Algoritmos voraces

Este documento está sujeto a una licencia de uso Creative Commons. No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original.

Algoritmos voraces

- Introducción y primer ejemplo 3
- El problema de la mochila 9
- Caminos mínimos en grafos 17
- Árboles de recubrimiento de coste mínimo 39
- Consideraciones sobre la corrección del esquema voraz 55
- Códigos de Huffman 58
- El problema de la selección de actividades 66
- El problema de la minimización del tiempo de espera 71
- Fundamentos teóricos del esquema voraz 75
- Un problema de planificación de tareas a plazo fijo 85
- Heurísticas voraces 91
  - Coloreado de grafos 91
  - El problema del viajante de comercio 94

El esquema voraz: Introducción y primer ejemplo

- Es uno de los esquemas más simples y al mismo tiempo de los más utilizados.
- Típicamente se emplea para resolver **problemas de optimización**:
  - existe una entrada de tamaño n que son los **candidatos** a formar parte de la solución;
  - existe un subconjunto de esos n candidatos que satisface ciertas restricciones: se llama **solución factible**;
  - hay que obtener la solución factible que maximice o minimice una cierta función objetivo: se llama **solución óptima**.
- Ejemplos:
  - encontrar la secuencia óptima para procesar un conjunto de tareas por un computador,
  - hallar un camino mínimo en un grafo,
  - problema de la mochila,…

El esquema voraz: Introducción y primer ejemplo

- El **esquema voraz** procede por pasos:
  - inicialmente el conjunto de candidatos escogidos es vacío;
  - en cada paso, se intenta añadir al conjunto de los escogidos "el mejor" de los no escogidos (sin pensar en el futuro), utilizando una **función de selección** basada en algún criterio de optimización (puede ser o no ser la función objetivo);
  - tras cada paso, hay que ver si el conjunto seleccionado es **completable** (i.e., si añadiendo más candidatos se puede llegar a una solución);
    - si el conjunto no es completable, se rechaza el último candidato elegido **y no se vuelve a considerar en el futuro**;
    - si es completable, se incorpora al conjunto de escogidos **y permanece siempre en él**;
  - tras cada incorporación se comprueba si el conjunto resultante es una solución;
  - el algoritmo termina cuando se obtiene una solución;
  - el algoritmo es correcto si la solución encontrada es siempre óptima;

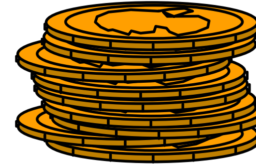
El esquema voraz: Introducción y primer ejemplo

- Esquema genérico:

```
función voraz(C:conjunto) devuelve conjunto
{C es el conjunto de todos los candidatos}
principio
  S:=Ø; {S es el conjunto en el que se construye la solución}
  mq ¬solución(S) ∧ C≠Ø hacer
    x:=elemento de C que maximiza seleccionar(x);
    C:=C-{x};
    si completable(S∪{x})
      entonces S:=S∪{x}
    fsi
  fmq;
  si solución(S)
    entonces devuelve S
    sino devuelve no hay solución
  fsi
fin
```

El esquema voraz: Introducción y primer ejemplo

- Problema del cambio en monedas.
  - Se trata de devolver una cantidad de euros con el menor número posible de monedas.



  - Se parte de:
    - un conjunto de tipos de monedas válidas, de las que se supone que hay cantidad suficiente para realizar el desglose, y de
    - un importe a devolver.
- Elementos fundamentales del esquema:
  - **Conjunto de candidatos**: cada una de las monedas de los diferentes tipos que se pueden usar para realizar el desglose del importe dado.
  - **Solución factible**: un conjunto de monedas devuelto tras el desglose y cuyo valor total es igual al importe a desglosar.
  - **Completable**: la suma de los valores de las monedas escogidas en un momento dado no supera el importe a desglosar.
  - **Función de selección**: elegir si es posible la moneda de mayor valor de entre las candidatas.
  - **Función objetivo**: número total de monedas utilizadas en la solución (debe minimizarse).

El esquema voraz: Introducción y primer ejemplo

- Solución:

```
tipo moneda=(M25,M10,M5,M1)    {por ejemplo}
```

```
función cambia(importe:nat; valor:vector[moneda] de nat)
        devuelve vector[moneda] de nat
variables mon:moneda; cambio:vector[moneda] de nat
principio
  para todo mon en moneda hacer
    cambio[mon]:=0
  fpara;
  para mon:=M25 hasta M1 hacer
    mq valor[mon]≤importe hacer
      cambio[mon]:=cambio[mon]+1;
      importe:=importe-valor[mon]
    fmq
  fpara;
  devuelve cambio
fin
```

El esquema voraz: Introducción y primer ejemplo

- Problemas sobre el problema del cambio en monedas:
  - Demostrar la corrección del algoritmo.
  - Demostrar, buscando contraejemplos, que el algoritmo no es óptimo si se añade un nuevo tipo de moneda de 12 euros o si se elimina alguno de los tipos existentes.
    Demostrar que, en esas condiciones, el algoritmo puede incluso no encontrar solución alguna aunque ésta exista.
- Otro problema:
  - ¿Es el método de ordenación por selección directa un algoritmo voraz?

    Si es así, ¿cuáles son las funciones utilizadas (“selección”, “completable”, “solución”)?

El problema de la mochila

- Se tienen n objetos fraccionables y una mochila.
- El objeto i tiene peso p_i, $1 \leq i \leq n$.
- La mochila tiene capacidad C.
- Si se mete una fracción x_i, $0 \leq x_i \leq 1$, del objeto i en la mochila, entonces se consigue un beneficio $b_i x_i$.
- El objetivo es llenar la mochila de manera que se maximice el beneficio total.
- Pero como la mochila tiene capacidad C, el peso total de todos los objetos metidos en ella no puede superar esa cantidad.

El problema de la mochila

- Formalmente:

$$\text{maximizar} \sum_{1 \le i \le n} b_i x_i \qquad (1)$$

$$\text{sujeto a} \sum_{1 \le i \le n} p_i x_i \le C \qquad (2)$$

$$\text{con } 0 \le x_i \le 1,\ b_i > 0,\ p_i > 0,\ 1 \le i \le n \qquad (3)$$

- Una solución factible es cualquier n-tupla $(x_1,\ldots,x_n)$ que satisfaga (2) y (3).
- Una solución óptima es cualquier solución factible para la que (1) sea máximo.
- Si $p_1 + \cdots + p_n \le C$, entonces obviamente $x_i = 1$, $1 \le i \le n$, es una solución óptima.
- Por tanto, supongamos que $p_1 + \cdots + p_n > C$.
- Nótese además que todas las soluciones óptimas llenarán la mochila por completo (podemos poner ‘=‘ en (2)).

El problema de la mochila

- Ejemplo:

  n=3 C=17

  $(b_1,b_2,b_3)=(40,36,22)$

  $(p_1,p_2,p_3)=(15,12,8)$

- Tres soluciones factibles (entre otras muchas):

	(x_1,x_2,x_3)	$\sum_{1\le i\le 3} b_i x_i$
(i)	(1,1/6,0)	46
(ii)	(0,3/4,1)	49
(iii)	(0,1,5/8)	49’75

El problema de la mochila

- Solución voraz:
  - El conjunto de candidatos son los objetos, tomándose de ellos cierta fracción.
  - Un conjunto de candidatos es completable si la suma de sus pesos no supera la capacidad de la mochila, y es una solución si iguala dicha capacidad.
  - La función objetivo a maximizar es $\sum_{1 \le i \le n} b_i x_i$.
  - La función de selección es la más difícil de determinar.
    - Si procedemos vorazmente, en cada paso debemos considerar un objeto y tomar cierta fracción suya.
    - La cuestión de qué fracción se toma es más fácil de resolver: si hemos elegido el mejor candidato, tomamos todo lo que podamos de él.

El problema de la mochila

- ¿Cuál es el mejor candidato (es decir, la mejor función de selección)?
- Volvamos al ejemplo:
  - Primera estrategia: elegir el objeto con mayor beneficio total (el primero). Sin embargo, la mochila se llena muy rápidamente con poco beneficio total.
  - Segunda estrategia: elegir el objeto que llene menos la mochila, para acumular beneficios de un número mayor de objetos. Sin embargo, es posible que se elija un objeto con poco beneficio simplemente porque pesa poco.
  - La tercera estrategia, que es la óptima, es tomar siempre el objeto que proporcione mayor **beneficio por unidad de peso**.
- Los algoritmos resultantes de aplicar cualquiera de las dos primeras estrategias también son voraces, pero no calculan la solución óptima.

El problema de la mochila

```
constante n=... {número de objetos}
tipo vectReal=vector[1..n] de real
```

```
{Pre:∀i∈1..n:peso[i]>0 ∧
     ∀i∈1..n-1:benef[i]/peso[i]≥benef[i+1]/peso[i+1]}
función mochila(benef,peso:vectReal; cap:real) devuelve vectReal
variables resto:real; i:entero; sol:vectReal
principio
  para todo i en 1..n hacer
    sol[i]:=0.0 {inicializar solución}
  fpara;
  resto:=cap; {capacidad restante}
  i:=1;
  mq (i≤n) and (peso[i]≤resto) hacer
    sol[i]:=1; resto:=resto-peso[i]; i:=i+1
  fmq;
  si i≤n entonces sol[i]:=resto/peso[i] fsi;
  devuelve sol
fin
{Post: sol es solución óptima del problema de la mochila}
```

El problema de la mochila

- Coste temporal: $\Theta(n)$

  $\Theta(n\log n)$ si se tiene en cuenta que hay que ordenar primero los vectores.

- Demostración de la corrección:

  Sea $X=(x_1,\ldots,x_n)$ la solución generada por el algoritmo voraz.

  Si todos los x_i son iguales a 1, la solución es claramente óptima.

  Luego, sea j el menor índice tal que $x_j \neq 1$.

  Por tanto $x_i=1$ para $1 \leq i < j$, $x_i=0$ para $j < i \leq n$, y $0 \leq x_j < 1$.

  Si X no es óptima ent. existe otra solución factible $Y=(y_1,\ldots,y_n)$ tal que $\sum b_i y_i > \sum b_i x_i$.

  Sin perdida de generalidad, asumimos que $\sum p_i y_i = C$.

  Sea k el menor índice tal que $y_k \neq x_k$ (claramente, existe ese k).

  Se verifica: $y_k < x_k$. En efecto:

  - Si $k<j$ entonces $x_k=1$. Pero $y_k \neq x_k$ luego $y_k < x_k$.
  - Si $k=j$ ent., puesto que $\sum p_i y_i = C$ y $y_i = x_i$ para $1 \leq i < j$, se sigue que $y_k < x_k$ ó $\sum p_i y_i > C$.
  - Si $k>j$ entonces $\sum p_i y_i > C$, y eso no es posible.

El problema de la mochila

Ahora, supongamos que aumentamos y_k hasta x_k y disminuímos tantos $(y_{k+1},\ldots,y_n)$ como sea necesario para que la capacidad utilizada siga siendo C.

Se obtiene así una nueva solución $Z=(z_1,\ldots,z_n)$

con $z_i=x_i$, para $1\leq i\leq k$ y $\sum_{k<i\leq n} p_i(y_i - z_i)=p_k(z_k - y_k)$.

Entonces:

$$\begin{aligned}\sum_{1\leq i\leq n} b_i z_i &= \sum_{1\leq i\leq n} b_i y_i + (z_k - y_k)p_k b_k/p_k - \\ &\quad - \sum_{k<i\leq n} (y_i - z_i)p_i b_i/p_i \\ &\geq \sum_{1\leq i\leq n} b_i y_i + \\ &\quad + [(z_k - y_k)p_k - \sum_{k<i\leq n} (y_i - z_i)p_i] b_k/p_k \\ &= \sum_{1\leq i\leq n} b_i y_i\end{aligned}$$

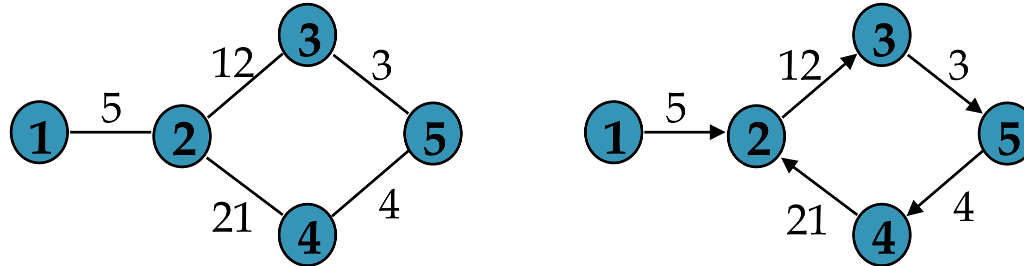
Si $\sum_{1\leq i\leq n} b_i z_i > \sum_{1\leq i\leq n} b_i y_i$ entonces Y no puede ser óptima.

Si esas sumas son iguales entonces bien $Z=X$ y X es óptima, o bien $Z\neq X$.

En este último caso, se usa de nuevo el mismo argumento hasta demostrar que bien Y no es óptima o bien Y se transforma en X, demostrando que X también es óptima. ■

Caminos mínimos en grafos

- Grafos etiquetados con pesos no negativos

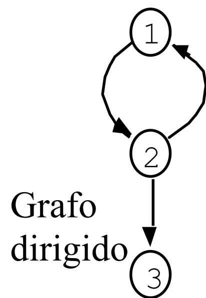


- Búsqueda de caminos de longitud mínima
  - longitud o coste de un camino: suma de los pesos de las aristas que lo componen
  - cálculo de la longitud mínima de los caminos existentes entre dos vértices dados
- Características del tipo de dato de las etiquetas (o pesos):
  - dominio ordenado ($<$, $=$, $\leq$)
  - operación suma ($+$):
    - conmutativa
    - con elemento neutro (0), que es además el peso más pequeño posible
    - con elemento idempotente (∞), que es además el peso más grande posible

Caminos mínimos en grafos

- Representaciones de grafos: Matriz de adyacencia
  - Grafo dirigido $G=(V,A)$ con $V=\{1,2,\ldots,n\}$.
  - La matriz de adyacencia para G es una matriz A de dimensiones $n \times n$ de elementos booleanos en la que $A[i,j]$ es verdad si y sólo si existe una arista en G que va del vértice i al vértice j.

```
constante n = ... {cardinal de V}
tipo grafo = vector[1..n,1..n]de booleano
```



Grafo dirigido

$$A = \begin{bmatrix} \text{falso} & \text{verdad} & \text{falso} \\ \text{verdad} & \text{falso} & \text{verdad} \\ \text{falso} & \text{falso} & \text{falso} \end{bmatrix}$$

Matriz de adyacencia

  - En el caso de un grafo no dirigido, la matriz de adyacencia tiene la particularidad de ser simétrica.

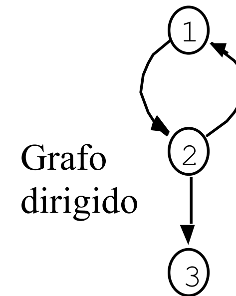
Caminos mínimos en grafos

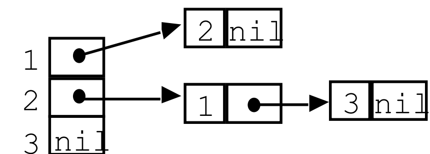
- La representación de la matriz de adyacencia es útil para aquellos algoritmos que precisan saber si existe o no una arista entre dos vértices dados.
- En el caso de **grafos etiquetados** con pesos, los elementos de la matriz pueden ser enteros o reales (en lugar de booleanos) y representar el peso de la arista. Si no existe una arista de i a j, debe emplearse un valor que no pueda ser una etiqueta válida para almacenarse en $A[i,j]$.
- Un inconveniente de la representación de la matriz de adyacencia es que requiere un espacio en $\Theta(n^2)$ aunque el grafo tenga muy pocas aristas.
- Dado un grafo $G=(V,A)$ representado mediante su matriz de adyacencia, la pregunta ¿$(u,v)\in A$? se puede contestar en un tiempo en $\Theta(1)$.
- Sin embargo, la operación sucesores(g,v) necesita un tiempo en $\Theta(n)$ (pues hay que recorrer una fila completa de la matriz).
- Más aún, para saber, por ejemplo, si un grafo no dirigido representado mediante su matriz de adyacencia es conexo, o simplemente para conocer el número de aristas, los algoritmos requieren un tiempo en $\Theta(n^2)$, lo cual es más de lo que cabría esperar.

Caminos mínimos en grafos

- Listas de adyacencia
  - Esta representación sirve para mejorar la eficiencia de los algoritmos sobre grafos que necesitan acceder a los vértices sucesores a uno dado.
  - Consiste en almacenar *n* listas enlazadas mediante punteros de forma que la lista *i*-ésima contiene los vértices sucesores del vértice *i*.

```
constante n = ... {cardinal de V}
tipos ptNodo = ↑nodo;
      nodo = registro
               vértice:1..n;
               sig:ptNodo
             freg
      grafo = vector[1..n] de ptNodo
```





Vector de listas de adyacencia

Caminos mínimos en grafos

– La representación de un grafo con listas de adyacencia requiere un espacio del orden del máximo entre n (el número de vértices) y a (el nº de aristas).

– En el caso de un grafo no dirigido, si $(u,v) \in A$ entonces el vértice v estará en la lista correspondiente al vértice u, y el vértice u lo estará a su vez en la lista de adyacencia del vértice v.

– Para representar un grafo etiquetado con pesos basta con añadir otro campo al tipo nodo que almacene el peso correspondiente.

– Acceder a la lista de los vértices sucesores de uno dado lleva un tiempo constante.

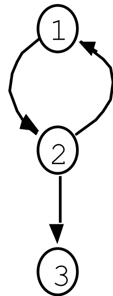
– Sin embargo, consultar si una determinada arista (u,v) está en el grafo precisa recorrer la lista asociada al vértice u, lo que en el caso peor se realiza en $\Theta(n)$.

Caminos mínimos en grafos

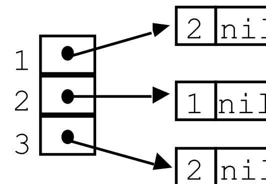
– En el caso de un grafo dirigido, si se necesita saber los predecesores de un vértice dado, la representación con listas de adyacencia no es adecuada.

  Ese problema puede superarse si se almacena otro vector de n listas de forma que la lista i-ésima contenga los vértices predecesores del vértice i.

  Estas listas se llaman listas de adyacencia inversa.



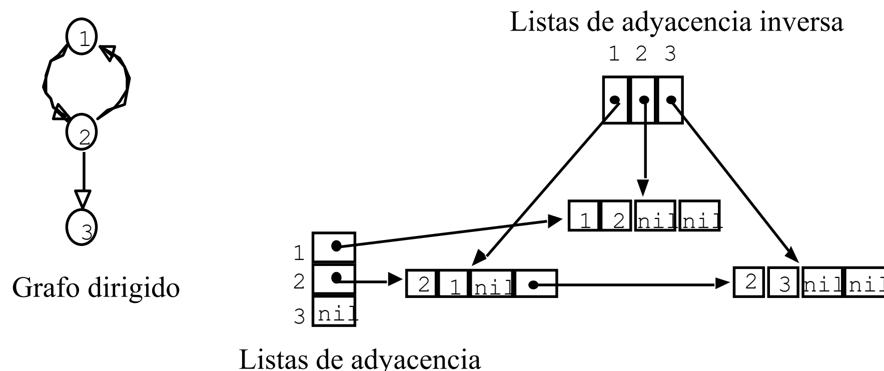
Grafo dirigido



Vector de listas de adyacencia inversa

Caminos mínimos en grafos

- Listas múltiples de adyacencia
  - En el caso de grafos dirigidos, las listas de adyacencia directa e inversa pueden representarse de forma más compacta…
  - Una lista múltiple es una estructura dinámica de datos enlazados mediante punteros en la que cada dato puede pertenecer a dos o más listas.
  - Hay un nodo por cada una de las aristas del grafo.
  - Cada nodo guarda la clave de dos vértices (origen y destino de la arista) y dos punteros: el primero al nodo que guarda la siguiente arista que tiene el mismo vértice destino y el segundo al nodo que guarda la siguiente arista que tiene el mismo vértice origen.



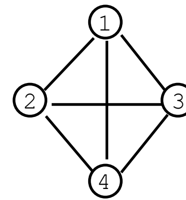
Caminos mínimos en grafos

```
constante n = ... {cardinal de V}
tipos ptNodo = ↑nodo;
      nodo = registro
               origen,destino: 1..n;
               sigInvAdy,sigAdy: ptNodo
             freg
      grafoDirigido =
           = registro
               adyacentes,invAdyacentes: vector[1..n] de ptNodo
             freg
```

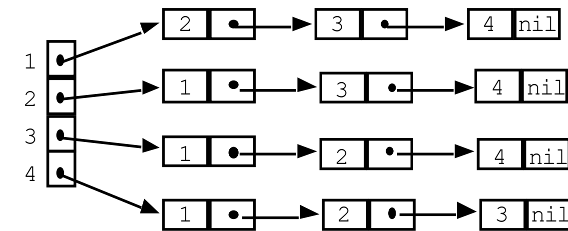
Nótese que se emplea la palabra “adyacente” con el significado de “sucesor”, mientras que “adyacente inverso” se refiere a “predecesor”.

Caminos mínimos en grafos

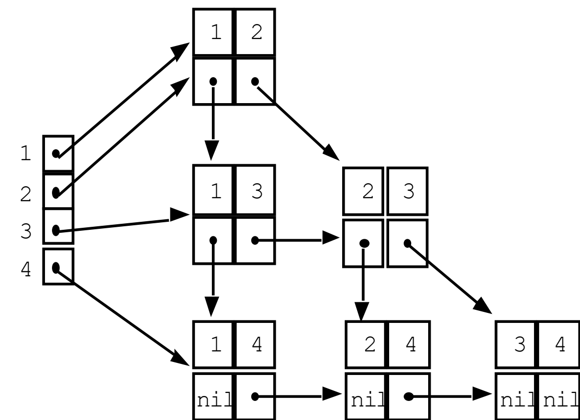
- Las listas múltiples pueden utilizarse también para representar grafos no dirigidos.
- De esta forma, en lugar de que a cada arista le correspondan dos nodos en la estructura dinámica (como ocurre con la representación basada en listas " simples" de adyacencia), puede representarse cada arista con un solo nodo y hacer que éste pertenezca a dos listas de adyacencia diferentes.



Grafo no dirigido



Listas de adyacencia



Listas múltiples de adyacencia

Caminos mínimos en grafos

– La definición de la estructura de datos es la siguiente:

```
constante n = ... {cardinal de V}
tipos ptNodo = ↑nodo;
      nodo = registro
                v1,v2:1..n;
                sig1,sig2:ptNodo
             freg
      grafoNoDirigido = vector[1..n] de ptNodo
```

– Una ventaja de esta representación es que es útil para implementar aquellos algoritmos que necesitan recorrer un grafo y al mismo tiempo "marcar" las aristas por las que se pasa (bastaría con añadir un campo booleano al tipo nodo).

Caminos mínimos en grafos

- Cálculo del coste del camino mínimo desde un vértice dado al resto, en un grafo etiquetado con pesos no negativos

- Utilidad:

  el grafo representa una distribución geográfica, donde las aristas dan el coste (precio, distancia...) de la conexión entre dos lugares y se desea averiguar el camino más corto (barato...) para llegar a un punto partiendo de otro

Caminos mínimos en grafos

E.W. Dijkstra:
"A note on two problems in connexion with graphs",
Numerical Mathematica, 1, pp. 269-271, 1959.

- Solución voraz: Algoritmo de Dijkstra
  - para grafos dirigidos (la extensión a no dirigidos es inmediata)
  - genera uno a uno los caminos de un nodo v al resto por orden creciente de longitud
  - usa un conjunto de vértices donde, a cada paso, se guardan los nodos para los que ya se sabe el camino mínimo
  - devuelve un vector indexado por vértices: en cada posición w se guarda el coste del camino mínimo que conecta v con w
  - cada vez que se incorpora un nodo a la solución se comprueba si los caminos todavía no definitivos se pueden acortar pasando por él
  - se supone que el camino mínimo de un nodo a sí mismo tiene coste nulo
  - valor ∞ en la posición w del vector indica que no hay ningún camino desde v a w

Caminos mínimos en grafos

```
{Pre: g es un grafo dirigido etiquetado no neg.}
función Dijkstra(g:grafo; v:vért) devuelve vector[vért] de etiq
variables S:cjtVért; D:vector[vért] de etiq
principio
  ∀w∈vért:D[w]:=etiqueta(g,v,w);
  D[v]:=0; S:={v};
  mq S no contenga todos los vértices hacer
  {D contiene caminos mínimos formados íntegramente por nodos de S
   (excepto el último), y los nodos de S corresponden a los caminos
   mínimos más cortos calculados hasta el momento}
    elegir w∉S t.q. D[w] es mínimo;
    S:=S∪{w};
    ∀u∉S:actualizar dist.mín. comprobando si por w hay un atajo
  fmq;
  devuelve D
fin
{Post: D=caminosMínimos(g,v)}
```

Caminos mínimos en grafos

- Corrección:

  ver material adicional en la web.

- Se presenta a continuación una implementación más detallada
  - Se utiliza en lugar de S su complementario T
  - Se supone que n es el número de vértices

Caminos mínimos en grafos

```
{Pre: g es un grafo dirigido etiquetado no neg.}
función Dijkstra(g:grafo; v:vért) devuelve vector[vért] de etiq
variables T:cjtVért;
          D:vector[vért] de etiq;
          u,w:vért; val:etiq
principio
  T:=∅;
  para todo w en vért hacer
    D[w]:=etiqueta(g,v,w); T:=T∪{w}
  fpara;
  D[v]:=0; T:=T-{v};
  repetir n-2 veces {quedan n-1 caminos por determinar}
    {selección del w tal que D[w] es mín.: w∈T ∧ ∀u∈ T:D[w]≤D[u]}
    val:=∞;
    para todo u en T hacer
      si D[u]≤val ent w:=u; val:=D[u] fsi
      {siempre hay un nodo que cumple la condición}
    fpara;
    ...
```

Caminos mínimos en grafos

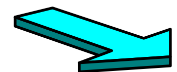
```
    ...
    {se marca w como vértice tratado}
    T:=T-{w};
    {se recalculan las nuevas dist. mínimas}
    para todo u en T hacer
      si D[w]+etiqueta(g,w,u)<D[u]
        ent D[u]:=D[w]+etiqueta(g,w,u)
      fsi
    fpara
  frepetir;
  devuelve D
fin
{Post: D=caminosMínimos(g,v)
```

Nota: el bucle principal se ejecuta *n*-2 veces porque el último camino queda calculado después del último paso (no quedan vértices para hallar atajos)

Caminos mínimos en grafos

- Tiempo de ejecución:
  - se supone que las operaciones sobre cjtos. están implementadas en tiempo constante, excepto la creación (p.ej., mediante un vector de booleanos)
  - fase de inicialización:
    - creación del cjto. y ejecución n veces de diversas operaciones constantes: $\Theta(n)$
  - fase de selección:
    - las instrucciones del interior del bucle son $\Theta(1)$
    - nº de ejecuciones del bucle:
      1ª vuelta: se consultan n-1 vértices,
      2ª vuelta: n-2, etc.
      (el cardinal de T decrece en 1 en cada paso)

      nº de ejecuciones: $n(n-1)/2-1 \Rightarrow \Theta(n^2)$
  - fase de "marcaje":
    - n supresiones a lo largo del algoritmo: $\Theta(n)$
  - fase de recálculo de las distancias mínimas:
    - queda $\Theta(n^2)$ por igual razón que la selección



Coste total: $\Theta(n^2)$

Caminos mínimos en grafos

- Mejoras en el tiempo de ejecución
  - Si la representación es por listas de adyacencia, la fase de recálculo se ejecuta sólo a ($a<n^2$) veces (sustituyendo el bucle sobre los vért. de T por otro bucle sobre los vért. sucesores de w).
  - Si el conjunto T se sustituye con una cola con prioridades, se rebaja también el coste de la fase de selección (puesto que se selecciona el mínimo).

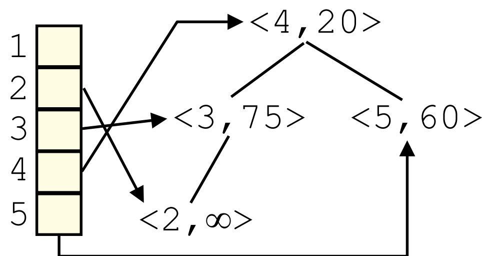
    Problema: la fase de recálculo puede exigir cambiar la prioridad de un elemento cualquiera de la cola.

    Solución: nuevo tipo de cola con prioridades que permita el acceso directo a cualquier elemento.

Caminos mínimos en grafos

```
género cpa {colaPriAristas}
operaciones
  creaVacía: → cpa                          Θ(1)
  inserta: cpa vért etiq → cpa              Θ(log n)
  primero: cpa → (vért,etiq)                Θ(1)
  borra: cpa → cpa                          Θ(log n)
  sustit: cpa vért etiq → cpa               Θ(log n)
  valor: cpa vért → etiq                    Θ(1)
  está?: cpa vért → bool                    Θ(1)
  vacía?: cpa → bool                        Θ(1)
```

– Implementación: montículo junto con un vector indexado por vértices



el valor ∞ de la etiqueta significa que el vértice no está en la cola

Caminos mínimos en grafos

```
{Pre: g es un grafo dirigido etiquetado no neg.}
función Dijkstra(g:grafo; v:vért) devuelve vector[vért] de etiq
variables A:cpa; {cola de aristas con prior.}
          D:vector[vért] de etiq; u,w:vért; et,val:etiq
principio
  {crear la cola con todas las aristas <v,w>}
  creaVacía(A);
  para todo w en vért hacer
    inserta(A,w,etiqueta(g,v,w))
  fpara;
  mq not esVacía(A) hacer
    <w,val>:=primero(A); {selecc. arista mín.}
    D[w]:=val; borra(A); {marca w como tratado}
    {se recalculan las nuevas dist. mínimas}
    para todo <u,et> en suc(g,w) hacer
      si está(A,u) and_then val+et<valor(A,u) entonces
        sustituye(A,u,val+et) fsi
    fpara
  fmq;
  D[v]:=0; devuelve D
fin {Post: D=caminosMínimos(g,v)}
```

Caminos mínimos en grafos

- Eficiencia temporal:
  - inicialización: $\Theta(n\log n)$
  - selección y supresión: $\Theta(n\log n)$
  - bucle interno: examina todas las aristas del grafo y en el caso peor efectúa una sustitución por arista, por tanto: $\Theta(a\log n)$
- El coste total es: $\Theta((a+n)\log n)$, luego es mejor que la versión anterior si el grafo es disperso.
- Si el grafo es denso, el algoritmo original es mejor.

- Puede conseguirse un coste en $\Theta(n^2)$ para grafos densos y en $\Theta(a\log n)$ para no densos, representando el montículo en un árbol k-ario en lugar de binario [BB97, pp. 243].

Caminos mínimos en grafos

- Ejercicio: cálculo de la secuencia de nodos que componen el camino mínimo
  - si el camino mínimo entre *v* y *w* pasa por *u*, el camino mínimo entre *v* y *u* es un prefijo del camino mínimo entre *v* y *w*
  - basta con devolver un vector *pred* tal que *pred*[*w*] contenga el nodo predecesor de *w* en el camino mínimo de *v* a *w* (que será *v* si está directamente unido al nodo de partida o si no hay camino entre *v* y *w*)
  - el vector debe actualizarse al encontrarse un atajo en el camino
  - es necesario también diseñar un nuevo algoritmo para recuperar el camino a un nodo dado, que tendría como parámetro *pred*

Árboles de recubrimiento de coste mínimo

- Objetivo: dado un grafo, obtener un nuevo grafo que sólo contenga las aristas imprescindibles para una optimización global de las conexiones entre todos los nodos

  “optimización global”: algunos pares de nodos pueden no quedar conectados entre ellos con el mínimo coste posible en el grafo original

- Aplicación: problemas que tienen que ver con distribuciones geográficas

  conjunto de computadores distribuidos geográficamente en diversas ciudades de diferentes países a los que se quiere conectar para intercambiar datos, compartir recursos, etc.;

  se pide a las compañías telefónicas respectivas los precios de alquiler de líneas entre ciudades

  asegurar que todos los computadores pueden comunicar entre sí, minimizando el precio total de la red

Árboles de recubrimiento de coste mínimo

- Terminología:
  - árbol libre:
    es un grafo no dirigido conexo acíclico
    - todo árbol libre con n vértices tiene $n-1$ aristas
    - si se añade una arista se introduce un ciclo
    - si se borra una arista quedan vértices no conectados
    - cualquier par de vértices está unido por un único camino simple
    - un árbol libre con un vértice distinguido es un árbol con raíz
  - árbol de recubrimiento de un grafo no dirigido y etiquetado no negativamente: es cualquier subgrafo que contenga todos los vértices y que sea un árbol libre
  - árbol de recubrimiento de coste mínimo:
    es un árbol de recubrimiento y no hay ningún otro árbol de recubrimiento cuya suma de aristas sea menor
  - notación: arm(g)={árboles de recubrimiento de coste mínimo de g}

Árboles de recubrimiento de coste mínimo

- Propiedad fundamental de los árboles de recubrimiento de coste mínimo:

  "Sea g un grafo no dirigido conexo y etiquetado no negativamente, $g \in \{f: V \times V \rightarrow E\}$,

  sea U un conjunto de vértices, $U \subset V$, $U \neq \emptyset$,

  si $<u,v>$ es la arista más pequeña de g tal que $u \in U$ y $v \in V-U$,

  entonces existe algún árbol de recubrimiento de coste mínimo de g que la contiene."

Dem: Por reducción al absurdo (por ejemplo, en [AHU88] A.V. Aho, J.E. Hopcroft y J.D Ullman. *Estructuras de datos y algoritmos*. Addison-Wesley Iberoamericana, S.A., 1988.).

Árboles de recubrimiento de coste mínimo

V. Jarník: "O jistém problému minimálním", *Práca Moravské Prírodovedecké Spolecnosti*, 6, pp. 57-63, 1930.

R.C. Prim:
"Shortest connection networks and some generalizations", *Bell System Technical Journal*, 36, pp. 1389-1401, 1957.

- Algoritmo de Prim (debido a Jarník)
  - Aplica reiteradamente la propiedad de los árboles de recubrimiento de coste mínimo incorporando a cada paso una arista
  - Se usa un conjunto *U* de vértices tratados y se selecciona en cada paso la arista mínima que une un vértice de *U* con otro de su complementario

Árboles de recubrimiento de coste mínimo

```
{Pre: g es un grafo no dirigido conexo etiquetado no negativamente}
función Prim(g:grafo) devuelve grafo
variables U:cjtVért; gsol:grafo;
          u,v:vért; x:etiq
principio
  creaVacío(gsol); U:={cualquier vértice};
  mq U no contenga todos los vért. hacer
    seleccionar <u,v,x> mínima t.q. u∈U; v∉U
    añade(gsol,u,v,x); U:=U∪{v}
  fmq;
  devuelve gsol
fin
{Post: gsol∈arm(g)}
```

Coste: $\Theta(na)$
(es decir, $\Theta(n^3)$ si el grafo es denso)

Árboles de recubrimiento de coste mínimo

- La versión previa puede refinarse hasta obtener un algoritmo en $\Theta(n^2)$, es decir, mejor que el anterior ($a \geq n$-1).

  Se usa un vector *arisMín*, indexado por vértices, que contiene:

  si $v \notin U$: $arisMín[v]=\langle w,g(v,w)\rangle$ t.q. $\langle v,w\rangle$ es la arista más pequeña que conecta v con un vértice $w \in U$

  si $v \in U$: $arisMín[v]=\langle v,\infty\rangle$

Árboles de recubrimiento de coste mínimo

```
{Pre: g es un grafo no dirigido conexo etiquetado no negativamente}
función Prim(g:grafo) devuelve grafo
variables
  arisMín:vector[vért] de <vért,etiq>;
  gsol:grafo; prim,mín,v,w:vért; x:etiq
principio
  {inicialización del resultado}
  prim:=unVérticeCualquiera;
  para todo v en vért hacer
    arisMín[v]:=<prim,etiqueta(g,prim,v)>
  fpara;
  arisMín[prim]:=<prim,∞>;
  {a continuación se aplica el método}
  creaVacío(gsol);
  repetir n-1 veces
    mín:=prim; {centinela: arisMín[mín].et=∞}
    para todo v en vért hacer
      <w,x>:=arisMín[v];
      si x<arisMín[mín].et ent mín:=v {como mínimo habrá uno} fsi
    fpara;
    ...
```

Árboles de recubrimiento de coste mínimo

```
    ...
    {a continuación se añade a la solución}
    añade(gsol,mín,arisMín[mín].v,arisMín[mín].et);
    {se añade mín al conjunto de vértices tratados}
    arisMín[mín]:=<mín,∞>;
    {se reorganiza el vector comprobando si la arista mínima de los
     vértices todavía no tratados los conecta a mín}
    para todo <v,x> en adyacentes(g,mín) hacer
      si (arisMín[v].v≠v)and(x<arisMín[v].et)
        entonces arisMín[v]:=<mín,x>
      fsi
    fpara
  frepetir;
  devuelve gsol
fin
{Post: gsol∈arm(g)}
```

Árboles de recubrimiento de coste mínimo

- Eficiencia temporal:
  - inicialización: lineal en caso de matriz de adyacencia y cuadrática en caso de listas
  - bucle principal:
    - el bucle de selección: $\Theta(n)$
    - el añadido de una arista al grafo: constante usando matriz, lineal usando listas
    - el bucle de reorganización:
      - con matriz de adyacencia: el cálculo de los adyacentes es $\Theta(n)$ y el coste total queda $\Theta(n^2)$
      - con listas: el coste total es $\Theta(a+n)$

Coste total: $\Theta(n^2)$, independientemente de la representación.
Coste espacial: $\Theta(n)$ de espacio adicional.

Árboles de recubrimiento de coste mínimo

J.B. Kruskal: "On the shortest spanning subtree of a graph and the traveling salesman problem", Proceedings of the American Mathematical Society, 7, pp. 48-50, 1956.

- Algoritmo de Kruskal:
  - Se basa también en la propiedad de los árboles de recubrimiento de coste mínimo:

    Partiendo del árbol vacío, se selecciona en cada paso la arista de menor etiqueta que no provoque ciclo sin requerir ninguna otra condición sobre sus extremos.

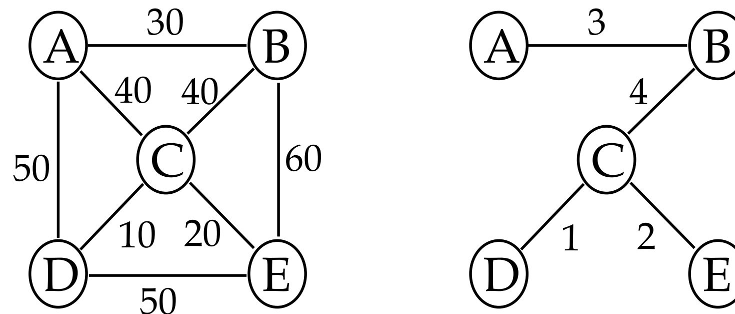
Árboles de recubrimiento de coste mínimo

```
{Pre: g es un grafo no dirigido conexo etiquetado no negativamente}
función Kruskal(g:grafo) devuelve grafo
variables gsol:grafo; u,v:vért; x:etiq
principio
  creaVacío(gsol);
  mq gsol no contenga todos los vértices hacer
    seleccionar <u,v,x> mínima(*) no examinada;
    si no provoca ciclo
      entonces añade(gsol,u,v,x)
    fsi
  fmq;
  devuelve gsol
fin
{Post: gsol∈arm(g)}
```

(∗) Utilizar una cola con prioridades.

Árboles de recubrimiento de coste mínimo

- Implementación eficiente:
  - En cada momento, los vértices que están dentro de una componente conexa en la solución forman una clase de equivalencia, y el algoritmo se puede considerar como la fusión continuada de clases hasta obtener una única componente con todos los vértices del grafo.



Evolución de las clases de equivalencia:

{[A],[B],[C],[D],[E]} → {[A],[B],[C,D],[E]} → {[A],[B],[C,D,E]} →

→ {[A,B],[C,D,E]} → {[A,B,C,D,E]}

Árboles de recubrimiento de coste mínimo

- Se utiliza el TAD "relación de equivalencia" sobre los vértices

```
género rev {relac. de equiv. sobre vért.}
operaciones
  creaREV: → rev {cada vért. una clase}
  clase: rev vért → nat
  fusiona: rev nat nat → rev
  numClases: rev → nat
```

Implementación asintóticamente óptima:

- basada en una representación del tipo `rev` usando árboles y una técnica de compresión de caminos [AHU88, pp. 182-191]
  (o ver aquí: http://webdiis.unizar.es/asignaturas/TAP/material/3.5.disjuntos.pdf)
- el coste de `creaREV` es lineal
- el coste de `numClases` es constante
- el coste de k ejecuciones combinadas de `fusiona` y `clase` es $\Theta(k\alpha(k,n))$, lo cual es prácticamente constante, porque α es una función inversa de la función de Ackerman que crece MUY despacio ($\alpha(k,n) \leq 4$, para todos los valores de k y n "imaginables")

Árboles de recubrimiento de coste mínimo

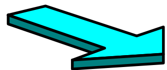
```
{Pre: g es un grafo no dirigido conexo etiquetado no negativamente}
función Kruskal(g:grafo) devuelve grafo
variables T:cpa; gsol:grafo;
          u,v:vért; x:etiq;
          C:rev; ucomp,vcomp:nat
principio
  creaREV(C); {cada vért. forma una clase}
  creaVacío(gsol);
  creaVacía(T);
  {se colocan todas las aristas en la cola}
  para todo v en vért hacer
    para todo <u,x> en adyacentes(g,v) hacer
      inserta(T,v,u,x)
    fpara
  fpara;
  ...
```

Árboles de recubrimiento de coste mínimo

```
  ...
  mq numClases(C)>1 hacer
    {obtener y elim. la arista mín.de la cola}
    <u,v,x>:=primero(T); borra(T);
    {si la arista no provoca ciclo se añade a la
     solución y se fusionan las clases corresp.}
    ucomp:=clase(C,u); vcomp:=clase(C,v);
    si ucomp≠vcomp entonces
      fusiona(C,ucomp,vcomp);
      añade(gsol,u,v,x)
    fsi
  fmq;
  devuelve gsol
fin
{Post: gsol∈arm(g)}
```

Árboles de recubrimiento de coste mínimo

- Coste del algoritmo:
  - las a inserciones consecutivas de aristas en la cola con prioridades dan $\Theta(a\log a)$; como $a \leq n^2$: $\Theta(a\log a) \subseteq \Theta(a\log n^2) = \Theta(2a\log n) = \Theta(a\log n)$
  - como mucho hay a consultas y supresiones de la arista mínima, el coste de la consulta es constante y el de la supresión es $\Theta(\log a)$; por ello, este paso queda en $\Theta(a\log n)$
  - averiguar cuántas clases hay en la relación de equivalencia es constante
  - en el caso peor, la operación de fusión de clases se ejecuta n-1 veces y la operación de localizar la clase $2a$ veces; por tanto, el coste total es en la práctica $\Theta(a)$
  - las n-1 inserciones de aristas quedan en $\Theta(n)$ con matriz de adyacencia y $\Theta(n^2)$ con listas, aunque en el caso de las listas puede reducirse también a $\Theta(n)$ si se elimina en la operación de añade la comprobación de existencia de la arista (el algoritmo de Kruskal garantiza que no habrá inserciones repetidas de aristas)



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

(menos que el algoritmo de Prim, aunque con mayor espacio adicional)

Consideraciones sobre la corrección del esquema voraz

- La selección de una candidata óptima en cada paso es una estrategia heurística que no siempre conduce a la solución óptima.
- En ocasiones, se utilizan heurísticas voraces para obtener soluciones subóptimas cuando el cálculo de las óptimas es demasiado costoso.
- *¿Puede saberse que familias de problemas (de optimización u otros) pueden resolverse con una estrategia voraz?*

  La respuesta es: NO exactamente (aunque hay resultados).
- Sin embargo, existen ciertos indicios… más bien, técnicas de demostración:

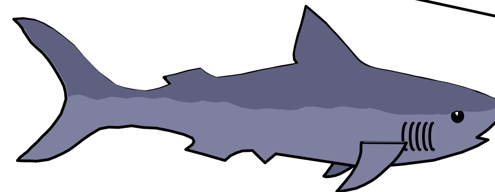
  la *propiedad de la selección voraz* y la existencia de una *subestructura óptima*.

Consideraciones sobre la corrección del esquema voraz

- La propiedad de la selección voraz:

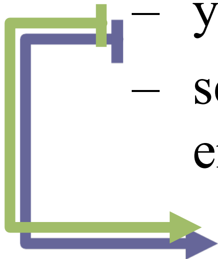
  – Se verifica esta propiedad cuando una solución globalmente óptima puede ser alcanzada mediante selecciones localmente óptimas que son tomadas en cada paso sin depender de las selecciones futuras;

  – en otras palabras, una estrategia voraz progresa de arriba hacia abajo, tomando una decisión voraz tras otra, reduciendo iterativamente el problema a otro más pequeño.



Consideraciones sobre la corrección del esquema voraz

- ¿Cómo se comprueba si se verifica la propiedad de la selección voraz?
  - Normalmente, se examina una solución globalmente óptima,
  - se trata de demostrar que esa solución puede ser manipulada de forma que se obtiene tras una primera selección voraz (localmente óptima),
  - y esa selección reduce el problema a otro similar pero más pequeño;
  - se aplica inducción para demostrar que se puede usar una selección voraz en cada paso



hay que demostrar que una solución óptima posee una "subestructura óptima"

- Subestructura óptima:
  - Un problema posee una subestructura óptima si una solución óptima de ese problema incluye soluciones óptimas de subproblemas.

Códigos de Huffman

D.A. Huffman: “A method for the construction of minimum-redundancy codes”, *Proceedings of the IRE*, 40(9), pp. 1098-1101, 1952.

- Los códigos de Huffman son una técnica muy útil para comprimir ficheros.
- El algoritmo voraz de Huffman utiliza una tabla de frecuencias de aparición de cada carácter para construir una forma óptima de representar los caracteres con códigos binarios.
- Ejemplo:
  – Se tiene un fichero con 100.000 caracteres que se desea comprimir. Las frecuencias de aparición de caracteres en el fichero son las siguientes:

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

  – Puede usarse un código de longitud fija (de 3 bits). El fichero requeriría 300.000 bits.

	a	b	c	d	e	f
cód.long.fija	000	001	010	011	100	101

Códigos de Huffman

– Puede hacerse mejor con un **código de longitud variable**, dando codificaciones cortas a los caracteres más frecuentes y más largas a los menos frecuentes.

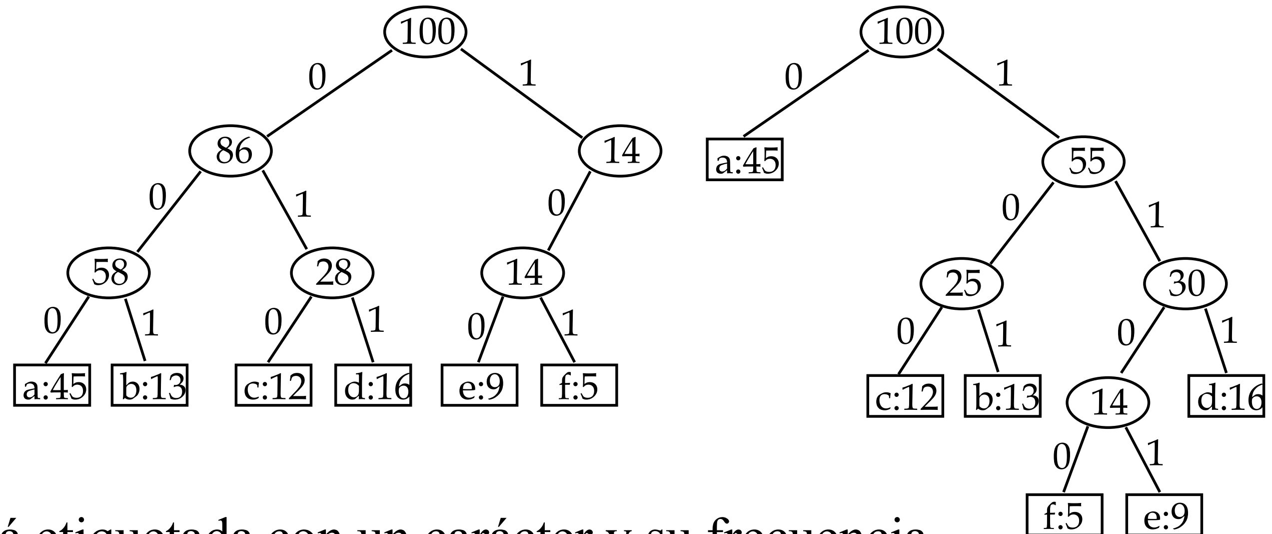
	a	b	c	d	e	f
cód.long.var.	0	101	100	111	1101	1100

Este código ahorra algo más del 25% (requiere 224.000 bits en lugar de 300.000).

- Se precisa un **código libre de prefijos**:
  – Ninguna codificación puede ser prefijo de otra.
  – De esta forma, la decodificación es inmediata pues no hay ambigüedades.
  – Por ejemplo: ‘001011101’ sólo puede ser ‘aabe’.
- El código se representa mediante un **trie** (árbol lexicográfico):
  – árbol binario cuyas hojas son los caracteres codificados;
  – el código de cada carácter es el camino desde la raíz hasta la hoja, donde ir al hijo izquierdo significa ‘0’ e ir hacia el derecho significa ‘1’.

Códigos de Huffman

- Ejemplo: árboles de los dos códigos anteriores.



  – Cada hoja está etiquetada con un carácter y su frecuencia.
  – Cada nodo interno está etiquetado con la suma de los pesos de las hojas de su subárbol.
  – **Un código óptimo siempre está representado por un árbol lleno:** cada nodo interno tiene dos hijos.
    - Si el alfabeto que se quiere codificar es C, entonces el árbol del código óptimo tiene $|C|$ hojas y $|C| - 1$ nodos internos.

Códigos de Huffman

- El algoritmo voraz de Huffman construye el árbol A de un código óptimo de abajo hacia arriba.
- Utiliza una cola Q de árboles con prioridades (las frecuencias hacen de prioridades).
- Empieza con un conjunto de $|C|$ hojas en Q y realiza una secuencia de $|C| - 1$ "mezclas" hasta crear el árbol final.
- En cada paso, se "mezclan" los dos objetos (árboles) de Q que tienen menos frecuencia y el resultado es un nuevo objeto (árbol) cuya frecuencia es la suma de las frecuencias de los dos objetos mezclados.

Códigos de Huffman

```
{Pre: C es un conjunto de caracteres y f es el
      vector de frecuencias de esos caracteres}
función Huffman(C:conjunto;f:vectFrec) devuelve árbol
variables Q:colaPri; i,fx,fy,fz:entero; z,x,y:árbol
principio
  creaVacía(Q);
  para todo x en C hacer
    inserta(Q,<x,f[x]>)
  fpara;
  para i:=1 hasta |C|-1 hacer
    <x,fx>:=primero(Q); borra(Q);
    <y,fy>:=primero(Q); borra(Q);
    fz:=fx+fy;
    z:=creaÁrbol(raíz=>fz, hijoIzq=>x, hijoDer=>y);
    inserta(Q,<z,fz>)
  fpara;
  <z,fz>:=primero(Q); borra(Q)
  devuelve z
fin
{Post: z es el árbol de un código libre de prefijos óptimo para
(C,f)}
```

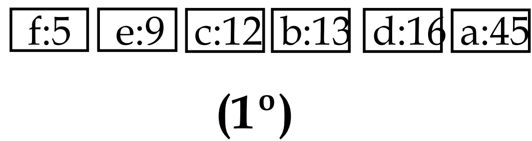
El algoritmo típico de construcción de un montículo (o cola con prioridades) con n datos, a partir de un vector que contiene esos datos en cualquier orden, tiene la siguiente forma:

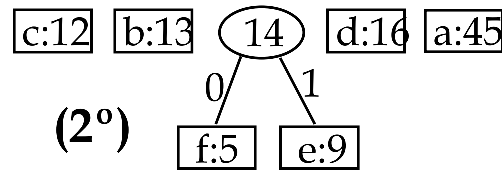
```
para i:=n div 2 descendiendoHasta 1 hacer
   empujar(v,i,n)
fpara
```

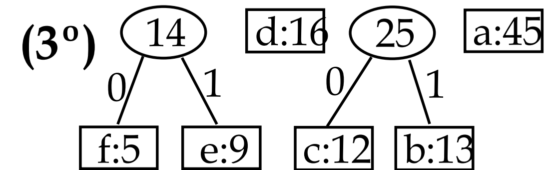
Donde el subalgoritmo *empujar* (*heapify*, en la literatura anglosajona), es el que aparece en la página 171-172 de los apuntes de EDA.

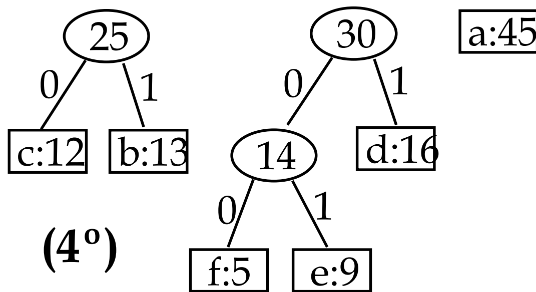
Códigos de Huffman

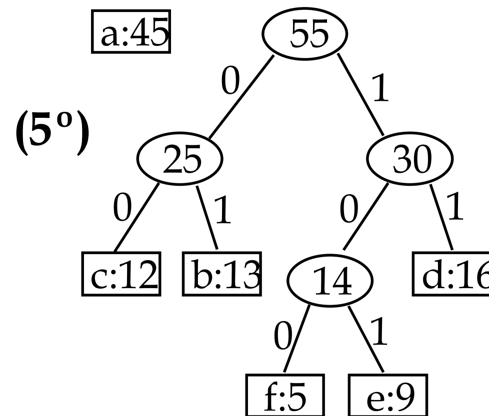
- Para el ejemplo anterior:

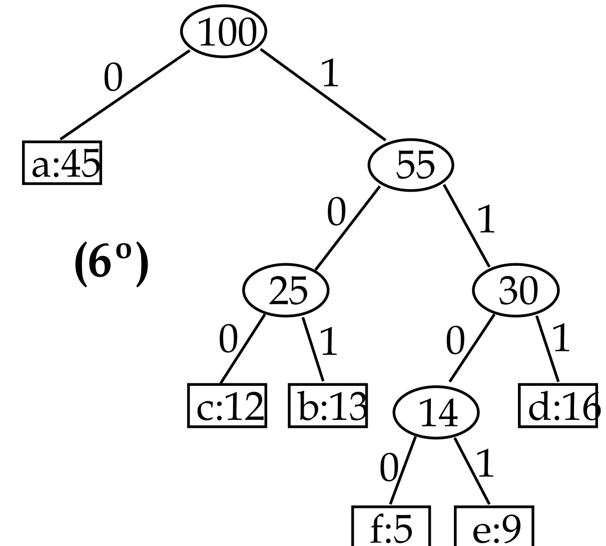












Códigos de Huffman

- Coste temporal:
  - inicialización de Q: $\Theta(|C|)$ (ejercicio)
  - el interior del bucle: $\Theta(\log |C|)$

  Coste total: $\Theta(|C|\log |C|)$
- Corrección: ver que se verifican las propiedades de la selección voraz y de la subestructura óptima.
- Propiedad de la selección voraz:
  - Sea C un alfabeto en el que cada carácter c tiene frecuencia $f[c]$. Sean x e y dos caracteres de C con frecuencias mínimas. Entonces existe un código libre de prefijos óptimo para C en el que las codificaciones de x e y tienen igual longitud y se diferencian sólo en el último bit.

    La idea de la demostración consiste en tomar un árbol que represente un código libre de prefijos óptimo arbitrario y modificarlo hasta convertirlo en otro árbol que represente otro código libre de prefijos óptimo y tal que los caracteres x e y aparezcan como hojas hermanas de máxima profundidad.

    Detalles en [CLRS09, pp. 433-434].

Códigos de Huffman

  - Por tanto, el proceso de construir un árbol óptimo mediante mezclas puede empezar, sin pérdida de generalidad, con la elección voraz de mezclar los dos caracteres de menor frecuencia.
- Propiedad de la subestructura óptima:
  - Sea T un árbol binario lleno que representa un código libre de prefijos sobre un alfabeto C, con frecuencia $f[c]$ para cada carácter $c \in C$.
    Sean x e y dos caracteres de mínima frecuencia que aparecen como hojas hermanas en T, y sea z su padre.
    Considerando z como un carácter con frecuencia $f[z]=f[x]+f[y]$, el árbol $T'=T-\{x,y\}$ representa un código libre de prefijos para el alfabeto $C'=C-\{x,y\}\cup\{z\}$.
    Si T' es óptimo para C', entonces T es óptimo para C.
    La demostración, en [CLRS09, p. 435].
- La corrección del algoritmo se sigue de los dos resultados anteriores, aplicando inducción.

El problema de la selección de actividades

- Es un problema de **planificación de tareas** (en inglés, *scheduling*).
- Se tiene un conjunto $S=\{1,2,\dots,n\}$ de actividades (por ej., clases) que deben usar un recurso (por ej., un aula) que sólo puede ser usado por una actividad en cada instante.
- Cada actividad i tiene asociado un instante de comienzo c_i y un instante de finalización f_i, tales que $c_i \leq f_i$, de forma que la actividad i, si se realiza, debe hacerse durante $[c_i,f_i)$.
- Dos actividades i, j se dicen compatibles si los intervalos $[c_i,f_i)$ y $[c_j,f_j)$ no se superponen (i.e., $(c_i \geq f_j) \vee (c_j \geq f_i)$).
- El problema de selección de actividades consiste en seleccionar un conjunto de actividades mutuamente compatibles que tenga cardinal máximo.

El problema de la selección de actividades

- Ejemplos de estrategias voraces (funciones de selección voraz) para resolver el problema:
  - Seleccionar primero la tarea que empieza primero.
  - Seleccionar primero la tarea que termina primero.
  - Seleccionar primero la tarea más corta.
  - Seleccionar primero la tarea menos conflictiva.

El problema de la selección de actividades

```
constante n=... {número de actividades}
tipo vectReal=vector[1..n] de real
```

```
{Pre: ∀i∈1..n:c[i]≤f[i] ∧ ∀i∈1..n-1:f[i]≤f[i+1]}
función selec(c,f:vectReal)devuelve conjunto
variables i,j:entero; A:conjunto
principio
  A:={1};
  j:=1; {j es la última actividad seleccionada}
  para i:=2 hasta n hacer
    si c[i]≥f[j]
      entonces A:=A∪{i}; j:=i
    fsi
  fpara;
  devuelve A
fin
{Post: A es soluc. óptima del problema de la selección de
actividades.}
```

El problema de la selección de actividades

- Coste temporal: $\Theta(n)$

  $\Theta(n\log n)$ si se tiene en cuenta que hay que ordenar primero los vectores.

- Demostración de la corrección:

  La actividad 1 tiene el primer instante de finalización. Veamos que existe una solución óptima que comienza con una selección voraz, es decir, con la actividad 1.

  Supongamos que $A \subseteq S$ es una solución óptima y que las actividades de A están ordenadas por tiempos de finalización. Supongamos que la primera actividad de A es la actividad k.

  Si $k=1$, entonces A empieza con una selección voraz.

  Si $k \neq 1$, veamos que existe otra solución óptima B que comienza con 1.

  Sea $B=A-\{k\}\cup\{1\}$. Como $f_1 \leq f_k$, las actividades en B son disjuntas y como tiene el mismo número de actividades que A entonces B es también óptima.

El problema de la selección de actividades

Después de la selección voraz de la actividad 1, el problema se reduce a encontrar una solución óptima del problema de selección actividades S' sobre las actividades de S que son compatibles con la actividad 1.

Si A' es una solución óptima para el problema $S' = \{i \in S \mid c_i \geq f_1\}$,

entonces $A = A' \cup \{1\}$ es una solución óptima para el problema original S.

Trivial, si existiese alguna solución mejor que A para S, las actividades de esa solución eliminando la tarea 1 serían una solución mejor para S' que A'.

Por tanto, después de cada selección voraz, nos encontramos con otro problema de optimización con igual forma que el problema original.

Por inducción sobre el número de selecciones, y realizando una selección voraz en cada paso, se obtiene una solución óptima. ■

El problema de la minimización del tiempo de espera

- Es otro problema de planificación de tareas.
- Un servidor (por ej., un procesador, un cajero automático, un surtidor de gasolina, etc.) tiene que atender n clientes que llegan todos juntos al sistema.
- El tiempo de servicio para cada cliente es t_i, i=1,2,…,n.
- Se quiere minimizar:

$$T = \sum_{i=1}^{n} \text{tiempo total que el cliente } i \text{ está en el sistema}$$

El problema de la minimización del tiempo de espera

- Ejemplo:

3 clientes con t_1=5, t_2=10, t_3=3.

orden	T			
1 2 3:	5 +	(5+10) +	(5+10+3)	= 38
1 3 2:	5 +	(5+3) +	(5+3+10)	= 31
2 1 3:	10 +	(10+5) +	(10+5+3)	= 43
2 3 1:	10 +	(10+3) +	(10+3+5)	= 41
3 1 2:	3 +	(3+5) +	(3+5+10)	= 29 ← óptimo
3 2 1:	3 +	(3+10) +	(3+10+5)	= 34

El problema de la minimización del tiempo de espera

- Estrategia voraz: Atender en cada paso al cliente no atendido con menor tiempo de servicio.
- Demostración de la corrección:

Sea $I=(i_1,i_2,\ldots,i_n)$ una permutación cualquiera de los naturales $\{1,2,\ldots,n\}$.

Si se atiende a los clientes según la secuencia I, el tiempo total que los clientes están en el sistema es:

$$
\begin{aligned}
T(I) &= t_{i1} + (t_{i1}+t_{i2}) + ((t_{i1}+t_{i2})+t_{i3}) + \cdots \\
&= n t_{i1} + (n\text{-}1)t_{i2} + (n\text{-}2)t_{i3} + \cdots \\
&= \sum_{k=1}^{n} (n\text{-}k+1)t_{ik}
\end{aligned}
$$

Supongamos que I es tal que es posible encontrar dos naturales a y b con $a<b$ y $t_{ia}>t_{ib}$, es decir, que el cliente a-ésimo es atendido antes que el b-ésimo aunque su tiempo de servicio es mayor.

Si se invierten sus posiciones en I se obtiene una nueva secuencia I'.

El problema de la minimización del tiempo de espera

Entonces:

$$T(I') = (n\text{-}a+1)t_{ib} + (n\text{-}b+1)t_{ia} + \sum_{\substack{k=1 \\ k\neq a,b}}^{n} (n\text{-}k+1)t_{ik}$$

$$\begin{aligned} T(I) - T(I') &= (n\text{-}a+1)(t_{ia}\text{-}t_{ib}) + (n\text{-}b+1)(t_{ib}\text{-}t_{ia}) \\ &= (b\text{-}a)(t_{ia}\text{-}t_{ib}) \\ &> 0 \end{aligned}$$

Es decir, se puede mejorar cualquier secuencia en la que un cliente que necesita más tiempo sea atendido antes que otro que necesita menos.

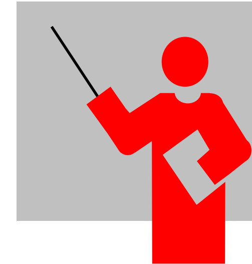
Las únicas secuencias que no pueden mejorarse, y por tanto son óptimas, son aquéllas en las que los clientes están ordenados por tiempos de servicio decrecientes (todas ellas dan igual tiempo total). ■

- Puede generalizarse a *s* procesadores.
  - Se numeran los clientes de forma que $t_1 \leq t_2 \leq \cdots \leq t_n$.
  - El procesador *i*, $1 \leq i \leq s$, atenderá a los clientes *i*, *i*+*s*, *i*+2*s*, …

Fundamentos teóricos del esquema voraz

- Existe una teoría matemática que sirve de base a los algoritmos voraces:

  Teoría de matroides



- Un matroide es un par $M=(S,I)$ tal que:
  - S es un conjunto finito no vacío.
  - I es una familia no vacía de subconjuntos de S, llamados subconjuntos **independientes**, tal que si $B \in I$ y $A \subseteq B$, entonces $A \in I$.
    Por cumplir esta propiedad, I se dice **hereditaria**
    (nótese que el conjunto vacío es necesariamente un miembro de I).
  - Si $A \in I$, $B \in I$ y $|A|<|B|$, entonces existe algún $x \in B-A$ tal que $A \cup \{x\} \in I$.
    Se dice que M satisface la **propiedad de intercambio**.

Fundamentos teóricos del esquema voraz

- Ejemplos de matroides:
  - Dada una matriz de números reales, sea S el conjunto de sus vectores fila, y considerese que un subconjunto de S es independiente si los vectores que lo componen son linealmente independientes (en el sentido habitual).

    Se llama "matroide matricial" y es el primero que se estudió (H. Whitney, 1935).

  - Sea un grafo no dirigido $G=(V,A)$.
    Sea $M_G=(S_G,I_G)$, donde:
    - $S_G=A$ (el conjunto de aristas).
    - Si $B\subseteq A$, entonces $B\in I_G$ si y sólo si B es acíclico.

    Se llama "matroide gráfico" y está muy relacionado con el cálculo de árboles de recubrimiento de coste mínimo.

Fundamentos teóricos del esquema voraz

- Sea M=(S,I) y $B \in I$.
  Un elemento $x \notin B$ se dice **extensión** de B si $B \cup \{x\} \in I$.
  - Ejemplo: en un matroide gráfico M_G, si B es un conjunto independiente de aristas, entonces la arista a es una extensión de B si y sólo si a no está en B y la incorporación de a a B no crea un ciclo.
- Si A es un subconjunto independiente de un matroide M, se dice que A es **maximal** si no tiene extensiones.
- **Teorema.** Todos los subconjuntos independientes maximales de un matroide tienen el mismo cardinal.
  (Dem.: muy fácil [CLRS09, p. 439].)
  - Ejemplo: Dado un grafo no dirigido y conexo G, todo subconjunto independiente maximal de M_G es un árbol libre con $|V|-1$ aristas que conecta todos los vértices de G.
    Ese árbol se llama **árbol de recubrimiento** de G.

Fundamentos teóricos del esquema voraz

- Un matroide $M=(S,I)$ al que se le asocia una función que asigna un "peso" estrictamente positivo $w(x)$ a cada $x \in S$ se llama matroide **ponderado**.

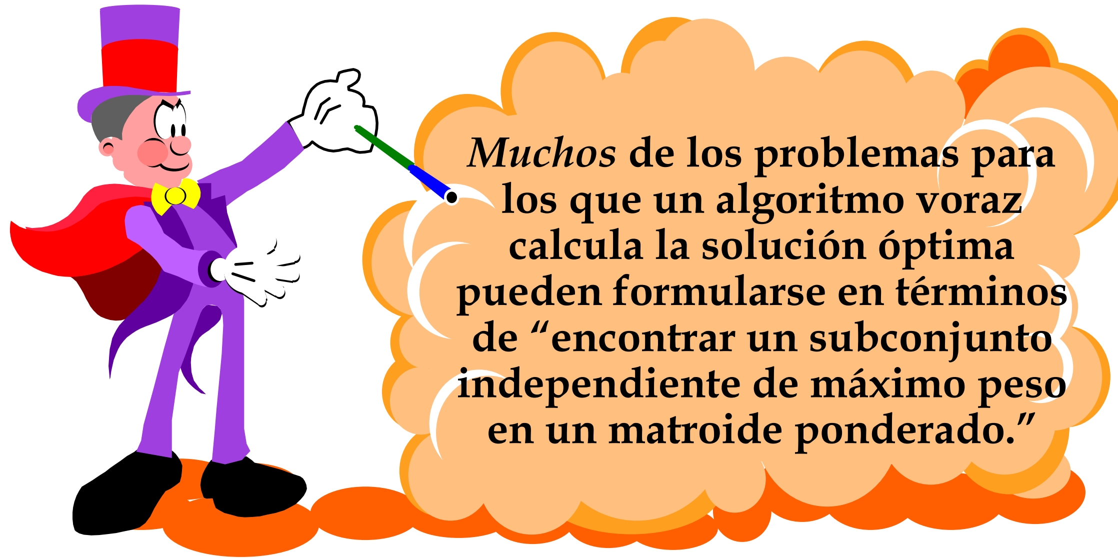
  La función de peso w se extiende a los subconjuntos de S mediante la suma:

$$w(A) = \sum_{x \in A} w(x), \text{ para todo } A \subseteq S.$$

  - Ejemplo: si $w(a)$ denota la longitud de la arista a en un matroide gráfico M_G, entonces $w(B)$ es la longitud total de todas las aristas del subconjunto de aristas B.

Fundamentos teóricos del esquema voraz

- Voracidad y matroides.



  - Es decir, dado un matroide ponderado $M=(S,I)$, se quiere encontrar un subconjunto A independiente (i.e., $A \in I$) tal que $w(A)$ sea máximo.
  - Un subconjunto independiente y de **máximo peso** en un matroide ponderado se llama subconjunto **óptimo** del matroide.
  - Como la función de peso w es positiva, un subconjunto óptimo siempre es maximal (i.e., no tiene extensiones, ya que aumentar un conjunto siempre incrementa su peso).

Fundamentos teóricos del esquema voraz

- Ejemplo: el problema del árbol de recubrimiento de coste mínimo (*).
  - Sea un grafo no dirigido y conexo $G=(V,A)$ y una función de longitud w tal que $w(a)$ es la longitud positiva de la arista a (llamamos longitudes a las etiquetas del grafo en lugar de pesos por reservar esta palabra para la función definida sobre un matroide).
  - El problema(*) consiste en encontrar un subconjunto de aristas que conecte todos los vértices del grafo y tenga longitud mínima.
  - Considerar M_G y la función de peso w' definida como: $w'(a) = w_0 - w(a)$, donde w_0 es un valor mayor que la longitud de todas las aristas.
  - En este matroide ponderado, todos los pesos son positivos y un subconjunto óptimo es precisamente un árbol de recubrimiento del grafo original con longitud total mínima.
  - Más específicamente, cada subconjunto independiente maximal B corresponde a un árbol de recubrimiento, y puesto que

    $$w'(B) = (|V| - 1)w_0 - w(B)$$

    para todo subconjunto independiente maximal B, el subconjunto independiente que maximiza $w'(B)$ debe minimizar $w(B)$.

Fundamentos teóricos del esquema voraz

- Algoritmo voraz de cálculo de un subconjunto óptimo en un matroide arbitrario:

```
{Pre: S y la función indep definen un matroide;
      w es una función de peso positiva sobre S}
función voraz(S:conjunto) devuelve conjunto
variable A:conjunto
principio
  A:=∅;
  ordenar S por valores decrecientes del peso w;
  para todo x en S tomado en orden decreciente de w(x) hacer
    si indep(A∪{x}) entonces
      A:=A∪{x}
    fsi
  fpara;
  devuelve A
fin
{Post: A es subconjunto óptimo de (S,indep,w)}
```

Fundamentos teóricos del esquema voraz

- Eficiencia temporal:

  Denotando $|S|$ como n,

  - fase de ordenación: $\Theta(n\log n)$
  - bucle: $\Theta(n\,f(n))$, suponiendo que el coste de evaluar la función `indep` es $\Theta(f(n))$

  Total: $\Theta(n\log n + n\,f(n))$

- Corrección:

  Teorema 16.11 en [CLRS09, pp. 441-442]

  (un par de páginas; sin gran dificultad).

Fundamentos teóricos del esquema voraz

- **Utilidad** fundamental:

Dados un **problema** y una **solución voraz**,

para demostrar la **corrección** de esa solución basta con

identificar una estructura de **matroide subyacente** al problema

Fundamentos teóricos del esquema voraz

- Resumen:
  - La teoría de matroides no cubre todos los problemas en los que el esquema voraz calcula la solución óptima.
  - Sin embargo cubre muchos casos interesantes (por ej., árboles de recubrimiento de coste mínimo).
  - Más aún, se ha demostrado que una estructura es un matroide si y sólo si el esquema voraz calcula para ella una solución óptima sea cual sea la función de peso escogida

    (la demostración puede verse en pp. 13-18 de

    [Koz92] D.C. Kozen. *The Design and Analysis of Algorithms*. Springer-Verlag, 1992.)
  - La investigación continúa…

Un problema de planificación de tareas a plazo fijo

- Se tiene un conjunto $S=\{1,2,\ldots,n\}$ de tareas de duración unidad.
- En cada instante, sólo se puede realizar una tarea.
- La tarea i, $1\leq i\leq n$, debe terminarse antes del instante (o **plazo**) d_i, con $1\leq d_i\leq n$.
- Hay una **penalización** w_i, $1\leq i\leq n$, si la tarea i no termina antes de su plazo.
- Se trata de encontrar una secuencia de realización de las tareas que minimice la penalización total.

Un problema de planificación de tareas a plazo fijo

- Una tarea en una secuencia se dice **puntual** si termina antes de su plazo, y **tardía** si lo hace después de su plazo.
- Una secuencia está en **forma “puntuales primero”** si todas las tareas puntuales están antes de las tardías.
- Cualquier secuencia puede transformarse en forma puntuales primero (manteniendo la penalización).

    En efecto, si una tarea puntual x va detrás de una tarea tardía y, pueden intercambiarse sus posiciones de forma que x sigue siendo puntual e y tardía.

- Una secuencia está en **forma canónica** si está en forma puntuales primero y las tareas puntuales están ordenadas por valores crecientes de sus plazos.
- Cualquier secuencia puede transformarse en forma canónica (manteniendo la penalización).

    Argumentación similar a la anterior.

Un problema de planificación de tareas a plazo fijo

- Encontrar una secuencia óptima es encontrar un subconjunto A de tareas que sean puntuales en la secuencia óptima.
- Una vez encontrado A, la secuencia se crea ordenando los elementos de A por plazos crecientes y a continuación las tareas tardías (en $S - A$) en cualquier orden.
- Decimos que un subconjunto A de tareas es **independiente** si existe una ordenación de las mismas en la que ninguna es tardía.

    Por ejemplo, el conjunto de tareas puntuales de una secuencia es un conjunto independiente.

- Denotamos con I el conjunto de todos los subconjuntos independientes de tareas.

Un problema de planificación de tareas a plazo fijo

- Sea $N_t(A)$, t=1,…,n, el nº de tareas de A cuyo plazo es menor o igual que t.
- **Lema.** Sea A un conjunto de tareas. Son equivalentes:
  1. A es independiente.
  2. $N_t(A) \leq t$, para t=1,…,n.
  3. Si las tareas de A se ordenan por plazos crecientes, entonces ninguna tarea es tardía.

  Dem.: (1)⇒(2) Si $N_t(A) > t$ para algún t, entonces no hay forma de construir una secuencia con las tareas de A de forma que ninguna sea tardía.

  (2)⇒(3) El i-ésimo plazo mayor es como mucho i. Por tanto, ordenando las tareas por plazos crecientes, no hay forma de que alguna tarea sea tardía.

  (3)⇒(1) Trivial. ■
- **Corolario.** La propiedad (2) del lema anterior sirve para decidir en tiempo $O(|A|)$ si un conjunto de tareas es independiente.

  Dem.: Ejercicio

  (suponer las tareas **sin** ordenar previamente; puede usarse espacio adicional $O(|A|)$).

Un problema de planificación de tareas a plazo fijo

- El problema de minimizar la suma de penalizaciones de las tareas tardías de una secuencia es el mismo que el de maximizar la suma de penalizaciones de sus tareas puntuales.
- **Teorema.** Si S es un conjunto de tareas de duración unidad con penalizaciones e I es el conjunto de todos los conjuntos independientes de tareas, entonces (S,I) es un matroide.

  Dem.: Es claro que todo subconjunto de un conjunto independiente de tareas es también independiente.

  Para demostrar la propiedad del intercambio, supongamos que B y A son conjuntos independientes de tareas con $|B|>|A|$.

  Sea k el mayor t tal que $N_t(B) \le N_t(A)$.

  Como $N_n(B)=|B|$ y $N_n(A)=|A|$, se sigue que $k<n$ y que $N_j(B)>N_j(A)$ para todo j tal que $k+1 \le j \le n$.

  Por tanto, B tiene más tareas con plazo $k+1$ que A.

Un problema de planificación de tareas a plazo fijo

Sea x una tarea de $B \backslash A$ con plazo $k+1$.

Sea $A' = A \cup \{x\}$.

Demostramos ahora que A' es independiente usando la propiedad (2) del Lema anterior.

Como A es independiente, $N_t(A') = N_t(A) \le t$, para $1 \le t \le k$.

Como B es independiente, $N_t(A') \le N_t(B) \le t$, para $k < t \le n$.

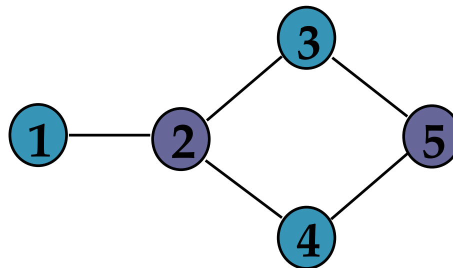
Por tanto, A' es independiente. ■

- **Corolario.** Se puede utilizar un algoritmo voraz para encontrar un conjunto independiente de tareas A con penalización máxima.
- A continuación, se puede crear una secuencia óptima que tiene como tareas puntuales las de A.
- Tiempo de ejecución: $O(n^2)$

  ¿Por qué?

Heurísticas voraces: Coloreado de grafos

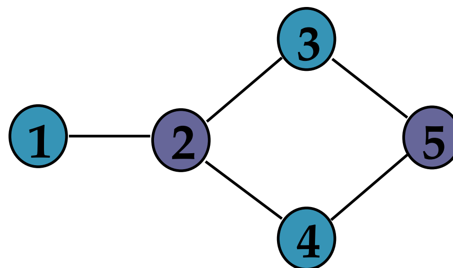
- A veces se utilizan algoritmos voraces a pesar de que no calculan soluciones óptimas:
  - bien porque el cálculo de una solución óptima es demasiado costoso,
  - bien porque el algoritmo voraz calcula un solución "subóptima" que resulta suficiente.
- Problema del coloreado de un grafo.
  - Sea $G=(V,A)$ un grafo no dirigido cuyos vértices se desea colorear.
  - Se exige que todo par de vértices unidos por una arista tengan asignados colores diferentes.
  - Se pretende emplear el menor número posible de colores.



Heurísticas voraces: Coloreado de grafos

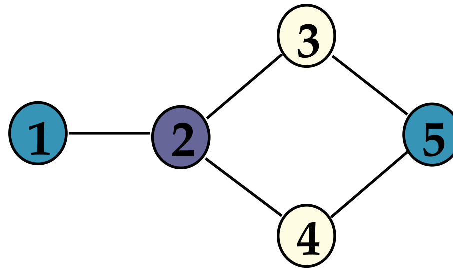
- Algoritmo voraz de coloreado de un grafo:
  - escoger inicialmente un color y un vértice arbitrario como punto de partida;
  - tratar de asignarle ese color al mayor número posible de vértices, respetando la restricción impuesta (vértices adyacentes deben tener distinto color);
  - escoger otro vértice aún no coloreado y un color distinto y repetir el proceso hasta haber coloreado todos los vértices.

- Aplicado al ejemplo anterior, se obtiene la solución óptima: dos colores.



Heurísticas voraces: Coloreado de grafos

– Sin embargo, tomando el mismo grafo pero ordenando los vértices de otra forma, 1,5,2,3,4, el algoritmo voraz colorearía 1 y 5 de un color, 2 en otro, y necesitaría un tercer color para 3 y 4.



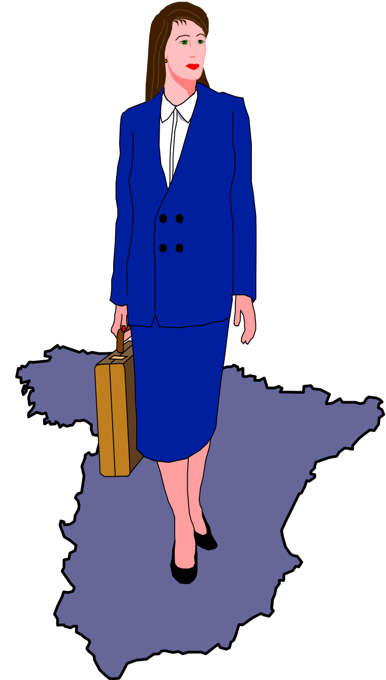
– **Por tanto es un algoritmo heurístico que tiene la posibilidad, pero no la certeza, de encontrar la solución óptima.**

– Todos los algoritmos exactos conocidos para el problema del coloreado de un grafo emplean un tiempo exponencial, de ahí la utilidad de la heurística voraz.

Heurísticas voraces: El problema del viajante de comercio

- El problema del viajante de comercio consiste en encontrar un recorrido de longitud mínima para un viajante que tiene que visitar varias ciudades y volver al punto de partida, conocida la distancia existente entre cada dos ciudades.

- En términos matemáticos es el problema del cálculo del hamiltoniano de longitud mínima.



Heurísticas voraces: El problema del viajante de comercio

- Ejemplo:

  Cinco cabinas de teléfonos, *b*, *c*, *d*, *e*, *f*, para las que se conocen sus coordenadas relativas a la central telefónica, *a*, desde la que parten los recolectores y a donde deben regresar al terminar, y se supone que la distancia entre cada dos viene dada por la línea recta.

c ● (1,7) *d* ● (15,7)

e ● (15,4)

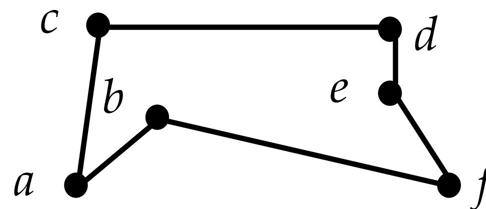
b ● (4,3)

a ● (0,0) *f* ● (18,0)

a						
b	5					
c	7,07	5				
d	16,55	11,70	14			
e	15,52	11,05	14,32	3		
f	18	14,32	18,38	7,6	5	
distan.	*a*	*b*	*c*	*d*	*e*	*f*

Heurísticas voraces: El problema del viajante de comercio

- El **algoritmo de *fuerza bruta* para resolver el problema** consiste en **intentar todas las posibilidades**, es decir, calcular las longitudes de todos los recorridos posibles, y seleccionar la de longitud mínima (veremos una solución algo mejor mediante programación dinámica).
- Obviamente, el coste de tal algoritmo crece exponencialmente con el número de puntos a visitar.
- En el ejemplo anterior, la solución viene dada por el siguiente recorrido (en realidad dos recorridos, pues ambos sentidos de marcha son posibles) de longitud 48,39:



Heurísticas voraces: El problema del viajante de comercio

- Heurística voraz:

  Ir seleccionando parejas de puntos que serán visitados de forma consecutiva:

  - se seleccionará primero aquella pareja de puntos entre los que la distancia sea mínima;
  - a continuación, se selecciona la siguiente pareja separada con una distancia mínima siempre que esta nueva elección no haga que:
    - se visite un punto dos veces o más (es decir, que el punto aparezca tres o más veces en las parejas de puntos seleccionadas), o
    - se cierre un recorrido antes de haber visitado todos los puntos.

  De esta forma, si hay que visitar *n* puntos (incluido el origen), se selecciona un conjunto de *n* parejas de puntos (que serán visitados de forma consecutiva) y la solución consiste en reordenar todas esas parejas de forma que constituyan un recorrido.

Heurísticas voraces: El problema del viajante de comercio

- En el ejemplo anterior:
  - Las parejas ordenadas por distancia entre sus puntos son: (d,e), (a,b), (b,c), (e,f), (a,c), (d,f), (b,e), (b,d), (c,d), (b,f), (c,e), (a,e), (a,d), (a,f) y (c,f).
  - Se selecciona primero la pareja (d,e) pues la distancia entre ambos puntos es mínima (3 unidades).

    (d,e)
  - Después se seleccionan las parejas (a,b), (b,c) y (e,f). La distancia es para todas ellas igual (5 unidades).

    (d,e), (a,b), (b,c), (e,f)
  - La siguiente pareja de distancia mínima es (a,c), con longitud 7,07. Sin embargo, esta pareja cierra un recorrido junto con otras dos ya seleccionadas, (b,c) y (a,b), por lo que se descarta.
  - La siguiente pareja es (d,f) y se descarta también por motivos similares.

Heurísticas voraces: El problema del viajante de comercio

(d,e), (a,b), (b,c), (e,f)

– La siguiente es la pareja (b,e), pero debe rechazarse también porque su inclusión haría visitar más de una vez los puntos b y e.

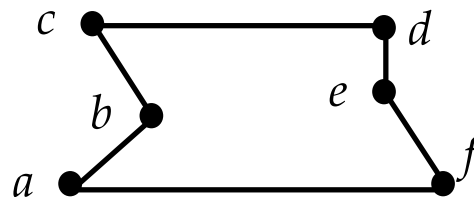
– La pareja (b,d) se rechaza por motivos similares (el punto b habría que visitarlo dos veces).

– La siguiente pareja es (c,d), y se selecciona.

(d,e), (a,b), (b,c), (e,f), (c,d)

– Las parejas (b,f), (c,e), (a,e) y (a,d) no son aceptables.

– Finalmente, la pareja (a,f) cierra el recorrido:



– Este recorrido no es el óptimo pues su longitud es de 50 unidades.

No obstante, es el 4º mejor recorrido de entre los sesenta (esencialmente distintos) posibles y es más costoso que el óptimo en sólo un 3,3%.