Facultat d'Informàtica, U.P.C.

ALGORITMOS VORACES

Anàlisi i Disseny d'Algorismes

María Teresa Abad Soriano
Departamento L.S.I., curso 2007/2008

CONTENIDOS

ALGORITMOS VORACES

1. CARACTERIZACION Y ESQUEMA

Este tema se dedica a presentar un esquema de resolución de problemas denominado método Voraz, *Greedy Method*. Los algoritmos que se obtienen aplicando este esquema se denominan, por extensión, algoritmos voraces. El esquema forma parte de una familia de algoritmos mucho más amplia denominada ALGORITMOS DE BUSQUEDA LOCAL de la que también forman parte, por ejemplo, el método del gradiente, los algoritmos *Hill-Climbing*, los algoritmos genéticos o los *Simulated Annealing*.

Antes de ver propiamente el esquema de resolución, comenzaremos por caracterizar de forma general las condiciones que deben cumplir los problemas que son candidatos a ser resueltos usando un algoritmo voraz:

- El problema a resolver ha de ser de optimización $[MIN/MAX]$ y debe existir una función, la función `objetivo`, que es la que hay que minimizar o maximizar. La siguiente función, que es lineal y multivariable, es una función objetivo típica.

  $$f: \mathbb{N} \times \mathbb{N} \times \ldots \times \mathbb{N} \rightarrow \mathbb{N} \ (\text{o } \mathcal{R}^+)$$
  $$f(x_1, x_2, \ldots, x_n) = c_1 \cdot x_1 + c_2 \cdot x_2 + \cdots + c_n \cdot x_n$$

- Existe un conjunto de valores posibles para cada una de las variables de la función objetivo, su `dominio`.
- Puede existir un conjunto de restricciones que imponen condiciones a los valores del dominio que pueden tomar las variables de la función objetivo.
- La solución al problema debe ser expresable en forma de secuencia de decisiones y debe existir una función que permita determinar cuándo una secuencia de decisiones es solución para el problema (función `solución`). Entendemos por decisión la asociación a una variable de un valor de su domino.
- Debe existir una función que permita determinar si una secuencia de decisiones viola o no las restricciones, la función `factible`.

Esta es una caracterización muy genérica y en temas posteriores se comprobará que también sirve para los problemas que pueden ser resueltos usando *Backtracking* o *Branch&Bound*.

El propósito de un algoritmo voraz es encontrar una solución, es decir, una asociación de valores a todas las variables tal que el valor de la función objetivo sea óptimo. Para ello sigue un proceso secuencial en el que a cada paso toma una decisión (decide qué valor del dominio le ha de asignar a la variable actual) aplicando siempre el mismo criterio (función de **selección**). La decisión es localmente óptima, es decir, ningún otro valor de los disponibles para esa variable lograría que la función objetivo tuviera un valor mejor, y luego comprueba si la puede incorporar a la secuencia de decisiones que ha tomado hasta el momento, es decir, comprueba que la nueva decisión junto con todas las tomadas anteriormente no violan las restricciones y así consigue una nueva secuencia de decisiones factible.
En el siguiente paso el algoritmo voraz se encuentra con un problema idéntico, pero estrictamente menor, al que tenía en el paso anterior y vuelve a aplicar la misma función de selección para tomar la siguiente decisión. Esta es, por tanto, una técnica descendente.
Pero nunca se vuelve a reconsiderar ninguna de las decisiones tomadas. Una vez que a una variable se le ha asignado un valor localmente óptimo y que hace que la secuencia de decisiones sea factible, nunca más se va a intentar asignar un nuevo valor a esa misma variable.

El esquema de un algoritmo voraz puede tener el siguiente aspecto:

```
función VORAZ(C es conjunto) dev (S es conjunto; v es valor)
```
{*Pre:* C es el conjunto de valores posibles que hay que asociar a las variables de la función objetivo para obtener la solución. Sin pérdida de generalidad, se supone que todas las variables comparten el mismo dominio de valores que son, precisamente, los que contiene C. En este caso el número de variables que hay que asignar es desconocido y la longitud de la solución indicará cuantas variables han sido asignadas}

```
    S:= secuencia_vacía;            /* inicialmente la solución está vacía */
```
{*Inv:* S es una secuencia de decisiones factible y C contiene los valores del dominio que todavía se pueden utilizar}

```
    mientras (¬solución(S) ∧ ¬vacio(C)) hacer
        x:= selección(C);           /* se obtiene el candidato localmente óptimo */
        si (factible((S)∪{x})) entonces S:= añadir(S,{x});
        sino C:= C-{x};             /* este candidato no se puede volver a utilizar */
        fsi
```

```
    fmientras
/* la iteración finaliza porque S es una solución o porque el conjunto de candidatos está vacío */
    si (solución(S)) entonces v:= valor(objetivo(S))
    sino S:= sec_vacia
    fsi
```

{*Post:* (S = sec_vacia ⇒ no se ha encontrado solución) ∧ (S ≠ sec_vacia ⇒ S es una secuencia factible de decisiones y es solución ∧ v = valor(S))}

```
    dev (S,v)
ffunción
```

El coste de este algoritmo depende de dos factores:

1. del número de iteraciones, que a su vez depende del tamaño de la solución construida y del tamaño del conjunto de candidatos.
2. del coste de las funciones **selección** y **factible** ya que el resto de las operaciones del interior del bucle pueden tener coste constante. El coste de la función **factible** dependerá del problema (número y complejidad de las restricciones). La función de **selección** ha de explorar el conjunto de valores candidatos y obtener el mejor en ese momento y por eso su coste depende, entre otras cosas, del tamaño del conjunto de candidatos. Para reducir el coste de la función de **selección**, siempre que sea posible se prepara (preprocesa) el conjunto de candidatos antes de entrar en el bucle. Normalmente el preproceso consiste en ordenar el conjunto de valores posibles lo que hace que el coste del algoritmo voraz acabe siendo del orden de $\theta(n \cdot log\, n)$, con n el tamaño del conjunto a ordenar.

El proceso de construcción de la solución produce un comportamiento de los algoritmos voraces muy especial. Por un lado, el hecho de utilizar un algoritmo voraz para obtener la solución de un problema no garantiza que la solución obtenida sea la óptima pero, por el contrario e independientemente de que la consigan o no, el coste invertido es pequeño ya que el algoritmo sólo genera UNA de entre todas las posibles secuencias de decisiones. Y como las decisiones localmente óptimas no garantizan que siempre se vaya a obtener la combinación de valores que optimiza el valor de la función objetivo, el óptimo global, siempre habrá que intentar demostrar que la solución obtenida es la óptima. La inducción es la técnica de demostración de optimalidad más utilizada.

2. PROBLEMAS SIMPLES

El conjunto de problemas que se puede resolver aplicando el esquema voraz es muy amplio. En éste y los siguientes apartados se va a resolver un subconjunto de los denominados simples y se van a presentar las técnicas básicas de demostración de optimalidad.

Para cada problema el proceso a seguir es siempre el mismo. En primer lugar, determinar la función de **selección**, definir la función factible y definir qué caracteriza a una **solución** del problema. A continuación hay que demostrar que la función de selección conduce siempre al óptimo. Si la demostración ha tenido éxito, en último lugar hay que construir el algoritmo pero resulta tan sencillo después de los pasos previos que en muchas ocasiones ni tan siquiera se considera necesario. Si la demostración no ha tenido éxito, se pueden buscar nuevas funciones de selección y si se continúa sin tener éxito es probable que haya que intentarlo con un esquema distinto.

2.1. **El problema del cambio**

Dado un conjunto C con N tipos de monedas y con un número inagotable de ejemplares de cada tipo, hay que conseguir, si se puede, formar la cantidad M empleando el mínimo número de ejemplares de monedas.

Este es un problema de minimización que satisface el denominado PRINICIPIO DE OPTIMALIDAD lo que puede resultar útil para las demostraciones de optimalidad de la solución obtenida por la función de selección. En este caso podemos enunciar el principio de optimalidad de la siguiente forma:

Sea $A = \{e_1, e_2, \ldots, e_k\}$ una solución óptima para el problema de formar la cantidad M, siendo e_i , con $1 \leq i \leq k$, ejemplares de algunos de los tipos de monedas de C, entonces sucede que $A - \{e_1\}$ es también solución óptima para el problema de formar la cantidad $M - valor(e_1)$. Volveremos sobre este principio en el tema de Programación Dinámica.

Para el problema que nos ocupa se puede comenzar por definir factible y **solución**. En este caso el problema no tiene restricciones explícitas pero podemos decir que un conjunto de monedas es factible si la suma de sus valores es menor o igual que M. Y se tiene una **solución** cuando los valores de las monedas elegidas suman exactamente M, si es que esto es posible.

La forma en que habitualmente se devuelve el cambio nos ofrece un candidato a función de **selección**: elegir a cada paso la moneda disponible de mayor valor.

Como pretendemos minimizar el número de monedas utilizado, al elegir la moneda de mayor valor nos acercamos lo más posible a la cantidad a formar y, además, empleando una única moneda. Es una decisión localmente óptima porque minimiza la cantidad que nos queda por formar con las monedas de C. La función factible tendrá que comprobar que el valor de la moneda seleccionada junto con el valor de las monedas que ya hemos incluido hasta ahora en la solución no supera M. Inmediatamente se observa que para facilitar el trabajo de la función de selección se puede ordenar el conjunto C, de tipos de monedas de la entrada, por orden decreciente de valor. De este modo, la función de selección sólo tiene que tomar la primera moneda de la secuencia ordenada para conseguir la de mayor valor del conjunto.

```
función MONEDAS(C es conj_tipo_monedas; M es natural) dev (S es sec_monedas)
```
{*Pre:* C contiene los tipos de monedas a utilizar. Hay infinitos ejemplares de cada tipo. M es la cantidad que hay que formar con las monedas de C}

```
        S:=secuencia_vacia;        /* aquí guardamos la solución */
        L:= ORDENAR_DECREC(C);     /* se ordenan los ejemplares por orden decreciente de valor */
        mientras (suma(S)<M ∧ ¬vacia(L)) hacer
                x:= primero(L);    /* x es un ejemplar de un tipo de moneda de las de C */
                si ((suma(S) + x) ≤ M) entonces
                      S:= añadir(S, {x});
                /* es factible y se añade un ejemplar de la moneda de tipo x a la solución en curso*/
                sino L:= avanzar(L);
 /* El tipo de moneda x se elimina del conjunto de candidatos para no ser considerado nunca más */
                fsi
        fmientras
        si (suma(S) < M) entonces S:= secuencia_vacia;
        fsi
```
{*Post:* si S vacia es que no hay solución, en caso contrario S contiene monedas tal que la suma de sus valores es exactamente M}

```
        dev (S)
ffunción
```

Cubierta la primera parte del proceso, ahora hay que demostrar la optimalidad de este criterio, es decir que sea cual sea C y sea cual sea M, el proceso descrito garantiza que siempre va a encontrar una solución que emplee el mínimo número de monedas. Basta con un contraejemplo para comprobar que no siempre es así. Si $C = \{1, 5, 11, 25, 50\}$ y $M = 65$, la estrategia voraz descrita obtiene la solución formada por seis monedas: una moneda de 50, una moneda de 11 y cuatro

monedas de 1. Es fácil comprobar que la solución óptima sólo necesita cuatro monedas que son: una de 50 y tres de 5.
No está todo perdido y se puede intentar caracterizar C de modo que esta estrategia funcione siempre correctamente. En primer lugar se necesita que C contenga la monedad unidad. Caso de que esta moneda no exista, no se puede garantizar que el problema tenga solución y, tampoco, que el algoritmo la encuentre. En segundo lugar C ha de estar compuesto por tipos de monedas que sean potencia de un tipo básico t, con $t > 1$. Con estas dos condiciones sobre C, el problema se reduce a encontrar la descomposición de M en base t. Se sabe que esa descomposición es única y, por tanto, mínima con lo que queda demostrado que en estas condiciones este criterio conduce siempre al óptimo.

Intencionadamente en la postcondición del algoritmo presentado no se dice nada de la optimalidad de la solución porque en la precondición tampoco se dice nada sobre las características de C y no podemos asegurar que cumple las condiciones que garantizan un solución óptima.

El coste del algoritmo es $O(max(n \log n, m))$ siendo n el número de tipos de monedas de C y m el número de iteraciones que realiza el bucle. Una cota superior de m es $M + n$.

2.2. **Planificación de tareas**

Un procesador ha de atender n procesos. Se conoce de antemano el tiempo que necesita cada proceso. Determinar en qué orden ha de atender el procesador a los procesos para que se minimice la suma del tiempo que los procesos esperan a ejecutarse.

Se trata de un problema que no está sometido a restricciones y una solución es una secuencia de tamaño *n* en la que aparecen los *n* procesos a atender. Hay obtener la permutación de las *n* tareas que minimice la función objetivo. El orden de aparición de una tarea en la permutación indicará el orden de atención por parte del procesador.

Un ejemplo dará ideas sobre la función de selección. Supongamos que hay 3 procesos p_1, p_2 y p_3 y que el tiempo de proceso que requiere cada uno de ellos es 5, 10 y 3 respectivamente. Vamos a calcular la suma del tiempo que los procesos

están esperando en el sistema para cada una de los seis posibles órdenes en que se podrían atender las 3 tareas:

orden de atención	tiempo de espera
p_1, p_2, p_3	5 + (5 + 10) = 20
p_1, p_3, p_2	5 + (5 + 3) = 13
p_2, p_1, p_3	10 + (10 + 5) = 25
p_2, p_3, p_1	10 + (10 + 3) = 23
p_3, p_1, p_2	3 + (3 + 5) = 11
p_3, p_2, p_1	3 + (3 + 10) = 16

La ordenación que produce el tiempo de espera mínimo es la $\{p_3, p_1, p_2\}$ o, lo que es lo mismo, la que atiende en orden creciente de tiempo de proceso.
Hay que comprobar que aplicando este criterio de selección, es decir, elegir la tarea con tiempo de proceso menor de entre todas las que quedan por procesar, conduce siempre al óptimo.

Prueba: Sea S_1 una permutación cualquiera de las *n* tareas y distinta de la obtenida por el algoritmo voraz. Necesariamente en S_1 habrá tareas no ordenadas crecientemente por su tiempo de proceso. Sean x e y dos de esos procesos tales que $p_x > p_y$ y supongamos que en i-ésimo lugar se ha colocado el proceso x y que en j-ésimo lugar se ha colocado el proceso y con $i < j$. El tiempo de espera de S_1 es:

$$\begin{aligned} TE(S_1) = p_1 + (p_1 + p_2) + (p_1 + p_2 + p_3) + \cdots + (p_1 + \cdots + p_{i-1}) + \cdots \\ + \left(p_1 + \cdots + p_x + \cdots + p_y + \cdots + p_{n-1}\right) \\ = (n-1)p_1 + (n-2)p_2 + \ \ldots + (n-i)p_x + \cdots + (n-j)p_y + \cdots \\ + \big(n - (n-1)\big)p_{n-1} \end{aligned}$$

Para que S_1 se parezca un poco más a la solución voraz podemos permutar las posiciones que ocupan los procesos x e y, de este modo, por lo menos, ellos dos estarán ordenados. Sea S_2 la permutación obtenida a partir de S_1 y en la que hemos efectuado esa permutación. El tiempo de espera de S_2 será:

$$TE(S_2) = (n-1)p_1 + (n-2)p_2 + \ \ldots + (n-i)p_y + \cdots + (n-j)p_x + \cdots + \big(n - (n-1)\big)p_{n-1}$$

¿Qué habrá sucedido con el tiempo de espera de S_2?, ¿será mayor o menor que el de S_1? Calculemos su diferencia:

$$\begin{aligned} TE(S_1) - TE(S_2) &= (n-i)p_x + (n-j)p_y - (n-i)p_y - (n-j)p_x \\ &= (n-i)\big(p_x - p_y\big) + (n-j)\big(p_y - p_x\big) \end{aligned}$$

Y como $i < j$ y $p_x > p_y$ resulta que

$$TE(S_1) - TE(S_2) > 0$$

De este resultado se deduce que ordenar parcialmente reduce el tiempo de espera. Repetimos el proceso descrito para todo par de procesos desordenado obteniendo una nueva permutación que tendrá un tiempo de espera menor que la de partida. Este proceso acabará cuando no queden más pares desordenados. La permutación final estará ordenada en orden creciente de tiempo de proceso y ya no será posible reducir su tiempo de espera por lo que será óptima. El algoritmo voraz propuesto nos da una solución que coincide con la óptima por lo que el criterio de selección propuesto siempre conduce a una solución óptima.

El coste del algoritmo voraz para producir la secuencia solución óptima es el de ordenar los procesos $\theta(n\ log\ n)$.

2.3. **Planificación de tareas (otro)**

Dadas n actividades, $A = \{1, 2, \ldots, n\}$, que han de usar un recurso en exclusión mutua, y dado que cada actividad i tiene asociado un instante de inicio y otro de fin de utilización del recurso, s_i y f_i respectivamente con $s_i \leq f_i$, seleccionar el conjunto que contenga el máximo de actividades sin que se viole la exclusión mutua.

En este caso hay que maximizar la cardinalidad del conjunto solución que estará formado por un subconjunto de las actividades iniciales. La función factible tendrá que tener en cuenta que al añadir una nueva actividad a la secuencia solución no se viole la exclusión mutua (no se solape con ninguna de las ya colocadas).

Existen varios criterios que pueden ser utilizados como función de selección:
1/ elegir la actividad que acaba antes de entre todas las candidatas
2/ elegir la actividad que empieza antes de entre todas las candidatas.
3/ elegir la actividad de menor duración de entre todas las candidatas.
4/ elegir la actividad de mayor duración de entre todas las candidatas.
Es fácil encontrar contraejemplos simples para los criterios 2, 3 y 4 pero parece que el primero siempre funciona, es decir, encuentra el conjunto de tamaño máximo. Intentemos demostrar la optimalidad del criterio. Para facilitar la escritura de la demostración diremos que las actividades i y j son COMPATIBLES si $f_j \leq s_i$ ó $f_i \leq s_j$ (no se solapan sea cual sea la secuencia de ejecución entre ambas).

Prueba: Sea $SO = < o_1, o_2, \dots, o_m >$ una solución óptima para el conjunto de actividades $A = \{t_1, t_2, \dots, t_n\}$ con $m \leq n$, tal que se presentan ordenados por tiempo de finalización, es decir, $f_{o1} \leq f_{o2} \leq \cdots \leq f_{om}$.

Sea $G = < g_1, g_2, \dots, g_k >$, con $SO \neq G$, la solución que ha calculado el algoritmo voraz para el mismo conjunto de tareas A y que, por construcción, cumple que $k \leq n$ y que las tareas están ordenadas por instante de finalización, $f_{g1} \leq f_{g2} \leq \cdots \leq f_{gk}$

1º/ Vamos a ver que toda solución óptima puede comenzar con la actividad g_1, que es la que acaba antes de todas las de A y que es la que elige el algoritmo voraz.

- Si $o_1 = g_1$ no hay nada que comprobar. En este caso la primera elección del voraz coincide con la primera decisión de la solución óptima.
- Si $o_1 \neq g_1$, entonces SO comienza por una tarea, o_1, que acaba después de la que ha colocado el voraz, g_1 ($f_{g1} \leq f_{o1}$) porque el voraz coloca en primer lugar a la tarea que acaba antes de todo el conjunto. Además, como el resto de tareas que figuran en SO son compatibles con o_1, también lo son con g_1 ya que ésta acaba incluso antes.

Sea $SO' = SO - \{o_1\} \cup \{g_1\}$, entonces SO' sigue siendo una solución óptima ya que no hemos variado su cardinalidad.

Acabamos de demostrar que toda solución óptima puede comenzar por la misma tarea que aparece en primer lugar en la solución del algoritmo voraz.

2º/ Una vez analizada la primera elección, la actividad g_1, el nuevo problema que hay que resolver es maximizar el conjunto de actividades que sean compatibles con g_1 (ahora ya sabemos con toda seguridad que g_1 forma parte de la solución óptima y que todas las tareas que forman la solución óptima son compatibles entre ellas) y las soluciones que debemos comparar son $G' = G - \{g_1\}$ y $SO'' = SO' - \{g_1\}$. Aplicamos sobre ellas exactamente el mismo razonamiento del punto anterior el número de veces necesario y acabamos demostrando que una solución óptima es la que ha encontrado el voraz, G.

Finalmente, el algoritmo voraz que se propone ordena por instante de terminación el conjunto de actividades y luego las procesa en ese orden decidiendo en cada iteración si la que acaba antes de todas las que quedan por procesar se puede colocar o no en la solución. El coste del algoritmo es $\theta(n \log n)$ y se debe a la ordenación previa de las actividades ya que el coste del bucle es $\theta(n)$.

```
función PLAN_TAREAS(l es lista_tareas) dev (s es lista_tareas)
{Pre: l contiene la lista de tareas y cada una de ellas tiene asociado un instante de inicio y otro de fin}
        lo:= ordenar_crec_por_instante_fin(l);
        tfin:=0;
```

```
        mientras (¬vacia(lo)) hacer
                t:= primera(lo); avanzar(lo);
                si (instante_inicio(t) ≥ tfin) entonces
                        s:= añadir(s, t);
                        tfin:= instante_fin(t);
                fsi
        fmientras
        dev s
```

{*Post*: s contiene un conjunto de actividades compatibles entre sí (no se solapan) y de tamaño máximo}

```
Ffuncion
```

3. MOCHILA

Se dispone de una colección de n objetos y cada uno de ellos tiene asociado un peso y un valor. Más concretamente, se tiene que para el objeto i, con $1 \leq i \leq n$, su valor es v_i y su peso es p_i. La mochila es capaz de soportar, como máximo, el peso $PMAX$. Determinar qué objetos hay que colocar en la mochila de modo que el valor total que transporta sea máximo pero sin que se sobrepase el peso máximo $PMAX$ que puede soportar.

Este problema es conocido por *The Knapsack problem*. Existen dos posibles formulaciones de la solución:

1/ *Mochila fraccionada*: se admite fragmentación, es decir, se puede colocar en la solución un trozo de alguno de los objetos.

2/ *Mochila entera*: NO se admite fragmentación, es decir, un objeto o está completo en la solución o no está.

Para ambas formulaciones el problema satisface el principio de optimalidad. Ahora bien, mochila fraccionada se puede resolver aplicando una estrategia voraz mientras que, de momento, mochila entera no y es necesario un Vuelta Atrás o Programación Dinámica.

Intentaremos, por tanto, encontrar una buena función de selección que garantice el óptimo global para el problema de mochila fraccionada. Formalmente se pretende:

$$[MAX] \sum_{i=1}^{n} x_i \cdot v_i$$

$$\sum_{i=1}^{n} x_i \cdot p_i \leq PMAX$$

$$0 \leq x_i \leq 1$$

La solución al problema viene dada por el conjunto de valores x_i con $1 \leq i \leq n$, siendo x_i un valor real entre 0 y 1 que se asocia al objeto i. Así, si el objeto *3* tiene asociado un 0 significará que este objeto no forma parte de la solución, pero si tiene asociado un 0.6 significará que un 60% de él está en la solución.

Es posible formular unas cuantas funciones de selección. De las que se presentan a continuación las dos primeras son bastante evidentes mientras que la tercera surge después del fracaso de las otras dos:

1/ Seleccionar los objetos por orden creciente de peso. Se colocan los objetos menos pesados primero para que la restricción de peso se viole lo más tarde posible. Con un contraejemplo se ve que no funciona.

2/ Seleccionar los objetos por orden decreciente de valor. De este modo se asegura que los más valiosos serán elegidos primero. Tampoco funciona.
3/ Seleccionar los objetos por orden decreciente de relación valor/peso. Así se consigue colocar primero aquellos objetos cuya unidad de peso tenga mayor valor. Parece que si funciona. Ahora se ha de demostrar la optimalidad de este criterio.

Demostración:
Sea $X = (x_1, x_2, \dots, x_n)$ la secuencia solución generada por el algoritmo voraz aplicando el criterio de seleccionar los objetos por orden decreciente de relación valor/peso. Por construcción, la secuencia solución es de la forma $X = (1,1,1,\dots,1,0.x,0,\dots,0,0)$, es decir, los primeros objetos se colocan enteros, aparecen con valor 1 en la solución, un sólo objeto se coloca parcialmente, valor $0.x$, y todos los objetos restantes no se colocan en la mochila, por tanto valor 0.
Si la solución X es de la forma $\forall i : 1 \leq i \leq n : x_i = 1$, seguro que es óptima ya que no es mejorable. Pero si X es de la forma $\exists i : 1 \leq i \leq n : x_i \neq 1$, entonces no se sabe si es óptima.
Sea j el índice más pequeño tal que $(\forall i : 1 \leq i < j : x_i = 1)$ y $(x_j \neq 1)$ y $(\forall i : j+1 \leq i \leq n : x_i = 0)$. El índice j marca la posición que ocupa el objeto que se fragmenta en la secuencia solución.
- Supongamos que X NO es óptima. Esta suposición implica que existe otra solución, Y, que obtiene más beneficio que X, es decir,

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

Además, por tratarse de mochila fraccionada, ambas soluciones llenan la mochila completamente y satisfacen:

$$\sum_{i=1}^{n} x_i \cdot p_i = \sum_{i=1}^{n} y_i \cdot p_i = PMAX$$

Sea k el índice más pequeño de Y tal que $x_k \neq y_k$ (comenzando por la izquierda, en la posición k se encuentra la primera diferencia entre X e Y). Se pueden producir tres situaciones:
1/ $k < j$. En este caso $x_k = 1$ lo que implica que $x_k > y_k$.
2/ $k = j$. Forzosamente $x_k > y_k$ porque en caso contrario la suma de los pesos de Y sería mayor que $PMAX$ y, por tanto, Y no sería solución.
3/ $k > j$. Imposible que suceda porque implicaría que la suma de los pesos de Y es mayor que $PMAX$ y, por tanto, Y no sería solución.

Del análisis de los tres casos se deduce que si Y es una solución óptima distinta de X, entonces $x_k > y_k$. Como $x_k > y_k$, se puede incrementar y_k hasta que $y_k = x_k$, y decrementar todo lo que sea necesario desde y_{k+1} hasta y_n para que el peso total continue siendo $PMAX$. De este modo se consigue una tercera solución $Z = (z_1, z_2, \ldots, z_n)$ tal que:

$$(\forall i : 1 \leq i \leq k : z_i = x_i) \qquad \text{y}$$
$$\sum_{i=k+1}^{n} (y_i - z_i) \cdot p_i = (z_k - y_k) \cdot p_k$$

Para la nueva solución Z se tiene:

$$\sum_{i=1}^{n} z_i \cdot v_i = \sum_{i=1}^{n} y_i \cdot v_i + \left(v_k \frac{p_k}{p_k}\right) \cdot (z_k - y_k) - \sum_{i=k+1}^{n} \left(v_i \frac{p_i}{p_i}\right) \cdot (y_i - z_i) \geq \sum_{i=1}^{n} y_i \cdot v_i$$
$$+ (p_k \cdot (z_k - y_k) - \sum_{i=k+1}^{n} p_i \cdot (y_i - z_i)) \cdot \frac{v_k}{p_k} = \sum_{i=1}^{n} y_i \cdot v_i$$

Con lo que finalmente obtenemos la desigualdad:

$$\sum_{i=1}^{n} z_i \cdot v_i \geq \sum_{i=1}^{n} y_i \cdot v_i$$

Pero, en realidad, sólo es posible la igualdad ya que el menor indicaría que Y no es óptima lo que contradice la hipótesis de partida. Se puede concluir que Z también es una solución óptima. Basta repetir el proceso el número de veces que sea necesario para que se consiga una solución óptima, T, idéntica a X. Concluimos que X, la solución obtenida por el algoritmo voraz, es óptima.

Un algoritmo que resuelve este problema, y que tiene un coste $\theta(n \cdot log\, n)$, es el que viene a continuación. En él los pesos y los valores de los objetos están almacenados en los vectores `p` y `v` respectivamente:

```
función MOCHILA(v, p es vector de reales; n, PMAX es nat ) dev
                          ( x es vector[1..n] de reales; VX es real )
```

{Pre: v y p son vectores que contienen los valores y los pesos de los n objetos y PMAX es el peso máximo que puede soportar la mochila*}*

```
        Para j=1 hasta n hacer
                VC[j].c:= v[j]/p[j];  VC[j].id:= j;
                x[j]:= 0;
        fpara
```

```
        VC:= ORDENAR_DECREC( VC, c );
```
/* se ordenan los objetos por el campo c, el que contiene el cociente entre valor y peso */
```
        s:= 0;     /* peso acumulado hasta el momento en la mochila */
        VX:= 0 ;  /* valor acumulado hasta el momento*/
        i:=1;
        mientras ((i ≤ n) ∧ (s < PMAX)) hacer
            k:= VC[i].id;
        /* el objeto i de la entrada ordenada corresponde al objeto k de la entrada sin ordenar */
            si (s+p[k] ≤ PMAX) entonces
                x[k]:= 1;      /*anotamos que el objeto k forma parte de la solución */
                VX:= VX + v[k] ; s:= s + p[k];
            sino
                x[k]:= (PMAX-s)/p[k];   /* porción del objeto k que colocamos en la solución */
                s:= PMAX;
                VX:= VX+(v[k] × x[k]);
            fsi
                i:= i+1;
         fmientras
```
{Post : x es la solución que maximiza el valor de los objetos colocados en la mochila. El valor de x es VX *}*
```
        dev (x, VX)
ffunción
```

4. ÁRBOLES DE EXPANSIÓN MÍNIMA

Sea $G =< V, E >$ un grafo no dirigido conexo y etiquetado con valores naturales. Diseñar un algoritmo que calcule un subgrafo de G, $T =< V, F >$ con $F \subseteq E$, conexo y sin ciclos, tal que la suma de los pesos de las aristas de T sea mínima. El subgrafo T que hay que calcular se denomina ÁRBOL LIBRE ASOCIADO A G. El árbol libre de G que cumple que la suma de los pesos de las aristas que lo componen es mínima se denomina el ARBOL DE EXPANSIÓN MÍNIMA DE G, *minimun spaning tree* ó MST para abreviar.

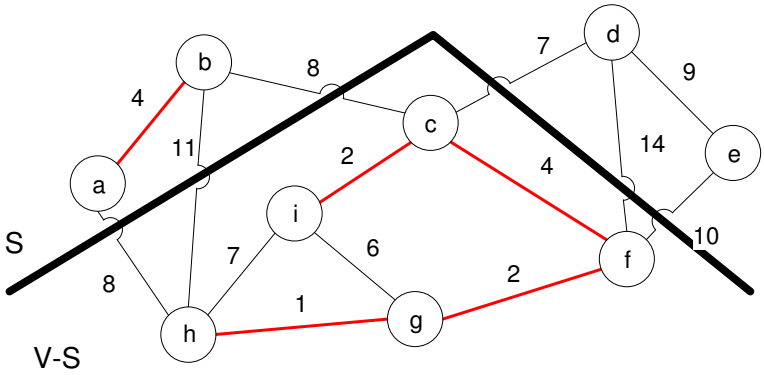
4.1. **Algoritmo genérico**

El MST del grafo dado se puede representar como un conjunto de aristas. Vamos a presentar un algoritmo genérico que construye el árbol de expansión mínima a base de ir añadiendo una arista cada vez. Se trata de un algoritmo voraz. El razonamiento es bastante simple: si T es un árbol de expansión mínima de G y A es el conjunto de aristas que hasta el momento se han incluido en la solución, el invariante del algoritmo dice que A siempre es un subconjunto del conjunto de aristas de T ($A \subseteq T$). En cada iteración la función de selección, `SELECCIONAR_ARISTA_BUENA`, elige una arista que permite que se mantenga el invariante, es decir, elige una arista $(u, v) \in E$ de modo que si A es un subconjunto de T, entonces $A \cup \{(u, v)\}$ sigue siendo un subconjunto de T. La arista (u, v) que elige la función de selección se dice que es una ARISTA BUENA para A.

```
función MST_GENERICO( g es grafo ) dev ( A es conj_aristas; w es natural )
{Pre: g=(V,E) es un grafo no dirigido, conexo y etiquetado con valores naturales}
        A:= conjunto_vacio ; w:= 0;
{Inv: A es un subconjunto de un árbol de expansión mínima de g}
        mientras (|A| < n-1) hacer
                (u,v):= SELECCIONAR_ARISTA_BUENA(g, A);
                A:= A∪{(u,v)}
                w:= w + valor(g, u, v) ;
        fmientras
{Post : A es un MST de g y su peso, la suma de las etiquetas de las aristas que lo forman, es w y es el mínimo posible}
        dev (A, w)
ffunción
```

El problema es encontrar una arista buena para A de forma eficiente y haciendo que se satisfaga el invariante. Antes de abordar la cuestión de la eficiencia vamos con unas cuantas definiciones previas.

*De*finició*n*: Un *corte* $(S, V–S)$ de un grafo no dirigido $G =< V, E >$ es una partición cualquiera de V.
Definición: Se dice que una arista *cruza el corte* si uno de los vértices de la arista pertenece a S y el otro pertenece a $V–S$.
Definición: Se dice que el corte *respeta* a A si ninguna arista de A cruza el corte.
Definición: Una arista es una *c_arista* si cruza el corte y tiene la etiqueta mínima de entre todas las que cruzan el corte.



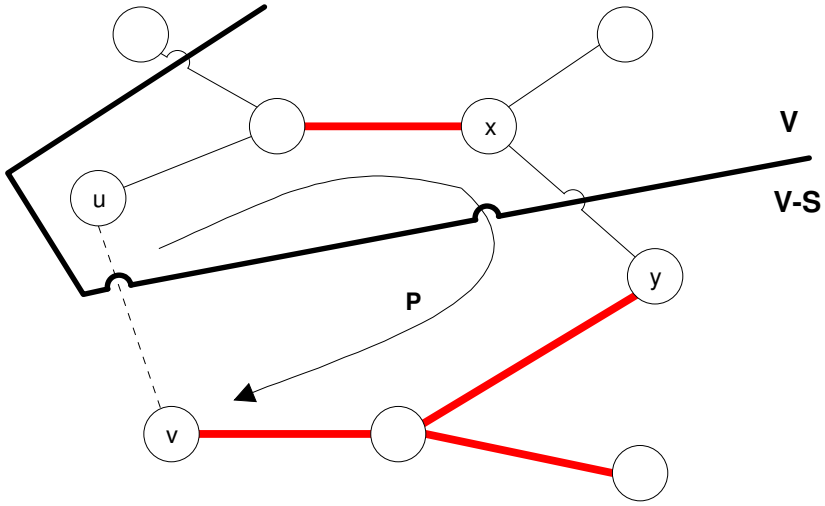
La línea en trazo grueso fija el corte. Las aristas en rojo pertenecen a A. El corte respeta a A y la arista (c,d) es una c_arista (cruza el corte y es de peso mínimo).

El siguiente teorema enuncia las condiciones que ha de cumplir una arista para que se considere que es una arista buena para A y son las que usaremos en la función de selección en el algoritmo voraz genérico:

Teorema: Sea $G =< V, E >$ un grafo no dirigido, conexo y etiquetado con valores naturales. Sea A un subconjunto de algún árbol de expansión mínima de G. Sea $(S, V–S)$ un corte de G que respeta a A y sea (u, v) una *c_arista* que cruza el corte, entonces (u, v) es una arista buena para A.

En primer lugar el teorema fija un corte que respeta a A (cualquiera que lo respete es válido) y a continuación, de entre todas las aristas que cruzan ese corte, elige la arista de peso mínimo. La función de selección que nos propone no sólo devuelve una arista localmente óptima sino que garantiza que la unión de la arista seleccionada con las que forman la solución en curso forman un conjunto de aristas factible.

Demostración: Supongamos que A es un subconjunto de T, siendo T un árbol de expansión mínima de G, y que $(u,v) \notin T$, siendo (u,v) una *c_arista* para un corte que respeta a A. La demostración va a construir otro árbol de expansión mínima, T', que incluye a $A \cup \{(u,v)\}$ usando una técnica de 'cortar&pegar', y además se verá que (u,v) es una arista buena para A.



Todas las aristas, excepto (u,v), forman parte de T. Las marcadas en rojo son las que pertenecen a A. El corte está marcado en trazo negro grueso.

La situación de partida la muestra el dibujo previo. En él se observa que la arista (u,v) forma un ciclo con las aristas en el camino P que va de u a v en T. Ya que u y v están en lados opuestos del corte $(S, V-S)$, hay como mínimo en T una arista que también cruza el corte. Sea (x,y) esa arista. (x,y) no pertenece a A porque el corte respeta a A. Ya que (x,y) se encuentra en el único camino de u a v en T, eliminando (x,y) se fragmenta T en dos árboles. Si se le añade la arista (u,v) se vuelve a conectar formándose el nuevo árbol $T' = T - \{(x,y)\} \cup \{(u,v)\}$.

Calculando el peso de T' se tiene que:

$$peso(T') = peso(T) - valor(x,y) + valor(u,v)$$

y como (u,v) es una *c_arista*, entonces

$$valor(u,v) \leq valor(x,y)$$

y, por tanto:

[1] $peso(T') \leq peso(T)$.

Por otro lado, se tiene que T es un árbol de expansión mínima de G y, por eso,

[2] $peso(T) \leq peso(T')$

De la certeza de los hechos **[1]** y **[2]** se deduce que

$$peso(T) = peso(T')$$

y que, por tanto, T' es también un árbol de expansión mínima de G.

Queda por comprobar que (u, v) es una buena arista para A, o sea que $A \cup \{(u, v)\}$ sigue siendo un subconjunto de algún árbol de expansión mínima de G. Se tiene que $A \subseteq T'$ y como también $A \subseteq T$ y $(x, y) \notin A$, entonces $A \cup \{(u, v)\} \subseteq T'$. Se ha comprobado que T' es un árbol de expansión mínima y de ello se deduce que (u, v) es una arista buena para A.

El invariante del bucle, además de contener que A es siempre un subconjunto de T, incluye el hecho de que el grafo $G_A =< V, A >$ es un bosque y cada una de sus componentes conexas es un árbol libre. Se deduce del hecho de que (u, v) es una arista buena para A y por tanto esta arista conecta dos componentes conexas distintas. En caso contrario, sucedería que $A \cup \{(u, v)\}$ contendría un ciclo. También se puede asegurar que la arista de valor mínimo de todo el grafo forma parte de algún árbol de expansión mínima de G.

Los algoritmos de Kruskal y Prim que se van a presentar a continuación, calculan el MST de un grafo dado. Se diferencian en la forma de construcción del mismo. Kruskal se conforma con mantener un conjunto de aristas, T, como un bosque de componentes conexas para a base de iterar lograr una sola componente conexa. Prim impone que el conjunto de aristas contenga, de principio a fin, una sola componente conexa. Veámoslos.

4.2. **Kruskal**

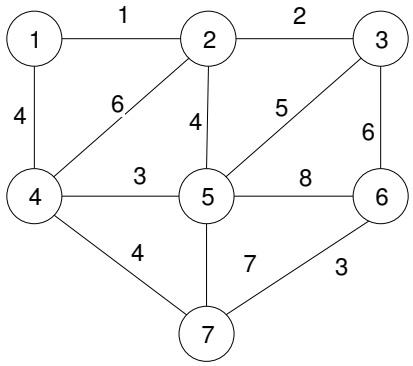
El algoritmo de Kruskal parte de un subgrafo de G, $T =< V, conjunto\_vacio >$, y construye, a base de añadir una arista de G cada vez, un subgrafo $T =< V, A >$ que es el MST deseado. Utiliza una función de selección que elige aquella arista de valor mínimo, de entre todas las que no se han procesado, y que conecta dos vértices que forman parte de dos componentes conexas distintas. Este hecho garantiza que no se forman ciclos y que T siempre es un bosque compuesto por árboles libres. Para reducir el coste de la función de selección, se ordenan previamente las aristas de G por orden creciente del valor de su etiqueta.

Para demostrar la optimalidad de la solución obtenida por el algoritmo de Kruskal bastará con comprobar que su funcionamiento se ajusta al del algoritmo genérico presentado en la sección anterior. Según el teorema, en cada iteración se fija un corte que respete a A y se elige la arista de peso mínimo de entre todas las que cruzan el corte. Kruskal procede en orden inverso: primero selecciona la arista de peso mínimo de entre todas las del grafo que aún no ha considerado y luego

comprueba si esa arista cruza algún corte que respeta a A simplemente comprobando que no forma ningún ciclo con las aristas que ya forman parte de la solución. Una implementación inicial del algoritmo de Kruskal con un coste de $O(n \cdot |E|)$ se encuentra a continuación:

```
función KRUSKAL(g es grafo) dev (T es conj_aristas)
{Pre: g=<V,E> no dirigido, conexo y etiquetado con valores naturales}
        EO:= ORDENAR_CRECIENTE(E);
        n:= |V|;     T:= conjunto_vacio;
        mientras (|T| < n-1) hacer
            (u,v):= primero(EO);
            EO:= avanzar(EO);
        /* (u,v) es la arista seleccionada, es una arista de peso mínimo pero falta comprobar que es
        buena, es decir que cruza algún corte que respeta T */
            b:=F;
            b:= forma_ciclo(T, (u,v));
         /* si b es falso es porque (u,v) no forma ciclo con ninguna de las
         aristas que ya hemos colocado en T por tanto, es de peso mínimo y cruza
         algún corte que respeta a T. (u,v) es una arista buena para T */
            si no(b) entonces
                    T:= T∪{(u,v)};
            fsi
          fmientras
{Post: T es un MST de g}
        dev (T)
ffunción
```

Veamos, con un ejemplo, como funciona el algoritmo. Sea G el grafo de la siguiente figura:



La lista de las aristas de G ordenadas por orden creciente de valor es:

$$\{ (1,2), (2,3), (4,5), (6,7), (1,4), (2,5), (4,7), (3,5), (2,4), (3,6), (5,7), (5,6) \}$$

La tabla que viene a continuación muestra el trabajo de las siete iteraciones del bucle. Inicialmente cada vértice pertenece a un grupo distinto y al final todos forman parte del mismo grupo. Las aristas examinadas y no rechazadas son las aristas que forman el MST.

ETAPA	ARISTA EXAMINADA	GRUPOS DE VERTICES
Incializ.	----	{1}, {2}, {3}, {4}, {5}, {6}, {7}
1	(1,2), valor 1	{1, 2}, {3}, {4}, {5}, {6}, {7}
2	(2,3), valor 2	{1, 2, 3}, {4}, {5}, {6}, {7}
3	(4,5), valor 3	{1, 2, 3}, {4, 5}, {6}, {7}
4	(6,7), valor 3	{1, 2, 3}, {4, 5}, {6, 7}
5	(1,4), valor 4	{1, 2, 3, 4, 5}, {6, 7}
6	(2,5) ARISTA RECHAZADA (forma ciclo)	
7	(4,7), valor 4 peso(T) = 17	{1, 2, 3, 4, 5, 6, 7}

En la implementación del algoritmo que viene a continuación, se utiliza la variable *MF* que es del tipo *MFSet* y que se utiliza para mantener los distintos grupos de vértices. De esta forma se consigue un algoritmo más eficiente que la versión inicial.

```
función KRUSKAL(g es grafo) dev (T es conj_aristas)
{Pre: g=<V,E> no dirigido, conexo y etiquetado con valores naturales}
        EO:= ORDENAR_CRECIENTE(E);
        n:= |V|;     T:= conjunto_vacio;     MF:= iniciar();
        Para cada v∈V hacer MF:= añadir(MF, {v}) fpara
        /* al comienzo cada vértice forma parte de un grupo distinto */
        mientras (|T| < n-1) hacer
             (u,v):= primero(EO);
             EO:= avanzar(EO);
             /* (u,v) es la arista seleccionada, es una arista buena para T */
             x:= find(MF, u);   /* x=identificador del conjunto en el que está u*/
             y:= find(MF, v);   /* lo mismo para v */
/*en el caso en que  x=y, la arista (u,v) se rechaza porque provoca ciclo */
             si (x≠y) entonces
                  /* x≠y, se fusionan los conjuntos y la arista (u,v) se añade al MST */
                  MF:= merge(MF, x, y);
                  T:= T∪{(u,v)};
             fsi
          fmientras
{Post: T es un MST de g}
        dev (T)
ffunción
```

El coste de ordenar el conjunto de aristas es $\theta(|E| \cdot log\ n)$ y el de inicializar el MFSet es $\theta(n)$. Cada iteración lleva a cabo dos operaciones `find` y, algunas veces, también una operación `merge`. El bucle, en el peor de los casos, lleva a cabo $2 \cdot |E|$ operaciones `find` y exactamente $n-1$ operaciones `merge`. Sea $m = 2 \cdot |E| + n - 1$, entonces el bucle cuesta $O(m \cdot log^* n)$ que es $O(|E| \cdot log^* n)$ aproximadamente. Como el coste del bucle es menor que el coste de las inicializaciones, el coste del algoritmo queda $\theta(|E| \cdot log\ n)$.
Y para acabar este apartado una implementación de este algoritmo en C++ usando *MFSets* y una cola de prioridad.

```
//DEFINICIONES COMUNES PARA KRUSKAL Y PRIM

#include <vector>
#include <list>
struct aresta {
        int u; // vértice origen
        int v; // vértice destino
        double p; // peso de la arista
        aresta (int u, int v, double p) : u(u), v(v), p(p) { }
        bool operator< (const aresta& b) const {
        return p > b.p;         // al revés porque las priority_queue son para
                                // máximos y en estos algoritmos interesa el
                                // mínimo
        }
};

typedef vector< list<aresta> > wgraph;
typedef list<aresta>::iterator arc;
#define forall_adj(uv,L) for (arc uv=(L).begin(); uv!=(L).end(); ++uv)
#define forall_ver(u,G) for (int u=0; u<int((G).size()); ++u)


// ALGORITMO DE KRUSKAL

class Kruskal {
private:
        list<aresta> L;
public:
Kruskal (wgraph& G) {
        int n = G.size();
        Particio P(n);
        priority_queue<aresta> CP;
        forall_ver(u,G) {
                forall_adj(uv,G[u]) {
                        CP.push(*uv);
        } }
```

```
        while (not CP.empty()) {
                aresta a = CP.top(); CP.pop();
                if (P[a.u]!=P[a.v]) {
                L.push_back(a);
                P.unir(a.u,a.v);
        } } }

list<aresta> arbre () {
        return L;
}
};
```

4.3. **Prim**

Este algoritmo de 1957 (un antecedente de 1930 se debe a Jarnik) hace crecer el conjunto T de aristas de tal modo que siempre es un árbol. Por este motivo, la función de selección escoge la arista de peso mínimo tal que un extremo de ella es uno de los vértices que ya están en T y el otro extremo es un vértice que no forma parte aún de T (es una arista que cruza el corte $(vértices(T), V - vértices(T))$ y es de peso mínimo). Como al final todos los vértices han de aparecer en T, el algoritmo comienza eligiendo cualquier vértice como punto de partida (será el único vértice que forme parte de T en ese instante) y ya empieza a aplicar la función de selección para hacer crecer T hasta conseguir que contenga, exactamente, $n-1$ aristas. Esta condición de finalización equivale a la que se utiliza en el algoritmo que viene a continuación en el que, en lugar de contar las aristas de *T*, se cuentan sus vértices que son los que están en `VISTOS`.

```
función PRIM(g es grafo) dev (T es conj_aristas)
{Pre: g=(V,E) es un grafo no dirigido, conexo y etiquetado con valores naturales}
            T:= conjunto_vacio;
            VISTOS:= añadir(conjunto_vacio, un vértice cualquiera de V);
            mientras (|VISTOS| < |V|) hacer
                 (u,v):= SELECCIONAR(g, V, VISTOS);
            /* devuelve una arista (u,v)  de peso mínimio tal que u∈ VISTOS y v∈ (V–VISTOS) */
                 T:= T ∪{(u,v)};
                 VISTOS := VISTOS ∪{v};
            fmientras
{Post: T es un árbol de expansión mínima de g}
            dev (T)
ffunción
```

Veamos el comportamiento de este algoritmo sobre el grafo que se ha utilizado en la sección 4.2. En este caso no es preciso ordenar previamente las aristas de G. La tabla describe el trabajo de cada una de las iteraciones del algoritmo:

ETAPA	ARISTA EXAMINADA	VISTOS
Incializ.	----	{1}
1	(1,2), valor 1	{1, 2}
2	(2,3), valor 2	{1, 2, 3}
3	(1,4), valor 4	{1, 2, 3, 4}
4	(4,5), valor 3	{1, 2, 3, 4, 5}
5	(4,7), valor 4	{1, 2, 3, 4, 5, 7}
6	(7,6), valor 3	{1, 2, 3, 4, 5, 6,7}
	peso(T) = 17	

Analizando el coste del algoritmo se ve que en cada iteración la función de selección precisa consultar todas las aristas que incidan en alguno de los vértices de `VISTOS` y que respeten el corte, eligiendo la de valor mínimo. Este proceso se ha de repetir $n-1$ veces, tantas como aristas ha de contener T. El algoritmo queda $O(n^3)$. Hay que buscar una forma de reducir este coste. La idea es mantener para cada vértice $v, v \in (V - \text{VISTOS})$, cual es el vértice más cercano de los que están en `VISTOS` y a qué distancia se encuentra. Si se mantiene esta información siempre coherente, la función de selección sólo necesita $O(n)$.

El siguiente algoritmo implementa esta aproximación. En el vector `VECINO` se guarda, para cada vértice $v \in (V - \text{VISTOS})$, cual es el más cercano de los que están en `VISTOS` y en el vector `COSTMIN` a qué distancia se encuentra. Cuando un vértice v pasa a `VISTOS`, entonces $\text{COSTMIN}[v] = -1$. Se supone que el grafo está implementado en la matriz de adyacencia $M[1\ldots n, 1\ldots n]$.

```
función PRIM_EFICIENTE(g es grafo) dev (T es conj_aristas)
```
{*Pre:* g=(V,E) es un grafo no dirigido, conexo y etiquetado con valores naturales, implementado en una matriz de adyacencia M[1...n,1...n]}

```
        T:= conjunto_vacio;
        VISTOS:= añadir( conjunto_vacio,{1});
        /* se comienza por un vértice cualquiera, por ejemplo el 1 */
        Para  i=2 hasta n hacer
              VECINO[i]:= 1;
              COSTMIN[i]:= M[i,1];
        fpara
```
/* el único vértice en VISTOS es 1. Por tanto a todos los demás vértices les pasa que el más cercano en VISTOS es el 1 y se encuentra a distancia M[i,1] (si no hay arista entonces M[i,1] contiene el valor infinito) */

```
        mientras (|VISTOS| < n) hacer
/* vamos a buscar la arista de peso mínimo que cruza el corte */
                min:= ∞;
                Para j=2 hasta n hacer
                    si (0 ≤ COSTMIN[j] < min) entonces
                            min:= COSTMIN[j];  k:= j;
                    fsi
                fpara
/* la arista ( k, VECINO[k] ) es la arista seleccionada */
                T:= T∪{(k, VECINO[k])};          /* la arista se añade a T */
                VISTOS:= VISTOS∪{k};             /* k pasa a VISTOS */
                COSTMIN[k]:= -1;
/* como VISTOS se ha modificado, hay que ajustar VECINO y COSTMIN para que contenga una información
coherente con la situación actual. Sólo afectará a  aquellos vértices que no pertenezcan a VISTOS y que son
adyacentes a k */
                para  j=2 hasta n hacer
                    si  (M[k,j] < COSTMIN[j]) entonces
                            COSTMIN[j]:= M[k,j];
                            VECINO[j]:= k;
                    fsi
                fpara
        fmientras
{Post: T es el MST de g}
        dev (T)
ffunción
```

Analizando el coste de esta nueva versión se tiene que el coste de cada iteración en cualquiera de los dos bucles es $\theta(n)$, y como se ejecutan $n-1$ iteraciones, el coste total es $\theta(n^2)$.

A continuación, una versión con coste $\theta(|E| \cdot log\ n)$ en C++ y que usa las mismas definiciones vistas en el apartado de Kruskal:

```
class Prim {
private:
        list<aresta> L;
public:

// ALGORITMO DE PRIM

Prim (wgraph& G) {
        int n = G.size();
        priority_queue<aresta> CP;
        vector<boolean> S(n,false);        // vector de vertices VISTOS
        forall_adj(uv,G[0]) CP.push(*uv);
```

```
        S[0] = true;
        int c = 1;
        while (c!=n) {
                aresta uv = CP.top(); CP.pop();
                if (not S[uv.v]) {
                        L.push_back(uv);
                        forall_adj(vw,G[uv.v]) {
                                if (not S[vw->v]) {
                                CP.push(*vw);
                        } }
                        S[uv.v] = true;
                        ++c;
} } }

list<aresta> arbre () {
        return L;
}
};
```

5. CAMINOS MINIMOS

Existe una colección de problemas denominados de caminos mínimos o *Shortest-paths problema.* Antes de presentarla veamos los conceptos básicos.

Sea $G =< V, E >$ un grafo dirigido y etiquetado con valores naturales. Se define el peso del camino p, $p = \{v_0, v_1, v_2, \ldots, v_k\}$, como la suma de los valores de las aristas que lo componen.

Formalmente:

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

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

$$\delta(u,v) = \begin{cases} MIN\{peso(p), \forall\, p \in camino(G,u,v)\}, & \text{si hay camino} \\ \infty, & \text{en otro caso} \end{cases}$$

Por tanto, el camino mínimo de u a v en G se define como cualquier camino p tal que $peso(p) = \delta(u,v)$.

La colección de problemas está compuesta por cuatro variantes:

1/ *Single source shortest paths problem*: Hay que encontrar el camino de peso mínimo entre un vértice fijado, *source*, y el resto de vértices del grafo. Este problema se resuelve eficientemente utilizando el algoritmo de Dijkstra.

2/ *Single destination shortest paths problem*: Hay que encontrar el camino de peso mínimo desde todos los vértices a uno fijado, *destination*. Se resuelve aplicando Dijkstra al grafo de partida pero cambiando el sentido de todas las aristas.

3/ *Single_pair shortest_paths problem*: Fijados dos vértices del grafo, *source* y *destination*, encontrar el camino de peso mínimo entre ellos. En el caso peor no hay un algoritmo mejor que el propio Dijkstra.

4/ *All_pairs shortest_paths problem*: Encontrar el camino de peso mínimo entre todo par de vértices, u y v, del grafo. Se resuelve aplicando n veces el algoritmo de Dijkstra o usando el algoritmo de Floyd que sigue el esquema de Programación Dinámica. Ambas posibilidades tienen igual coste.

5.1. **Dijkstra**

El algoritmo de Dijkstra, de 1959, resuelve el problema de encontrar el camino mínimo entre un vértice dado, el llamado inicial, y todos los restantes del grafo. Funciona de una forma semejante al algoritmo de Prim: supongamos que en `VISTOS` se tiene un conjunto de vértices tales que, para cada uno de ellos, se

conoce ya cual es el camino mínimo entre el vértice inicial y él. Para el resto de vértices u, $u \in V–VISTOS$, se guarda en `D[u]` el peso del camino mínimo desde el vértice inicial a u que sólo pasa por vértices de `VISTOS`.
La función de selección elige el vértice v, $v \in V–VISTOS$, con `D[v]` mínima que es lo mismo que hace Prim al seleccionar el vértice con el menor valor en `COSTMIN`.
Finalmente Dijkstra necesita actualizar convenientemente los valores de `D` cada vez que un nuevo vértice entra en `VISTOS` para mantener la corrección del contenido de `D`. Veamos el algoritmo.

```
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 y puede ser dirigido o no. En este caso es dirigido. Para facilitar la lectura del algoritmo se supone que el grafo está implementado en una matriz y que M[i,j] contiene el valor de la etiqueta 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 el peso mínimo del camino que va 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 el peso mínimo del camino desde v_ini a u}

```
        mientras (|VISTOS| < |V|) hacer
                u:= MINIMO(D, u∈V–VISTOS);
                /* se obtiene el vértice u∈V-VISTOS que tiene D mínima */
                VISTOS:= VISTOS ∪{u};
                /* Ajustar D */
                para cada v ∈ suc(g,u) tal que v ∈V–VISTOS hacer
                        /* aquí se ajusta D[v] para que se restablezca el invariante */
                fpara
        fmientras
```

{*Post:* ∀u: u∈V: D[u] contiene el peso del camino 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
```

El coste del algoritmo, si el grafo está implementado en una matriz, es $\theta(n^2)$ y se obtiene de la siguiente forma:

- El bucle que inicializa el vector `D` cuesta $\theta(n)$.

- El bucle principal efectúa $n-1$ iteraciones.
- El coste de cada iteración es $\theta(n)$ ya que la obtención del mínimo tiene coste lineal porque hay que recorrer `D` y el ajuste de `D` también es $\theta(n)$.

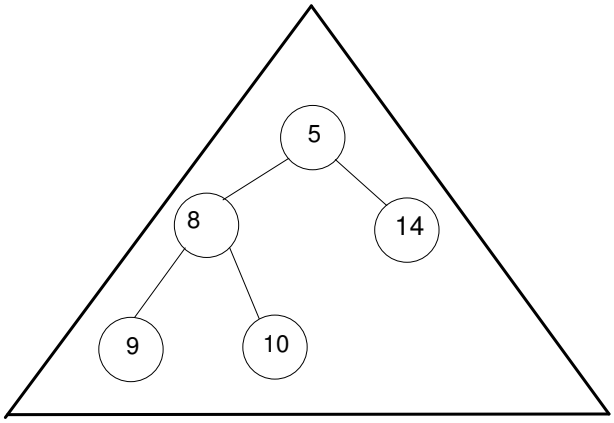
Ahora bien, si el grafo está implementado en listas de adyacencia y se utiliza una cola de prioridad implementada en un Heap para acelerar la elección del mínimo, entonces, el coste es $\theta((n+|E|)\cdot log\ n)$ y se obtiene de:

- La construcción del Heap es $\theta(n)$,
- Seleccionar el mínimo cuesta $\theta(1)$,
- Cada operación de ajuste que trabaja sobre la cola de prioridad (`insertar`, `sustituir`) requerirá $\theta(log\ n)$.
- En total se efectúan $\theta(|E|)$ operaciones de ajustar `D`.

La estructura de datos resultante podría presentar el siguiente aspecto:

D

	valor	pos en cola prioridad
1	8	2
i	5	1
j	9	4
k	14	3
	10	5



Cola de prioridad de mínimos implementada en un heap

valor	5	8	14	9	10			
pos en D	i	1	k	j	k+1			

HEAP

Algoritmo de Dijkstra empleando una cola de 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;
     C:=cola_prioridad_vacia;
     insertar(C, v_ini, D[v_ini]);

     mientras  ¬vacia(C) hacer
           u:= elemento_mínimo(C);
           eliminar_mínimo(C);
           para cada v∈suc(g,u) hacer
               si (D[v] ≥ D[u]+valor(g,u,v)) entonces
                    D[v]:= D[u]+valor(g,u,v);
                    si (esta(C,v)) entonces sustituir(C,v,D[v]);
                    sino insertar(C,v,D[v]);
                    fsi
               fsi
           fpara
       fmientras
     dev (D)
ffunción
```

Falta comprobar la corrección del criterio de selección, su optimalidad.

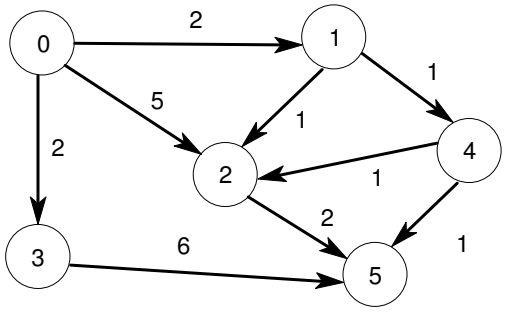
Demostración: Sea u un vértice tal que $u \in V - VISTOS$. Supongamos que $D[u]$ contiene información cierta, es decir, contiene el peso del camino mínimo entre el vértice inicial y u siguiendo por un camino que sólo pasa por vértices que pertenecen a $VISTOS$. Si u es el vértice con el menor valor en D, 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, supongamos que existe un camino mínimo con un peso inferior y aún no considerado desde el vértice inicial a u que pasa por algún v, obviamente $v \in V - VISTOS$. Si v se encuentra en el camino de peso mínimo desde el vértice inicial a u, necesariamente es por qué $D[v] < D[u]$, lo que contradice la elección de u por parte de la función de selección. De este modo se puede concluir que siempre que

D contenga información correcta, la función de selección elige un vértice con un valor de D ya definitivo y ninguno de los vértices que no están en $VISTOS$ lograrían que ese valor se redujera.

En el algoritmo propuesto se observa que una vez que se ha seleccionado un vértice se procede a `ajustar D`. Este proceso de ajuste consiste en lo siguiente: Para todos los vértices que todavía continúan en $V–VISTOS$, su valor de D contiene la distancia mínima al vértice inicial pero sin tener en cuenta que u acaba de entrar en $VISTOS$. Hay que actualizar ese valor y a los únicos vértices que les afecta es a los sucesores de u. Para cada vértice v, $v \in suc(G,u)$ y $v \in V–VISTOS$, hay que determinar qué es menor si el peso de llegar a v sin pasar por u, $D[v]$, o llegar a u y siguiendo la arista (u,v) llegar a v, $D[u] + valor(G,u,v)$. Hay que actualizar $D[v]$ convenientemente y guardar en $D[v]$ el menor valor de estos dos posibles.

Tomando como punto de partida el grafo de la siguiente figura, veamos como funciona el algoritmo.



Se elige como vértice inicial el vértice con etiqueta *0*. La tabla que viene a continuación muestra en qué orden van entrando los vértices en $VISTOS$ y cómo se va modificando el vector que contiene las distancias mínimas. Los valores en *cursiva* de la tabla corresponden a valores definitivos y, por tanto, contienen la distancia mínima entre el vértice *0* y el que indica la columna. Se asume que la distancia de un vértice a si mismo es cero.

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}

5.2. **Reconstrucción de caminos mínimos**

El algoritmo presentado calcula el peso de los caminos de peso mínimo pero no mantiene la suficiente información como para saber qué vértices los forman. Se puede modificar el algoritmo para almacenar la información necesaria y luego poder recuperar los vértices que integran los caminos de peso mínimo. Basta con anotar, cada vez que se modifica el valor de `D`, qué vértice que ha provocado esa modificación. Si el hecho de que v entre en `VISTOS` hace que `D[`u`]` se modifique, es debido a que el camino mínimo del vértice inicial a u tiene como última arista la (v,u). Esta es la información que hay que guardar para cada vértice. El vector `CAMINO` se emplea en el siguiente algoritmo con ese propósito.

```
función DIJKSTRA_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 );
        mientras (|VISTOS| < |V|) hacer
                u:= MINIMO(D, u∈V−VISTOS);
                VISTOS:= VISTOS∪{u};
                para cada v ∈ suc(g,u) tal que v ∈V−VISTOS hacer
                     si (D[v] > D[u] + valor(g,u,v)) entonces
                             D[v]:= D[u] + valor( g,u,v );
                             CAMINO[v]:= u;
                     fsi
                fpara;
        fmientras;
```

{*Post:* ∀u: u∈V: D[u] contiene el peso del camino 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. ∀u : u∈V: CAMINO[u] contiene el otro vértice de la última arista del camino mínimo de v_ini a u}

```
        dev (D, CAMINO)
ffunción
```

Para poder reconstruir los caminos se utiliza un procedimiento recursivo que recibe el vértice para el cual se quiere reconstruir el camino de peso mínimo, v, y el vector CAMINO. Una nueva llamada recursiva con el vértice CAMINO[v] devolverá el camino desde v_ini a este vértice y sólo hará falta añadirle la arista (CAMINO[v],v) para tener el camino de peso mínimo hasta v.

```
función RECONST(v, v_ini es vértice; CAMINO es vector[1..n] de vértices)
                                        dev (s es secuencia_aristas)
        si (v = v_ini) entonces s:= secuencia_vacia;
        sino     u:= CAMINO[v];
                 s:= RECONST(u, v_ini, CAMINO);
                 s:= concatenar(s, (u, v));
        fsi
```

{*Post:* s contiene las aristas del camino de peso mínimo desde v_ini a v}

```
        dev (s)
ffunción
```

Se presenta a continuación una versión en C++ que permite la reconstrucción de caminos y que usa una cola de prioridad. Destacar que no se sobrescriben las prioridades de un elemento que ya esté en la cola de prioridad si no que se añade el mismo elemento pero con una prioridad mayor (en este caso con un valor d[w] menor).

```
class Dijkstra {
private:
        vector<int> p;        // vector con los padres de cada vértice en el camino
                              // de peso mínimo
        vector<double> d;     // vector de las distancias desde u al resto de vértices
public:

// Algoritmo de DIJKSTRA: Calcula los pesos de los caminos de peso mínimo desde
// u al resto  de vértices del grafo G.

Dijkstra (wgraph& G, int u) {
// inicialización
        int n = G.size();
        vector<boolean> S(n,false);        // vector de 'vistos'
        p = vector<int>(n,-1);
        d = vector<double>(n,infinit);
        d[u] = 0;
        priority_queue<aresta> CP;
        CP.push(aresta(-1,u,0));
                // en la cola de prioridad se guardan tripletas (x,y,z)
                // x = padre de y en el camino de peso mínimo actual
```

```
                // y = vértice que añadimos a la cola.
                // z = d[y] el peso del camino mínimo que llega a y
                // z es la prioridad del elemento en la cola de prioridad
// bucle principal
        while (not CP.empty()) {
                aresta a = CP.top(); CP.pop();
                int v = a.v;
                if (not S[v]) {
                        S[v] = true;
                        forall_adj(vw,G[v]) {
                                int w = vw->v;
                                if (not S[w]) {
                                        if (d[w] > d[v]+vw->p) {
                                                d[w] = d[v]+vw->p;
                                                p[w] = v;
                                                CP.push(aresta(v,w,d[w]));
} } } } } }

double distancia (int v) {
        return d[v];
}
vector<double> distancies () {
        return d;
}
int pare (int v) {
        return p[v];
}
vector<int> pares () {
        return p;
}
};
```

6. CÓDIGOS DE HUFFMAN

La codificación de Huffman es una técnica para la compresión de datos ampliamente usada y muy efectiva. El siguiente ejemplo ilustra esta técnica de codificación.
Disponemos de un fichero con 100000 caracteres. Se sabe que aparecen 6 caracteres diferentes y la frecuencia de aparición de cada uno de ellos es:

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

El problema es cómo codificar los caracteres para reducir el espacio que ocupan y utilizando un código binario.
La primera alternativa es utilizar un código de longitud fija. En este caso para distinguir 6 caracteres se necesitan 3 bits. El espacio necesario para almacenar los 100000 caracteres, sabiendo que cada uno de ellos requiere 3 bits, es de 300000 bits. Una codificación posible sería:

Caracteres	*a*	*b*	*c*	*d*	*e*	*f*
Long. Fija	000	001	010	011	100	101

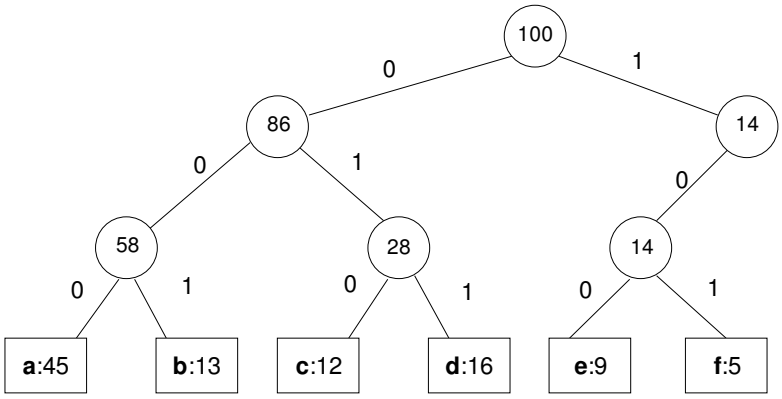
Pero podemos utilizar un código de longitud variable en el que los caracteres más frecuentes tienen el código más corto. Al hacer la codificación hay que tener en cuenta que ningún código ha de ser prefijo de otro. En caso contrario la descodificación no será única. El espacio necesario es ahora de 224000 bits.

Caracteres	*a*	*b*	*c*	*d*	*e*	*f*
Long.Variable	0	101	100	111	1101	1100

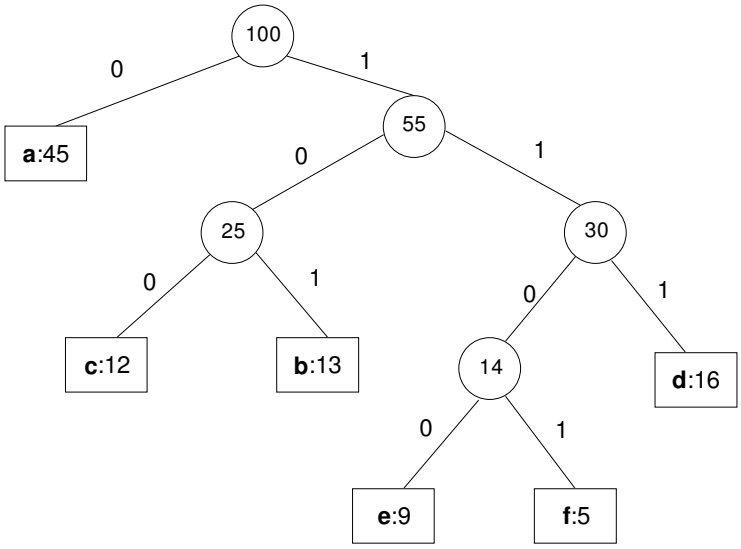
Esta técnica de codificación se denomina *código prefijo*. La ventaja es que para codificar basta con concatenar el código de cada uno de los caracteres y la descodificación también es simple y no se produce ninguna ambigüedad ya que ningún código es prefijo de otro código.

Codificación: $aabacd \equiv 0{\cdot}0{\cdot}101{\cdot}0{\cdot}100{\cdot}111 \equiv 001010100111$
Descodificación: $101011101111011100 \equiv badadcf$ y es la única posibilidad

La forma habitual de representar el código, tanto el de longitud variable como el de fija, es usando un árbol binario de forma que el proceso de descodificación se simplifica. En este árbol, las hojas son los caracteres y 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.



Árbol binario de codificación de longitud fija



Árbol binario de codificación de longitud variable

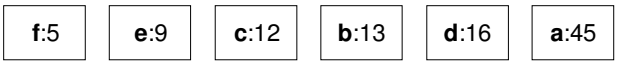
Sea T el árbol binario que corresponde a una codificación prefijo entonces el número de bits necesarios para codificar el fichero, $B(T)$, se calcula de la siguiente manera:
Para cada carácter c diferente del alfabeto C que aparece en el fichero,
- sea $f(c)$ la frecuencia de c en la entrada,
- sea $d_T(c)$ la profundidad de la hoja c en el árbol T, entonces el número de bits requeridos es:

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

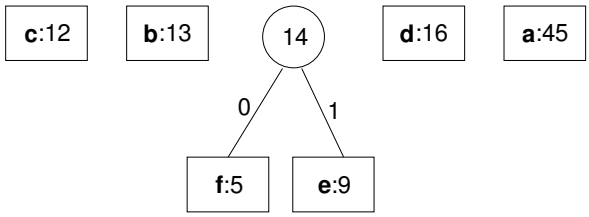
Huffman propuso un algoritmo voraz que obtiene una codificación prefijo óptima. Para ello construye un árbol binario de códigos de longitud variable de manera ascendente de modo que $[MIN]\ B(T)$.

El algoritmo funciona de la siguiente manera: Parte de una secuencia inicial en la que los caracteres a codificar están colocados en orden creciente de frecuencia. Esta secuencia inicial se va transformando, a base de fusiones, hasta llegar a una secuencia con un único elemento que es el árbol de codificación óptimo:

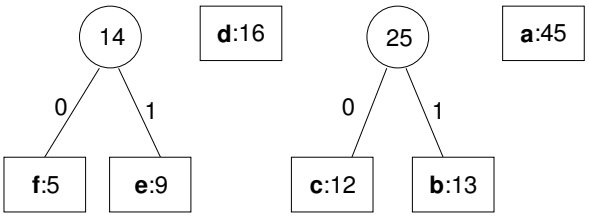
ETAPA 1



ETAPA 2



ETAPA 3



Y así sucesivamente hasta llegar a obtener el árbol de codificación óptimo.
Para implementar el algoritmo de los códigos de Huffman, se usa una cola de prioridad. La cola inicialmente contiene los elementos de C y acaba conteniendo un único elemento que es el árbol de fusión construido. La frecuencia es la prioridad de los elementos de la cola y coincide con la frecuencia de la raíz del árbol. El resultado de fusionar dos elementos/árboles es un árbol cuya frecuencia es la suma de frecuencias de los dos fusionados. Su coste es $\theta(n\ log\ n)$ con $n = |C|$.

```
función COD_HUF(C es conj<carácter,frecuencia_aparición>) dev (A es arbin)
```

{*Pre*: C está bien construido y no es vacío}

```
    var Q es cola_prioridad;
    n:= |C|; Q:= insertar_todos(C);
    /* la cola contiene todos los elementos */
```

```
    para i=1 hasta n-1 hacer
        z:= arbol_vacio;
        /* elección de los candidatos */
        x:= primero(Q);  Q:= avanzar(Q);
        y:= primero(Q);  Q:= avanzar(Q);
        /* construcción del nuevo árbol de fusión */
        z.frecuencia:= x.frecuencia + y.frecuencia;
        z.hijo_izquierdo:= x; z.hijo_derecho:= y;
        Q:= insertar(Q, z);
    fpara
```

{*Post* : Q contiene un único elemento que es un árbol de codificación de prefijo óptimo }

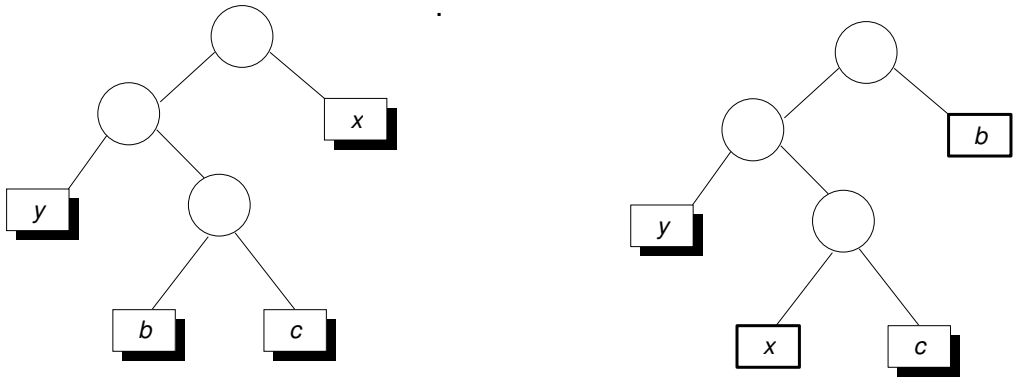
```
    dev (primero(Q))
ffunción
```

Algoritmo de codificación prefija de Huffman

Demostración: Sea T un árbol binario de codificación prefija 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. El caso interesante para la demostración se produce cuando $(b, c) \neq (x, y)$.

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. El árbol que genera el algoritmo voraz cumple exactamente esa condición.

Podemos suponer que $f(b) \leq f(c)$ y que $f(x) \leq f(y)$. También se puede deducir que $f(x) \leq f(b)$ y $f(y) \leq f(c)$. Construimos un nuevo árbol, T', en el que se intercambia la posición que ocupan en T las hojas b y x.



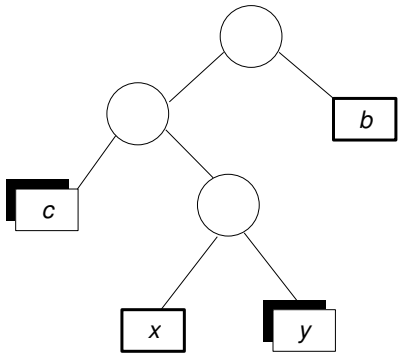
árbol T *árbol T'*

El número de bits para el nuevo árbol de codificación T' lo denotamos por $B(T')$. Tenemos entonces que:

$$\begin{aligned} B(T) - B(T') &= \sum_{c \in C} f(c) \cdot d_T(c) - \sum_{c \in C} f(c) \cdot d_{T'}(c) \\ &= f(x) \cdot d_T(x) + f(b) \cdot d_T(b) - f(x) \cdot d_{T'}(x) - f(b) \cdot d_{T'}(b) \\ &= f(x) \cdot d_T(x) + f(b) \cdot d_T(b) - f(x) \cdot d_T(b) - f(b) \cdot d_T(x) \\ &= \big(f(b) - f(x)\big) \cdot (d_T(b) - d_T(x)) \geq 0 \end{aligned}$$

De modo que $B(T) \geq B(T')$ pero dado que T es óptimo, mínimo, T' no puede ser menor y $B(T) = B(T')$ por lo que T' también es óptimo.

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



árbol T''

Con este intercambio tampoco se incrementa el coste y $B(T') - B(T'') \geq 0$. Por tanto, $B(T'') = B(T)$ y como T es óptimo, entonces T'' también lo es.
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 en T. 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 frecuencia la suma de los dos anteriores y repite el proceso de seleccionar los dos elementos con frecuencia más baja ahora para un alfabeto con un elemento menos.

Para acabar, una posible codificación en C++.

```
class Huffman {
private:
        struct node {
                node *fe, *fd, *pare;
                char c;
                double f;

                node (node* fe, node* fd, char c, double f) :
                        fe(fe), fd(fd), pare(null), c(c), f(f) {
                                if (fe) fe->pare = this;
                                if (fd) fd->pare = this;
                        }

                ~node () { delete fe; delete fd; }
        };

        node* arrel;
        map<char,node*> fulles;

        struct comparador {
                bool operator() (node* p, node* q) {
                return p->f > q->f;
                }
        };
public:
        ~Huffman () {
                delete arrel;
        }

        Huffman (map<char,double>& F) {

                priority_queue<node*,vector<node*>,comparador> CP;

                foreach(it,F) {
                        node* p = new node(null,null,it->first,it->second);
                        CP.push(p);
                        fulles[it->first] = p;
                }

                while (CP.size()!=1) {
                        node* p = CP.top(); CP.pop();
                        node* q = CP.top(); CP.pop();
                        CP.push(new node(p,q,' ',p->f+q->f));
                }
                arrel = CP.top();
        }

        string decodifica (string s) {
                string r;
                node* p = arrel;
                unsigned i = 0;
                while (i<=s.size()) {
                        if (p->c) {
                                r += p->c;
                                p = arrel;
                        } else {
                                p = s[i++]=='0' ? p->fe : p->fd;
                } }
                return r;
        }
```

```
        string codifica (string s) {
                string r;
                for (unsigned i=0; i<s.size(); ++i)
                        r += codifica(fulles[s[i]]);
                        return r;
        }
private:
        string codifica (node* p) {
                if (p->pare==null) {
                        return "";
                } else if (p->pare->fe == p) {
                        return codifica(p->pare) + '0';
                } else {
                        return codifica(p->pare) + '1';
        } }
};
```

NOTA:
Todos los algoritmos codificados en C++ que aparecen en el texto pueden encontrarse en la página web de ADA, http://www.lsi.upc.edu/~ada, y sus autores son J.Petit, S.Roura y A.Atserias