FACULTAD DE INFORMATICA

Ingeniería Informática

INTRODUCCIÓN A LOS ESQUEMAS ALGORÍTMICOS

Apuntes y Colección de Problemas

Mª Teresa Abad Soriano
Dept. L.S.I. - U.P.C.
Report LSI-97-6-T

INDICE

RECONOCIMIENTOS

1. DIVIDE Y VENCERAS : Divide and Conquer
- **1.1.** PRINCIPIOS
- **1.2.** MULTIPLICACION DE ENTEROS GRANDES : Algoritmo de Karatsuba y Ofman
- **1.3.** PRODUCTO DE MATRICES : Algoritmo de Strassen
- **1.4.** UN ALGORITMO DE ORDENACION : Mergesort
- **1.5.** EL PROBLEMA DE LA SELECCION

2. GRAFOS
- **2.1.** ESPECIFICACION ALGEBRAICA Y DEFINICIONES
  - 2.1.1. ESPECIFICACION
  - 2.1.2. DEFINICIONES
    - 2.1.2.1. Adyacencias
    - 2.1.2.2. Caminos
    - 2.1.2.3. Conectividad
    - 2.1.2.4. Algunos grafos particulares
- **2.2.** IMPLEMENTACIONES. ANALISIS DEL COSTE DE LAS OPERACIONES
  - 2.2.1. MATRICES DE ADYACENCIA
  - 2.2.2. LISTAS Y MULTILISTAS DE ADYACENCIA
  - 2.2.3. EL PROBLEMA DE LA CELEBRIDAD
- **2.3.** ALGORITMOS SOBRE GRAFOS
  - 2.3.1. RECORRIDO EN PROFUNDIDAD
    - 2.3.1.1. Conectividad
    - 2.3.1.2. Numerar vértices
    - 2.3.1.3. Arbol asociado al Recorrido en profundidad
    - 2.3.1.4. Test de ciclicidad
    - 2.3.1.5. Un TAD útil : MFSets
  - 2.3.2. RECORRIDO EN ANCHURA
  - 2.3.3. ORDENACION TOPOLOGICA

3. ALGORITMOS VORACES : Greedy
- **3.1.** CARACTERIZACION Y ESQUEMA
- **3.2.** PROBLEMAS SIMPLES
  - 3.2.1. MONEDAS : EL PROBLEMA DEL CAMBIO
  - 3.2.2. MINIMIZAR TIEMPOS DE ESPERA
  - 3.2.3. MAXIMIZAR NUMERO DE TAREAS EN EXCLUSION MUTUA
- **3.3.** MOCHILA

3.4. ARBOLES DE EXPANSION MINIMA
3.4.1. KRUSKAL
3.4.2. PRIM
3.5. CAMINOS MINIMOS
3.5.1. DIJKSTRA
3.5.2. RECONSTRUCCION DE CAMINOS

4. ESQUEMA DE VUELTA ATRAS : Backtracking
4.1. CARACTERIZACION
4.2. TERMINOLOGÍA Y ESQUEMAS
4.2.1. UNA SOLUCION
4.2.2. TODAS LAS SOLUCIONES
4.2.3. LA MEJOR SOLUCION
4.2.4. MOCHILA ENTERA
4.3. MARCAJE
4.4. PODA BASADA EN EL COSTE DE LA MEJOR SOLUCION EN CURSO

5. RAMIFICACION Y PODA : Branch & Bound
5.1. CARACTERIZACION Y DEFINICIONES
5.2. EL ESQUEMA Y SU EFICIENCIA
5.2.1. EL ESQUEMA ALGORITMICO
5.2.2. CONSIDERACIONES SOBRE LA EFICIENCIA
5.3. UN EJEMPLO : MOCHILA ENTERA
5.4. OTRAS CUESTIONES
5.4.1. COMPARACION CON OTRAS ESTRATEGIAS DE BUSQUEDA
5.4.2. ESTRATEGIAS DE BUSQUEDA *BEST-FIRST*
5.4.2.1. La función de estimación en un problema de Minimización
5.4.2.2. Terminación y la Admisibilidad de A*
5.4.2.3. La restricción monótona

REFERENCIAS

APENDICE : Colección de Problemas
I. DIVIDE Y VENCERAS
II. GRAFOS
III. VORACES
IV. VUELTA ATRAS Y RAMIFICACION Y PODA
V. MISCELANEA
VI. TEST

RECONOCIMIENTOS

Estos apuntes están específicamente realizados para ser utilizados por los alumnos de la asignatura de Introducción a los Esquemas Algorítmicos. Es una asignatura optativa de cuarto nivel de la Ingeniería de Informática de la F.I.B. Sin embargo, creo que también pueden ser de alguna utilidad para aquellas personas que quieran conocer, o simplemente 'refrescar', los esquemas algorítmicos que aquí se presentan.

En su confección he utilizado como punto de partida el material de dos profesores que me han precedido y ayudado : Ricardo Peña y Albert Llamosí. Sus apuntes de la asignatura de Tecnología de la Programación de la antigua Licenciatura de Informática de la F.I.B. han sido básicos. A estos apuntes les falta, y con toda la intención, un capítulo inicial dedicado al análisis de la eficiencia de algoritmos. Pero ese trabajo ya está hecho por José Luis Balcázar cubriendo perfectamente el nivel que la asignatura requiere. Su trabajo se encuentra en [Bal 93].
Temas como Programación Dinámica, MiniMax, etc. han quedado pendientes y espero que aparezcan en alguna edición futura.

La filiación de la colección de problemas es muy diversa. Todas las personas que en algún momento han impartido las asignaturas de Tecnología de la Programación o Estructuras de la Información, ambas de la antigua Licenciatura de Informática de la F.I.B., han aportado algo. La colección inicial ha ido creciendo con los enunciados de exámen que he ido produciendo para la asignatura de Introducción a los Esquemas Algorítmicos.

Este material docente es el resultado de la fusión y revisión de publicaciones previas a lo largo de los cursos 94-95 y 95-96. Estas publicaciones ya han ido pasando el filtro de los alumnos y a ellos he de agradecer la detección de errores y de explicaciones poco claras. De todos modos, seguro que todavía hay errores y su notificación será siempre bien recibida.
También tengo que agradecer a mis amigos y ex-compañeros de despacho Rosa María Jiménez y David Guijarro su paciencia (mucha, mucha, mucha) y sus innumerables aportaciones técnicas.

María Teresa Abad.
Febrero, 1997

1. DIVIDE Y VENCERAS : Divide & Conquer

1.1. PRINCIPIOS

El esquema de Divide y Vencerás es una aplicación directa de las técnicas de diseño recursivo de algoritmos.

Hay dos rasgos fundamentales que caracterizan los problemas que son resolubles aplicando el esquema de Divide y Vencerás. El primero de ellos es que es necesario que el problema admita una formulación recursiva. Hay que poder resolver el problema inicial a base de combinar los resultados obtenidos en la resolución de un número reducido de subproblemas. Estos subproblemas son del mismo tipo que el problema inicial pero han de trabajar con datos de tamaño estrictamente menor. Y el segundo rasgo es que el tamaño de los datos que manipulan los subproblemas ha de ser lo más parecido posible y debe decrecer en progresión geométrica. Si n denota el tamaño del problema inicial entonces n/c, siendo $c>0$ una constante natural, denota el tamaño de los datos que recibe cada uno de los subproblemas en que se descompone.

Estas dos condiciones están caracterizando un algoritmo, generalmente recursivo múltiple, en el que se realizan operaciones para *fragmentar* el tamaño de los datos y para *combinar* los resultados de los diferentes subproblemas resueltos.

Además, es conveniente que el problema satisfaga una serie de condiciones adicionales para que sea rentable, en términos de eficiencia, aplicar esta estrategia de resolución. Algunas de ellas se ennumeran a continuación :

1/ En la formulación recursiva nunca se resuelve el mismo subproblema más de una vez.

2/ Las operaciones de fragmentación del problema inicial en subproblemas y las de combinación de los resultados de esos subproblemas han de ser eficientes, es decir, han de costar poco.

3/ El tamaño de los subproblemas ha de ser lo más parecido posible.

4/ Hay que evitar generar nuevas llamadas cuando el tamaño de los datos que recibe el subproblema es suficientemente pequeño. En [BB 90] se discute cómo determinar el tamaño umbral a partir del cual no conviene seguir utilizando el algoritmo recursivo y es mejor utilizar el algoritmo del caso directo.

Ahora ya podemos proponer un esquema algorítmico genérico para el Divide y Vencerás como el que viene a continuación :

función DIVIDE_VENCERAS (n es T1) dev (y es T2)
{ *Pre :* Q (n) }
  [caso_directo(n) ---> y := solución_directa(n)
  [] caso_recursivo(n) --->
    $< n_1, n_2, ..., n_k >$:= FRAGMENTAR(n)
    /* se fragmenta n en k trozos de tamaño n/c cada uno */
    r_1 := **DIVIDE_VENCERAS**(n_1); r_2 := **DIVIDE_VENCERAS**(n_2);
    … ; r_k := **DIVIDE_VENCERAS**(n_k);
    y := COMBINAR($n_1, n_2, \ldots, n_k, r_1, r_2, \ldots, r_k$);
  ]
{ *Post :* y = SOLUCION (n) }
  dev (y)
ffunción

Existen casos particulares de aplicación del esquema que se convierten en degeneraciones del mismo. Por ejemplo cuando alguna de las llamadas recursivas sólo se produce dependiendo de los resultados de llamadas recursivas previas, o cuando el problema se fragmenta en más de un fragmento pero sólo se produce una llamada recursiva. Esto último sucede en el algoritmo de *Búsqueda dicotómica* sobre un vector ordenado en el que se obtienen dos fragmentos pero sólo se efectúa la búsqueda recursiva en uno de ellos.

El coste del algoritmo genérico DIVIDE_VENCERAS se puede calcular utilizando el segundo teorema de reducción siempre que se haya conseguido que los subproblemas reciban datos de tamaño similar. Entonces, la expresión del caso recursivo queda de la siguiente forma:

$$T_2(n) = k \cdot T_2(n/c) + \text{coste} (\mathit{MAX}(\text{FRAGMENTAR, COMBINAR}))$$

Conviene destacar que la utilización de este esquema NO GARANTIZA NADA acerca de la eficiencia del algoritmo obtenido. Puede suceder que el coste empeore, se mantenga o mejore respecto al de un algoritmo, probablemente iterativo, que ya se conocía de antemano. En los apartados siguientes veremos las tres situaciones posibles.

El algoritmo de Búsqueda dicotómica y el de ordenación rápida, *Quicksort* (Hoare, 1962), son dos aplicaciones clásicas del esquema de Divide y Vencerás (se supone que ambos algoritmos son bien conocidos por el lector).
La recurrencia de la Búsqueda dicotómica es $T_{bd}(n) = T_{bd}(n/2) + 1$, y su coste es $\theta(\log n)$. Quicksort presenta un comportamiento menos homogéneo. En él el tamaño de los

fragmentos a ordenar depende de la calidad del pivote. Un buen pivote, por ejemplo la mediana, construye dos fragmentos de tamaño *n/2*, pero un mal pivote puede producir un fragmento de tamaño *1* y otro de tamaño *n-1*. En el caso de utilizar un buen pivote la recurrencia es $T_b(n) = 2 \cdot T_b(n/2) + \theta(n)$ lo que da un coste de $\theta(n \cdot \log n)$. En el otro caso la recurrencia es absolutamente diferente, $T_m(n) = T_m(n-1) + \theta(n)$, que tiene un coste de $\theta(n^2)$. Estos son los costes en el caso mejor y el caso peor, respectivamente, del algoritmo del Quicksort. A pesar del coste en el caso peor, su coste en el caso medio es también $\theta(n \cdot \log n)$ lo que le convierte en un algoritmo de ordenación eficiente.

1.2. MULTIPLICACION DE ENTEROS GRANDES : Karatsuba y Ofman

El algoritmo de Karatsuba y Ofman es una aplicación directa del esquema de Divide y Vencerás. El problema que resuelve es el de obtener el producto de dos enteros de gran tamaño. Sin pérdida de generalidad, podemos suponer que el tamaño de los enteros a multiplicar es *n*, es decir, cada entero ocupa *n* posiciones que pueden ser bits o dígitos, y que *n* es potencia de dos.

Existe un algoritmo muy conocido y que aprendimos en el colegio que tiene un coste de $\theta(n^2)$. Se puede intentar resolver el mismo problema pero aplicando Divide y Vencerás. El tamaño de los enteros a multiplicar se puede reducir a base de dividir entre dos su longitud. De esta forma los dos enteros iniciales de tamaño *n* se convierten en cuatro enteros de tamaño *n/2*. Una vez obtenidos los fragmentos, se calculan recursivamente los nuevos productos entre ellos y luego se combinan adecuadamente sus resultados para obtener el producto de los dos enteros iniciales. Concretando, supongamos que se quiere calcular $X \cdot Y$, y que *A* contiene los *n/2* dígitos más significativos de *X*, y *B* contiene los *n/2* dígitos menos significativos de *X* (*C* y *D* representan lo mismo pero respecto de *Y*), es decir :

$$X = A \cdot b^{n/2} + B, \qquad Y = C \cdot b^{n/2} + D \quad (\ b = \text{base en que están expresados } X \text{ e } Y\ )$$

Fácilmente puede comprobarse que

$$X \cdot Y = A \cdot C \cdot b^n + (\ A \cdot D + B \cdot C\ ) \cdot b^{n/2} + B \cdot D,$$

Se tiene que para calcular $X \cdot Y$ hay que efectuar 4 productos entre enteros de tamaño *n/2*. La recurrencia que se obtiene es $T(n) = 4 \cdot T(n/2) + \theta(n)$ y su coste asociado es $\theta(n^2)$.

Esta solución, pese a utilizar la técnica de Divide y Vencerás, no consigue mejorar el coste de la solución escolar, sólo lo iguala. En el algoritmo propuesto por Karatsuba y Ofman se aplica exactamente la misma fragmentación del problema que acabamos de utilizar, pero se reduce el número de nuevos productos a calcular. Se tiene que :

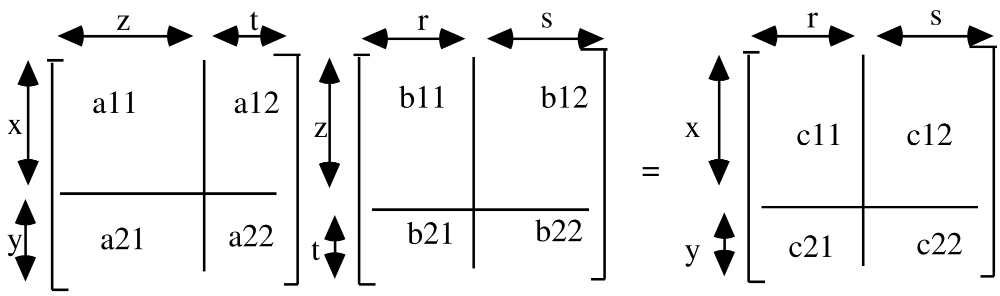
$$X \cdot Y = A \cdot C \cdot b^n + (\ (A-B) \cdot (D-C) + A \cdot C + B \cdot D\ ) \cdot b^{n/2} + B \cdot D$$

De este modo se calculan tres nuevos productos entre enteros de tamaño *n/2*. La nueva recurrencia que se obtiene es $T(n) = 3 \cdot T(n/2) + \theta(n)$ y su coste asociado es $\theta(n^{1.57})$. El tamaño umbral es $n \leq 500$, es decir, si hay que multiplicar enteros de longitud superior a 500 bits ó dígitos conviene utilizar el algoritmo de Karatsuba y Ofman, pero si los enteros son más pequeños es más eficiente utilizar el algoritmo del colegio. En [BB 90] y [AHU 83] pueden encontrarse diferentes versiones de este algoritmo.

En [BM93] se presenta una solución distinta para este problema. Aplicando una técnica de Divide y Vencerás, similar a la explicada aquí, se obtiene una solución que parece tener un coste ¡ lineal !, pero después de calcular el coste real se llega a que la solución tiene un coste cuadrático. Resulta muy instructivo desarrollar todo el cálculo del coste.

1.3. PRODUCTO DE MATRICES : Algoritmo de Strassen

El problema que ahora se plantea es el de calcular el producto de matrices de gran tamaño. Si se supone que las matrices son cuadradas y de tamaño *n* (*n* filas y *n* columnas) el algoritmo utilizado habitualmente y aprendido en el bachillerato tiene coste $\theta(n^3)$.
Sin embargo, una aplicación directa de la técnica de Divide y Vencerás produce la siguiente solución : fragmentar cada matriz en cuatro submatrices cuadradas y después efectuar ocho productos con esas submatrices para obtener el producto de las dos matrices iniciales. Como el número de llamadas recursivas es ocho, la dimensión de cada submatriz es la mitad respecto de la inicial, es decir *n/2*, y sumar matrices cuesta $\theta(n^2)$, se puede comprobar que el coste de este algoritmo es también $\theta(n^3)$ y que, por tanto, no se ha conseguido mejorar el coste del algoritmo escolar.
La siguiente figura ilustra esta solución Divide y Vencerás :



c11 = a11·b11 + a12·b21
c12 = a11·b12 + a12·b22
c21 = a21·b11 + a22·b21
c22 = a21·b12 + a22·b22

El algoritmo de Strassen, al igual que el de Karatsuba y Ofman, intenta mejorar el coste del algoritmo a base de reducir el número de llamadas recursivas (nuevos productos a calcular) y consigue siete en lugar de los ocho de la solución anterior. A continuación se muestra la solución de Strassen en la que *c11*, *c12*, *c21* y *c22* son las cuatro submatrices que forman la matriz que contiene el resultado del producto final y P, Q, ..., V son los nuevos siete productos que hay que calcular a partir de la fragmentación de las dos matrices iniciales en ocho submatrices cuadradas.

c11 = P + S - T + V, *c12* = R + T, *c21* = Q + S y *c22* = P+ R - Q + U

donde P = (a11 + a22)·(b11 + b22)
Q = (a21 + a22)·b11
R = a11·(b12 - b22)
S = a22·(b21 - b11)
T = (a11 + a12)·b22
U = (a21 - a11)·(b11 + b12)
V = (a12 - a22)·(b21 + b22)

La recurrencia que se obtiene es $T(n) = 7 \cdot T(n/2) + \theta(n^2)$ y su coste asociado es $\theta(n^{2.81})$. Cuando $n \geq 120$ esta solución es mejor que la clásica. El algoritmo puede verse en [HS 78].

1.4. UN ALGORITMO DE ORDENACION : Mergesort

Otro algoritmo interesante y sumamente sencillo, que utiliza la técnica que aquí se presenta, es el algoritmo de ordenación por fusión, *Mergesort* (Von Neumann, 1945). Supongamos que se ha de ordenar una secuencia de *n* elementos. La solución Divide y Vencerás aplicada consiste en fragmentar la secuencia de entrada en dos subsecuencias de tamaño similar, ordenar recursivamente cada una de ellas y, finalmente, fusionarlas del modo adecuado para generar una sola secuencia en la que aparezcan todos los elementos ordenados.
En el algoritmo del Mergesort que se presenta a continuación, y con el fin de facilitar la escritura del mismo, se supone que la secuencia a ordenar es de naturales y está almacenada en el vector *a*.

```
función MS ( a es vector[1..n]; i, j es nat ) dev ( a es vector[1..n] )
{ Pre :  (1≤ i≤j≤n ) ∧ (a=A) }
            [  i = j ---> seguir
            [] i < j ---> q := ⌊( i + j ) div 2⌋
                          a := MS( a, i, q )
                          a := MS( a, q+1, j )
                          a := FUSION( a, i, q, j )
            ]
{ Post : es_permutación(a,A) ∧ ordenado(a,i,j) }
            dev ( a )
ffunción
```

La especificación de la función FUSION, que corresponde a la función genérica COMBINAR del esquema, queda como sigue:

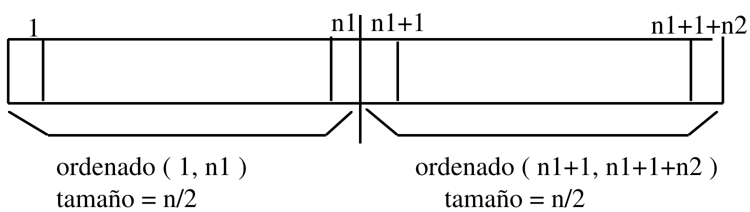
```
función FUSION ( a es vector; i, q, j es nat ) dev ( a es vector )
{ Pre : (a=A) ∧ ordenado(a,i,q) ∧ ordenado(a,q+1,j) }
{ Post : es_permutación(a,A) ∧ ordenado(a,i,j) }
```

La implementación tradicional de la función FUSION requiere un coste $\theta(n)$ y consiste en procesar secuencialmente las dos subsecuencias ordenadas de la entrada. A cada paso se decide de qué subsecuencia de la entrada se elige el primer elemento para que vaya a la secuencia de salida, al lugar que ocupará definitivamente. Usando una fusión de este tipo el Mergesort tiene un coste de $\theta(n \cdot \log n)$ que se obtiene al resolver la recurrencia $T(n)= 2 \cdot T(n/2) + \theta(n)$.

¿ Por qué no intentar aplicar de nuevo la técnica de Divide y Vencerás pero ahora para resolver el problema de la fusión ?. En la precondición de FUSION se indica que recibe dos secuencias ordenadas de tamaño *n/2*, y en su Postcondición se dice que ha de producir otra secuencia, también ordenada, de tamaño *n*. El nuevo algoritmo de fusión, que llamaremos FUSION_DC, parte cada una de las dos secuencias de entrada en dos nuevas mitades. Así se consigue que cada mitad tenga tamaño *n/4*. Luego se aplica FUSION_DC tres veces sobre las nuevas cuatro subsecuencias de entrada para obtener la de salida ordenada. El siguiente algoritmo muestra esta solución con todos los detalles.

<u>función</u> FUSION_DC (a <u>es</u> vector; i, q, r, j <u>es</u> nat) <u>dev</u> (b <u>es</u> vector)
{ *Pre :* $(a=A) \wedge$ ordenado(a,i,q) $\wedge$ ordenado(a,r,j) $\wedge$ tamaño(a,i,q)=tamaño(a,r,j) $\wedge$ $(i \le q \le r \le j)$ }
[$i = q \wedge r=j$ ---> [$a(i) \le a(j)$ ---> b:= a
[] $a(i) > a(j)$ ---> b:= int(a, i, j)
]
[] $i<q \vee r<j$ ---> /* cálculo de las nuevas zonas de partición */
k := (i + q) <u>div</u> 2; f := (r + j) <u>div</u> 2;
/* estas son las 4 zonas que hay que explorar : [i ... k] [k+1 ... q] () [r ... f] [f+1 ... j], cada zona tiene tamaño n/4 y está ordenada */
b1 := **FUSION_DC**(a, i, k, r, f)
/* se han fusionado las partes izquierdas de los intervalos: [i ... k] con [r ... f] */
{ n1 = (k-i+1) + (f-r+1) y ordenado(b1, 1, n1) }

b2 := **FUSION_DC**(a, k+1,q, f+1, j)
/* se han fusionado las partes derechas de los intervalos: [k+1 ... q] con [f+1 ... j] */
{ n2 = (q- k + j - f) y ordenado (b2, 1, n2) }
b:= CONCATENAR (b1, b2)
/* se ha construido un nuevo vector b simplemente concatenando b1 y b2. El nuevo vector b satisface :



*/

/* Notar que los elementos de b que ocupan las posiciones desde 1 a n1/2 y los que ocupan las posiciones desde (n1+1+n2)/2 hasta n1+1+n2 ya están definitivamente bien colocados y, por tanto, sólo hace falta ordenar el resto de los elementos. Como se repiten las condiciones de entrada del algoritmo, 2 zonas ordenadas que hay que fusionar, se procede exactamente de la misma forma : generando 4 zonas pero ahora sólo hace falta fusionar las dos centrales */

```
                    ii := 1;   qq := n1;   rr:= n1+1;   jj := n1+1+n2;
                    kk := ( ii + qq ) div 2;     ff := ( rr + jj ) div 2;
                    b := FUSION_DC( b, kk+1, qq, rr, ff )
/* se han fusionado las partes centrales de los intervalos: [kk+1 ... qq]  con [ rr ... ff ] */
                    { ordenado(b, 1, jj) }
          ]
{ Post : es_permutación(b, A[i..q]&A[r..j] ) ∧ ordenado(b,1,jj) ∧ jj=(q-i+1)+(j-r+1) }
          dev  ( b )
ffunción
```

La llamada inicial a FUSION_DC se hace desde MS con los argumentos (a, i, q, q+1, j). Argumentar la corrección de esta solución resulta un ejercicio interesante.

El siguiente ejemplo ilustra el funcionamiento del algoritmo propuesto. Se tienen las dos siguientes secuencias ordenadas, con cuatro elementos cada una, que hay que fusionar para obtener una sola ordenada y con todos los elementos.

1, 20, 30, 40 (()) 2, 3, 5, 7

La llamada a FUSION_DC desde MS efectua la fragmentación y se obtienen 4 zonas con los siguientes elementos [1,20] [30,40] (()) [2,3] [5,7]. Después de dos llamadas a FUSIÓN_DC en que se habrán fusionado las partes izquierdas ([1,20] con [2,3]) y las derechas ([30,40] con [5,7]) de cada una de las zonas, el resultado será :

1, 2, 3, 20 (()) 5, 7, 30, 40

bien colocados

y sólo queda ordenar la parte central ([3,20] y [5,7]) lo que se consigue con la tercera llamada a FUSION_DC. Finalmente se tiene una secuencia totalmente ordenada.

El coste de FUSION_DC se puede calcular aplicando la recurrencia :

$$T(n,n) = \begin{cases} 1 & n=2 \\ 3.T\ (n/2,n/2) + 1, & n>2 \end{cases}$$

con lo que se consigue un coste de $\theta(n^{\log_2 3}) = \theta(n^{1.57})$. Este resultado es una muestra de lo mencionado en la sección 1.1 : esta técnica no garantiza nada sobre la calidad de la solución obtenida. En este caso, el algoritmo de fusión obtenido aplicando Divide y Vencerás es peor que la solución clásica para fusión que tiene un coste de $\theta(n)$.

1.5. EL PROBLEMA DE LA SELECCION

Sea *T* un vector de *n* elementos naturales y sea *k* una constante tal que $1 \le k \le n$. Diseñar un algoritmo que devuelve aquel elemento de *T* que cumple que si *T* estuviera ordenado crecientemente, el elemento devuelto sería el que ocuparía el *k*-ésimo lugar.

Existe una estrategia evidente para obtener la solución a este problema. Consiste en ordenar crecientemente el vector y luego devolver el elemento que ocupa la posición *k*-ésima. Su coste es $\theta(n \cdot \log n)$ debido a la ordenación previa que hay que realizar.

Sin embargo, la estrategia buena es, como no, un Divide y Vencerás que no necesita ordenar previamente el vector de entrada. Esta solución utiliza mecanismos que ya aparecen en otros algoritmos : del Quicksort hereda el mecanismo de selección de un pivote y la idea de organizar la información respecto de ese pivote (los menores a un lado y los mayores a otro), y como en la búsqueda dicotómica, sólo busca en la zona que le interesa. En concreto, el algoritmo que se presenta a continuación, una vez seleccionado un pivote, organiza el vector en 3 zonas : los elementos menores que el pivote, los que son igual que el pivote y los mayores que él. Al tiempo que hace la distribución, cuenta el número de elementos que compone cada zona, y el tamaño de cada una y el valor de *k* le permiten decidir en qué zona se encuentra el elemento *k*-ésimo que anda buscando.

```
función SELECCIONAR ( T es vector; n, k es nat ) dev ( x es nat )
{ Pre : 1≤k≤ n }
/* n es la dimensión del vector T y k es el elemento buscado */
    [  n≤tamaño_umbral --->  T := ORDENAR_CRECIENTEMENTE( T );
                             x := T[k]
    [] n>tamaño_umbral --->  p:= PIVOTE(T, 1, n);
                             u := num_elems(T, 1, n, < p);
                             v := num_elems(T, 1, n, ≤ p );
    /* u cuenta los elementos de T, entre 1 y n, que son estrictamente menores que el
    pivote y v aquellos que son menores o iguales que el pivote. Queda claro que
    estrictamente mayores que el pivote habrá n-v */

    /* la cantidad de elementos de u y v y su relación con el valor de la k, permiten
    plantear la búsqueda en 3 zonas */
                             [     k ≤ u    ---> U := CONST_VECTOR( T , <p, 1, u )
                              /* U contiene los elementos de T que son menores que
                              el pivote. Los índices de U van desde 1 a u */
                                          x := SELECCIONAR( U, u, k )
                             /* llamada recursiva sobre U buscando el k-ésimo */
```

[] u < k ≤ v ---> x := p
/* el pivote es el elemento buscado */
[] k > v ---> V := CONST_VECTOR(T , >p, 1, n-v)
/* V contiene los elementos de T que son mayores que
el pivote. Los índices de V van desde 1 a n-v */
x := **SELECCIONAR**(V, n-v, k-v)
]
{ *Post :* (N i : 1≤i≤ n : T[i] ≤ x) ≥ k }
dev (x)
ffunción

Para calcular el coste de este algoritmo se necesita conocer, previamente, el coste de las funciones que utiliza. Supongamos, inicialmente, que PIVOTE selecciona un buen pivote, es decir, la mediana del vector y que es capaz de hacerlo en, como mucho, O(n). Por otro lado, no es difícil ver que num_elems y CONST_VECT pueden hacerse a la vez y que requieren O(n). De este modo se tiene que la recurrencia que se plantea es :

$T(n) = T(n/2) + \theta(n)$, de donde se obtiene que $T(n) = \theta(n)$

No está nada mal : se pasa de coste $\theta(n \cdot \log n)$, usando la solución evidente, a coste $\theta(n)$ con una solución más sofisticada. Pero este cálculo se ha hecho suponiendo que PIVOTE se comportaba de la mejor manera posible, devolviendo la mediana de los elementos del vector. Hay que conseguir que PIVOTE funcione de la forma deseada y el problema está en cómo calcular la mediana en tiempo $\theta(n)$. La idea es recurrir al propio SELECCIONAR para obtener la mediana y utilizarla como pivote. En realidad, es suficiente con disponer de una aproximación lo bastante buena de la mediana, la pseudomediana. El siguiente algoritmo calcula la pseudomediana como pivote con un coste de O(n) :

función PIVOTE (S es vector; n es nat) dev (m es nat)
{ *Pre :* CIERTO }
j := n div 5; [n mod 5 = 0 ---> seguir
[] n mod 5 ≠ 0 ---> j := j+ 1
]
/* j indica cuántos grupos de 5 elementos se pueden construir a partir del vector de entrada. A lo sumo existe un grupo con menos de 5 elementos y no vacio */
Para i = 1 hasta j hacer
T[i] := MEDIANA (S, 5i -4, 5i)
/* T[i] contiene el valor de la mediana exacta de los 5 elementos de S que se están explorando (los que van de la posición 5i -4 a la 5i). Al mismo tiempo que se calcula la mediana, se ordenan los 5 elementos tratados. El coste de la operación MEDIANA es O(1) */
fpara

/* T desde 1 a j contiene las medianas exactas de los j grupos de 5 elementos que se han obtenido a partir del vector de entrada */

/* llamada a Seleccionar para calcular la mediana (j div 2) de las j medianas */

m := **SELECCIONAR**(T, j, j div 2)

{ *Post :* m es la pseudomediana de los n elementos del vector de entrada S }

dev (m)

ffunción

El coste de calcular el pivote de esta forma es $T_{pivote}(n) = T_{seleccionar}(n/5) + \theta(n)$. Esta expresión se puede generalizar suponiendo que en lugar de formar grupos de 5 elementos, se forman grupos de *q* elementos. Entonces la expresión del coste que se obtiene es :

$$T_{pivote}(n) = T_{seleccionar}(n/q) + \theta(n)$$

El coste de SELECCIONAR se puede expresar ahora de la siguiente forma :

$T_{seleccionar}(n) =$ coste de calcular pivote --------------------> $T_{seleccionar}(n/q) + \theta(n)$

+ coste de construir *U* y *V* -------------------> $+ \theta(n)$

+ coste de la nueva llamada a seleccionar --> $+ T_{seleccionar}(n')$ y $(n' < n)$.

Ahora hay que determinar exactamente el valor del tamaño de datos, *n'*, con el que se llama nuevamente a seleccionar. El valor que devuelve PIVOTE, *m*, se sabe que es la pseudomediana de *n/q* elementos. Esto implica que existen *n/2q* elementos mayores o iguales que *m* en *T*.

Cada uno de los *n/q* elementos es, a su vez, la mediana de un conjunto de *q* elementos. Esto implica que cada uno de los *n/q* elementos tiene *q/2* elementos mayores o iguales.

Entonces, $n/2q \cdot q/2 = n/4$ elementos mayores o iguales que *m*. De esto se puede deducir que $|U| \leq 3n/4$. Realizando un razonamiento idéntico se obtiene que $|V| \leq 3n/4$. De este modo se tiene que $T_{seleccionar}(n') = T_{seleccionar}(3n/4)$ con lo que

$$T_{seleccionar}(n) = T_{seleccionar}(3n/4) + T_{seleccionar}(n/q) + 2 \cdot \theta(n)$$

Un valor de *q* que hace que $T_{seleccionar}(n) = \theta(n)$ es, precisamente, el $q=5$ que se utiliza en el algoritmo PIVOTE.

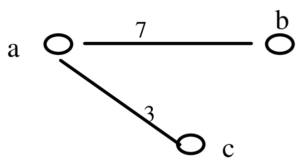
En [BB 90] hay un análisis más profundo sobre la calidad de la pseudomediana que obtiene el algoritmo PIVOTE presentado.

2. GRAFOS

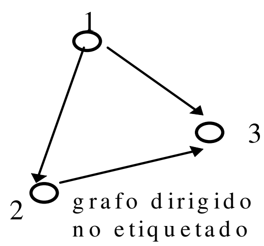
2.1. ESPECIFICACION ALGEBRAICA Y DEFINICIONES

El lector o lectora ya conocen el tipo de datos grafo porque ha sido introducido en otras asignaturas que preceden a ésta. Esta sección se va a dedicar a formalizar la noción intuitiva de grafo y recordar la terminología que se emplea con ellos. En general, un grafo se utiliza para representar relaciones arbitrarias entre objetos del mismo tipo. Los objetos reciben el nombre de nodos o vértices y las relaciones entre ellos se denominan aristas. Normalmente un grafo *G* formado por el conjunto de vértices *V* y por el conjunto de aristas *E*, se denota por el par G=(V,E).

Existen grafos dirigidos y no dirigidos dependiendo de si las aristas están orientadas o no lo están, es decir, si una arista entre dos vértices se puede recorrer en un sólo sentido o en ambos. También existen grafos etiquetados o no en función de si las aristas tienen o no información asociada. Gráficamente (los círculos representan los vértices y las líneas que los unen representan las aristas) :



grafo NO dirigido y etiquetado



grafo dirigido y no etiquetado

2.1.1. ESPECIFICACION

Se dispone de un método, la especificación algebraica, para describir formalmente la noción de grafo. A continuación se definen los grafos dirigidos y etiquetados pero, fácilmente, puede obtenerse a partir de ésta la definición de los grafos no dirigidos y/o sin etiquetar.

<u>Universo</u> GRAFO-DIRIG-ETIQ (vértice, etiqueta) <u>usa</u> bool

<u>género</u> grafo

<u>operaciones</u>

crea : ---> grafo

añ-v : grafo, vértice ---> grafo

añ-a : grafo, vértice, vértice, etiqueta ---> grafo

/* constructoras generadoras: crea, añade vértice y añade arista */

valor : grafo, vértice, vértice ---> etiqueta

ex-v : grafo, vértice ---> bool

ex-a: grafo, vértice, vértice ---> bool

suc : grafo, vértice ---> conj-vértices

/* consultoras : valor de la etiqueta de una arista, existe vértice, existe arista y vértices sucesores de uno dado */

borra-v : grafo, vértice ---> grafo

borra-a : grafo, vértice, vértice ---> grafo

/* constructoras modificadoras : borra vértice y borra arista */

<u>ecuaciones de error</u>

EE1. añ-a (g,v1,v1,e) ≡ error

/* no se permite que una arista tenga como origen y destino el mismo vértice */

EE2. (¬ex-v (g,v1)) ∨ (¬ex-v(g,v2)) ⇒ añ-a (g,v1,v2,e) ≡ error

EE3. valor (crea, v1,v2) ≡ error

<u>ecuaciones</u>

1. añ-v (añ-v (g,v), v) ≡ añ-v (g, v)
2. añ-v (añ-v (g, v1), v2) = añ-v (añ-v (g, v2), v1)
3. (v1≠v2) ∧ (v1≠v3) ⇒
   añ-a (añ-v (g, v1), v2, v3, e) ≡ añ-v (añ-a (g, v2,v3, e), v1)
4. añ-a (añ-a (g, v1, v2, e1), v1, v2, e2) ≡ añ-a (g, v1, v2, e2)
5. (v1≠v3) ∨ (v2≠v4) ⇒
   añ-a (añ-a (g, v1, v2, e1), v3, v4, e2) ≡ añ-a (añ-a (g, v3, v4, e2), v1, v2, e1)
6. valor (añ-v (g, v), v1, v2) ≡ valor (g, v1, v2)
7. valor (añ-a (g, v1, v2, e) , v1, v2) = e
8. (v1≠v3) ∨ (v2≠v4)⇒ valor (añ-a (g, v1, v2, e), v3, v4) ≡ valor (g, v3, v4)
9. ex-v (crear, v) ≡ FALSO
10. ex-v (añ-v (g, v), v) ≡ CIERTO
11. (v1≠v2) ⇒ ex-v (añ-v (g, v1), v2) ≡ ex-v (g, v2)
12. ex-v (añ-a (g, v1, v2, e), v) ≡ ex-v (g, v)
13. ex-a (crear, v1, v2) ≡ FALSO
14. ex-a (añ-v (g, v), v1, v2) ≡ ex-a (g, v1, v2)
15. ex-a (añ-a (g, v1,v2, e), v1, v2) ≡ CIERTO
16. (v1≠v3) ∨ (v2≠v4) ⇒ ex-a (añ-a (g, v1, v2, e), v3, v4) ≡ ex-a (g, v3, v4)
17. suc (crea, v) ≡ conj-vacio
18. suc (añ-v (g, v1), v2) ≡ suc (g, v2)
19. suc (añ-a (g, v1, v2, e), v) ≡ suc (g, v)
20. suc (añ-a (g, v1, v2, e), v1) ≡ {v2} ∪ suc (g, v1)

Como puede apreciarse, dados dos vértices *a* y *b*, no pueden existir varias aristas de *a* hacia *b* con distintas, ni con las mismas, etiquetas. Tampoco es posible que exista una arista que tenga como origen y destino el mismo vértice.

Esta especificación se puede convertir en la de un GRAFO-NODIRIGIDO-ETIQ con tan sólo añadir la ecuación :

añ-a (g, v1, v2,e) ≡ añ-a (g, v2, v1, e)

Se puede enriquecer la especificación que se acaba de proporcionar con operaciones auxiliares tales como existe camino entre dos vértices (ex-camino) o descendientes de un vértice *v* (descen). Esta última operación retorna el conjunto de todos aquellos vértices del grafo tales que existe camino desde *v* a cualquiera de ellos.

<u>operaciones</u>

ex-camino : grafo, vértice, vértice ---> bool

descen : grafo, vértice ---> conj-vértices

<u>ecuaciones</u>

1. ex-camino (crea, v1, v2) ≡ (v1 = v2)
2. ex-camino (añ-v (g, v), v1, v2) ≡ ex-camino (g, v1, v2)
3. ex-camino (añ-a (g, v1, v2 ,e), v1, v2) ≡ CIERTO
4. ex-camino (añ-a (g, v1, v2, e), v3, v4) ≡ ex-camino (g, v3, v4) ∨
   (ex_camino (g, v3, v1) ∧ ex-camino (g, v2, v4))
5. descen (crea, v) ≡ conj-vacio
6. descen (añ-v (g, v1), v2) ≡ descen (g, v2)
7. descen (añ-a (g, v1, v2, e), v1) ≡ descen (g, v1) ∪ {v2} ∪ descen (g, v2)
8. descen (añ-a (g, v1, v2, e), v2) ≡ descen (g, v2)
9. ¬ ex-camino (g, v3, v1) ⇒ descen (añ-a (g, v1, v2, e), v3) ≡ descen (g, v3)
10. ex-camino (g, v3, v1) ⇒
    descen (añ-a (g, v1, v2, e), v3) ≡ descen (g, v3) ∪ {v2} ∪ descen (g, v2)

2.1.2. DEFINICIONES

Una vez descritos los grafos con los que vamos a trabajar, pasemos a repasar la terminología habitual que se emplea en su entorno.

2.1.2.1. **Adyacencias**

Sea G=(V,E) un grafo NO DIRIGIDO. Sea *v* un vértice de *G*, *v*∈V. Se define :

- adyacentes de *v*, ady(v) = { v'∈V | (v,v')∈E }
- grado de *v*, grado(v) = | ady(v) |. Si un vértice está aislado, su grado es cero.

Sea G=(V,E) un grafo DIRIGIDO. Sea *v* un vértice de *G*, *v*∈V. Se define :

- sucesores de *v*, suc(v) = { v'∈V | (v,v')∈E }

- predecesores de *v*, pred(v) = { v'∈V | (v',v)∈E }
- adyacentes de *v*, ady(v) = suc(v)∪pred(v)
- grado de *v*, grado(v) = | |suc(v)| - |pred(v)| |
- grado de entrada de *v* , grado_e(v) = |pred(v)|
- grado de salida de *v*, grado_s(v) = |suc(v)|

2.1.2.2. **Caminos**

Un CAMINO de longitud n≥0 en un grafo G=(V,E) es una sucesión $\{v_0,v_1, ...,v_n\}$ tal que :
- todos los elementos de la sucesión son vértices, es decir, ∀i: 0≤i≤n : v_i∈V, y
- existe arista entre todo par de vértices consecutivos en la sucesión, o sea, ∀i: 0≤i<n : (v_i,v_{i+1})∈E.

Dado un camino $\{v_0,v_1, ...,v_n\}$ se dice que :
- sus extremos son v_0 y v_n
- es PROPIO si n>0. Equivale a que, como mínimo, hay dos vértices en la secuencia y, por tanto, su longitud es ≥ 1.
- es ABIERTO si $v_0 \neq v_n$
- es CERRADO si $v_0=v_n$
- es SIMPLE si no se repiten aristas
- es ELEMENTAL si no se repiten vértices, excepto quizás los extremos. Todo camino elemental es simple (si no se repiten los vértices seguro que no se repiten las aristas).

Un CICLO ELEMENTAL es un camino cerrado, propio y elemental, es decir, es una secuencia de vértices, de longitud mayor que 0, en la que coinciden los extremos y no se repiten ni aristas ni vértices. Se puede especificar esta operación sobre un grafo DIRIGIDO como sigue:

<u>operación</u>

cíclico : grafo ---> bool

<u>ecuaciones</u>

cíclico (crea) ≡ FALSO
cíclico (añ-v (g, v)) ≡ cíclico (g)
cíclico (añ-a (g, v1, v2, e)) ≡ (cíclico (g)) ∨ ex-camino (g, v2, v1)

2.1.2.3. **Conectividad**

Sea G=(V,E) un grafo NO DIRIGIDO. Se dice que :
- es CONEXO si existe camino entre todo par de vértices. La especificación de esta operación viene a continuación.

conexo : grafo ---> bool

<u>ecuaciones</u>

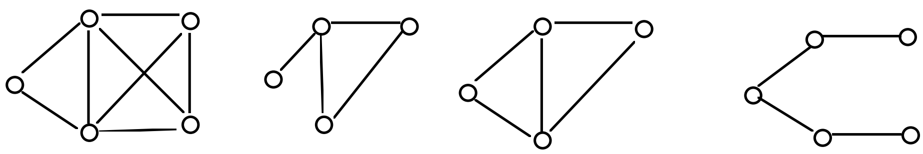
conexo (crea) ≡ FALSO
conexo (añ-v (crea, v)) ≡ CIERTO
conexo (añ-v (g, v)) ≡ (conexo (g)) ∧ (ex-arista (g, v, (algún otro vértice de g)))

conexo (añ-a (g, v1, v2, e)) $\equiv$

conexo (g) $\vee$ (descen(v1) $\cup$ descen(v2) $\cup$ $\{v1\}$ $\cup$ $\{v2\}$) = V

- es un BOSQUE si no contiene ciclos
- es un ARBOL NO DIRIGIDO si es un bosque conexo

Un SUBGRAFO H=(U,F) del grafo *G*, es el grafo *H* tal que $U \subseteq V$ y $F \subseteq E$ y $F \subseteq UxU$.
Un ARBOL LIBRE del grafo *G* es un subgrafo de él, H=(U,F), tal que es un árbol no dirigido y contiene todos los vértices de *G*, es decir, U=V. Los árboles libres son árboles (bosque conexo) sin un elemento distinguido o raíz y, sin embargo, cualquier vértice del árbol puede actuar como tal.
Un SUBGRAFO INDUCIDO del grafo G es el grafo H=(U,F) tal que $U \subseteq V$ y *F* contiene aquellas aristas de *E* tal que sus vértices pertenecen a *U*.
Ejemplo :



G= (V, E) — H = (U,F) Subgrafo de G — H= (U, F) Subgrafo inducido de G — Arbol libre asociado a G

Sea G=(V,E) un grafo NO DIRIGIDO. Se tiene que una COMPONENTE CONEXA de *G* es un subgrafo conexo de *G*, H=(U,F), tal que ningún otro subgrafo conexo de *G* contiene a *H*. Dicho de otra forma, una componente conexa es un subgrafo conexo MAXIMAL. Un grafo NO CONEXO *G* se puede partir de una sola forma en un conjunto de subgrafos conexos. Cada uno de ellos es una COMPONENTE CONEXA de *G*.
Sea G=(V,E) un grafo NO DIRIGIDO, se dice que *G* es BIPARTIDO si todos sus vértices se pueden dividir en dos conjuntos disjuntos tal que todas sus aristas enlazan 2 vértices en que cada uno de ellos pertenece a un conjunto distinto.

Sea G=(V,E) un grafo DIRIGIDO. Se dice que :
- es FUERTEMENTE CONEXO si existe camino entre todo par de vértices en ambos sentidos.
- una COMPONENTE FUERTEMENTE CONEXA de *G* es un subgrafo fuertemente conexo de *G*, H=(U,F), tal que ningún otro subgrafo fuertemente conexo de *G* contiene a *H*. Equivale a que una componente fuertemente conexa es un subgrafo fuertemente conexo MAXIMAL.

2.1.2.4. **Algunos grafos particulares**

COMPLETO

Un grafo no dirigido G=(V,E) es COMPLETO si existe arista entre todo par de vértices de V. El número de aristas, |E|, de un grafo completo no dirigido es exactamente $n \cdot (n-1)/2$. Si se trata de un grafo dirigido entonces $|E|=n \cdot (n-1)$.

GRAFOS EULERIANOS

Se dice que un grafo no dirigido G=(V,E) es EULERIANO si existe un camino cerrado, de longitud mayor que cero, simple (no se repiten aristas) pero no necesariamente elemental que incluye todas las aristas de *G*.

Los siguientes lemas permiten determinar si un grafo dado es euleariano :

Lema : Un grafo no dirigido y conexo es euleriano si y sólo si el grado de todo vértice es par

Lema : Un grafo dirigido y fuertemente conexo es euleriano si y sólo si el grado de todo vértice es cero

Averiguar si un grafo no dirigido y conexo es euleriano tiene un coste de $\theta(n)$, si se supone conocido el grado de cada vértice. Gracias al lema basta con recorrer el conjunto de vértices y comprobar si el grado de cada uno de ellos es par o no lo es.

GRAFOS HAMILTONIANOS

Un grafo no dirigido G=(V,E) es HAMILTONIANO si existe un camino cerrado y elemental (no se repiten vértices) que contiene todos los vértices de *G*. Si existe, el camino se llama circuito hamiltoniano.

En este caso no existe ninguna propiedad que permita determinar la existencia del camino. Esto implica que averiguar si existe camino tiene el mismo coste que calcular directamente el camino. La forma habitual de resolverlo es ir construyendo todos los caminos posibles y, para cada uno de ellos, comprobar si cumple la condición de hamiltoniano. Esta forma de resolver el problema tiene coste exponencial respecto del número de vértices del grafo y el esquema usado es Vuelta Atrás.

2.2. IMPLEMENTACIONES. ANALISIS DEL COSTE DE LAS OPERACIONES

Son suficientemente conocidas las implementaciones típicas de los grafos : usando matrices de adyacencia, listas de adyacencia o multilistas de adyacencia. A continuación daremos un breve repaso al coste de algunas de las operaciones básicas para cada una de estas implementaciones. Para más detalles consultar [Fra 93].

2.2.1. MATRICES DE ADYACENCIA

Sea G=(V,E) y sea n=|V|. Supongamos implementado el grafo *G* en una matriz de booleanos M[1..n,1..n] de modo que ($\forall v,w \in V$: M[v,w]=CIERTO $\Leftrightarrow$ $(v,w) \in E$). Si el grafo está etiquetado será necesaria, en vez de una matriz de booleanos, una matriz del tipo de las etiquetas del grafo.

El espacio ocupado por la matriz es del orden de $\theta(n^2)$. En realidad, un grafo no dirigido sólo necesita la mitad del espacio $(n^2/2)$-n, pero uno dirigido lo necesita todo excepto la diagonal, es decir, n^2-n.

Respecto al coste temporal de las operaciones básicas del grafo se tiene que varían entre $\theta(1)$ y $\theta(n^2)$. Por ejemplo, para un grafo no dirigido :

- Crear el grafo (crea) es $\theta(n^2)$

- Adyacentes a un vértice (ady), borrar vértice (borra-v) son $\theta(n)$
- Añadir arista (añ-a), existe vértice (ex-v), existe arista (ex-a), borrar arista (borra-a), valor de una etiqueta (valor) son $\theta(1)$.

Para un grafo dirigido destacar que todo es igual y que el coste de calcular los predecesores de un vértice dado (pred) es $\theta(n)$.

En general, si el número de aristas del grafo es elevado, las matrices de adyacencia tienen buenos costes espacial y temporal para las operaciones habituales.

2.2.2. LISTAS Y MULTILISTAS DE ADYACENCIA

En el caso de las listas de adyacencia, la estructura que se emplea para implementar un grafo $G=(V,E)$ con $n=|V|$, es un vector $L[1..n]$ tal que $L[i]$, con $1\leq i\leq n$, es una lista formada por los identificadores de los vértices que son adyacentes (sucesores, si el grafo es dirigido) al vértice con identificador *i*.

El espacio ocupado por esta implementación es de orden $\theta(n+e)$, con $e=|E|$.

Examinando el coste temporal de algunas operaciones básicas, se tiene que :

- crear la estructura (crea) es $\theta(n)$.
- Añadir vértice (añ-v) y existe vértice (ex-v) son $\theta(1)$.
- Añadir arista (añ-a), necesita comprobar que la arista que se añade no exista previamente y luego debe añadirla si es el caso. Esto implica recorrer toda la lista asociada al vértice origen de la arista para detectar si ya existe. El peor caso requerirá recorrer la lista completa, que puede tener un tamaño máximo de *n-1* elementos, para efectuar posteriormente la inserción. Así se tiene un coste $\theta(n)$ para la operación de añadir. Una implementación de la lista de aristas en un AVL (árbol binario equilibrado) permite reducir el coste de esta operación a $\theta(\log n)$.

Un razonamiento similar puede hacerse para las operaciones de existencia de arista (ex-a) y de borrado de aristas (borra-a).

El coste que requieren las operaciones sucesores (suc) y predecesores (pred) de un vértice dado en un grafo dirigido es el siguiente : para obtener los sucesores de un vértice basta con recorrer toda su lista asociada, por tanto, $\theta(n)$. Sin embargo, para obtener los predecesores hay que recorrer, en el peor caso, toda la estructura para ver en qué listas aparece el vértice del que se están buscando sus predecesores, por tanto, $\theta(n+e)$. Obviamente, si las listas de sucesores de un vértice están implementadas en un AVL entonces el coste de esta operación es $\theta(n\cdot\log n)$.

La implementación del grafo usando multilistas sólo tiene sentido para grafos dirigidos. Su objetivo es mejorar el coste de la operación que obtiene los predecesores para que pase a tener coste $\theta(n)$ en lugar del coste $\theta(n+e)$ con listas. Se consigue, además, que el coste de las demás operaciones sea el mismo que el que se tenía usando listas.

En general, para implementar un grafo es conveniente usar listas de adyacencia cuando el grafo es poco denso, es decir, tiene pocas aristas y, además, el problema a resolver tiene que recorrerlas todas.

2.2.3. EL PROBLEMA DE LA CELEBRIDAD

Se plantea el siguiente problema : demostrar que determinar si un grafo dirigido G=(V,E) con n=|V| contiene una celebridad (también sumidero o pozo), es decir, tiene un único vértice con grado de entrada *n-1* y grado de salida cero, puede averiguarse en O(n) cuando el grafo está implementado en una matriz de adyacencia.

Se supone que los vértices están identificados por un natural entre *1* y *n* y que la matriz es de booleanos con la interpretación habitual. Si existe arista del vértice *i* al vértice *j* se dice que *i conoce a j*. Esta es la especificación de la función a diseñar.

<u>función</u> CELEBRIDAD (M <u>es</u> matriz[1..n,1..n] <u>de</u> bool) <u>dev</u> (b <u>es</u> bool, v <u>es</u> vértice)
{ *Pre :* CIERTO }
{ *Post :* $b \Rightarrow (1 \le v \le n) \wedge (\forall i : (1 \le i \le n) \wedge (i \neq v) : (M[i,v]=\text{CIERTO}) \wedge (M[v,i]=\text{FALSO}))$
$\wedge\ \neg b \Rightarrow \neg(\exists j : 1 \le j \le n : (\forall i : (1 \le i \le n) \wedge (i \neq j) : (M[i,j]=\text{CIERTO}) \wedge (M[j,i]=\text{FALSO})))$ }

De la especificación se deduce que hay que recorrer toda la matriz, en el peor de los casos, para determinar la no existencia de celebridad. Se tiene, por tanto, un coste de $\theta(n^2)$ que es más elevado que el pedido en el enunciado.
Se puede intentar un planteamiento recursivo (inducción) para mejorar el coste :
- BASE INDUCCION : Si el grafo tiene dos nodos, ¿ cuántas consultas hay que realizar para averiguar si existe celebridad ?. Es evidente que se necesitan dos consultas.
- HIPOTESIS INDUCCION : Supongamos que se puede encontrar la celebridad para el caso *n-1*.
- PASO INDUCTIVO : ¿ cómo se resuelve para *n* nodos ?. Se pueden producir una de las tres situaciones siguientes :

 1/ la celebridad es uno de los *n-1* nodos ya explorados. En ese caso hay que efectuar dos consultas : la celebridad no conoce al nodo *n* y *n* conoce a la celebridad (no existe arista de la celebridad a *n* pero si existe arista del nodo *n* a la celebridad) para determinar si es definitivamente la celebridad.

 2/ la celebridad es el nodo *n*. En ese caso hay que efectuar 2·(n-1) consultas para averiguar si los *n-1* nodos anteriores conocen a *n* y *n* no conoce a ninguno de los *n-1*. De este modo se averigua si *n* es la celebridad

 3/ no existe la celebridad

Se comprueba que para *n* nodos, y en el peor de los casos, son necesarias 2·(n-1) consultas. Para el paso *n-1* serían necesarias 2·(n-2), ..., y así hasta el paso n=2 en que se necesitarían 2·(2-1)=2 consultas. Calculando el número total de consultas se obtiene n·(n-1). ¡Continúa siendo cuadrático!.

La forma correcta de plantear el problema consiste en aplicar la técnica denominada *búsqueda por eliminación*. Consiste en lo siguiente : celebridades sólo hay una, si es que existe, pero no_celebridades hay, como mínimo, *n-1*. ¿ Cuántas consultas hay que efectuar para averiguar, dado un par de nodos, si alguno de ellos no es celebridad ?. Sólo una pregunta y ya se puede descartar a un nodo como candidato a celebridad.
Formalizando este razonamiento :
- si existe arista del nodo *i* al nodo *j*, *i conoce a j*, entonces *i* no es celebridad y se puede descartar.
- si NO existe arista del nodo *i* al nodo *j*, *i no conoce a j*, entonces *j* no es celebridad y se puede descartar.

Dados los *n* nodos del grafo, se necesitan *n-1* consultas para obtener el nodo candidato a celebridad. Luego hay que comprobar que efectivamente lo es y, como ya se ha mencionado anteriormente, son necesarias 2·(n-1) consultas. En total hay que efectuar (n-1)+2·(n-1) consultas para resolver el problema, lo que corresponde a un tiempo O(n), que ahora si coincide con lo que pedía el enunciado del problema .
Un algoritmo que implementa esta búsqueda es el que viene a continuación.

```
función CELEBRIDAD ( M es matriz[1..n,1..n] de bool ) dev ( b es bool; v es vértice )
{ Pre : CIERTO }
        i:=1; j:=2; p:=3; b:= FALSO;
        *[ p ≤ n+1 ---> [    M[i,j] = CIERTO ---> i:=p      /* se descarta i */
                        []   M[i,j] = FALSO ---> j:= p      /* se descarta j */
                        ]
                        p:= p+1;
         ]
        { p=n+2 ∧ ( (i= n+1 => candidato el j) ∨ ( j=n+1 => candidato el i ))}
        [ i = n+1 ---> v := j
        [] i≠ n+1 ---> v := i
        ]
/* Aquí vendría el bucle para comprobar si el candidato v es o no la celebridad. Necesitará
efectuar, como máximo, 2·(n-1) consultas. Se deja como ejercicio para el lector */
        ...
        dev ( b, v )
{ Post : b ⇒ (1≤v≤n ) ∧ (∀i : (1≤i≤n)  ∧  (i≠v) : (M[i,v]=CIERTO) ∧ (M[v,i]=FALSO))
 ∧  ¬b ⇒ ¬ (∃j : 1≤j≤n : (∀i : (1≤i≤n)  ∧  (i≠j) : (M[i,j]=CIERTO) ∧ (M[j,i]=FALSO)))}
ffunción
```

2.3. ALGORITMOS SOBRE GRAFOS

Este apartado está dedicado a algoritmos que permiten recorrer completamente un grafo: el recorrido en profundidad y el recorrido en anchura. Existen otros algoritmos de recorrido muy conocidos (Prim, Kruskal, Dijkstra, Floyd, etc.) que se verán en temas posteriores como ejemplos de instanciación de otros esquemas.

Para facilitar la escritura de los algoritmos que se van a presentar, se amplia el repertorio de operaciones del grafo. Más de una vez se necesitará anotar, en cada uno de los vértices del grafo, alguna información que luego el recorrido actualizará y consultará. Para ello se introduce la operación :

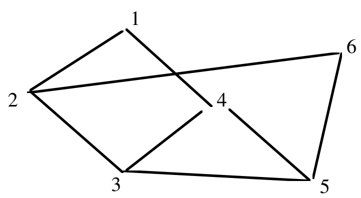
Marcar : grafo, vértice, *inf* ---> grafo

donde *inf* corresponde al tipo de información que hay que asociar al vértice. Para consultar el valor de *inf* se utilizaran operaciones dependientes del propio tipo de *inf*.

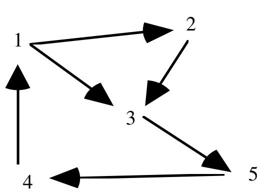
2.3.1. RECORRIDO EN PROFUNDIDAD

El algoritmo de recorrido en profundidad, en inglés *depth-first search* y que se escribirá a partir de ahora DFS para abreviar, se caracteriza porque permite recorrer completamente el grafo. Si el grafo es no dirigido, cada arista se visita 2 veces (una en cada sentido) y si es dirigido cada arista se visita una sola vez.
El orden de recorrido es tal que los vértices se recorren según el orden 'primero en profundidad'. Primero en profundidad significa que dado un vértice *v* que no haya sido visitado, DFS primero visitará *v* y luego, recursivamente aplicará DFS sobre cada uno de los adyacentes/sucesores de ese vértice que aún no hayan sido visitados. Para poner en marcha el recorrido se elige un vértice cualquiera como punto de partida y el algoritmo acaba cuando todos los vértices han sido visitados. Se puede establecer un claro paralelismo entre el recorrido en preorden de un árbol y el DFS sobre un grafo.

El orden de recorrido de los vértices del grafo que produce el DFS no es único, es decir, para un mismo grafo se pueden obtener distintas secuencias de vértices de acuerdo con el orden de visitas. Todo depende de en qué vértice se comienza el recorrido y de en qué orden se van tomando los adyacentes de un vértice dado. La siguiente figura ilustra las diferentes secuencias que puede producir el DFS.



Recorrido 1 : 1, 2, 3, 4, 5, 6
Recorrido 2 : 3, 5,4, 1, 2, 6



Recorrido 1 : 1, 2, 3, 5, 4
Recorrido 2 : 3, 5, 4, 1, 2
Recorrido 3 : 4, 1, 3, 5, 2

En la figura de la izquierda se presenta un grafo no dirigido y dos, de las muchas, secuencias de recorrido en profundidad posibles. En la primera se ha comenzado por el vértice *1* y en la segunda por el vértice *3*. La figura de la derecha muestra un grafo dirigido sobre el que se han llevado a cabo tres recorridos, comenzando por los vértices *1*, *3* y *4*, respectivamente.

En el algoritmo de recorrido en profundidad para grafos NO DIRIGIDOS que se presenta a continuación, a cada vértice se le asocia un valor inicial, 'no_visto', para indicar que el recorrido aún no ha pasado por él. En el momento en que el recorrido alcanza un vértice con ese valor, lo modifica poniéndolo a 'visto'. Significa que se ha llegado a ese vértice por uno, el primero explorado, de los caminos que lo alcanzan y que se viene de un vértice que también ha sido ya visitado (excepto para el vértice inicial). Nunca más se alcanzará ese vértice por el mismo camino, aunque puede hacerse por otros distintos del mencionado.

```
acción REC-PROF ( g es grafo )
{ Pre : CIERTO }
        Para cada v∈V  hacer  g:= Marcar( g, v, 'no_visto' ) fpara
        Para cada v∈V  hacer
        [    Visto( g, v ) ---> seguir
        []¬Visto( g, v ) ---> g:= REC-PROF-1( g, v )
        ]
        fpara
{ Post : Todos los vértices de g han sido marcados a visto en el orden que sigue el recorrido
en profundidad de g }
faccción

función  REC-PROF-1 ( g es grafo; u es vértice ) dev ( g es grafo )
{ Pre : (u∈V) ∧ ¬Visto(g,u)  ∧ (g=g')}
        g:= Marcar( g, u, 'visto' );
        TRATAR( u );            /* PREWORK */
        Para cada w ∈ ady(g,u) hacer
        [     Visto( g,w ) ---> seguir
        [] ¬Visto( g,w ) ---> g:= REC-PROF-1( g,w )
        ]
        TRATAR( u,w );          /* POSTWORK */
        fpara
{ Post : Visto(g,u) ∧ marcados a visto todos los vértices de g accesibles desde u por caminos
formados exclusivamente por vértices que no estaban vistos en g',  según el orden de visita
del recorrido en profundidad }
        dev  ( g )
ffunción
```

La función auxiliar REC-PROF-1 es la que realmente hace el recorrido en profundidad. En ella aparecen dos operaciones TRATAR(u) y TRATAR(u,w) que pueden corresponder a acciones vacías, como sucede en el caso del DFS, o pueden corresponder a acciones relevantes, como se verá en algún algoritmo posterior. En general, la acción denominada *prework* se refiere al trabajo previo sobre el vértice visitado, y la denominada *postwork* corresponde al trabajo posterior sobre la última arista tratada (o sobre el mismo vértice sobre el que se hace el prework).

Se puede analizar el coste de este algoritmo suponiendo que prework y postwork tienen coste constante. Si el grafo está implementado sobre una matriz de adyacencia se tiene, claramente, $\theta(n^2)$, pero si lo está sobre listas el coste es $\theta(n+e)$ debido a que se recorren todas las aristas dos veces, una en cada sentido, y a que el bucle exterior recorre todos los vértices.

A continuación se presentan algunas de las aplicaciones del DFS : determinar si un grafo es conexo, numerar los vértices según el orden del recorrido, determinar la existencia de ciclos, etc. Todas estas aplicaciones se implementan a base de modificar, levemente, el algoritmo de recorrido en profundidad, lo que da una idea de su potencia. En [CLR 92] se pueden encontrar éstas y otras aplicaciones.

2.3.1.1. **Conectividad**

Se trata de determinar si un grafo no dirigido es conexo o no lo es. Para ello se puede utilizar la propiedad de que si un grafo es conexo es posible alcanzar todos los vértices comenzando su recorrido por cualquiera de ellos. Se necesita un algoritmo que recorra todo el grafo y que permita averiguar qué vértices son alcanzables a partir del elegido. El DFS es el adecuado. En la función más externa, que aquí se denomina COMPO_CONEXAS, el bucle examina todos los vértices y si alguno no ha sido visitado comienza un recorrido en profundidad a partir de él. Las llamadas desde COMPO_CONEXAS a CONEX-1 indican el comienzo de la visita de una nueva componente conexa y se puede asegurar que, una vez que ha finalizado la visita, no hay más vértices que formen parte de esa misma componente conexa. En la implementación del algoritmo se ha utilizado un natural, *nc*, que se asocia a cada vértice y que indica a qué número de componente conexa pertenece.

<u>función</u> COMPO_CONEXAS (g <u>es</u> grafo) <u>dev</u> (g <u>es</u> grafo; nc <u>es</u> nat)
{ *Pre :* CIERTO }
        <u>Para cada</u> v∈V <u>hacer</u> g:= Marcar(g, v, <0,'no_visto'>) <u>fpara</u>
        nc := 0;
        <u>Para cada</u> v∈V <u>hacer</u>
        [Visto(g, v) ---> seguir

[] ¬Visto(g, v) ---> nc := nc+1;
g:= **CONEX-1**(g, v, nc);
]
<u>fpara</u>

{ *Post :* el grafo ha sido modificado según figura en la Post de CONEX-1 ∧ *nc* contiene el número de componentes conexas que tiene el grafo g}

<u>dev</u> (g, nc)
<u>ffunción</u>

<u>función</u> CONEX-1 (g <u>es</u> grafo; u <u>es</u> vértice; num_compo <u>es</u> nat) <u>dev</u> (g <u>es</u> grafo)

{ *Pre :* (u∈V) ∧ ¬Visto(g,u) ∧ (g=g') ∧ (num_compo = nº de componente conexa que se está recorriendo) }

g:= Marcar(g, u, < num_compo, 'visto'>);

/* Este es el PREWORK, anotar el número de componente conexa a la que pertenece el vértice. En este caso se ha incluido dentro de la operación Marcar */

<u>Para cada</u> w ∈ ady(g,u) <u>hacer</u>
[Visto(g, w) ---> seguir
[]¬Visto(g, w) ---> g:= **CONEX-1**(g, w, num_compo)
]
<u>fpara</u>

{ *Post :* Visto(g,u) ∧ marcados a visto todos los vértices de g accesibles desde u por caminos formados exclusivamente por vértices que no estaban vistos en g', según el orden de visita del recorrido en profundidad. Además a todos los vértices visitados se les ha añadido una información que indica a qué nº de componente conexa pertenecen }

<u>dev</u> (g)
<u>ffunción</u>

El coste de este algoritmo es idéntico al de REC-PROF.

2.3.1.2. **Numerar vértices**

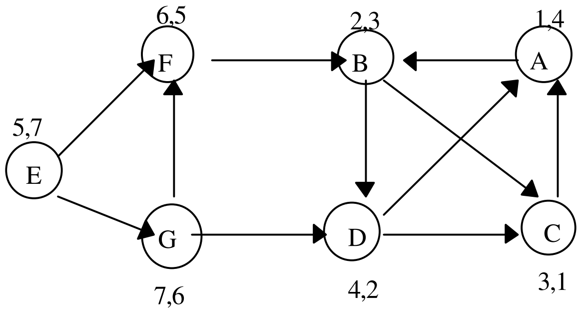
Aprovechando el recorrido en profundidad se puede asociar a cada vértice un valor natural que indicará el orden en que han sido visitados por él (han pasado a 'visto'). A esta asociación se la conoce como la numeración en *el orden del recorrido*. También se puede asociar a cada vértice un natural, no en el momento en que pasa a 'visto', sino en el momento en que toda su descendencia ya ha sido visitada por el DFS. A esta otra numeración se la conoce como la del *orden inverso del recorrido*.

En el algoritmo que se presenta a continuación hay dos valores, de tipo natural, que se almacenan en cada vértice. *Num-dfs*, que contiene la numeración en el orden del recorrido, y *num-invdfs* conteniendo la numeración en orden inverso. Se asume que el tipo vértice es un tipo estructurado, una n-tupla. Con una pequeña modificación se puede conseguir que el algoritmo funcione para grafos dirigidos.

```
función NUMERAR_VERTICES ( g es grafo ) dev ( g es grafo )
{ Pre : g es un grafo no dirigido }
        Para cada v∈V  hacer
                v.visitado := 'no_visto';      v.num-dfs := 0;    v.num-invdfs := 0
        fpara
        ndfs := 0; ninv := 0 ;
        Para cada v∈V  hacer
        [    Visto( g, v )  ---> seguir
        [] ¬Visto( g, v ) --->  <g, ndfs, ninv> := NV-1( g, v, ndfs, ninv );
        ]
        fpara
{ Post : los vértices del grafo han sido marcados según se indica en la Post de NV-1}
        dev ( g  )
ffunción
```

```
función  NV-1 ( g es grafo; u es vértice; nd, ni es nat ) dev ( g es grafo; nd,ni es nat)
{ Pre : (u∈V) ∧ ¬Visto(g,u)  ∧ (g=g') ∧ (nd es el número asociado al último vértice que ha
pasado a 'visto' en el recorrido) ∧ (ni es el número asociado al último vértice que ha
conseguido tener toda su descendencia a 'visto') }
        g:= Marcar( g, u, 'visto' );
        nd:= nd+1;
        u.num-dfs := nd;                              /* PREWORK sobre el vértice */
        Para cada w ∈ ady(g,u) hacer
        [     Visto( g, w ) ---> seguir
        [] ¬Visto( g, w ) ---> < g, nd, ni > := NV-1( g, w, nd, ni)
        ]
        fpara
        ni := ni+1;
        u.num-invdfs := ni;                           /* POSTWORK sobre el vértice */
{ Post : Visto(g,u) ∧ marcados a visto todos los vértices de g accesibles desde u por caminos
formados exclusivamente por vértices que no estaban vistos en g', según el orden de visita
del recorrido en profundidad ∧ u.num-invdfs es la numeración en orden inverso del vértice u
que es el último que ha conseguido tener toda su descendencia a 'visto' ∧ nd es la
numeración en el orden de recorrido del último vértice de g que ha pasado a visto }
        dev  ( g, nd, ni )
ffunción
```

En la figura se muestra un grafo dirigido que ha sido recorrido en profundidad y numerado en el orden del recorrido (valor que aparece a la izquierda) y en el orden inverso (valor que aparece a la derecha separado por una coma del anterior). Se ha comenzado por el vértice con identificador *A* y el siguiente se ha elegido según el orden alfabético.



2.3.1.3. **Arbol asociado al Recorrido en profundidad**

Al mismo tiempo que se efectúa el recorrido en profundidad de un grafo se puede obtener lo que se denomina el *árbol asociado al recorrido en profundidad*. En realidad no siempre se obtiene un árbol sino que depende del grafo de partida. Así, un grafo no dirigido y conexo produce un árbol, un grafo no dirigido y no conexo produce un bosque, un grafo dirigido y fuertemente conexo produce un árbol, y un grafo dirigido y no fuertemente conexo produce un bosque (que puede tener un único árbol). El árbol se caracteriza por contener todos los vértices del grafo de partida junto con todas aquellas aristas que durante el recorrido del grafo cumplen la condición de que uno de sus extremos sea un vértice marcado a 'visto' y el otro extremo sea un vértice que aún no ha pasado a 'visto'.

```
función ARBOL-DFS ( g es grafo ) dev ( B es bosque )
{ Pre : g es un grafo no dirigido }
          Para cada v∈V  hacer  g:= Marcar( g, v, 'no_visto' ) fpara
          B := bosque_vacio;
          Para cada v∈V  hacer
          [    Visto( g, v ) --->   seguir
          []¬Visto( g, v ) --->   T := arbol_vacio ;
                                  < g, T > := TDFS-1( g, v, T );
                                  B:= añadir_arbol( B, T );
          ]
          fpara
{ Post : los vértices del grafo han sido visitados según el orden de recorrido en profundidad
∧ B es el bosque de árboles DFS que ha producido este recorrido}
          dev ( B )
ffunción
```

<u>función</u> TDFS-1 (g <u>es</u> grafo; u <u>es</u> vértice; T <u>es</u> árbol) <u>dev</u> (g <u>es</u> grafo; T <u>es</u> árbol)
{ *Pre :* ($u \in V$) $\wedge$ $\neg$Visto(g,u) $\wedge$ (g=g') $\wedge$ (T es el árbol asociado al recorrido en profundidad de g hasta este momento) $\wedge$ (T=T') }

```
        g:= Marcar( g, u, 'visto' );
        T:= añadir_vértice( T, u );                  /* PREWORK */
        Para cada w ∈ ady(g,u) hacer
        [    Visto( g, w ) --->   seguir
        [] ¬Visto( g, w ) --->   T:= añadir_vértice( T, w);
                                 T:= añadir_arista( T, u, w );
/* el POSTWORK se realiza antes de la llamada recursiva. El algoritmo añade a T aristas
que cumplen la condición de que uno de los extremos ya está marcado a 'visto' y el otro no.
El tipo árbol usado aquí soporta las inserciones repetidas de vértices */
                                 <g, T> := TDFS-1( g, w, T );
        ]
        fpara
```

{ *Post :* Visto(g,u) $\wedge$ marcados a visto todos los vértices de g accesibles desde u por caminos formados exclusivamente por vértices que no estaban vistos en g', según el orden de visita del recorrido en profundidad $\wedge$ T=T'$\cup${aristas visitadas después de hacer el recorrido en profundidad a partir del vértice u} }

<u>dev</u> (g, T)

<u>ffunción</u>

Conviene caracterizar formalmente el árbol asociado al DFS, al que se denominará T_{DFS}, y para ello se presentan las siguientes definiciones y lema.

Definición : Un vértice *v* es un antecesor del vértice *w* en un árbol *T* con raíz *r*, si *v* está en el único camino de *r* a *w* en *T*.

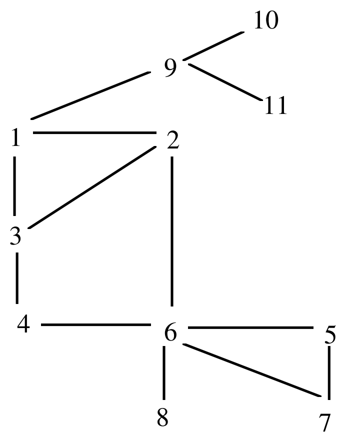
Definición : Si *v* es un antecesor de *w*, entonces *w* es un descendiente de *v*.

Lema : Sea G=(V,E) un grafo conexo no dirigido, sea T_{DFS}=(V,F) el árbol asociado al recorrido en profundidad de *G*. Entonces toda arista *e*, $e \in E$, o bien aparece en T_{DFS}, es decir, $e \in F$, o bien conecta dos vértices de *G* uno de los cuales es antecesor del otro en T_{DFS}.

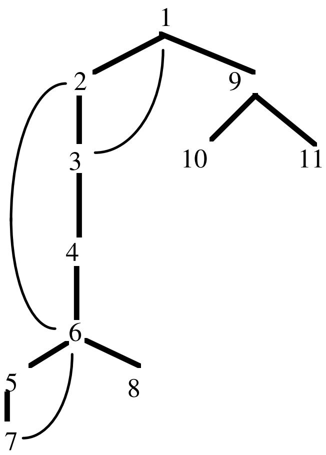
Este lema permite clasificar las aristas de *G* en dos clases una vez fijado un T_{DFS} :

- las aristas de *G* que aparecen en el T_{DFS} que son las *tree edges* o aristas del árbol. Estas aristas conectan padres con hijos en el árbol.
- las aristas de *G* que no aparecen en el T_{DFS} y que son las *back edges* o aristas de retroceso o de antecesores. Estas aristas conectan descendientes con antecesores en el árbol.

En la siguiente figura se muestra un grafo no dirigido y el T_{DFS} que se ha obtenido iniciando el recorrido de *G* en el vértice *1* y decidiendo el siguiente en orden numérico. Las aristas marcadas en negrita corresponden a las que forman parte del T_{DFS}, tree edges, mientras que las restantes son las back edges, es decir, aristas de *G* que no aparecen en el árbol y que conectan descendientes con antecesores.



Grafo



Tdfs (árbol)

Para un grafo G=(V,E) DIRIGIDO sus aristas se pueden clasificar en cuatro tipos en función del T_{DFS} que produce el recorrido en profundidad. Así se tienen :

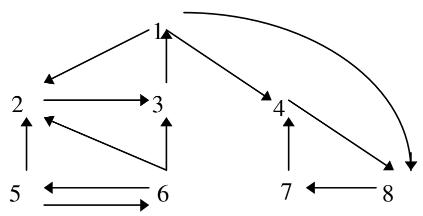
- tree edges y back edges con la misma definición que se ha dado para los grafos no dirigidos.

- *forward edges*, o aristas descendientes, que conectan antecesores con descendientes en el árbol.

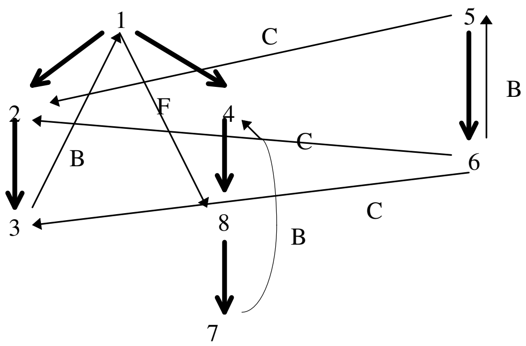
- *cross edges*, o aristas cruzadas, que conectan vértices no relacionados en el T_{DFS} . Estas aristas siempre van de derecha a izquierda en el árbol (o de izquierda a derecha, todo depende de cómo se dibuje).

La numeración en el orden del recorrido en profundidad de los vértices del grafo permite determinar el tipo de las aristas del grafo o las relaciones en el T_{DFS} entre los vértices de una arista. El siguiente lema es una muestra de ello.

Lema : Sea G=(V,E) un grafo dirigido y sea T_{DFS}=(V,F) el árbol asociado al recorrido en profundidad de *G*. Si *(v,w)*$\in$E y *v.num-dfs* < *w.num-dfs* entonces *w* es un descendiente de *v* en el árbol T_{DFS}.



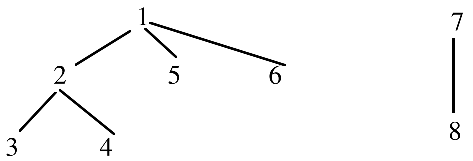
Grafo dirigido G = (V, E)



Aristas del Tdfs (negrita) junto con todas las demás de G convenientemente etiquetadas (B, C, F).

En la figura anterior se muestra un grafo dirigido y el árbol asociado al recorrido en profundidad que se ha iniciado en el vértice *1* y se ha aplicado orden numérico para elegir el siguiente vértice. Las aristas en negrita son las tree edges, es decir, las que forman el T_{DFS}. El resto están etiquetadas con *B*, *F* y *C* que significan back, forward y cross respectivamente.

Observando únicamente el árbol asociado al recorrido en profundidad se pueden determinar algunas de las características del grafo de partida. Por ejemplo, dado el siguiente árbol producido por el DFS de un grafo *G* :



se puede deducir que el grafo *G* no es conexo ni dirigido y que el vértice 5 no es adyacente ni al vértice 3 ni al 4 y que los vértices 3 y 4 no son adyacentes entre sí, etc.

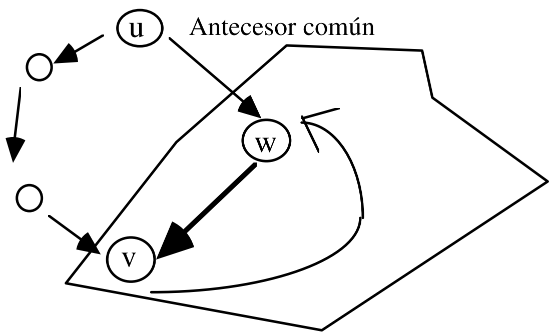
2.3.1.4. **Test de ciclicidad**

Una de las utilidades del árbol asociado al recorrido en profundidad, visto en la sección anterior, es que permite averiguar si el grafo, dirigido o no, contiene ciclos. El siguiente lema, para grafos dirigidos, así lo indica :

Lema : Sea G=(V,E) un grafo dirigido y T_{DFS} el árbol del recorrido en profundidad de *G*. *G* contiene un ciclo dirigido si y sólo si *G* contiene una arista de retroceso.

Demostración:

Sea *C* un ciclo en *G* y sea *v* el vértice de *C* con la numeración más baja en el recorrido en profundidad, es decir, *v.num-dfs* es menor que el *num-dfs* de cualquier otro vértice que forma parte de *C*. Sea *(w,v)* una de las aristas de *C*. ¿ qué clase de arista es ésta ?. Forzosamente *w.num-dfs* será mayor que *v.num-dfs* y esto hace que se descarte que la arista sea una tree edge o una forward edge. Podría tratarse solamente de una back edge o de una cross edge. Si fuera una arista cruzada significaría que *v* y *w* tienen un antecesor común *u*, con *u.num-dfs* menor que *v.num-dfs*, que es la única forma de conexión entre ellos, además de a través de la arista *(w,v)*. El gráfico siguiente ilustra esta situación.



Sin embargo, esto no es posible ya que *v* y *w* comparten un ciclo y, por tanto, existe camino entre *v* y *w* pero ¡ sólo atravesando vértices con una numeración mayor que la de *v* !
La única posibilidad que nos queda es que *(w,v)* sea un back edge, lo cual implica que si hay arista de retroceso es que hay ciclo.

Fin demostración.

El algoritmo que se presenta a continuación detecta la presencia de una arista de retroceso. El método que sigue es mantener, para el camino que va desde la raíz al vértice actual, qué vértices lo componen. Si un vértice aparece más de una vez en ese camino es que hay ciclo. Esta versión tiene el mismo coste que el algoritmo REC-PROF.

```
función CICLOS ( g es grafo ) dev ( b es bool )
{ Pre : g es un grafo dirigido }
          Para cada v∈V  hacer
                    g:= Marcar( g, v, 'no_visto' );
                    v.en-camino := FALSO ;  /* no hay ningún vértice en el camino de la raíz
          fpara
          b := FALSO;                        /* inicialmente no hay ciclos */
          Para cada v∈V  hacer
          [    Visto( g, v ) --->  seguir
          [] ¬Visto( g, v ) --->  <g,b1> := CICLOS-1( g, v );
                                  b := b1 ∨ b;
          ]
          fpara
{ Post : los vértices del grafo han sido recorridos según el orden de recorrido en profundidad
∧ b = Cíclico ( g ) }
          dev ( b )
ffunción
```

```
función  CICLOS-1 ( g es grafo; u es vértice ) dev ( g es grafo; b se bool )
{ Pre : (u∈V) ∧ ¬Visto(g,u)  ∧ (g=g')}
          g:= Marcar( g, u, 'visto' );
          u.en-camino := CIERTO;
/* PREWORK :Se anota que se ha visitado u y que éste se encuentra en el camino de la raíz
a él mismo*/
          b := FALSO;
          Para cada w ∈ suc(g,u) hacer
          [    Visto( g, w ) ---> [    w.en-camino  ---> b1 := CIERTO
                                                  /* ¡Ciclo !, se recorre la arista (u,w)
                                                  pero ya existía camino de w a u */
```

[] ¬w.en-camino ---> seguir
]
[] ¬Visto(g, w) ---> b1:= CICLOS-1 (g,w)
]
b := b1 ∨ b
<u>fpara</u>

/* ya se ha recorrido toda la descendencia de u y, por tanto, se abandona el camino actual desde la raíz (se va 'desmontando' el camino). La siguiente asignación corresponde al POSTWORK */

u.en-camino := FALSO

{ *Post :* Visto(g,u) ∧ marcados a visto todos los vértices de g accesibles desde u por caminos formados exclusivamente por vértices que no estaban vistos en g', según el orden de visita del recorrido en profundidad ∧ *b* dice si en el conjunto de caminos que tienen en común desde la raíz hasta u y luego se completan con la descendencia de u, hay ciclos.}

<u>dev</u> (g, b)

<u>ffunción</u>

Los grafos dirigidos y acíclicos, *directed acyclic graphs* DAG, se utilizan en muchos ámbitos. Por ejemplo, el plan de estudios de la Facultad es un DAG y las expresiones aritméticas con subexpresiones comunes que se pueden escribir en cualquier lenguaje de programación también deben ser un DAG.

Si en lugar de trabajar con un grafo dirigido se trabajara con uno no dirigido, el algoritmo anterior sería más simple. La existencia de un ciclo se detectaría cuando el recorrido a partir de los adyacentes se encontrara con un vértice que ya estuviera marcado a 'visto'. Para prevenir soluciones erróneas, sería necesario que cada llamada recibiera, además, el vértice que la ha provocado (el padre de la llamada).

Un problema interesante, y que se deja como ejercicio para el lector, es el siguiente : Dado un grafo no dirigido, G=(V,E), proponer un algoritmo que en tiempo O(n), n=|V|, determine si existen ciclos o no. Notar que el coste ha de ser independiente de |E|.

2.3.1.5. **Un TAD útil : MFSets**

El tipo de datos denominado *Merge-Find Sets*, MFSets o *Disjoint-Set Data Structure* (en [Fra 93] se denomina *Relación de equivalencia*) es el más adecuado para problemas del tipo 'determinar que elementos de un conjunto pertenecen a la misma clase de equivalencia respecto de la propiedad P'. Una aplicación inmediata sobre grafos es la de determinar las componentes conexas de un grafo no dirigido o averiguar si existe camino entre dos vértices dados.

En general, el objetivo de esta estructura es mantener una colección, $S=\{S_1, S_2, \ldots, S_n\}$, de conjuntos disjuntos de forma dinámica. Una posible signatura de esta estructura, suponiendo que los elementos que hay que clasificar son vértices, podría ser la siguiente :

<u>Universo</u> MFSet
<u>usa</u> conj_vértice, vértice
<u>género</u> mf
<u>operaciones</u>
    iniciar : ---> mf
    añade : mf, conj_vértice ---> mf
    merge : mf, conj_vértice, conj_vértice ---> mf
    find : mf, v ---> conj_vértice

La operación 'iniciar' crea la estructura vacía (colección vacía), 'añade' coloca un conjunto en la colección, 'find' indica en qué conjunto de la colección se encuentra el elemento y 'merge' fusiona dos conjuntos de la colección. Estas dos últimas operaciones son las más frecuentes durante la vida de la estructura, por eso todas las implementaciones van orientadas a rebajar su coste.

La primera implementación es muy simple. La colección se implementa sobre un vector de *n* posiciones, tantas como elementos haya que clasificar. El índice del vector identifica el elemento y el contenido del vector sobre ese índice contiene la etiqueta del conjunto al que pertenece (la etiqueta coincide con alguno de los elementos del conjunto). Inicialmente cada conjunto contiene un único elemento (se realizan *n* operaciones 'añade') y el índice y la etiqueta del conjunto coinciden. El siguiente algoritmo es una implementación de la operación 'merge'.

<u>función</u> MERGE (mf <u>es</u> vector[1.. n] <u>de</u> nats; a, b <u>es</u> nat) <u>dev</u> (mf <u>es</u> vector[1.. n] <u>de</u> nats)
{ *Pre :* a y b son las dos etiquetas de los conjuntos que hay que fusionar. En este caso, y como los elementos son naturales, las etiquetas también lo son }

```
        i := 0;
        *[ i < n --->  i := i+1;
                       [   mf [i] = a ---> seguir
                       []  mf [i] = b ---> mf [i] := a
                       ]
```

{ *Post :* todos los elementos entre 1 y n que pertenecían al conjunto con etiqueta b, ahora pertenecen al conjunto con etiqueta a (se ha fusionado a y b). Los demás elementos siguen en el mismo conjunto en el que estaban }
        <u>dev</u> (mf)
<u>ffunción</u>

Para esta implementación, el coste de la operación 'merge' es θ(n) mientras que el de la operación 'find' es O(1).

La segunda implementación es más astuta. Todos los elementos de un conjunto están en un árbol en el que sus elementos tienen puntero al padre. La colección se implementa sobre un vector de tamaño *n* (el número de elementos a clasificar) y cada componente contiene el puntero al padre del árbol al que pertenece. Un análisis simple de esta implementación nos permite observar que 'iniciar' + *n* operaciones 'añade' tiene un coste θ(n), 'merge' tiene coste θ(1) y 'find' O(n). No parece que se haya mejorado nada.
Sin embargo, se pueden utilizar dos técnicas que permiten reducir la suma del coste de todas las operaciones que se pueden realizar. La *unión por rango* es la primera técnica. Al aplicarla se consigue que la altura de los árboles esté acotada por la función *log n* y que, por tanto, 'find' cueste θ(log n) y 'merge' O(1). La segunda de ellas es la *compresión de caminos* y consiste en reducir a una unidad la distancia desde cualquier vértice del árbol a la raíz del mismo con lo que, la gran mayoría de las veces, la altura del árbol es menor que *log n*.

La aplicación conjunta de las dos técnicas hace que el coste de efectuar *m* operaciones 'find' y 'merge', $n \leq m$, sea $\theta(m \cdot \log^{*} n)$, donde $\log^{*} n$ es la función *logaritmo iterado de n* que tiene un crecimiento lentísimo. Esta función se define de la siguiente forma :

$\log^{*}1 = \log^{*}2 = 1$

$\forall n > 2, \quad \log^{*}n = 1 + \log^{*}(\lceil \log_2 n \rceil)$

Estos son algunos valores de referencia de esta función :

$\log^{*}4 = 1 + \log^{*}2 = 2$

$\log^{*}16 = 1 + \log^{*}4 = 3$

$\log^{*}60000 = 1 + \log^{*}16 = 4$

Como ya se ha mencionado al comienzo de la sección, esta estructura se puede utilizar para resolver el problema de calcular las componentes conexas de un grafo no dirigido. A continuación se muestra el algoritmo correspondiente. El número de 'find' y 'merge' que efectua es m=(n + e) y por tanto el coste de este algoritmo es $O((n+e) \cdot \log^{*} n)$

```
función CCC ( g es grafo ) dev ( m es MFSet )
{ Pre : g es un grafo no dirigido }
        m:= iniciar;
        Para cada v∈V  hacer
                g:= Marcar( g, v, 'no_visto' ) ;
                m:= añade( m, {v});
        fpara
        Para cada v∈V  hacer
        [     Visto( g, v ) ---> seguir
```

```
          [] ¬Visto( g, v ) ---> <g, m> := CCC-1( g, v, m );
          ]
          fpara
{ Post : m contiene todas las componentes conexas de g }
          dev ( m)
ffunción

función CCC-1 ( g es grafo; u es vértice; m es MFSet ) dev ( g es grafo; m es MFSet )
{ Pre : (u∈V) ∧ ¬Visto(g,u) ∧ (g=g') ∧ (m contiene una clasificación de todos los vértices
ya vistos en g)}
          g:= Marcar( g, u, 'visto' );      v1 := find( m, u );
          Para cada w ∈ ady(g,u) hacer
          v2 := find( m, w);
          [    Visto( g, w ) --->  [  v1 = v2 ---> seguir
                                   [] v1 ≠ v2 ---> m := merge( m, v1, v2 );
                                   ]
          [] ¬Visto( g, w ) ---> m := merge( m, v1, v2 )
                                   < g, m > := CCC-1( g,w, m)
          ]
          fpara
{ Post : Visto(g,u) ∧ marcados a vistos todos los vértices de g accesibles desde u por
caminos formados exclusivamente por vértices que no estaban vistos en g', según el orden de
visita del recorrido en profundidad. Además todos los vértices ya vistos están
convenientemente clasificados en m }
          dev ( g, m )
ffunción
```

2.3.2. RECORRIDO EN ANCHURA

El algoritmo de recorrido en anchura, en inglés *breadth-first search* y que a partir de ahora escribiremos BFS para abreviar, se caracteriza porque permite recorrer completamente el grafo. El BFS visita cada arista las mismas veces que lo hace el DFS. Lo que diferencia a ambos recorridos es el orden de visita de los vértices. En BFS el orden de recorrido es tal que los vértices se visitan según el orden primero en anchura o por niveles. Este orden implica que, fijado un vértice, primero se visitan los vértices que se encuentran a distancia mínima uno de él, luego los que se encuentran a distancia mínima dos, etc.

Para poner en marcha el recorrido se elige un vértice cualquiera como punto de partida. No importa si el grafo es dirigido o no, en cualquier caso el algoritmo acaba cuando todos los vértices han sido visitados.

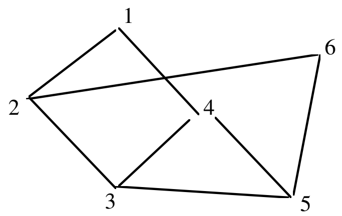
Una vez visto el recorrido en profundidad es sencillo obtener el algoritmo que efectúa el recorrido en anchura de un grafo. Es suficiente con obtener una versión iterativa del DFS y

sustituir la pila, que mantiene los vértices que aún tienen adyacentes por visitar, por una cola. En la cola se almacenan los vértices adyacentes al último visitado y que aún están sin visitar. A lo sumo habrá vértices pertenecientes a dos niveles consecutivos del grafo, los que se encuentran a distancia mínima *k* y a distancia mínima *k+1* del vértice a partir del cual se ha iniciado el recorrido. También es posible que existan vértices repetidos en la cola.Toda esta información debería aparecer en el invariante del bucle del algoritmo REC-ANCHO-1.
Un algoritmo que implementa el recorrido en anchura sobre un grafo no dirigido es el que se presenta a continuación (la versión para un grafo dirigido se deja como ejercicio):

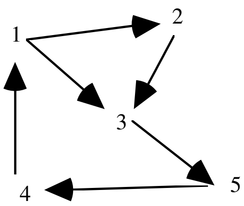
```
acción REC-ANCHO ( g es grafo )
{ Pre : g es un grafo no dirigido }
        Para cada v∈V  hacer  g:= Marcar( g, v, 'no_visto' ) fpara
        Para cada v∈V  hacer
        [    Visto( g, v ) ---> seguir
        [] ¬Visto( g, v ) ---> g:= REC-ANCHO-1( g, v )
        ]
        fpara
{ Post : Todos los vértices de g han sido marcados a 'visto' en el orden que sigue el recorrido
en anchura de g }
faccción

función  REC-ANCHO-1 ( g es grafo; u es vértice ) dev ( g es grafo )
{ Pre : (u∈V) ∧ ¬Visto(g,u)  ∧ (g=g')}
    c := cola_vacia;
    c := pedir_turno( c, u );
    * [¬vacia (c ) ---> u := primero( c );
                        c := avanzar( c);
                        [    Visto( g, u ) --->  seguir
                         []¬Visto( g, u ) --->  g:= Marcar( g, u, 'visto' );
                                                Para cada w ∈ ady(g,u) hacer
                                                [    Visto( g, w ) ---> seguir
                                                []¬Visto( g, w ) ---> c := pedir_turno( c, w )
                                                ]
                                                fpara
                        ]
      ]
{ Post : Visto(g,u) ∧ marcados a 'visto' todos los vértices de g accesibles desde u que no
estaban vistos en g' según el orden de visita del recorrido en anchura }
    dev ( g )
ffunción
```

El coste del algoritmo para un grafo implementado con listas de adyacencia es θ(n+e), el mismo que el DFS, aunque requiere más espacio debido a la cola de vértices que utiliza.

Existe un gran paralelismo entre el DFS y el BFS y casi todo lo visto para el DFS también es aplicable al BFS. Por ejemplo, algo bien simple en lo que coinciden : la secuencia de vértices producida por el recorrido en anchura no es única sino que depende de cual sea el vértice elegido para comenzar el recorrido y de cómo se vayan eligiendo los adyacentes. Más adelante se comentaran otras características comunes. La siguiente figura ilustra las distintas secuencias que puede producir el BFS. En la figura de la izquierda tenemos un grafo no dirigido y dos secuencias de su recorrido en anchura. En la primera secuencia se ha comenzado por el vértice *1* y en la segunda por el vértice *3*. La figura de la derecha muestra un grafo dirigido sobre el que se han llevado a cabo tres recorridos.



Recorrido 1 : 1, 2,4, 3, 6, 5
Recorrido 2 : 3, 2,4, 5, 1, 6



Recorrido 1 : 1, 2, 3, 5, 4
Recorrido 2 : 3, 5, 4, 1, 2
Recorrido 3 : 4, 1, 2, 3, 5

Se puede asociar un número a cada vértice del grafo según el orden de visita que produce el recorrido en anchura. También se puede obtener el árbol asociado al recorrido en anchura, T_{BFS}, que se construye igual que el T_{DFS}.

Algunas características destacables del T_{BFS} se citan a continuación :

Sea G=(V,E) un grafo y sea T_{BFS}=(V,F) el árbol asociado al recorrido en anchura de *G*.

Definición : se define la distancia más corta entre los vértices *s* y *v*, $\delta(s,v)$, como el mínimo número de aristas en cualquier camino entre *s* y *v* en *G*.

Lema : Sea $s \in V$ un vértice arbitrario de G, entonces para cualquier arista $(u,v) \in E$ sucede que $\delta(s,v) \leq \delta(s,u) + 1$

Lema : Para cada vértice *w*, la longitud del camino desde la raíz hasta *w* en T_{BFS} coincide con la longitud del camino más corto desde la raíz hasta *w* en G.

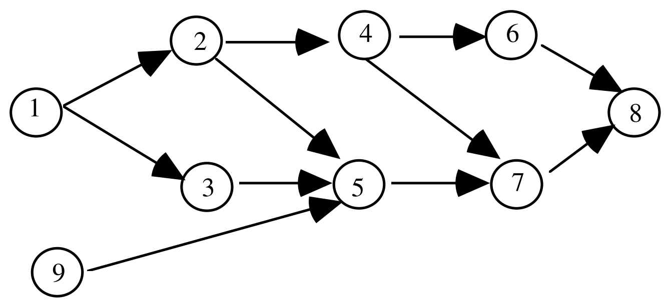
Hay que mencionar que el árbol asociado al recorrido en anchura, T_{BFS}, también permite clasificar las aristas del grafo de partida. Si se trata de un grafo no dirigido entonces existen aristas de dos tipos : tree edges y cross edges; estas últimas son las que permiten detectar la existencia de ciclos. Sin embargo, si se trata de un grafo dirigido, se tienen aristas de tres tipos : tree edges, back edges y cross edges con la particularidad de que éstas últimas pueden ir tanto de derecha a izquierda como de izquierda a derecha. No existen forward edges. Se deja como ejercicio para el lector el razonar por qué motivo no pueden existir aristas de este tipo.

Una de las aplicaciones clásicas del recorrido en anchura es en la resolución de problemas en los que se puede modelar cada situación del problema como un estado (vértice), existe una función que permite pasar de un estado a otro (aristas), y hay que encontrar el camino más corto, si existe, entre dos estados dados (que corresponden a 2 vértices del grafo). Esta aplicación es consecuencia inmediata de los lemas enunciados anteriormente, ya que ellos garantizan que el BFS llega a un vértice siempre por el camino más corto posible desde el vértice del que parte el recorrido.

2.3.3. ORDENACION TOPOLOGICA

El algoritmo de ordenación topológica, *Topological Sort*, genera una ordenación lineal de todos los vértices del grafo tal que si $(u,v) \in E$ entonces u aparece antes que v en la ordenación. Este algoritmo funciona sólo con grafos DIRIGIDOS y ACICLICOS. En realidad este algoritmo no es más que un caso particular del recorrido en profundidad.

Al igual que sucede con el recorrido en anchura o con el de profundidad, la ordenación topológica aplicada a un grafo *G* puede generar diferentes ordenaciones lineales. De nuevo, todo depende del vértice de partida y de cual se elija como siguiente vértice a visitar. Para el grafo que se presenta en la siguiente figura, resulta que una ordenación posible es O_1 = { 1, 2, 4, 3, 9, 5, 7, 6, 8 }, mientras que otra es O_2 = { 9, 1, 3, 2, 5, 4, 6, 7, 8 }. Existen más posibilidades además de las dos mencionadas.



Podemos aplicar dos aproximaciones distintas para diseñar el algoritmo de ordenación topológica. La primera de ellas se basa en el hecho de que un vértice sólo puede aparecer en la secuencia cuando todos sus predecesores han sido visitados y, a partir de ese momento, ya puede aparecer en cualquier lugar posterior. Un aplicación inmediata de esta idea conduce a llevar un contador para cada vértice de modo que indique cuántos de sus predecesores faltan por aparecer en la secuencia; en el momento en que ese contador sea cero, el vértice en cuestión ya puede aparecer.

La segunda aproximación es justo la lectura inversa de la primera : Un vértice ocupa un lugar definitivo en la secuencia cuando toda su descendencia ha sido visitada y siempre irá

delante de todos ellos, de hecho, esa es la posición más alejada del comienzo de la secuencia en la que puede aparecer, más lejos ya no es posible.
El siguiente algoritmo es el que se obtiene de la primera aproximación, el de la segunda versión es una aplicación inmediata del recorrido en profundidad.

```
función ORD-TOPO ( g es grafo ) dev ( s es secuencia ( vértices ) )
{ Pre : g es un grafo dirigido y sin ciclos }
        c := conj_vacio ;
        Para cada v∈V  hacer
                NP[v] := 0;      c := añadir( c, v );
        fpara
        Para cada v∈V  hacer
           Para cada w ∈ suc(g,v)  hacer
                NP [w] := NP [w] + 1;      c:= eliminar( c, w );
           fpara
        fpara
/* para cada v∈V, NP[v] contiene el número predecesores y c contiene sólo aquellos vértices
que tienen cero predecesores. El recorrido puede comenzar por cualquiera de ellos  */
        s := sec_vacia;
        *[ ¬vacio ( c ) ---> v := obtener_elem( c ) ;
                             c := eliminar( c, v );
                             s := concatenar( s, v );
                             Para cada w ∈ suc(g,v)  hacer
                                NP[w] := NP[w] -1;
                                [  NP[w] = 0 ---> c := añadir( c, w );
                                [] NP[w] ≠ 0 ---> seguir
                                ]
                             fpara
        ]
{ Post : s = orden_topológico( g ) }
        dev ( s )
ffunción
```

El coste de este algoritmo es el mismo que el del recorrido en profundidad para listas de adyacencia, es decir, $\theta(n+e)$.

3. ALGORITMOS VORACES : Greedy

3.1. CARACTERIZACION Y ESQUEMA

En este capítulo se va a presentar el esquema que se denomina método Voraz, *Greedy Method*, y los algoritmos obtenidos aplicando este esquema se denominan, por extensión, algoritmos Voraces. Este método, junto con el del gradiente y otros, forman parte de una familia mucho más amplia de algoritmos denominados de *Búsqueda local* o *Hill-Climbing*.

El conjunto de condiciones que deben satisfacer los problemas que se pueden resolver aplicando este esquema es un tanto variopinto y, a continuación, vamos a ennumerar algunas de ellas :

1/ El problema a resolver ha de ser de optimización y debe existir una función que es la que hay que minimizar o maximizar. Es la denominada *función objetivo*. La siguiente función que es lineal y multivariable, es una función objetivo típica.

$$f : N_x N_x \ldots_x N \text{ -------->} N \text{ (ó } R \text{)}$$

$$f(x_1,\ldots, x_n) = c_1\, x_1 + c_2\, x_2 + \ldots + c_n\, x_n$$

Se pretende encontrar una asociación de valores a todas las variables tal que el valor de *f* sea el óptimo. Es posible que el problema no tenga solución.

2/ Existe un conjunto de valores candidatos para cada una de las variables de la función objetivo. Ese conjunto de valores posibles para una variable es su *dominio*.

3/ Existe un conjunto de restricciones lineales que imponen condiciones a los valores que pueden tomar las variables de la función objetivo. El conjunto puede estar vacio o contener una única restricción. La siguiente expresión es una restricción habitual :

$$K \leq \sum_{i=1}^{n} x_i$$

4/ Existe una función, que en nuestra notación denominaremos función *solución*, que permite averiguar si un conjunto dado de asociaciones variable-valor es solución al problema. La asociación de un valor a una variable se denomina *decisión*.

5/ Y ya por último, existe una función que indica si el conjunto de decisiones tomadas hasta el momento viola o no las restricciones. Esta función recibe el nombre de *factible* y al conjunto de decisiones factibles tomadas hasta el momento se le suele denominar *solución en curso*.

Esta es una caracterización muy genérica y en temas posteriores se comprobará que algunas de sus características son compartidas por otros esquemas como Vuelta Atrás, Programación Dinámica, etc. ¿ Qué es lo que diferencia al método Voraz de estos otros esquemas ? : La diferencia fundamental estriba en el proceso de selección de las decisiones que forman parte de la solución. En un algoritmo Voraz este proceso es secuencial y a cada paso se determina, utilizando una función adicional que denominamos de *selección*, el valor de una de las variables de la función objetivo. A continuación 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 obtener la siguiente decisión. Esta es, por tanto, una técnica descendente.

En cada iteración del algoritmo la función de selección aplica un criterio fijo para escoger el valor candidato. Este criterio es tal que hace que la decisión sea localmente óptima, es decir, la decisión seleccionada logra que la función objetivo alcance el mejor valor posible (ningún otro valor de los disponibles para esa variable lograría que la función objetivo tuviera un valor mejor). Pero nunca se vuelve a reconsiderar ninguna de las decisones tomadas. Una vez que a una variable se le ha asignado un valor localmente óptimo, se comprueba si esa decisión junto con la solución en curso es un conjunto factible. Si es factible se añade la decisión a la solución en curso y se comprueba si ésta es o no solución para el problema. Ahora bien, si no es factible simplemente la última decisión tomada no se añade a la solución en curso y se continua el proceso aplicando la función de selección al conjunto de variables que todavía no tienen un valor asignado.

El esquema de un algoritmo Voraz que se presenta a continuación, refleja el proceso que se acaba de explicar y en él aparecen todas las funciones que se han descrito anteriormente.

<u>función</u> VORAZ (C <u>es</u> conjunto) <u>dev</u> (S <u>es</u> conjunto; v <u>es</u> valor)
{ *Pre :* C es el conjunto de valores candidatos 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 tienen asociado 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 := conjunto_vacio;              /* inicialmente la solución está vacía */
{ Inv : S es la solución en curso que no viola las restricciones y C contiene los candidatos que todavía no han sido elegidos }
    *[ ¬SOLUCION( S ) ∧ ¬vacio( C ) --->
          x := SELECCION( C );        /* se obtiene el candidato localmente óptimo */
          C := C-{x};                 /* este candidato no se va a considerar nunca más */
          [    FACTIBLE((S)∪{x}) ---> S := S∪{x};
```

```
            [] ¬ FACTIBLE((S)∪{x}) ---> seguir
            ]
    ]
    /* finaliza la iteración porque S es ya una solución o porque el conjunto de
    candidatos está vacio */
    [    SOLUCION( S ) ---> v:= OBJETIVO( S )
    []¬SOLUCION( S ) ---> S:= conjunto_vacio;
    ]
```

{ *Post :* (S=conj_vacio ⇒ no se ha encontrado solución) ∧ (S ≠conj_vacio ⇒ S contiene los candidatos elegidos ∧ 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 SELECCION y FACTIBLE ya que el resto de las operaciones del interior del bucle tienen coste constante. La función FACTIBLE, en la mayoría de los casos, sólo necesita tiempo constante, pero la función SELECCION ha de explorar el conjunto de candidatos y obtener el mejor en ese momento. Por eso depende del tamaño del conjunto de candidatos. Normalmente se prepara (preprocesa) el conjunto de candidatos antes de entrar en el bucle para rebajar el coste de la función de SELECCION y conseguir que éste sea lo más cercano posible a constante. Precisamente, debido a su bajo coste, los algoritmos Voraces resultan sumamente atractivos en aplicaciones reales.

El proceso de construcción de la solución y la función de selección asociada que se ha descrito produce un comportamiento de los algoritmos Voraces muy especial. Lo más característico es que esta forma de resolución del problema no garantiza que se consiga el óptimo global para la función objetivo, ni tan solo que se consiga obtener una solución cuando el problema sí la tiene. Las decisiones localmente óptimas no garantizan que se vaya a obtener la combinación de valores que optimiza el valor de la función objetivo (el óptimo global). La propia función de selección, es decir, el criterio que se aplica y el orden en que se toman las decisiones son las que provocan esta circunstancia.

En cualquier caso, a menos que se diga lo contrario, sólo se considera que el problema está resuelto cuando la secuencia de decisiones consigue el óptimo global. El esquema Voraz se utiliza para encontrar la mejor solución y no es suficiente con una solución que, por supuesto sin violar las restricciones, no sea óptima. En este contexto, si se utiliza un algoritmo Voraz para resolver un problema, habrá que demostrar que la solución obtenida es la óptima. La inducción es la técnica de demostración de optimalidad más utilizada.

La gran mayoría de problemas que aparecen en la bibliografía y que han sido resueltos aplicando un esquema Voraz satisfacen el denominado *Principio de Optimalidad* que se enunciará más adelante. Esto da una pista : si un problema satisface el principio de optimalidad tal vez sea resoluble aplicando un algoritmo Voraz, más exactamente, es posible que exista una función de selección que conduzca al óptimo global. Aunque se pueda utilizar esta información para intentarlo, no existe certeza y, por tanto, no hay garantía de que se vaya a encontrar tal función. En [CLR 92] se exponen algunos otros fundamentos teóricos del método Voraz tales como la estructura matroidal que exhiben ciertos problemas.

El *Principio de Optimalidad* se puede enunciar de diversas formas. Algunas de ellas se presentan a continuación :
[BB 90] : En una secuencia óptima de decisiones, toda subsecuencia ha de ser también óptima.
[CLR 92] : *A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems*.
[Peñ 86] : Un problema satisface el principio de optimalidad si una secuencia de decisiones óptima para él tiene la propiedad de que sea cual sea el estado inicial y la decisión inicial, el resto de las decisiones son una secuencia de decisiones óptima para el problema que resulta después de haber tomado la primera decisión.

Precisamente, el esquema de Programación Dinámica utiliza el Principio de Optimalidad para caracterizar los problemas que puede resolver.

3.2. 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 ellos. Se ha dejado fuera, por ejemplo, un problema muy típico como el de los códigos de Huffman del que puede obtenerse información consultando [HS 78] ó [BB 90] ó [CLR 92].

3.2.1. MONEDAS : EL PROBLEMA DEL CAMBIO

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

Este es un problema de minimización que satisface el principio de optimalidad. Se puede demostrar que si se tiene una solución óptima $A=\{e_1,e_2, ..., e_k\}$ para el problema de formar la cantidad *M*, siendo los e_i , $1 \leq i \leq k$, ejemplares de las monedas de *C*, entonces sucede que $A-\{e_1\}$ es también solución óptima para el problema de formar la cantidad $M-e_1$. En estas condiciones puede pensarse en un Voraz como esquema aplicable.

El proceso a seguir para cualquier problema que se intente resolver con un algoritmo Voraz siempre es el mismo. En primer lugar, determinar la función de selección, definir la función factible y definir qué es solución para ese 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 continua sin tener éxito es probable que haya que intentarlo con un esquema distinto.

Para el problema que nos ocupa se puede comenzar por definir *factible* y *solución*. En este caso, el problema no tiene restricciones y se tiene una solución cuando los ejemplares de 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. La función factible tendrá que comprobar que el valor de la moneda seleccionada junto con el valor de las monedas de la solución en curso 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.
Cubierta la primera parte del proceso, ahora hay que demostrar la optimalidad de este criterio y basta con un contraejemplo para comprobar que no conduce siempre al óptimo. Por ejemplo, si C={1,5,11,25,50} y M=65, la estrategia Voraz descrita obtiene la solución : una moneda de *50*, una moneda de *11* y cuatro monedas de *1*. Necesita seis monedas, mientras que la solución óptima sólo necesita cuatro monedas que son : una de *50* y tres de *5*.
No está todo perdido porque 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.

<u>función</u> MONEDAS (C <u>es</u> conjunto_monedas; M <u>es</u> natural) <u>dev</u> (S <u>es</u> conjunto_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 }

```
        L := ORDENAR_DECREC( C );        S := conjunto_vacio;
        /* se ordenan los ejemplares por orden decreciente de valor */
        *[ suma(S)<M ∧ ¬vacia(L) --->
                x:= primero(L);
```

```
                    /* x es un ejemplar de un tipo de moneda de las de C */
                    [  (suma(S) + x) ≤ M --->S:= S∪{x};
                                            /* es factible y se añade la decisión a la
                                            solución en curso */
                    [] (suma(S) + x) > M ---> L:= avanzar (L);
                                            /* no es factible. Ese tipo de moneda se
                                            elimina del conjunto de candidatos para no ser
                                            considerado nunca más */
                    ]
          ]
          [  suma(S)=M  ---> seguir
          [] suma(S)<M  ---> S :=conjunto_vacio
          ]
```

{ *Post :* si S vacio es que no hay solución, en caso contrario S contiene monedas tal que la suma de sus valores es exactamente M }

<u>dev</u> (S)

<u>ffunción</u>

En la postcondición no se dice nada de la optimalidad de la solución porque en la precondición no se dice nada sobre las características de *C*.

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.

3.2.2. MINIMIZAR TIEMPO DE ESPERA

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 están en el sistema.

Al igual que en el problema anterior, se trata de un problema de minimización que satisface el principio de optimalidad. No está sometido a restricciones y una solución es una secuencia de tamaño *n*, en la que aparecen los *n* procesos a atender. El orden de aparición en la secuencia indica 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. Se puede calcular la suma del tiempo que los procesos están en el sistema para cada una de los seis posibles órdenes en que se podrían atender :

<u>orden de atención</u>	<u>tiempo de espera</u>
p_1, p_2, p_3	5 + (5 + 10) + (5 + 10 + 3) = 38
p_1, p_3, p_2	5 + (5 + 3) + (5 + 3 + 10) = 31
p_2, p_1, p_3	10 + (10 + 5) + (10 + 5 + 3) = 43

p_2, p_3, p_1	$10 + (10 + 3) + (10 + 3 + 5) = 41$
p_3, p_1, p_2	$3 + (3 + 5) + (3 + 5 + 10) = 29$
p_3, p_2, p_1	$3 + (3 + 10) + (3 + 10 + 5) = 34$

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 este criterio conduce siempre al óptimo. Para ello se procede de la siguiente forma : dada una permutación cualquiera de *n* naturales (cada natural representa un proceso y tiene un tiempo de espera asociado) almacenada en una secuencia, comprobar que si en la secuencia aparece un proceso en una posición *x* con un tiempo de proceso mayor que el de otro proceso que aparece en una posición *y*, con $y>x$, entonces el tiempo de espera de esta secuencia se puede reducir a base de intercambiar esos dos elementos. Sólo cuando los procesos de la secuencia están ordenados por orden creciente de tiempo de proceso no hay forma de mejorar el tiempo de espera total. Más detalles en [BB 90].

El coste de esta solución es el de ordenar los procesos para producir la secuencia solución óptima, lo que se puede conseguir en $\theta(n\cdot\log n)$.

3.2.3. MAXIMIZAR NUMERO DE TAREAS EN EXCLUSION MUTUA

Dadas *n* actividades, A={1,2, ...,n}, que han de usar un recurso en exclusión mutua, y dado que cada actividad 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 viola la exclusión mutua.

Antes de buscar una función de selección veamos que el problema planteado satisface el principio de optimalidad (aunque no se demuestra). Sea *S* la solución óptima, de tamaño *m*, con $m\leq n$, para el conjunto de actividades $A=\{t_1,t_2, ...,t_n\}$. Sea a_k una actividad de *A* y $a_k \in S$. Si se elimina a_k de *S*, el problema original queda dividido en dos subproblemas : el de obtener el conjunto de tamaño máximo para todas aquellas actividades de *A* que acaban antes de que empiece a_k, y el de obtener el conjunto de tamaño máximo para todas aquellas otras actividades de *A* que empiezan después de que acabe a_k. Entonces, todas las actividades incluidas en *S* y que acaban antes de que empiece a_k son una solución óptima para el primer subproblema mientras que el resto de actividades de *S* son, también, una solución óptima para el segundo subproblema.
Existen varios criterios que pueden ser utilizados como función de selección :
1/ elegir la actividad que acaba antes de entre todas las que quedan

2/ elegir la actividad que empieza antes de entre todas.
3/ elegir la actividad de menor duración.
4/ elegir la actividad de mayor duración.
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 $f_i \leq s_j$.

Demostración :
Sea $S=<a_1,a_2,...,a_m>$ una lista que contiene una solución óptima para el conjunto de actividades $A=\{t_1,t_2, ...,t_n\}$ con $m\leq n$, tal que $f_{a1}\leq f_{a2}\leq...\leq f_{am}$.
1º/ Vamos a ver que toda solución óptima comienza con la actividad a_1, que es la que acaba antes de todas las de *A*. Precisamente es la que habría elegido el criterio Voraz.
- Si *S* comienza por a_1 no hay nada que comprobar.
- Si *S* NO comienza por a_1 lo hará por otra actividad, por ejemplo la a_k.
Supongamos que $B = S-\{a_k\}\cup\{a_1\}$. La sustitución de a_k por a_1 en *B* puede hacerse sin problemas ya que $f_{ak} \geq f_{a1}$ y, por tanto, todas las actividades de *B* que son compatibles con a_k también son compatibles con a_1. Como $|S|=|B|$, se puede asegurar que esta nueva solución también es óptima y comienza por a_1.

2º/ Una vez que se ha hecho la primera elección, la actividad a_1, el problema se reduce a encontrar una solución óptima para el problema de maximizar el conjunto de actividades que sean compatibles con a_1. Aplicando el primer criterio de selección volvemos a estar en la demostración anterior. Dicho de otra forma, si *S* es una solución óptima para el problema *A* entonces $S'= S-\{a_1\}$ es una solución óptima para el problema $A'=\{t_i\in A \mid s_{ti}\geq f_{a1}\}$. Si pudieramos encontrar una solución B' al problema A' tal que $|B'|>|S'|$ significaría que $|B'\cup\{a_1\}| > |S|$, lo que contradice la optimalidad de *S*. Por tanto, S' es una solución óptima para A' y el criterio de selección escogido lo construye correctamente.
Fin demostración.

El coste del algoritmo Voraz para resolver este problema es $\theta(n\cdot\log n)$ y se debe a la ordenación previa de las actividades. El bucle efectúa, en el peor de los casos, *n* iteraciones y el coste de cada iteración es $\theta(1)$.

3.3. 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 dado el objeto *i* con $1\leq i\leq n$, su valor es v_i y su peso es p_i. También se tiene una mochila capaz de soportar, como máximo, el peso PMAX. Determinar qué objetos hay que colocar en la mochila de modo que el valor total que se transporte 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 en ella.
Para ambas formulaciones el problema satisface el principio de optimalidad. Ahora bien, mochila fraccionada se puede resolver aplicando un Voraz mientras que, de momento, mochila entera no (se necesita un Vuelta Atrás y, si se introduce alguna otra restricción en el problema, es posible incluso aplicar 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 :

$$[\text{MAX}] \quad \sum_{i=1}^{n} x_i \cdot v_i, \quad \text{sujeto a} \quad \sum_{i=1}^{n} x_i \cdot p_i \leq \text{PMAX} \quad \text{y con} \quad 0 \leq x_i \leq 1$$

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. No 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 $X=(x_1,x_2, ...,x_n)$ 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, ...,1,*0.x*,0,...,0,0), es decir, unos cuantos 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 $\forall i : 1 \le i \le n : x_i=1$, seguro que es óptima ya que no es mejorable. Pero si X es de la forma $\exists i : 1 \le i \le n : x_i \ne 1$, entonces no se sabe si es óptima.
Sea j el índice más pequeño tal que $(\forall i : 1 \le i < j : x_i=1)$ y $(x_j \ne 1)$ y $(\forall i : j+1 \le i \le n : x_i=0)$. 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 satisfacen :

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

Sea k el índice más pequeño de Y tal que $x_k \ne y_k$. 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 $x_k=y_k$, 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 $Z=(z_1,z_2, ...,z_n)$ tal que $(\forall i : 1 \le i \le k : z_i=x_i)$ y

$$\sum_{i=k+1}^{n} p_i \cdot (y_i - z_i) = p_k \cdot (z_k - y_k)$$

Para Z se tiene :

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

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

Después de este proceso se llega a la siguiente desigualdad :

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

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 y que, a base de modificarla como se ha hecho en este proceso, se consigue una solución, T, idéntica a X y, por tanto, X también es óptima.

Fin demostración.

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 valor, peso */
          s:= 0;   VX := 0 ;  i :=1;
          *[ (i ≤ n) ∧ (s < PMAX) --->
                    k := VC[i].id;
                    [  s + p[k] ≤ PMAX ---> x[k] := 1 ; VX := VX + v[k] ;  s := s + p[k];
                    [] s + p[k] >PMAX --->  x[k] := (PMAX - s) / p[k] ;  s := PMAX;
                                             VX := VX + v[k] · x[k];
                    ]
                    i := i +1;
           ]
{ 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
```

3.4. ARBOLES DE EXPANSION MINIMA

Sea G=(V,E) un grafo no dirigido, etiquetado con valores naturales y conexo. Diseñar un algoritmo que calcule un subgrafo de *G*, T=(V,F) con F⊆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 el 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 *árbol de expansión mínima* de *G*, *minimun spaning tree* ó MST para abreviar.

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 de este algoritmo contiene la condición de que *A* siempre es un subconjunto del conjunto de aristas de *T*. En cada iteración la función SELECCIONAR_ARISTA_BUENA elige una arista que permite que se mantenga el invariante, es decir, elige una arista *(u,v)*∈E de modo que si *A* es un subconjunto de *T*, entonces A∪{(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 }
            *[ |A| < n-1  --->  (u,v) := SELECCIONAR_ARISTA_BUENA(g, A);
                                A := A∪{(u,v)}
                                w := w + valor (g, u, v) ;
             ]
{ 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
```

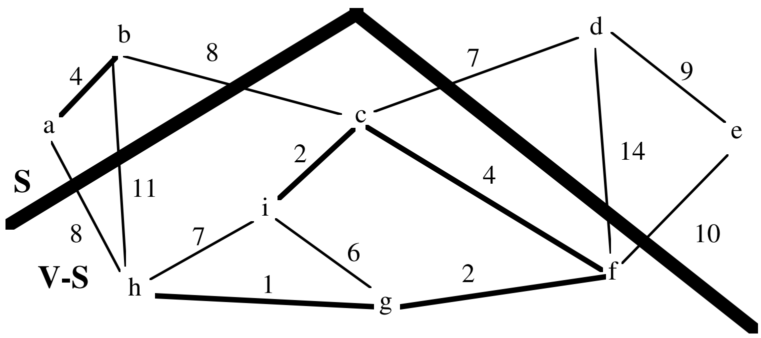
El problema es encontrar una arista buena para *A* de forma eficiente. Antes de abordar la cuestión de la eficiencia vamos con unas cuantas definiciones previas.

Definición : Un *corte* (S,V–S) de un grafo no dirigido G=(V,E) es una partición 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.



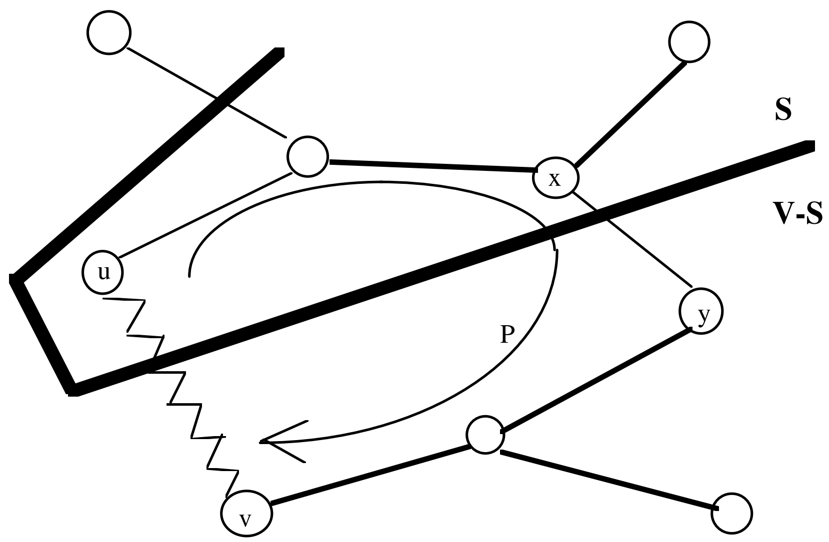
Las aristas en negrita pertenecen a A. El corte, marcado en trazo grueso, respeta a A. La arista (c,d) es una c_arista

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*.

Este teorema está proponiendo una función de selección consistente en elegir a cada paso aquella arista que es buena para *A*. Para ello lo que ha de hacer es fijar un corte que respete a *A* y devolver la arista de peso mínimo de entre todas las que crucen el corte. En realidad, la función de selección 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 son 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 la (u,v), forman parte de T. De éstas, las que están marcadas en negrita, pertenecen también a A. El corte está dibujado con trazo 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 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 :

$$\text{peso(T')} = \text{peso(T)} - \text{valor(g,x,y)} + \text{valor(g,u,v)}$$

y como *(u,v)* es una c_arista, entonces valor(g,u,v) $\leq$ valor(g,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*.

Fin demostración.

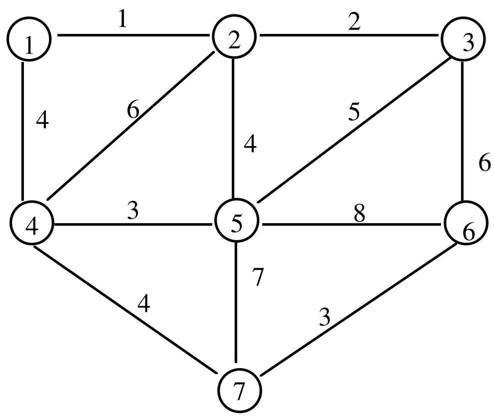
El invariante del bucle, además de contener que *A* es siempre un subconjunto de *T*, incluye el hecho de que el grafo G_A=(V,A) es un bosque y cada una de sus componentes conexas es un árbol libre. Se deduce del hecho de que *(u,v)* es una arista buena para *A* y por tanto esta arista conecta dos componentes 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 finalmente lograr una sola componente conexa. Prim impone que el conjunto de aristas contenga, de principio a fin, una sola componente conexa. Veámoslos.

3.4.1. 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.

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 que lleva a cabo el 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	{1, 2, 3, 4, 5, 6, 7}
	peso (T) = 17	

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.

```
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 */
        *[ | T | < n-1 --->
                (u,v) := primero(EO);
                EO := avanzar(EO);
                /* (u,v) es la arista seleccionada, es una arista buena para T */
                x := find( MF, u );
                y := find( MF, v );
```

```
                    [  x=y ---> seguir              /* se rechaza porque provoca ciclo */
                    [] x≠y ---> MF := merge( MF, x, y );    T := T∪{(u,v)}
                    ]
            ]
{Post : T es un MST de g }
          dev ( T )
ffunción
```

El coste de ordenar el conjunto de aristas es θ(|E|·log n) y el de inicializar el MFSet es θ(n). Cada iteración lleva a cabo dos operaciones 'find' y, algunas veces, exactamente *n-1*, también una operación 'merge'. El bucle, en el peor de los casos, lleva a cabo *2·|E|* operaciones 'find' y *n-1* operaciones 'merge'. Sea m=2·|E| + n-1, entonces el bucle cuesta O(m·log*n) que es O(|E|· log*n) aproximadamente. Como el coste del bucle es menor que el coste de las inicializaciones, el coste del algoritmo queda θ(|E|·log n).

3.4.2. PRIM

Este algoritmo 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). 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);
          *[ |VISTOS| < |V| --->
                    (u,v) := SELECCIONAR( g, VISTOS );
/* devuelve una arista (u,v)  de peso mínimio tal que u∈VISTOS y v∈(V−VISTOS) */
                    T:= T ∪{(u,v)};
                    VISTOS := VISTOS ∪{v};
          ]
{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 3.4.1. En este caso no es preciso ordenar previamente las aristas de G y 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	

Si se analiza 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−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−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 COSTMIN[v]=-1. Se supone que el grafo está implementado en la matriz de adyacencia M[1...n,1...n].

<u>función</u> PRIM_EFICIENTE (g <u>es</u> grafo) <u>dev</u> (T <u>es</u> 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 el vértice 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 */

```
        *[ |VISTOS| < n --->
        /* seleccionar la arista buena */
                min := ∞;
                Para  j=2 hasta n hacer
                [  0 ≤ COSTMIN[j] < min ---> min := COSTMIN[j];  k := j;
                [] (COSTMIN[j] = -1) ∨ (COSTMIN[j] > min) ---> seguir
                ]
                fpara
        /* la arista ( k, VECINO[k] ) es la arista seleccionada */
                T:= T ∪{(k, VECINO[k])};          /* arista que 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
                [   M[k,j] < COSTMIN[j] ---> COSTMIN[j] := M[k,j]; VECINO[j] := k;
                []  M[k,j] ≥ COSTMIN[j] ---> seguir
                ]
                fpara
        ]
{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 $\theta(n^2)$.
Existe una versión más eficiente que se puede encontrar en [CLR 92] y que mantiene los vértices no VISTOS en una cola de prioridad implementada en un *Heap*.

3.5. CAMINOS MINIMOS

Existe una colección de problemas, denominados problemas de caminos mínimos o *Shortest-paths problem,* basados en el concepto de camino mínimo. Antes de presentar la colección veamos algunos de estos conceptos.
Sea G=(V,E) un grafo dirigido y etiquetado con valores naturales. Se define el peso del camino *p*, $p=\langle v_0,v_1,v_2, ...,v_k\rangle$, como la suma de los valores de las aristas que lo componen.
Formalmente :

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

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

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

entonces el camino más corto, camino mínimo, de *u* a *v* en *G*, se define como cualquier camino *p* tal que peso(p) = δ(u,v).

La colección de problemas está compuesta por cuatro variantes :

1/ *Single_source shortest_paths problem* : Hay que encontrar el camino más corto entre un vértice fijado, *source*, y todos los vértices restantes del grafo. Este problema se resuelve eficientemente utilizando el algoritmo de Dijkstra.

2/ *Single_destination shortest_paths problem* : Hay que encontrar el camino más corto 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 más corto entre ellos. En el caso peor no hay un algoritmo mejor que el propio Dijkstra.

4/ *All_pairs shortest_paths problem* : Encontrar el camino más corto entre los vértices *u* y *v*, para todo par de vértices del grafo. Se resuelve aplicando Dijkstra a todos los vértices del grafo. El algoritmo de Floyd es más elegante pero no más eficiente y sigue el esquema de Programación Dinámica.

3.5.1. DIJKSTRA

El algoritmo de Dijkstra resuelve el problema de encontrar el camino más corto 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 más corto entre el vértice inicial y él. Para el resto de vértices, que se encuentran en V–VISTOS, se guarda la siguiente información : (∀u : u∈ V−VISTOS: D[u] contiene la longitud del camino más corto desde el vértice inicial a *u* que no sale de VISTOS), en realidad se dispone de esta información para todos los vértices. Una función de selección que eligiera el vértice *v* de V–VISTOS con *D[v]* mínima sería muy parecida a la que selecciona el vértice con el valor de COSTMIN más pequeño en Prim.

Dijkstra funciona exactamente de esta forma, pero necesita actualizar convenientemente los valores de *D* cada vez que se modifica VISTOS. Habrá que demostrar que si el vector *D* satisface estas condiciones, la función de selección propuesta obtiene la solución óptima, es decir, calcula los caminos mínimos entre el vértice inicial y todos los restantes. Veamos el algoritmo.

<u>función</u> DIJKSTRA (g <u>es</u> grafo; v_ini <u>es</u> vértice) <u>dev</u> (D <u>es</u> vector[1..n] <u>de</u> naturales)
{ *Pre :* g=(V,E) es un grafo etiquetado con valores naturales y puede ser dirigido o no dirigido. 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 arista que va del vértice i al j y, si no hay arista, contiene el valor infinito }

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

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

        *[|VISTOS| < |V| --->
                u := MINIMO(D, u∈V−VISTOS);
                /* se obtiene el vértice u∈V-VISTOS que tiene D mínima */
                VISTOS := VISTOS ∪{u};
                /* Ajustar D */
                <u>Para cada</u> v ∈ suc(g,u) tal que v ∈V−VISTOS <u>hacer</u> ajustar D[v] <u>fpara</u>;
         ]

{ *Post :* ∀u : u∈V: D[u] contiene la longitud del camino más corto 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>dev</u> (D)
<u>ffunción</u>

El bucle que inicializa el vector *D* cuesta θ(n) y el bucle principal, que calcula los caminos mínimos, efectua *n-1* iteraciones. En el coste de cada iteración intervienen dos factores : la obtención del mínimo, coste lineal, y el ajuste de *D*. Respecto a esta última operación, si el grafo está implementado en una matriz de adyacencia, su coste también es θ(n) y se tiene que el coste total del algoritmo es $θ(n^2)$.
Ahora bien, si el grafo está implementado en listas de adyacencia y se utiliza un Heap para acelerar la elección del mínimo, entonces, seleccionar el mínimo cuesta θ(1), la construcción del Heap θ(n·log n) y cada operación de ajuste requerirá θ(log n). Como en total se efectuan θ(|E|·log n) operaciones de ajustar D , el algoritmo requiere θ((n+|E|)·log n) que resulta mejor que el de las matrices cuando el grafo es poco denso.

Vamos a comprobar la corrección del criterio de selección, su optimalidad.

Demostración:

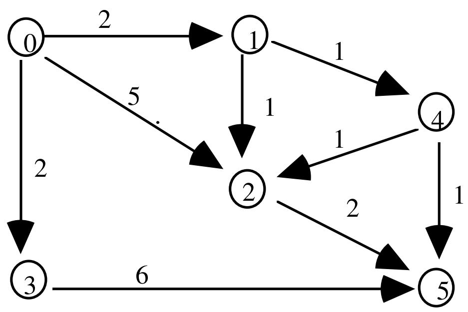
Sea *u* un vértice tal que *u*∈ V−VISTOS. Supongamos que *D[u]* contiene información cierta, es decir, la distancia mínima entre el vértice inicial y *u* siguiendo por un camino que sólo contiene vértices que pertenecen a VISTOS. Si *u* es el vértice con el valor de *D* más pequeño, el criterio de selección lo elegirá como candidato e inmediatamente pasará a formar parte de VISTOS y se considerará que su *D[u]* es una distancia definitiva.

Supongamos que no es CIERTO, es decir, que existe un camino más corto, aún no considerado, desde el vértice inicial a *u* que pasa por *v*, obviamente *v*∈ V−VISTOS. Si *v* se encuentra en el camino más corto desde el vértice inicial a *u*, ¡ es que D[v]<D[u] ! lo que contradice la elección de *u*. 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 (ninguno de los vértices que no están en VISTOS lograrán que se reduzca).

Fin demostración.

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 continuan 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. Para cada vértice *v*∈ suc(g,u) tal que *v*∈ V−VISTOS, hay que efectuar la siguiente comparación : *D[v]* comparado con (D[u] + valor(g,u,v)) y dependiendo del resultado de la comparación modificar *D[v]* convenientemente. Consultar [Bal 86] para más información.

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}

D :	0	1	2	3	4	5	**VISTOS**
	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}

3.5.2. RECONSTRUCCION DE CAMINOS MINIMOS

El algoritmo presentado calcula las distancias mínimas pero no mantiene la suficiente información como para saber cuales son los caminos que permiten alcanzarlos. Se puede modificar el algoritmo para almacenar la información necesaria y luego poder recuperar los caminos mínimos. Basta con anotar, cada vez que se modifica el valor de *D*, el 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 );
        *[ |VISTOS| < |V| --->
                u := MINIMO( D, u∈V−VISTOS );
                VISTOS := VISTOS∪{u};
                Para cada v ∈ suc(g,u) tal que v ∈V−VISTOS hacer
                [  D[v] > D[u] + valor( g,u,v ) ---> D[v] := D[u] + valor( g,u,v );
                                                     CAMINO[v]:= u;
                [] D[v] ≤ D[u] + valor ( g,u,v ) ---> seguir
                ]
                 fpara;
         ]
{ Post : ∀u : u∈V:  D[u] contiene la longitud del camino más corto 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, *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 mínimo hasta *v*.

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

4. ESQUEMA DE VUELTA ATRAS : Backtracking

4.1. CARACTERIZACION

En este capítulo nos vamos a ocupar del esquema de Vuelta Atrás, más conocido por su denominación inglesa de *Backtracking*. La caracterización de los problemas que son resolubles aplicando este esquema no es muy distinta de la que ya se ha visto en el capítulo anterior y se utiliza la misma terminología como, por ejemplo, decisión, restricción, solución, solución en curso, etc. Recordemos algunas de las características de estos problemas :

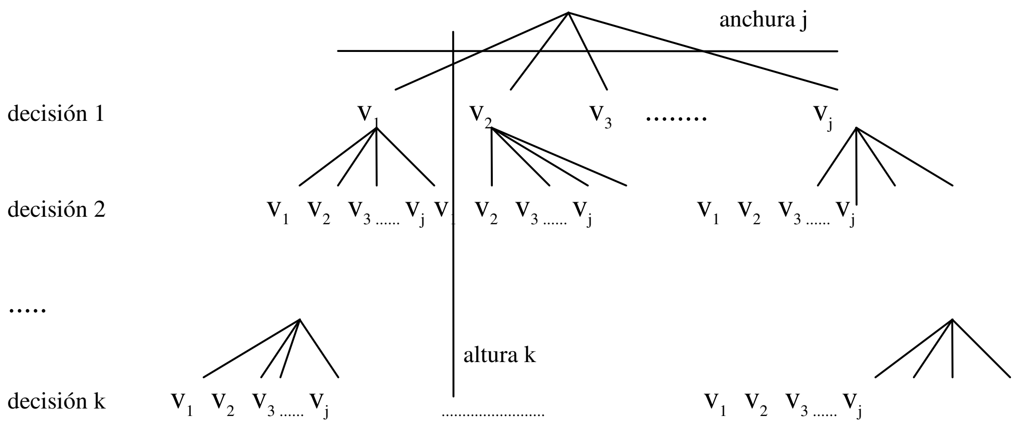
1/ se trata generalmente de problemas de optimización, con o sin restricciones.
2/ la solución es expresable en forma de secuencia de decisiones.
3/ existe una función denominada *factible* que permite averiguar si una secuencia de decisiones, la solución en curso actual, viola o no las restricciones.
4/ existe una función, denominada *solución*, que permite determinar si una secuencia de decisiones factible es solución al problema planteado.

No parece que exista una gran diferencia entre los problemas que se pueden resolver utilizando un esquema Voraz y los que se pueden resolver aplicando este nuevo esquema de Vuelta Atrás. Y es que en realidad no existe. La técnica de resolución que emplean cada uno de ellos es la gran diferencia.
Básicamente, Vuelta Atrás es un esquema que genera TODAS las secuencias posibles de decisiones. Esta tarea también la lleva a cabo el algoritmo conocido como de *fuerza bruta* pero lo hace de una forma menos eficiente. Vuelta Atrás, de forma sistemática y organizada, genera y recorre un espacio que contiene todas las posibles secuencias de decisiones. Este espacio se denomina el *espacio de búsqueda* del problema. Una de las primeras implicaciones de esta forma de resolver el problema es que, si el problema tiene solución, Vuelta Atrás seguro que la encuentra.

La siguiente figura muestra un posible espacio de búsqueda asociado a un problema en el que hay *k* decisiones que tomar (son las que definen la altura del espacio), y en el que cada decisión tiene asociado un dominio formado por *j* valores distintos, que son los que determinan la anchura del espacio. Habitualmente el espacio de búsqueda es un árbol, aunque puede ser un grafo, como en el caso de los grafos de juego. Si en el árbol hay un gran número de nodos repetidos y se dan otras condiciones adicionales, es posible que el problema se pueda resolver aplicando el esquema de Programación Dinámica.

El primer paso para resolver un problema utilizando Vuelta Atrás consiste en determinar el espacio de búsqueda asociado. Puede que exista más de un espacio para el mismo problema. Por regla general se elige el más pequeño o el de generación menos costosa, aunque un espacio dado no tiene porqué satisfacer ambos criterios simultáneamente. El espacio de búsqueda de la figura tiene j^k hojas y su número de nodos es

$n\_nodos = \sum_{i=0}^{k} j^i$ lo que determina que ocupe un espacio de orden $O(j^k)$, siendo *k* una variable que depende del tamaño de la entrada del problema. El tiempo necesario para recorrerlo es exactamente del mismo orden. El coste exponencial en el caso peor es otra de las características de Vuelta Atrás.



Todos los nodos que forman parte de cualquier camino que va desde la raíz del espacio de búsqueda a cualquier nodo del mismo representan una secuencia de decisiones. Como ya se ha mencionado anteriormente, una secuencia de decisiones es *factible* si no viola las restricciones. Se dice además que es *prolongable* si es posible añadir más decisiones a la secuencia y *no_prolongable* en caso contrario. Que una secuencia sea no_prolongable equivale a que ha llegado a la frontera del espacio de búsqueda, es decir, que el último nodo de la secuencia es una hoja del espacio. Para muchos problemas y dependiendo del espacio de búsqueda elegido se tiene que una *solución* es cualquier secuencia de decisiones factible y no_prolongable (topológicamente hablando, sólo cuando se está en una hoja se tiene una solución). En otros casos el concepto de solución es más amplio y cualquier secuencia factible, prolongable o no_prolongable, se considera solución.

Vuelta Atrás hace un recorrido en profundidad del espacio de búsqueda partiendo de la raíz del mismo. Precisamente la denominación de Vuelta Atrás procede de que el recorrido en profundidad regresa sobre sus pasos, retrocede, cada vez que encuentra un camino que se ha acabado o por el que no puede continuar. En un instante dado del recorrido, Vuelta Atrás se encuentra sobre un nodo *v* del espacio de búsqueda. La secuencia de decisiones factible formada por los nodos en el camino que va desde la raíz a *v* se denomina *solución en curso*, y *v* es el *nodo en curso*. Por abuso de lenguaje se dice que el nodo *v*, que es el último del camino, satisface una propiedad cuando debería decirse que el camino que va desde la raíz a *v* es el que satisface esa propiedad. Por ejemplo, si el camino de la raíz a *v* es factible, se dice que *v* es un *nodo factible*.

En un recorrido en profundidad o en un recorrido en anchura de un espacio de búsqueda se conoce de antemano el orden en que se van a generar, recorrer, sus nodos. Ambos son recorridos ciegos porque, independientemente del problema, fijado un nodo del espacio se sabe cual es el siguiente que se va a generar. En algunos textos el término Backtracking tiene un sentido más amplio que el que se le ha dado aquí y se utiliza para describir cualquier recorrido ciego.

4.2. TERMINOLOGIA Y ESQUEMAS

Vuelta Atrás puede recorrer el espacio de búsqueda con diferentes propósitos dependiendo del tipo de problema planteado. Así, para un problema de optimización, el objetivo del recorrido será encontrar una solución óptima, la mejor. Sin embargo, para un problema de búsqueda puede resultar interesante encontrar una solución, por ejemplo la primera que se encuentre durante el recorrido o, por el contrario, se puede querer obtener todas las soluciones existentes. A partir del esquema básico que encuentra una solución, e introduciendo pequeñas modificaciones, se puede obtener el esquema válido para cada una de las variantes propuestas.

El esquema algorítmico de Vuelta Atrás que se presenta implementa un recorrido en profundidad de forma recursiva. En él se asume que una solución es una secuencia de decisiones factible y no_prolongable. El algoritmo funciona de la siguiente forma : A la llamada recursiva le llega un nodo, que ya ha sido analizado en la llamada recursiva previa, y del que se sabe que es factible pero que todavía no es solución. Entonces el algoritmo genera todos los hijos de ese nodo pero sólo provocan nueva llamada recursiva aquellos hijos que a su vez sean factibles y no sean solución. Los hijos restantes, que o bien son solución o bien no son factibles, son tratados convenientemente en la misma llamada recursiva en la que han sido generados. Existe una versión algorítmica alternativa en la que a una llamada recursiva le llega un nodo del que no se sabe nada. Lo primero que hace es analizar el nodo (factible, prolongable, etc.) y caso de ser factible y no solución, genera todos sus hijos y cada uno de ellos produce nueva llamada recursiva. He adoptado la primera alternativa algorítmica porque a mi entender facilita el razonamiento y la argumentación de la corrección. Cuestión de gustos.
Las operaciones asociadas al espacio de búsqueda y que se utilizan para facilitar la lectura del algoritmo son : *preparar_recorrido*, *existan_hermanos* y *siguiente_hermano*. Todas ellas

asumen que la solución en curso, implementada sobre un vector *x*, contiene *k-1* decisiones almacenadas en las posiciones de la *1* a la *k-1* en el vector *x*. El valor de *k* indica la profundidad a la que se encuentra el recorrido. La decisión contenida en *x[k-1]* corresponde al último nodo tratado del espacio de búsqueda. Cada una de estas operaciones consiste en :
- *preparar_recorrido_nivel_k* : esta operación inicializa toda la información necesaria antes de empezar a generar y tratar todos los hijos de *x[k-1].*
- *existan_hermanos_nivel_k* : función que detecta si ya se han generado todos los hijos del nodo *x[k-1].*
- *siguiente_hermano_nivel_k* : función que produce el siguiente hijo aún no generado, en algún orden, del nodo *x[k-1].*

Cuando se decida utilizar este esquema para resolver un problema es conveniente definir antes todos los elementos básicos del algoritmo : el espacio de búsqueda, la solución, las funciones de generación y las de recorrido del espacio de búsqueda.
- *Espacio de búsqueda* : su altura, anchura, que aspecto tiene (árbol, grafo, etc.).
- Hay que decidir cómo va a expresarse la *solución* : su tamaño (fijo o variable), cómo se va a implementar el tipo solución, que condiciones determinan que una secuencia de decisiones sea solución para el problema, etc.

El espacio de búsqueda y la solución son dos elementos que están completamente relacionados. Dependiendo de qué espacio de búsqueda se elija (su tamaño, forma, el orden de las decisiones), la solución se expresará de una cierta forma o de otra. Y también al revés, dependiendo de cómo se quiera expresar la solución (con una estructura de tamaño fijo o variable), el espacio de búsqueda tendrá un aspecto u otro.

Por una pura cuestión de eficiencia es conveniente que el espacio de búsqueda no contenga soluciones repetidas. Esto es fundamental para aquellos problemas en los que no importa en que orden se toman las decisiones que aparecen en la solución. Las mismas decisiones en cualquier orden están describiendo la misma solución y, si no se tiene en cuenta, Vuelta Atrás generará soluciones repetidas.
- *Funciones de generación* : para el espacio de búsqueda elegido diseñar las funciones preparar_recorrido, existe_hermano y siguiente_hermano.
- *Funciones de recorrido* : una de ellas ya se ha definido cuando se ha caracterizado la solución. Además hay que definir la función *factible* y la función *prolongable*. De su buena definición depende que los caminos explorados sean válidos, es decir, no violen las restricciones y no salgan fuera de los límites del espacio de búsqueda.

4.2.1. UNA SOLUCIÓN

Este es el esquema que obtiene UNA solución, la primera que encuentra. El parámetro de entrada *x*, de tipo solución, contiene precisamente la solución en curso y se ha implementado sobre un vector para facilitar la escritura del algoritmo. Conviene recordar que se ha asumido que cuando *la solución en curso es factible pero no_prolongable se tiene una solución.*

Entonces, la secuencia de decisiones asociada a esa solución corresponde a un camino que va desde la raíz a una hoja del espacio de búsqueda. En el apartado 4.2.4.Mochila Entera se interpreta *solución* de forma distinta.

```
función BACK_1SOL ( x es solución; k es nat ) dev ( b es bool; sol es solución )
{ Pre : (x [1..k-1] es factible ∧ no es solución) ∧  1≤ k ≤ altura(espacio búsqueda) }
    b:= FALSO; sol:= sol_vacia;
    preparar_recorrido_nivel_k;
    *[ existan_hermanos_nivel_k  ∧ ¬b --->
          x[k] := sig_hermano_nivel_k;
          [   factible(x,k) --->   [   solucion(x,k)  --- >  tratar_solución(x);
                                                             sol := x;  b := CIERTO;
                                   [] ¬solucion(x,k) --- >  <sol,b> := BACK_1SOL(x, k+1);
                                   ]
          [] ¬factible(x,k) ---> seguir
          ]
    ]
```

{ *Post :* si b es FALSO quiere decir que dado el prefijo x[1...k-1] se han generado todas las formas posibles de rellenar x desde k hasta longitud(solución) y no se ha encontrado solución. Si b es CIERTO quiere decir que durante ese proceso de generación de todas las formas posibles de rellenar x desde k hasta longitud(solución) se ha encontrado una solución al problema, sol. La generación de todas las formas posibles de rellenar x desde k hasta longitud(solución), se ha hecho siguiendo el orden de 'primero en profundidad' }

```
    dev ( sol, b )
ffunción
```

La primera llamada a esta función es **<s,b>:= BACK_1SOL (sol_vacia, 1).**

4.2.2. TODAS LAS SOLUCIONES

Este es el esquema que obtiene TODAS las soluciones. El conjunto de soluciones se devuelve en la secuencia *s*.

```
función BACK_TODAS ( x es solución; k es nat ) dev ( s es secuencia(solución ))
{ Pre : (x [1..k-1] es factible ∧ no es solucion ) ∧  1≤ k ≤ altura(espacio búsqueda) }
    s:= sec_vacia;       preparar_recorrido_nivel_k ;
    *[ existan_hermanos_nivel_k   --->
          x[k] := sig_hermano_nivel_k
          [    factible(x,k) ---> [   solucion(x,k)  ---> tratar_solución(x);     s := añadir(s, x)
                                  [] ¬solucion(x,k) ---> s1 := BACK_TODAS(x, k+1);
                                                         s:=  concat(s1,s )
                                  ]
```

```
            [] ¬factible(x,k) ----> seguir
            ]
      ]
```

{ *Post :* Dado el prefijo x[1...k-1] se han generado todas las formas posibles de rellenar x desde k hasta longitud(solución) y s contiene todas las soluciones que se han encontrado. La generación de todas las formas posibles de rellenar x desde k hasta longitud(solución), se ha hecho siguiendo el orden de 'primero en profundidad' }

```
      dev ( s )
ffunción
```

La primera llamada a esta función es **sec := BACK_TODAS (sol_vacia, 1).**

4.2.3. LA MEJOR SOLUCION

Este es el esquema que obtiene la mejor solución para un problema de MAXIMIZACION. , Después de recorrer todo el espacio de búsqueda, devuelve la mejor solución encontrada en *xmejor*. El valor de la solución óptima se devuelve en *vmejor*.

```
función BACK_MEJOR ( x es solución; k es nat ) dev
                              ( xmejor es solucion;  vmejor es valor (xmejor) )
{ Pre : (x [1..k-1] es factible ∧ no es solución ) ∧ 1≤ k ≤ altura(espacio búsqueda) }
      <xmejor,vmejor> := <sol_vacia, 0 > ;        preparar_recorrido_nivel_k ;
      *[ existan_hermanos_nivel_k   --->
            x[k] := sig_hermano_nivel_k;
            [     factible(x,k) ---> [     solucion(x,k) --->    tratar_solución(x) ;
                                                                  <sol,val> := < x, valor(x) >
                                     [] ¬solucion (x,k) --->  <sol,val> := BACK_MEJOR(x, k+1)
                                     ]
                                     [  vmejor ≥ val --->       seguir
                                     [] vmejor < val --->       < xmejor, vmejor > := < sol, val >;
                                     ]
            [] ¬factible(x,k) ---> seguir
            ]
      ]
```

{ *Post :* Dado el prefijo x[1...k-1] se han generado todas las formas posibles de rellenar x desde k hasta longitud(solución) y xmejor es la mejor solución que se ha encontrado en el subárbol cuya raíz es x[k-1]. El valor de xmejor es vmejor. La generación de todas las formas posibles de rellenar x desde k hasta longitud(solución), se ha hecho siguiendo el orden de 'primero en profundidad' }

```
      dev ( xmejor, vmejor )
ffunción
```

La primera llamada a esta función es **<xm,vm> := BACK_MEJOR (sol_vacia, 1).**

4.2.4. MOCHILA ENTERA

El problema de la Mochila entera se ha formulado en el capítulo 3 (Algoritmos Voraces). Allí se dice que no existe una solución Voraz para este problema. La formulación ya utilizada es :

$$[\text{MAX}]\ \sum_{i=1}^{n} x(i)\cdot v(i), \quad \text{sujeto a } \left(\sum_{i=1}^{n} x(i)\cdot p(i) \right) \leq \text{PMAX} \ \wedge \left(\forall i : 1 \leq i \leq n : x(i) \in \{0,1\} \right)$$

Se supone que *v* y *p* son vectores de *1* a *n* que contienen el *valor* y el *peso*, respectivamente, de los n objetos que hay que intentar empaquetar en la mochila. Como máximo, la mochila soporta un peso de PMAX. Se trata de un problema de optimización y se quiere obtener la solución que tenga el máximo valor y que satisfaga la restricción de peso.

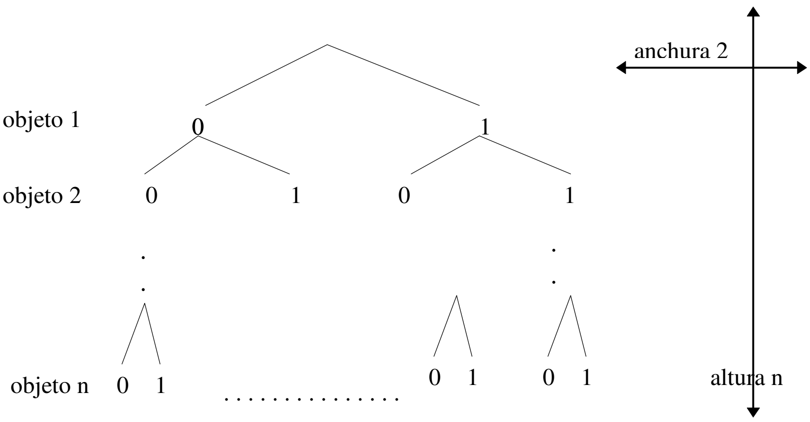
Alternativa A : **Espacio de búsqueda binario** y **solución ≡ factible ∧ no_prolongable**

Hay que determinar cómo es el espacio de búsqueda y la solución. Vista la formulación anterior de mochila, la solución podría expresarse como una secuencia que se implementa en un vector de tamaño *n* y que contiene los valores *{0 ,1}*.

Concretando,   tipo solución es vector [1..n] de {0,1};
                    var x : solución ;

∀i : 1≤i≤n : (x[i]=0 indica que el objeto *i* NO está en la mochila)
              (x[i]=1 indica que el objeto *i* SI está en la mochila)

La solución expresada de esta forma es una solución de *tamaño fijo*, exactamente contiene *n* decisiones, una para cada objeto. El espacio de búsqueda asociado es un árbol binario de altura *n* y que ocupa un espacio de $O(2^n)$.



La función factible ha de comprobar que la suma del peso de los objetos colocados hasta el momento en la mochila, no supera el máximo permitido. Se tiene una solución cuando se ha decidido qué hacer con los *n* objetos, lo que corresponde a estar en una hoja del espacio de búsqueda. Por tanto, cualquier camino desde la raíz a una hoja que no supere el peso máximo de la mochila puede ser solución óptima del problema.

A continuación se presenta un algoritmo para implementar esta alternativa. Se ha utilizado el bucle para generar los dos hijos de cada uno de los nodos del espacio de búsqueda con el propósito de ilustrar la utilización del esquema, aunque para este problema resulta excesivo y se podía haber resulto de otra forma.

```
función MOCHILA-BINARIO (x es solución; k es nat) dev
                              ( xmejor es solución; vmejor es valor (xmejor))
         k-1
{ Pre :  (Σ x(i)·p(i) ≤ PMAX)  ∧ (1≤k≤ n) }
         i=1
    <xmejor,vmejor>:=<sol_vacia, 0>;
    x(k):= -1;                                    /* preparar_recorrido de nivel k */
    *[ x(k) < 1 ----> x(k):= x(k) +1 ;             /* siguiente hermano nivel k */
                  peso := SUMA_PESO(x,1,k,p);
                  [ peso ≤ PMAX --- >             /* la solución en curso es factible */
                          [ k=n  --->             /* es una hoja y por tanto solución */
                               <sol,val> := < x, SUMA_VALOR(x,1,n,v) >
                          []k<n --->              /* no es una hoja, es prolongable  */
                               <sol,val> := MOCHILA-BINARIO( x, k+1);
                          ]
                          [ vmejor ≥ val ---> seguir
                          [] vmejor < val ---> < xmejor, vmejor > := < sol, val >;
                          ]
                  [] peso >PMAX ---> seguir;      /* viola las restricciones */
                  ]
    ]
{ Post : xmejor es la mejor solución que se ha encontrado explorando todo el subárbol que
cuelga del nodo x(k-1), estando ya establecido el camino desde la raíz a este nodo, y
          n
vmejor = Σ xmejor(i)·v(i) }
          i=1
    dev ( xmejor, vmejor )
ffunción
```

La Primera llamada a la función será **<s,v> := MOCHILA-BINARIO (sol_vacia, 1).**

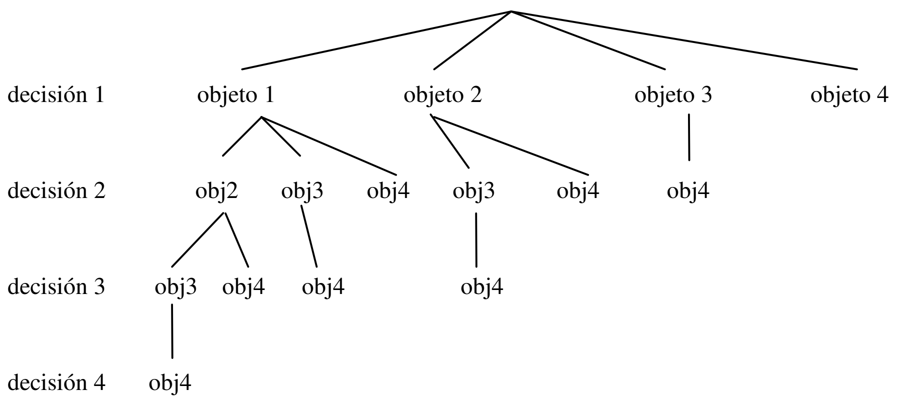
El coste de esta solución es $O(2^n)$ debido al tamaño máximo del espacio de búsqueda. Este coste ha de ser multiplicado por : el de calcular el valor de la variable *peso* en cada nodo del espacio de búsqueda, y el de calcular el valor de la variable *val* sólo en las hojas. Las funciones SUMA_PESO y SUMA_VALOR calculan, respectivamente, ambos valores.
Ambos cálculos requieren coste $\theta(n)$, con lo que el coste total del algoritmo es $O(n \cdot 2^n)$.

Alternativa B : **Espacio de búsqueda n-ario** y **solución ≡ factible**

La solución también se puede plantear como una secuencia que contiene los índices de objetos que se han podido empaquetar en la mochila sin violar la restricción de peso. Eso significa que la secuencia tendrá un tamaño variable que dependerá del número de objetos que se hayan incluido en la solución. Se puede deducir que como mínimo habrá un elemento y como máximo los *n* de la entrada.

Para esta solución, el espacio de búsqueda ha de contener todas las combinaciones de *n* elementos tomados de *1* en *1*, de *2* en *2*,..., de *n* en *n*. El tamaño del espacio sigue siendo del orden de $O(2^n)$.

La siguiente figura muestra el espacio de búsqueda para un problema con n=4. El espacio es de tamaño mínimo porque no se repite ninguna combinación, aunque sea en orden distinto. Tampoco se intenta colocar el mismo objeto más de una vez.



Existen bastantes diferencias entre este espacio de búsqueda y el binario de la alternativa *A*. En éste una solución se puede encontrar en cualquier camino que vaya de la raíz a cualquier otro nodo del árbol y, además, las hojas se encuentran a distintas profundidades.

Se puede detectar si la solución en curso es una solución para el problema inicial de dos formas distintas :

1/ cualquier camino que vaya desde la raíz a cualquier otro nodo del árbol y que no viole la restricción de peso máximo es solución.

2/ cuando se alcanza un nodo que viola la restricción de peso máximo, se puede asegurar que el camino desde la raíz hasta el padre de ese nodo es una solución.

En el algoritmo que se presenta a continuación se ha elegido la primera posibilidad por ser la más homogénea con el tipo de tratamiento que se está haciendo hasta ahora. La definición del tipo solución que se ha utilizado es :

<u>tipo</u> solución <u>es</u> vector [0..n] de {0..n};
<u>var</u> x : solución ;

Los valores de *x* van de *0* a *n* y tienen la siguiente interpretación : x(i)=j indica que en *i*-ésimo lugar se ha colocado en la mochila el objeto identificado por el índice *j*. Si x(i)=0 indica que en *i*-ésimo lugar no se ha colocado en la mochila ningún objeto. Inicialmente x(0)=0. El algoritmo devuelve en *t* el número de objetos que forman parte de la solución. El coste de este algoritmo es el mismo que el obtenido para la alternativa anterior.

```
función MOCHILA-NARIO ( x es solución; k es nat) dev
                                  ( xmejor es solución; t es nat; vmejor es valor (xmejor))
        k-1
{ Pre : ( Σ p(x(i)) ≤ PMAX )  ∧ (1≤k≤ n) }
        i=1
    <xmejor,t,vmejor>:=<sol_vacia, 0, 0>;
    x(k):= x(k-1);                              /* preparar recorrido nivel k */
    *[ x(k) < n ----> x(k):= x(k) +1 ;
                 peso := SUMA_PESO(x,1,k,p);
                 [ peso ≤ PMAX --- >             /* la solución en curso es factible y
                                                 también solución para el problema */
                       val := SUMA_VALOR(x,1,k,v);
                       [ vmejor ≥ val ---> seguir
                       [] vmejor < val ---> <xmejor ,vmejor> := <x ,val>;
                                             t := k;
                       ]
                       [ x (k) = n  ---> seguir; /* es una hoja y , por tanto, solución
                                                 pero ya se ha tratado como tal en la
                                                 alternativa anterior */
                       [] x(k) < n  --->         /* no es hoja, hay que prolongar */
                             <sol, tt, val > := MOCHILA-NARIO( x, k+1);
                             [ vmejor ≥ val ---> seguir
                             [] vmejor < val ---> <xmejor ,vmejor> := <sol ,val>;
                                                   t := tt;
                             ]
                       ]
                 [] peso >PMAX ---> seguir;      /* viola las restricciones */
                 ]
    ]
{ Post : xmejor[1..t] contiene los índices de los objetos tal que la suma de sus valores es
vmejor. xmejor es la mejor solución que se ha encontrado explorando todo el subárbol que
cuelga del nodo x(k-1) estando ya establecido el camino desde la raíz a este nodo}
    dev ( xmejor, t, vmejor )
ffunción
```

La primera llamada será **<s,t,v> := MOCHILA-NARIO (sol_vacia, 1).**

4.3. MARCAJE

La técnica que se presenta a continuación ya se ha introducido en otras asignaturas de Programación. Se trata de la conocida *inmersión de eficiencia* que, si se recuerda, puede ser de parámetros o de resultados. En el contexto de Vuelta Atrás la inmersión se denomina *marcaje* y suele ser de parámetros. Su objetivo es reducir el coste de los algoritmos. Basta con observar las dos soluciones que se han presentado de Mochila entera en el apartado anterior para apreciar que es posible reducir el coste de $O(n \cdot 2^n)$ a $O(2^n)$. Es suficiente con no tener que calcular, para cada nodo del espacio, la suma del peso de los objetos que se encuentran en la solución en curso asociada. Si un nodo le comunicara a todos sus hijos el peso que él ha conseguido acumular hasta el momento, se conseguiría evitar ese cálculo. Más adelante se presentará el algoritmo de Mochila entera, alternativa *A*, que incorpora esta inmersión de eficiencia.

Para mostrar la utilización que hace Vuelta Atrás del marcaje se ha partido del esquema que obtiene todas las soluciones y se han introducido las operaciones de *marcar* y *desmarcar*, que son las que permiten manipular el marcaje, así como los puntos en que se realizan habitualmente esas operaciones.

```
función BACK_TODAS_MARCAJE ( x es solución; k es nat; m es marcaje ) dev
                                        (s es secuencia(solución))
{ Pre : (x [1..k-1] es factible ∧ no es solución ) ∧ 1≤k≤altura(espacio búsqueda) ∧ m
contiene el marcaje del nodo x[k-1]}
   s:= sec_vacia;    preparar_recorrido_nivel_k ;
   *[ existan_hermanos_nivel_k   --->
        x[k] := sig_hermano_nivel_k;
        m:= MARCAR ( m, x, k );  [ 1a ]
        [ factible(x, k) --->
              [ 1b ]
              [    solucion (x,k) ---> tratar_solucion(x);   s := añadir(s, x);
              []¬solucion (x,k) ---> s1:= BACK_TODAS_MARCAJE(x, k+1, m);
                                     s :=  concat(s1,s );
              ]
              [ 2b ]
        [] ¬factible(x,k) ----> seguir
        ]
        m:= DESMARCAR ( m, x, k ); [ 2a ]
   ]
{ Post : Dado el prefijo x[1...k-1] se han generado todas las formas posibles de rellenar x
desde k hasta longitud(solución) y s contiene todas las soluciones que se  han encontrado. La
```

generación de todas las formas posibles de rellenar x desde k hasta longitud(solución), se ha hecho siguiendo el orden de 'primero en profundidad' }

dev (s)

ffunción

La función BACK_TODAS_MARCAJE recibe como parámetro el marcaje *m*, que contiene toda la información asociada al camino que va desde la raíz hasta el nodo *x(k-1)* y que necesitarán sus descendientes si no quieren tener que volver a calcular esa información y quieren mejorar la eficiencia. En **[1a]** el nodo *x(k)* ya es alguno de los hijos del nodo *x(k-1)* y lo que hace la operación MARCAR es asociar a *x(k)* toda la información del padre, que está en *m*, junto con la nueva que él aporta. El marcaje de *x(k)* se almacena en la misma variable que contenía el marcaje de su padre, es decir, en *m*. En **[2a]** se vuelve a recuperar el valor que tenía *m* como parámetro de entrada para que la próxima vez que se repita **[1a]** se calcule el marcaje del nuevo hijo correctamente y siempre respecto del marcaje del padre.

Hay ocasiones en las que conviene MARCAR y DESMARCAR en **[1b]** y **[2b],** respectivamente, en lugar de hacerlo en **[1a]** y **[2a]**. Todo depende del problema. También dependiendo del problema y de cómo se implemente y manipule el marcaje, a veces no será necesario DESMARCAR.

Y este es el algoritmo de Mochila entera, alternativa *A*, pero con marcaje. En este caso el marcaje está formado por dos variables *pac* y *vac* que contienen, respectivamente, el peso y el valor acumulado por la solución en curso. Interesa destacar que no es preciso DESMARCAR debido a que el espacio de búsqueda es un árbol binario, con 0 a la izquierda y DESMARCAR sólo restaría el valor 0 a *pac* y *vac*.

función MOCHILAB_MARCAJE (x es solución, k, pac, vac es nat) dev
(xmejor es solución, vmejor es valor (xmejor))

{ *Pre :* $(\sum_{i=1}^{k-1} x(i)\cdot p(i) \leq \text{PMAX}) \wedge (pac = \sum_{i=1}^{k-1} x(i)\cdot p(i)) \wedge (vac = \sum_{i=1}^{k-1} x(i)\cdot v(i)) \wedge (1\leq k \leq n)$ }

```
<xmejor,vmejor>:=<sol_vacia, 0>;
x(k):= -1;                                  /* preparar recorrido de nivel k */
*[ x(k) < 1 ----> x(k):= x(k) +1 ;          /* siguiente hermano nivel k */
            pac := pac + x(k) · p(k);       /* marcar */
            vac := vac + x(k) · v(k);
            [ pac ≤ PMAX --- > /* la solución en curso es factible */
                [ k = n  --->               /* es una hoja y por tanto solución */
                    <sol,val> := < x, vac >
                [] k < n  --->              /* no es una hoja, hay que seguir  */
                    <sol,val> := MOCHILAB_MARCAJE( x, k+1, pac, vac);
                ]
```

```
                        [ vmejor ≥ val --->        seguir
                        [] vmejor < val --->       <xmejor, vmejor> := <sol, val>;
                        ]
                [] pac >PMAX ---> seguir;       /* viola las restricciones */
                ]
                                /* no hace falta desmarcar */
    ]
```

{ *Post :* xmejor es la mejor solución que se ha encontrado explorando todo el subárbol que cuelga del nodo x(k-1) estando ya establecido el camino desde la raíz a este nodo y

$vmejor = \sum_{i=1}^{n} xmejor(i) \cdot v(i)$ }

  <u>dev</u> (xmejor, vmejor)
<u>ffunción</u>

Primera llamada **<s,v> := MOCHILAB_MARCAJE (sol_vacia, 1, 0, 0).**
El coste de este algoritmo es ahora $O(2^n)$ y se ha conseguido rebajarlo gracias al marcaje.

4.4. PODA BASADA EN EL COSTE DE LA MEJOR SOLUCION EN CURSO

La *poda* es un mecanismo que permite descartar el recorrido de ciertas zonas del espacio de búsqueda. La poda básica de Vuelta Atrás la lleva a cabo la función factible y consiste en NO continuar expandiendo uno nodo que viola las restricciones. Esto evita la generación de una cierta cantidad de nodos del espacio de búsqueda. La *poda basada en el coste de la mejor solución en curso*, PBCMSC para abreviar, es otra poda adicional, distinta a la de factibilidad, que se aplica en problemas de optimización. Su objetivo es incrementar el número de nodos podados o no generados, y para que sea lo más efectiva posible es aconsejable que actue desde el comienzo y así ya se pueden notar sus efectos desde los primeros niveles del espacio de búsqueda.

Para que PBCMSC funcione hay que disponer de la mejor solución hasta el momento, la llamada *mejor solución en curso*. La idea es que el nodo en curso se poda cuando, pese a ser factible y prolongable, no es posible conseguir una solución mejor que la mejor solución en curso aunque se genere todo el árbol que cuelga de él. Entonces se dice que no merece la pena expandir el nodo en curso. La PBCMSC empezará a actuar cuando Vuelta Atrás haya conseguido encontrar la primera solución en su recorrido habitual. Es mejor que Vuelta Atrás reciba en la primera llamada una solución inicial para que la PBCMSC actue desde el comienzo. Esta solución inicial se puede calcular con una función externa a Vuelta Atrás como, por ejemplo, con un algoritmo Voraz.
Otra cuestión importante es que hay que poder calcular de forma eficiente el valor de la mejor solución que se puede conseguir expandiendo el nodo en curso. Una forma evidente de

calcularlo es aplicar Vuelta Atrás sobre el nodo en curso, pero ya sabemos que tiene un coste exponencial y, entonces, la PBCMSC no nos ahorrará nada.

Supongamos que se está resolviendo un problema de MINIMIZACION. Sea *v* el nodo en curso, *k* la profundidad de *v* en un espacio de búsqueda de altura *n*, *x* la solución en curso tal que x(k)=v, *vsol* el valor de la solución en curso (vsol=VALOR(x, 1, k)) y *vmejor* el valor de la mejor solución en curso. Sea *f* una función tal que aplicada a *v*, y=f(v), devuelve un valor *y* que es una COTA INFERIOR del valor de cualquier forma de completar la solución en curso de modo que ésta consiga ser solución al problema. Entonces *vsol+y* es una COTA INFERIOR del valor de la mejor solución que se puede conseguir expandiendo el nodo en curso *v*. Ahora ya se puede aplicar la PBCMSC sobre *v* :
- Si *vmejor<(vsol + y)*, entonces hay que podar el nodo en curso ya que expandiéndolo no se consigue una solución de valor menor que la mejor solución en curso.
- Si *vmejor≥(vsol + y)*, entonces hay que continuar expandiendo el nodo en curso ya que tal vez sea posible conseguir una solución de menor valor que el de la mejor solución en curso.

En un problema de MAXIMIZACIÓN, la función *f* ha de proporcionar una COTA SUPERIOR del valor de cualquier forma de completar la solución en curso de modo que ésta consiga ser solución al problema. Entonces el nodo en curso no se expandirá si *vmejor>(vsol + y)*.

La función *f*, denominada *función de estimación* o heurístico, y el cálculo de su valor ha de satisfacer ciertas condiciones :
1/ *f* no puede engañar : El valor devuelto por *f* ha de ser una una cota inferior de verdad. En caso contrario es posible que la PBCMSC decida podar el nodo en curso cuando, en realidad, hay un camino desde la raíz, pasando por *v* y llegando hasta algún descendiente de *v*, que es solución y con un valor menor que el de la mejor solución en curso. Si *f* engaña ya no se puede garantizar que Vuelta Atrás encuentra siempre la solución del problema, si es que la tiene.
2/ *f* no ha de ser costosa de calcular : Se ha de calcular el valor de *f* para todo nodo factible y prolongable del espacio de búsqueda. El coste de este cálculo multiplica al del coste de recorrer el espacio de búsqueda, lo que da lugar a costes elevadísimos en el caso peor. No obstante este coste es bastante improbable ya que la aplicación de la PBCMSC puede reducir considerablemente el espacio de búsqueda.

Resumiendo, dado un problema al que se le quierea aplicar PBCMSC, primero hay que buscar una función de estimación que no engañe, que cueste poco de calcular y que sea muy efectiva (que pode mucho) y, segundo, hay que calcular una solución inicial para que la poda actue desde el principio de Vuelta Atrás. Para ciertos problemas las buenas funciones de estimación que se pueden encontrar son tan costosas que resulta más barato aplicar directamente Vuelta Atrás. La solución del problema que se puede obtener aplicando un algoritmo Voraz puede ser una buena solución inicial para Vuelta Atrás con PBCMSC.

A continuación se presenta el esquema de Vuelta Atrás con PBCMSC para un problema de MINIMIZACIÓN. Como en esquemas anteriores, se ha considerarado que una solución es una secuencia de decisiones factible y no_prolongable (es un camino de la raíz a una hoja). Los parámetros de la llamada recursiva incluyen la mejor solución en curso y su valor, *xini* y *vini* respectivamente, y el valor de la solución inicial que se aporta en la primera llamada recursiva, VSOLI. Este último parámetro es innecesario pero facilita la escritura de la especificación del algoritmo.

```
función BACKMEJOR_P ( x es solución; k es nat; xini es solución; vini es valor(xini);
                      VSOLI es valor )  dev ( xmejor es solucion; vmejor es valor (xmejor) )
{ Pre :  (x [1..k-1] es factible ∧ no es solución ) ∧ 1≤ k≤altura(espacio búsqueda) ∧ xini es la
mejor solución en curso ∧ vini es el valor de xini tal que vini=MIN(valor de la mejor solución
encontrada hasta el momento,VSOLI ) }
    <xmejor,vmejor> := <xini,vini > ;           /* inicializar mejor solución en curso */
    preparar_recorrido_nivel_k ;
    *[ existan_hermanos de nivel_k   --->
          x[k] := sig_hermano_nivel_k;
          est := ESTIMAR(x,k);                   /* est =f(nodo en curso ) */
          [ factible(x,k) ∧ (COSTE(x,k) + est) < vmejor --->
                    [solucion (x,k) --->
                          solución ;  <sol,val> := < x, valor(x) >
                    []¬solucion (x,k) --->
                          <sol,val> := BACKMEJOR_P(x, k+1, xmejor, vmejor, VSOLI );
                    ]
                    [ vmejor ≤ val  ---> seguir
                    [] vmejor > val ---> <xmejor, vmejor> := <sol, val>;
                    ]
          [] (¬factible(x,k)) ∨ (COSTE(x,k) + est) ≥ vmejor ----> seguir
          ]
    ]
{ Post : Dado el prefijo x[1...k-1] se han generado todas las formas posibles de rellenar x
desde k hasta longitud(solución) y xmejor es la mejor solución que se ha encontrado en el
subárbol cuya raíz es x[k-1]. El valor de xmejor es vmejor. La generación de todas las
formas posibles de rellenar x desde k hasta longitud(solución), se ha hecho siguiendo el
orden de 'primero en profundidad' }
    dev ( xmejor, vmejor )
ffunción
```

La primera llamada será **<xm,vm> := backmejor_p (sol_vacia, 1, xini, vini, vini).**

La variable *xini* contiene una solución calculada con una función externa y *vini* contiene el valor de esa solución. Si no se puede calcular una solución inicial, entonces *xini* se inicializa a *sol_vacia* y *vini* a *0*, ya que se trata de un problema de minimización.

ESTIMAR es la función de estimación del problema y devuelve una cota inferior del mejor valor que se puede conseguir explorando el subárbol cuya raíz es el nodo *x(k).* La función COSTE(x,k) devuelve el valor de la solución en curso, el coste del camino que va desde la raíz a *x(k)*. Sobre este esquema se pueden introducir los marcajes que haga falta para mejorar la eficiencia.

A continuación se presenta el algoritmo de Vuelta Atrás con PBCMSC para resolver el problema de mochila entera con un espacio binario. En la primera llamada a MOCHILAB_P, los valores de *xini* y *vini* se pueden obtener con el algoritmo Voraz que se presentó en la sección 3.3. Basta con eliminar de esa solución el objeto fraccionado y ya se tiene una solución entera. Como función de estimación se utiliza el mismo algoritmo Voraz. Este algoritmo se aplica, tal cual, sobre los elementos que todavía no se han intentado colocar en la mochila y así se obtiene el máximo alcanzable con los objetos restantes (una cota superior). Para reducir el coste del cálculo de la función de estimación, los objetos del espacio de búsqueda se van a considerar en el mismo orden en que son considerados por el algoritmo Voraz, es decir, en orden decreciente de relación valor/peso.

función MOCHILAB_P (x es solución, k,pac,vac es nat , xini es solución, vini es valor(xini), VSOLI es valor) dev (xmejor es solución, vmejor es valor (xmejor))

{ *Pre :* $\left(\sum_{i=1}^{k-1} x(i)\cdot p(i) \le \text{PMAX}\right) \wedge \left(pac = \sum_{i=1}^{k-1} x(i)\cdot p(i)\right) \wedge \left(vac = \sum_{i=1}^{k-1} x(i)\cdot v(i)\right) \wedge (1 \le k \le n) \wedge$ *xini* es la mejor solución en curso $\wedge$ *vini* es el valor de *xini* tal que vini=MAX(valor de la mejor solución encontrada hasta el momento, VSOLI) }

```
    <xmejor,vmejor>:=<xini,vini>;
    x(k):= -1;                          /* preparar recorrido de nivel k */
    *[ x(k) < 1 ----> x(k):= x(k) +1 ;  /* siguiente hermano nivel k */
                 pac := pac + x(k) · p(k);
                 vac := vac + x(k) · v(k);
         /* se resuelve mochila fraccionada para una mochila capaz de soportar
         PMAX-pac y los objetos del k+1 al n. Recordar que los objetos se
         consideran en orden decreciente de relación valor/peso */
                 est := MOCHILA_FRACCIONADA ( x, k, pac, PMAX );
                 [ (pac ≤ PMAX) ∧ (vac+est >vmejor) --->
```

```
                    /* la solución en curso es factible y merece la pena seguir */
                        [ k = n  --->             /* es una hoja y por tanto solución */
                              <sol,val> := < x, vac >
                        [] k < n  --->            /* no es una hoja, hay que seguir  */
                              <sol,val> := MOCHILAB_P( x, k+1, pac, vac, xmejor,
                                                         vmejor, VSOLI )
                        ]
                        [ vmejor ≥ val ---> seguir
                        [] vmejor < val ---> vmejor := val; xmejor :=sol ;
                        ]
                    [] (pac >PMAX) ∨ (vac+est ≤vmejor) ---> seguir;
                    ]
                    /* no hace falta desmarcar */
    ]
```

{ *Post :* xmejor es la mejor solución en curso y vmejor = $\sum_{i=1}^{n}$ xmejor(i)·v(i) }

<u>dev</u> (xmejor, vmejor)
<u>ffunción</u>

En un caso real en que el espacio de búsqueda tenía 2^9-1 nodos, la utilización de esta PBCMSC lo redujo a 33 nodos.

5. RAMIFICACION Y PODA : Branch & Bound

5.1. CARACTERIZACION Y DEFINICIONES

El esquema de Ramificación y Poda, *Branch & Bound* en la literatura inglesa, es, básicamente, una mejora considerable del esquema de Vuelta Atrás. Se considera que es el algoritmo de búsqueda más eficiente en espacios de búsqueda que sean árboles.
Los típicos problemas que se suelen resolver con Ramificación y Poda son los de optimización con restricciones en los que la solución es expresable en forma de secuencia de decisiones, aunque cualquier problema resoluble con Vuelta Atrás también se puede resolver con Ramificación y Poda, pero a lo mejor no merece la pena.

Todos los conceptos y toda la terminología introducida en el capítulo anterior la vamos a utilizar de nuevo en este capítulo, pero va a ser necesario ampliar el repertorio para diferenciar Ramificación y Poda de Vuelta Atrás. Para ello, primero vamos a introducir unas cuantas definiciones y luego vamos a describir las diferencias entre los dos esquemas.
En Ramificación y Poda los nodos del espacio de búsqueda se pueden etiquetar de tres formas distintas. Así tenemos que uno nodo está *vivo*, *muerto* o *en expansión*. Un nodo *vivo* es un nodo factible y *prometedor* del que no se han generado todos sus hijos. Un nodo *muert*o es un nodo del que no van a generarse más hijos por alguna de las tres razones siguientes : ya se han generado todos sus hijos o no es factible o no es prometedor. Por último, en cualquier instante del algoritmo pueden existir muchos nodos vivos y muchos nodos muertos pero sólo existe un nodo *en expansión* que es aquel del que se están generando sus hijos en ese instante.
En estas definiciones ha aparecido el concepto de *nodo prometedor*. Esta idea ya se ha introducido sin ese nombre en el capítulo anterior, en el apartado 4.4. Se dice que un nodo es *prometedor* si la información que tenemos de ese nodo indica que expandiéndolo se puede conseguir una solución mejor que la mejor solución en curso. Esta información la proporcionan dos fuentes distintas : una es la función de estimación que evalua el camino que queda hasta llegar a una solución, la otra es la evaluación del camino que ya se ha recorrido y que va desde la raíz del espacio de búsqueda hasta el nodo en curso.

Tanto Vuelta Atrás como Ramificación y Poda son algoritmos de búsqueda, pero mientras Vuelta Atrás es una búsqueda ciega, Ramificación y Poda es una búsqueda *informada*. La principal diferencia entre una búsqueda o recorrido ciego, bien sea en profundidad o en anchura, y una búsqueda informada es el orden en que se recorren los nodos del espacio de búsqueda. Fijado un nodo x del espacio de búsqueda, en un recorrido ciego se sabe perfectamente cual es el siguiente nodo a visitar (el primer hijo de x sin visitar, en un recorrido en profundidad y el siguiente hermano de x, en un recorrido en anchura). Sin embargo, en un recorrido informado el siguiente nodo es el más prometedor de entre todos los nodos vivos; es el que se va a convertir en el próximo nodo en expansión.

En Vuelta Atrás tan pronto como se genera un nuevo hijo del nodo en curso, este hijo se convierte en el nuevo nodo en curso o nodo en expansión, según la terminología de Ramificación y Poda, y los únicos nodos vivos son los que se encuentran en el camino que va desde la raíz al actual nodo en expansión. En Ramificación y Poda se generan todos los hijos del nodo en expansión antes de que cualquier otro nodo vivo pase a ser el nuevo nodo en expansión por lo que es preciso conservar en algún lugar muchos más nodos vivos que pertenecen a distintos caminos. Ramificación y Poda utiliza una lista para almacenar y manipular los nodos vivos.

Dependiendo de la estrategia para decidir cual es el siguiente nodo vivo que se va a expandir de la lista, se producen distintos órdenes de recorrido del espacio de búsqueda :
1/ estrategia FIFO (la lista es una *cola*) : da lugar a un recorrido del espacio de búsqueda por niveles o en anchura.
2/ estrategia LIFO (la lista es una *pila*) : produce un recorrido en profundidad aunque en un sentido distinto al que se da en Vuelta Atrás.
3/ estrategia LEAST COST (la lista es una *cola de prioridad*) : este es el recorrido que produce Ramificación y Poda para un problema de minimización y usando función de estimación. El valor de lo que un nodo vivo promete, es decir, el coste de lo que ya se ha hecho junto con la estimación de lo que costará como mínimo alcanzar una solución, es la clave de ordenación en la cola de prioridad. Para abreviar diremos que la clave de un nodo x es *coste(x)* que es igual al *coste_real(x)+coste_estimado(x).* De este modo se expande aquél nodo que promete la solución de coste más bajo de entre todos los nodos vivos. Es de esperar que el nodo con la clave más pequeña conduzca a soluciones con un bajo coste real.

5.2. EL ESQUEMA Y SU EFICIENCIA

A continuación vamos a presentar el esquema algorítmico de Ramificación y Poda. Se trata de un esquema iterativo que itera mientras quedan nodos vivos por tratar. Este esquema garantiza que si existe solución seguro que la encuentra. Se hace un recorrido exhaustivo del espacio de búsqueda en el que sólo se descartan aquellos caminos que no son factibles o que no pueden conducir a una solución mejor que la que se tiene en estos momentos. La utilización de una función de estimación que no engañe es imprescindible para garantizar el buen funcionamiento del algoritmo.

5.2.1. EL ESQUEMA ALGORITMICO

Partimos del hecho de que se expande el nodo vivo más prometedor de la lista de nodos vivos (el de mínimo coste en un problema de minimización y el de máximo coste en uno de maximización). Por definición de nodo vivo se sabe que no se ha generado toda su descendencia y que es factible y todavía no es solución. Se generan todos sus hijos y a cada uno de ellos se les aplica el análisis que se explica a continuación. Sea *x* el hijo que se está analizando:

- si *x* no es factible, entonces *x* pasa a ser un nodo muerto.
- si *x* es factible y tiene un *coste(x)* peor que el de la mejor solución en curso, *x* pasa a ser un nodo muerto y nunca más se volverá a considerar.
- si *x* es factible y tiene un *coste(x)* mejor que el de la mejor solución en curso pero *x* no es solución, entonces *x* se inserta con clave *coste(x)* en la cola de prioridad de nodos vivos.
- si *x* es factible y tiene un *coste(x)* mejor que el de la mejor solución en curso y *x* es solución entonces pasa a ser la nueva mejor solución en curso. Además se revisan todos los nodos de la lista de nodos vivos y se eliminan de ella todos aquellos que tengan una clave peor o igual que *coste(x)*.

Una vez generados todos los hijos del nodo en expansión, éste se convierte en un nodo muerto y se vuelve a repetir el proceso de seleccionar el nodo más prometedor de entre todos los nodos vivos. El proceso acaba cuando la cola de prioridad está vacía. En ese momento la mejor solución en curso se convierte en la solución óptima del problema.

De cara a la implementación del esquema hace falta definir con toda precisión el contenido de un nodo. Intuitivamente un nodo debe almacenar toda la información que llegaba como parámetro a una llamada recursiva de Vuelta Atrás y también todo lo que era devuelto por ella. Habitualmente un *nodo* es una tupla que almacena la solución en curso, la profundidad alcanzada en el espacio de búsqueda, el valor de todos los marcajes, el coste de la solución en curso y su coste estimado. La mejor solución en curso será una variable de tipo nodo que a lo largo de las iteraciones del bucle se irá actualizando convenientemente.

```
función R&P ( D es datos_problema ) dev ( x es  nodo )
{ Pre : El espacio de búsqueda es un árbol y se trata de un problema de MINIMIZACION }
var
     x, y, mejor es nodo;
     list es lista_nodos_vivos(<nodo,valor>);    /* valor es la clave de la cola de prioridad.
                                                    Esta ordenada por orden creciente de clave */
     valmejor es valor;
fvar
     y := CREAR_RAIZ(D);
     list := INSERTAR(lista_vacia, <y, ESTIMAR(y) > )
```

```
valmejor := llamada a un algoritmo para encontrar el valor de una solución inicial. Si no
            lo hay, entonces inicializar valmejor a infinito.
/ * si valmejor tiene el valor de una solución inicial hará que la PBCMSC empiece a
actuar desde el comienzo */
*[ ¬vacia(list) --->
    y := primero(list);    list := avanzar(list);   /* y es el nodo vivo en expansión  */
    preparar_recorrido_hijos(y);
    *[ existan_hijos(y) --->
        x:= siguiente_hijo(y);
        x:= MARCAR(y, x);
        est := ESTIMAR(x);         /* est = coste_real(x)+coste_estimado(x) */
         [ factible(x) ∧ prometedor(x,est)--->
              [ solución(x) --->  <mejor, valmejor>:= <x, coste_real(x)>;
                                  list := DEPURAR(list,valmejor);
                                  /* eliminar de la lista todos los nodos vivos que
                                  prometan una solución con un coste peor que el
                                  coste de la solución que se acaba de conseguir */
              []¬solución(x) -->  list:= INSERTAR(list, <x, est >);
                                  /* x es factible y prometedor, por tanto es un
                                  nodo vivo que hay que insertar en la lista */
              ]
        []¬factible(x)∨ ¬ prometedor(x,est) ---> seguir
        ]
        /* Desmarcar si es preciso */
    ]
]
```

{ *Post :* Los nodos generados equivalen a un recorrido completo del espacio de búsqueda y en ese recorrido se ha encontrado el nodo que contiene la solución de mínimo coste real de entre todas las soluciones que hay para el problema. Ese nodo es *mejor* y el valor de la solución es *valmejor*}

<u>dev</u> (mejor, valmejor)

<u>ffunción</u>

La función de cota del bucle no es el tamaño de la lista de nodos vivos sino el número de nodos del espacio de búsqueda que aún quedan por generar. En el invariante del bucle se podría incluir que la lista de nodos vivos está ordenada por orden creciente de valor y que, de entre todos los nodos generados, la lista sólo contiene aquellos que son factibles y prometen una solución mejor que la mejor solución en curso y marcan la "frontera" en el espacio de búsqueda de lo que ya ha sido generado y merece la pena expandir.

Para resolver un problema usando este esquema se recomienda seguir los mismos pasos que en Vuelta Atrás con PBCMSC. En primer lugar identificar el espacio de búsqueda y la solución. A continuación definir las funciones de generación y de recorrido del espacio de búsqueda. Especial atención requieren las funciones ESTIMAR(x), incializar *valmejor* y CREAR_RAIZ(D). La función ESTIMAR(x) calcula la clave *coste(x)* del nodo *x* sumando *coste_real(x)* con *coste_estimado(x).* El coste estimado se calcula utilizando una función de estimación como las explicadas en Vuelta Atrás con PBCMSC. La inicialización de *valmejor* se consigue con una función externa. Es imprescindible que esta función externa genere una secuencia de decisiones que sea verdadera solución para el problema. También es conveniente que el coste de calcularla no sea excesivo. Si de partida el algoritmo dispone de este valor, la poda resultará mucho más efectiva y se reducirá el número de nodos que consiguen llegar a ser nodos vivos. Y ya por último, la función CREAR_RAIZ(D) se encarga de la generación de un nodo fantasma que permite la construcción de los nodos del primer nivel del espacio de búsqueda.

5.2.2. CONSIDERACIONES SOBRE LA EFICIENCIA

El esquema de Ramifiación y Poda es la manera más eficiente que se conoce de recorrer un espacio de búsqueda que sea un árbol. A pesar de la poda, se garantiza que encuentra una solución de coste mínimo, si ésta existe. Por tanto, equivale al recorrido completo del espacio. La efectividad de la poda depende de la calidad de la función de estimación. El coste de calcularla no ha de ser excesivo y es bueno disponer de una solución inicial para aumentar la eficiencia.

El hecho de tener que manipular una lista de nodos vivos hace que el coste aumente. Las operaciones *primero*, *avanzar*, *insertar* y *depurar* son las que trabajan sobre la lista. Para mejorar la eficiencia de las mismas se puede implementar la lista en un montículo de mínimos, un *Heap*. De este modo *primero* tiene coste constante, *avanzar* e *insertar* tienen coste *log n*, ya que hay que reconstruir el montículo, y *depurar* tiene, en el peor de los casos, coste $n \cdot logn$, siendo *n* el tamaño de la lista.

5.3. UN EJEMPLO : MOCHILA ENTERA

A estas alturas no hace falta volver a formular mochila entera. La formulación del problema, la caracterización del espacio de búsqueda en el que los objetos se consideran en orden decreciente de relación valor/peso, el algoritmo para calcular la solución inicial y la función de estimación que se dan en la sección 4.4., las vamos a reutilizar para resolver el problema pero ahora empleando Ramificación y Poda.

Tan solo hay que definir la estructura de un nodo. Vamos con ello.

```
tipo nodo es tupla
        sol es vector[1..n] de {0,1};          /* solución en curso*/
        k es nat;                              /* índice del objeto ≡ profundidad*/
```

```
        pac es nat;                        /* peso acumulado */
        vac es nat;                        /* valor  acumulado ≡ coste_real */
        est es nat;                        /* coste_estimado */
        ftupla

función MOCHILA_RP (xini es solución, vini es valor(xini)) dev
                    (xmejor es nodo, vmejor es valor (xmejor))
{ Pre : Los objetos a considerar están ordenados en orden decreciente de relación valor/peso
∧ xini es la solución inicial ∧ vini el valor de esa solución}
    x.sol:= solución_vacía;                /* creación del nodo inicial */
    x.k:= 0;    x.pac:= 0;    x.vac:= 0;
        /* coste_estimado(x) :se resuelve mochila fraccionada para una mochila capaz de
        soportar PMAX-pac y los objetos del x.k+1 al n. Recordar que los objetos se
        consideran en orden decreciente de relación valor/peso. Devolverá la solución de
        mochila fraccionada para este problema */
    x.est:= MOCHILA_FRACCIONADA( x, x.k+1, x.pac, PMAX );
        /* creación de la cola de prioridad que contendrá los nodos vivos en orden
        decreciente de clave porque el problema es de maximización. La clave es
        x.vac+x.est */
    cp:= insertar(cola_vacia, <x, x.vac+x.est>);
    <xmejor,vmejor> := <xini,vini>;        /* inic. la mejor solución en curso */
    *[¬vacia(cp) --->
        y := primero(cp);   cp := avanzar(cp);
        i := -1;                           /*preparar recorrido */
        *[ i<1 ---> i := i+1 ;             /* siguiente hermano nivel k */
                x := y;    x.k := y.k+1;   /* creación del hijo igual que el padre */
                x.sol[x.k] := i;
                x.pac := y.pac + (peso(x.k)*i);
                x.vac := y.vac + (valor(x.k) *i);
                x.est := MOCHILA_FRACCIONADA( x, x.k+1, x.pac, PMAX );
                [ (x.pac ≤ PMAX) ∧ (x.vac+x.est >vmejor) --->
                /* la solución en curso es factible y merece la pena seguir */
                    [ k = n  --->          /* es una hoja y por tanto solución */
                        <xmejor,valmejor> := < x, x.vac>;
                        cp := depurar(cp, valmejor);
                    [] k < n  --->          /* no es una hoja, hay que seguir  */
                        cp := insertar(cp,<x, x.vac+x.est>);
                    ]
                [] (x.pac >PMAX) ∨ (x.vac+x.est ≤vmejor) ---> seguir;
                ]
```

/* no hace falta desmarcar porque cada hijo se inicializa siempre con el valor del padre */

]

]

{ *Post :* xmejor es la mejor solución en curso y vmejor = $\sum_{i=1}^{n}$ x.sol(i)·valor(i) }

dev (xmejor, vmejor)

ffunción

La llamada a MOCHILA_RP va precedida por una llamada al algoritmo Voraz que calcula la solución para mochila fraccionada pero eliminando de esa solución el valor del objeto que se haya fraccionado. Basta con inicializar *vini* con el resultado del Voraz convenientemente modificado e inicializar *xini* a *nodo_vacio*.

5.4. OTRAS CUESTIONES

El algoritmo de Ramificación y Poda tiene, por decirlo de alguna manera, hermanos mayores. Existe una familia de algoritmos denominada *Best-First* que permite resolver problemas bastante parecidos a los definidos para Ramificación y Poda. En las dos secciones que vienen a continuación se va a comparar Ramificación y Poda con los algoritmos de búsqueda que ya se han visto en capítulos anteriores y se va a describir, de forma muy general, algunas de las características de los algoritmos Best-First.

5.4.1. COMPARACION CON OTRAS ESTRATEGIAS DE BÚSQUEDA

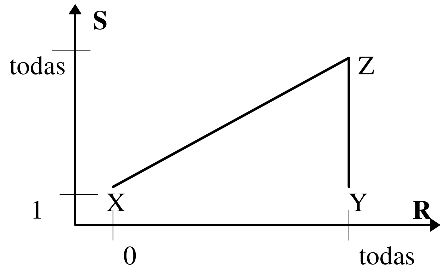
Sea *X* un algoritmo Voraz, *Y* un Vuelta Atrás y *Z* un Ramificación y Poda, todos ellos son algoritmos de búsqueda que implementan distintas estrategias. Podemos compararlos respecto de dos dimensiones :

1/ *R* ó la recuperación de alternativas suspendidas y

2/ *S* ó el número de caminos abiertos en el espacio de búsqueda.

Respecto de *R*, un algoritmo Voraz no puede recuperar ninguna de las alternativas no consideradas pero un Vuelta Atrás o un Ramifiación y Poda pueden recuperarlas todas si merecen la pena. Respecto de *S*, los algoritmos Voraces y de Vuelta Atrás mantienen un único camino abierto que va desde la raíz al nodo en curso, sin embargo, Ramificación y Poda mantiene tantos caminos abiertos como elementos hay en la lista de nodos vivos. No es posible que dos nodos vivos de la lista pertenezcan al mismo camino desde la raíz cuando el espacio de búsqueda es un árbol. Significaría que uno de ellos es antecesor del otro y si está en la lista es porque no se han generado sus hijos, por tanto, es imposible que un descendiente suyo haya sido generado ya.

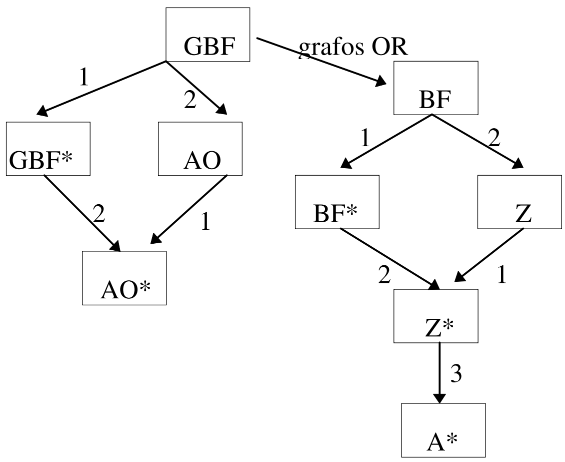
La siguiente figura ilustra gráficamente la comparación de los tres esquemas respecto de las dos dimensiones propuestas.



5.4.2. ESTRATEGIAS DE BUSQUEDA *BEST-FIRST*

Ramificación y Poda es una estrategia de búsqueda muy particular en la que el espacio de búsqueda es un árbol OR y en la que la función de estimación ha de satisfacer unos requisitos muy concretos. Este esquema pertenece a una familia de algoritmos mucho más amplia y que se utiliza básicamente en el área de Inteligencia Artificial. Son los algoritmos *primero el mejor* o *Best First*. A su vez, esta familia forma parte de una familia todavía más amplia que agrupa los algoritmos de Búsqueda basados en heurísticos. Los algoritmos Voraces también pertenecen a esta última familia.

El algoritmo GBF, *General Best First*, que funciona sobre grafos AND/OR es el algoritmo más general. La siguiente figura muestra una jerarquía de los diferentes algoritmos de búsqueda de esta familia. En ella no aparece Ramificación y Poda como tal ya que es una variante, en algunos sentidos más restrictiva y en otros más amplia, del algoritmo denominado A*.



Los números que aparecen sobre las flechas del gráfico significan lo siguiente :

1/ *terminación pospuesta*. El algoritmo acaba cuando encuentra una solución y su condición de terminación garantiza que esa solución es precisamente la óptima.

2/ *funciones de peso recursivas*. Una función de peso $W_G(n)$ es recursiva si para todo nodo *n* del grafo sucede que :

$$W_G(n)=F[E(n),W_G(n_1),W_G(n_2),...,W_G(n_b)]$$

donde $n_1,n_2,...,n_b$ son los sucesores inmediatos de *n*,

E(n) es un valor obtenido a partir de propiedades particulares del nodo *n,*
F es una función de combinación arbitraria.

La función $W_G(n)$ evalua el coste del camino desde la raíz hasta *n* y si *n* es una hoja, entonces, es el valor de la solución. Que la función de coste sea recursiva implica, según la definición anterior, que el valor se puede calcular de abajo hacia arriba, es decir, desde las hojas hacia la raíz.

3/ *evaluación aditiva*. Se refire a que la evaluación del coste de un nodo *f(x)* se hace *sumando* el valor de dos funciones: la que calcula el coste real de alcanzar *x*, *g(x)* ó *coste_real(x),* y la que estima el coste de llegar a la mejor solución alcanzable desde *x*, *h(x)* ó *coste_estimado(x).* Se tiene que :

f(x)=g(x)+h(x)

Además ∀x': x'=hijo(x) : g(x')=g(x)+c(x,x') con *c(x,x')* el coste de pasar del padre al hijo.

En las secciones que siguen vamos a ver algunas características del A* que también son aplicables a Ramificación y Poda. Sin embargo, hay diferencias que destacar como que A* funciona sobre grafos OR en los que el camino entre dos nodos no tiene porque ser único y que, gracias a las características de la función de estimación, cuando A* encuentra una solución ya se puede garantizar que es la solución óptima.

A continuación se presenta una versión de A*. La lista OPEN es la cola de prioridad de los nodos vivos y la cola de prioridad CLOSED contiene los nodos muertos. Un nodo pasa a la lista CLOSED en el momento en que se extrae de OPEN para generar sus hijos. Como un nodo es alcanzable por más de un camino, es posible que estando en CLOSED pase de nuevo a OPEN, todo depende del coste del nodo para los distintos caminos que le alcanzan. El coste de un nodo, *f(x),* es la suma del coste real del camino desde la raíz a ese nodo y del coste estimado de la mejor solución que se puede alcanzar a partir de ese nodo, exactamente igual que en Ramificación y Poda o en Vuelta Atrás con PBCMSC. Otro aspecto interesante del algoritmo A* es que acaba en el momento en que detecta que el nodo vivo extraído es solución. Se garantiza que ese nodo solución es, además, la solución óptima.

```
función A*( D es datos ) dev ( b es bool; sol es lista(nodo) )
{ Pre : El espacio de búsqueda es un grafo OR y el problema es de MINIMIZACIÓN }
    s:=CREAR_RAIZ(D);     s:= ANOTAR( s, padre=vacio);
    OPEN := insertar(lista_vacia, s);       CLOSED := lista_vacia;
    b:= FALSO;
    *[ ¬vacia(OPEN) ∧ ¬b --->
        x:=primero(OPEN);
        OPEN := avanzar(OPEN);                    /* nodo de clave mínima */
        CLOSED := insertar( CLOSED, x);           /* pasa a muerto inmediatamente*/
```

```
            [    solucion(x)  --->
                         b:= CIERTO;
                        sol:= CONCATENAR( x, padre(x));
                        /* es la primera solución encontrada y la óptima */
            []¬solucion(x) --->
                        preparar_recorrido_hijos(x);
                        *[ existan_hijos(x) --->
                                    y:=siguiente_hijo(x);
/* si fuera un problema con restricciones aquí se comprobaría si es un nodo factible */
                                    y:= ANOTAR( y, padre=x);
                                    f(y):= g(y)+h(y);              /* cálculo del coste del nodo */
                                    [ (y∉CLOSED) ∧ (y∉OPEN) --->
                                                                   /* primera vez que se llega a y */
                                          OPEN:=insertar(OPEN,y,f(y));
                                    [] (y∈CLOSED) --->             /* no es la primera vez y el nodo
                                                                   está muerto */
                                          yC:= obtener(CLOSE, y);
                                          [  (f (yC)≤f(y)) ---> seguir;
                                          [] (f (yC)>f(y)) --->
                                                      CLOSED:=eliminar(CLOSED,yC);
                                                      OPEN:=insertar(OPEN,y,f (y));
                                          ]
                                    [] (y∈OPEN) --->               /* no es la primera vez y el nodo
                                                                   está vivo */
                                          yO:= obtener(OPEN, y);
                                          [  (f (yO)≤f(y)) ---> seguir;
                                          [] (f (yO)>f(y)) --->
                                                      OPEN:=eliminar(OPEN,yO);
                                                      OPEN:=insertar(OPEN,y,f(y));
                                          ]

                                    ]
                        ]
            ]
      ]
{ Post : ( b⇒ sol=solución óptima  ) ∧ (¬b⇒ no existe solución ) }
      dev ( b,sol)
ffunción
```

La función CONCATENAR construye recursivamente la lista de nodos que forman la solución.

<u>función</u> CONCATENAR(a,b <u>es</u> nodo) <u>dev</u> (l <u>es</u> lista(nodo))
{ *Pre :* padre(a)=b}
    [b=*vacio* ---> l:= concat({a},lista_vacia);
    [] b≠*vacio* ---> l1:= **CONCATENAR**(b, padre(b));
                      l:= concat({a},l1);
    ]
{*Post : l* contiene la secuencia de nodos que se ha atravesado para llegar al nodo *a* desde el nodo inicial }
    <u>dev</u> (l)
<u>ffunción</u>

5.4.2.1. **La función de estimación en un problema de Minimización**

Sea *f(n)* el coste del nodo *n* del espacio de búsqueda. En realidad *f(n)* es una estimación del coste del camino de coste mínimo desde el nodo inicial, *s*, a un nodo objetivo (un nodo solución) pasando por el nodo *n.* El valor de *f(n)* se utiliza para ordenar la lista de nodos vivos y se tiene que : $f(n) = g(n) + h(n)$

Es necesario definir una serie de funciones para comprender el significado de cada uno de los elementos de esta expresión.

Definición : $h^*(n) = \{ \textit{MIN}\ k(n,t_i), \forall\ t_i \}$, donde, dado un nodo objetivo t_i , $k(n,t_i)$ es el coste del camino mínimo de *n* a t_i.

En un problema de optimización, queremos conocer *k(s,n)* que es el coste de un camino óptimo desde el nodo inicial *s* a un nodo arbitrario *n*.

Definición : $g^*(n) = k(s,n)$ para todo *n* accesible a partir de *s*.

Definición : Para todo *n* , $f^*(n)=g^*(n) + h^*(n)$, donde $g^*(n)$ es el coste real del camino óptimo de *s* a *n* y $h^*(n)$ es el coste de un camino óptimo de *n* a un nodo objetivo y $f^*(n)$ es el coste de un camino óptimo desde *s* a un nodo objetivo pasando por *n*.

La función $f(n)=g(n)+h(n)$ es, por tanto, una estimación de $f^*(n)$ donde $g(n)$ y $h(n)$ son a su vez una estimación de $g^*(n)$ y $h^*(n)$, respectivamente.

Para calcular el coste real del camino desde el nodo *s* al nodo *n* en el espacio de búsqueda, basta con sumar el peso de los arcos encontrados en ese camino. Así se tiene que $g(n) \geq g^*(n)$. Sin embargo para calcular el valor de *h(n),* el coste estimado desde *n* a un nodo objetivo, se ha de utilizar información heurística sobre el dominio del problema.

Las diferentes definiciones de *g(n)* y *h(n)* dan lugar a diferentes órdenes en el recorrido del espacio de búsqueda. Por ejemplo :

1/ si $g(n)=0$ y $h(x) \geq h(y)$ para todo nodo *y* que sea hijo de *x* (el coste estimado del padre es mayor o igual que el coste estimado de sus hijos), entonces se produce un recorrido en

profundidad. Sucede que una vez seleccionado *x,* los que tienen más probabilidades de ser elegidos de entre los que están en la lista de nodos vivos son sus propios hijos y no alguno de sus antecesores.
2/ si g(x)=nivel de *x* en el espacio de búsqueda y h(x)=0, entonces se produce un recorrido en *anchura*.

5.4.2.2. **Terminación y la Admisibilidad de A***

El algoritmo A* acaba cuando encuentra un nodo solución o cuando ha recorrido todo el espacio de búsqueda. No le afecta el hecho de que el espacio de búsqueda tenga ciclos ya que el número de caminos acíclicos es finito. Ramificación y Poda acaba cuando la lista de nodos vivos está vacia. Por tanto, ambos algoritmos garantizan la terminación en espacios finitos. En espacios infinitos en los que existe solución se puede garantizar la terminación si se caracteriza la función *f(n)* convenientemente. En espacios infinitos y sin nodo solución A* no acaba. En ese caso la técnica que se utiliza para provocar la finalización es buscar soluciones con un coste no superior a un cierto valor *C*.

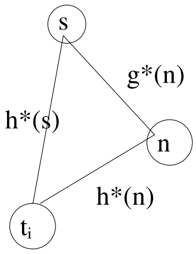
Se dice que un algoritmo de búsqueda es *admisible* si, para cualquier grafo, siempre acaba encontrando un camino óptimo del nodo inicial *s* a un nodo objetivo, si es que tal camino existe. A* es admisible si utiliza una función de estimación admisible y *f(n)* se calcula de forma aditiva.
Definición : Una función heurística *h(n)* se dice que es admisible si h(n)≤h*(n) para todo nodo *n*.
Esta definición corresponde al concepto de función que NO ENGAÑA que hemos utilizado en Vuelta Atrás y Ramificación y Poda. En [Pea 84] se pueden encontrar los teoremas y las demostraciones de la terminación y admisibilidad de A*.

5.4.2.3. **La restricción monótona**

Es evidente que A* pierde eficiencia cuando ha de buscar un nodo en OPEN y CLOSED. Y todavía es peor cuando ha de mover un nodo de CLOSED a OPEN porque eso implica que volverá a generar de nuevo los hijos de ese nodo. La restricción monótona es una propiedad que tienen algunas funciones de estimación que permiten que A* sea más eficiente.
Definición : Se dice que *h(x),* la función heurística, satisface la *restricción monótona* si el valor de la función heurística aplicado al padre es menor o igual que la suma del coste de pasar del padre al hijo más el valor de la función heurística aplicada al hijo.

$$\forall x : x=\text{hijo}(y) : \; h(x) + \text{coste}(y,x) \geq h(y)$$

Veamos de donde sale esta desigualdad y las implicaciones que tiene. En primer lugar, para todo nodo *n* del espacio de búsqueda se cumple que :



$g^*(n) + h^*(n) \geq h^*(s)$ ó, lo que es lo mismo, $k(s,n) + h^*(n) \geq h^*(s)$.

Esta es una desigualdad triangular aplicable a cualquier par de nodos, *x*,*y*, sin necesidad de que intervenga forzosamente *s*. Se puede reescribir de la siguiente forma : si *x* es cualquier descendiente de *y* entonces :

$$k(y,x) + h^*(x) \geq h^*(y)$$

y como $h(n) < h^*(n)$ entonces

$$k(y,x) + h(x) \geq h(y) \text{ para todo par } x,y$$

Definición : Se dice que *h(x)* es *consistente* si para todo par de nodos *x,y*, cumple :

$$k(y,x) + h(x) \geq h(y)$$

Definición : Se dice que *h(x)* es monótona si para todo par de nodos *x,y* tal que *x* es un hijo de *y*, cumple :

$$coste(y,x) + h(x) \geq h(y)$$

Con esta última definición hemos comenzado esta sección pero volvemos a repetirla para ver el proceso seguido en su construcción.

Los siguientes teoremas muestran las relaciones entre los conceptos definidos anteriormente e indican como afectan al comportamiento de A*

Teorema : Monotonicidad y consistencia son propiedades equivalentes

Monotonicidad parece menos restrictivo que consistencia porque sólo relaciona el heurístico de un nodo con el heurístico sobre sus descendientes directos pero se puede probar por inducción que la consistencia implica la monotonicidad.

Teorema : Toda heurística consistente es admisible.

Enlazando con un teorema anterior si la heurística es admisible entonces A* es admisible, lo que significa que si existe el óptimo, lo encuentra.

Teorema : El algoritmo A* guiado por una heurística monótona encuentra el camino óptimo a todos los nodos que expande, es decir, $g(n)=g^*(n)$ para todo nodo que haya sido generado.

La implicación inmediata de este teorema es que cuando un nodo entra en CLOSED nunca más vuelve a salir de allí.

Teorema : Los nodos que expande A* forman una sucesión no decreciente f(1º)≤f(2º)≤f(3º), etc.

Las demostraciones de estos teoremas y muchos otros pueden encontrarse en [Pea 84].

REFERENCIAS

[AHU 83] A.V.Aho, J.Hopcroft, J.D.Ullman
Data Structures and Algorithms.
Addison-Wesley, 1983

[Bal 86] J.L.Balcázar
Algoritmos de distancias mínimas en grafos.
RT86/12. Facultat d'Informàtica de Barcelona, UPC, 1986

[Bal 93] J.L.Balcázar
Apuntes sobre el cálculo de la eficiencia de los algoritmos
Report LSI 93-14-T. UPC 1993

[BB 90] G.Brassard, P.Bratley
Algorítmica, Concepción y Análisis.
Ed. Masson, 1990 (original en francés)

[BM 93] J.L.Balcázar, J. Marco
Analisi d'un algorisme de multiplicació d'enters.
Report LSI 93-11-T.UPC 1993

[CLR 92] T.Cormen, C.Leisersson, R.Rivest
Introduction to Algorithms
M.I.T. Press, 1992, 6ª ed.

[Fra 93] X.Franch
Estructura de dades. Especificació, disseny i implementació
Edicions UPC, 1993

[HS 78] E.Horowitz, S.Sahni
Fundamentals of Computer Algorithms.
Computer Science Press, 1978

[Peñ 86] R.Peña
Memoria oposición T.U.
U.P.C., Junio 1986

[Pea 84] J.Pearl
Heuristics : Intelligent Search Strategies for Computer Problem Solving
Addison-Wesley, 1984