

# **Resumen de programación 3**

## **Tema 9. Exploración de grafos.**

## Índice:

9.1. Grafos y juegos: introducción .....	3
9.2. Recorrido de árboles .....	4
9.2.1. Precondicionamiento .....	6
9.3. Recorrido en profundidad: grafos no dirigidos .....	7
9.3.1. Puntos de articulación .....	8
9.4. Recorrido en profundidad: grafos dirigidos .....	9
9.5. Recorrido en anchura .....	10
9.6. Vuelta atrás .....	21
9.6.1. El problema de la mochila (3) .....	23
9.6.2. El problema de las 8 reinas .....	25
9.6.3. El caso general .....	29
9.7. Ramificación y poda .....	29
9.7.1. El problema de la asignación .....	33
9.7.2. El problema de la mochila (4) .....	37
9.7.3. Consideraciones generales .....	38

## Bibliografía:

Se han tomado apuntes de los libros:

- *Fundamentos de algoritmia*. G. Brassard y P. Bratley
- *Estructuras de Datos y Algoritmos*. R. Hernández

En este capítulo presentamos algunas técnicas generales que se pueden utilizar cuando no se requiere ningún orden concreto en nuestro recorrido.

### 9.1. Grafos y juegos: introducción

Veremos brevemente un **juego** llamado Nim, que consiste en que dos jugadores escogen una serie de casillas hasta que gane el jugador que se quede con la última cerilla. No se permiten empates.

Ambos jugadores cumplen las mismas normas. Por tanto, para ganar la partida tendremos que imaginarnos mentalmente las jugadas tanto las nuestras como la del contrincante presentes y futuras. Para formalizar este proceso de **pensamiento anticipatorio**, representamos el juego mediante un grafo dirigido. Cada nodo del grafo corresponde a una situación del juego y cada arista corresponde a una jugada que nos lleva de una situación a otra.

Los nodos del grafo que corresponden a este juego son, por tanto, parejas de la forma  $\langle i, j \rangle$ . En general,  $\langle i, j \rangle$  con  $1 \leq j \leq i$  indica que en la mesa quedan  $i$  cerillas, y que en la jugada siguiente se puede tomar cualquier número de cerillas entre 1 y  $j$ . Las aristas que salen de esta situación, esto es, las jugadas que se pueden hacer, van a los  $j$  nodos  $\langle i - k, \min(2 * k, i - k) \rangle, 1 \leq k \leq j$ .

Para decidir cuáles son las situaciones de victoria y derrota, partimos de la situación de derrota  $\langle 0, 0 \rangle$  y retrocedemos. Este nodo no tiene sucesor y el jugador que se encuentre en esta situación perderá la partida. Podemos, por tanto, resumir las reglas vistas antes: una situación será de victoria si al menos uno de los sucesores es una situación de derrota.

Tendremos el siguiente algoritmo que determina si una situación es de victoria o de derrota:

```
funcion ganarec (i, j)
{ Devuelve verdadero si y sólo si la situación  $\langle i, j \rangle$  es de victoria,
  suponemos que  $0 \leq j \leq i$  }
para  $k \leftarrow 1$  hasta  $n$  hacer
    si no ganarec  $(i - k, \min(2 * k, i - k))$  entonces
        devolver verdadero
    devolver falso
```

Este algoritmo tiene un **inconveniente** muy grande, que es que calcula el mismo valor una y otra vez. Para solucionarlo tendremos dos enfoques:

1. Aplicando *programación dinámica*, cuyo algoritmo no veremos, por no estar en nuestro temario (sería el tema 8 que nos falta por estudiar).
2. Utilizando una *función con memoria*, donde usaremos un vector de booleanos indicando qué nodos hemos visitado durante el cálculo recursivo.

Asociamos a cada nodo una etiqueta para indicar si es victoria, derrota o tablas.

A lo largo del capítulo usaremos la palabra **grafo** de dos maneras distintas:

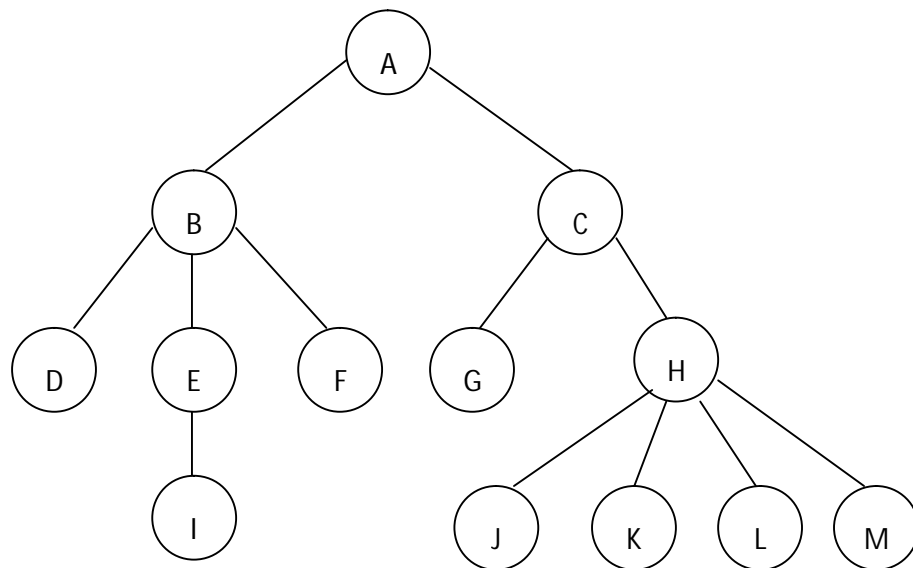
1. Un grafo puede ser una estructura de datos en la memoria de una computadora.
2. El grafo solamente existe de forma implícita. Este grafo nunca llega a existir en la memoria de la máquina. Lo veremos más adelante y será básico para la vuelta atrás.

## 9.2. Recorrido de árboles

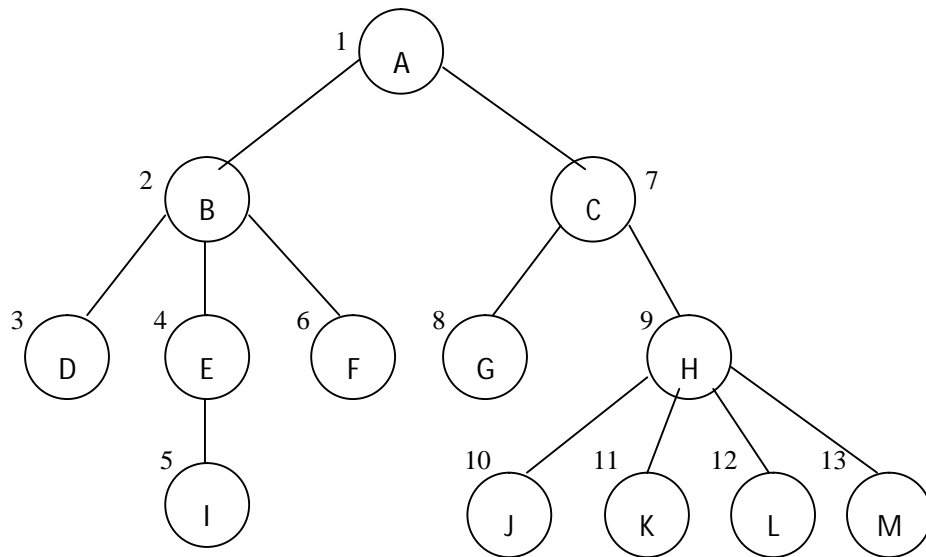
Tendremos tres técnicas para recorrer árboles, que recordemos es un grafo acíclico, conexo y no dirigido:

- **Preorden:** Visitamos primero el nodo, segundo el subárbol izquierdo y, por último, el subárbol derecho.
- **Orden infijo:** Visitamos primero el subárbol izquierdo, segundo el nodo y, por último, el subárbol derecho.
- **Postorden:** Visitamos primero el subárbol izquierdo, después el subárbol derecho y, por último, el nodo.

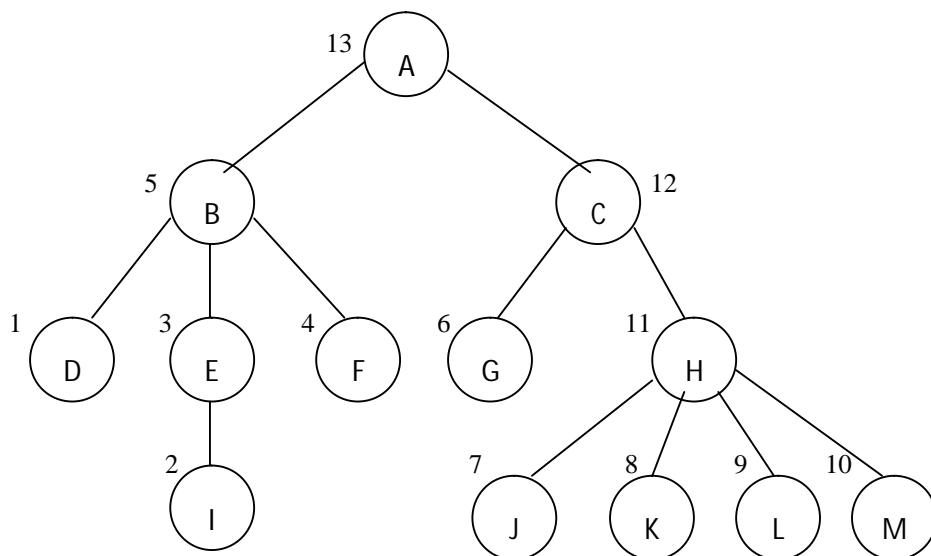
Generalmente, usaremos la primera y tercera técnica (preorden y postorden), por lo que veremos varios ejemplos de recorrido de árboles. Pondremos el árbol siguiente:



Empezamos a ver el recorrido en preorden, que será el que usemos normalmente. El orden de los nodos visitados lo pondremos en el siguiente grafo:



El recorrido en postorden será:



Exploramos el árbol de izquierda a derecha en estas técnicas aunque se puede recorrer de derecha a izquierda igualmente.

Veremos este lema correspondiente con estas técnicas, aunque no lo demostraremos formalmente:

**Lema 9.2.1** Para cada una de las seis técnicas mencionadas, el tiempo total  $T(n)$  que se necesita para explorar el árbol binario que contiene  $n$  nodos se encuentra en  $\theta(n)$ .

### 9.2.1. Precondicionamiento

Emplearemos el precondicionamiento en dos **casos** distintos (aunque no están distinguidos en el libro directamente por lo que lo haremos en el resumen):

**1<sup>er</sup> caso:** Realización de cálculos anticipados de información auxiliar. Puede merecer la pena invertir una cierta cantidad de tiempo en calcular resultados auxiliares que pueden ser utilizados en el futuro para acelerar la resolución de cada paso. Esto es el precondicionamiento.

Sea  $a$  el tiempo que necesita para resolver un caso típico cuando **no** se dispone de información auxiliar, sea  $b$  el tiempo **si** se dispone de información auxiliar y  $p$  el tiempo para calcular esta información auxiliar.

Sin precondicionamiento, para resolver  $n$  casos típicos necesitaremos un tiempo  $n * a$ . Con precondicionamiento, empleamos un tiempo  $p + n * b$ . Siempre que  $b < a$ , resulta ventajoso el precondicionamiento cuando  $n > \frac{p}{(a-b)}$ .

El empleo de este caso es para *aplicaciones en tiempo real*, donde se necesita asegurar una respuesta suficientemente rápida.

**2º caso:** Usaremos el problema de determinar los antecesores dentro de un árbol con raíz. Sea  $T$  un árbol con raíz, no necesariamente binario, decimos que un nodo  $v$  de  $T$  es un antecesor del nodo  $w$  si  $v$  está en el camino que va desde  $w$  hasta la raíz de  $T$ . El **problema** es dado un par de nodos  $(v, w)$  de  $T$  determinar si  $v$  es o no antecesor de  $w$ .

Para precondicionar el árbol, recorreremos en preorden y luego en postorden, numerando secuencialmente los nodos a medida que los visitamos. Tendremos, por tanto, que:

$prenum[v]$  es el número asignado a  $v$  cuando se recorre el árbol en preorden.

$postnum[v]$  es el número asignado a  $v$  cuando se recorre el árbol en postorden.

En preorden, recordemos que primero numeramos el nodo y después sus subárboles de izquierda a derecha tendremos:

$$prenum[v] \leq prenum[w] \Leftrightarrow \begin{cases} v \text{ es un antecesor de } w, \text{ o bien} \\ w \text{ está a la izq. de } w \text{ en el árbol} \end{cases}$$

En postorden, recordemos que primero numeramos sus subárboles de izquierda a derecha y después numeramos el nodo tendremos:

$$postnum[v] \geq postnum[w] \Leftrightarrow \begin{cases} v \text{ es un antecesor de } w, \text{ o bien} \\ w \text{ está a la der. de } w \text{ en el árbol} \end{cases}$$

Se sigue que:

$$prenum[v] \leq prenum[v] \text{ y } postnum[v] \geq postnum[w] \Leftrightarrow \\ v \text{ es antecesor de } w$$

Los valores de *premun* y *postnum* se han calculado en un tiempo que está en  $\theta(n)$ , mientras que la condición requerida se puede comprobar en un tiempo que está en  $\theta(1)$ . Esta es la importancia realmente de este caso, que luego tarda poco en comprobar y ahorra mucho tiempo.

### 9.3. Recorrido en profundidad: grafos no dirigidos

Para el **recorrido en profundidad** se siguen estos pasos:

- Se selecciona cualquier nodo  $v \in N$  como punto de partida.
- Se marca este nodo para mostrar que ya ha sido visitado.
- Si hay un nodo adyacente a  $v$  que no haya sido visitado todavía, se toma este nodo como punto de partida y se invoca recursivamente al procedimiento en profundidad. Al volver de la llamada recursiva, si hay otro nodo adyacente a  $v$  que no haya sido visitado se toma este nodo como punto de partida siguiente, se llama recursivamente al procedimiento y, así sucesivamente.
- Cuando están marcados todos los nodos adyacentes a  $v$  el recorrido que comenzó en  $v$  ha finalizado. Si queda algún nodo de  $G$  que no haya sido visitado tomamos cualquiera de ellos como nuevo punto de partida y (como en los grafos no conexos), volvemos a invocar al procedimiento. Se sigue así hasta que estén marcados todos los nodos de  $G$ .

El procedimiento **de inicialización y arranque** será:

```

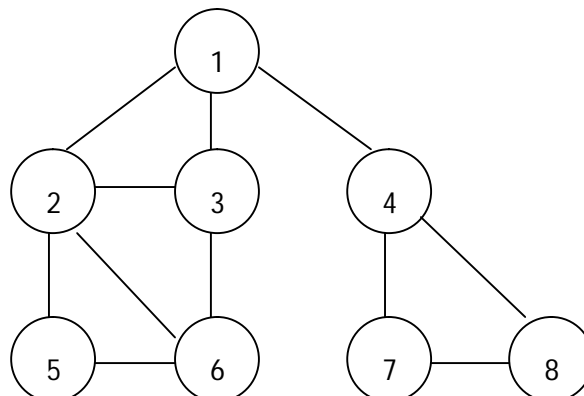
procedimiento recorridop (G)
  para cada  $v \in N$  hacer  $\text{marca}[v] \leftarrow$  no visitado
  para cada  $v \in N$  hacer
    si  $\text{marca}[v] \neq$  visitado entonces rp ( $v$ )
  
```

El algoritmo de recorrido en profundidad siguiendo los pasos anteriores es:

```

procedimiento rp ( $v$ )
  { El nodo  $v$  no ha sido visitado anteriormente }
   $\text{marca}[v] \leftarrow$  visitado
  para cada nodo  $w$  adyacente a  $v$  hacer
    si  $\text{marca}[w] \neq$  visitado entonces rp ( $w$ )
  
```

**Ejemplo:** Se nos da este grafo no dirigido:



Suponemos que el nodo de partida es el 1. La exploración del grafo en profundidad progresa en la forma siguiente:

- |    |                             |  |
|----|-----------------------------|--|
| 1. | rp(1)                       | Llamada inicial                          |
| 2. | rp(2)                       | Llamada recursiva                        |
| 3. | rp(3)                       | Llamada recursiva                        |
| 4. | rp(6)                       | Llamada recursiva                        |
| 5. | rp(5)                       | Llamada recursiva; no se puede continuar |
| 6. | rp(4)                       | No se ha visitado un vecino del nodo 1   |
| 7. | rp(7)                       | Llamada recursiva                        |
| 8. | rp(8)                       | Llamada recursiva; no se puede continuar |
| 9. | No quedan nodos por visitar |  |

**Análisis del coste:** Si se representa el grafo de tal manera que la lista de nodos adyacentes tenga un acceso directo, empleando para ello lista de adyacencias (recordemos grafolista del tema 5), entonces este trabajo es proporcional a  $a$  en total. El algoritmo requiere un tiempo que está en  $\theta(n)$  para las llamadas al procedimiento y un tiempo en  $\theta(a)$  para inspeccionar las marcas. Por tanto, el tiempo de ejecución está en  $\theta(\max(a, n))$ .

El recorrido en profundidad de un **grafo conexo** asocia al grafo un árbol de recubrimiento. Sea  $T$  este árbol. Las aristas de  $T$  corresponden a las aristas utilizadas para recorrer el grafo; están dirigidas del primer nodo visitado al segundo. Las aristas que no se utilizan en el recorrido del grafo no tienen una arista correspondiente en  $T$ . El punto inicial de partida de la exploración pasa a ser la raíz del árbol.

Resulta fácil mostrar que una arista de  $G$  que no tenga una arista correspondiente en  $T$  une necesariamente un nodo  $v$  con alguno de sus antecesores en  $T$ .

Si el grafo que se está explorando **no es conexo**, entonces un recorrido en profundidad le asocia no sólo a un único árbol, sino a un todo un bosque de árboles, uno por cada componente conexa del árbol. Un recorrido en profundidad también ofrece una manera de numerar los nodos del grafo que se está visitando. Los nodos del árbol asociado se numeran en preorden.

### 9.3.1. Puntos de articulación

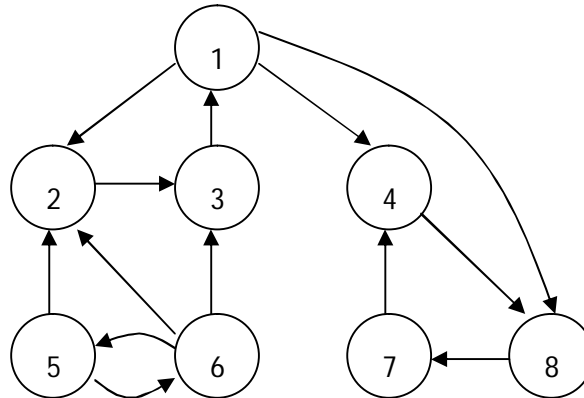
Un nodo  $v$  de un grafo conexo es un punto de articulación si el subgrafo que se obtiene borrando  $v$  y todas las aristas en  $v$  ya no es conexo. Nos evitaremos más detalles, puesto que no nos interesará saber más de este apartado.



#### 9.4. Recorrido en profundidad: grafos dirigidos

El algoritmo es esencialmente el mismo que para los grafos no dirigidos; la diferencia reside en la interpretación de la palabra “adyacente”. En un grafo dirigido, el nodo  $w$  es adyacente al  $v$  si existe la arista dirigida  $(v, w)$ .

**Ejemplo:** En el siguiente grafo dirigido:



Suponemos que el nodo de partida es el 1. La exploración del grafo en profundidad progresa en la forma siguiente:

- |                                |  |
|--------------------------------|--|
| 1. rp(1)                       | Llamada inicial                          |
| 2.     rp(2)                   | Llamada recursiva                        |
| 3.         rp(3)               | Llamada recursiva; no se puede continuar |
| 4.     rp(4)                   | No se ha visitado un vecino del nodo 1   |
| 5.         rp(8)               | Llamada recursiva                        |
| 6.             rp(7)           | Llamada recursiva, no se puede continuar |
| 7. rp(5)                       | Nuevo punto de comienzo                  |
| 8.     rp(6)                   | Llamada recursiva, no se puede continuar |
| 9. No quedan nodos por visitar |  |

**Análisis del coste:** El tiempo que requiere este algoritmo también está en  $\theta(\max(a, n))$ . En este caso, las aristas utilizadas para visitar todos los nodos de un grafo dirigido  $G = \langle N, A \rangle$  pueden formar un bosque de varios árboles aunque  $G$  sea conexo.

### 9.5. Recorrido en anchura

Cuando un recorrido en profundidad llega a un nodo  $v$ , intenta a continuación, visitar algún vecino de  $v$ , después algún vecino del vecino y, así sucesivamente. Daremos una formulación no recursiva del algoritmo de recorrido en profundidad:

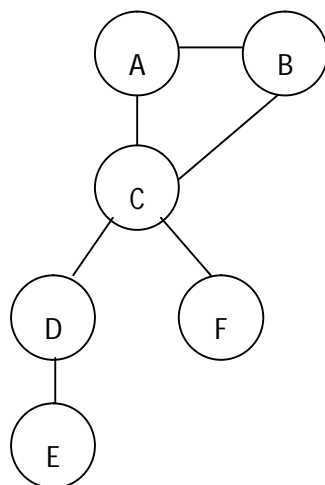
Sea **pila** un tipo de datos que admite dos valores *apilar* y *desapilar*. Se pretende que este tipo represente una lista de elementos que hay que manejar por el orden “primero en llegar, primero en salir”. La función *cima* denota el elemento que se encuentra en la parte superior de la pila.

El algoritmo de recorrido en profundidad ya modificado es:

```
procedimiento rp2 (v)
  P ← pila-vacía
  marca[w] ← visitado
  apilar w en P
  mientras P no esté vacía hacer
    mientras exista un nodo w adyacente a cima (P)
      tal que marca[w] ≠ visitado hacer
        marca[w] ← visitado
        apilar w en P      { w es la nueva cima (P) }
    desapilar P
```

**NOTA DEL AUTOR:** Tras ver el código de nuevo se ha encontrado lo que a mi parecer es una errata, y es que nunca entraría en el bucle “mientras” a no ser que apiles algún nodo en P, por ello se añade la línea ‘apilar w en P’. Está en la fe de erratas del libro de problemas, por lo que se escribe correctamente (el código está en la página 338).

**Ejemplo:** Tendremos el siguiente ejemplo de profundidad con pila, donde señalaremos con flechas el sentido de apilar y desapilar, para verlo de modo más grafico:

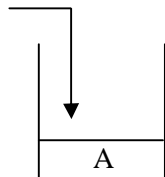


Lo veremos paso a paso empezando a visitar el nodo A:

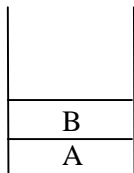
**Inicialmente:** La pila está vacía:



**1<sup>er</sup> paso:** Apilamos el nodo A y lo marcamos como visitado. En la pila: A

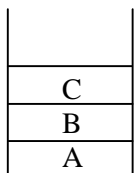


**2<sup>o</sup> paso:** Apilamos el nodo B y lo marcamos como visitado. En la pila: A, B

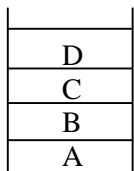


NOTA: Escogemos ese nodo sin ningún criterio en especial, ya que estimo que sería más 'lógico' escoger el nodo C, pero ahí queda. Más abajo se aclarará como se hace la búsqueda en profundidad.

**3<sup>er</sup> paso:** Apilamos el nodo C y lo marcamos como visitado. En la pila: A, B, C



**4<sup>o</sup> paso:** Apilamos el nodo D y lo marcamos como visitado. En la pila: A, B, C, D

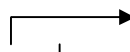


**5º paso:** Apilamos el nodo E y lo marcamos como visitado. En la pila: A, B, C, D, E

E
D
C
B
A

**6º paso:** Desapilamos el nodo E, porque no tiene más nodos adyacentes. En la pila: A, B, C, D

D
C
B
A



**7º paso:** Desapilamos el nodo D, porque no tiene más nodos adyacentes. En la pila: A, B, C

C
B
A

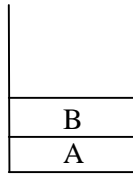
**8º paso:** Apilamos el nodo F, por ser un hijo del nodo C y lo marcamos como visitado. En la pila: A, B, C, F

F
C
B
A

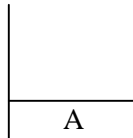
**9º paso:** Desapilamos el nodo F, porque no tiene más nodos adyacentes. En la pila: A, B, C

C
B
A

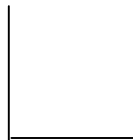
**10º paso:** Desapilamos el nodo C, porque no tiene más nodos adyacentes. En la pila: A, B



**11º paso:** Desapilamos el nodo B, porque no tiene más nodos adyacentes. En la pila: A



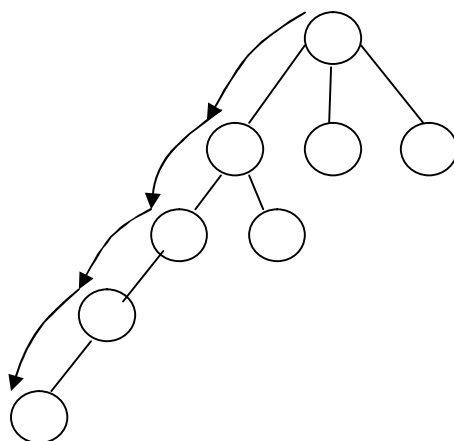
**12º paso y último:** Desapilamos el nodo A, porque no hay más nodos que explorar. La pila está vacía, llegamos al final.



El orden de exploración será: A, B, C, D, E, F

Es una coincidencia el recorrerlos por orden, pero no tiene porque ser así.

Para verlo más gráficamente, tendremos este árbol que iremos marcando con flechas por donde iríamos recorriendo:



Como se observa el recorrido iría del nodo de arriba hasta el de abajo para luego cuando no se pueda continuar seguir con los demás nodos, tal y como hemos visto en el ejemplo anterior.

En cuanto al **recorrido en anchura** seguiremos estos pasos:

- Se toma cualquier nodo  $v \in N$  como punto de partida.
- Se marca este nodo como visitado.
- Después se visita a todos los adyacentes antes de seguir con nodos más profundos.

El procedimiento de **inicialización y arranque** será:

```
procedimiento recorrido (G)
  para cada  $v \in N$  hacer  $\text{marca}[v] \leftarrow$  no visitado
  para cada  $v \in N$  hacer
    si  $\text{marca}[v] \neq$  visitado entonces  $\{rp2 \text{ o } ra\}(v)$ 
```

Para el algoritmo de recorrido en anchura necesitamos un tipo **cola** que admite las dos operaciones *poner* o *quitar*. Este tipo representa una lista de elementos que hay que manejar por el orden “primero en llegar, primero en salir”. La función *primero* denota el elemento que ocupa la primera posición en la cola.

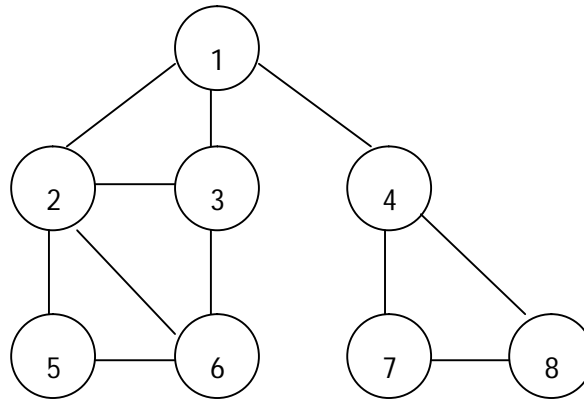
El **recorrido en anchura** no es naturalmente recursivo, por lo que el algoritmo será:

```
procedimiento ra (v)
   $Q \leftarrow$  cola-vacía
  poner v en Q
  mientras Q no esté vacía hacer
     $v \leftarrow$  primero (Q)
    quitar u de Q
    para cada nodo w adyacente a u hacer
      si  $\text{marca}[w] \neq$  visitado entonces
         $\text{marca}[w] \leftarrow$  visitado
        poner w en Q
```

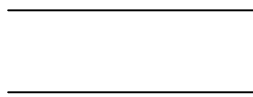
NOTA DEL AUTOR: Al igual que pasaba con el algoritmo rp2, añadiremos una nueva línea ‘poner v en Q’, de tal manera que al encolar el primer nodo ya la cola Q no está vacía y entraría en el bucle “mientras”, exactamente como pasaba antes.

**Ejemplos:** Veremos un par de ejemplos que hace este tipo de búsqueda en anchura, aunque con uno de los dos sería suficiente para saber hacerlo. De nuevo pondremos flechas que indican el sentido de encolar y desencolar.

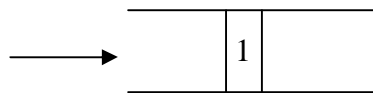
El primero de ellos es el siguiente, sacado del libro completamente y respetando el orden de visitas, en el que empezaremos a visitarlo por el nodo 1:



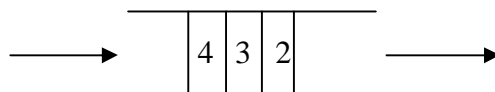
**Inicialmente:** La cola está vacía (se añade este paso a la teoría del libro):



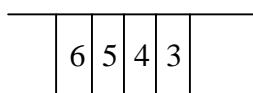
**1<sup>er</sup> paso:** Encolamos el nodo de partida, 1 (se añade este paso a la teoría del libro). En la cola: 1



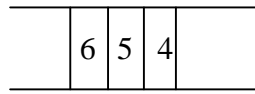
**2<sup>o</sup> paso:** Visitamos (desencolamos) el nodo 1 y encolamos los hijos de él, que son el 2, 3 y 4. En la cola: 2, 3, 4



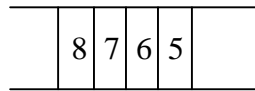
**3<sup>er</sup> paso:** Visitamos (desencolamos) el nodo 2 y encolamos los hijos, añadiéndolos a la cola. En la cola: 3, 4, 5, 6



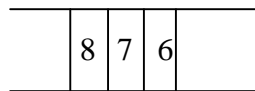
**4º paso:** Visitamos (desencolamos) el nodo 3 y al no tener hijos no encolamos ningún nodo más. En la cola: 4, 5, 6



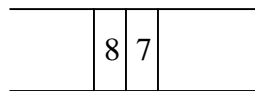
**5º paso:** Visitamos (desencolamos) el nodo 4 y encolamos los hijos del mismo, de nuevo. En la cola: 5, 6, 7, 8



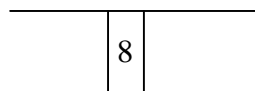
**6º paso:** Visitamos (desencolamos) el nodo 5 y al tener el resto ya recorridos no apilamos ninguno más. En la cola: 6, 7, 8



**7º paso:** Visitamos (desencolamos) el nodo 6, que por los mismos motivos de antes no apilamos ningún nodo más. En la cola: 7, 8



**8º paso:** Visitamos (desencolamos) el nodo 7 y no añadimos ningún nodo más. En la cola: 8



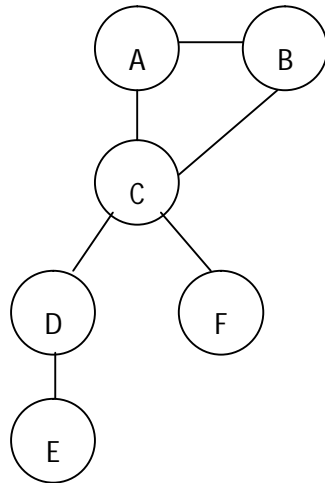
**9º paso y último:** Visitamos el último nodo y ya queda la cola vacía.



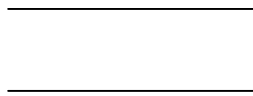
El orden de exploración será: 1, 2, 3, 4, 5, 6, 7, 8



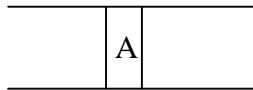
El **segundo ejemplo** es idéntico al que vimos anteriormente en el recorrido en profundidad, en el que veremos todavía más desgranados los pasos, por lo que puede quedar algo distinto al anterior, pero usando la misma técnica. Es un ejemplo complementario para comprender más este tipo de estructuras de datos. Empezamos por el nodo A:



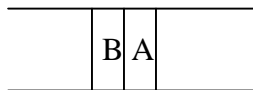
**Inicialmente:** La cola está vacía:



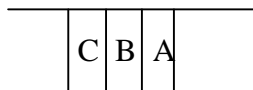
**1<sup>er</sup> paso:** Encolamos el nodo de partida, A. En la cola: A



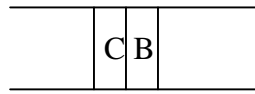
**2<sup>o</sup> paso:** Encolamos uno de los hijos de A, que es el B, marcándolo como recorrido. En la cola: A, B



**3<sup>er</sup> paso:** Encolamos el otro hijo de A, que es el C. En la cola: A, B, C

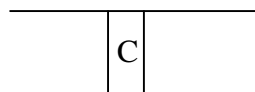


**4º paso:** Desencolamos el nodo A, ya que no hay hijos sin visitar que explorar. En la cola: B, C

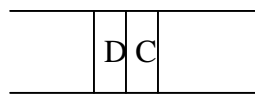


NOTA: Vemos que en el ejemplo anterior estos tres primeros pasos lo hemos hecho en sólo uno, en los que primero encolamos los hijos, hasta recorrerlos todos y luego desencolamos el padre.

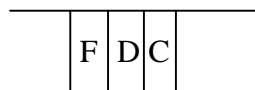
**5º paso:** Desencolamos el nodo B, por no tener ningún descendiente no explorado previamente. En la cola: C



**6º paso:** Encolamos uno de los hijos de C, que es el D, marcándolo como recorrido. En la cola: C, D



**7º paso:** Encolamos el otro hijo de C, que es el F. En la cola: C, D, F



**8º paso:** Desencolamos el nodo C, ya que no hay hijos sin visitar que explorar. Nos fijamos que de nuevo, pasa algo similar al paso 2. En la cola: D, F



**9º paso:** Encolamos el otro hijo de D, que es el E. En la cola: D, F, E



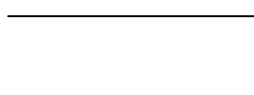
**10º paso:** Desencolamos el nodo D, ya que no hay hijos sin visitar que explorar. En la cola: F, E



**11º paso:** Desencolamos el nodo F, por no tener ningún descendiente no explorado previamente. En la cola: E

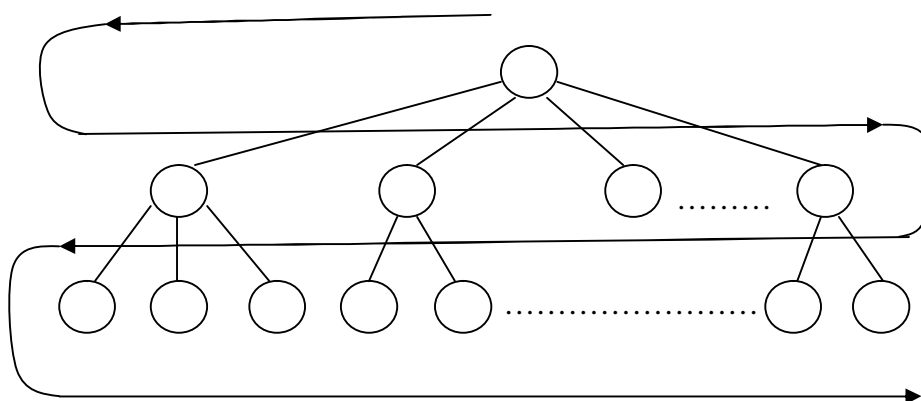


**12º paso:** Desencolamos el nodo E, por no tener ningún descendiente no explorado previamente, por lo que la cola ya está vacía.



El orden de exploración será: A, B, C, D, F, E

Como hemos hecho en la exploración en profundidad veremos cómo lo haremos en anchura de modo más grafico:

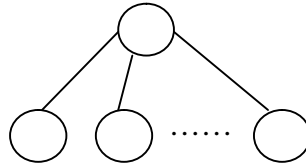


Al igual que el recorrido en profundidad, podemos asociar un árbol al recorrido en anchura. Si el grafo  $G$  que se está recorriendo es no conexo, el recorrido en anchura genera un bosque de árboles, uno por cada componente de  $G$ .

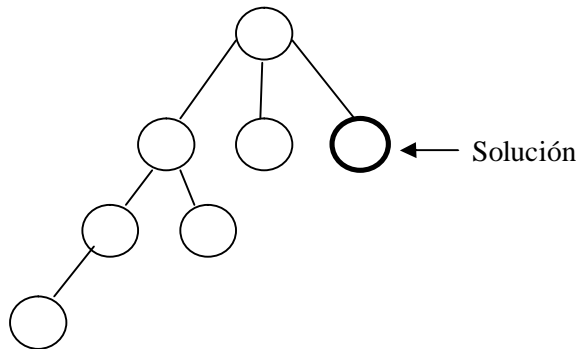
**Análisis del coste:** Igual que el recorrido en profundidad tendremos que el coste es  $\theta(\max(a, n))$ . Se puede aplicar el recorrido en anchura tanto en grafos dirigidos como en no dirigidos.

La comparación entre ambos recorridos será la siguiente:

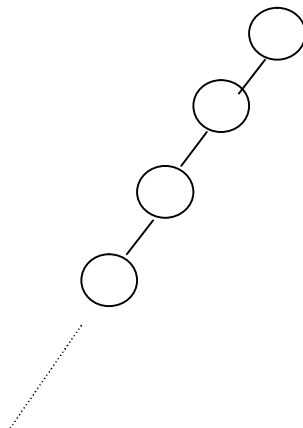
- El recorrido en anchura se realizará en una de estas situaciones:
  1. Cuando haya que efectuar una exploración parcial de un **grafo infinito**. En ocasiones puede no terminar si hay niveles con un número infinito de vecinos (no se suele dar en la práctica), como pudiera ser el siguiente grafo:



2. Para hallar el **camino más corto** desde un punto de un grafo a otro, es decir, la solución será el nodo más cercano a la raíz y tiene la certeza de hallar una solución si existe. Un ejemplo pudiera ser este grafo:



- El recorrido en profundidad puede **no** terminar si las *ramas son infinitas*. Una rama infinita puede ser la siguiente:



## 9.6. Vuelta atrás

Hay problemas que son inabordables mediante grafos abstractos (almacenados en memoria). Si el grafo contiene un número elevado de nodos y es infinito, puede resultar inútil construirlo implícitamente en memoria. En tales situaciones emplearemos un **grafo implícito**, que será aquél para el cual se dispone de una descripción de sus nodos y aristas, de tal manera que se pueden construir partes relevantes del grafo a medida que progresa el recorrido.

En su forma básica, la vuelta atrás se asemeja a un recorrido en profundidad dentro de un grafo dirigido. Esto se consigue construyendo soluciones parciales a medida que progresa el recorrido; estas soluciones parciales limitan las regiones en las que se puede encontrar una solución completa. Se nos darán estos casos:

- El recorrido **tendrá éxito** si se puede definir por completo una solución. En este caso, el algoritmo puede o detenerse (si sólo necesita una solución al problema) o bien seguir buscando soluciones alternativas (si deseamos examinarlas todas).
- Por otra parte, el recorrido **no tiene éxito** si en alguna etapa de la solución parcial construida hasta el momento no se puede completar, lo cual denominaremos condición de poda (no se construye esa parte del árbol). Cuando vuelve a un nodo que tiene uno o más vecinos sin explorar, prosigue el recorrido de una solución.

Hemos visto antes que podemos explorar buscando soluciones alternativas, en ese caso, la vuelta atrás se puede usar para *problemas de optimización*, lo que implica que:

- Exige encontrar todas las soluciones y quedarnos con la óptima.
- No es el más adecuado, ya que lo veríamos más adelante en este tema (usaríamos para ello ramificación y poda).

Este **primer esquema** de vuelta atrás es el general, en nuestro caso, será el básico que tengamos que saber para la asignatura:

```
fun vuelta-atrás (ensayo)
  si valido (ensayo) entonces
    devolver ensayo
  si no
    para cada hijo en compleciones (ensayo) hacer
      si condiciones-de-poda (hijo) entonces
        vuelta-atrás (hijo)
      fsi
    fpara
  fsi
ffun
```

Necesitaremos especificar lo siguiente:

1. **Ensayo:** Es el nodo del árbol.
2. **Función válido:** Determina si un nodo es solución al problema o no.
3. **Función compleciones:** Genera los hijos de un nodo dado.
4. **Función condiciones-de-poda:** Verifica si se puede descartar de antemano una rama del árbol, aplicando los criterios de poda sobre el nodo origen de esa rama. Adoptaremos el convenio de que la función condiciones-de-poda devuelve *cierto* si ha de explorarse el nodo, y *falso* si puede abandonarse.

El **segundo esquema** que veremos y será una variación del visto anteriormente será aquél en el que finaliza al encontrar la primera solución.

```
fun vuelta-atrás (ensayo) dev (solución)
  si valido (ensayo) entonces
    devolver ensayo
  si no
    lista ← compleciones (ensayo)
    solución ← solución_vacia
    mientras no vacía (lista) y solución = solución_vacia hacer
      hijo ← primero (lista)
      lista ← resto (lista)
      si condiciones-de-poda (hijo) entonces
        solución ← vuelta-atrás (hijo)
      fsi
    fmientras
    devolver solución
  fsi
ffun
```

Nos fijamos que almacena en una estructura de datos *lista* los nodos que se pueden seguir explorando, es decir, que no cumple las condiciones de poda, como hemos explicado previamente.

El **tercer esquema** y último en el que igualmente finaliza al encontrar la primera solución será el siguiente:

```
fun vuelta-atrás (ensayo) dev (es_solucion, solución)
  si valido (ensayo) entonces
    devolver (verdadero, ensayo)
  si no
    es_solucion ← false
    lista ← compleciones (ensayo)
    solución ← solución_vacia
    mientras no vacía (lista) y no es_solución hacer
      hijo ← primero (lista)
      lista ← resto (lista)
      si condiciones-de-poda (hijo) entonces
        (es_solucion, solución) ← vuelta-atrás (hijo)
      fsi
    fmientras
    devolver (es_solucion, solución)
  fsi
ffun
```

Este último esquema tendrá una leve modificación con respecto al anterior, por lo que añadiremos una variable *booleana*, que nos indicará si hay solución o no. Esta variable nos servirá para, por ejemplo, verificar mediante una llamada externa a la función si ha encontrado una solución (cuando la encuentre se para el algoritmo).

NOTA DEL AUTOR: El primer esquema se ha sacado directamente del libro de problemas, aunque adaptado para cerrar los bucles (como puede ser “si”, “para”,...) y así quede más didáctico. Los dos siguientes son adaptaciones de otros de problemas. Se ponen por si entrara en examen.

### 9.6.1. El problema de la mochila (3)

Se nos da un cierto número de objetos y una mochila. En esta ocasión y a diferencia de los algoritmos voraces, en lugar de suponer que estén disponibles  $n$  objetos, supondremos que los que tenemos son  $n$  **tipos de objetos** y que está disponible un número adecuado de objetos de cada tipo.

Cada objeto de un tipo  $i$   $i = 1, 2, \dots, n$  tiene un peso positivo  $w_i$  ( $w_i > 0$ ) y un valor positivo  $v_i$  ( $v_i > 0$ ). La mochila puede llevar un peso que no exceda de  $W$  (al igual que en los voraces, insistimos de nuevo).

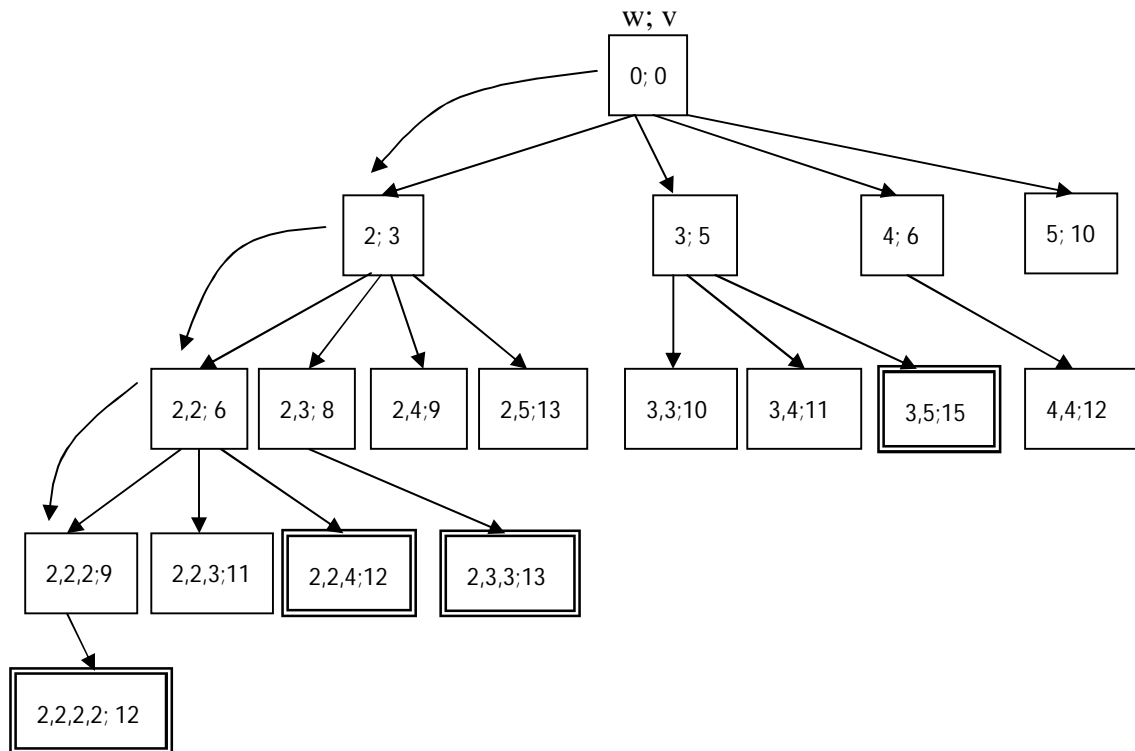
Nuestro **objetivo** es llenar la mochila de tal manera que se **maximice** el valor de los objetos incluidos, respetando la restricción de capacidad. No encuentra función de selección que garantice que es solución óptima, por eso se descarta el esquema voraz (aunque el esquema de vuelta atrás no es el más adecuado por ser un problema de optimización). No podemos aplicar tampoco búsqueda en profundidad o en anchura porque el árbol puede ser infinito.

**Ejemplo:** Se nos dan cuatro tipos de objetos distintos con peso máximo  $W = 8$

$$n = 4, W = 8$$

w (pesos)	1	2	3	4
v (valores)	3	5	6	10

Desarrollaremos el árbol implícito del problema de la mochila, que será el siguiente:



Hemos marcado las **posibles soluciones** con un doble cuadrado. Podemos acordar que cargaremos los objetos en la mochila por orden creciente de peso, como hemos marcado en nuestro dibujo. Reducimos el tamaño del árbol a explorar aunque podemos usar otro orden, que no sea el que hemos hecho antes.

El **procedimiento** es el siguiente para el ejemplo anterior:

- Inicialmente, la solución parcial está vacía.
- El algoritmo de vuelta atrás explora el árbol como un recorrido en profundidad, construyendo nodos y soluciones parciales a medida que avanza. En el ejemplo, el primer nodo que se visita es el (2; 3), el siguiente el (2,2; 6), el tercero el (2,2,2; 9) y el cuarto el (2,2,2,2; 12).
- A medida que se visita cada nodo, se extiende la solución parcial. Después visitar estos cuatro nodos, se bloquea el recorrido en profundidad: el nodo (2,2,2,2; 12) no posee sucesores por no cumplir las restricciones del problema. Dado que esta solución parcial, puede resultar ser la solución óptima, de nuestro caso, la memorizamos.



- El algoritmo de recorrido en profundidad, vuelve atrás ahora en busca de otras soluciones. En el ejemplo, el recorrido vuelve primero a (2,2,2;9), que carece de sucesores no visitados, sin embargo, al retroceder un paso más por el árbol hasta el nodo (2,2;6) hay 2 sucesores que quedan por visitar.
- Explorando de esta manera encontramos que (2,3,3;13) es una solución mejor que la que ya tenemos y que (3,5;15) es aun mejor, siendo ésta la *solución óptima*, por ser la que tiene mayor valor de todas.

Programar el algoritmo es sencillo e ilustra la íntima relación existente entre recursión y recorrido en profundidad. Supongamos que los valores de  $n$  y de  $W$  y los valores de las matrices  $w[1..n]$  y de  $v[1..n]$  correspondientes al caso que hay que resolver están disponibles como **variables globales**. La ordenación de los tipos de elementos es irrelevante. El algoritmo será:

```
funcion mochilava (i, r)
    { Calcula el valor de la mejor carga que se puede construir empleando
      elementos de los tipos 1 a n cuyo peso total no sobrepase r }
     $b \leftarrow 0$ ;
    { Se prueban por turno las clases de objetos admisibles }
    para  $k \leftarrow 1$  hasta n hacer
        si  $w[k] \leq r$  entonces
             $b \leftarrow \max(b, v[k] + mochilava(k, r - w[k]))$ 
    devolver b
```

La llamada inicial es mochilava (1, W).

Cada llamada recursiva a mochilava se corresponde con extender el recorrido en profundidad hasta el nivel inmediatamente inferior del árbol, mientras que el bucle “para” se encarga de examinar todas las posibilidades en un nivel dado.

### 9.6.2. El problema de las 8 reinas

Este es el segundo problema que veremos de vuelta atrás. Consiste en situar ocho reinas en un tablero de ajedrez de tal manera que ninguna de ellas amenace a ninguna de las demás. Recordemos que una reina amenaza a los cuadrados de la misma fila, columna o diagonal.

Veremos las distintas maneras de resolver el problema:

La forma más **evidente** consiste en generar todas las posibilidades de colocar 8 reinas en un tablero. Esto será un enfoque exhaustivo, es decir, recorriendo todas las posibilidades (por “fuerza bruta”). Nos queda el número de situaciones siguiente:

$$\binom{64}{8} = 4.426.165.368 \text{ situaciones hasta llegar a solución.}$$

Una **primera mejora** consiste en no poner nunca más de una reina en una fila. Reduce la representación del tablero a un vector de ocho elementos, cada uno de los cuales da la posición de la reina dentro de la fila correspondiente.

Quedaría:

	1						8
1							
8							

donde:

$v[i]$ : Indica la fila en la que está la reina en la  $i$ -ésima columna.

En esta mejora el número de situaciones que hay que considerar será:

$$8^8 = 16.277.216$$

Una **segunda mejora** consiste en hacer lo mismo que antes sólo que en las columnas. Ahora representaremos el tablero mediante un vector formado por ocho números diferentes entre 1 y 8, es decir, mediante una permutación de los ocho primeros números enteros.

Por lo que el número de situaciones posibles es:

$$8! = 40.320$$

Usaremos el siguiente algoritmo:

```

proc perm (i)
  si  $i = n$  entonces usar (T)      { T es una nueva permutación }
  si no
    para  $j = i$  hasta n hacer
      intercambiar  $T[i]$  y  $T[j]$ 
      perm ( $i + 1$ )
      intercambiar  $T[i]$  y  $T[j]$ 

```

Si se utiliza el algoritmo anterior para generar las permutaciones sólo se consideran 2.830 situaciones antes de que encuentre una solución.

Una **tercera mejora** será aquélla en la que ninguna reina está en la misma diagonal. Evidentemente, el número de situaciones posibles es aún menor.

Todos estos algoritmos comparten un defecto común: nunca comprueban si una situación es una solución mientras no se hayan colocado todas las reinas en el tablero (son exhaustivos).

La vuelta atrás permite mejorar esto, utilizando otro **enfoque distinto**. Como primer paso reformulamos el problema de las ocho reinas como un problema de búsqueda en un árbol. Decimos que un vector  $V[1..k]$  de enteros entre 1 y 8 es *k-prometedor* para  $0 \leq k \leq 8$ , si ninguna de las  $k$  reinas colocadas en las posiciones  $(1, V[1]), (2, V[2]), \dots, (k, V[k])$  amenaza a ninguna de las otras.

Matemáticamente,  $V$  es *k-prometedor* si, para todo par de enteros  $i$  y  $j$  entre 1 y  $k$ , con  $i \neq j$ , tenemos que  $V[i] - V[j] \notin \{i - j, 0, j - i\}$ . Para  $k \leq 1$ , todo vector  $V$  es *k-prometedor*.

Las **soluciones** del problema de las ocho reinas se corresponden con aquellos vectores que son 8-prometedores.

**Ejemplos:**

	1							8
1	X							
		X						
8								

$V[1] - V[2] = 1 - 2 = -1$ .  $V[1,2]$  No vale, no es *k-prometedor*

	1							8
1	X							
			X					
8								

$V[1] - V[3] = 1 - 3 = -2$ .  $V[1,3]$  Vale, ya que es *k-prometedor*

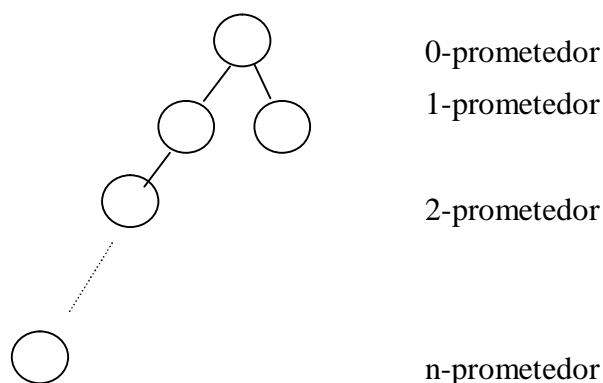
Sea  $N$  el conjunto de vectores *k-prometedores*,  $0 \leq k \leq 8$ . Sea  $G = \langle N, A \rangle$  el grafo dirigido tal que  $(U, V) \in A$  si y sólo si existe un entero  $k$ , con  $0 \leq k \leq 8$ , tal que:

- $U$  es *k-prometedor*
- $V$  es  $(k+1)$ -prometedor
- $U[i] = V[i]$  para todo  $i \in [1..k]$

Este grafo es un **árbol**. Su raíz es el árbol vacío correspondiente a  $k = 0$ . Sus hojas son o bien soluciones ( $k = 8$ ) o posiciones sin salida ( $k < 8$ ) tales como  $[1,4,2,5,8]$  (o como hemos visto previamente en los ejemplos anteriores): en tal situación, resulta imposible colocar una reina en la fila siguiente sin amenazar por lo menos a una de las reinas que ya están en el

tablero. Las soluciones del problema de las ocho reinas se pueden obtener explorando este árbol. Sin embargo, no generamos explícitamente el árbol para explorarlo después. Más bien, se generan y abandonan los nodos en el transcurso de la exploración. El recorrido en profundidad es el método evidente, sobre todo si sólo necesitamos una solución.

**Gráficamente**, puede ser algo así:



Esta técnica tiene dos **ventajas** con respecto a las anteriores:

1. El número de nodos del árbol es menor que  $8! = 40.320$ . Bastaría con explorar 114 nodos para obtener la primera solución.
2. Para decidir si un vector es  $k$ -prometedor, sabiendo que es una extensión de un vector  $(k-1)$ -prometedor, sólo necesitamos comprobar la última reina que haya que añadir.

En el procedimiento siguiente,  $sol[1..8]$  es una matriz global. Tendremos:

```

procedimiento reinas (k, col, diag45, diag135)
{  $sol[1..k]$  es  $k$ -prometedor,  $col = \{sol[i] | 1 \leq i \leq k\}$ ,
   $diag45 = \{sol[i] - i + 1 | 1 \leq i \leq k\}$  y
   $diag135 = \{sol[i] + i - 1 | 1 \leq i \leq k\}$  }
si  $k = 8$  entonces      { un vector 8-prometedor es una solución }
  escribir sol
si no
  para  $j \leftarrow 1$  hasta 8 hacer
    si  $j \notin col$  y  $j - k \notin diag45$  y  $j + k \notin diag135$  entonces
       $sol[k + 1] \leftarrow j$ 
      {  $sol[1..k + 1]$  es  $(k+1)$ -prometedor }
      reinas( $k + 1$ ,  $col \cup \{j\}$ ,  $diag45 \cup \{j - k\}$ ,
         $diag135 \cup \{j + k\}$ )

```

La llamada inicial es *reinas*  $(0, \emptyset, \emptyset, \emptyset)$ .

La *ventaja* obtenida al utilizar la vuelta atrás en lugar de un enfoque exhaustivo (la forma evidente y sus mejoras) se vuelve más pronunciada a medida que crece  $n$ . Por ejemplo, para  $n = 12$  son 479.001.600 las posibles permutaciones que hay que considerar.

### 9.6.3. El caso general.

Los algoritmos de vuelta atrás se pueden utilizar aun cuando las soluciones buscadas no tengan todas necesariamente la misma longitud. Siguiendo con el planteamiento anterior de los  $k$ -prometedores tendremos este **cuarto esquema**, que será:

```
fun vueltaatrás ( $v[1..k]$ )
  {  $v$  es un vector  $k$ -prometedor }
  si  $v$  es una solución entonces escribir  $v$ 
  si no
    para cada vector  $(k+1)$ -prometedor  $w$ 
      tal que  $w[1..k] = v[1..k]$  hacer
        vueltaatrás ( $w[1..k + 1]$ )
```

Tanto el problema de la mochila como el de las  $n$  reinas se resolvían empleando una búsqueda en profundidad en el árbol correspondiente. Algunos problemas pueden llegar a ser un grafo infinito, que como vimos antes se resolverían empleando un recorrido en anchura.

NOTA DEL AUTOR: Veremos varios ejercicios resueltos que compararán ambas técnicas, usando vuelta atrás como la conocemos (haciendo más exhaustiva la búsqueda y el grafo implícito mayor) y vectores  $k$ -prometedores (justo al contrario, el grafo implícito será menor).

IMPORTANTE: Hemos puesto cuatro posibles esquemas, los cuales son **básicos** el primero y el cuarto. Los otros dos intermedios son variaciones sobre el primero de ellos. Se recalca esto, al igual que en una nota anterior.

### 9.7. Ramificación y poda

Es otra técnica para explorar un grafo dirigido implícito y la última que veremos en este capítulo, además de ser la más complicada de entender.

Esta vez vamos a buscar la **solución óptima** de algún problema. En cada nodo calculamos una cota del posible valor de aquellas soluciones que pudieran encontrarse más adelante en el grafo. Si la cota muestra que cualquiera de estas soluciones tiene que ser necesariamente peor que la mejor solución hallada hasta el momento, entonces no necesitaremos seguir explorando esta parte del grafo.

Tendremos dos posibles implementaciones:

El **primer esquema** posible que tendremos será aquél en el que el cálculo de las cotas se combina con un recorrido en anchura, que situará la solución más cerca de la raíz (nota del autor) y las cotas solamente sirven para podar ciertas ramas de un árbol.

```
fun ramificación-y-poda (ensayo)
  p ← cola-vacía
  cota-superior ← inicializar-cota-superior
  solución ← solución-vacía
  encolar (ensayo, p)
  mientras no vacía (p) hacer
    nodo ← desencolar (p)
    si valido (nodo) entonces
      si coste (nodo) < cota-superior entonces
        solución ← nodo
        cota-superior ← coste (nodo)
      fsi
    si no { Nodo no es válido (solución) }
      para cada hijo en compleciones (nodo) hacer
        si condiciones-de-poda (hijo) y
          cota-inferior (hijo) < cota-superior entonces
          encolar (hijo, p)
        fsi
      fpara
    fsi
  fmientras
ffun
```

Nos fijamos que usamos estructura de *cola*, ya que es un recorrido en anchura, por encontrar la solución más cerca de la raíz (apreciación del autor). Veremos con algo más de detalle las distintas funciones y variables que son nuevas correspondientes a dicho esquema, el resto de funciones (compleciones, condiciones-de-poda) ya las hemos estudiado previamente en el esquema general de vuelta atrás. Por tanto, tenemos estas nuevas funciones y variables:

- **Cota-superior:** Será, en cada momento, el coste de la mejor solución encontrada hasta el momento. En el esquema anterior es una pequeña modificación respecto al mismo, en la que llaman *c* a esta variable, pero conceptualmente es similar.
- **Cota-superior-inicial:** Será aquella cota que en un primer momento estimemos. Podrá ser tanto un valor muy alto, que luego haga podar menos ramas tanto un valor cercano a la solución, en todo caso, dependerá del tipo del problema en cuestión.
- **Función cota-inferior (nodo):** Será aquél valor de cota en el nodo que se estime para alcanzar la solución.
- **Función coste (nodo):** Será aquel valor del nodo una vez alcanzada la solución. Podrá mejorar al valor de la cota-superior, ante lo cual se actualiza esta última.

El **segundo esquema** será aquél en el que el cálculo de las cotas se utilizan para seleccionar el camino que, entre los abiertos, parezca más prometedor para explorarlo primero y además como el esquema anterior para podar ramas. Será el que más empleemos en los distintos ejercicios que se nos den. Tenemos el algoritmo:

```

fun ramificación-y-poda (ensayo)
  m ← montículo-vacío
  cota-superior ← inicializar-cota-superior
  solución ← solución-vacía
  añadir-nodo (ensayo, m)
  mientras no vacío (m) hacer
    nodo ← extraer-raíz (m)
    si valido (nodo) entonces
      si coste (nodo) < cota-superior entonces
        solución ← nodo
        cota-superior ← coste (nodo)
      fsi
    si no { Nodo no es válido (solución) }
      si cota-inferior (nodo) ≥ cota-superior entonces
        devolver solución
      si no { cota-inferior (nodo) < cota-superior }
        para cada hijo en compleciones (nodo) hacer
          si condiciones-de-poda (hijo) y
            cota-inferior (hijo) < cota-superior entonces
            añadir-nodo (hijo, m)
          fsi
        fsi
      fpara
    fsi
  fmientras
ffun

```

En este caso usaremos una estructura de datos *montículo* que nos hará escoger siempre el nodo más prometedor (lo usaremos como una lista con prioridad). Es el mismo esquema que previamente, sólo que añadimos la selección del camino que nos lleve antes a solución.

Se añaden un par de líneas en este esquema que son:

```

    si cota-inferior (nodo) ≥ cota-superior entonces
      devolver solución

```

Esto significará que cuando en un nodo tengamos una cota-inferior (recordemos que es la estimación hasta encontrar la solución) igual o mayor a la cota-superior (que es el coste de la mejor solución encontrada hasta el momento) entonces no podremos llegar por ningún otro nodo del montículo a una solución mejor, por lo que dejamos de explorar el resto del grafo implícito (devolvemos la solución mejor y salimos del bucle).

Los dos esquemas anteriores para **problemas de minimización**, ya que iremos rebajando la cota superior una vez encontrada la solución si la mejora (podaremos más ramas).

NOTA DEL AUTOR: Estos dos esquemas se han sacado completamente del libro de problemas, y el siguiente es totalmente personal, a partir del anterior.

Igualmente (y como añadido del autor) podremos emplearlo para problemas de maximización como sigue (aunque lo veremos en los problemas resueltos). La única diferencia apreciable será que se actualizará la cota inferior (se incrementará) al encontrar una mejor solución y las comparaciones serán con respecto a la cota-superior del nodo, no obstante, la filosofía será similar.

```

fun ramificación-y-poda (ensayo)
  m ← montículo-vacío
  cota-inferior ← inicializar-cota-inferior
  solución ← solución-vacía
  añadir-nodo (ensayo, m)
  mientras no vacío (m) hacer
    nodo ← extraer-raíz (m)
    si valido (nodo) entonces
      si coste (nodo) > cota-inferior entonces
        solución ← nodo
        cota-inferior ← coste (nodo)
      fsi
    si no { Nodo no es válido (solución) }
      si cota-superior (nodo) ≤ cota-inferior entonces
        devolver solución
      si no { cota-superior (nodo) > cota-inferior }
        para cada hijo en compleciones (nodo) hacer
          si condiciones-de-poda (hijo) y
            cota-superior (hijo) > cota-inferior entonces
            añadir-nodo (hijo, m)
          fsi
        fpara
      fsi
    fmientras
  ffun

```



### 9.7.1. El problema de la asignación

Hay que asignar  $n$  tareas a  $n$  agentes, de forma que cada agente realice exactamente una tarea. Si el agente  $i$ , con  $1 \leq i \leq n$ , se le asigna la tarea  $j$ , con  $1 \leq j \leq n$ , entonces el coste de realizar esta tarea será  $C_{ij}$ . Dada la matriz de costes completa, el problema consiste en asignar tareas a los agentes de tal manera que **minimice** el coste de ejecución de las  $n$  tareas.

Veremos un **ejemplo** de este problema, en la que se nos da esta matriz de costes:

	Tareas			
		1	2	3
	a	4	7	3
	b	2	6	1
	c	3	9	4

Si asignamos la tarea 1 al agente a, la tarea 2 al agente b y la tarea 3 al agente c, nuestro coste total será:  $4 + 6 + 4 = 14$ . La asignación óptima es  $a \rightarrow 2, b \rightarrow 3$  y  $c \rightarrow 1$ , cuyo coste es  $7 + 3 + 1 = 11$ .

Este problema tiene numerosas aplicaciones. En general, con  $n$  agentes y  $n$  tareas, hay  $n!$  posibles asignaciones que considerar, que son demasiadas incluso para valores moderados de  $n$ . Por tanto, recurriremos a la ramificación y poda.

Veremos otro **ejemplo** más e iremos paso a paso resolviéndolo. De nuevo, la matriz de costes será:

	Tareas				
		1	2	3	4
	a	11	12	18	40
	b	14	15	13	22
	c	11	17	19	23
	d	17	14	20	28

El **primer paso** es obtener la cota superior inicial del problema:

1ª solución posible: Diagonal principal:  $11 + 15 + 19 + 28 = 73$ .

2ª solución posible: Diagonal secundaria:  $40 + 13 + 17 + 17 = 83$ .

La segunda solución posible no supone mejora respecto a la primera. Por tanto, tomamos la primera de ellas como *cota superior*.

El **segundo paso** es obtener la cota inferior del problema. Para ello sumamos los elementos más pequeños. En este ejemplo calcularemos dos posibles cotas inferiores (son estimaciones, no son soluciones reales):

1ª cota inferior: Será aquella que sale de asignar a cada agente la tarea que mejor sabe hacer. En este caso, inicialmente tendremos  $11 + 12 + 13 + 22 = 58$

2ª cota inferior: Será aquella que sale de asignar a cada tarea el agente que mejor la realiza. Por lo que tendremos  $11 + 13 + 11 + 14 = 49$ .

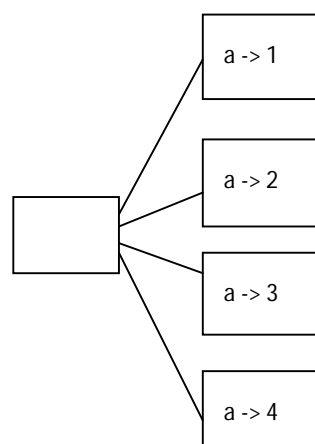
En este caso, al ser un problema de minimización, tendremos que la segunda cota inferior será peor que la primera, ya que la solución óptima no puede ser mejor que 49, por ser imposible este resultado. Por ello, tendremos la mayor de las cotas inferiores.

La cota donde, por tanto, estará ubicada la solución será la del intervalo  $[58..73]$ .

NOTA DEL AUTOR: Al ser problema de minimización la cota inferior será la mayor de las cotas inferiores, a diferencia de los problemas de maximización que será la menor de las cotas superiores y calcularíamos las dos posibles cotas superiores.

Realizaremos el cálculo de las cotas inferiores de manera recursiva hasta encontrar una solución válida, que nos resuelva el problema.

Exploraremos un árbol cuyos nodos conexos corresponden a asignaciones parciales. En la raíz del árbol no se han hecho asignaciones. En lo sucesivo, en cada nivel se determina la asignación de un agente más. Para cada nodo, calculamos una cota de las soluciones que se pueden obtener completando la asignación parcial correspondiente y utilizar esta cota para cerrar caminos y guiar la búsqueda. Asignamos la tarea al agente a, por lo que nos queda:

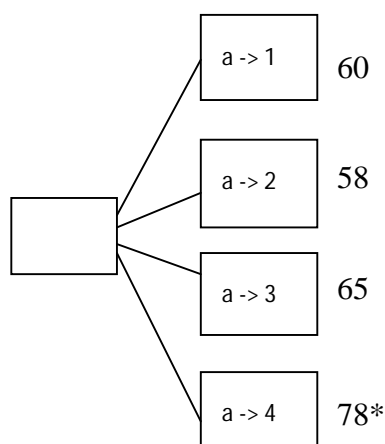


Las cotas inferiores para la asignación  $a \rightarrow 1$  serán:

1ª cota inferior:  $11 + 13 + 17 + 14 = 55$  (asignar a cada agente la tarea).

2ª cota inferior:  $11 + 14 + 13 + 12 = 60$  (asignar a cada tarea el agente).

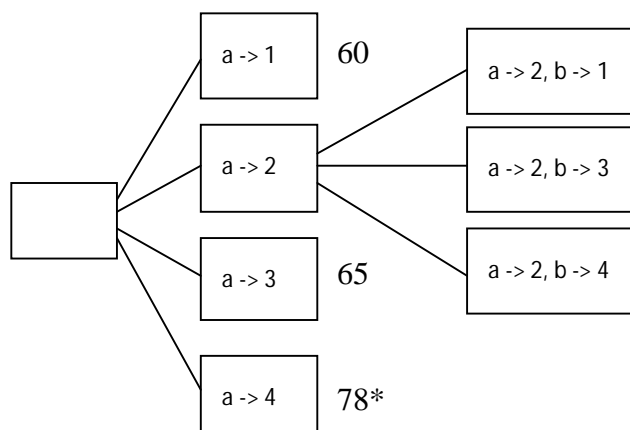
Cualquier solución que se pueda alcanzar por esta rama va a tener coste máximo de 60. Por tanto, calculando igual para el resto de asignaciones tendremos el siguiente árbol:



El asterisco (\*) indica que la rama se poda por superar el valor máximo del intervalo antes puesto, por ello, no se seguirá explorando. Lo denominaremos **nodo muerto**.

Seguimos por la rama de cota inferior más baja: 58, que es la rama más prometedora. Las ramas que se pueden seguir explorando las denominaremos **nodo vivo**.

Seguimos por la rama  $a \rightarrow 2$ , que como hemos puesto es la rama de cota inferior más baja (la que nos optimice la solución) como sigue:

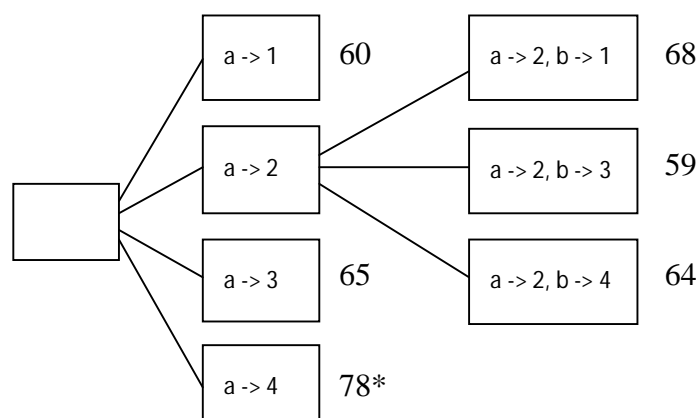


Continuamos haciendo el mismo procedimiento de antes y llegamos a calcularlo para los nodos que salen de  $a \rightarrow 2$ . Por ejemplo,  $a \rightarrow 2, b \rightarrow 1$ :

1ª cota inferior:  $a \rightarrow 2, b \rightarrow 1, c \rightarrow 3, d \rightarrow 3$ :  $12 + 14 + 19 + 20 = 65$  (asignar a cada agente la tarea).

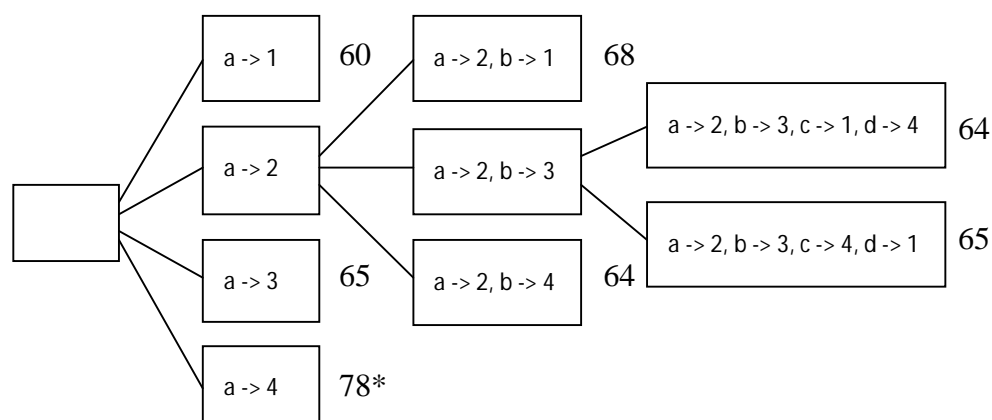
2ª cota inferior:  $a \rightarrow 2, b \rightarrow 1, c \rightarrow 3, c \rightarrow 4$ :  $12 + 14 + 19 + 23 = 68$  (asignar a cada tarea el agente).

La cota inferior está limitada por el valor 68. Calcularemos de igual manera las demás cotas. Por lo que tendremos:



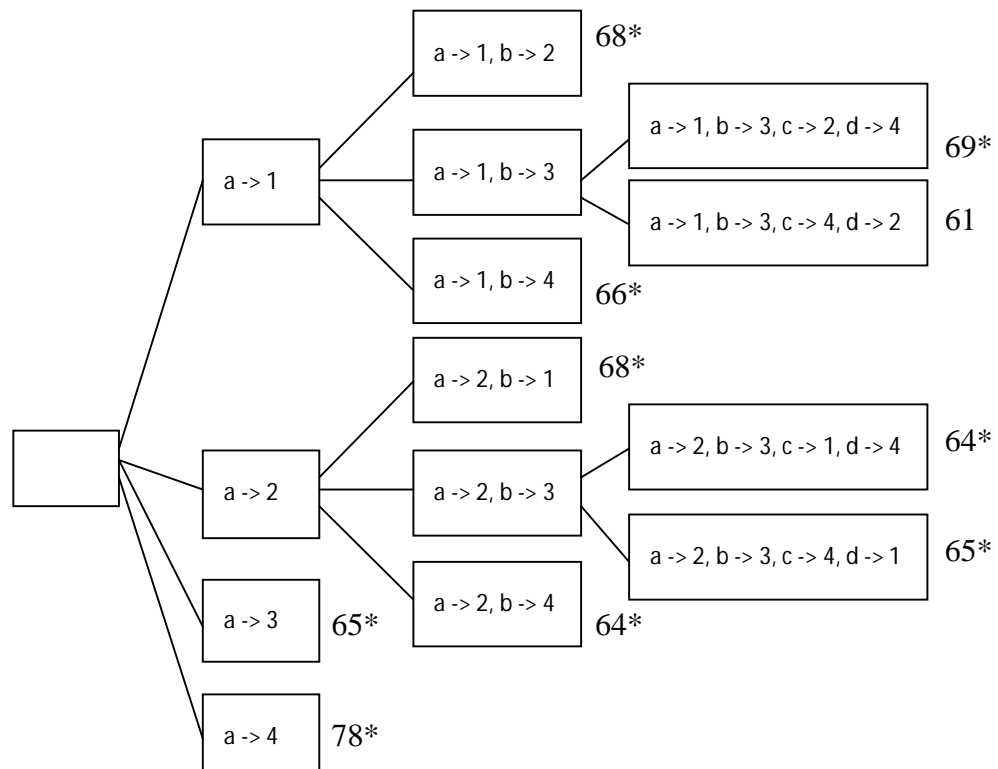
NOTA DEL AUTOR: Hemos visto que el valor 59 es la menor estimación de cota inferior mejorando a la del intervalo, ya que el coste de la solución nunca será inferior a 59. Por ello se actualiza el intervalo a  $[59..73]$ . No está hecho así en el ejercicio, simplemente es para comprenderlo.

De nuevo, seguimos por  $a \rightarrow 2, b \rightarrow 3$ , que nos quedará:



Ya encontramos dos posibles soluciones. Como es similar al paso anterior, nos evitamos hacer otro grafo de nuevo y ponemos las cotas inferiores. Puesto que la primera solución mejora a la del intervalo será nuestra nueva cota inferior, quedando  $[59..64]$ .

Inmediatamente, podemos las ramas tras la actualización que sean superiores a 64. Sólo queda una rama menor de 64, que es la de  $a \rightarrow 1$ . Por tanto, desarrollando ese árbol como hemos hecho antes, tendremos lo siguiente:



Como no quedan más ramas que explorar la solución  $a \rightarrow 1, b \rightarrow 3, c \rightarrow 4, d \rightarrow 2$  es la **óptima**. En este algoritmo lo hemos resuelto siguiendo nuestro segundo esquema (aquél que guiaba la búsqueda).

#### 9.7.2. El problema de la mochila (4)

Se nos pide **maximizar**  $\sum_{i=1}^n x_i * v_i$  sometido a la restricción  $\sum_{i=1}^n x_i * w_i \leq W$ , donde  $v_i$  y  $w_i$  son estrictamente positivas y los  $x_i$  son enteros no negativos. Resolveremos este problema por ramificación y poda.

Las variables están numeradas de  $\frac{v_i}{w_i} \geq \frac{v_{i+1}}{w_{i+1}}$ . Si los valores de  $x_1, x_2, \dots, x_k, 0 \leq k \leq n$  quedan fijados, con  $\sum_{i=1}^n x_i * w_i \leq W$ , es fácil ver que el valor que se puede obtener sumando más elementos de los tipos  $k + 1, \dots, n$  a la mochila no puede sobrepasar el valor:

$$\underbrace{\sum_{i=1}^k x_i * v_i}_{\text{Elemento ya en la mochila}} + \underbrace{\left( W - \sum_{i=1}^k x_i * w_i \right) * \frac{v_{k+1}}{w_{k+1}}}_{\text{Cota del valor que se puede añadir}}$$

Para resolver el problema de ramificación y poda, exploramos un árbol en cuya raíz no está fijado el valor de ninguno de los  $x_i$  y en cada nivel sucesivo se va determinando el valor de una variable más, por orden numérico de variables. En cada nodo que exploremos, sólo generamos aquellos sucesores que satisfagan la restricción de peso, de tal manera que cada nodo tiene un número finito de sucesores. Siempre que se genera un nodo, calculamos una cota superior del valor de la solución que se puede obtener la carga

parcialmente especificada y utilizando estas cotas para cortar ramas inútiles y guiar la exploración del árbol.

NOTA DEL AUTOR: Hay una errata del libro en la que el símbolo de la restricción es mayor o igual, cuando debería ser menor o igual, por lo que la errata se ha subsanado anteriormente.

### 9.7.3. Consideraciones generales

Usaremos **montículos** para almacenar una lista de nodos generados pero no explorados. No se dispone de una formulación recursiva elegante de ramificación y poda.

Resulta imposible dar una idea precisa de lo bien que se va a comportar esta técnica en un problema, empleando una cota dada. Hay que llegar a un compromiso en respuesta a la cota calculada. Si es cota mejor examinaremos menos nodos y llegaremos a la solución óptima más rápidamente si hay suerte. Pero podemos pasar mucho tiempo calculando la cota correspondiente.

En el caso peor, tendremos una cota excelente, pero no podamos ninguna rama, por lo que desperdiciamos tiempo. En la práctica, casi siempre es rentable invertir el tiempo necesario para calcular la mejor cota posible.