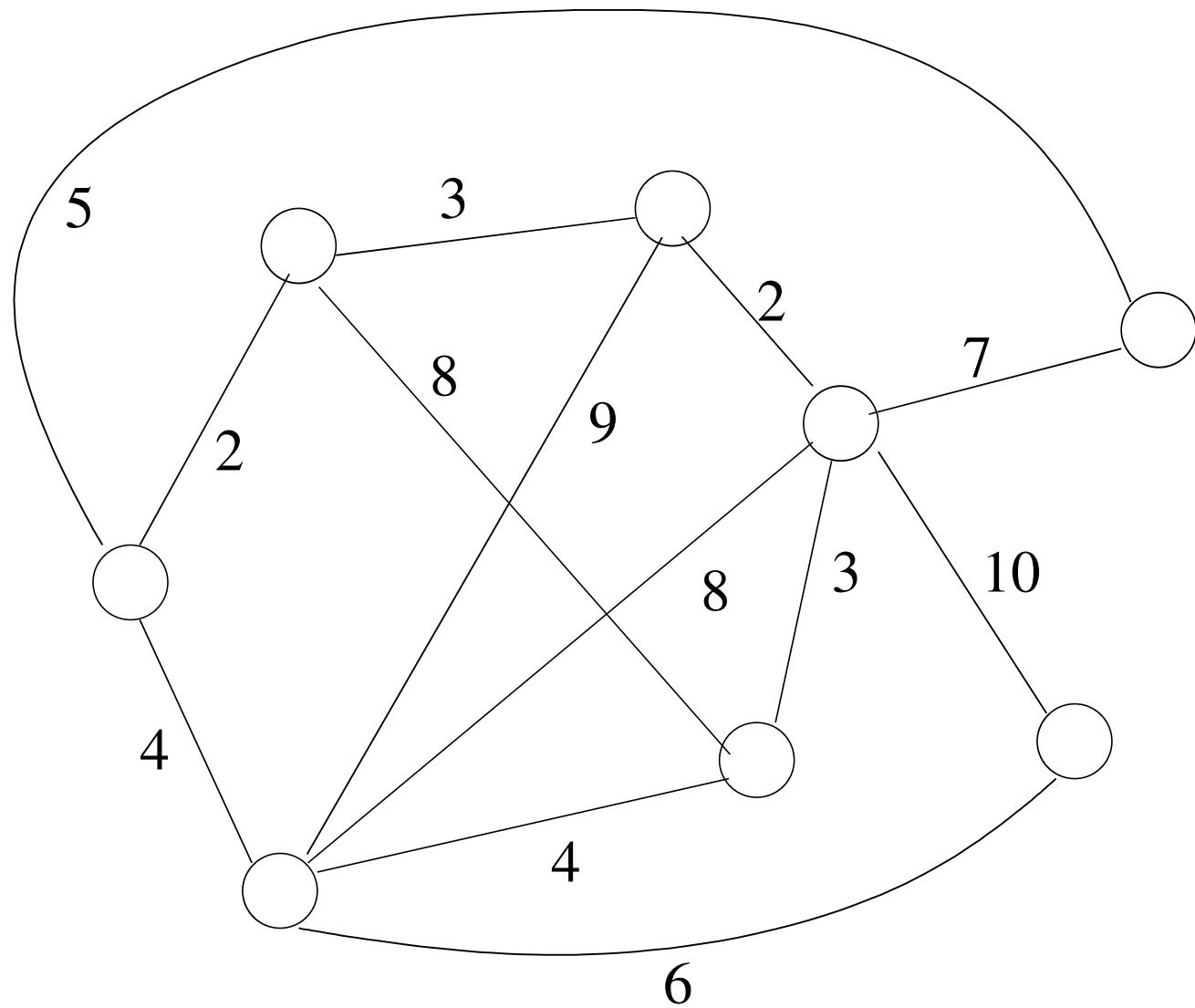
En total: $\Theta((n + m) \log m)$

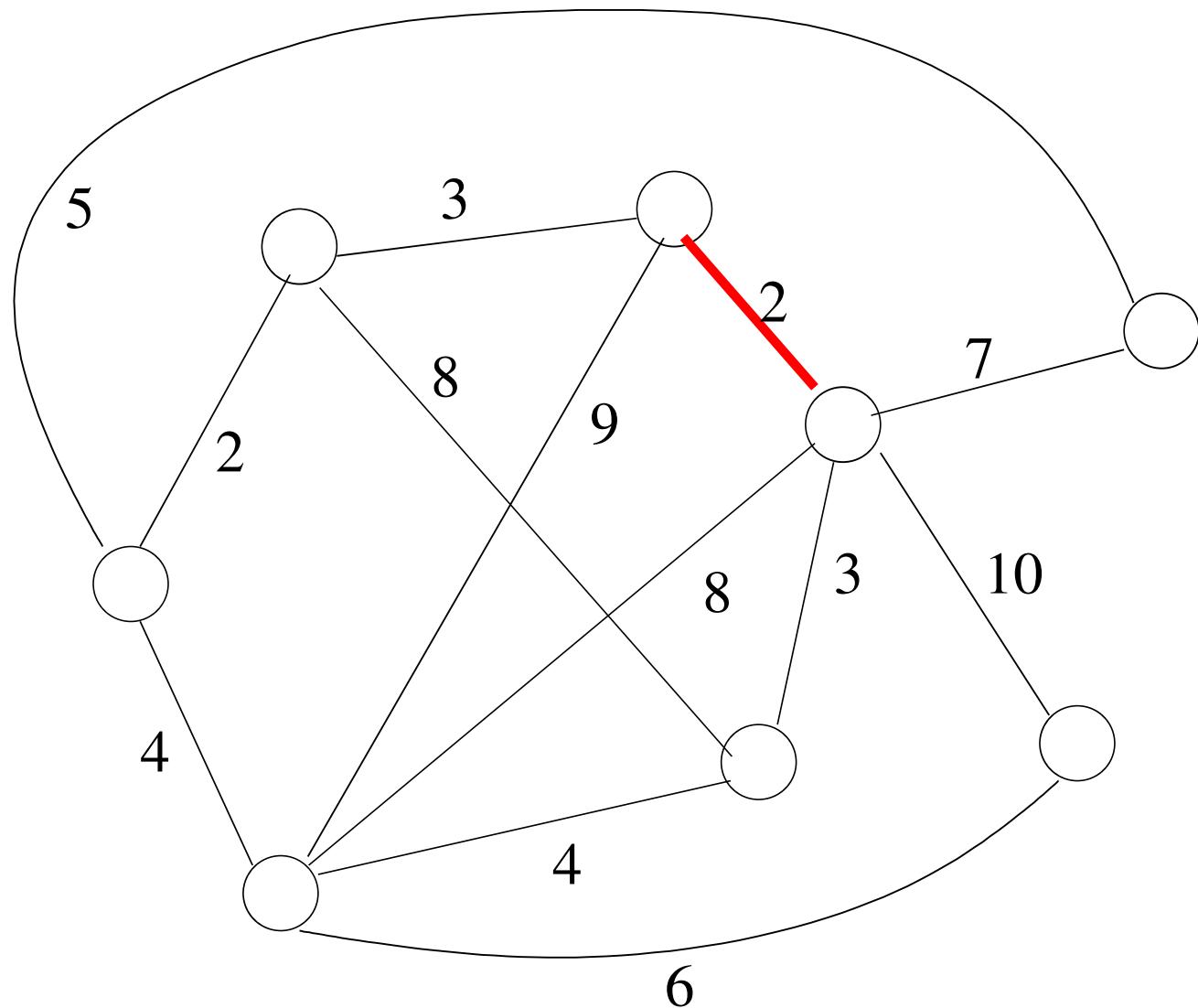
Algoritmo de Kruskal

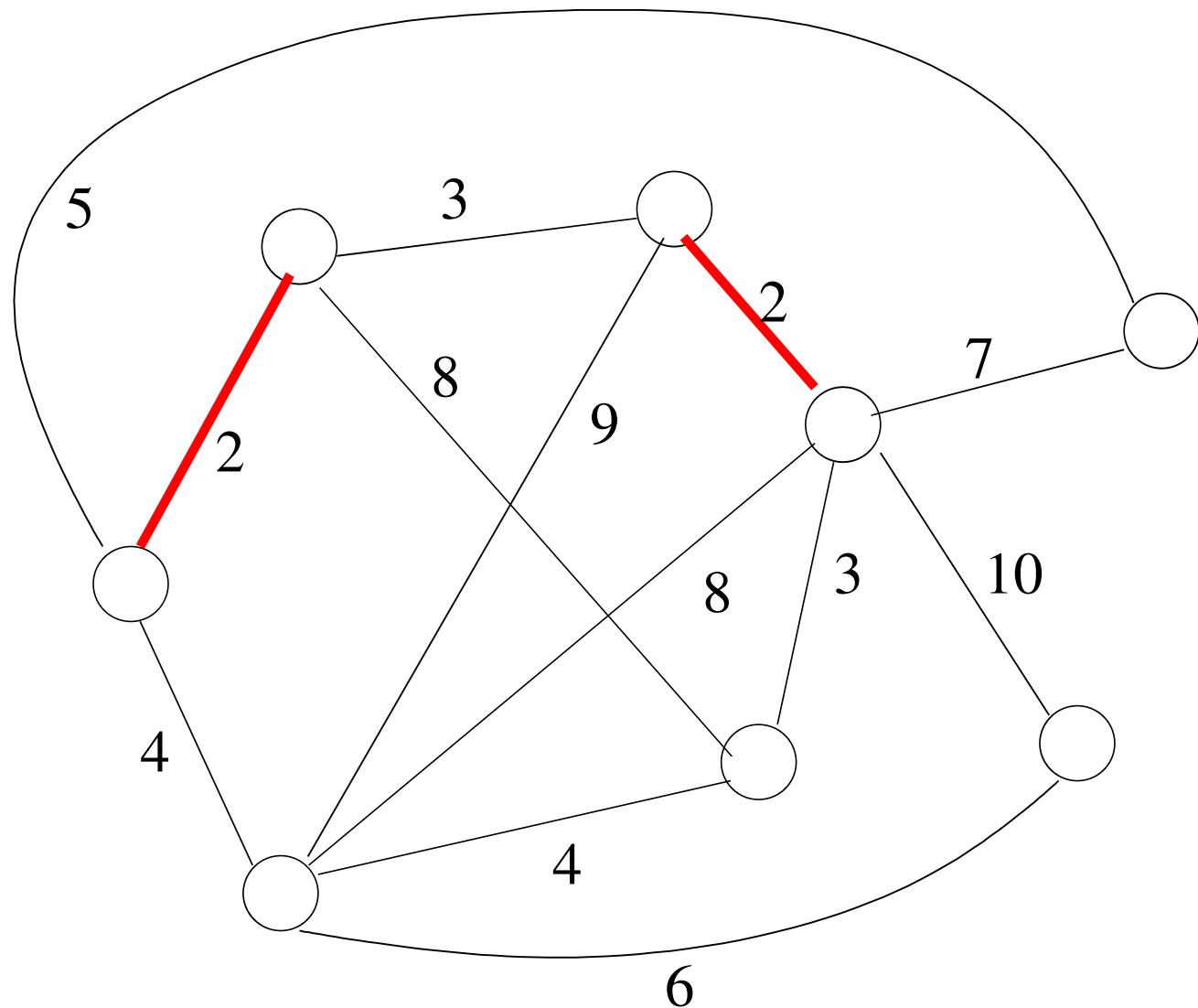
En el algoritmo de Kruskal se mantiene en todo momento un conjunto A de aristas que es un bosque (un conjunto de árboles). En cada paso se toma una arista e de mínimo peso entre todas las posibles; si dicha arista cierra un ciclo se descarta, si no cierra un ciclo, e se agrega a A .

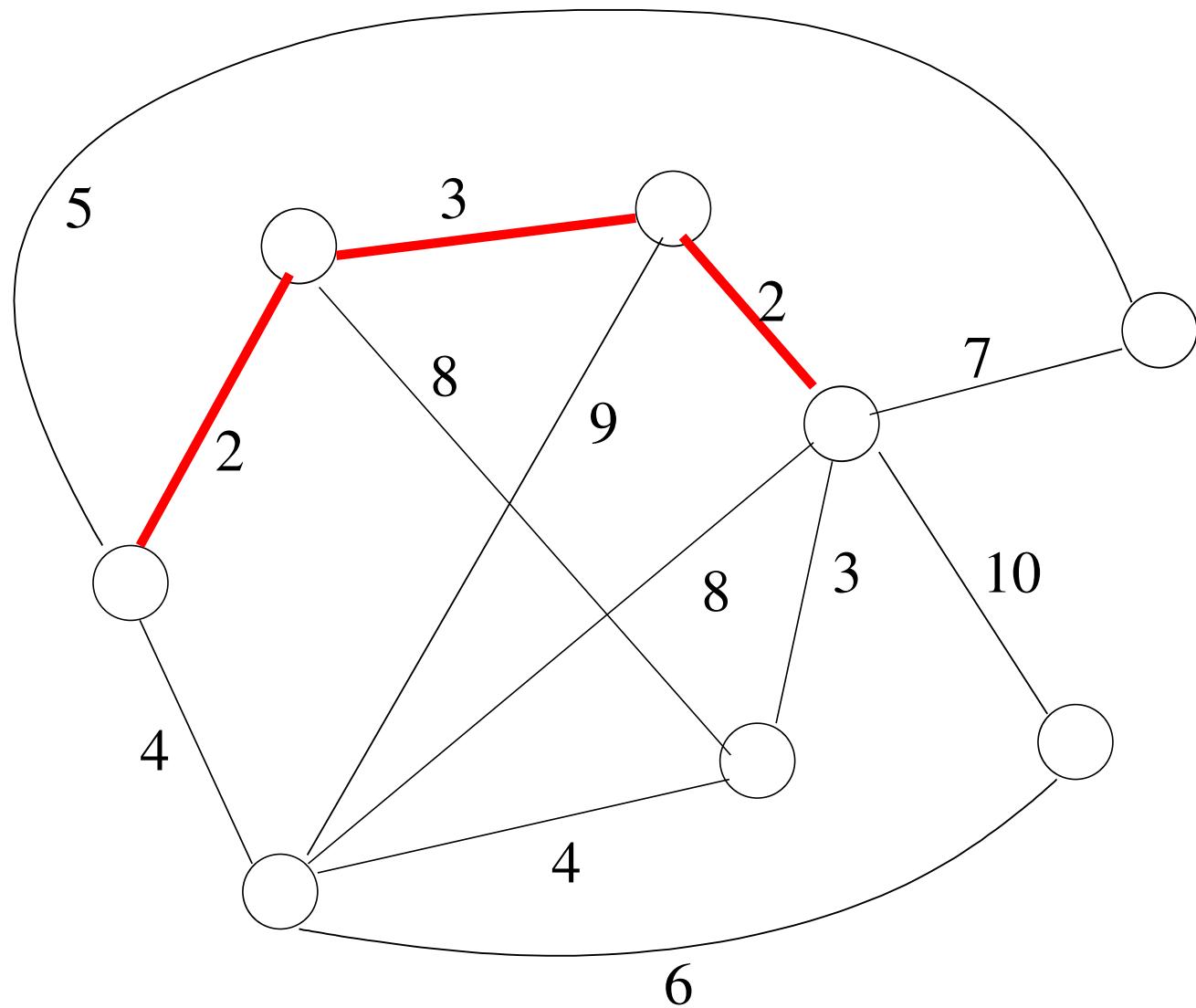
Cuando la arista $e = (u, v)$ escogida no cierra un ciclo, se define como corte el que forman los vértices en la misma componente conexa de u (mediante aristas de A) por un lado, y los restantes vértices del grafo por otro. Claramente A respeta el corte y e es la arista de mínimo peso que respeta ese corte.

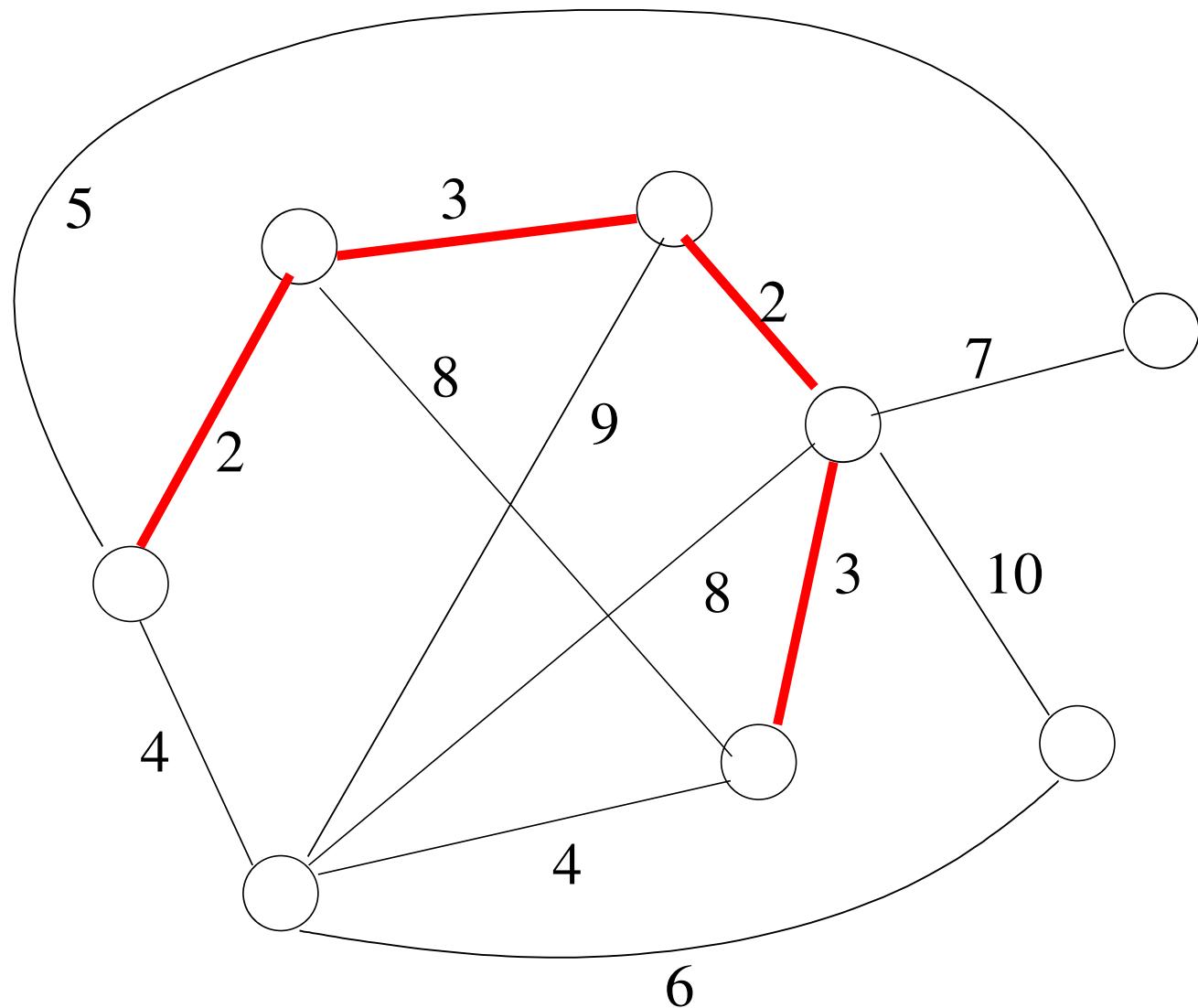
```
procedure KRUSKAL( $G = \langle V, E \rangle$ )
    Ordenar  $E$  por peso creciente
     $A := \emptyset$ 
    while  $|A| \neq |V(G)| - 1$  do
         $e = (u, v) := \text{SIGUIENTE}(E)$ 
        if  $e$  no cierra un ciclo en  $A$  then
             $A := A \cup \{e\}$ 
        end if
    end while
    return  $A$ 
end procedure
```

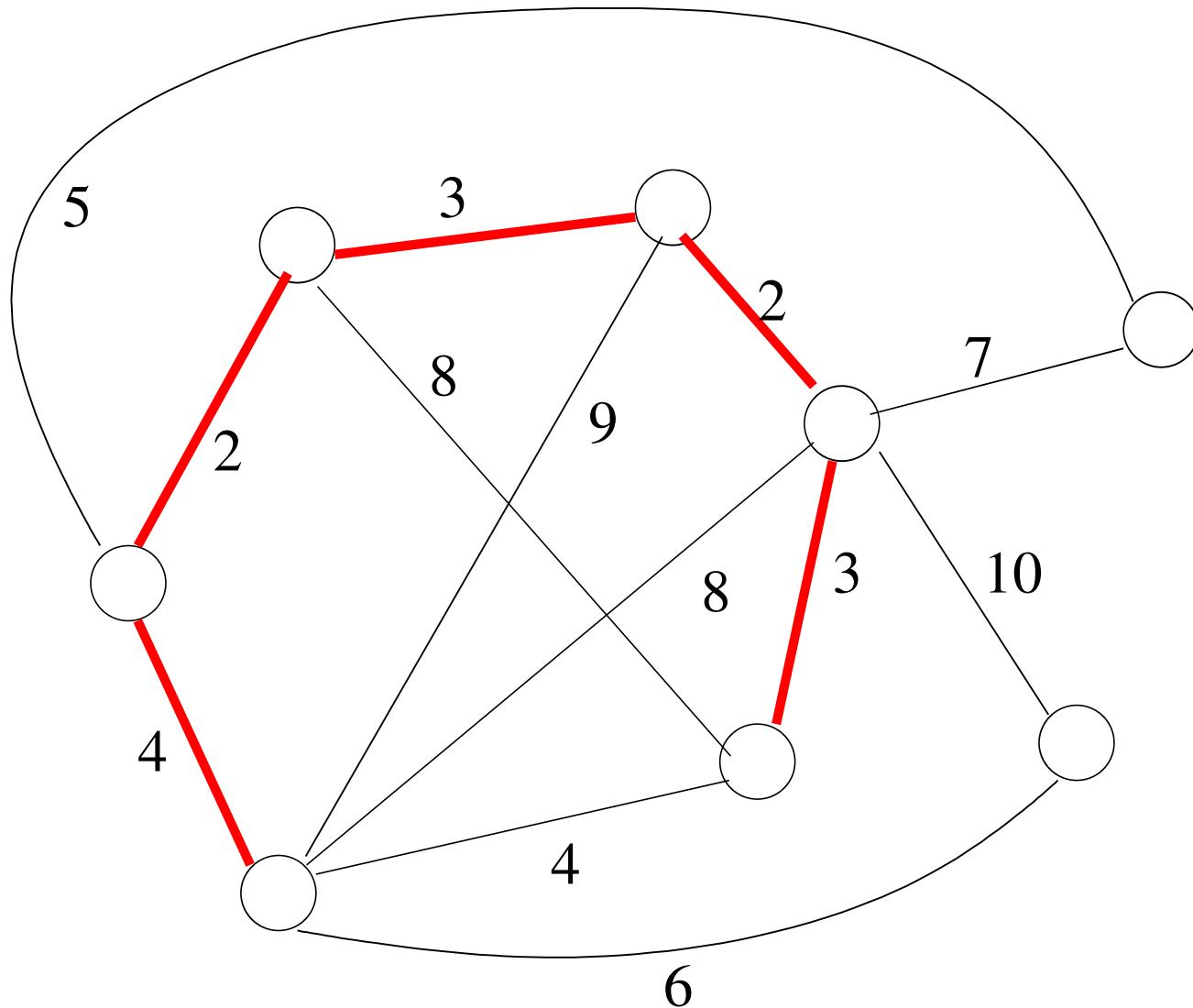


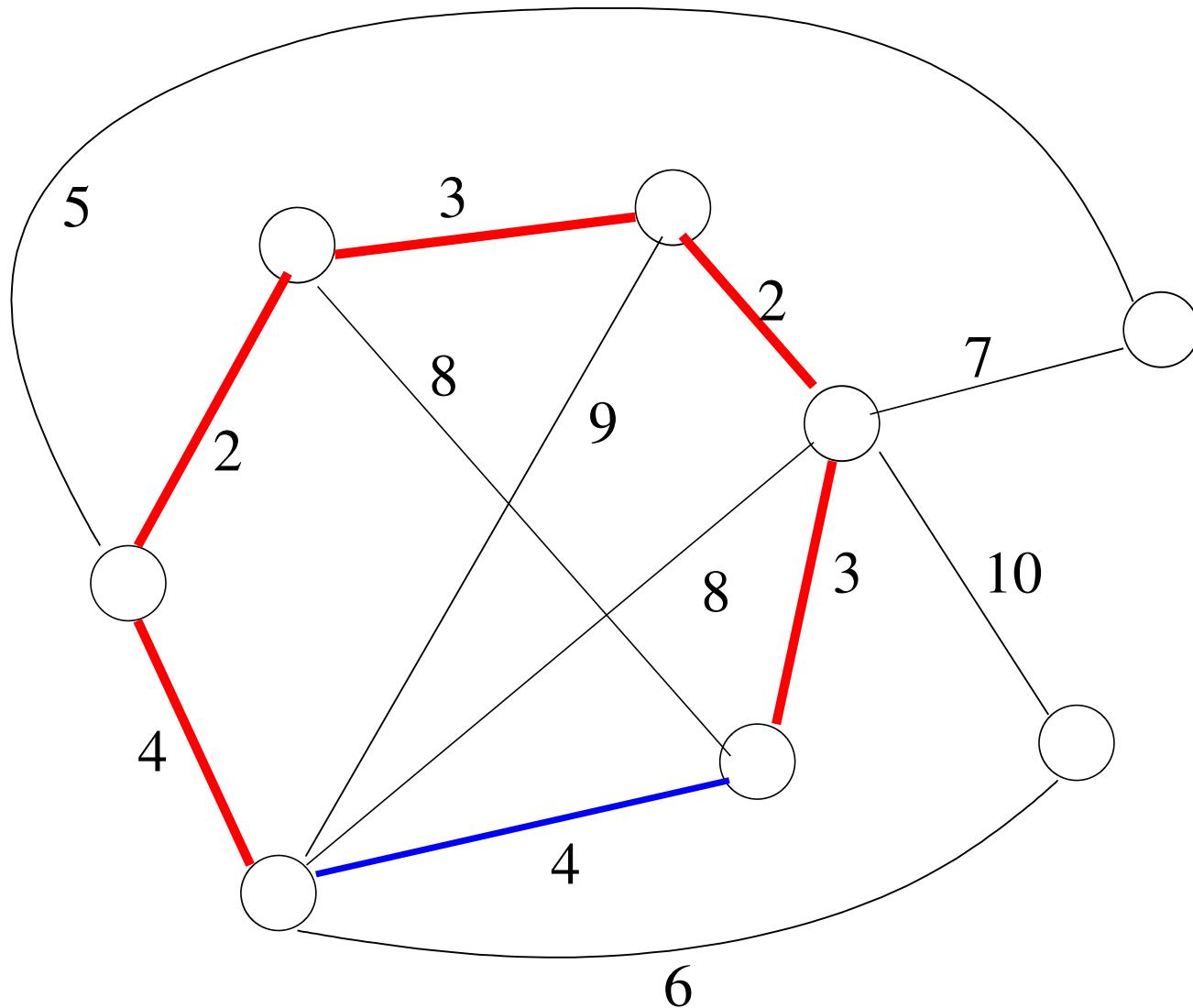


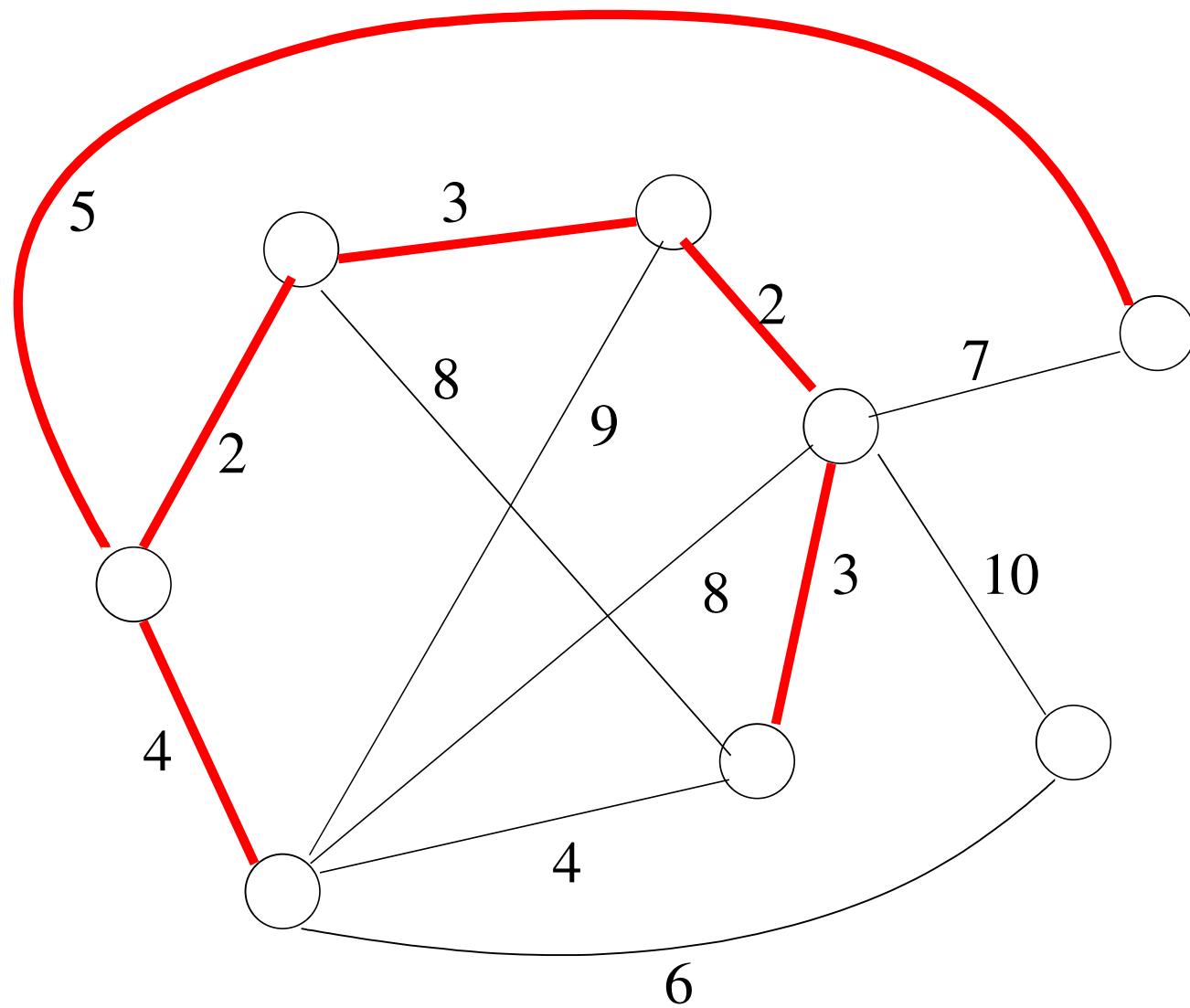


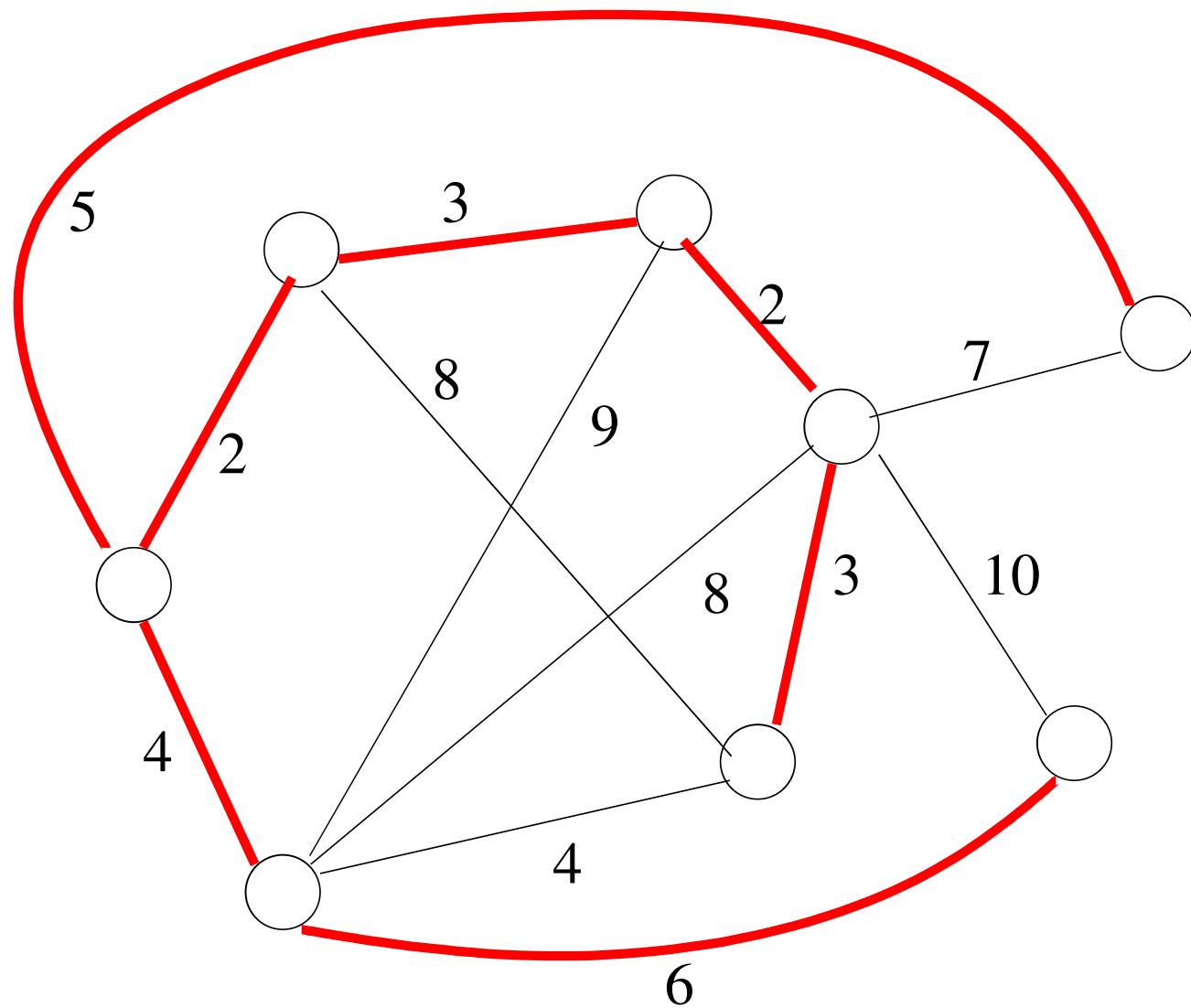












Para determinar si una cierta arista e cierra un ciclo en A de manera eficiente usaremos una estructura de datos “astuta” denominada **PARTICION** (también se les conoce como MFSETS y UNION-FIND).

Inicialmente creamos una PARTICION en la que cada vértice del grafo constituye un **bloque** por sí solo. Cada vez que agreguemos una arista $e = (u, v)$ al conjunto A , los bloques a los que pertenecen u y v se fusionan en un solo bloque (*merge, union*).

Cada bloque de vértices es pues una componente conexa de A ; si existe un camino entre los vértices w y w' usando las aristas de A , w y w' pertenecerán a un mismo bloque de la PARTICIÓN.

El test que determina si una nueva arista cierra un ciclo entonces consiste simplemente en ver si los vértices u y v pertenecen a un mismo bloque de la PARTICIÓN o no. Si ya están conectados, la nueva arista cerraría un ciclo; en caso contrario, la nueva arista no cierra un ciclo y los vértices pasan a estar conectados.

Habitualmente la clase PARTICIÓN proporciona una operación FIND que dado un elemento u nos devuelve el **representante** del bloque en el que está u . Dos elementos están en el mismo bloque si y sólo si los representantes de sus bloques son idénticos.

procedure KRUSKAL($G = \langle V, E \rangle$)

 Ordenar E por peso creciente

 ▷ Se crea una partición inicial, cada vértice en un
 bloque distinto

$P.\text{MAKE}(V)$

$A := \emptyset$

while $|A| \neq |V(G)| - 1$ **do**

$e = (u, v) := \text{SIGUIENTE}(E)$

if $P.\text{FIND}(u) \neq P.\text{FIND}(v)$ **then**

$A := A \cup \{e\}$

$P.\text{UNION}(u, v)$

end if

end while

return A

end procedure

Para calcular el coste del algoritmo de Kruskal tenemos por un lado el coste de la ordenación de las aristas

$\Theta(m \log m) = \Theta(m \log n)$, siendo $m = |E(G)|$. La creación de la partición inicial tiene coste $\Theta(n)$, siendo $n = |V(G)|$. Por otro lado, el bucle siguiente hace al menos $n - 1$ iteraciones (tantas como aristas tiene el MST hallado) pero puede llegar a hacer m iteraciones en caso peor. El coste del cuerpo del bucle vendrá dominado por el coste de las llamadas a las operaciones FIND y UNION. Si el bucle principal realiza N iteraciones, $n - 1 \leq N \leq m$, se harán $2N$ llamadas a FIND. Por otro lado, se hacen siempre exactamente $n - 1$ llamadas a UNION.

Es muy fácil obtener implementaciones de la PARTICIÓN que garanticen que el coste de FIND y UNION es $\mathcal{O}(\log n)$, lo que nos llevaría a un coste

$$\Theta(m \log n) + \mathcal{O}(m \log n) = \Theta(m \log n)$$

para el algoritmo.

Se puede mejorar el rendimiento notablemente (aunque no en caso peor) de la siguiente forma:

- 1 En vez de ordenar las aristas por peso, se crea un min-heap con coste $\Theta(m)$
- 2 Se usa una versión de UNION-FIND que garantiza que $\mathcal{O}(m)$ FINDS + $\mathcal{O}(n)$ UNIONS tienen coste $\mathcal{O}((m + n)\alpha(m, n))$. La función $\alpha(m, n)$, llamada función inversa de Ackermann crece de manera extremadamente lenta, a los efectos prácticos, por muy grandes que sean m y n , $\alpha(m, n) \leq 4$.
- 3 El bucle principal seguirá teniendo coste $\mathcal{O}(m \log n)$ pero en la mayoría de casos será más cercano a $\mathcal{O}(n \log n)$ porque no se suelen hacer muchas más del mínimo de $n - 1$ iteraciones.

El coste del algoritmo vendrá dominado por el coste de extraer aristas durante las iteraciones del bucle principal y en caso peor es $\Theta(m \log n)$; en promedio será cercano a $\Theta(n \log n)$, lo cual es muy ventajoso, sobre todo si $m \gg n$.

Parte III

Algoritmos Voraces

- Introducción a los Algoritmos Voraces
- Compresión de Datos: Códigos de Huffman
- Árboles de Expansión Mínimos: Algoritmos de Kruskal y de Prim
- Particiones

Particiones

Una *partición* Π de un conjunto no vacío \mathcal{A} es una colección de subconjuntos no vacíos $\Pi = \{\mathcal{A}_1, \dots, \mathcal{A}_k\}$ tal que

- 1 Si $i \neq j$ entonces $\mathcal{A}_i \cap \mathcal{A}_j = \emptyset$.
- 2 $\mathcal{A} = \bigcup_{1 \leq i \leq k} \mathcal{A}_i$.

Se suele denominar *bloque* de la partición Π a cada uno de los A_i 's. Las particiones y las relaciones de equivalencia están estrechamente ligadas. Recordemos que \equiv es una relación de equivalencia en \mathcal{A} si y sólo si

- 1 \equiv es reflexiva: para todo $a \in \mathcal{A}$, $a \equiv a$.
- 2 \equiv es transitiva: si $a \equiv b$ y $b \equiv c$, entonces $a \equiv c$, para cualesquiera a, b y c en \mathcal{A} .
- 3 \equiv es simétrica: $a \equiv b$ si y sólo si $b \equiv a$, para cualesquiera a y b en \mathcal{A} .

Dada una partición Π de \mathcal{A} , ésta induce una relación de equivalencia \equiv_Π definida por

$$x \equiv_\Pi y \iff x \text{ e } y \text{ pertenecen a un mismo bloque } \mathcal{A}_i \in \Pi$$

Y a la inversa, dada una relación de equivalencia \equiv en \mathcal{A} , ésta induce una partición $\Pi = \{\mathcal{A}_x\}_{x \in \mathcal{A}}$, donde

$$\mathcal{A}_x = \{y \in \mathcal{A} \mid y \equiv x\}.$$

Al subconjunto de elementos equivalentes a x se le denomina *clase de equivalencia* de x . Cada uno de los bloques de la partición inducida por una relación \equiv es por lo tanto una clase de equivalencia. Nótese que si $x \equiv y$ entonces $A_x = A_y$. Un elemento cualquiera de la clase A_x se denomina *representante* de la clase.

En muchos algoritmos, especialmente en algoritmos sobre grafos, es importante poder representar particiones de un conjunto finito de manera eficiente.

Vamos a suponer que el conjunto *soporte* sobre el que se define la partición es $\{1, \dots, n\}$; sin excesiva dificultad, empleando alguna de las técnicas vistas en temas precedentes podemos representar de manera eficiente una biyección entre un conjunto finito \mathcal{A} de cardinalidad n y el conjunto $\{1, \dots, n\}$ y con ello una partición sobre el conjunto soporte \mathcal{A} en caso necesario.

Adicionalmente, supondremos que el conjunto soporte es estático, es decir, ni se añaden ni se eliminan elementos. No obstante, con poca dificultad extra podemos obtener representaciones eficientes para particiones de conjuntos dinámicos.

Generalmente el tipo de operaciones que una clase Partition debe soportar son las siguientes: 1) dados dos elementos i y j , determinar si pertenecen al mismo bloque o no; 2) dados dos elementos i y j fusionar los bloques a los que pertenecen (si procede) en un sólo bloque, devolviendo la partición resultante.

Frecuentemente el primer tipo de operación se realiza mediante una operación FIND que dado un elemento i , devuelve un representante de la clase a la que pertenece i . Si dos elementos i y j tienen el mismo representante, entonces han de estar en el mismo bloque.

El segundo tipo de operación se llama MERGE o UNION. De ahí que las particiones se les llame a menudo estructuras **union-find** o **mfsets** (abreviación de *merge-find sets*).

La operación MAKE crea una partición de $\{1, \dots, n\}$ consistente en n bloques cada uno de los cuales contiene un elemento.

En muchas aplicaciones antes de hacer cualquier UNION se habrá determinado previamente si los elementos i y j cuyos bloques habrían de unirse, están o no en el mismo bloque; para ello se habrá preguntado si $P.\text{FIND}(i) = P.\text{FIND}(j)$ o no. Por esta razón es habitual que en una clase `Particion` la operación UNION sólo actúe sobre elementos que son representantes de sus respectivas clases.

```
procedure  $F(\dots)$ 
```

```
...
```

```
 $ri := P.\text{FIND}(i)$ 
```

```
 $rj := P.\text{FIND}(j)$ 
```

```
if  $ri \neq rj$  then
```

```
 $P.\text{UNION}(ri, rj)$ 
```

```
...
```

```
end if
```

```
end procedure
```

Puesto que la operación MAKE recibe como parámetro el número de elementos n del conjunto soporte, podremos utilizar implementaciones en vector, reclamando un vector con el número apropiado de componentes a la memoria dinámica si nuestro lenguaje de programación lo soporta. También será posible utilizar este tipo de implementación si el valor de n está acotado y dicha cota no es irrazonablemente grande. En cualquier caso podremos modificar sin demasiado esfuerzo las implementaciones que se verán a continuación para que sean completamente dinámicas y funcionen correctamente en aquellos casos en que no se puedan crear vectores cuyo tamaño se fija en tiempo de ejecución o si el conjunto soporte ha de soportar inserciones y borrados.

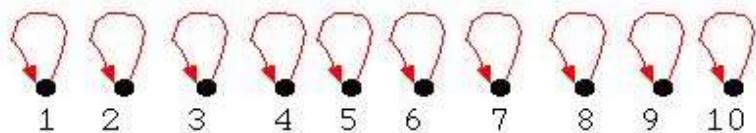
Quick-find consiste representar la partición mediante un vector P y almacenar en la componente i de P el representante de la clase a la que pertenece i . De este modo la operación FIND tiene coste constante, ya que basta examinar $P[i]$. Sin embargo, la operación UNION tendrá coste $\Theta(n)$ ya que cada uno de los elementos de la clase en la que está j (los k 's tales que $P[k] = P[j]$) han de cambiar de representante ($P[k] := P[i]$). O bien los elementos del mismo bloque que i han de pasar al bloque de j .

Con algunas modificaciones puede evitarse el recorrido completo del vector P y restringirlo a los elementos del bloque de j (o del bloque de i); pero aún así el coste de una UNION sigue siendo lineal en n en el caso peor, ya que cualquiera de los dos bloques pude contener una fracción considerable del total de los elementos.

Aunque resulta un tanto forzado, conviene contemplar la representación *quick-find* como un bosque de árboles; cada árbol representa a un bloque de la partición en un momento dado. Los árboles están representados mediante apuntadores al padre, siendo la raíz de cada árbol el representante del bloque correspondiente. Puesto que la raíz de un árbol no tiene padre, las raíces se apuntan a sí mismas. Del invariante de la representación de *quick-find* se sigue que todos los árboles tienen altura 1 (si sólo contienen un elemento) o altura 2 (todos los elementos de un bloque excepto el representante están en el segundo nivel, apuntando a la raíz).

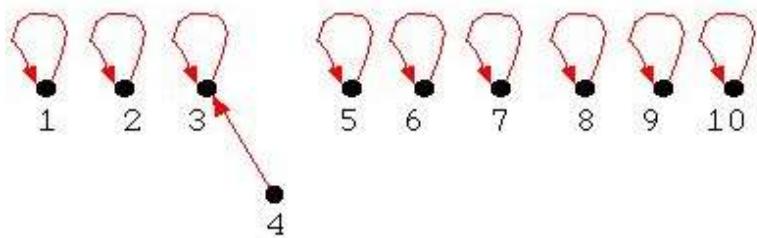
make(10)

1	2	3	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10



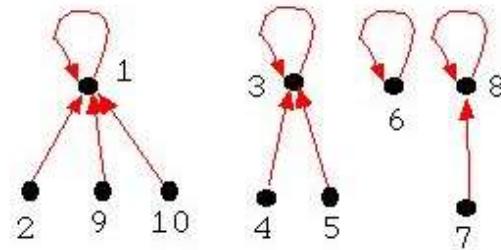
union(3, 4)

1	2	3	3	5	6	7	8	9	10
1	2	3	3	5	6	7	8	9	10



union(1, 2); union(4, 5); union(1, 9);
union(2, 10); union(8, 7)

1	1	3	3	3	6	8	8	1	1
1	2	3	4	5	6	7	8	9	10

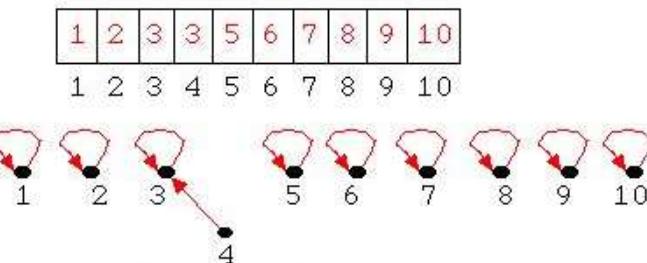


Otra estrategia, *quick-union*, explota la correspondencia entre bloques y árboles del siguiente modo: para unir los bloques de i y j se localizan al representante de i , digamos u , y se coloca a u como hijo de j . Alternativamente, podemos localizar ambos representantes y hacer que uno de ellos sea hijo del otro.

```
procedure MAKE( $n$ )
  for  $i := 1$  to  $n$  do
     $P[i] := i$ 
  end for
end procedure
procedure UNION( $i, j$ )
   $u := \text{FIND}(i)$ 
   $v := \text{FIND}(j)$ 
   $P[u] := v$ 
end procedure
▷  $1 \leq i \leq n$ 
procedure FIND( $i$ )
  while  $P[i] \neq i$  do
     $i := P[i]$ 
  end while
  return  $i$ 
end procedure
```

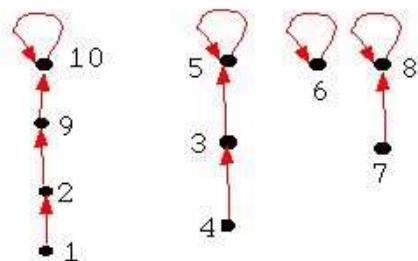
Aunque en general los árboles resultantes de una secuencia de uniones serán relativamente equilibrados y el coste de las operaciones será bajo, en caso peor podemos crear árboles poco equilibrados de modo que tanto una UNION como un FIND tengan coste proporcional al número de elementos involucrados. Por ejemplo, si realizamos una secuencia de UNIONes de tal modo que la clase en la que está j sólo contenga a j , obtendremos árboles equivalentes a listas.

```
make(10); union(4, 3)
```



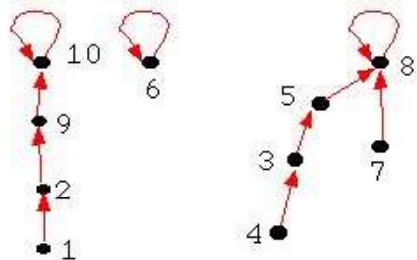
```
union(1,2); union(4,5); union(1,9);  
union(2, 10); union(7,8)
```

2	9	5	3	5	6	8	8	10	10
1	2	3	4	5	6	7	8	9	10



```
union(4,7);
```

2	9	5	3	8	6	8	8	10	10
1	2	3	4	5	6	7	8	9	10



Los comentarios previos sugieren posibles soluciones al problema. En la *unión por peso* el árbol con menos elementos es que el que se añade como hijo del que tiene más elementos. En la *unión por rango* el árbol de menor altura es el que se pone como hijo del de mayor altura. Tanto una como otra estrategia son fáciles de implementar, pero requieren, en principio, que se almacene información auxiliar sobre el tamaño o la altura de los árboles.

Se puede evitar el uso de espacio auxiliar observando que sólo se necesita esta información de tamaño o altura para las raíces y que el espacio correspondiente a sus apuntadores es esencialmente inútil, ya que sólo se precisaría un bit que indique que i es una raíz o no. Por ejemplo, podemos adoptar el convenio de que si $P[i] < 0$ entonces i es una raíz y $-P[i]$ es el tamaño del árbol.

```

procedure UNION( $i, j$ )
     $u := \text{FIND}(i)$ 
     $v := \text{FIND}(j)$ 
    if  $-P[u] > -P[v]$  then
        ▷  $u$  es el árbol “grande”
         $u := v$ 
    end if
        ▷  $u$  es el árbol “pequeño”
     $P[v] := P[v] + P[u]$ 
     $P[u] := v$ 
end procedure
▷  $1 \leq i \leq n$ 

procedure FIND( $i$ )
    while  $P[i] > 0$  do
         $i := P[i]$ 
    end while
    return  $i$ 
end procedure

```

El rendimiento $\mathcal{O}(\log n)$ de estas operaciones es consecuencia directa del siguiente lema.

Lema —

Dado un bloque de tamaño k en una Particion con unión por peso, la altura del árbol correspondiente es $\leq \log_2 k$.

Demostración

Si $k = 0$, el lema es obviamente cierto. Supongamos que es cierto para todos los tamaños hasta k y demostraremos entonces que es cierto para $k + 1$. Sea t el árbol correspondiente a un bloque de tamaño $k + 1$. Dicho bloque es el resultado de la unión de dos bloques de tamaños r y s , $r \leq s \leq k$. El árbol t tiene altura $h(t) \leq \max\{\log_2 r + 1, \log_2 s\}$, aplicando la hipótesis de inducción, y por la definición de unión por peso. Supongamos que $\log_2 r + 1 \leq \log_2 s$. Entonces

$$\begin{aligned} h(t) &\leq \max\{\log_2 r + 1, \log_2 s\} \\ &= \log_2 s < \log_2(k + 1). \end{aligned}$$

Demostración (continúa)

Por otro lado, si $\log_2 r + 1 > \log_2 s$, y teniendo en cuenta que $k + 1 = r + s \geq 2r$,

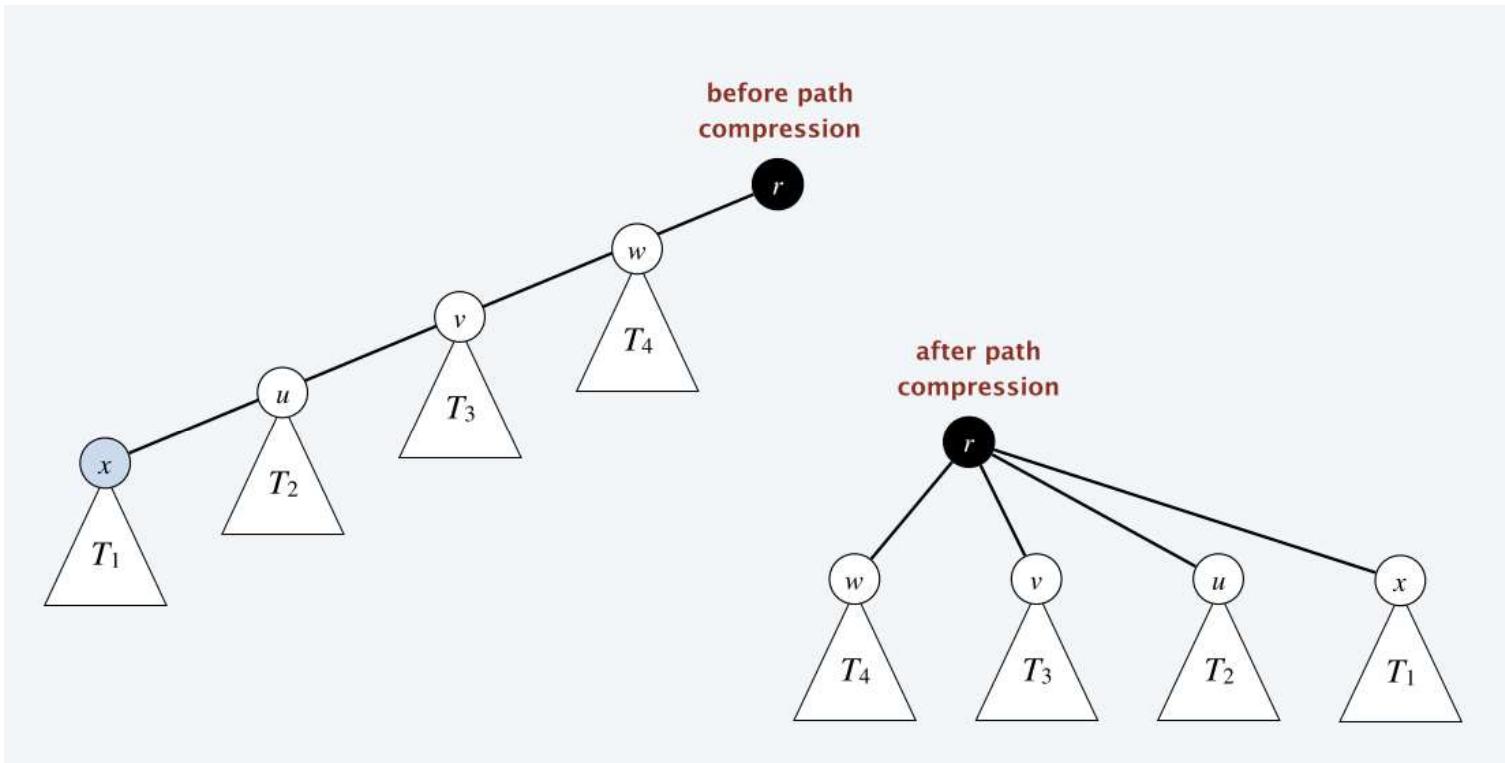
$$\begin{aligned} h(t) &\leq \max\{\log_2 r + 1, \log_2 s\} \\ &= \log_2 r + 1 = \log_2(2r) \leq \log_2(k + 1). \end{aligned}$$



Todavía puede conseguirse un mejor rendimiento empleando una heurística de *compresión de caminos*. La idea es reducir la distancia a la raíz de los elementos en el camino de i hasta la raíz durante una operación $\text{FIND}(i)$. Mientras se asciende desde i hasta la raíz se disminuye la altura del árbol.

Subsiguentes FINDs que afecten a i o alguno de los elementos que eran antecesores suyos serán más rápidos. La compresión de caminos acorta caminos de manera que poco a poco los árboles adoptan la forma que tendrían con *quick-find*, pero no teniendo que encargarse de ello la operación UNION ni compactándose todo un árbol de una sola vez.

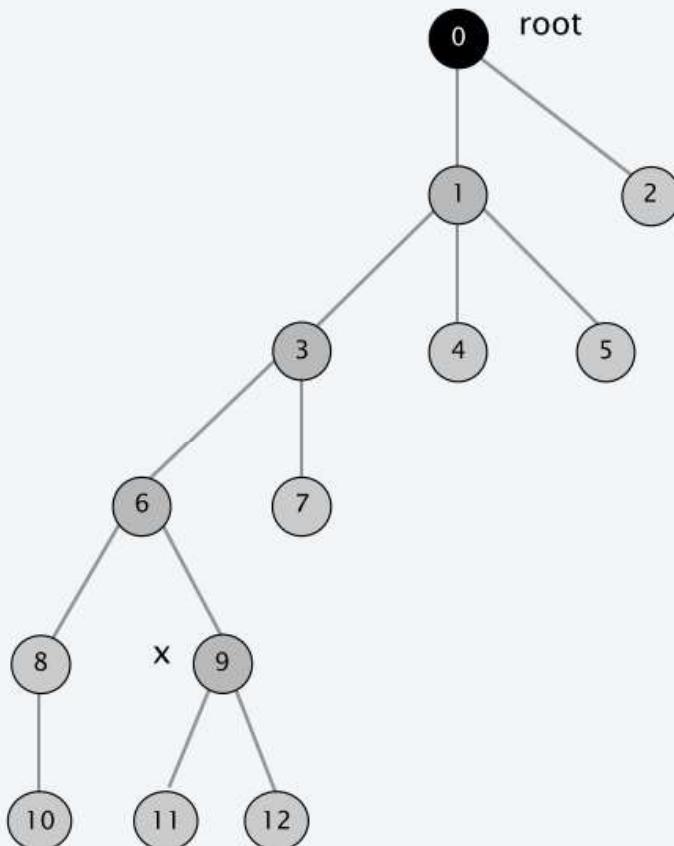
Compresión total : se modifica el apuntador de cada elemento en el camino desde i hasta la raíz, para que apunten a la raíz.



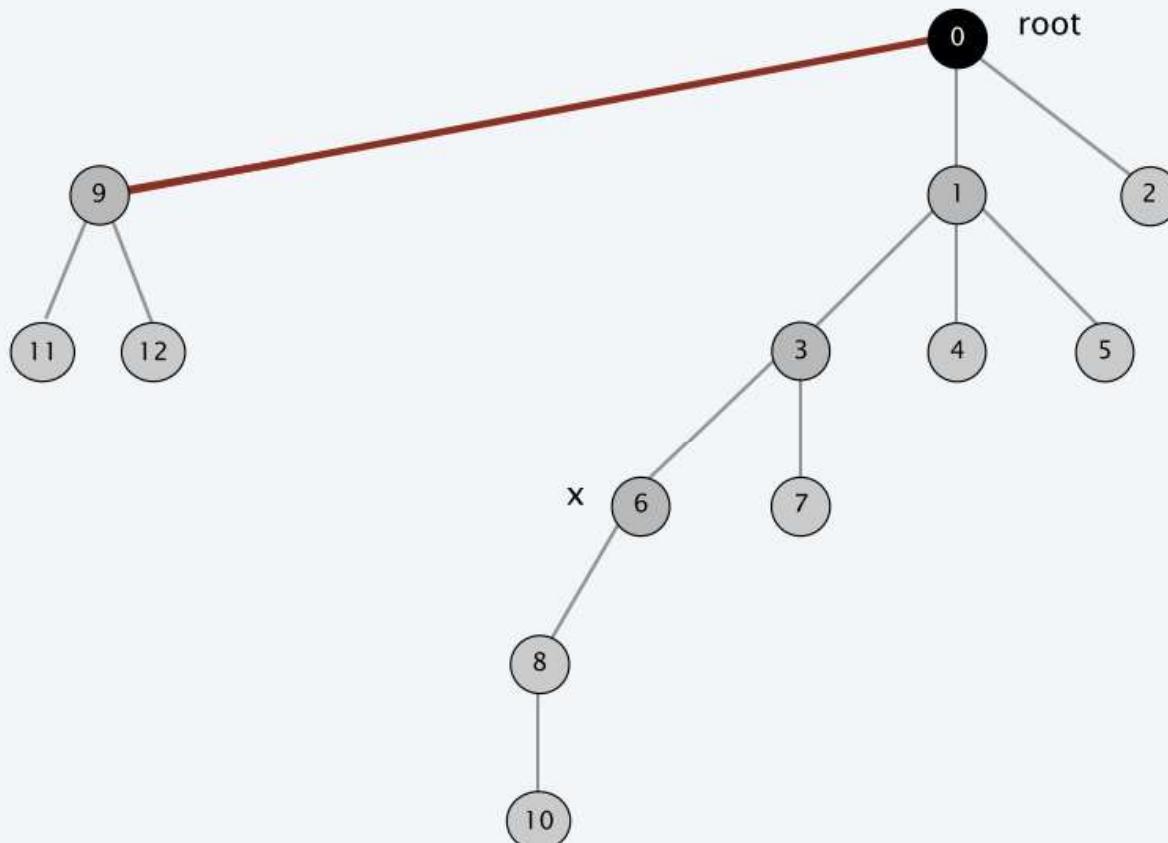
Fuente: Kevin Wayne

(<https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/UnionFind.pdf>)

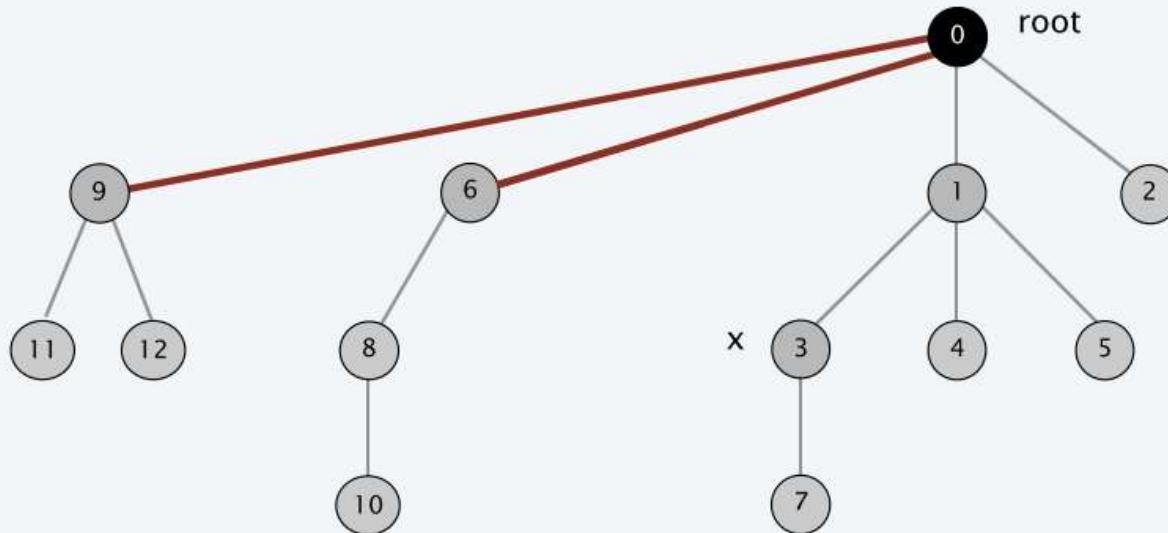
Compresión total : se modifica el apuntador de cada elemento en el camino desde i hasta la raíz, para que apunten a la raíz.



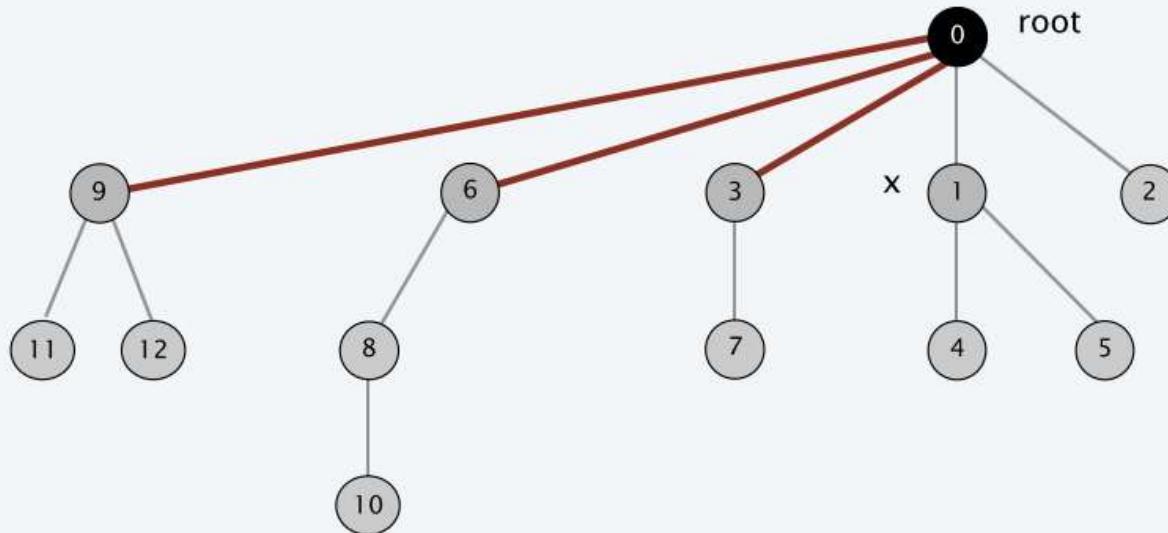
Compresión total : se modifica el apuntador de cada elemento en el camino desde i hasta la raíz, para que apunten a la raíz.



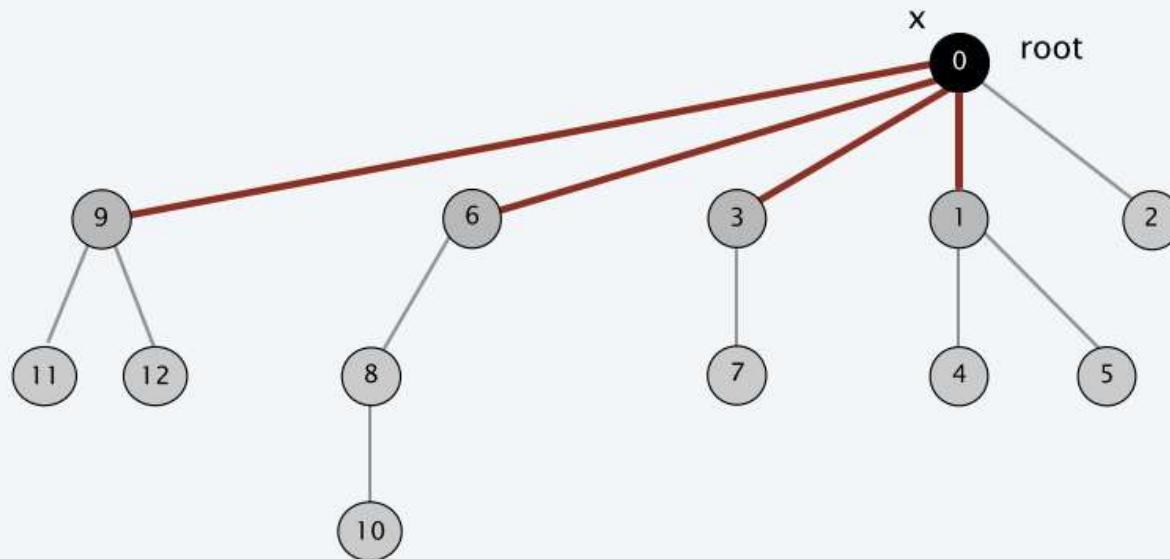
Compresión total : se modifica el apuntador de cada elemento en el camino desde i hasta la raíz, para que apunten a la raíz.



Compresión total : se modifica el apuntador de cada elemento en el camino desde i hasta la raíz, para que apunten a la raíz.

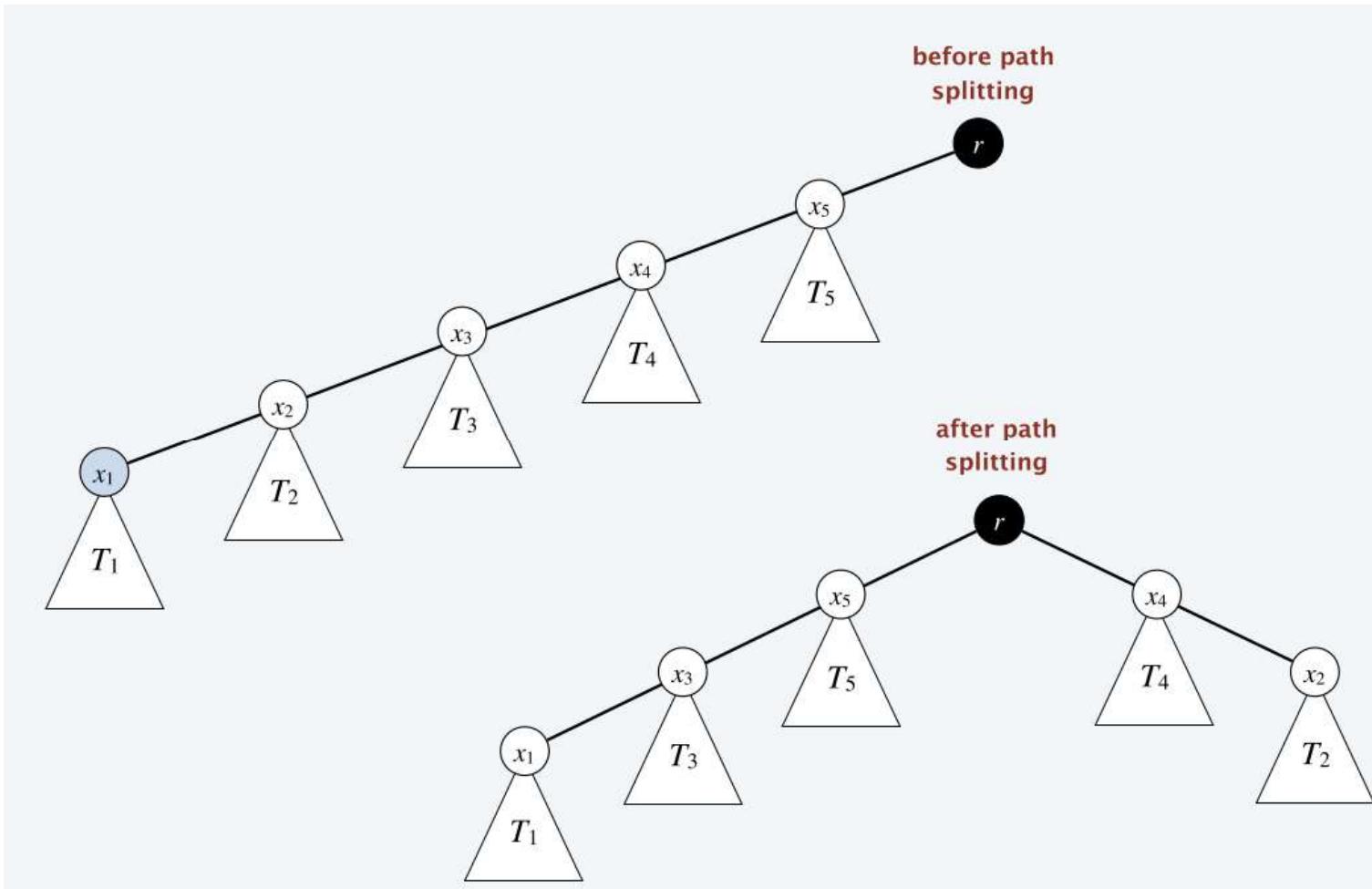


Compresión total : se modifica el apuntador de cada elemento en el camino desde i hasta la raíz, para que apunten a la raíz.



```
procedure FIND( $i$ )
  if  $P[i] < 0$  then
    return  $i$ 
  else
     $P[i] := \text{find}(P[i])$ 
    return  $P[i]$ 
  end if
end procedure
```

Path splitting: se modifica el apuntador de cada elemento en el camino desde i hasta $\text{FIND}(i)$, excepto a $\text{FIND}(i)$ y el hijo de éste, para que apunte a su “abuelo”.



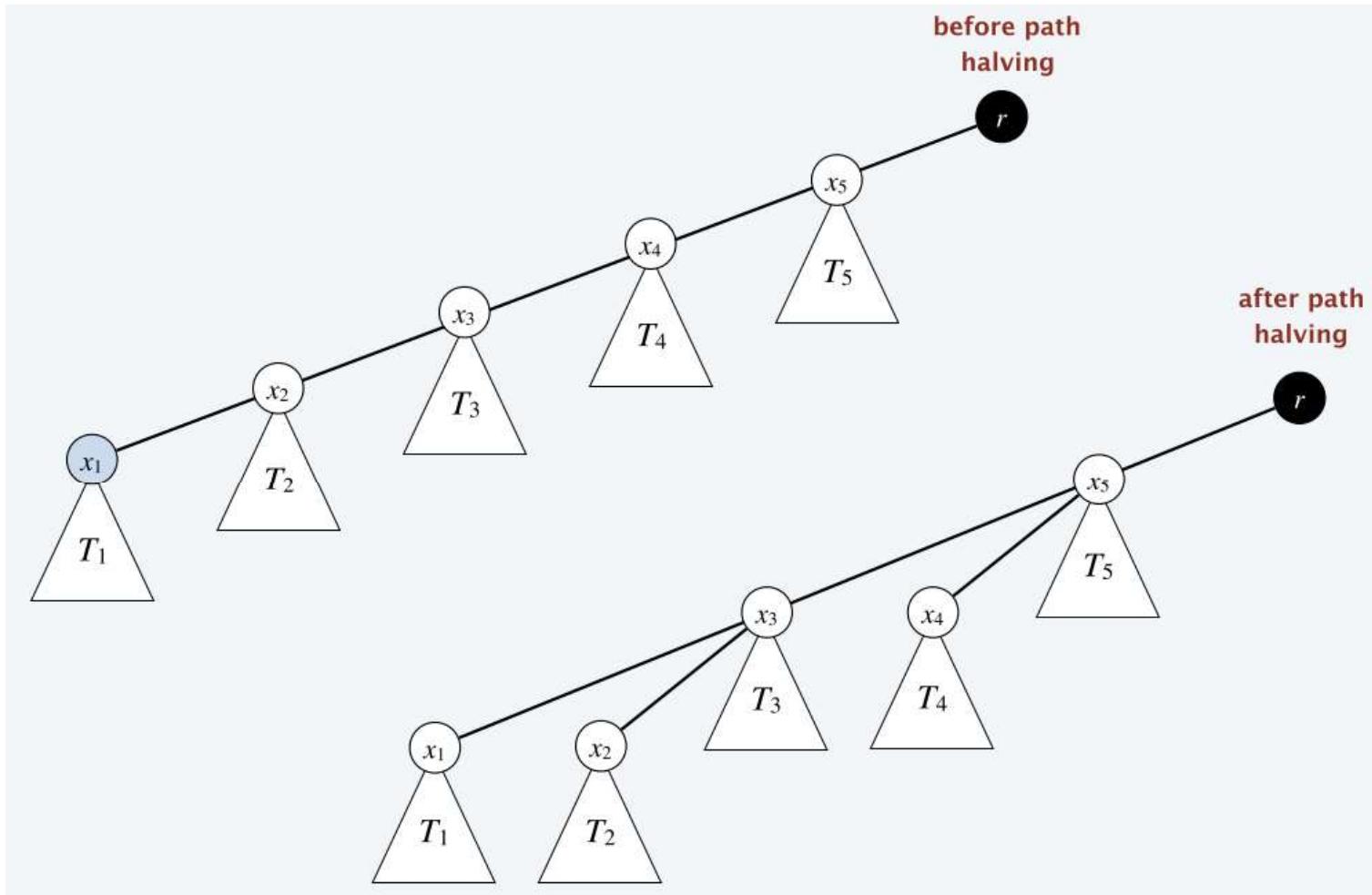
Fuente: Kevin Wayne

(<https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/UnionFind.pdf>)

Path splitting: se modifica el apuntador de cada elemento en el camino desde i hasta $\text{FIND}(i)$, excepto a $\text{FIND}(i)$ y el hijo de éste, para que apunte a su “abuelo”.

```
procedure FIND( $i$ )
     $j := i; pj := P[j]$ 
    while  $P[pj] > 0$  do
         $P[j] := P[pj]$ 
         $j := pj$ 
         $pj := P[j]$ 
    end while
    return  $pj$ 
end procedure
```

Path halving: se modifican los apuntadores de los elementos alternados en el camino desde i hasta $\text{FIND}(i)$ para que apunten a su “abuelo”.



Fuente: Kevin Wayne

(<https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/UnionFind.pdf>)

Se ha demostrado que una secuencia de m UNIONes y n FINDs usando una de estas dos técnicas tiene coste $\mathcal{O}((m + n) \cdot \alpha(m, n))$, donde $\alpha(m, n)$ es una *función inversa de Ackermann*. Crece extremadamente despacio. Puesto que $\alpha(m, n) \leq 4$ para cualesquiera valores de m y n concebibles en la práctica, el coste puede considerarse $\mathcal{O}(m + n)$. Aunque el coste de una UNION o un FIND no es constante, el **coste amortizado** (prácticamente) sí lo es ya que el coste de las $m + n$ operaciones es $\mathcal{O}((m + n) \alpha(m, n))$ en caso peor.