ALGORITMOS VORACES

- **Códigos de Huffman**
- **Dijkstra**

María Teresa Abad

Mayo, 2005

CÓDIGOS DE HUFFMAN

La codificación de Huffman es una técnica para la *compresión de datos* ampliamente usada y muy efectiva

Ejemplo: Fichero con 100.000 caracteres. Se sabe que aparecen 6 caracteres diferentes y la frecuencia de aparición de cada uno de ellos es:

	a	b	c	d	e	f
Frecuencia (en miles)	45	13	12	16	9	5

¿Cómo codificar los caracteres para comprimir el espacio ocupado utilizando un **código binario**?

***Solución 1*:** Código de longitud fija.
Para 6 caracteres se necesitan 3 bits (300000 bits)

Fija	000	001	010	011	100	101

***Solución 2*:** Código de longitud variable en el que los más frecuentes tienen el código más corto. Restricción: ningún código es prefijo de otro.
(224000 bits)

Variable	0	101	100	111	1101	1100

Esta técnica de codificación se denomina **código prefijo**.

⇒ **Codificación**: Basta con concatenar el código de cada uno de los caracteres.

Ejemplo :

aabacd ≡ 0·0·101·0·100·111≡ 001010100111

⇒ **Descodificación**: Fácil porque ningún código es prefijo de otro código ⇒ NO hay ambigüedad.

Ejemplo :

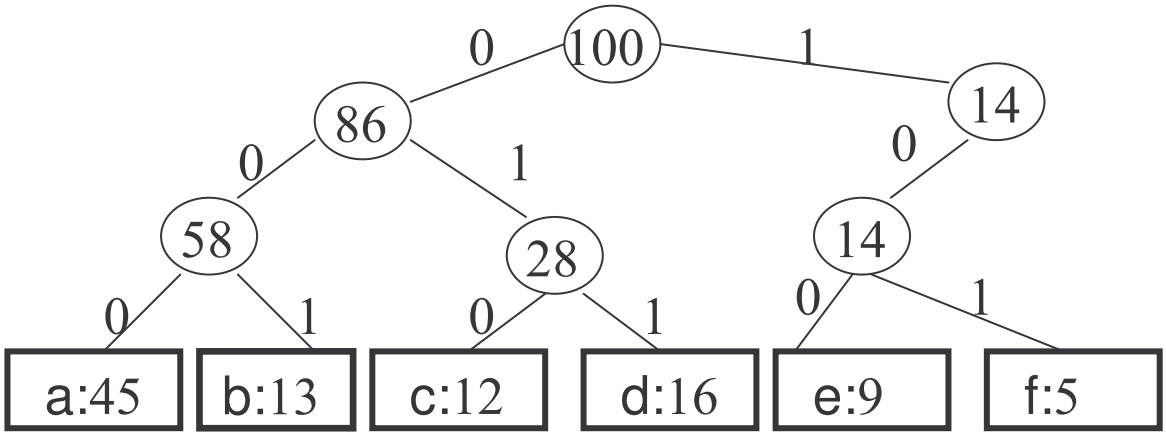
1010111011111011100 ≡ badadcf

¡Es la única posibilidad!

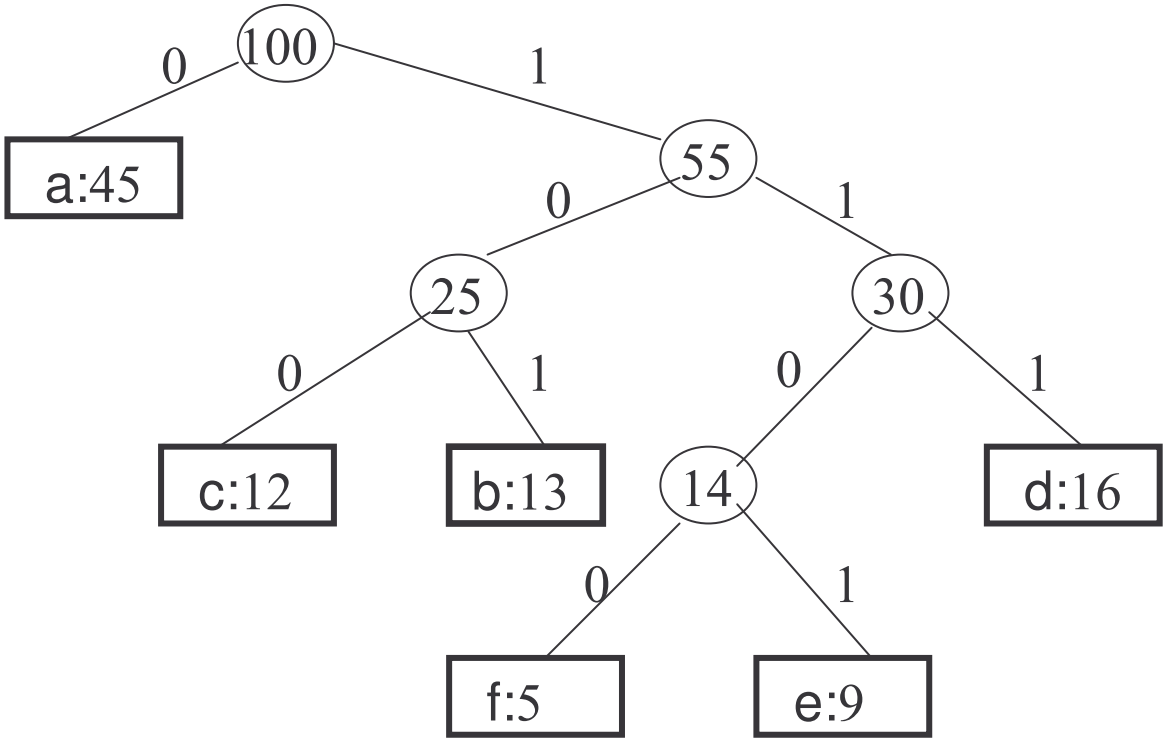
Un árbol binario es una forma de representar el código prefijo que simplifica el proceso de descodificación:

- las *hojas* son los caracteres,
- el *camino* de la raíz a la hojas con la interpretación *0* a la izquierda y *1* a la derecha nos da el código de cada hoja.

Este sería el árbol binario de la codificación de **longitud fija**:



Y éste el de la codificación de **longitud variable**:



Dado ***T***, el árbol binario que corresponde a una codificación prefijo, es fácil averiguar el número de bits necesarios para codificar el fichero:
Para cada carácter *c* diferente del alfabeto *C* que aparece en el fichero,

⇒ sea f*(c)* la frecuencia de *c* en la entrada,

⇒ sea $d_T(c)$ la profundidad de la hoja *c* en el árbol *T*, entonces el número de bits requeridos es :

$$B(T) = \sum_{c \in C} f(c) \cdot d_T(c)$$

B(*T*) nos da el coste de *T*.

Algoritmo Greedy

Huffman propuso un algoritmo voraz que construye una codificación prefijo óptima.

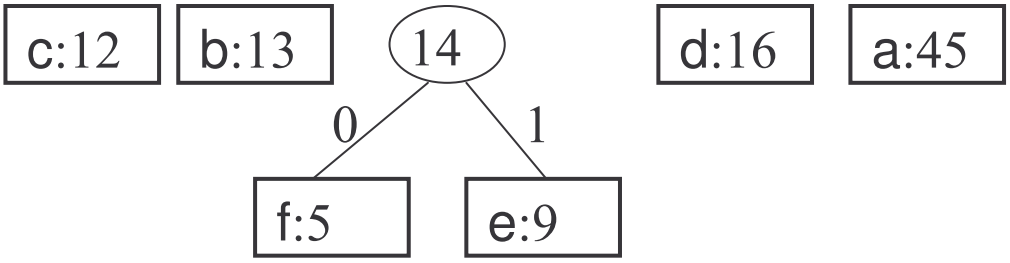
⇒ Construye un árbol binario de códigos de longitud variable de manera ascendente de modo que [MIN] B(*T*).

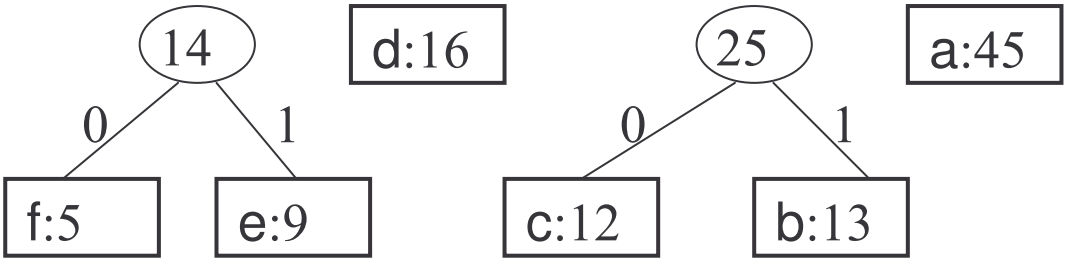
Ejemplo de funcionamiento

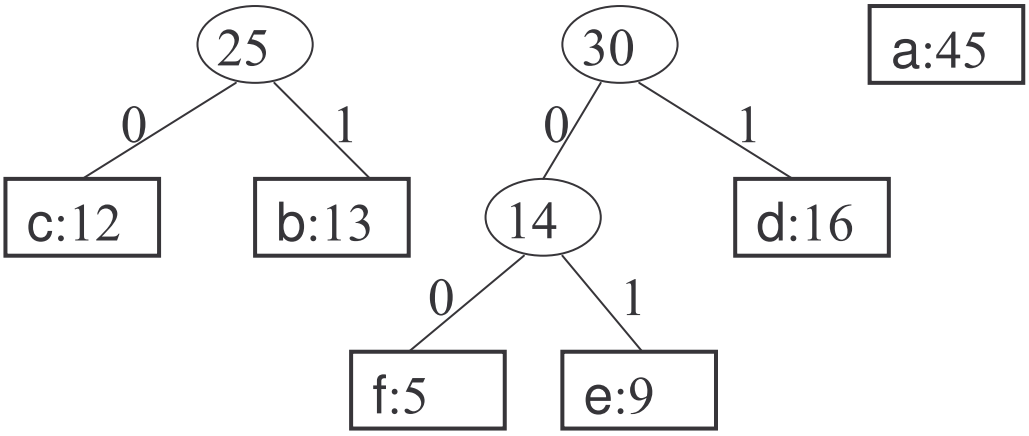
Fase 1: Caracteres colocados en orden creciente de frecuencia.

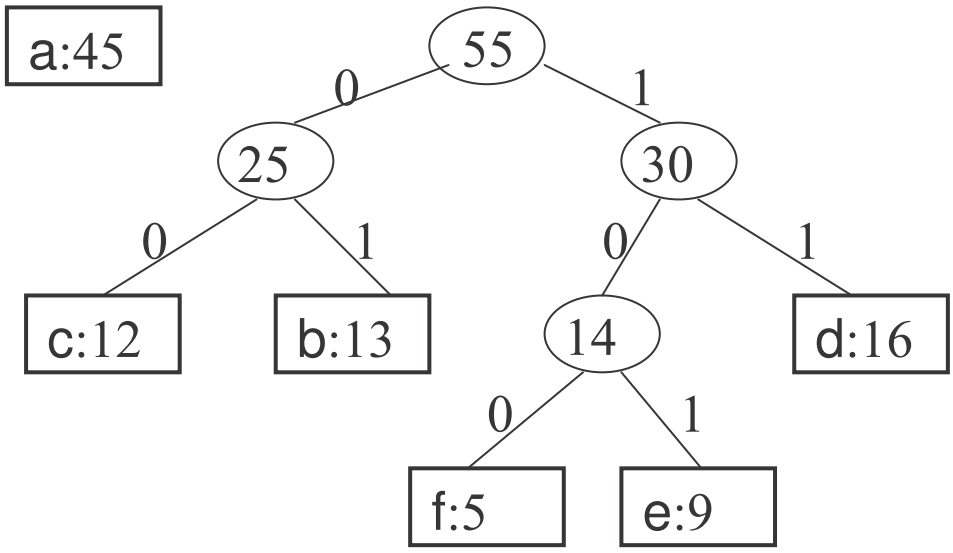
f:5	e:9	c:12	b:13	d:16	a:45

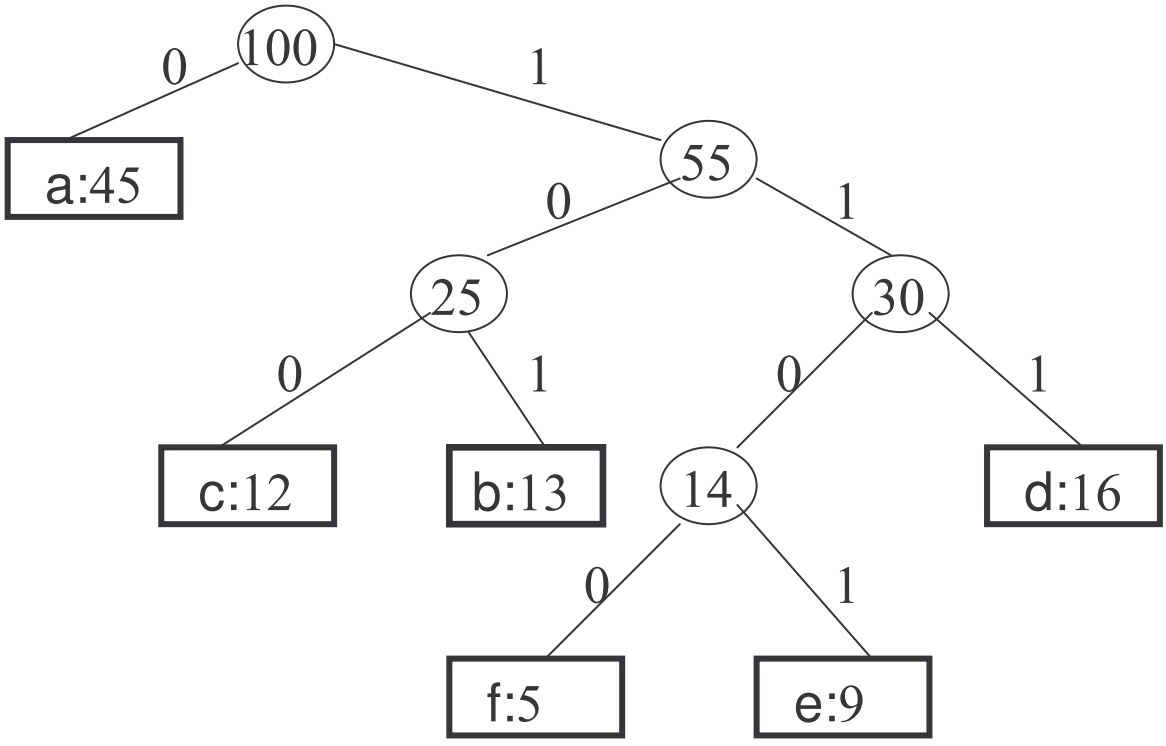
Fase 2 y posteriores: Fusionar hasta obtener un sólo árbol manteniendo la ordenación creciente.








a:45
55
0
1
25
30
0
1
0
1
c:12
b:13
14
d:16
0
1
f:5
e:9


100
0
1
a:45
55
0
1
25
30
0
1
0
1
c:12
b:13
14
d:16
0
1
f:5
e:9

Implementación del algoritmo

Se usa una cola de prioridad, *Q*, con clave la frecuencia lo que permite seleccionar los dos objetos de la cola con la frecuencia más baja.

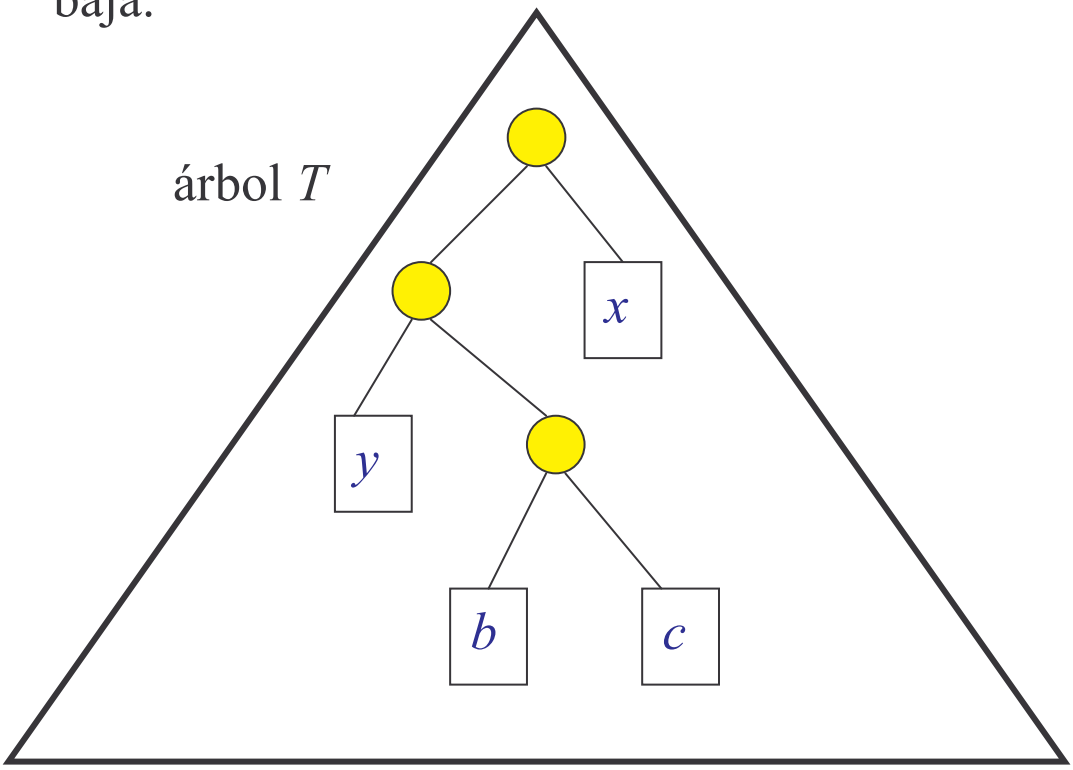
El resultado de fusionar dos objetos es un nuevo objeto cuya frecuencia es la suma de frecuencias de los dos objetos fusionados.

```
función COD_HUF ( C es conj_<car, frec> )
{ Pre : C está bien construido y no es vacío}
  n := |C|; Q es cola_prio;
  Q := Insertar_todos (C);
/* la cola contiene todos los elementos */
  Para i=1 hasta n-1 hacer
       z:= crear_objeto();
       /* elección de los candidatos */
       x := izq(z) := primero(Q);   Q:= avanzar(Q);
       y:= der(z) := primero(Q);    Q:= avanzar(Q);
       frec[z] := frec[x] + frec[y];
       /* actualiza solución */
       Q:= insertar(Q, z);
  fpara
{Post : Q contiene un único elemento que es un
árbol de codificación de prefijo óptimo }
  dev ( primero(Q))
ffunción
```

Coste: $\theta(n.\log n)$

Demostración de optimalidad del criterio

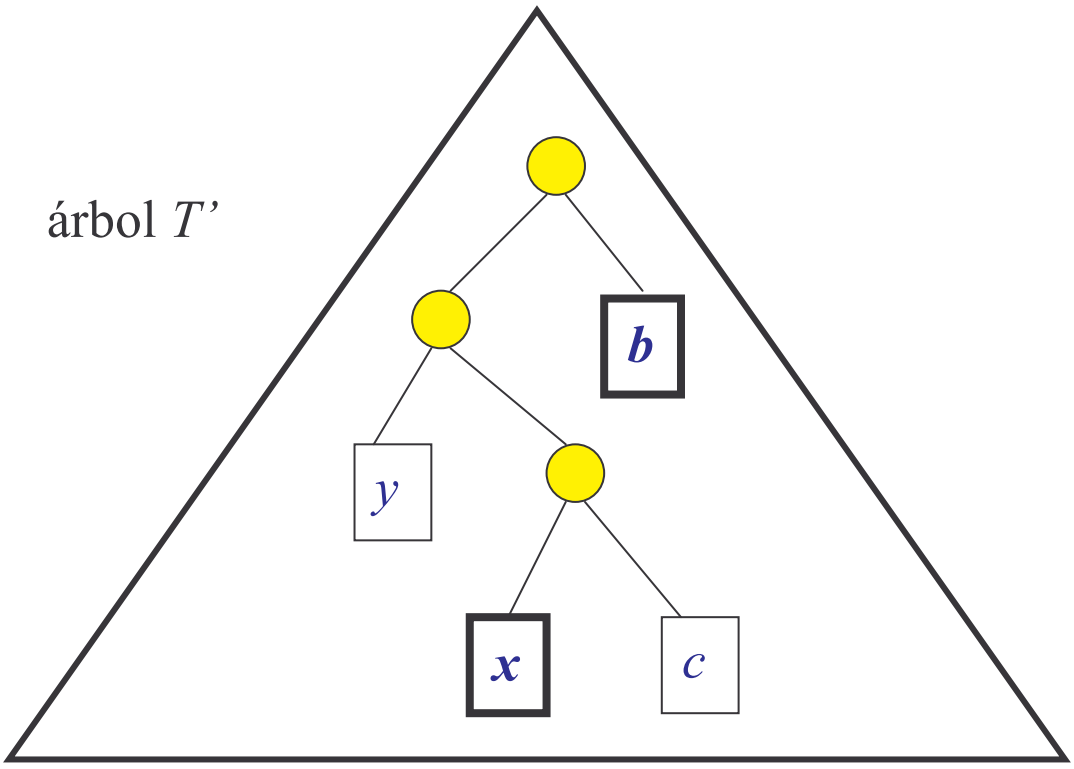
- Sea T un árbol binario de codificación óptimo.
- Sean b y c dos hojas hermanas en T que se encuentran a profundidad máxima.
- Sean x e y dos hojas de T tales que son los 2 caracteres del alfabeto C con la frecuencia más baja.



Vamos a ver que T, que es un árbol óptimo, se puede transformar en otro árbol T'', también óptimo, en el que los 2 caracteres, x e y, con la frecuencia más baja serán hojas hermanas que estarán a la máxima profundidad $\Rightarrow$ El árbol que genera el algoritmo voraz cumple exactamente esa condición.

Podemos suponer que $f[b] \leq f[c]$ y que $f[x] \leq f[y]$. Además, se puede deducir que $f[x] \leq f[b]$ y $f[y] \leq f[c]$.

Se puede construir un nuevo árbol, *T'*, en el que se intercambia la posición que ocupan en *T* las hojas *b* y *x*.



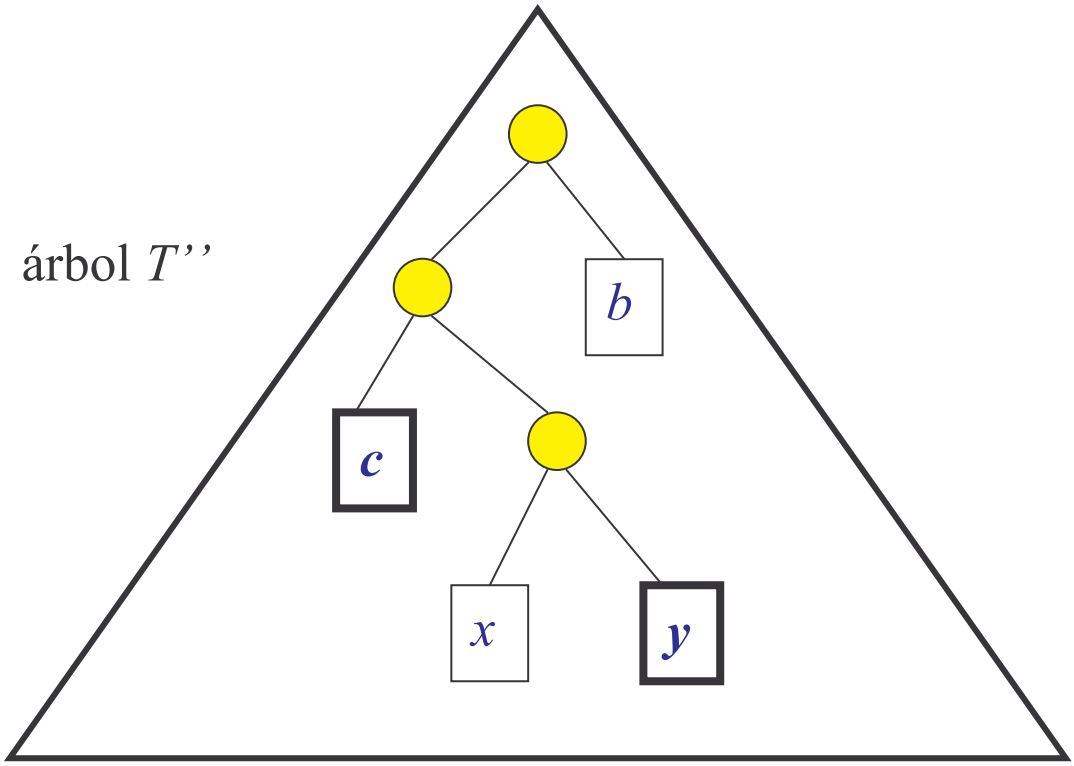
$$B(T) - B(T') = \sum_{c \in C} f[c].d_T(c) - \sum_{c \in C} f[c].d_{T'}(c) =$$

$$= f[x].d_T(x) + f[b].d_T(b) - f[x].d_{T'}(x) - f[b].d_{T'}(b) =$$

$$= f[x].d_T(x) + f[b].d_T(b) - f[x].d_T(b) - f[b].d_T(x) =$$

$$= (f[b] - f[x]) . (d_T(b) - d_T(x)) \geq 0$$

De forma similar, se construye el árbol *T''* intercambiando *c* e *y*.



Con este intercambio tampoco se incrementa el coste y B(*T'*) – B(T'') ≥ 0.
Por tanto, B(*T''*) ≤ B(*T*) y como *T* es óptimo, entonces *T''* también lo es y B(*T''*) = B(*T*).

Y ya para acabar la demostración:
Sea *T* un árbol binario que representa un código prefijo óptimo para un alfabeto *C*. Consideremos 2 hojas hermanas, *x* e *y*, de *T* y sea *z* el padre de ambas. Consideremos que la frecuencia de *z* es f[*z*] = f[*x*] + f[*y*].

Entonces, el árbol $T' = T–\{x,y\}$ representa un código prefijo óptimo para el alfabeto $C' = C – \{x, y\} \cup \{z\}$.

Precisamente eso es lo que hace el algoritmo voraz : una vez que ha fusionado los dos caracteres de frecuencia más baja, inserta un nuevo elemento en el alfabeto con su frecuencia y repite el proceso de seleccionar los dos elementos con frecuencia más baja ahora para un alfabeto con un elemento menos.

CAMINOS MINIMOS

Shortest-paths problem

Definiciones

Sea G=(V, E) un grafo dirigido y etiquetado con valores naturales.

Se define el peso del camino p, con $p=<v_0,v_1,v_2, ...,v_k>$, como la suma de los valores de las aristas que lo componen.

$$\text{peso}(p) = \sum_{i=1}^{k} \text{valor}(G,v_{i-1},v_i)$$

Se define el camino de peso mínimo del vértice u al v en G, con $u,v \in V$, con la siguiente función :

$$\delta(u,v) = \begin{cases} MIN\{\ \text{peso}(p) : u \rightsquigarrow v\ \} & \text{si hay camino de } u \text{ a } v \\ \infty & \text{en otro caso} \end{cases}$$

El camino de peso mínimo, **camino mínimo**, de u a v en G, se define como cualquier camino p tal que $\text{peso}(p) = \delta(u,v)$.

Los problemas de caminos mínimos

1/ *Single_source shortest_paths problem* :

Encontrar el camino de peso mínimo entre un vértice fijado, *source*, y todos los vértices restantes del grafo.

2/ *Single_destination shortest_paths problem*
Encontrar el camino mínimo desde todos los vértices a uno fijado, *destination*.

3/ *Single_pair shortest_paths problem*
Fijados dos vértices del grafo, *source* y *destination*, encontrar el camino mínimo entre ellos.

4/ *All_pairs shortest_paths problem*
Encontrar el camino mínimo entre todo par de vértices.

El algoritmo de Dijkstra (1959)
Resuelve el problema de encontrar el camino mínimo entre un vértice dado y todos los restantes del grafo.

Funciona de la siguiente manera:
- El conjunto de vértices del grafo se halla distribuido en dos conjuntos disjuntos: el conjunto de VISTOS y el de V–VISTOS.
- En VISTOS están los vértices para los que ya se conoce cual es la longitud del camino más corto entre el vértice inicial y él.
- En el conjunto V–VISTOS están los vértices restantes y de ellos se guarda la siguiente información :
  ($\forall u$: $u \in$ V–VISTOS: D[u] contiene la longitud del camino más corto desde el vértice inicial a u que no sale de VISTOS),
- En cada iteración se elige el vértice v de V–VISTOS con D[v] mínima de modo que v deja V–VISTOS, pasa

a VISTOS y la longitud del camino más corto entre el vértice inicial y *v* es, precisamente, D[*v*].
Todos aquellos vértices *u* de V–VISTOS que sean sucesores de *v* actualizan convenientemente su D[*u*].

- El algoritmo acaba cuando todos los vértices están en VISTOS

función **DIJKSTRA** (g es grafo; v_ini es vértice)
dev (D es vector[1..n] de naturales)

*{**Pre** : g=(V,E) es un grafo etiquetado con valores naturales dirigido. Se supone que el grafo está implementado en una matriz y que M[i,j] contiene el valor de la arista que va del vértice i al j y, si no hay arista, contiene el valor infinito }*

```
Para cada v∈ V hacer D[v] := M[v_ini, v] fpara;
D[v_ini]:= 0;
VISTOS := añadir( conjunto_vacio, v_ini );
```

*{ **Inv** : ∀u : u∈ **V–VISTOS**: D[u] contiene la longitud del camino más corto desde v_ini a u que no sale de VISTOS, es decir, el camino está formado por v_ini y una serie de vértices todos ellos pertenecientes a VISTOS, excepto el propio u. ∀u: u∈**VISTOS** : D[u] contiene la longitud del camino más corto desde v_ini a u }*

```
    *[ |VISTOS| < |V| --->
        u := MINIMO( D, u∈ V−VISTOS );
        /* el candidato es el vértice u∈V-VISTOS que
        tiene D mínima */
        VISTOS := VISTOS ∪{u};
        Para cada v ∈ suc(g,u) t.q. v ∈ V−VISTOS hacer
            /* Ajustar D : Recalcular los valores para
            aplicar la función de selección sobre los
            candidatos que quedan */
            [  D[v] > D[u] + valor( g,u,v ) --->
                        D[v]:= D[u] + valor( g,u,v );
            [] D[v] ≤ D[u] + valor( g,u,v ) ---> seguir
            ]
        fpara;
    ]
{ Post : ∀u:u∈V: D[u] contiene la longitud del camino
de peso mínimo desde v_ini a u que no sale de VISTOS,
y como VISTOS ya contiene todos los vértices, se tienen
en D las distancias mínimas definitivas }
        dev ( D )
ffunción
```

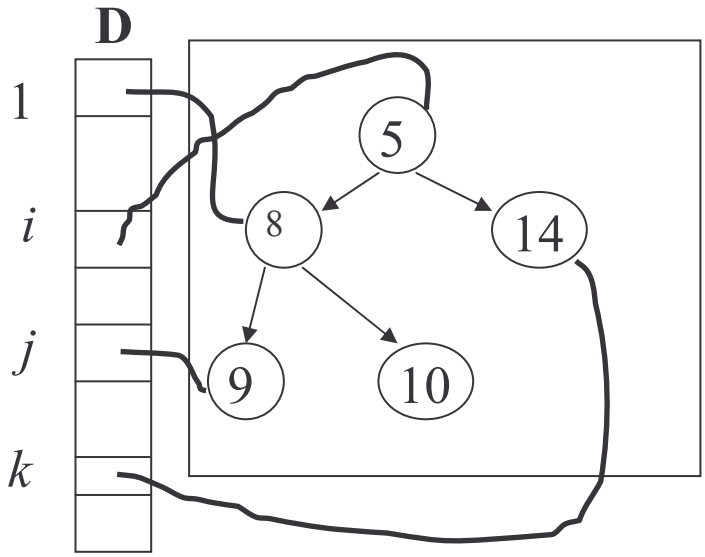
Coste

Grafo implementado en matriz: $\theta(n^2)$:

- El bucle que inicializa el vector D cuesta $\theta(n)$.
- El bucle principal efectúa n-1 iteraciones.
- Coste de cada iteración: la obtención del mínimo, coste lineal porque hay que recorrer D, y el ajuste de D que también es $\theta(n)$.

Grafo implementado con listas + Heap de mínimos $\theta((n+|E|)\cdot\log n)$:

- La construcción del Heap $\theta(n)$,
- Seleccionar el mínimo cuesta $\theta(1)$,
- Cada operación de ajuste requerirá $\theta(\log n)$.
- En total se efectúan $\theta(|E|)$ operaciones de ajustar D.



Cola prioridad implementada en un min-heap

Implementación del min-heap sobre un vector:

5	8	14	9	10		
i	1	*k*	*j*			

En la primera fila tenemos el valor de D.
En la segunda fila guardamos el vértice al que corresponde ese valor de D.

Algoritmo de Dijkstra empleando una cola de prioridad extendida (se puede acceder a cualquier elemento de la cola en θ(1) y modificar su prioridad).

```
función DIJKSTRA ( g es grafo; v_ini es vértice ) dev
( D es vector[1..n] de naturales )

var C : cola_prioridad;
        Para cada v∈ V hacer D[v] := M[v_ini, v] fpara;
        D[v_ini]:= 0;
        crear(C);
        insertar(C, <v_ini, D[v_ini]>);

        *[  ¬vacia(C) --->
             u := primero(C);
             avanzar(C);
             Para cada  v∈  suc(g,u)  hacer
                [D[v] ≥ D[u]+valor(g,u,v) --->
                          D[v] := D[u]+valor(g,u,v);
                          [   esta(C,v) --->
                                sustituir(C,<v,D[v]>);
                          [] ¬esta(C,v) --->
                                insertar(C,<v,D[v]>);
                          ]
                [] D[v] < D[u]+valor(g,u,v) ---> seguir;
                ]
             fpara;
          ]
        dev ( D )
ffunción
```

Demostración

Sea u un vértice tal que $u \in$ V−VISTOS.

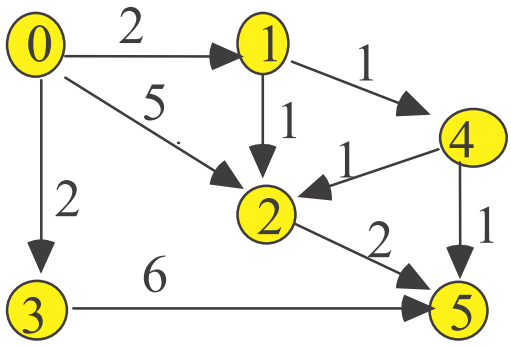
Supongamos que D[u] contiene información cierta, es decir, contiene la distancia mínima entre el vértice inicial y u siguiendo por un camino que sólo contiene vértices que pertenecen a VISTOS.

- Si u es el vértice con el valor de D más pequeño, el criterio de selección lo elegirá como candidato e inmediatamente pasará a formar parte de VISTOS y se considerará que su D[u] es una distancia definitiva.

Supongamos que no es CIERTO, es decir, que existe un camino más corto, aún no considerado, desde el vértice inicial a u que pasa por v, obviamente $v \in$ V−VISTOS. Si v se encuentra en el camino más corto desde el vértice inicial a u, ¡es que D[v]<D[u]! lo que contradice la elección de u.

$\Rightarrow$ siempre que D contenga información correcta, la función de selección elige un vértice con un valor de D ya definitivo (ninguno de los vértices que no están en VISTOS lograrán que se reduzca).

Ejemplo de funcionamiento



Vértice inicial: vértice con etiqueta 0.

D:	0	1	2	3	4	5	**VISTOS**
	0	2	5	2	∞	∞	{0}
	0	***2***	3	2	3	∞	{0, 1}
	0	***2***	3	***2***	3	8	{0, 1, 3}
	0	***2***	***3***	***2***	3	5	{0, 1, 3, 2}
	0	***2***	***3***	***2***	***3***	4	{0, 1, 3, 2 ,4}
	0	***2***	***3***	***2***	***3***	***4***	{0, 1, 3, 2 ,4 ,5}

Reconstrucción de caminos mínimos

Es necesario saber qué vértices forman parte del camino de longitud mínima que sale del vértice inicial.

```
función DIJ_CAM ( g es grafo; v_ini es vértice )
dev ( D, CAMINO es vector[1..n] de natural )
{ Pre : la misma que DIJKSTRA }
    Para cada v∈ V hacer
        D[v] := M[v_ini, v]; CAMINO[v]:= v_ini;
    fpara;
    D[v_ini]:= 0;
    VISTOS := añadir( conjunto_vacio, v_ini );
    *[ |VISTOS| < |V| --->
        u := MINIMO( D, u∈ V−VISTOS );
        VISTOS := VISTOS∪{u};
        Para cada v ∈ suc(g,u) t.q. v ∈ V−VISTOS hacer
            [  D[v] > D[u] + valor( g,u,v ) --->
                        D[v] := D[u] + valor( g,u,v );
                        CAMINO[v]:= u;
            [] D[v] ≤ D[u] + valor ( g,u,v ) ---> seguir
            ]
        fpara;
     ]
{ Post : La misma de Dijkstra y ∀u : u∈V: CAMINO[u]
contiene el vértice del que sale la última arista del
camino mínimo que va de v_ini a u}
        dev ( D, CAMINO )
ffunción
```

Ahora se puede utilizar un procedimiento recursivo, **RECONSTRUIR**, que recibe el vértice *v* para el cual se quiere reconstruir el camino mínimo desde *v_ini* y el vector CAMINO.

```
función RECONSTRUIR ( v, v_ini es vértice;
        CAMINO es vector[1..n] de vértices )
        dev ( s es secuencia_aristas )
        [  v = v_ini ---> s := secuencia_vacia;
        [] v ≠ v_ini --->
                u := CAMINO[v];
                s := RECONSTRUIR( u,v_ini,CAMINO);
                s := concatenar( s, (u, v) );
        ]
{ Post : s contiene las aristas del camino mínimo desde
v_ini a v }
        dev ( s )
ffunción
```