Metodología y Tecnología de la Programación

Ingeniería en Informática

Curso 2008-2009

Esquemas algorítmicos. Ramificación y Poda

Yolanda García Ruiz D228 **ygarciar@fdi.ucm.es**
Jesús Correas D228 **jcorreas@fdi.ucm.es**

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

(elaborado a partir de [GV00], [PS03] y notas de S. Estévez y R. González del Campo)

Bibliografía

- **Importante**: Estas transparencias son un material de apoyo a las clases presenciales y no sustituyen a la bibliografía básica ni a las propias clases presenciales para el estudio de la asignatura
- Bibliografía básica:
  - [GV00]: capítulo 7
  - [PS03][1]: capítulo 8
- Bibliografía complementaria:
  - [GC01][2]: capítulo 7
- Ejercicios resueltos:
  - [MOV04]

(1) [PS03] J. I. Peláez Sánchez et al. *Análisis y Diseño de algoritmos: un enfoque teórico y práctico*, Universidad de Málaga, 2003

(2) [GC01] D. Giménez Cánovas *Apuntes y problemas de algorítmica*, Universidad de Murcia, 2001. Disponible en `http://servinf.dif.um.es/~domingo/apuntes/Algoritmica/apuntes.pdf`

Esquemas algorítmicos. Ramificación y Poda

Características generales

- **Branch and Bound** es una variante del esquema de *backtracking*
- Al igual que *backtracking*, se basa en el recorrido del árbol de expansión en busca de soluciones
- Se aplica sobre todo a **problemas de optimización**: búsqueda de la mejor solución a un problema, aunque también para buscar una o todas las soluciones a un problema
- La principal diferencia con *backtracking* es que la generación de nodos del árbol de expansión se puede realizar aplicando distintas estrategias $\rightarrow$ **estrategias de ramificación**
- Además se utilizan **cotas** que permiten podar ramas que no conducen a una solución óptima (se evita ramificar nodos) $\rightarrow$ **estrategias de poda**

Características generales

- En cuanto a la **estrategia de ramificación**
  - En el esquema de *backtracking*, el recorrido del árbol de expansión siempre es en profundidad
  - En ramificación y poda, la generación de los nodos del árbol de expansión puede seguir varias extrategias:
    - ★ en profundidad (LIFO)
    - ★ en anchura (FIFO)
    - ★ aquella que selecciona el nodo más prometedor

    El objetivo es utilizar la estrategia que permita encontrar la solución más rápidamente
- Se realiza una **estrategia de poda** en la que en cada nodo se calcula una **cota** del posible valor de aquellas soluciones que pudieren encontrarse más adelante en el árbol. Esto permite no explorar aquellas ramas que no conducen a una solución válida u óptima (en el caso de problemas de optimización)

Esquemas algorítmicos. Ramificación y Poda

1. Características generales
2. Estrategias de Ramificación
3. Estrategias de Poda
4. Esquema algorítmico de Ramificación y Poda
5. Estudio de complejidad
6. El problema de la asignación de tareas
7. Problema de la mochila [0,1]
8. Programación Lineal
9. El viajante de comercio
10. El laberinto

Estrategias de Ramificación

- Distintos tipos de recorrido del árbol de expansión: profundidad, anchura, etc.
- Para determinar qué nodo va a ser expandido, dependiendo de la estrategia de ramificación, necesitamos una estructura capaz de almacenar aquellos nodos pendientes de ser expandidos
- Para hacer el recorrido se utiliza una lista de nodos a la que llamaremos **Lista de Nodos Vivos** (LNV)
- Un **nodo vivo** del árbol es aquel que tiene posibilidades de ser ramificado, es decir, aquel que ha sido creado y no ha sido explorado ni podado todavía

La lista de nodos vivos contiene nodos pendientes de tratar por el algoritmo

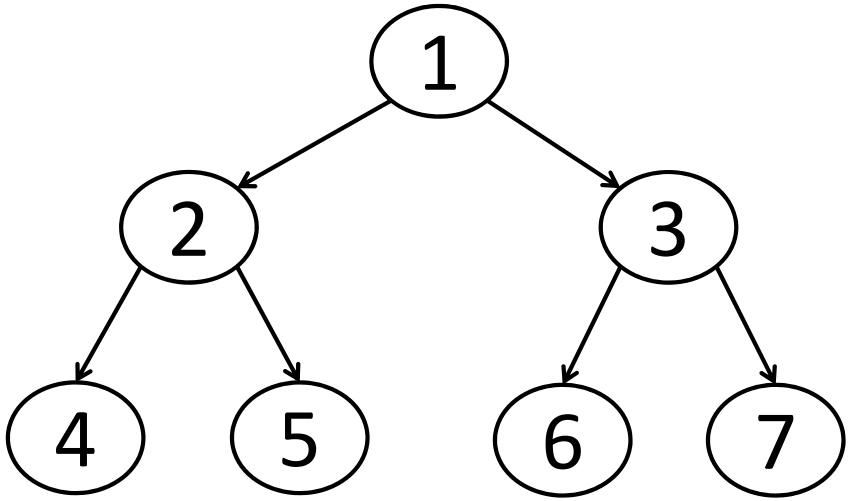
Estrategias de Ramificación. Método general

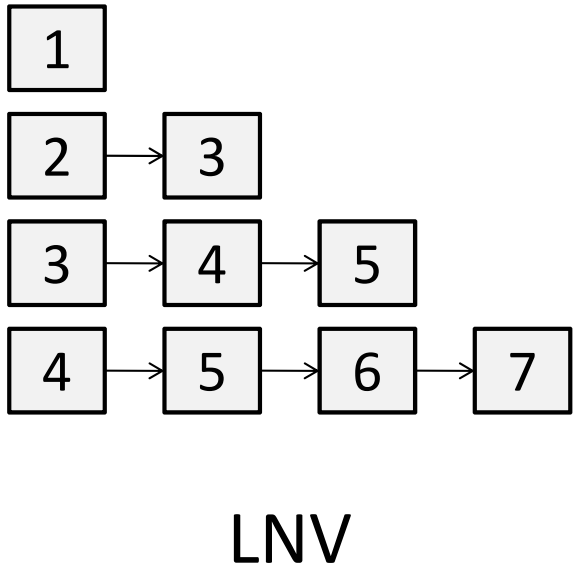
- La idea básica es la siguiente:
  1. Sacar un elemento N de la lista de nodos vivos (LNV)
  2. Generar todos los descendientes de N
  3. Si no se podan y no son solución, se introducen en la LNV

- El recorrido del árbol depende de cómo se maneje la lista
  - Recorrido en profundidad $\Rightarrow$ La lista se trata como una **pila**
  - Recorrido en anchura $\Rightarrow$ La lista se trata como una **cola**
  - La estrategia de mínimo coste $\Rightarrow$ Se utiliza una función denominada **mínimo coste** para determinar qué elemento de la lista de nodos vivos va a ser explorado en cada momento. Se utiliza una **cola con prioridades** o **montículo** para almacenar nodos ordenados por su coste

Estrategias de Ramificación. En anchura

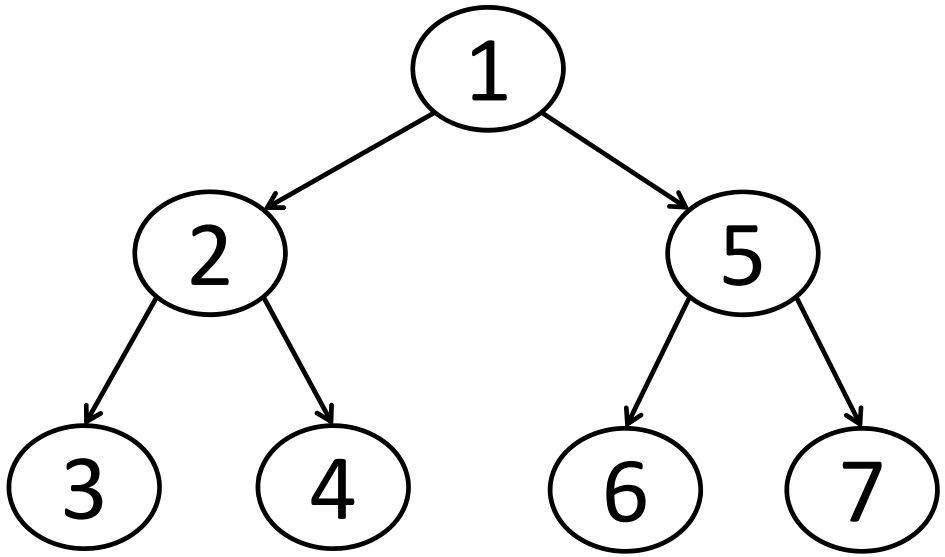
- Si la estructura LNV se trata como una cola (FIFO)

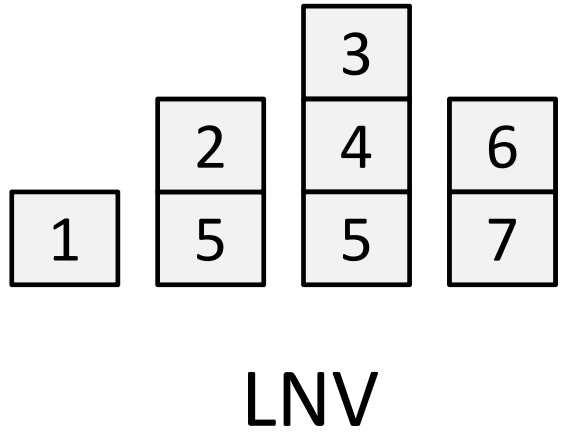




Estrategias de Ramificación. En profundidad

- Si la estructura LNV se trata como una pila (LIFO)





Esquemas algorítmicos. Ramificación y Poda

1. Características generales
2. Estrategias de Ramificación
3. Estrategias de Poda
4. Esquema algorítmico de Ramificación y Poda
5. Estudio de complejidad
6. El problema de la asignación de tareas
7. Problema de la mochila [0,1]
8. Programación Lineal
9. El viajante de comercio
10. El laberinto

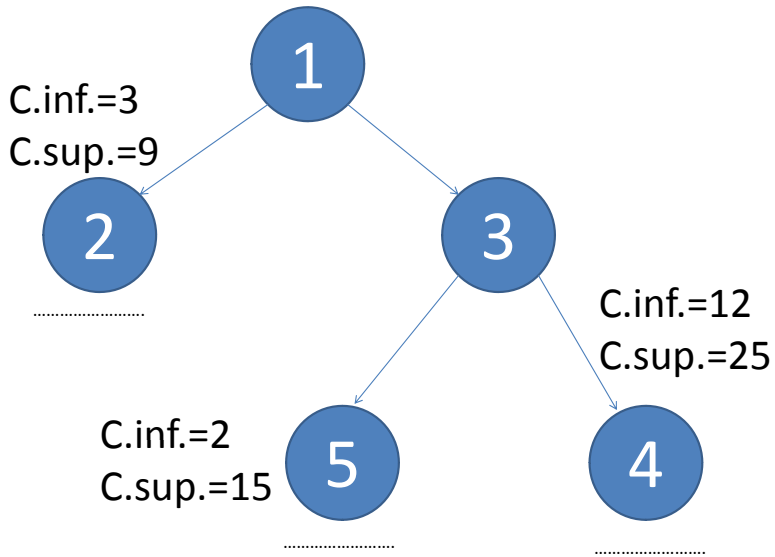
Estrategias de Poda en problemas de optimización

- Para cada nodo establecemos una estimación de la mejor solución posible a partir de él (cota)
- Las cotas determinan cuándo se puede realizar una poda del árbol: si el valor de la cota es peor que la mejor solución obtenida hasta ese momento $\longrightarrow$ no se exploran sus hijos (poda).
- Para cada nodo i podemos tener:
  - **Cota Inferior(i)** de la mejor solución alcanzable a partir del nodo i
  - **Cota Superior(i)** de la mejor solución alcanzable a partir del nodo i
  - **Cota Estimada(i)** de la mejor solución alcanzable a partir del nodo i. Ayuda a decidir que parte del árbol evaluar primero
- Si M(i) es la mejor solución alcanzable a partir del nodo i, se verifica

$$CotaInferior(i) \leq M(i) \leq CotaSuperior(i)$$

Estrategias de Poda

- Supongamos que tenemos un problema de **maximización**
- Tenemos varios nodos y para cada uno de ellos se calcula la cota superior, y la cota inferior



  - Si nos encontramos en el nodo 2, ¿merece la pena expandir dicho nodo?
  - ¿Y el nodo 5? ¿Merece la pena expandirlo?
  - Y si se tratase de un problema de minimización

Estrategias de Poda

Maximización

- Para podar un nodo i se tiene que verificar una de las siguientes condiciones:
  a) $CotaSuperior(i) < CotaInferior(j)$ para algún nodo j ya generado
  b) $CotaSuperior(i) \leq Valor(s)$ para todo nodo s que sea solución final

- O lo que es lo mismo:
  Podar el nodo i si se cumple:

$$CotaSuperior(i) < \texttt{Cota}\ (*)$$

$$\text{siendo } \texttt{Cota} = maximo \begin{cases} CotaInferior(j), & \forall \text{ j generado,} \\ Valor(s), & \forall \text{ s solucion final} \end{cases}$$

- Hay casos en los que no se puede utilizar CotaInferior(). En estos casos, la condición (*) de poda puede ser: $CotaSuperior(i) \leq \texttt{Cota}$

Minimización $\rightarrow$ se deja como ejercicio

Esquemas algorítmicos. Ramificación y Poda

Esquema algorítmico de Ramificación y Poda

- En un algoritmo de ramificación y poda se realizan tres etapas:
  1. Etapa de **Selección**: se encarga de extraer un nodo de la LNV. La forma de escogerlo depende de la estrategia de ramificación
  2. Etapa de **Ramificación**: se generan los posibles hijos del nodo seleccionado en la etapa anterior
  3. Etapa de **Poda**: se estudian los nodos generados en la etapa de ramificación. Solo aquellos que pasan *cierto filtro* se introducen en la LNV. El resto de nodos son podados.
- Se repiten estas tres etapas mientras la LNV tiene nodos pendientes de procesar.

Esquema algorítmico de Ramificación y Poda

- Esquema de alto nivel del algoritmo Ramificación y Poda

Inicialización:
Introducir la raiz del árbol en la estructura **LNV**
Inicializar la variable de poda **C** de forma conveniente
repetir
  Sacar un nodo de **LNV** según la estrategia de ramificación
  Comprobar si dicho nodo debe ser podado según la estrategia de poda
  si el nodo no debe ser podado **entonces**
    Generar todos sus hijos
    desde hijo 1 **hasta** último hijo **hacer**
      Comprobar si es solución final y tratarla
      Comprobar si debe ser podado
      Si no debe ser podado, meterlo en **LNV** y actualizar la cota **C**
    fin desde
  fin si
hasta que la lista **LNV** esté vacía

Esquema algorítmico de Ramificación y Poda

Observaciones:

- Solo se comprueba el criterio de poda cuando se introduce o se saca un elemento de la lista **LNV**
- Los nodos que son solución no se introducen en **LNV**. En este caso, se comprueba si esa solución es mejor que la actual y se actualiza la cota **C** y la **mejor solución** de forma adecuada

Esquema algorítmico de Ramificación y Poda: Una solución

```
proc RyP Una(sol : Nodo) //Busca la primera solución
   crear(nodoInicial) , introducir(Inv, nodoInicial), actualizar(Cota),
   sol ← ∅
   mientras not(vacia(Inv)) hacer
      sacar(Inv, x), calcularHijos(x, hijos)
      desde i = 1  hasta último hijo de x hacer
         si esAceptable(hijos[i]) entonces
            //¿se puede podar?
            si esSolucion(hijos[i]) entonces
               sol ← hijos[i],
               borrarRestoHijos(hijos,i), vaciar(Inv)
            si no
               introducir(Inv, hijos[i]), actualizar(Cota),
            fin si
         fin si
      fin desde
   fin mientras
fin proc
```

Funciones que aparecen en el esquema

- `calcularHijos(nodo, hijos)`: Es la que realiza el **proceso de ramificación** del algoritmo. Se encarga de generar los hijos de un `nodo` dado y los guarda en una tupla denominada `hijos`
- `esAceptable(nodo)`: Es la función que se encarga de realizar el **proceso de poda**. Dado un nodo vivo, decide si seguir analizándolo o bien rechazarlo
- `esSolucion(nodo)`: Decide cuándo un `nodo` es una hoja del árbol, es decir, un candidato a posible solución del problema. No tiene porqué ser la mejor, solo una de ellas
- `borrarRestoHijos(hijos,i)`: Borra el resto de los nodos del array `hijos` y así salir del bucle **desde**

Esquema Ramificación y Poda: Todas las soluciones

- Si queremos encontrar todas las soluciones, basta con modificar ligeramente el esquema anterior

```
proc RyP Todas() //Busca todas las soluciones
  crear(nodoInicial) , introducir(Inv, nodoInicial), actualizar(Cota),
  mientras not(vacia(Inv)) hacer
    sacar(Inv, x), calcularHijos(x, hijos)
    desde i = 1  hasta último hijo de x hacer
      si esAceptable(hijos[i]) entonces
        si esSolucion(hijos[i]) entonces
          sol ← hijos[i],
          procesar(sol)
        si no
          introducir(Inv, hijos[i])
        fin si
      fin si
    fin desde
  fin mientras
fin proc
```

Esquemas algorítmicos. Ramificación y Poda

1. Características generales
2. Estrategias de Ramificación
3. Estrategias de Poda
4. Esquema algorítmico de Ramificación y Poda
5. Estudio de complejidad
6. El problema de la asignación de tareas
7. Problema de la mochila [0,1]
8. Programación Lineal
9. El viajante de comercio
10. El laberinto

Estudio de complejidad

- El orden de complejidad depende del **número de nodos recorridos** y del tiempo gastado en cada nodo
  a) El número de nodos depende de la efectividad de la poda
  b) El tiempo requerido para cada nodo depende de la complejidad en el manejo de la LNV y tiempo gastado en el cálculo de las estimaciones del coste
- En el caso promedio, se obtienen mejoras con respecto al esquema de *backtracking*
- En el peor caso, se recorren tantos nodos como en backtracking $\longrightarrow$ el tiempo puede ser peor (coste del cálculo de cotas y manejo de la LNV)

Esquemas algorítmicos. Ramificación y Poda

1. Características generales
2. Estrategias de Ramificación
3. Estrategias de Poda
4. Esquema algorítmico de Ramificación y Poda
5. Estudio de complejidad
6. El problema de la asignación de tareas
7. Problema de la mochila [0,1]
8. Programación Lineal
9. El viajante de comercio
10. El laberinto

El problema de la asignación de tareas

- Disponemos de n empleados y n tareas a realizar
- Mediante una tabla M de tamaño $n \times n$ representamos el coste de realizar la tarea j por el empleado i, para $i, j = 1, \ldots, n$
- El problema consiste en asignar a cada operario i una tarea j de forma que se minimice el coste total.

...

- La solución se puede representar como una tupla $Sol = \{x_1, x_2, \ldots x_n\}$
- Restricciones explícitas: $x_i \in (1, \ldots n)$
  x_i es la tarea asignada al i-ésimo empleado
- Restricciones implícitas: $x_i \neq x_j$
- El objetivo es minimizar la función $\sum_{i=1}^{n} M[i, Sol[i]]$

El problema de la asignación de tareas (Cont.)

- En la etapa k del algoritmo asignamos las tareas posibles a la persona k
- Las tareas asignables a la persona k son aquellas que no han sido asignadas todavía, por tanto, cada nodo del nivel k tiene como mucho $n - k$ hijos
- En cada nodo vamos a almacenar información sobre la solución alcanzada hasta ese momento

Solución parcial
Nivel actual
Coste real acumulado
`Cinf`
`CSup`

- Si M(i) es la mejor solución alcanzable a partir del nodo i, se verifica

$$Cinf(i) \leq M(i) \leq Csup(i)$$

El problema de la asignación de tareas (Cont.)

- Dado un nodo x, ¿cómo sabemos si es solución?

  esSolución(x) = true **si** x.nivelActual = n
- Dado un nodo x, ¿cómo se ramifica?, ¿cómo se generan los hijos?

```
proc calcularHijos(x, hijos)
  desde i=1 hasta n hacer
    si not(esUsada(i)) entonces
      crearNuevoNodo(y)
      y.nivel ← x.nivel + 1
      y.sol ← x.sol
      y.sol[nivel] ← i
      y.costeReal ← x.costeReal + M[y.nivel, i]
      y.tareasUsadas ← añadir(x.tareasUsadas, i)
      y.cinf ←calcularCinf(y)
      y.csup ←calcularCsup(y)
      insertar(hijos,y)
    fin si
  fin desde
fin proc
```

El problema de la asignación de tareas (Cont.)

- Dado un nodo x, ¿cómo se calculan los valores para `Cinf` y `CSup`?
  - Posibilidad 1:
    - `Cinf` = coste acumulado hasta ese momento
    - `Csup` = coste acumulado hasta ese momento + coste máximo alcanzable
  - Posibilidad 2:
    - `Cinf` = `Coste real acumulado` + `Coste mínimo alcanzable`
    - `Csup` = coste acumulado hasta ese momento + coste máximo alcanzable

  donde:
  Coste mínimo alcanzable se define como resultado de asignar a cada operario la tarea de menor coste de entre aquellas que no han sido asignadas todavía (menor de cada fila aunque se repitan)
  Coste máximo alcanzable se define como resultado de asignar a cada operario la tarea de mayor coste de entre aquellas que no han sido asignadas todavía (mayor de cada fila aunque se repitan)

El problema de la asignación de tareas (Cont.)

- Ejemplo aplicando la alternativa 1

3	**5**	**1**
10	**10**	**1**
8	**5**	**5**

MATRIZ DE TAREAS

Sol	[0,0,0]
Nivel	0
Coste real	0
Cinf	**0**
Csup	**23**

Nodo RAIZ

HIJOS DE LA RAIZ

Sol	[1,0,0]
Nivel	1
Coste real	3
Cinf	**3**
Csup	**18**

Sol	[2,0,0]
Nivel	1
Coste real	5
Cinf	**5**
Csup	**23**

Sol	[3,0,0]
Nivel	1
Coste real	1
Cinf	**1**
Csup	**19**

El problema de la asignación de tareas (Cont.)

- Ejemplo aplicando la alternativa 2

3	**5**	**1**
10	**10**	**1**
8	**5**	**5**

MATRIZ DE TAREAS

Sol	[0,0,0]
Nivel	0
Coste real	0
Cinf	**7**
Csup	**23**

Nodo RAIZ

HIJOS DE LA RAIZ

Sol	[1,0,0]
Nivel	1
Coste real	3
Cinf	**9**
Csup	**18**

Sol	[2,0,0]
Nivel	1
Coste real	5
Cinf	**11**
Csup	**23**

Sol	[3,0,0]
Nivel	1
Coste real	1
Cinf	**16**
Csup	**19**

El problema de la asignación de tareas (Cont.)

En cuanto a la **estrategia de poda**,

- ¿cual es el valor de la **variable de poda C**? El valor de la variable de poda C es el **mínimo** entre las **cotas superiores** calculadas hasta el momento y el **valor de las soluciones finales**
  En el ejemplo anterior, el valor inicial para la variable de poda es C=23
- Dado un nodo x, ¿Qué debe cumplirse para poder podarlo?
  Condición de poda: podar el nodo x si se cumple

$$x.cinf > C$$

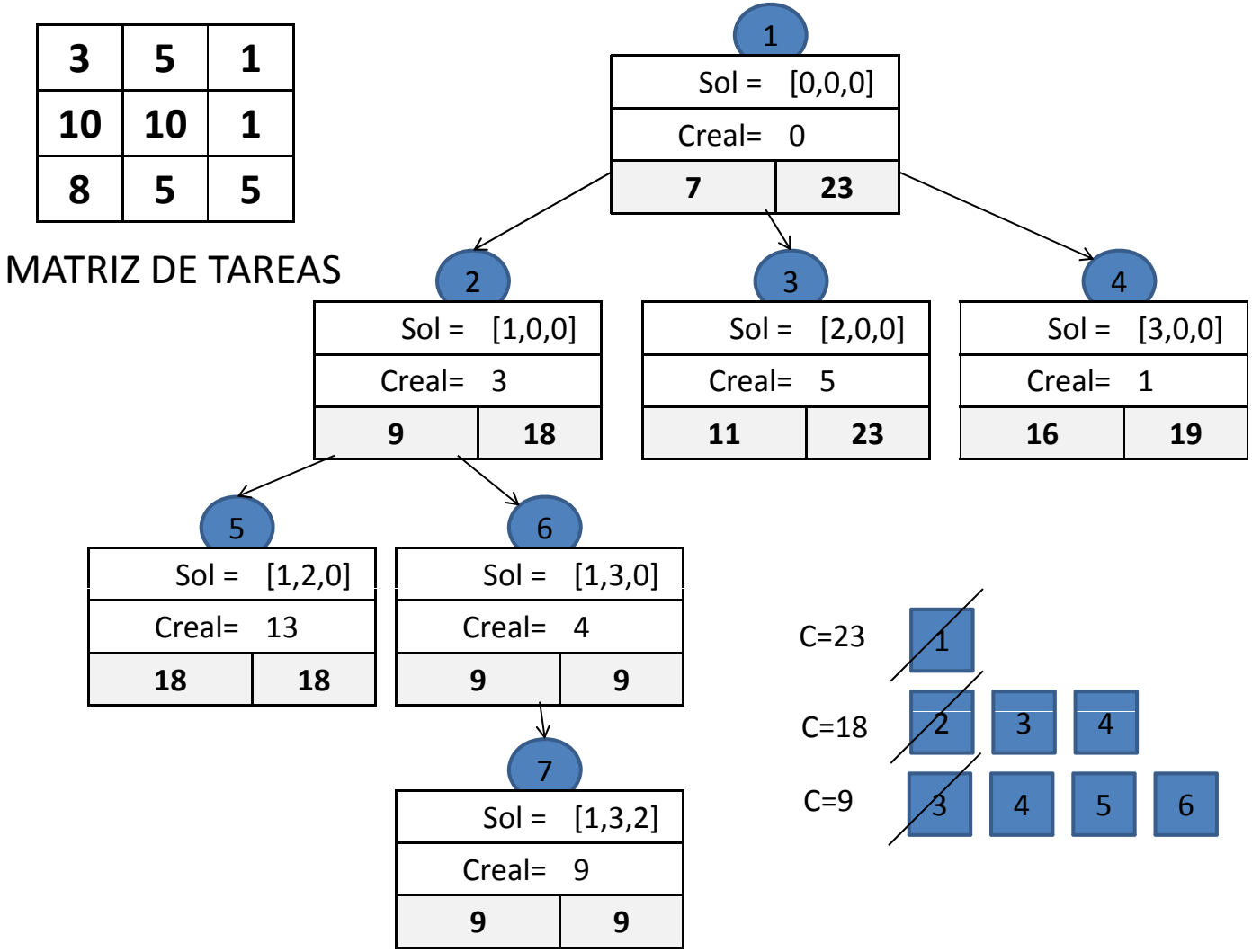
esAceptable(x) = true **si** **x.cinf** $\leq$ **C**

El problema de la asignación de tareas (Cont.)

```
proc RyP Tareas(raiz, M, mejorSol, valormejorSol)
  introducir(Lnv,raiz) ; C ← raiz.Csup ; mejorSol ← ∅
  mientras not(vacia(Lnv)) hacer
    sacar(Lnv,x)
    si x.cinf ≤ C entonces
      calcularHijos(x, hijos)
      desde i = 1  hasta último hijo de x hacer
        si hijos[i].cinf ≤ C entonces
          si esSolucion(hijos[i]) entonces
            si hijos[i].costeReal < valormejorSol entonces
              valormejorSol ← hijos[i].costeReal, mejorSol ← hijos[i].sol
              C ← hijos[i].costeReal
            fin si
          si no introducir(Lnv,hijos[i]) ; C ← min { C, hijos[i].csup }
        fin si
      fin desde
    fin si
  fin mientras
fin proc
```

El problema de la asignación de tareas (Cont.)

- Ejemplo aplicando la alternativa 2 con estrategia FIFO



El problema de la asignación de tareas (Cont.)

- Utilizando alternativa 2 para el cálculo de los valores de `Cinf` y `Csup`, se han generado 7 nodos
- ¿Cuántos nodos se generan utilizando alternativa 1 con una estrategia LIFO?

Esquemas algorítmicos. Ramificación y Poda

1. Características generales
2. Estrategias de Ramificación
3. Estrategias de Poda
4. Esquema algorítmico de Ramificación y Poda
5. Estudio de complejidad
6. El problema de la asignación de tareas
7. Problema de la mochila [0,1]
8. Programación Lineal
9. El viajante de comercio
10. El laberinto

Problema de la mochila [0,1]

- Se dispone de n objetos y una mochila de capacidad $C > 0$,
  - El peso del objeto i es $w_i > 0$
  - La inclusión del objeto i en la mochila produce un beneficio $b_i > 0$
- El objetivo consiste en llenar la mochila maximizando el valor de los objetos transportados sin sobrepasar la capacidad de la mochila
- Los objetos no son fraccionables y suponemos que estan ordenados de mayor a menor $\frac{beneficio}{peso}$

Problema de la mochila [0,1] (cont.)

- La solución se puede representar como una tupla $\{x_1, x_2, \ldots x_n\}$
  $x_i \in (0, 1)$
  Si $x_i = 0$ el objeto i no se introduce en la mochila
  Si $x_i = 1$ el objeto i se introduce en la mochila
- **Restricciones**: $\sum_{i=1}^{n} x_i \cdot w_i \leq C$
- El **objetivo** es maximizar la función $\sum_{i=1}^{n} x_i \cdot b_i$

Problema de la mochila [0,1] (cont.)

- En cada nodo vamos a almacenar información sobre la solución alcanzada hasta ese momento. La estructura de cada nodo es la aiguiente:

Solución actual
Nivel actual
Peso real acumulado
Beneficio real acumulado
Cinf_beneficio
CSup_beneficio

- Cada nodo va a tener como mucho dos hijos, dependiendo en cada paso si incluímos o no el siguiente elemento

Problema de la mochila [0,1] (cont.)

- Hemos de generar el nodo raiz

proc *NodoRaiz*(*raiz*, *Capacidad*, *elem*[1..*n*])
  crearNodo(raiz)
  desde i ← 1 **hasta** n **hacer**
    raiz.*sol*[*i*] = 0
  fin desde
  raiz.pesoAcumulado ← 0
  raiz.benefAcumulado ← 0
  raiz.etapa ← 0
  raiz.CinfBenef ← 0
  raiz.CSupBenef ← calcularCotaSup(raiz, Capacidad, elem)
fin proc

Problema de la mochila [0,1] (cont.)

- Cálculo de las cotas para un nodo i en la etapa k
  - Partimos de la base de que los elementos están ordenados de mayor a menor $\frac{beneficio}{peso}$
  - Cota inferior = Beneficio Acumulado
  - Cota superior = Beneficio Acumulado + (Capacidad - Peso Acumulado) * $\frac{beneficio\ \ del\ \ objeto\ \ k+1}{peso\ \ del\ \ objeto\ \ k+1}$
- Para calcular la cota superior, es decir, el valor máximo que podríamos alcanzar a partir del nodo i, vamos a suponer que rellenáramos el resto de la mochila con el mejor de los elementos que nos quedan por analizar. Como los tenemos dispuestos en orden decreciente de ratio $\frac{beneficio}{peso}$, este mejor elemento será el siguiente ($i.etapa + 1$).
- Este valor, aunque no tiene por qué ser alcanzable, nos permite dar una cota superior del valor al que podemos aspirar si seguimos por esa rama del árbol

Problema de la mochila [0,1] (cont.)

```
fun calcularCotaSup(nodo, Capacidad, elem[1..n])
   mejorObj ← nodo.etapa + 1
   si mejorObj≤ n entonces
      benefPorUnidad ← elem[mejorObj].beneficio / elem[mejorObj].peso
      cotaSup ← nodo.benefAcumulado +
            (Capacidad - nodo.pesoAcumulado)* benefPorUnidad
   si no
      cotaSup ← nodo.benefAcumulado
   fin si
   devolver cotaSup
fin fun
```

Problema de la mochila [0,1] (cont.)

- La estrategia de ramificación está a cargo del procedimiento generarHijos
- Cada nodo del árbol va a tener como mucho 2 hijos, dependiendo si introducimos el siguiente elemento en la mochila o no.
- Solo vamos a generar aquellos nodos que sean válidos en el sentido de si caben en la mochila

Problema de la mochila [0,1] (cont.)

```
proc generarHijos(nodo, hijos[1..2], numHijos, Capacidad, elem[1..n])
  numHijos ← 0, newEtapa ← nodo.etapa +1
  si nodo.etapa < n entonces
    numHijos ← 1 // primer caso: no se introduce el elemento en la mochila
    hijos[1].sol ← nodo.sol ; hijos[1].etapa ← newEtapa
    hijos[1].pesoAcumulado ← nodo.pesoAcumulado
    hijos[1].benefAcumulado ← nodo.benefAcumulado
    hijos[1].cinf ← nodo.benefAcumulado
    hijos[1].csup ← calcularCotaSup(hijos[1], Capacidad, elem)
    si (n.pesoAcumulado + elem[newEtapa].peso) ≤ Capacidad entonces
      // segundo caso: se introduce el elemento en la mochila
      numHijos ← 2 ; hijos[2].etapa ← newEtapa
      hijos[2].sol ← nodo.sol ; hijos[2].sol[newEtapa] ← 1
      hijos[2].pesoAcumulado ← nodo.pesoAcumulado + elem[newEtapa].peso
      hijos[2].benefAcumulado ← nodo.benefAcumulado + elem[newEtapa].beneficio
      hijos[2].cinf ← hijos[2].benefAcumulado
      hijos[2].csup ← calcularCotaSup(hijos[2], Capacidad, elem)
    fin si
  fin si
fin proc
```

Problema de la mochila [0,1] (cont.)

- En cuanto a la estrategia de poda, es necesario actualizar la variable de poda Cotacada vez que guardamos un nodo en la lista de nodos vivos o encontramos una nueva solución

  proc *valorPoda*(*nodo*, *Cota*)
  Cota $\leftarrow$ max $\{$ Cota, nodo.cinf $\}$
  fin proc

- Así, un nodo i es podado, si se cumple

$$i.csup < Cota$$

Problema de la mochila [0,1] (cont.)

- Otra de las funciones que es necesario implementar es la que determina cuándo un nodo es solución

  fun *esSolucion*(*nodo*)
  devolver (nodo.etapa = n)
  fin fun

Problema de la mochila [0,1] (cont.)

```
proc RyP_Mochila01(mejorSol[1..n], Cota, Capacidad, elem[1..n])
  nodoRaiz(raiz,Capacidad,elem), mejorSol ← raiz.sol, Cota ← 0
  introducir(lnv, raiz), valorPoda(Cota, raiz),
  mientras not(vacia(Lnv)) hacer
    sacar(Lnv,x)
    si x.csup ≥ Cota entonces
      generarHijos(x, hijos, numhijos, Capacidad, elem)
      desde i = 1  hasta numhijos hacer
        si (hijos[i].csup ≥ Cota) entonces
          si  esSolucion(hijos[i])
            si hijos[i].benefAcumulado ≥ Cota entonces
              mejorSol ← hijos[i].sol ; Cota ← hijos[i].benefAcumulado
            fin si
          si no
            valorPoda(hijos[i], Cota) ; introducir(Lnv,hijos[i])
          fin si
        fin si
      fin desde
    fin si
  fin mientras
fin proc
```

Esquemas algorítmicos. Ramificación y Poda

1. Características generales
2. Estrategias de Ramificación
3. Estrategias de Poda
4. Esquema algorítmico de Ramificación y Poda
5. Estudio de complejidad
6. El problema de la asignación de tareas
7. Problema de la mochila [0,1]
8. Programación Lineal
9. El viajante de comercio
10. El laberinto

Programación Lineal

- Consideremos la siguiente ecuación:

$$A_n X_n + ... + A_1 X_1 + A_0 = C$$
$$0 \leq x_i < C$$
$$x_i \text{ enteros}$$
$$A_i, C \text{ reales positivos (datos conocidos)}$$

- Diseña un algoritmo de ramificación y poda que maximice el valor de la función

$$B_n X_n^n + ... + B_1 X_1 + B_0$$
$$B_i \text{ reales positivos (datos conocidos)}$$

Programación Lineal (cont.)

- La solución se puede representar como una tupla $\{x_1, x_2, \ldots x_n\}$ con las siguienes restricciones:
  1. x_i enteros
  2. $0 \leq x_i < C$
- La información del problema la podemos representar de la siguiente manera:
  1. Los coeficientes $A_0, A_1, ..., A_n$ los representamos mediante una tupla de n+1 componentes $\{A_0, A_1, \ldots A_n\}$
  2. Los coeficientes $B_0, B_1, ..., B_n$ los representamos mediante una tupla de n+1 componentes $\{B_0, B_1, \ldots B_n\}$
- El **objetivo** es maximizar la función $B_n X_n^n + ... + B_1 X_1 + B_0$

Programación Lineal (cont.)

- Cada nivel del árbol de expansión representa el procesamiento de una variable, por tanto, tendremos un **inbluemáximo** de n niveles
- Cada arista de un nodo en el nivel K, se corresponde con la asignación de un valor para la variable x_{k+1}
- ¿Cuantos hijos tiene cada nodo del árbol de expansión?
  Cada nodo del árbol tiene a lo sumo $\lfloor C \rfloor + 1$ hijos
- ¿Qué información debe contener cada nodo del árbol?
  Cada nodo del árbol debe contener la información necesaria para recuperar a partir de él la solución construida hasta ese momento. La estructura de cada nodo es la siguiente:

Solución actual
Nivel actual
Valor de la ecuación
Valor de la función a maximizar
CSup_función

Programación Lineal (cont.)

- Hemos de generar el nodo raiz

```
proc nodoRaiz(raiz,B[0..n],A[0..n],C)
    crearNodo(raiz)
    desde i ← 1 hasta n hacer
        raiz.sol[i] ← 0
    fin desde
    raiz.etapa ← 0
    raiz.ValorFun ← B[0]
    raiz.ValorEcu ← A[0]
    raiz.CsupFun ← calcularCotaSup(raiz, B,C)
fin proc
```

Programación Lineal (cont.)

- Cálculo de las cotas para un nodo i en la etapa k
- Para calcular la cota superior, es decir, el valor máximo que podríamos alcanzar a partir del nodo i, vamos a suponer que las variables que quedan por analizar toman su valor máximo.
  - Cota superior de la función = Valor de función suponiendo que el resto de variables que quedan por analizar $(x_{k+1}, ..., x_n)$ toman el valor máximo C
- Como siempre, este valor puede que no sea alcanzable, nos permite dar una cota superior del valor al que podemos aspirar si seguimos por esa rama del árbol

Programación Lineal (cont.)

```
fun calcularCotaSup(nodo, B[0..n], C)
   cotaSup ← nodo.ValorFun
   nivel ← nodo.etapa
   desde i ← nivel + 1 hasta n hacer
      cotaSup ← cotaSup + exp(C, i) × B[i]
   fin desde
   devolver cotaSup
fin fun
```

Programación Lineal (cont.)

- La estrategia de ramificación está a cargo del procedimiento generarHijos
- Cada nodo del árbol va a tener como mucho $\lfloor C \rfloor + 1$ hijos
- $\lfloor C \rfloor$ representa la parte entera de C.

```
proc generarHijos(padre, hijos[0..⌊C⌋], numHijos, A[0..n], B[0..n], C)
  numHijos ← 0 ; newEtapa ← padre.etapa +1
  si padre.etapa < n entonces
    desde i = 0  hasta ⌊C⌋ hacer
      crearNodo(hijos[i])
      hijos[i].sol ← padre.sol
      hijos[i].sol[newEtapa] ← i
      hijos[i].etapa ← newEtapa
      hijos[i].ValorEcu ← padre.ValorEcu + i×A[newEtapa]
      hijos[i].ValorFun ← padre.ValorFun + exp(i,newEtapa)×B[newEtapa]
      hijos[i].CsupFun ← calcularCotaSup(hijos[i], B, C)
      numHijos ← numHijos +1
    fin desde
  fin si
fin proc
```

Programación Lineal (cont.)

- En cuanto a la estrategia de poda, es necesario actualizar la variable de poda Cota cada vez que encontremos una solución. Si nodo.sol es solución, entonces:

  proc *valorPoda*(*nodo*, *Cota*)
  Cota $\leftarrow$ max { Cota, nodo.ValorFun }
  fin proc

- El valor inicial de la variable de poda es -1
- Un nodo es podado, si se cumple una de las dos siguientes condiciones:
  - nodo.ValorEcu $\geq$ C
  - nodo.CsupFun $<$ Cota

Programación Lineal (cont.)

- Otra de las funciones que es necesario implementar es la que determina cuándo un nodo es solución
- Cualquier nodo n del árbol puede ser solución. Basta con comprobar que el valor de la ecuación hasta ese momento sea igual a C

```
fun esSolucion(nodo, C)
   si nodo.ValorEcu = C entonces
      devolver true
   si no
      devolver false
   fin si
fin fun
```

Programación Lineal (cont.)

```
proc RyP_ProgLineal(mejorSol[1..n], A[0..n], B[0..n], C)
   nodoRaiz(raiz,B,A,C), Cota ← -1
   si esSolucion(raiz,C) entonces
      mejorSol ← raiz.sol, Cota ← B[0]
   si no
      introducir(Lnv, raiz)
      mientras not(vacia(Lnv)) hacer
         sacar(Lnv,x)
         si (x.CsupFun ≥ Cota) and (x.ValorEcu ≤ C) entonces
            generarHijos(x, hijos, numhijos, A, B, C)
            añadirHijosLNV(hijos, numhijos, mejorSol, C, lnv, Cota)
         fin si
      fin mientras
   fin si
   si Cota = -1 entonces No hay solucion
fin proc
```

Programación Lineal (cont.)

- Se añaden a la lista de nodos vivos LNV los hijos que no puedan se podados y no sean solución

```
proc añadirHijosLNV(hijos[0..⌊C⌋], numhijos, mejorSol[1..n], C, lnv, Cota)
  desde i ← 1  hasta numhijos hacer
    si (hijos[i].CsupFun ≥ Cota) and hijos[i].ValorEcu ≤ C) entonces
      si esSolucion(hijos[i],C) entonces
        si hijos[i].valorFun > Cota entonces
          mejorSol ← hijos[i].sol
          valorPoda(hijos[i], Cota)
        fin si
      si no
        introducir(Lnv, hijos[i])
      fin si
    fin si
  fin desde
fin proc
```

Esquemas algorítmicos. Ramificación y Poda

1. Características generales
2. Estrategias de Ramificación
3. Estrategias de Poda
4. Esquema algorítmico de Ramificación y Poda
5. Estudio de complejidad
6. El problema de la asignación de tareas
7. Problema de la mochila [0,1]
8. Programación Lineal
9. El viajante de comercio
10. El laberinto

El viajante de comercio

([NN98], p. 236.)

- Se conocen las distancias entre un cierto número de ciudades.
- Un viajante debe, a partir de una de ellas, visitar cada ciudad exactamente una vez y regresar al punto de partida habiendo recorrido en total la menor distancia posible.
- Este problema se reduce a encontrar el ciclo hamiltoniano de menor coste que comienza y termina en un vértice v_1.

El viajante de comercio (cont.)

- Las distancias entre las ciudades se pueden representar mediante la matriz de adyacencia del grafo que representa el mapa.
- Debemos plantear la solución del problema como una secuencia de decisiones, una en cada etapa.
- La solución del problema será un vector que indica el orden en el que se deben visitar los vértices del grafo de ciudades.
- Cada elemento del vector debe contener un número entre 1 y N
- Inicialmente el vector solución contiene un solo elemento que representa el vértice de origen (el 1).
- Como se deben considerar los ciclos hamiltonianos, en la solución no se puede repetir ningún vértice.
- Las posibles soluciones del problema estarán exclusivamente en las hojas del árbol correspondientes a la etapa N. Puede haber ramas que no lleven a ninguna solución
- El resto de nodos del árbol contienen soluciones parciales

El viajante de comercio (cont.)

- **notación**: utilizaremos **vértice** para representar las ciudades del grafo, y **nodo** para referirnos a los nodos del árbol de expansión
- Como es un problema de minimización, debemos calcular una cota inferior para cada nodo del árbol de expansión: un valor que sea menor a la longitud de cualquier recorrido formado a partir de la solución parcial contenida en ese nodo.
- En cualquier recorrido que realicemos, la longitud de la arista que se seleccione para abandonar un vértice dado debe ser como mínimo la longitud de la arista mínima que sale de dicho vértice.
- Podemos utilizar como cota inferior de un nodo la siguiente:
  a) la suma de las longitudes de las aristas mínimas que salen de los vértices pendientes de incluir en la solución,
  b) más la longitud del camino que forma la solución parcial contenida en el nodo.
- Lo habitual es que esta cota inferior no corresponda a ningún ciclo hamiltoniano válido, pero con seguridad no habrá ningún ciclo de longitud inferior a esta cota.

El viajante de comercio (cont.)

- Por ejemplo, dada la siguiente matriz de adyacencia:

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

- El camino de longitud mínima es: [1, 4, 5, 2, 3, 1]
- Supongamos que ya tenemos una solución parcial [1, 4].
  - La longitud de la solución parcial es 10.
  - Las aristas de longitud mínima que salen de los vértices no visitados son:

2:	mín(14,7,7)=	7	4:	mín(7,9,2)=	2
3:	mín(4,5,16)=	4	5:	mín(18,7,17)=	7

  - No hemos considerado las aristas dirigidas al vértice 4, pues no es posible volver a visitar este vértice, ni la arista del 4 al 1
  - Por tanto, la cota inferior de la solución parcial [1, 4] es

$$10 + 7 + 4 + 2 + 7 = 30$$

El viajante de comercio (cont.)

- Cada nodo del árbol debe contener toda la información necesaria para realizar la ramificación y la poda, así como la solución parcial obtenida hasta el momento.
- La estructura de cada nodo puede ser:

Solución actual
Nivel actual
longitud solución actual
Cota inferior

- Puede haber ramas del árbol que no lleven a ninguna solución: la cota solamente se actualiza cuando se encuentra una solución.
- Como se va a calcular el camino de longitud mínima, no es necesario calcular la cota superior de cada nodo.
- La longitud de la solución actual no es estrictamente necesaria, pero evita recalcularla para cada nodo.

El viajante de comercio (cont.)

```
proc RyP_viajante(D[1..N,1..N],mejorSol[1..N], Cota)
   crear Lnv // montículo
   nodoRaiz(raiz)
   Cota ← ∞
   introducir(Lnv, raiz)
   mientras not(vacia(Lnv)) hacer
      sacar(Lnv,x)
      si x.CinfLongitud < Cota entonces
         generarHijos(x, hijos, numhijos, D)
         añadirHijosLNV(hijos, numhijos, mejorSol, Lnv, Cota)
      fin si
   fin mientras
fin proc
```

El viajante de comercio (cont.)

- Solamente se generan los hijos de un nodo que no han sido visitados anteriormente.

```
proc generarHijos(padre, hijos[1..N], numHijos, D[1..n,1..n])
  numHijos ← 0 ; newEtapa ← padre.etapa +1
  si padre.etapa < n entonces
    desde i = 1  hasta N hacer
      si no_visitado(i, padre.sol, padre.etapa) entonces
        numHijos ← numHijos +1
        crear hijos[numHijos]
        hijos[numHijos].sol ← padre.sol
        hijos[numHijos].sol[newEtapa] ← i
        hijos[numHijos].etapa ← newEtapa
        hijos[numHijos].longitud ← padre.longitud + D[padre.sol[padre.etapa],i]
        hijos[numHijos].CinfLongitud ← calcularCotaInf(hijos[numHijos],D)
      fin si
    fin desde
  fin si
fin proc
```

El viajante de comercio (cont.)

```
proc añadirHijosLNV(hijos[1..N], numhijos, mejorSol[1..n], Lnv, Cota)
  desde i ← 1  hasta numhijos hacer
    si hijos[i].CinfLongitud < Cota entonces
      si hijos[i].etapa = N entonces
        longCamino ← hijos[i].longitud + D[hijos[i].sol[hijos[i].etapa],1]
        si longCamino < Cota entonces
          mejorSol ← hijos[i].sol
          Cota ← longCamino //nuevo valor de la cota inferior
        fin si
      si no
        introducir(Lnv, hijos[i], hijos[i].CinfLongitud)
        //montículo ordenado por la cota inferior
      fin si
    fin si
  fin desde
fin proc
```

El viajante de comercio (cont.)

proc nodoRaiz(raiz,D[1..N,1..N])
  crear raiz
  raiz.sol[1] $\leftarrow$ 1
  raiz.etapa $\leftarrow$ 1
  raiz.longitud $\leftarrow$ 0
  raiz.CinfLongitud $\leftarrow$ calcularCotaInf(raiz,D)
fin proc
fun no_visitado(v,sol[1..N],etapa)
  desde i $\leftarrow$ 1 **hasta** etapa **hacer**
    si v = sol[i] **entonces devolver** falso
  fin desde
  devolver cierto
fin fun

- ¿Cómo se puede diseñar la función `calcularCotaInf(nodo,D[1..N,1..N])`?

El viajante de comercio (cont.)

```
fun calcularCotaInf(nodo,D[1..N,1..N])
   cotaInferior ← nodo.longitud
   minFila ←∞
   desde i ← 1 hasta N hacer
      si no_visitado(i, nodo.sol, (nodo.etapa)-1) entonces
         minFila ←∞
         desde j ← 1 hasta N hacer
            si (i≠j) and (no_visitado(j, nodo.sol, nodo.etapa) or (j=1)) entonces

               minFila ← min{ minFila, D[i,j]}
            fin si
         fin desde
         cotaInferior ← cotaInferior+ minFila
      fin si
   fin desde
   devolver cotaInferior
fin fun
```

Esquemas algorítmicos. Ramificación y Poda

1. Características generales
2. Estrategias de Ramificación
3. Estrategias de Poda
4. Esquema algorítmico de Ramificación y Poda
5. Estudio de complejidad
6. El problema de la asignación de tareas
7. Problema de la mochila [0,1]
8. Programación Lineal
9. El viajante de comercio
10. El laberinto

El laberinto

(basado en [GV00], p. 278).

- Una matriz bidimensional $n \times n$ puede representar un laberinto cuadrado. Cada posición contiene un entero no negativo que indica si la casilla es transitable (0) o si no lo es (∞).
- Las casillas $(1, 1)$ y (n, n) corresponden a la entrada y salida del laberinto y siempre son transitables.
- Dada una matriz con un laberinto, el problema consiste en diseñar un algoritmo que encuentre un camino, si existe, para ir de la entrada a la salida con el menor número de pasos.

El laberinto (Cont.)

- Vamos a utilizar la propia matriz que define el laberinto (a la que llamaremos L) para definir el estado en que se encuentra cada casilla. Así:
  a) L[i,j] = 0 si la casilla no ha sido visitada
  b) L[i,j] = ∞ si la casilla no es transitable
  c) L[i,j] = k siendo k un número entre 1 y n$*$n
- La solución vendrá representada en la propia matriz, como la secuencia de movimientos que se han dado para alcanzar la casilla (n,n)

1			∞	∞	∞
	∞	∞		∞	
		∞		∞	
∞		∞			
				∞	

1			∞	∞	∞
2					
3	∞	∞		∞	
4	5	∞		∞	
∞	6	∞	10	11	12
	7	8	9	∞	13

El laberinto (Cont.)

- En cada nodo vamos a almacenar información sobre la solución alcanzada hasta ese momento. La estructura de cada nodo es la aiguiente:

Solución actual:L
Nivel actual
Casilla donde nos encontramos
Cota Inferior Pasos

- El nivel o etapa representa el número de pasos dados hasta el momento
- La cota inferior es el mínimo número estimado de pasos que hay que dar para llegar a la casilla (n,n) desde la casilla donde nos encontramos
- Los hijos de un nodo representan los posibles movimientos dentro del tablero desde la casilla donde nos encontramos.

El laberinto (Cont.)

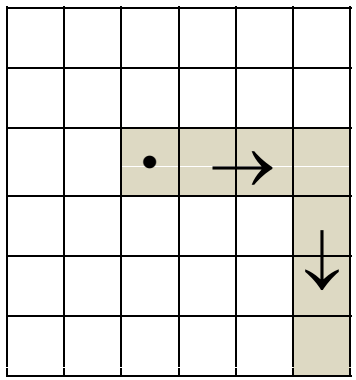
- Generación del nodo raiz

proc nodoRaiz(raiz,L[1..N,1..N])
  crear raiz
  raiz.sol ← L
  raiz.sol[1,1] ← 1
  raiz.etapa ← 1
  raiz.x ← 1
  raiz.y ← 1
  raiz.CinfPasos ← calcularCotaInf(raiz)
fin proc

El laberinto (Cont.)

- Cálculo de la cota inferior $\rightsquigarrow$ el mínimo número estimado de movimientos necesarios para alcanzar la casilla (n,n) desde la casilla en la que nos encontramos
- La cota inferior corresponde a la distancia de Manhattan desde la posición en la que nos encontramos a la casilla final

```
fun calcularCotaInf(nodo)
    devolver (n−nodo.x)+(n−nodo.y)
fin fun
```



El laberinto (Cont.)

- La estrategia de ramificación está a cargo del procedimiento generar hijos.
- Cada nodo puede generar un máximo de 4 hijos, que son los correspondientes a los posibles movimientos que podemos realizar desde una casilla (arriba, izquierda, abajo, derecha)
- Un movimiento es válido si se cumplen las siguientes restriciones:
  a) No salirse del tablero
  b) No moverse a una casilla previamente visitada
  c) No moverse a una casilla intransitable

El laberinto (Cont.)

```
dX = [ -1, 0,1, 0] ; dY = [ 0, -1, 0,1] // vectores globales
proc generarHijos(padre, hijos[1..numhijos], numhijos)
   numhijos ← 0
   desde k ← 1 hasta 4 hacer
      i ← padre.x+dX[k]
      j ← padre.y+dY[k]
      si (0 < i ≤ n) ∧ (0 < i ≤ n) ∧ (padre.sol[i,j]=0) entonces
         numhijos ← numhijos +1
         hijos[numhijos].sol ← padre.sol
         hijos[numhijos].sol[i,j] ← padre.etapa +1
         hijos[numhijos].x ← i
         hijos[numhijos].y ← j
         hijos[numhijos].CinfPasos ← padre.etapa + calcularCotaInf(hijos[numhijos])
      fin si
   fin desde
fin proc
```

El laberinto (Cont.)

- El orden en el que se generan los nodos es importante si seguimos una estrategia de ramificación FIFO o LIFO.
- En el caso de usar un montículo ordenado por el valor de la cota inferior como estructura de LNV, el orden en el que se generan los nodos no va a ser importante.

El laberinto (Cont.)

- En cuanto a la estrategia de poda, es necesario crear una variable de poda Cotaque se actualizará cada vez que encontremos un nodo solución. Se corresponde con el mínimo número de pasos que se han dado para alcanzar la casilla (n,n). Inicializaremos su valor a ∞.

$$Cota \leftarrow min\{Cota, nodo.etapa\}$$

- Un nodo es podado si cumple una de las dos condiciones siguientes
  a) nodo.nivel $>$ Cota
  b) nodo.CinfPasos $>$ Cota

El laberinto (Cont.)

```
proc RyP_laberinto(L[1..N,1..N],mejorSol[1..N,1..N], Cota)
   crear Lnv // montículo
   nodoRaiz(raiz,L )
   Cota ← ∞
   introducir(Lnv, raiz)
   mientras not(vacia(Lnv)) hacer
      sacar(Lnv,x)
      si x.cinfPasos < Cota entonces
         generarHijos(x, hijos, numhijos)
         añadirHijosLNV(hijos, numhijos, mejorSol, Lnv, Cota)
      fin si
   fin mientras
fin proc
```

El laberinto (Cont.)

```
proc añadirHijosLNV(hijos[1..N], numhijos, mejorSol[1..N,1..N], Lnv, Cota)
  desde i ← 1  hasta numhijos hacer
    si hijos[i].CinfPasos < Cota entonces
      si (hijos[i].x = N) AND (hijos[i].y = N) entonces
        mejorSol ← hijos[i].sol
        Cota ← hijos[i].etapa //nuevo valor de la cota inferior
      si no
        introducir(Lnv,hijos[i],hijos[i].CinfPasos)
        //montículo ordenado por la CinfPasos
      fin si
    fin si
  fin desde
fin proc
```