ESQUEMA DE VUELTA ATRÁS: Backtracking

1. INTRODUCCIÓN
2. LOS ESQUEMAS BÁSICOS
3. MOCHILA ENTERA: 2 versiones
4. MARCAJE
5. PODA BASADA EN EL COSTE DE LA MEJOR SOLUCIÓN EN CURSO
6. EL VIAJANTE DE COMERCIO
7. BACKTRACKING ITERATIVO

María Teresa Abad
Mayo 2005

1. INTRODUCCIÓN

Caracterización de los 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.

Método de resolución

Vuelta Atrás es un esquema que de forma sistemática y organizada, genera y recorre un espacio que contiene *todas* las posibles secuencias de decisiones. Corresponde a un recorrido en profundidad o en preorden.
Este espacio se denomina el *espacio de búsqueda* del problema o el *espacio de soluciones*.

Primera implicación: Si existe solución, seguro que la encuentra.

El espacio de búsqueda (EB)

- Dimensiones:

- la altura del espacio: hay *k* decisiones que tomar para formar una solución.
- la anchura del espacio: cada decisión tiene asociado un dominio formado por *j* valores distintos.

- Topología:

Habitualmente el espacio de búsqueda es un *árbol*, aunque puede ser un grafo como en el caso de los grafos de juego.

- Terminología:

- Todos los nodos que forman parte de cualquier camino que va desde la raíz del EB a cualquier nodo del EB representan una *secuencia de decisiones*.
- Una secuencia de decisiones es *factible* si no viola las restricciones.
- Una secuencia de decisiones 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 el último nodo de la secuencia es una hoja del EB.
- Para muchos problemas se tiene que una *solución* es cualquier secuencia de decisiones factible y no_prolongable $\Rightarrow$ 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.
- 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*.

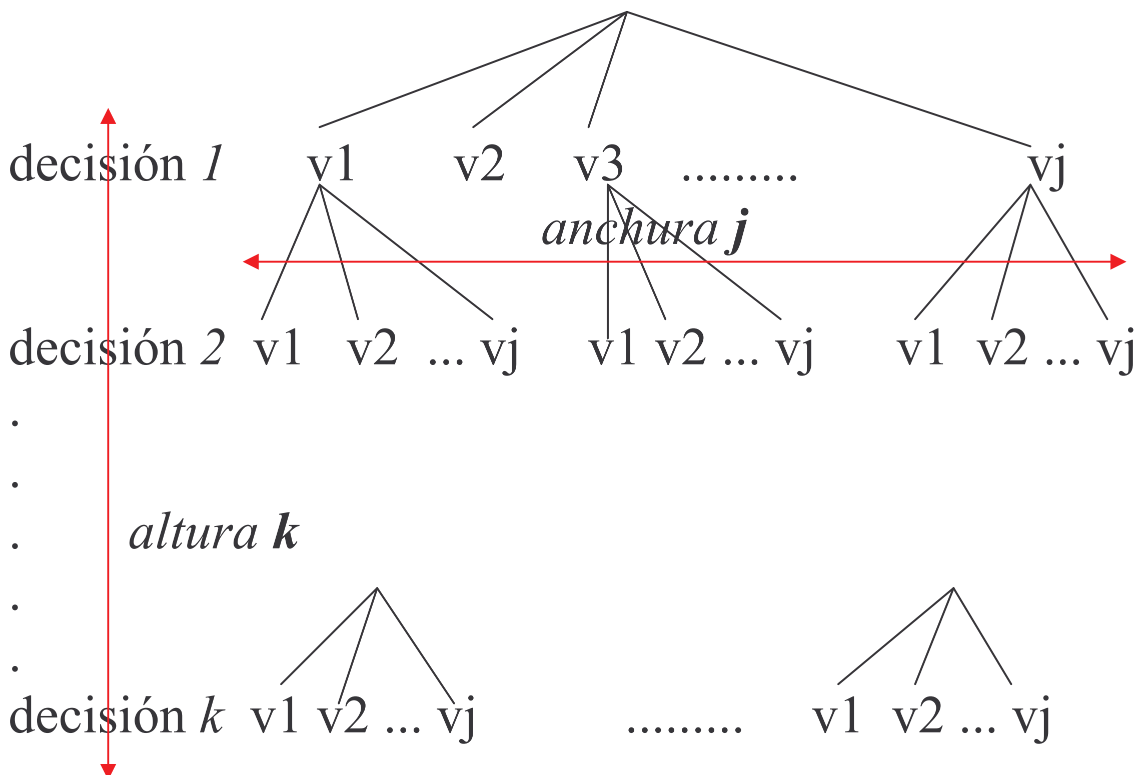
- Coste:

El espacio de búsqueda de la figura tiene

- j^k hojas y
- su número de nodos es

$$n_nodos = \sum_{i=0}^{k} j^i \in O(j^k)$$

El tiempo necesario para recorrerlo es del mismo orden. *Segunda implicación:* el coste exponencial en el caso peor de Vuelta Atrás.



- Elección del EB:

- 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.

2. LOS ESQUEMAS BÁSICOS

El esquema

- Vuelta Atrás hace un recorrido en profundidad del espacio de búsqueda partiendo de la raíz.
- 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 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. Son recorridos **ciegos** porque fijado un nodo del espacio se sabe cual es el siguiente que se va a generar.

- Operaciones sobre el EB:
preparar_recorrido, *existan_hermanos* y *siguiente_hermano*.

- Todas 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.

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]*.

2.1. UNA SOLUCIÓN

```
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;
        /* análisis de la nueva decisión x[k] */
        [    factible(x,k) --->
                [    solucion(x,k)  --- >
                        /* solución encontrada */
                        tratar_solución(x);
                        <sol,b> :=<x,CIERTO>;
                        sol := x;
                [] ¬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).

Otra versión con una precondición más débil (sigue el esquema de las transparencias en C++)

```
función BACK_1SOLV ( x es solución; k es nat )
dev ( b es bool; sol es solución )
{ Pre : (x [1..k-1] es factible }

   b:= FALSO;
   sol:= sol_vacia;
   [    solucion(x,k)  --- >
          /* solución encontrada */
          tratar_solución(x);
          <sol,b> :=<x,CIERTO>;
          sol := x;
   [] ¬solucion(x,k) --- >
        preparar_recorrido_nivel_k;
        *[ existan_hermanos_nivel_k  ∧ ¬b --->

                x[k] := sig_hermano_nivel_k;
                /* análisis de la nueva decisión x[k]  */
                [    factible(x,k) --->
                       <sol,b> := BACK_1SOLV(x, k+1);
                [] ¬factible(x,k) ---> seguir
                ]
        ]
   ]
   dev ( sol, b )
ffunción
```

2.2. TODAS LAS SOLUCIONES

```
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).

2.3. 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 $\wedge$ no es solución) $\wedge$ $1 \leq k \leq$ altura(espacio búsqueda) }*

```
<xmejor,vmejor> := <sol_vacia, 0 > ;
/* inicialmente la mejor solución está vacía */
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)
              ]
```

```
                /* actualización de la mejor solución en
                curso */
                [  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).

3. MOCHILA ENTERA

Ya hemos resuelto de forma óptima el problema de la mochila fraccionada con un algoritmo voraz. Pero no se ha encontrado una solución voraz para mochila entera. La formalización del problema a resolver se puede presentar de la siguiente forma.

$$[\text{MAX}] \sum_{i=1}^{n} x(i) \cdot v(i),$$

sujeto a

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

****

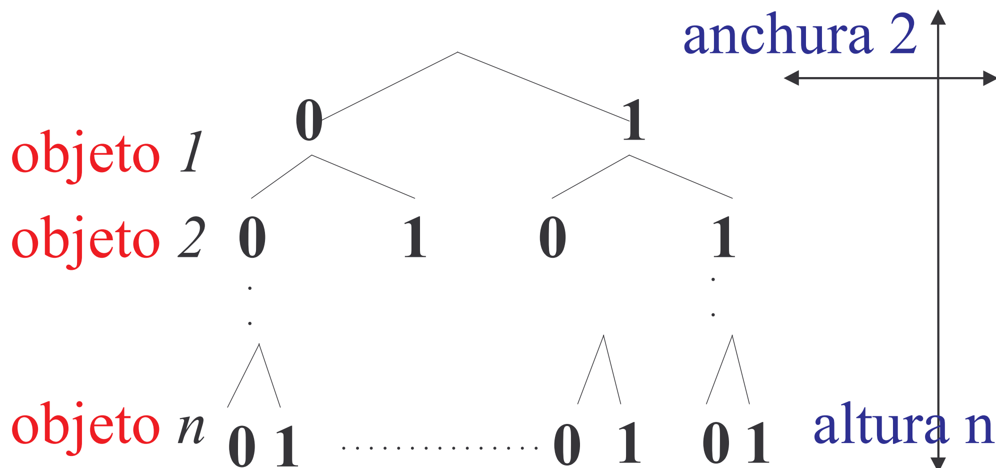
Alternativa A : **Espacio de búsqueda binario y solución ≡ factible ∧ no_prolongable**

****

<u>tipo</u> solución <u>es</u> vector [1..n] de {0,1};
<u>var</u> 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)

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 $O(2^n)$.



función **MOCHILA-BINARIO** (x es solución; k es nat) dev (xmejor es solución;
vmejor es valor (xmejor))

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

```
<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 */
    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*

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

    dev (xmejor, vmejor)
ffunción

La Primera llamada a la función será :

<s,v> := MOCHILA-BINARIO (sol_vacia, 1)

El coste es $O(2^n)$ debido al tamaño máximo del espacio de búsqueda.
Las funciones SUMA_PESO y SUMA_VALOR requieren coste $\theta(n) \Rightarrow$ Coste total del algoritmo es $O(n \cdot 2^n)$.

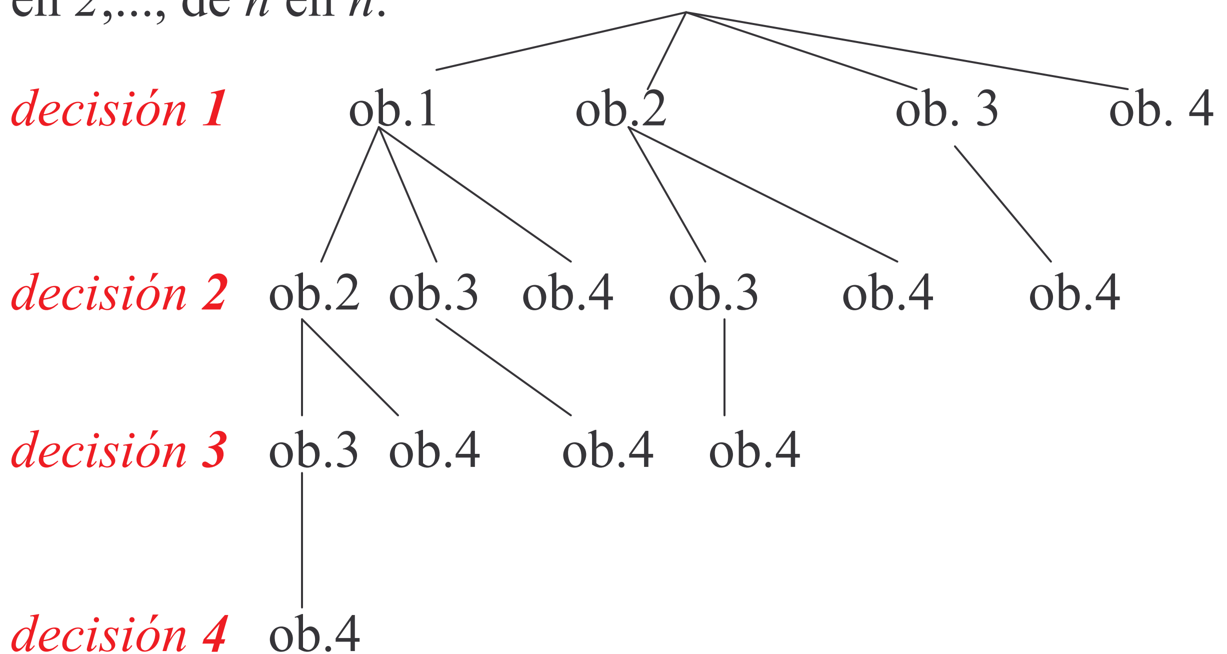
****

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

****

La solución es una secuencia que contiene los índices de objetos que se han podido empaquetar en la mochila sin violar la restricción de peso ⇒ *secuencia de tamaño variable.*

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.



Espacio de búsqueda para n=4. El tamaño del espacio sigue siendo del orden de $O(2^n)$.

El espacio es de tamaño mínimo porque :

- no se repite ninguna combinación, aunque sea en orden distinto.
- no se intenta colocar el mismo objeto más de una vez.

En esta alternativa una solución se puede encontrar en cualquier camino que va 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. *** es la usada en el algoritmo ***

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.

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

- x[i]=j indica que en *i*-ésimo lugar se ha colocado en la mochila el objeto identificado por el índice *j*.
- 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))

{ Pre: $\left(\sum_{i=1}^{k-1} p(x(i)) \leq PMAX \right) \wedge (x(k-1) \neq n) \wedge (1 \leq k \leq n)$ *}*

```
<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;
                  /* no es factible, viola las restricciones */
         ]
    ] /* fin del bucle que recorre los hijos de x[k-1] */
{ 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. MARCAJE

Técnica de marcaje ≡ Inmersión de eficiencia.
Idea: Un nodo aprovecha el trabajo que ya se ha hecho en otros nodos (su padre o sus hijos).

Algoritmo: TODAS LAS SOLUCIONES CON MARCAJE

```
función BACK_TODAS_MARC ( 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_MARC (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
```

Si marcamos en [1a] hay que desmarcar en [2a]. Lo mismo para la opción b.

Mochila binaria

función **MOCHILAB_MARC** (x es solución; k, pac, vac es nat) dev (xmejor es solución, vmejor es valor (xmejor))

$$\{ \textbf{Pre} : \left(\sum_{i=1}^{k-1} x(i)\cdot p(i) \leq PMAX \right) \wedge \left(pac = \sum_{i=1}^{k-1} x(i)\cdot p(i) \right) \wedge$$

$$\left(vac = \sum_{i=1}^{k-1} x(i)\cdot v(i) \right) \wedge (1 \leq k \leq n) \}$$

```
   <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_MARC( x, k+1, pac, vac);
              ]
              [ vmejor ≥ val ---> seguir
              [] vmejor < val --->
                      <xmejor, vmejor> := <sol, val>;
```

```
            ]

        [] pac >PMAX ---> seguir;
                        /* viola las restricciones */
        ]
        pac := pac - x(k) · p(k);   /* desmarcar */
        vac := vac - x(k) · v(k);
         /* realmente en este caso 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*

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

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

Primera llamada

<s,v> := MOCHILAB_MARC (sol_vacia, 1, 0, 0).

El coste de este algoritmo es ahora $O(2^n)$ y se ha rebajado gracias al marcaje.

5. 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.

2 PODAS:
- Poda de Factibilidad (
- Poda basada en el coste de la mejor solución en curso -PBCMSC-

Idea de la PBCMSC
El camino en curso no se sigue explorando cuando, pese a ser factible y prolongable, no se puede mejorar la mejor solución en curso que se tiene aunque se continúe el camino en curso de todas las formas posibles.

Ejemplo : Mochila entera con PMAX=50 y n=6.

Sea x[1..4] = <1,0,1,0> tal que
peso(x[1..4])=20 y
valor(x[1..4])=30

Supongamos que la mejor solución en curso tiene un valor, *vmejor* = 100.
Supongamos que **valor[5]=10** y **valor[6]= 5**.
La solución de valor máximo que se puede conseguir expandiendo la solución en curso es:

x[1..6] =<1,0,1,0,**1**,**1**> que tiene un
valor = valor(x[1..4])+ **valor[5]** + **valor[6]** = 45.

- El valor de esta nueva solución no mejora el de la mejor solución en curso.

- Hemos perdido el tiempo generando esa solución, hubiera sido mejor no expandir la solución en curso.

⇒ Si supiéramos *a priori* que con todas las formas posibles de completar la solución en curso no se consigue superar a *vmejor*, no seguiríamos expandiendo la solución en curso!!!!!, no se recorrería un buen trozo del EB y haríamos *poda basada en el coste de la mejor solución en curso*.

<u>Problema</u>: saber *a priori* ,y con poco coste, el valor de la mejor solución que se puede lograr completando la solución en curso.

Soluciones:

- Disponer lo antes posible de la mejor solución en curso <xmejor, vmejor> :

  ⇒ En vez de dejar que Backtracking encuentre la primera solución, un algoritmo externo puede proporcionarle una solución desde el momento inicial.

⇒ La poda comenzará a funcionar sobre los primeros niveles del árbol.

- Disponer de una función que prediga el valor de la mejor solución que se puede obtener a partir de un punto dado del EB.

⇒ Nos contentamos con una función que devuelva una estimación del mejor valor y que calcule:

- una **cota inferior** en caso de **minimización** :

... Como mínimo necesitas gastar 5...

Si ya llevas gastado 7 y la mejor solución tiene valor 9, no merece la pena continuar por ahí !!! – PODAR –

Si ya llevas gastado 7 y la mejor solución tiene valor 15, merece la pena continuar !!! – NO PODAR –

- una **cota superior** en caso de **maximización** :

... Como máximo vas a ganar 5 ...

Si llevas ganado 7 y la mejor solución tiene valor 15, no merece la pena continuar por ahí !!! – PODAR –

Si llevas ganado 7 y la mejor solución tiene valor 10, merece la pena continuar !!! – NO PODAR –

Esta función se denomina HEURISTICO o FUNCION DE ESTIMACION y no debe engañar (devuelve una cota real) y ha de ser barata de calcular.

Resumiendo, dado un problema al que se le quiera aplicar PBCMSC,

1º/ Hay que buscar una función de estimación

- que no engañe,
- que cueste poco de calcular y
- que sea muy efectiva (que pode mucho) y,

2º/ Hay que calcular una solución inicial para que la poda actué 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.

función **BACKM_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 =cota inferior del mejor valor que se puede
conseguir desde k+1 a n */
        [ factible(x,k) ∧
          (COSTE(x,k) + est) < vmejor --->
               /* la solución en curso es prometedora */
               [solucion(x,k) --->
                   <sol,val> := < x, valor(x) >
               []¬solucion(x,k) --->
<sol,val>:=BACKM_P(x,k+1,xmejor,vmejor,VSOLI);
               ]
```

```
/* actualización de la mejor solución en curso */
                  [ 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> := backm_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.

Mochila Binaria con PBCMSC

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**:* $\left(\sum_{i=1}^{k-1} x(i)\cdot p(i) \le PMAX\right) \wedge \left(pac = \sum_{i=1}^{k-1} x(i)\cdot p(i)\right) \wedge$

$\left(vac = \sum_{i=1}^{k-1} x(i)\cdot v(i)\right) \wedge (1 \le k \le n) \wedge$

***xini** es la mejor solución en curso ∧ **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 */
            /* MARCAR */
            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.
Atención : los objetos se consideran en orden
decreciente de cociente valor/peso */
            est := MOCHILAG_FR ( x, k, pac, PMAX );
            [ (pac ≤ PMAX) ∧ (vac+est >vmejor) --->
/* la solución en curso es factible y prometedora */
                  [ 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 )
                  ]
                  /* actualización de la mejor solución  */
                  [  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*

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

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

<u>ffunción</u>

- En la primera llamada a **MOCHILAB_P**, los valores de *xini* y *vini* se pueden obtener con el algoritmo Voraz que para mochila fraccionada. 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 (MOCHILAG_FR).
- 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.

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.

6. EL VIAJANTE DE COMERCIO

Ciclos Hamiltonianos o "The salesman problem"

Sea G=(V,E) un grafo dirigido y etiquetado, con etiquetas pertenecientes a los naturales, implementado en una matriz C tal que C[i,j] contiene el coste de la arista que va de i a j.
Un ciclo hamiltoniano es un ciclo que contiene todos los vértices de G una sóla vez, excepto el primero y el último que coinciden. Se desea obtener el tour o ciclo hamiltoniano de coste mínimo.

****

Alternativa A: **Un tour será una permutación de los n nodos (hay que pasar por todos ellos y volver al punto de partida .**

****

EB: Hay que generar todas las permutaciones.

Como hay que pasar por todos los nodos, podemos fijar el nodo 1 como inicio del tour y tenemos:

1

k=2 2 3 ← *n-1* → n

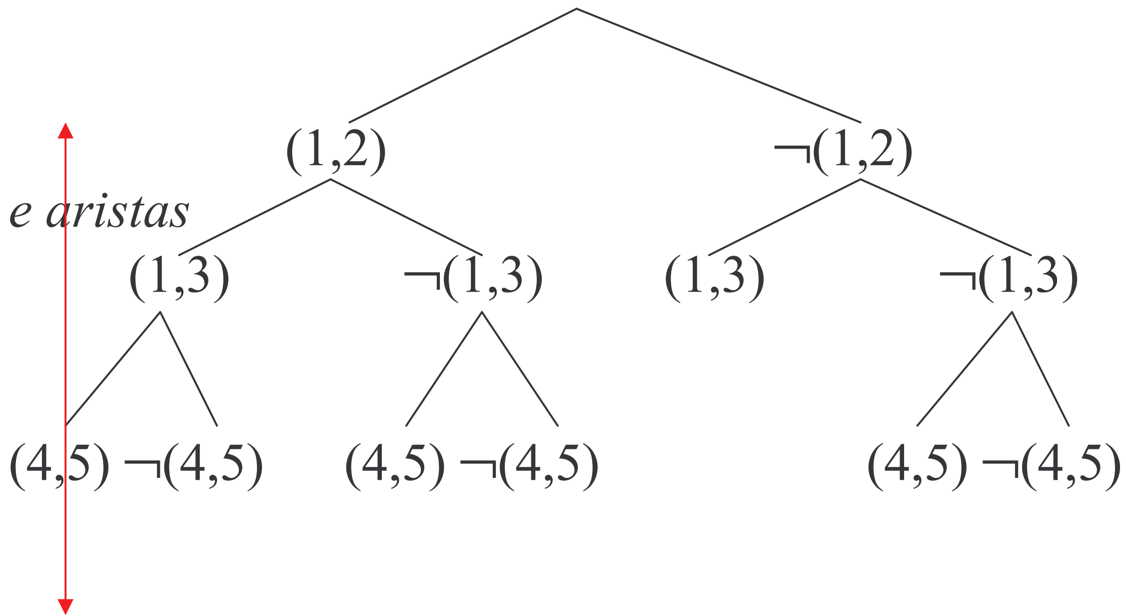
k=2 3 n 2 4 n ← *n-2* → 2 n-1

k=n n ← *1* → 2

****

Alternativa B: **Un tour está compuesto por aristas y una arista dada pueda estar o no en la solución.**

****



Tenemos un EB binario pero la función factible ha de controlar más cosas que en la alternativa anterior:

- la arista que se añade no debe formar un ciclo.
- sólo se puede entrar y salir una vez de cada uno de los nodos.
- el único ciclo permitido ha de contener n aristas y a los n nodos.

Algoritmo para la alternativa *A*

Factible: Existe arista desde el último nodo de la solución en curso al que estamos considerando.
Solución: Sólo cuando estamos en una hoja.

```
función TSP( x es solución; k es nat; ltour es nat;
lhnos es lista(nodos) )  dev ( xmejor es solucion;
vmejor es valor (xmejor) )
{ Pre : No se repite ningún vértice en x[1..k-1]∧ lhnos
= lista de nodos que se pueden alcanzar desde x(k-1) ∧
ltour = longitud_camino(x[1..k-1]) }
   <xmejor,vmejor> := <sol_vacia,∞ > ;
   i:=1 ;
   *[ i ≤ n-k+1   --->
       x[k] := primero(lhnos);  avanzar(lhnos);
       i:=i+1;
       [ ∃arista(C,x[k-1], x[k]) --->
               /* MARCAR */
               ltour := ltour + C[x[k-1], x[k]];
               [  k= n --->
                     val := ltour + C[x[n],x[1]];
                     sol:= x;
               [] k< n --->
                     <sol,val>:=TSP(x,k+1,ltour,lhnos);
               ]
```

```
/* actualización de la mejor solución en curso */
                    [ vmejor ≤ val  ---> seguir
                    [] vmejor > val --->
                              <xmejor, vmejor> := <sol, val>;
                    ]
                    /* DESMARCAR */
                    ltour := ltour - C[x[k-1], x[k]];

          [] (¬∃arista(x[k-1], x[k]) ----> seguir
          ]
          lhnos := añadir ( lhnos, x[k]);
    ]
{ Post : xmejor es la mejor solución encontrada y
vmejor es su valor }
    dev ( xmejor, vmejor )
ffunción
```

La primera llamada será :

```
x(1,2,...,n)=<1,0,0,...,0>;
k=2;
ltour = 0;
lhnos =<2,3,...,n>
<xm,vm> := TSP ( x, k, ltour, lhnos).
```

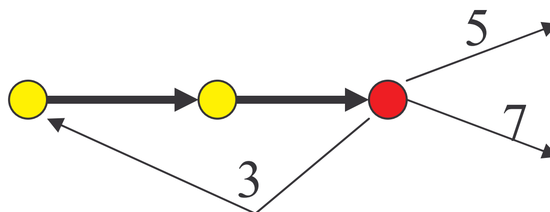
Versión con PBCMSC

Hay que definir los dos ingredientes:

1. Cómo calcular una solución inicial para dársela al backtracking.
2. Determinar la función de estimación a utilizar.

Ingrediente 1:
Utilizar un algoritmo voraz que siempre salga del último nodo del tour en curso por la arista de peso mínimo.
La arista seleccionada se podrá añadir al tour en curso si:

- No provoca un ciclo, es decir, no llega a un nodo que ya forma parte del camino.



  En este ejemplo, la función de selección elegiría la arista de peso 3 pero sería rechazada porque provoca un ciclo. Finalmente la arista elegida sería la de peso 5.

- Excepción: si en el camino tenemos n-1 aristas entonces forzosamente hay que seleccionar la arista que va del nodo n-ésimo al primer nodo del tour.

Coste del algoritmo voraz (suponiendo que las aristas que salen de un nodo están en orden creciente de peso) es O(n+e).

Ingrediente 2:
La función de estimación debe proporcionar una cota inferior de la longitud del resto de tour que queda por recorrer incluyendo la vuelta al nodo inicial.

Opción 1: Calcular el MST para los nodos que todavía no forman parte del camino.

- Habrá que incluir en el cálculo el último nodo del camino.
- Habrá que añadir al valor del MST el valor de una arista que permita alcanzar el primer nodo del camino: será la arista de peso mínimo que entra en el nodo 1.

Opción 2: Usar la siguiente función:

- Para todos los nodos que todavía **no forman parte** del camino sumar el peso de la arista de peso mínimo que entra y el peso de la arista de peso mínimo que sale. El resultado de la suma se divide entre 2. *"Pasar por un vértice cuesta como mínimo la mitad del mínimo de llegar y el mínimo de salir"*.
- Para el **último nodo** del tour en curso sólo hay que salir, por tanto, la arista de peso mínimo que sale dividido entre 2.
- Para el **primer nodo** del tour en curso sólo hay que entrar, por tanto, la arista de peso mínimo que entra dividido entre 2.

Se puede refinar y elegir la arista de peso mínimo que entre/salga y que además no forme ciclo y que enlace con vértices que no formen parte del camino, etc.

Ejemplo: Sea G un grafo completo con 5 vértices con la matriz de distancias siguiente:

0	*14*	*4*	*10*	*20*
14	*0*	*7*	*8*	*7*
4	*5*	*0*	*7*	*16*
11	*7*	*9*	*0*	*2*
18	*7*	*17*	*4*	*0*

Buscamos el camino más corto que sale del vértice 1, pasa por todos los otros nodos 1 sola vez y regresa al vértice 1.

Supongamos que el camino en curso sea {1,2}. Una cota inferior del coste del tour mínimo es:

- Coste de ir de 1 a 2: 14
- Salir de 2 hacia 3,4 ó 5: mínimo 7/2
- Pasar por 3 sin venir de 1 ni llegar a 2: mínimo 11/2. *(7+ 4) /2*
- Idem. para 4: mínimo 3. *(4+2)/2*
- Idem. para 5: mínimo 3.
- Llegar a 1 desde 3, 4 ó 5: mínimo 2.

TOTAL = 14 + 7/2 + 11/2 + 3 + 3 + 2 = 31.

función **TSP_PBC** (x es solución; k es nat; ltour es nat; lhnos es lista(nodos); xini es solución, vini es valor(xini)) dev (xmejor es solucion; vmejor es valor (xmejor))

*{ **Pre** : No se repite ningún vértice en x[1..k-1]∧ lhnos = lista de nodos que se pueden alcanzar desde x(k-1) ∧ ltour = longitud_camino(x[1..k-1]) ∧ **xini** es la mejor solución en curso ∧ **vini** es el valor de xini}*

```
    <xmejor,vmejor> := <xini,vini> ;
    i:=1 ;
    *[ i ≤ n-k+1    --->
         x[k] := primero(lhnos);  avanzar(lhnos); i:=i+1;
         [ ∃arista(x[k-1], x[k]) ---> /* factible */
                  /* MARCAR */
                  ltour := ltour + C[x[k-1], x[k]];
                  est := ESTIMACION(x,1,k,ltour);
/* est ya incluye el coste de llegar del nodo 1 al k*/
                  [ est <vmejor ---> /*prometedor*/
                       [  k= n --->
                              val := ltour + C[x[n],x[1]];
                              sol:= x;
                       [] k< n --->
<sol,val>:=TSP_PBC(x,k+1,ltour,lhnos,xmejor,vmejor);
                       ]
    /* actualización de la mejor solución en curso */
                       [  vmejor ≤ val  ---> seguir
                       [] vmejor > val --->
                            <xmejor, vmejor> := <sol, val>;
                       ]
```

```
                []est ≥vmejor --->/*no prometedor*/
                ]
                /* DESMARCAR */
                ltour := ltour - C[x[k-1], x[k]];
          [] (¬∃arista(x[k-1], x[k]) ----> seguir
          ]
          lhnos := añadir ( lhnos, x(k));
   ]
{ Post : xmejor es la mejor solución encontrada y
vmejor su valor}
   dev ( xmejor, vmejor )
ffunción
```

La primera llamada será :

```
x(1,2,...,n)=<1,0,0,...,0>;
k=2;
ltour = 0;
lhnos =<2,3,...,n>;
<xini,vini> = VORAZ( G);
```

<xm,vm> := TSP_PBC(x, k, ltour, lhnos, xini, vini).

7. BACKTRACKING ITERATIVO

```
función BACK_IT ( D es datos del problema )
dev ( MSC es solución )

    MSC := Inicializar_mejor_solución_en_curso;
    /* se puede usar un algoritmo externo como ya
    hemos hecho en la versión recursiva */
    var e, e’ es elem; /* elem contiene toda la
    información necesaria: solución en curso, nivel,
    marcajes, valor de la función de estimación, ... */
    var p es pila(elem);
    e:= nodo_inicial();    p:= apilar(p_vacia,e);
    *[ ¬vacia(p) --->
        e:= cima(p);   desapilar(p);
        k:= nivel(e)+1;
/* los hijos de e están 1 nivel por debajo de e */
        preparar_recorrido_nivel_k;
        *[existan_hermanos_nivel_k --->
            e’:=sig_hno_nivel_k(e);
            [ factible(e’)  ∧ prometedor(e’)    --->
                [ solución(e’)     ---> Actualizar( MSC);
                [] ¬solución(e’) --->  p:= apilar(p,e’);
                ]
            []¬(factible(e’)∧prometedor(e’)) ---> seguir;
            ]
        ] /* fin bucle recorrido hermanos */
    ] /* fin bucle por pila vacia*/
dev ( MSC );
```

ffunción

La PBCMSC se oculta tras la función ***prometedor*** en el sentido de que sólo avanzamos si podemos mejorar la mejor solución en curso.

La versión iterativa del backtracking utiliza una **pila** para mantener los nodos "factibles, prometedores y no solución" que están pendientes de expansión.
Para obtener un recorrido en anchura (o por niveles) del EB basta con sustituir la pila por una **cola**.
El algoritmo de *Branch&Bound* utiliza una **cola de prioridad** en lugar de la pila. La prioridad de los nodos viene dada por el valor de la función de estimación sobre ellos.