Programación y Estructuras de Datos Avanzadas

Capítulo 7: Ramificación y Poda

Capítulo 7: Ramificación y Poda

7.1 Planteamiento y esquema general

- La **ramificación y poda** (branch and bound) es un esquema para explorar un grafo dirigido implícito que se utiliza en **problemas de optimización** (cuando no se pueden utilizar algoritmos voraces).
- **Comparado con el algoritmo de vuelta atrás (o retroceso):**

  Puede ser vista como una mejora del algoritmo de vuelta atrás.

  – **Similitudes:**
    - Realiza un recorrido sistemático en el árbol/grafo (implícito, se va construyendo durante el recorrido) de soluciones.

  – **Diferencias:**
    - **Se utiliza en problemas de optimización (maximizar o minimizar algo).**
    - **Estrategia de ramificación**: los nodos no se exploran en profundidad (o anchura, aunque ya no sería un vuelta atrás estricto), sino escogiendo el más prometedor de entre los activos $\rightarrow$ cola de prioridad (montículo). Para cada nodo se calcula una **EstimacionOpt, una estimación optimista del mejor valor que se puede obtener a partir de él**. Se utiliza en la cola de prioridad.
    - **Estrategia de poda:** si un nodo tiene una EstimacionOpt peor que la de una solución factible ya encontrada (cota) $\rightarrow$ se poda esa rama (**poda por cota**). Además, se **poda por factibilidad** ya que sólo se incluyen los sucesores de un nodo que sean factibles (extensiones válidas).

- **Esquema general (Problema de minimización):**

```
fun RamificacionYPoda (nodoRaiz, mejorSolución: TNodo, cota: real)
    monticulo ← CrearMonticuloVacio()
    cota ← EstimacionPes(nodoRaiz)
    Insertar(nodoRaiz, monticulo)
    mientras ¬ MonticuloVacio?(monticulo) ∧
              EstimacionOpt(Primero(monticulo)) ≤ cota hacer
        nodo ← ObtenerCima(monticulo)
        para cada hijo extensión válida de nodo hacer
            si solución(hijo) entonces
                si coste(hijo) ≤ cota entonces
                    cota ← coste(hijo)
                    mejorSolucion ← hijo
                fsi
            sino
                si EstimacionOpt(hijo) ≤ cota entonces
                    Insertar(hijo, monticulo)
                    si EstimacionPes(hijo) < cota entonces
                        cota ← EstimacionPes(hijo)
                    fsi
                fsi
            fsi
        fpara
    fmientras
ffun
```

← parámetros de salida

← Montículo de **mínimos** (contiene nodos "ordenados" por EstimacionOpt)

← Se inicializa la cota al coste de una solución conocida o a la EstimacionPes del nodo raíz (sino a +∞).

Sólo sigue si el nodo más prometedor puede mejorar lo que hay (≤ por si todavía no se ha almacenado la solución) Poda por cota

← sólo sucesores que pueden llevar a la solución Poda por factibilidad.

comprueba si el nodo hijo es solución

Si el coste de esta solución mejora la que ya tenemos (o al menos la iguala) se actualiza tanto mejorSolucion como la cota

← El hijo sólo se incluye en el montículo si su EstimacionOpt es mejor que la cota (no incluirlo implica **podar toda la rama**). Poda por cota

Si EstimacionPes mejora la cota la actualizo

- **Funciones:**
  - **extensión válida de nodo/compleciones:** contiene todos los nodos sucesores de nodo factibles de ser solución (que pueden llevar a solución).
  - **Coste(nodo_solucion):** devuelve el coste de un nodo solución
  - **EstimacionOpt(nodo):** contiene un valor optimista para ese nodo de forma que ninguna de las soluciones obtenidas a partir de él tendrá un coste mejor que dicho valor (aunque sí lo podrá igualar).
    - Ejemplo: EstimacionOpt(nodo)=20 en problema de minimización indica que a partir de la rama de ese nodo, todos los nodos solución tendrán un coste≥20.
  - **EstimacionPes(nodo):** contiene un valor pesimista para ese nodo de forma que a partir de ese nodo tengo garantizada al menos una solución con un coste como mínimo tan bueno como dicho valor.
    - Ejemplo: EstimacionPes(nodo)=30 en problema de minimización indica que a partir de la rama de ese nodo, hay al menos un nodo solución con un coste≤30.
  - **Variable cota:** contiene el valor de la **mejor solución (problema maximización → mayor coste; problema minimización → menor coste)** obtenida hasta el momento o, en su defecto, la mejor EstimacionPes.
    - Se actualiza cuando se encuentra una solución con un coste mejor que la cota o cuando se encuentra un nodo cuya EstimaciónPes sea mejor que la cota actual
    - Para que un nodo se incluya en el montículo y/o se explore es necesario que estimacionOpt≤ cota.

• Ejemplo: Problema de la mochila entera (Mochila 0/1)

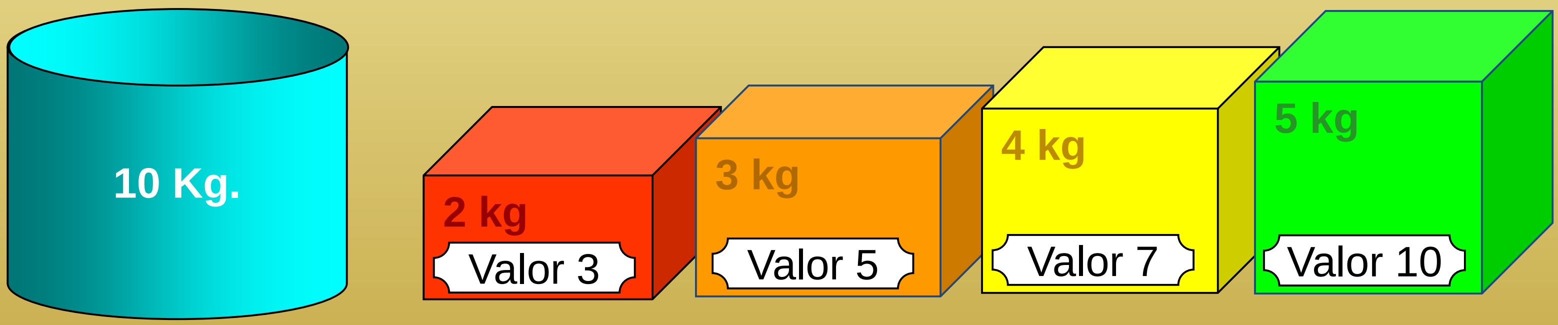
➢ Datos del problema:

- **n**: número de objetos disponibles.
- **P**: peso máximo admitido por la mochila.
- **p = (p_1, p_2, ..., p_n)** pesos de los objetos.
- **v = (v_1, v_2, ..., v_n)** valores/beneficios de los objetos.

➢ Formulación matemática:

$$\text{Maximizar} \sum_{i=1..n} x_i v_i \text{, sujeto a la restricción} \sum_{i=1..n} x_i p_i \leq P \;\; \text{y} \;\; x_i \in \{0,1\}$$

➢ Ejemplo: n = 4; P = 10; v = (3, 5, 7, 10); p = (2, 3, 4, 5)



- **El algoritmo voraz** resuelve el problema de la mochila con objetos fraccionables ($x_i \in [0,1]$), *introduciendo los elementos de mayor a menor v_i/p_i y una fracción del primero que no entre completo*.
- ¿Valdría para este problema?. No. Basta un contraejemplo:
  El esquema voraz metería primero el objeto nº 4 (v_4/p_4=10/5=2) y el nº 3 (v_3/p_3=7/4=1.75), valor total=10+7=17, mientras que si se meten el 1, el 2 y el 4, valor total=18 → Alg. RyP.

a) Estructuras de datos

- **Representación de solución** $\rightarrow$ vector de booleanos: **s= (x_1, x_2, ..., x_n)**, con $x_i \in \{0,1\}$
  - $x_i = 0 \rightarrow$ No se mete en la mochila el objeto **i**
  - $x_i = 1 \rightarrow$ Sí se mete en la mochila el objeto **i**

- **Espacio de búsqueda**: árbol binario de profundidad n $\rightarrow$ representación de los nodos:

tipo TNodo = **registro**

moch: matriz[0..n] de booleano (va almacenando la solución)

k: entero (representa el nivel del árbol=nº de objeto comprobado)

pesoT: real (peso de los objetos metidos hasta el momento)

valorT: real (valor/beneficio de los objetos metidos hasta el momento)

estOpt: real (estimación optimista del nodo)

fregistro

- **Problema de maximización** $\rightarrow$ utilizaré un montículo de máximos que contendrá los nodos "ordenados" por estOpt

b) Algunas cuestiones

- **¿Cómo es el nodo raíz?**

  raiz.noch=(0,0,..,0); raiz.k=0; raiz.pesoT=0; raiz.valorT=0; raiz.estOpt=¿estimacionOpt(raiz)?

- **¿Cómo generar los hijos (extensiones válidas) de un nodo padre?**

  para i ← 1 hasta 0 **hacer**

  si padre.pesoT+i*p[hijo.k]>P continue; fsi

  hijo.k ← padre.k+1

  hijo.moch ← padre.moch

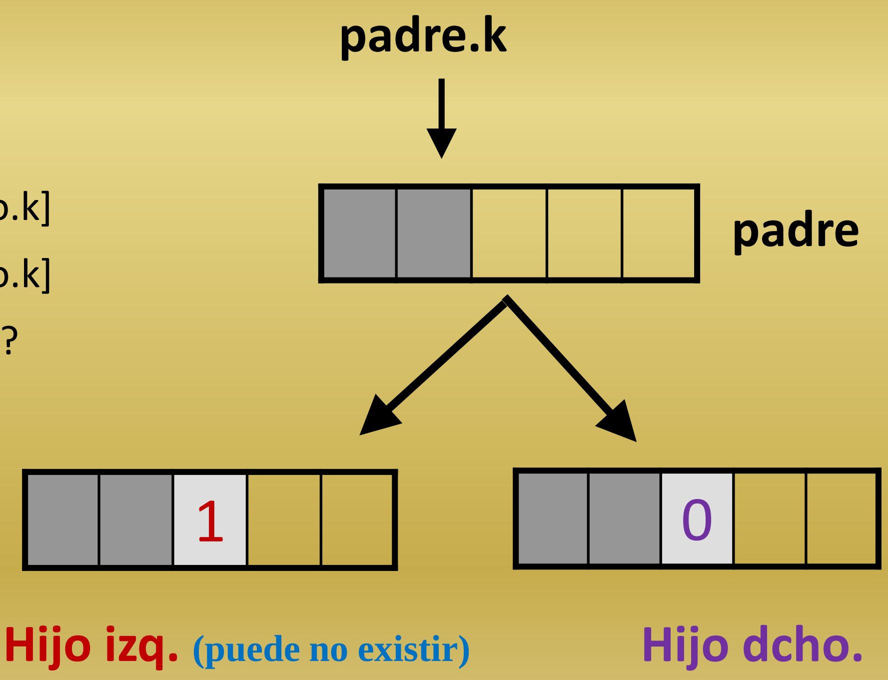
  hijo.moch[hijo.k] ← i

  hijo.pesoT ← padre.pesoT + i*p[hijo.k]

  hijo.valorT ← padre.valorT + i*v[hijo.k]

  hijo.estOpt ← ¿estimacionOpt(hijo)?

  fpara



- **¿Cómo es la función Solución(nodo: TNodo): booleano?**

  dev nodo.k=n

c) Cálculo de cotas

- **Estimación Optimista**
  - Al ser un problema de maximización será una cota superior al valor alcanzable por cualquier solución descendiente de ese nodo. A partir de ese nodo no podré obtener una solución con un coste mejor que estimacionOpt.
  - ***estimacionOpt(nodo)=valor de los objetos ya incluidos en la mochila (nodo.valorT) + estimación del valor de una asignación óptima de los restantes***
  - Para estimación del valor óptimo restante, se resuelve el problema como si los objetos fuesen divisibles ($x_i \in [0,1]$) $\rightarrow$ *Problema mochila con objetos fraccionables*

```
fun EstimacionOpt (pesos, valores: TVectorR, P: real, k: entero,
                   pesoT: real, valorT: real): real
   var
      capacidad, estimacion: real
      i: entero
   fvar
   capacidad ← P - pesoT
   estimacion ← valorT
   i ← k + 1
   mientras i ≤ n ∧ capacidad ≥ 0 hacer
      si pesos[i] ≤ capacidad entonces
         estimacion ← estimacion + valor[i]
         capacidad ← capacidad - pesos[i]
      sino
         estimacion ← estimacion + (capacidad / pesos[i])*valor[i]
         capacidad ← 0
      fsi
      i ← i + 1
   fmientras
   dev estimacion
ffun
```

Algoritmo voraz (para asignar los restantes)

I. Ordena los objetos de mayor a menor valor específico $e_i=v_i/p_i$ (valor/kg).
II. Los va incluyendo en la mochila por ese orden. Cuando uno no quepa completo, introduce la fracción adecuada de él para completar el peso de la mochila.

- ❖ Imposible encontrar una forma de asignar los objetos restantes con un valor mejor que el obtenido mediante este método.
- ❖ En este caso el peso final de la mochila es exactamente P (se aprovecha el espacio al completo)

- **Estimación Pesimista**
  - Al ser un problema de maximización será una cota inferior al valor alcanzable por cualquier solución descendiente de ese nodo. A partir de ese nodo tengo garantizada una solución con un coste al menos tan bueno (≥) como EstimacionPes.
  - ***estimacionPes(nodo)=valor de los objetos ya incluidos en la mochila (nodo.valorT) + estimación del coste de una solución cualquiera alcanzable desde ahí***
  - Para estimación del valor restante, asigno los objetos ordenados de mayor a menor valor específico $e_i=v_i/p_i$ (valor/kg), aunque sólo se introducen en la mochila si caben enteros.

```
fun EstimacionPes (pesos, valores: TVectorR, P: real, k: entero,
                   pesoT: real, valorT: real): real
   var
      capacidad, cota: real
      i: entero
   fvar
   capacidad ← P - pesoT
   cota ← valorT
   i ← k + 1
   mientras i ≤ n ∧ capacidad ≥ 0 hacer
      si pesos[i] ≤ capacidad entonces
         cota ← cota + valor[i]
         capacidad ← capacidad - pesos[i]
      fsi
      i ← i + 1
   fmientras
   dev cota
ffun
```

d) Algoritmo principal

```
tipo TVectorB = matriz[0..n] de booleano
tipo TVectorR = matriz[0..n] de real
tipo TNodo = registro
    moch: TVectorB
    k: entero
    pesoT: real
    valorT: real
    estOpt: real
fregistro
fun Mochila (pesos, valores: TVectorR, P: real, moch: TVertorB, valor: real)
    var
        monticulo: TMonticulo
        nodo,hijo: TNodo
        cota,estPes: real
    fvar
    monticulo ← CrearMonticuloVacio()
    valor ← 0
    {Construimos el primer nodo}
    nodo.moch ← moch
    nodo.k ← 0
    nodo.pesoT ← 0
    nodo.valorT ← 0
    nodo.estOpt ← EstimacionOpt(pesos,valores,P,nodo.k,nodo.pesoT,nodo.valorT)
    Insertar(nodo, monticulo)
    cota ← EstimacionPes(pesos, valores, P, nodo.k, nodo.PesoT, nodo.valorT)
    mientras ¬ MonticuloVacio?(monticulo) ∧
             EstimacionOpt(Primero(monticulo)) ≥ cota hacer      ← Poda por cota
        nodo ← ObtenerCima(monticulo)
        {se generan las extensiones válidas de nodo}
        {se mete el objeto en la mochila}
        hijo.k ← nodo.k + 1
        hijo.moch ← nodo.moch
        si nodo.pesoT + pesos[hijo.k] ≤ P entonces     ← Poda por factibilidad: si supero peso
            hijo.moch[hijo.k] ← cierto
            hijo.pesoT ← nodo.pesoT + pesos[hijo.k]
            hijo.valorT ← nodo.valorT + valores[hijo.k]
            hijo.estOpt ← nodo.estOpt
            si hijo.k = n entonces
                si valor < hijo.valorT entonces
                    moch ← hijo.moch
                    valor ← hijo.valorT
                    cota ← valor
                fsi
            sino {la solución no está completa}
                Insertar(hijo, monticulo)
            fsi
        fsi
        {no se mete el objeto en la mochila}
        hijo.estOpt ← EstimacionOpt(pesos, valores, P, hijo.k,
                          nodo.PesoT, nodo.valorT)
        si hijo.estOpt ≥ cota entonces     ← Poda por cota
            hijo.moch[hijo.k] ← falso
            hijo.pesoT ← nodo.pesoT
            hijo.valorT ← nodo.valorT
            si hijo.k = n entonces
                si valor < hijo.valorT entonces
                    moch ← hijo.moch
                    valor ← hijo.valorT
                    cota ← valor     ← Solución mejor: actualizo cota
                fsi
            sino {la solución no está completa}
                Insertar(hijo, monticulo)
                estPes ← EstimacionPes(pesos, valores, P, hijo.k,
                              hijo.PesoT, hijo.valorT)
                si cota < estPes entonces
                    cota ← estPes     ← Solución mejor (a través de estPes): actualizo cota
                fsi
            fsi
        fsi
    fmientras
ffun
```

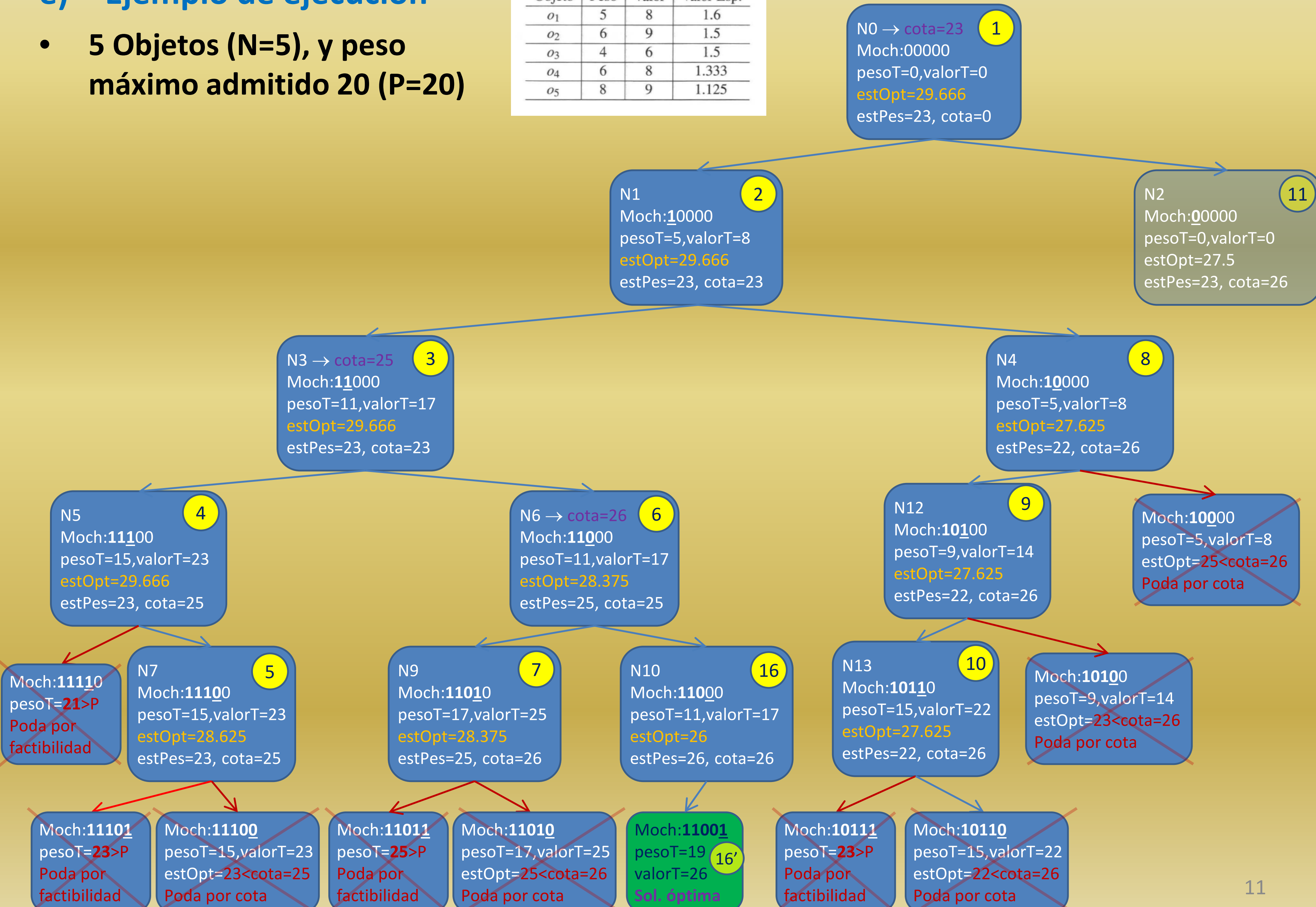
Hijo izquierdo

Hijo derecho (no meto objeto k)

e) Ejemplo de ejecución

- **5 Objetos (N=5), y peso máximo admitido 20 (P=20)**

Objeto	Peso	Valor	Valor Esp.
o_1	5	8	1.6
o_2	6	9	1.5
o_3	4	6	1.5
o_4	6	8	1.333
o_5	8	9	1.125



Objeto	Peso	Valor	Valor Esp.
o_1	5	8	1.6
o_2	6	9	1.5
o_3	4	6	1.5
o_4	6	8	1.333
o_5	8	9	1.125

Coste (caso peor):

- Difícil de contabilizar cotas.
- La cota superior del coste es el nº máximo de nodos del grafo: n objetos=n niveles y cada nodo ramifica en 2 nodos $\rightarrow 2^n$ nodos (realmente son el nº de hojas)$\rightarrow O(2^n)$

Realmente son muchos menos nodos, pero, para cada nodo, estOpt=O(n) y las operaciones del montículo=$O(\log 2^n)=O(n)$, por lo que el coste real sería $\theta(n\times$nº nodos$)$, aunque como nº nodos real no se conocen, se suele especificar como cota superior el nº de nodos máximo del árbol (aunque realmente el nº de hojas=nº de soluciones).

7.2 Asignación de tareas: pastelería

- Una pastelería tiene empleados a *n* pasteleros que tienen distinta destreza para hacer *m* tipos de pasteles.
  - **Tabla de costes C[1..*n*,1..*m*]** $\rightarrow$ c_{ij} el coste de que el pastelero i realice el pastel j.
  - Deseamos realizar *n* pasteles, de los tipos descritos en **pedidos[1..*n*]**
- **Objetivo:** asignar pasteleros a cada uno de los pasteles del pedido de forma que se **minimice el coste total** (a un mismo pastelero no se le puede asignar más de 1 pastel).
- **Ejemplo:** 5 pasteleros (*n*=5) y 3 tipos de pasteles (*m*=3), con la siguiente tabla de costes y pedidos:

	Pastel		
Pastelero	1	2	3
1	2	5	3
2	5	3	2
3	6	4	9
4	6	3	8
5	7	5	8

  - ¿Valdría un **algoritmo voraz** que tomase para cada pastel del pedido el pastelero que prepara con menos coste dicho pastel (si está libre y sino el siguiente mejor)?
  - **Ejemplo:** Pedidos[11321] (3 pasteles de tipo 1, 1 del tipo 2 y otro del tipo 3):
    - Solución algoritmo voraz [12435] $\rightarrow$ coste=26.
    - Solución óptima [13245] $\rightarrow$ coste=20.

¡Al no encontrar una función de selección apropiada hay que usar un algoritmo de ramificación y poda!

a) Estructuras de datos (*voy a considerar que n=m*)

- **Representación de solución** $\rightarrow$ vector de enteros: pasteleros**= (x_1, x_2, ..., x_n)**, con $x_i=j$ indica que el pedido *i*-ésimo se ha asignado al pastelero *j*
- **Espacio de búsqueda**: árbol profundidad *n* (cada nodo se expande en n-k nodos, donde k es el nivel del árbol) $\rightarrow$ representación de los nodos:

tipo TNodo = **registro**

pasteleros: matriz[0..n] de entero (va almacenando la solución)

asignados: matriz[0..n] de booleano (indica los pasteleros ya ocupados)

k: entero (representa el último pedido asignado=nivel del árbol)

costeT: real (coste total de las asignaciones realizadas hasta el momento)

estOpt: real (estimación optimista del nodo)

fregistro

- **Problema de minimización** $\rightarrow$ utilizaré un montículo de mínimos que contendrá los nodos "ordenados" por estOpt

b) Cálculo de cotas

- **Estimación Optimista**
  - Al ser un problema de minimización será una cota inferior al valor alcanzable por cualquier solución descendiente de ese nodo. A partir de ese nodo no podré obtener una solución con un coste menor que estimacionOpt.
  - ***estimacionOpt(nodo)=coste de las tareas ya asignadas (nodo.costeT) + el coste mínimo de las pendientes de asignar***
  - **Ejemplo:** Pedidos[11321] (3 pasteles de tipo 1, 1 del tipo 2 y otro del tipo 3):

EstimacionOpt(raiz)=C_{min}(pastel1)+C_{min}(pastel1)+C_{min}(pastel3)+C_{min}(pastel2)+C_{min}(pastel1)=2+2+2+3+2=11;

	Pastel		
Pastelero	1	2	3
1	2	5	3
2	5	3	2
3	6	4	9
4	6	3	8
5	7	5	8

```
fun EstimacionOpt (costes: TTabla, pedido: TVector, k: entero, costeT: real): real
    var
        estimacion, menorC: real
        i,j: entero
    fvar
    estimacion ← costeT
    para i ← k+1 hasta n hacer
        menorC ← costes[1,pedido[i]]
        para j ← 2 hasta n hacer
            si menorC > costes[j,pedido[i]] entonces
                menorC ← costes[j,pedido[i]]
            fsi
        fpara
        estimacion ← estimacion + menorC
    fpara
    dev estimacion
ffun
```

- **Estimación Pesimista**
  - Al ser un problema de minimización será una cota superior al valor alcanzable por cualquier solución descendiente de ese nodo. A partir de ese nodo tengo garantizada una solución con un coste al menos tan bueno (≤) como estimacionPes.
  - ***estimacionPes(nodo)=coste de las tareas ya asignadas (nodo.costeT) + el coste máximo de las pendientes de asignar***
  - **Ejemplo:** Pedidos[11321] (3 pasteles de tipo 1, 1 del tipo 2 y otro del tipo 3):

EstimacionPes(raiz)=C_{max}(pastel1)+C_{max}(pastel1)+C_{max}(pastel3)+C_{max} (pastel2)+C_{max}(pastel1)=7+7+9+5+7=35;

	Pastel		
Pastelero	1	2	3
1	2	5	3
2	5	3	2
3	6	4	$\boxed{9}$
4	6	3	8
5	$\boxed{7}$	$\boxed{5}$	8

```
fun EstimacionPes (costes: TTabla, pedido: TVector, k: entero, costeT: real): real
    var
        estimacion, mayorC: real
        i,j: entero
    fvar
    estimacion ← costeT
    para i ← k+1 hasta n hacer
        mayorC ← costes[1,pedido[i]]
        para j ← 2 hasta n hacer
            si mayorC < costes[j,pedido[i]] entonces
                mayorC ← costes[j,pedido[i]]
            fsi
        fpara
        estimacion ← estimacion + mayorC
    fpara
    dev estimacion
ffun
```

c) Algoritmo principal

```
tipo TVectorB = matriz[0..n] de booleano
tipo TVector = matriz[0..n] de entero
tipo TNodo = registro
    pasteleros: TVector
    asignados: TVectorB
    k: entero
    costeT: real
    estOpt: real
fregistro
tipo TTabla = matriz[1..n,1..n] de entero
fun AsignaPasteleros (costes: TTabla, pedido: TVector,
                      pasteleros: TVector, costeT: entero)
  var
     monticulo: TMonticulo
     nodo,hijo: TNodo
     cota,estPes: real
  fvar
  monticulo ← CrearMonticuloVacio()
  costeT ← 0
  {Construimos el primer nodo}
  nodo.pasteleros ← pasteleros
  para i ← 0 hasta n hacer
     nodo.asignados[i] ← falso
  fpara
  nodo.k ← 0
  nodo.costeT ← 0
  nodo.estOpt ← EstimacionOpt(costes,pedido,nodo.k,nodo.costeT)
  Insertar(nodo, monticulo)
  cota ← EstimacionPes(costes, pedido, nodo.k, nodo.costeT)
  mientras ¬ MonticuloVacio?(monticulo) ∧
        EstimacionOpt(Primero(monticulo)) ≤ cota hacer        ← Poda por cota
     nodo ← ObtenerCima(monticulo)
     {se generan las extensiones válidas del nodo}
     {para cada pastelero no asignado se crea un nodo}
     hijo.k ← nodo.k + 1
     hijo.pasteleros ← nodo.pasteleros
     hijo.asignados ← nodo.asignados
     para i ← 1 hasta n hacer
        si ¬ hijo.asignados[i] entonces      ← Poda por factibilidad: si no está libre el pastelero
           hijo.pasteleros[hijo.k] ← i
           hijo.asignados[i] ← cierto        ← Se asigna el pastelero i
           hijo.costeT ← nodo.costeT + coste[i,pedido[hijo.k]]
           si hijo.k = n entonces            ← Nodo solución
              si costeT > hijo.costeT entonces
                 pasteleros ← hijo.pasteleros
                 costeT ← costeT.valorT
                 cota ← costeT              ← Solución mejor: actualizo cota
              fsi
           sino {la solución no está completa}
              hijo.estOpt ← EstimacionOpt(costes, pedido, hijo.k, hijo.costeT)
              Insertar(hijo, monticulo)
              estPes ← EstimacionPes(costes,pedido, hijo.k, hijo.costeT)
              si cota > estPes entonces
                 cota ← estPes               ← Solución mejor (a través de estPes): actualizo cota
              fsi
           fsi
        hijo.asignados[i] ← falso {se desmarca}   ← Se vuelve a dejar la asignación como estaba en nodo (padre) para calcular el siguiente hermano
        fsi
     fpara
  fmientras
ffun
```

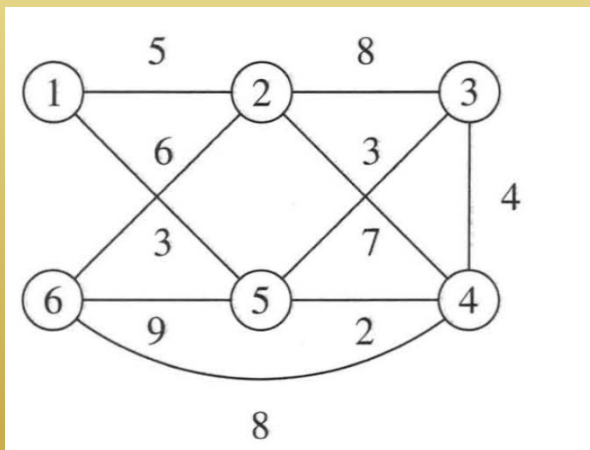
Hijos/Compleciones

Coste (caso peor):

– Difícil de contabilizar cotas. Una cota superior del coste es el nº máximo de nodos del grafo: n pasteles=n niveles y cada nodo se expande en las n-k asignaciones de pedidos pendiente → n! nodos → O(n!)

7.3 El viajante de comercio (ciclo Hamiltoniano mínimo)

- Tenemos un grafo no dirigido valorado de n nodos.
- **Objetivo:** encontrar un ciclo Hamiltoniano (camino que pasa una sola vez por cada nodo y terminan en el nodo inicial) de coste mínimo
  - → Problema del viajante de comercio que, partiendo de una ciudad origen, tiene que visitar todas las ciudades de su zona una y sólo una vez y volver a la ciudad origen, minimizando el coste de recorrido.
  - → No está garantizado que haya solución.
- **Ejemplo:** tenemos el siguiente grafo de 5 nodos/ciudades



	1	2	3	4	5	6
1	∞	5	∞	∞	3	∞
2	5	∞	8	7	∞	6
3	∞	8	∞	4	3	∞
4	∞	3	4	∞	2	8
5	3	∞	3	2	∞	9
6	∞	6	∞	8	9	∞

Matriz de adyacencia

¡El problema de búsqueda de ciclos Hamiltonianos se puede resolver mediante vuelta atrás, pero no éste!

a) Estructuras de datos

- **Representación de solución** $\rightarrow$ vector de enteros: ruta**= (x_1, x_2, ..., x_n)**, indica el orden de recorrido de los nodos.
  - P.ej. {1,5,3,4,6,2} indica el ciclo Hamiltoniano $1\rightarrow5\rightarrow3\rightarrow4\rightarrow6\rightarrow2\rightarrow1$
- **Espacio de búsqueda**: árbol profundidad *n* (cada nodo se expande en n-k nodos, donde k es el nivel del árbol) $\rightarrow$ representación de los nodos:

tipo TNodo = **registro**

ruta: matriz[1..n] de entero (va almacenando la solución)

asignados: matriz[1..n] de booleano (indica los nodos ya visitados)

k: entero (representa el último pedido asignado=nivel del árbol)

costeT: real (coste total del camino recorrido hasta el momento)

estOpt: real (estimación optimista del nodo)

fregistro

- **Problema de minimización** $\rightarrow$ utilizaré un montículo de mínimos que contendrá los nodos "ordenados" por estOpt

b) Cálculo de cotas

- **Estimación Optimista**
  - Al ser un problema de minimización será una cota inferior al valor alcanzable por cualquier solución descendiente de ese nodo. A partir de ese nodo no podré obtener una solución con un coste menor que estimacionOpt.
  - ***estimacionOpt(nodo)=coste del camino ya recorrido (nodo.costeT) + el coste mínimo del camino que falta (se calcula minArista que es el coste mínimo de una arista y se multiplica por los nodos que faltan)***
  - **Ejemplo:**

  EstimacionOpt(raiz)=raiz.costeT+minArista×(n-raiz.k+1)=0+2×(6-1+1)=12

	1	2	3	4	5	6
1	∞	5	∞	∞	3	∞
2	5	∞	8	7	∞	6
3	∞	8	∞	4	3	∞
4	∞	3	4	∞	$\boxed{2}$	8
5	3	∞	3	2	∞	9
6	∞	6	∞	8	9	∞

```
fun EstimacionOpt (grafo: Tgrafo, minArista: entero, k: entero, costeT: real): real
    estimacion, menorC: real
    estimacion ← costeT + (n-k+1) * minArista
    dev estimacion
ffun
```

- **Estimación Pesimista**
  - Puesto que no se puede encontrar rápidamente una solución (y, además, ni siquiera está garantizada su existencia) *no se puede establecer una estimación pesimista*.
  - Por tanto la cota inicial=∞ *(la cota=coste de la mejor solución encontrada hasta el momento sólo se actualizará cuando se encuentre una solución con mejor cota, no por estimacionPes)*

c) Algoritmo principal

```
tipo TVectorB = matriz[0..n] de booleano
tipo TVector = matriz[0..n] de entero
tipo TNodo = registro
    ruta: TVector
    asignados: TVectorB
    k: entero
    costeT: real
    estOpt: real
fregistro
tipo TGrafo = matriz[1..n,1..n] de entero
tipo TVector = matriz[1..n] de entero
fun Viajante (grafo: TGrafo, ruta: TVector, costeT: entero)
    var
        monticulo: TMonticulo
        nodo,hijo: TNodo
        cota, estPes, verticeAnt, minArista: entero
    fvar
    monticulo ← CrearMonticuloVacio()
    minArista ← menorArista(grafo)
    costeT ← ∞
    {Construimos el primer nodo}
    nodo.ruta ← ruta
    nodo.asignados[1] ← cierto
    para i ← 2 hasta n hacer
        nodo.asignados[i] ← falso
    fpara
    nodo.k ← 1
    nodo.costeT ← 0
    nodo.estOpt ← EstimacionOpt(grafo,minArista,nodo.k,nodo.costeT)
    Insertar(nodo, monticulo)
    cota ← ∞
```

No se puede establecer una solución/cota inicial

```
mientras ¬ MonticuloVacio?(monticulo) ∧
        EstimacionOpt(Primero(monticulo)) < cota hacer
```

Poda por cota

```
    nodo ← ObtenerCima(monticulo)
    {se generan las extensiones válidas de nodo}
    verticeAnt ← nodo.ruta[nodo.k]
    hijo.k ← nodo.k + 1
    hijo.ruta ← nodo.ruta
    hijo.asignados ← nodo.asignados
    para i ← 2 hasta n hacer
        si ¬ hijo.asignado[i] ∧ grafo[verticeAnt,i] ≠ ∞ entonces
```

Poda por factibilidad: si ya se visitado el nodo o no existe la arista

```
            hijo.ruta[hijo.k] ← i
```

Se asigna el nodo i a la ruta

Hijos/Compleciones

```
            hijo.asignados[i] ← cierto
            hijo.costeT ← nodo.costeT + grafo[verticeAnt,i]
            si hijo.k = n entonces
                si grafo[i,1] ≠ ∞ entonces
                    hijo.costeT ← hijo.costeT + grafo[i,1]
```

Poda por factibilidad: si el último nodo no conecta con el primero

```
                    si costeT > hijo.costeT entonces
                        ruta ← hijo.ruta
                        costeT ← hijo.costeT
                        cota ← costeT
```

Solución mejor: actualizo cota

```
                    fsi
                fsi
            sino {la solución no está completa}
                hijo.estOpt ← EstimacionOpt(grafo,minArista,hijo.k,hijo.costeT)
                si hijo.estOpt < costeT entonces
                    Insertar(hijo, monticulo)
                fsi
            fsi
            hijo.asignados[i] ← falso {se desmarca}
```

Se vuelve a dejar la asignación como estaba en nodo (padre) para calcular el siguiente hermano

```
        fsi
    fpara
fmientras
ffun
```

Coste (caso peor):

– Difícil de contabilizar cotas. Una cota superior del coste es el nº máximo de nodos del grafo: n nodos=n niveles y cada nodo se expande en las n-k nodos restantes → n! nodos → O(n!)

Otros problemas resueltos en el texto base

7.4 Selección de tareas: cursos de formación

7.5 Distancia de edición