Ejercicio 1. Ramificación y Poda

[ITIS/ITIG, junio 2007]

(4 puntos) En una región existen N ciudades comunicadas por carreteras. Algunas de las carreteras cruzan por debajo de puentes. Deseamos enviar un camión de una ciudad a otra con una carga de altura descomunal de tal forma que pueda no cruzar algunos puentes. Son conocidas las carreteras existentes y la localización y altura de todos los puentes. Diseña un algoritmo de ramificación y poda que determine el trayecto de longitud mínima para un par de ciudades dadas y una altura de la carga dada. Detalla lo siguiente:

- La declaración de tipos y/o variables para representar la información del problema (0,5 puntos).
- El árbol de búsqueda: significado de las aristas y de los niveles (0,5 puntos).
- El código del procedimiento (2 puntos).
- La función cota utilizada (1 punto).

Solución Ejercicio 1

- Las distancias entre las ciudades y los puentes se pueden representar mediante una matriz de estructuras con dos campos:
  1. M[i,j].distancia representa la longitud de la carretera que hay de la ciudad i a la ciudad j
  2. M[i,j].altura representa la altura del puente que hay en dicha carretera
- Por otro lado, debemos poder representar la existencia o no de puentes y la existencia o no de carreteras.
  1. M[i,j].distancia $= \infty$, si no hay carretera entre la ciudad i y la j
  2. M[i,j].altura $= \infty$, si no hay puente en dicha carretera
- La altura de la carga la representamos como una variable entera A

Solución Ejercicio 1 (Cont.)

- 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 las ciudades.

$$Sol[x_1, \ldots, x_k] \ con \ k \leq n$$

- Además tenemos las siguientes restricciones
  - x_1 se corresponde con la ciudad origen
  - x_k se corresponde con la ciudad destino
  - $x_i \neq x_j$ es decir, no podemos pasar dos veces por la misma ciudad
- Además:
  - Ha de existir una carretera desde x_i hasta x_{i+1} con $0 < i < k$

$$M[x_i, x_{i+1}].distancia \neq \infty$$

  - La altura del puente en caso de que exista, debe ser mayor que la altura del camión

$$M[x_i, x_{i+1}].altura \geq A$$

Solución Ejercicio 1 (Cont.)

- La estructura de cada nodo puede ser:

Solución actual
Nivel actual
Ciudades visitadas
Longitud solución actual
Cota inferior

- Puede haber ramas del árbol que no lleven a ninguna solución: la cota Superior no es necesario calcularla ya que no nos sirve para actualizar el valor de la variable de poda Cota
- La cota inferior de cada nodo se corresponde con la mínima distancia que hay que recorrer desde la ciudad origen hasta la ciudad destino. Dado un nodo, su cota inferior se calcula como la suma de:
  - Distancia ya recorrida
  - Mínima distancia que es necesario recorrer para llegar a la ciudad destino. Bastará con tomar la arista de menor coste desde la ciudad en la que nos encontramos, es decir, suponemos que con un paso más se llega a la ciudad destino por dicha arista.

Solución Ejercicio 1 (Cont.)

```
proc nodoRaiz(raiz,M[1..N,1..N],origen, destino)
   crear raiz
   desde i ← 1 hasta N hacer raiz.visitadas[i] ← falso ; raiz.sol[i] ← 0
   raiz.sol[1] ← origen
   raiz.visitadas[origen] ← cierto
   raiz.etapa ← 1
   raiz.longitud ← 0
   raiz.CinfLongitud ← calcularCotaInf(raiz,M, destino)
fin proc

fun no_visitado(v,visitadas[1..N])
   devolver ¬visitadas[v]
fin fun
```

Solución Ejercicio 1 (Cont.)

```
fun calcularCotaInf(nodo,M[1..N,1..N], destino)
   cotaInferior ← nodo.longitud
   actual ← nodo.sol[nodo.etapa]
   si actual ≠ destino entonces
      minFila ←∞
      desde i ← 1 hasta N hacer
         si no_visitado(i, nodo.visitadas) and (M[actual,i].altura ≥ A) entonces
            minFila ← min{ minFila, M[actual,i].distancia}
         fin si
      fin desde
      cotaInferior ← cotaInferior + minFila
   fin si
   devolver cotaInferior
fin fun
```

- Un nodo es podado si cumple una de las dos condiciones:
  1) nodo.longitud $\geq$ Cota
  2) nodo.CInfLongitud $\geq$ Cota

Solución Ejercicio 1 (Cont.)

- Solo son hijos válidos los correspondientes a ciudades no visitadas, accesibles desde la ciudad en la que nos encontramos y que tengan un puente suficientemente alto. En el nivel k hay a lo sumo n-k hijos

```
proc generarHijos(padre, hijos[1..N], numHijos, M[1..N,1..N], destino)
  numHijos ← 0 ; newEtapa ← padre.etapa +1
  desde i = 1  hasta N hacer
    si no_visitado(i, padre.visitadas) AND (M[padre.sol[padre.etapa], i].distancia
    ≠ ∞) AND (M[padre.etapa, i].altura ≥ A)  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 +
                                 M[padre.sol[padre.etapa],i].distancia
      hijos[numHijos].CinfLongitud ← calcularCotaInf(hijos[numHijos],M, destino)
      hijos[numHijos].visitadas ← padre.visitadas
      hijos[numHijos].visitadas[i] ← cierto
    fin si
  fin desde
fin proc
```

Solución Ejercicio 1 (Cont.)

```
proc RyP_trayectoCamion(M[1..N,1..N],mejorSol[1..N], Cota,
origen,destino)
   crear Lnv // montículo ordenado por cota inferior
   nodoRaiz(raiz, M, origen, destino)
   Cota ← ∞
   introducir(Lnv, raiz)
   mientras not(vacia(Lnv)) hacer
      sacar(Lnv,x)
      si x.CinfLongitud < Cota entonces
         generarHijos(x, hijos, numhijos, M, destino)
         añadirHijosLNV(hijos, numhijos, mejorSol, Lnv, Cota, destino)
      si no
         vaciar(Lnv)
      fin si
   fin mientras
fin proc
```

Solución Ejercicio 1 (Cont.)

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

Ejercicio 2. Ramificación y Poda

[ITIS/ITIG, Sep 2007] En el departamento de una empresa de traducciones se desea hacer traducciones de textos entre varios idiomas. Se dispone de algunos diccionarios. Cada diccionario permite la traducción (bidireccional) entre dos idiomas. En el caso más general, no se dispone de diccionarios para cada par de idiomas por lo que es preciso realizar varias traducciones. Dados N idiomas y M diccionarios, determina si es posible realizar la traducción entre dos idiomas dados y, en caso de ser posible, determina la cadena de traducciones de longitud mínima.
Diseña un algoritmo de ramificación y poda que resuelva el problema detallando lo siguiente:

1. El árbol de búsqueda: significado de las aristas y de los niveles (0,5 puntos).
2. El código del procedimiento (2,5 puntos).
3. La función cota utilizada (1 punto).

Solución Ejercicio 2

- Este problema ya lo hemos visto en los temas de programación dinámica y *backtracking*, y también se puede resolver con el algoritmo de Dijkstra.
- Como en *backtracking*, la implementación basada en ramificación y poda es muy ineficiente, comparada con las implementaciones voraz y dinámica.
- La entrada es la matriz de adyacencia (diccionarios) del grafo de traducciones (matriz binaria y **simétrica**).
- La solución de este algoritmo es la lista de idiomas por los que hay que pasar para obtener la traducción (o bien indicar que la traducción no es posible).
- La solución se puede representar mediante una secuencia $\langle x_1, \ldots, x_k \rangle$, donde x_1 es el idioma de origen y x_k el idioma de destino.
- No puede repetirse ninguno de los idiomas de la secuencia, ya que debe encontrarse la cadena de traducciones de longitud mínima.

Solución Ejercicio 2 (Cont.)

- La estructura de cada nodo puede ser:

Solución actual
Nivel actual
Idiomas visitados
Cota inferior

- Puede haber ramas del árbol que no lleven a ninguna solución: no es necesario calcular la cota Superior de cada nodo ya que no nos sirve para actualizar el valor de la variable de poda Cota
- La cota inferior de cada nodo debe ser inferior al número mínimo de traducciones que hay que realizar desde el idioma origen hasta el idioma destino. Dado un nodo (distinto del idioma destino), su cota inferior se puede calcular como la suma de:
  - traducciones ya realizadas.
  - Mínimo número de traducciones a realizar para llegar al idioma destino. Podemos considerar como cota inferior que una sola traducción más es suficiente para llegar al idioma de destino.

Solución Ejercicio 2 (Cont.)

```
proc RyP_trad(D[1..N,1..N],mejorSol[1..N], Cota, origen,destino)
  crear Lnv // montículo ordenado por la cota inferior
  nodoRaiz(raiz, D, origen,destino)
  Cota ← ∞
  introducir(Lnv, raiz)
  mientras not(vacia(Lnv)) hacer
    sacar(Lnv,x)
    si x.Cinf < Cota entonces
      generarHijos(x, hijos, numhijos, D, destino)
      añadirHijosLNV(hijos, numhijos, mejorSol, Lnv, Cota, destino)
    si no
      vaciar(Lnv)
    fin si
  fin mientras
fin proc
```

Solución Ejercicio 2 (Cont.)

```
proc nodoRaiz(raiz,D[1..N,1..N],origen, destino)
   crear raiz
   raiz.sol[1] ← origen
   raiz.etapa ← 1
   raiz.Cinf ← calcularCotaInf(raiz, destino)
   raiz.visitados[origen] ← true
fin proc

fun calcularCotaInf(nodo, destino)
   si nodo.sol[nodo.etapa] = destino entonces
      devolver nodo.etapa //valor real
   si no
      devolver nodo.etapa+1 //valor estimado
   fin si
fin fun
```

Solución Ejercicio 2 (Cont.)

- Solo son hijos válidos los idiomas no utilizados y a los que se puede traducir desde el idioma actual

proc generarHijos(padre, hijos[1..N], numHijos, D[1..N,1..N], destino)
  numHijos $\leftarrow$ 0 ; newEtapa $\leftarrow$ padre.etapa +1
  desde i = 1 **hasta** N **hacer**
    si ¬visitadas[i] AND D[padre.sol[padre.etapa], i] **entonces**
      numHijos $\leftarrow$ numHijos +1 ; **crear** hijos[numHijos]
      hijos[numHijos].sol $\leftarrow$ padre.sol ; hijos[numHijos].sol[newEtapa] $\leftarrow$ i
      hijos[numHijos].etapa $\leftarrow$ newEtapa
      hijos[numHijos].Cinf $\leftarrow$ calcularCotaInf(hijos[numHijos], destino)
      hijos[numHijos].visitadas $\leftarrow$ padre.visitadas
      hijos[numHijos].visitadas[i] $\leftarrow$ cierto
    fin si
  fin desde
fin proc

Solución Ejercicio 2 (Cont.)

```
proc añadirHijosLNV(hijos[1..N], numhijos, mejorSol[1..n], Lnv, Cota, destino)

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

Ejercicio 3. Ramificación y Poda

[ITIS/ITIG, Junio 2006] En un tablero de ajedrez de dimensiones $N \times N$ consideramos el siguiente juego. En el tablero hay colocados peones blancos y negros. Partiendo de una posición inicial y realizando movimientos válidos deseamos saltar todos los peones blancos de acuerdo con las siguientes reglas:

- Sólo se permiten movimientos en cruz.
- Tipos de movimientos posibles:
  - Movimiento a una casilla vacía. Coste en longitud: 1.
  - Salto de un peón blanco. Coste en longitud: 2.
  - No se pueden saltar peones negros.

Diseñad un algoritmo de ramificación y poda que determine, dada una posición inicial y un número máximo de movimientos, el mínimo número de movimientos necesario para saltar todos los peones blancos. Es preciso detallar lo siguiente:

1. El árbol de búsqueda utilizado en el algoritmo (1 punto).
2. El algoritmo completo (3 puntos).
3. El algoritmo de la función de cota utilizada(1 puntos).

Solución Ejercicio 3

- Como vimos en el tema de *backtracking*, suponemos que realizamos una secuencia de movimientos con un solo peón de acuerdo a los movimientos posibles indicados, y que los demás peones (blancos o negros) no se mueven.
- Los datos de entrada que se necesitan son los siguientes:
  - Disposición de los peones en el tablero: matriz T[1..n,1..n].
  - Coordenadas de la posición de inicio: Xini, Yini.
- A diferencia de la solución planteada en el tema anterior, debemos encontrar la solución con el mínimo número de movimientos.
- El algoritmo debe proporcionar el número mínimo de movimientos. No es necesario proporcionar el recorrido.

Solución Ejercicio 3 (Cont.)

- Para representar la solución parcial, podemos utilizar una matriz que represente el tablero, D[1..n,1..n], con el siguiente contenido:
    - 0 si la casilla está vacía.
    - 1 si la casilla contiene un peón negro.
    - 2 si la casilla contiene un peón blanco que no se ha saltado todavía.
    - 3 si la casilla contiene un peón blanco que ya se ha saltado.
- Inicialmente, el tablero contiene valores entre 0 y 2.
- Para comprobar si se han saltado todos los peones blancos, se puede utilizar un contador de peones blancos pendientes de saltar.

Solución Ejercicio 3 (Cont.)

- En algunos casos es necesario saltar más de una vez uno de los peones blancos, pero solamente deberá tenerse en cuenta como salto de peón blanco la primera vez.
- También puede haber casos en los que sea necesario pasar varias veces por la misma casilla vacía.
- En algunos casos no es posible llegar a una solución (cuando hay peones blancos en las esquinas, o cuando se requieren más movimientos del máximo): no es posible utilizar cota superior.
- Por tanto, **actualizaremos `cota` solamente con valores de soluciones.**

Solución Ejercicio 3 (Cont.)

- Una posible cota inferior consiste en tomar:
  - la distancia recorrida hasta el momento,
  - más la distancia de Manhattan al peón blanco más lejano pendiente de saltar,
  - menos el número de peones pendientes menos uno.

n		b		b		b	

Solución Ejercicio 3 (Cont.)

- La información necesaria en cada nodo es la siguiente:

Solución actual
Nivel actual
Posición actual
Peones pendientes de saltar
Cota inferior

Solución Ejercicio 3 (Cont.)

```
proc RyP_peones(T[1..N,1..N], Xini, Yini, M, Cota)
   crear Lnv //montículo ordenado por cota inferior
   nodoRaiz(raiz, T, Xini, Yini)
   Cota ← ∞
   introducir(Lnv, raiz)
   mientras not(vacia(Lnv)) hacer
      sacar(Lnv,x)
      si x.Cinf < Cota entonces
         generarHijos(x, hijos, numhijos, T)
         añadirHijosLNV(hijos, numhijos, Lnv, Cota, M)
      si no
         vaciar(Lnv)
      fin si
   fin mientras
fin proc
```

Solución Ejercicio 3 (Cont.)

proc nodoRaiz(raiz,T[1..N,1..N],Xini,Yini)
  crear raiz
  raiz.sol $\leftarrow$ T
  raiz.etapa $\leftarrow$ 0
  raiz.X $\leftarrow$ Xini
  raiz.Y $\leftarrow$ Yini
  raiz.peonesPtes $\leftarrow$ 0
  desde i $\leftarrow$ 1 **hasta** n **hacer**
    desde j $\leftarrow$ 1 **hasta** n **hacer**
      si T[i,j] $=$ 2 **entonces** raiz.peonesPtes $\leftarrow$ raiz.peonesPtes$+$1
    fin desde
  fin desde
  raiz.Cinf $\leftarrow$ calcularCotaInf(raiz)
fin proc

Solución Ejercicio 3 (Cont.)

fun calcularCotaInf(nodo)
  max $\leftarrow$ 0
  desde i $\leftarrow$ 1 **hasta** n **hacer**
    desde j $\leftarrow$ 1 **hasta** n **hacer**
      si nodo.sol[i,j] $= 2 \wedge$ max $<$ |nodo.X-i|+|nodo.Y-j| **entonces**
        max $\leftarrow$ |nodo.X-i|+|nodo.Y-j|
      fin si
    fin desde
  fin desde
  devolver nodo.etapa + |max - (nodo.peonesPtes-1)|
fin fun

Solución Ejercicio 3 (Cont.)

```
dX = (1, 0, -1, 0) ; dY = (0, 1, 0, -1) //constantes globales
proc generarHijos(padre, hijos[1..N], numHijos, T[1..N,1..N])
   numHijos ← 0 ; newEtapa ← padre.etapa +1
   desde i = 1  hasta 4 hacer
      X ← padre.X + dX[i] ; Y ← padre.Y + dY[i]
      si aceptable(padre.sol,X,Y,i,salto) entonces
         numHijos ← numHijos+1
         hijos[numHijos].sol ← padre.sol
         hijos[numHijos].peonesPtes ← padre.peonesPtes
         si salto entonces
            si hijos[numHijos].sol[X,Y] = 2 entonces
               hijos[numHijos].peonesPtes ← padre.peonesPtes-1
               hijos[numHijos].sol[X,Y] ← 3
            fin si
            X ← X+dX[i] ; Y ← Y+dY[i] //avanza otra casilla
         fin si
         hijos[numHijos].X ← X
         hijos[numHijos].Y ← Y
         hijos[numHijos].etapa ← newEtapa
         hijos[numHijos].Cinf ← calcularCotaInf(hijos[numHijos])
      fin si
   fin desde
fin proc
```

Solución ejercicio 3 (cont.)

- La función `aceptable` es similar a la vista en *backtracking*: comprueba que se puede realizar el movimiento `V` desde la posición `X,Y`:

fun aceptable(D[1..n,1..n],X,Y,v,salto)
  salto $\leftarrow$ falso
  si $1 \leq X \leq n \wedge 1 \leq Y \leq n$ **entonces**
    si D[X,Y] $= 0$ **entonces devolver** cierto
    si no si D[X,Y] $= 1$ **entonces devolver** falso
    si no si $1 \leq X+dX[v] \leq n \wedge 1 \leq Y+dY[v] \leq n \wedge D[X+dX[v], Y+dY[v]] = 0$
    entonces
      salto $\leftarrow$ cierto ; **devolver** cierto
    si no
      devolver falso
    fin si
  si no
    devolver falso
  fin si
fin fun

Solución Ejercicio 3 (Cont.)

```
proc añadirHijosLNV(hijos[1..N], numhijos, Lnv, Cota, M)
   desde i ← 1  hasta numhijos hacer
      si hijos[i].Cinf < Cota ∧ hijos[i].etapa ≤ M entonces
         si hijos[i].peonesPtes = 0 entonces
            Cota ← hijos[i].etapa //nuevo valor de cota
         si no
            introducir(Lnv,hijos[i],hijos[i].Cinf)
         fin si
      fin si
   fin desde
fin proc
```

Ejercicio 4. Ramificación y Poda

[ITIS/ITIG, Sep 2006] Dado el mapa de un país, deseamos colorearlo de tal manera que dos regiones adyacentes, con frontera común, no tengan el mismo color. Diseñad un procedimiento, mediante la metodología de Ramificación y Poda, que determine el mínimo número de colores necesario para colorear el mapa, detallando lo siguiente:

- La declaración de tipos y/o variables para representar la información del problema (0,5 puntos).
- El arbol de búsqueda (0,5 puntos).
- El código del procedimiento (2 puntos).
- La función cota (1 punto).

Solución Ejercicio 4

- La solución se puede representar como una tupla $\langle x_1, x_2, \ldots x_n \rangle$ donde x_i es el color del vértice i
- Si se dispone de m colores, en la etapa k, el algoritmo asigna un color $c \in \{1, \ldots m\}$ al vértice k.
- *Restricciones explícitas*: $x_i \in \{1, \ldots, m\}$
- *Restricciones implícitas*: Vértices adyacentes no pueden ser del mismo color
- El objetivo es buscar una forma de colorear el grafo utilizando solo m colores

Solución Ejercicio 4 (Cont.)

- La estructura de cada nodo puede ser:

Solución actual
Nivel actual
Número de Colores utilizados
Cota inferior (mínimo número de colores a utilizar)
Cota Superor (número máximo de colores necesarios)

- La cota inferior de cada nodo se corresponde con el número de colores utilizados hasta el momento
- La cota superior de cada nodo: podemos suponer que ya se han coloreado K vértices. Una cota superior se puede obtener suponiendo que los vértices aún no tratados van a colorearse con colores diferentes a los ya utilizados. Por supuesto, dicha cota superior no puede superar el número máximo de colores disponibles m. Así, la cota superior para un nodo en la etapa K, se obtiene como suma de:
  - número de colores utilizados para pintar los k-1 vértices
  - n-k

Solución Ejercicio 4 (Cont.)

```
fun calcularCotaSup(nodo,G[1..N,1..N], M)
  cotaSuperior ← nodo.colores + (N - nodo.etapa)
  si cotaSuperior > M entonces
    cotaSuperior ← M
  fin si
  devolver cotaSuperior
fin fun
```

- En el caso más general, en este problema no podríamos utilizar la cota superior para podar nodos
  - puede haber subárboles sin ninguna solución
  - Pero al utilizar un valor máximo M (con M=4 siempre es posible colorear un mapa), se garantiza que si la cota superior es menor a M, entonces sí hay solución en ese subárbol.
- El valor de `Cota` se calcula como el mínimo de las cotas superiores de los nodos generados hasta el momento y el número de colores utilizados en la mejor solución encontrada.
- Un nodo es podado si cumple:

  nodo.CInf > Cota (Debe ser **estrictamente mayor**)

Solución Ejercicio 4 (Cont.)

proc nodoRaiz(raiz,G[1..N,1..N],M)
  crear raiz
  raiz.sol[i] $\leftarrow$ 0
  raiz.etapa $\leftarrow$ 0
  raiz.numColores $\leftarrow$ 0
  raiz.Cinf $\leftarrow$ 0
  raiz.Csup $\leftarrow$ calcularCotaSup(raiz,G,M)
fin proc

Solución Ejercicio 4 (Cont.)

- Cada nodo tiene un máximo de M hijos que se corresponden con los distintos colores del siguiente vértice del grafo
- Los vértices de la etapa N no tienen hijos.

```
proc generarHijos(padre, hijos[1..N], numHijos, G[1..N,1..N], M)
  numHijos ← 0 ; newEtapa ← padre.etapa +1
  desde i = 1  hasta M hacer
    si  aceptable(padre, i, G)  entonces
      numHijos ← numHijos +1 ;    crear hijos[numHijos]
      hijos[numHijos].sol ← padre.sol ;    hijos[numHijos].sol[newEtapa] ← i
      hijos[numHijos].etapa ← newEtapa
      hijos[numHijos].numColores ← padre.numColores
      si not(colorUtilizado(padre, i)  entonces
        hijos[numHijos].numColores ← hijos[numHijos].numColores + 1
      fin si
      hijos[numHijos].Cinf ← hijos[numHijos].numColores
      hijos[numHijos].Csup ← calcularCotaSup(hijos[numHijos],G,M)
    fin si
  fin desde
fin proc
```

Solución Ejercicio 4 (Cont.)

```
fun colorUtilizado(nodo, color)
   utilizado ← falso
   i ← 1
   mientras (i ≤ nodo.etapa) ∧ ¬utilizado hacer
      si  nodo.sol[i] = color  entonces utilizado ← cierto
      i ← i+1
   fin mientras
   devolver utilizado
fin fun
```

Solución Ejercicio 4 (Cont.)

```
fun aceptable(nodo, color, G)
   valido ← cierto
   i ← 1
   mientras (i ≤ nodo.etapa) ∧ valido hacer
      si  (nodo.sol[i] = color) ∧ G[i,etapa+1] entonces
         // vértices adyacentes de igual color
         valido ← falso
      fin si
      i ← i +1
   fin mientras
   devolver valido
fin fun
```

Solución Ejercicio 4 (Cont.)

```
proc RyP_Colorear(G[1..N,1..N],mejorSol[1..N], Cota, M)
   crear Lnv
   nodoRaiz(raiz, G, M)
   Cota ← M
   introducir(Lnv, raiz)
   mientras not(vacia(Lnv)) hacer
      sacar(Lnv,x)
      si x.Cinf ≤ Cota entonces
         generarHijos(x, hijos, numHijos, G, M)
         añadirHijosLNV(hijos, numhijos, mejorSol, Lnv, Cota)
      fin si
   fin mientras
fin proc
```

Solución Ejercicio 4 (Cont.)

```
proc añadirHijosLNV(hijos[1..N], numhijos, mejorSol[1..n], Lnv, Cota)
  desde i ← 1  hasta numhijos hacer
    si hijos[i].Cinf ≤ Cota entonces
      si hijos[i].etapa] = N entonces
        mejorSol ← hijos[i].sol
        Cota ← hijos[i].numColores
      si no
        Cota ← min{ Cota, hijos[i].Csup }
        introducir(Lnv,hijos[i])
      fin si
    fin si
  fin desde
fin proc
```

Ejercicio 5. Ramificación y Poda

[ITIS/ITIG, Sep 2005] La compañía discográfica NPI quiere sacar un LP con los grandes éxitos de uno de sus artistas principales. Para ello dispone de M canciones a repartir entre las dos caras del LP. Se conocen tanto el tiempo de cada canción como el tiempo de música que puede almacenar cada cara del LP. Se pide:
Encontrar mediante un algoritmo de ramificación y poda que utilice un montículo la composición de canciones del disco de tal forma que maximice el número de canciones. Especificar el árbol de búsqueda. Suponed conocida la cota. (3 puntos)

Solución ejercicio 5.

- La solución se puede plantear como una tupla $\langle X_1, \ldots, X_M \rangle$ en la que X_i indica si la canción i-ésima aparece en la cara 1, en la cara 2, o bien no se incluye en el disco
- *Restricciones explícitas*: cada elemento de la solución debe tener un valor entre los siguientes: 0 (no se incluye en el disco), 1 y 2
- *Restricciones implícitas*: No se puede incluir una canción en una cara del disco si se supera capacidad total de esa cara
- Para comprobar la restricción implícita es necesario mantener en cada etapa la capacidad disponible de cada una de las caras
- Además, es necesario mantener el número de canciones seleccionadas en alguna de las caras para obtener la solución máxima

Solución Ejercicio 5 (Cont.)

- La estructura de cada nodo puede ser:

Solución actual
Nivel actual
Número de canciones incluidas
Capacidad disponible en cada cara del LP
Cota inferior (mínimo de canciones en el LP)
Cota superior (máximo de canciones en el LP)

- Una cota inferior de cada nodo puede ser el número de canciones incluidas en el LP hasta el momento
- Para el cálculo de la cota superior, podemos suponer que las canciones que aún no se han analizado van a ser incluidas en el LP. Así una cota superior del nodo se obtiene como la suma de:
  - número de canciones incluidas en el LP hasta el momento
  - M - nodo.etapa

Solución Ejercicio 5 (Cont.)

```
proc nodoRaiz(raiz, CapA, CapB,M)
   crear raiz
   desde i← 1 hasta M hacer raiz.sol[i] ← 0
   raiz.etapa ← 0
   raiz.numCanciones ← 0 // cota inferior
   raiz.Cap[1] ← CapA
   raiz.Cap[2] ← CapB
   raiz.Csup ← calcularCotaSup(raiz,M)
fin proc
```

Solución Ejercicio 5 (Cont.)

fun calcularCotaSup(nodo,M)
  cotaSuperior $\leftarrow$ nodo.numCanciones + (M - nodo.etapa)
  devolver cotaSuperior
fin fun

- La variable de poda Cota se actualiza como el máximo de las cotas inferiores y el número de canciones de la mejor solución encontrada hasta el momento. Se inicializa a 0.
- Un nodo es podado si cumple:

$$\text{nodo.Csup} < \text{Cota}$$

- Cada nodo tiene como mucho tres hijos, correspondientes a no incluir la siguiente canción en el LP, a incluirla en la cara A o incluirla en la cara B respectivamente.

Solución Ejercicio 5 (Cont.)

```
proc generarHijos(padre, hijos[1..3], numHijos, C[1..M])
  numHijos ← 0 ; newEtapa ← padre.etapa +1
  desde i = 0  hasta 2 hacer
    si (i = 0) OR (padre.Cap[i] ≥ C[newEtapa]) entonces
      numHijos ← numHijos +1 ;    crear hijos[numHijos]
      hijos[numHijos].sol ← padre.sol
      hijos[numHijos].sol[newEtapa] ← i
      hijos[numHijos].etapa ← newEtapa
      hijos[numHijos].Cap ← padre.Cap
      si i = 0 entonces
        hijos[numHijos].numCanciones ← padre.numCanciones // cota inferior
      si no si padre.Cap[i] ≥ C[newEtapa] entonces
        hijos[numHijos].Cap[i] ← padre.Cap[i] - C[newEtapa]
        hijos[numHijos].numCanciones ← padre.numCanciones +1 // cota inferior
      fin si
      hijos[numHijos].Csup ← calcularCotaSup(hijos[numHijos])
    fin si
  fin desde
fin proc
```

Solución Ejercicio 5 (Cont.)

```
proc RyP_Canciones(C[1..M],mejorSol[1..M], Cota, CapA, CapB)
  crear Lnv // montículo
  nodoRaiz(raiz, CapA, CapB, M)
  Cota ← 0
  introducir(Lnv, raiz)
  mientras not(vacia(Lnv)) hacer
    sacar(Lnv,x)
    si x.Csup ≥ Cota entonces
      generarHijos(x, hijos, numhijos, C)
      añadirHijosLNV(hijos, numhijos, mejorSol, Lnv, Cota)
    fin si
  fin mientras
fin proc
```

Solución Ejercicio 5 (Cont.)

```
proc añadirHijosLNV(hijos[1..3], numhijos, mejorSol[1..M], Lnv, Cota)
  desde i ← 1  hasta numhijos hacer
    si hijos[i].etapa = M AND hijos[i].numCanciones ≥ Cota entonces
        mejorSol ← hijos[i].sol
        Cota ← hijos[i].numCanciones
    si no si hijos[i].Csup ≥ Cota entonces
      Cota ← max { hijos[i].numCanciones, Cota}
      introducir(Lnv,hijos[i],hijos[i].Csup)
      //montículo ordenado por la cota superior
    fin si
  fin desde
fin proc
```

Ejercicio 6. Mochila [0,1] con múltiples elementos

- Se dispone de n tipos de objetos y una mochila de capacidad $C > 0$.
- Para cada tipo de objeto conocemos los siguientes datos:
  - El número de objetos del tipo i es m_i
  - El peso del objeto i es $w_i > 0$
  - La inclusión de un objeto de tipo 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 $\frac{beneficio}{peso}$
- Esta es una variación del problema de la mochila [0,1] en la que se introduce la restricción de que existe un número limitado de cada tipo de objeto.

Ejercicio 6. Mochila [0,1] con múltiples elementos (cont.)

- En este caso, la solución viene representada como una tupla $\{x_1, x_2, \ldots x_n\}$, $x_i \in [0, m_i]$, indicando el número de objetos de cada tipo que se introducen en la mochila
  - Si $x_i = 0$ el objeto i no se introduce en la mochila
  - Si $x_i = m$ se introducen m unidades del objeto i en la mochila
- **Restricciones**: $\sum_{i=1}^{n} x_i \cdot w_i \leq C$
- El **objetivo** es maximizar la función $\sum_{i=1}^{n} x_i \cdot b_i$

Ejercicio 6. Mochila [0,1] con múltiples elementos (cont.)

- La estructura de cada nodo no difiere de la que ya se vió en el problema de la mochila [0,1]. En cada nodo vamos a almacenar información sobre la solución alcanzada hasta ese momento.

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

- Cada nivel del árbol indica el procesamiento de un tipo de objeto.
- En este caso, cada nodo va a poder tener más de dos hijos. Así, cada arista de un nodo en la etapa k se corresponde con la inclusión del siguiente elemento $(k+1)$ en m unidades, siendo $m \in (0, m_{k+1})$.
- Además, un nodo tendrá tantos hijos como le permita la capacidad de la mochila.

Ejercicio 6. Mochila [0,1] con múltiples elementos (cont.)

- 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

Ejercicio 6. Mochila [0,1] con múltiples elementos (cont.)

- (Igual que en el problema de la mochila [0,1])
- Cálculo de las cotas para un nodo *i* en la etapa *k*
  - 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 rellenamos 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.

Ejercicio 6. Mochila [0,1] con múltiples elementos (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
```

Ejercicio 6. Mochila [0,1] con múltiples elementos (cont.)

- La estrategia de ramificación está a cargo del procedimiento `generarHijos`.
- Cada nodo del árbol del nivel i va a tener como mucho m_i hijos, dependiendo si incluimos el siguiente elemento en 1 unidad, 2 unidades, 3 unidades, ..., m_i unidades o por el contrario no lo incluimos en la mochila.
- Solo vamos a generar aquellos nodos que sean válidos: si el objeto correspondiente cabe en la mochila.

Ejercicio 6. Mochila [0,1] con múltiples elementos (cont.)

```
proc generarHijos(padre, hijos[1..m], numHijos, Capacidad, elem[1..n])
  numHijos ← 0, newEtapa ← padre.etapa +1
  si padre.etapa < n entonces
    j ← 0
    mientras (j ≤ elem[newEtapa].unidades) AND (padre.pesoAcumulado +
    (elem[newEtapa].peso)*j ≤ Capacidad) hacer
      numHijos ← numHijos + 1
      crear hijos[numhijos]
      hijos[numHijos].etapa ← newEtapa
      hijos[numHijos].sol ← padre.sol ; hijos[numHijos].sol[newEtapa] ← j
      hijos[numHijos].pesoAcumulado ← padre.pesoAcumulado +
                                         elem[newEtapa].peso * j
      hijos[numHijos].benefAcumulado ← padre.benefAcumulado +
                                         elem[newEtapa].beneficio * j
      hijos[numHijos].cinf ← hijos[numHijos].benefAcumulado
      hijos[numHijos].csup ← calcularCotaSup(hijos[numHijos], Capacidad, elem)
      j ← j+1
    fin mientras
  fin si
fin proc
```

Ejercicio 6. Mochila [0,1] con múltiples elementos (cont.)

- En cuanto a la estrategia de poda, es necesario actualizar la variable de poda Cota cada 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

$$i.csup < Cota$$

Ejercicio 6. Mochila [0,1] con múltiples elementos (cont.)

```
proc RyP_Mochila01_ME(mejorSol[1..n], Capacidad, elem[1..n])
  nodoRaiz(raiz,Capacidad,elem) ; Cota ← 0
  introducir(lnv, raiz) ; valorPoda(raiz, Cota)
  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  hijos[i].etapa = n entonces // es solución
            mejorSol ← hijos[i].sol ; Cota ← hijos[i].benefAcumulado
          si no
            valorPoda(hijos[i], Cota) ; introducir(Lnv,hijos[i])
          fin si
        fin si
      fin desde
    fin si
  fin mientras
fin proc
```

Ejercicio 7. Ramificación y Poda

[ITIS/ITIG, febrero 2009]

Una empresa de autobuses realiza recorridos turísticos por los N pueblos de una región. Algunos de estos pueblos tienen un monumento de interés turístico mientras otros no. Se quieren realizar recorridos turísticos que no pasen dos veces por el mismo pueblo, pero que al final del viaje regresen al punto de partida. Son conocidas las distancias entre pueblos así como los pueblos que tienen monumento.

Diseña un algoritmo de ramificación y poda que calcule la ruta que permita visitar **todos los monumentos** minimizando el número de kilómetros.

1. (0,25 puntos) El contenido de los nodos del árbol de expansión.
2. (1,75 puntos) El código del procedimiento.
3. (0,5 puntos) La función o funciones cota.

Solución Ejercicio 7

- Podemos representar el problema mediante una matriz de adyacencia, con tantas filas y columnas como pueblos.
- Para representar los pueblos que tienen monumento de interés turístico, utilizaremos un vector de valores booleanos.
- Como en el problema del viajante, podemos elegir un pueblo como origen de los caminos que busquemos. En la solución propuesta el pueblo de origen será un dato del problema.
- Los recorridos son **caminos circulares** **en los que no es necesario que todos los pueblos del camino tengan monumento ni que se pase por todos los pueblos**.
- Sin embargo, sí deben estar **todos los pueblos con monumento**.
- Al ser un problema de minimización, se puede podar utilizando cotas inferiores
- ¿Pueden utilizarse cotas superiores para obtener el valor de cota para podar?

Solución Ejercicio 7 (cont.)

```
proc RyP_bus(D[1..N,1..N],M[1..n],origen,mejorSol[1..N], Cota)
  crear Lnv // montículo ordenado por la cota inferior
  nodoRaiz(raiz,origen,D,M)
  maxMon ← 0
  desde i ← 1 hasta n hacer
    si M[i] entonces maxMon ← maxMon + 1
  fin desde
  Cota ← ∞
  introducir(Lnv, raiz)
  mientras not(vacia(Lnv)) hacer
    sacar(Lnv,x)
    si x.Cinf < Cota entonces
      generarHijos(x, hijos, numhijos, D, M)
      añadirHijosLNV(hijos, numhijos, mejorSol, Lnv, Cota, origen, maxMon)
    si no
      vaciar(Lnv)
    fin si
  fin mientras
fin proc
```

Solución Ejercicio 7 (cont.)

proc generarHijos(padre, hijos[1..N], numHijos, D[1..n,1..n],M[1..n])
  numHijos ← 0 ; newEtapa ← padre.etapa +1
  si padre.etapa < n **entonces**
    desde i = 1 **hasta** N **hacer**
      si ¬padre.visitado[i] ∧ D[padre.sol[padre.etapa],i]< ∞ **entonces**
        numHijos ← numHijos +1
        crear hijos[numHijos]
        hijos[numHijos].sol ← padre.sol
        hijos[numHijos].sol[newEtapa] ← i
        hijos[numHijos].visitado ← padre.visitado
        hijos[numHijos].visitado[i] ← *cierto*
        hijos[numHijos].etapa ← newEtapa
        hijos[numHijos].longitud ← padre.longitud + D[padre.sol[padre.etapa],i]
        si M[i] **entonces** hijos[numHijos].numMon ← padre.numMon + 1
        hijos[numHijos].Cinf ← calcularCotaInf(hijos[numHijos],D)
      fin si
    fin desde
  fin si
fin proc

Solución Ejercicio 7 (cont.)

```
proc añadirHijosLNV(hijos[1..N], numhijos, mejorSol[1..n], Lnv, Cota, origen
maxMon)
  desde i ← 1  hasta numhijos hacer
    si hijos[i].Cinf < Cota entonces
      retorno ← D[hijos[i].sol[hijos[i].etapa],origen]
      si hijos[i].numMon = maxMon ∧ retorno < ∞ entonces
        longCamino ← hijos[i].longitud + retorno
        si longCamino < Cota entonces
          mejorSol ← hijos[i].sol
          Cota ← longCamino //nuevo valor de la cota inferior
        fin si
      fin si
      introducir(Lnv,hijos[i],hijos[i].Cinf)
      //montículo ordenado por la cota inferior
    fin si
  fin desde
fin proc
```

Solución Ejercicio 7 (cont.)

```
proc nodoRaiz(raiz,origen,D[1..N,1..N],M[1..N])
   crear raiz
   desde i ← 1 hasta n hacer
      raiz.sol[i] ← 0
      raiz.visitado[i] ← falso
   fin desde
   raiz.sol[1] ← origen
   raiz.visitado[origen] ← cierto
   raiz.etapa ← 1
   raiz.longitud ← 0
   si M[origen] entonces raiz.numMon ← 1 si no raiz.numMon ← 0
   raiz.Cinf ← calcularCotaInf(raiz,D,M)
fin proc
```

Solución Ejercicio 7 (cont.)

- Para calcular la cota inferior podemos utilizar un esquema parecido al del problema del viajante
- Para llegar a una solución del problema tenemos que pasar por todos los pueblos con monumento
- Por tanto, una cota inferior puede ser la suma de:
  - la longitud del camino recorrido hasta el momento,
  - la suma de las aristas de longitud mínima que **llegan** a los pueblos con monumento que todavía no se han visitado y que procedan de un pueblo no visitado todavía (o bien el pueblo actual)

Solución Ejercicio 7 (cont.)

```
fun calcularCotaInf(nodo,D[1..N,1..N],M[1..N])
   cotaInferior ← nodo.longitud
   minCol ←∞
   desde i ← 1 hasta N hacer
      si ¬nodo.visitado[i] ∧ M[i] entonces
         minCol ←∞
         desde j ← 1 hasta N hacer
            si i≠j ∧ (¬nodo.visitado[j] ∨ j = nodo.sol[nodo.etapa]) entonces
               minCol ← mín{minCol, D[j,i]}
            fin si
         fin desde
         cotaInferior ← cotaInferior+ minCol
      fin si
   fin desde
   devolver cotaInferior
fin fun
```

Ejercicio 8. Ramificación y Poda

[II, septiembre 2009]
Se dispone de un grafo ponderado, conexo y no dirigido $G = \langle V, A \rangle$, y un subconjunto $T \subseteq V$ de vértices de G.
Un **árbol de Steiner** de T en G es un subconjunto de aristas de G que no forma ciclos y conecta todos los vértices que están en T. Un árbol de Steiner debe contener todos los vértices de T, pero puede incluir otros vértices de V que no están en T.
Dado un grafo $G = \langle V, A \rangle$ y un conjunto $T \subseteq V$, diseña un algoritmo de *ramificación y poda* que obtenga el árbol de Steiner de T de coste mínimo (la suma de los pesos de las aristas del árbol debe ser mínima), incluyendo lo siguiente:

a) (3,5 puntos) Pseudocódigo del algoritmo, incluyendo una descripción de las estructuras de datos necesarias, el árbol de expansión y el contenido de cada nodo.
b) (0,5 puntos) Función o funciones cota necesarias.

Nota: es conveniente mantener en cada nodo del árbol de expansión los vértices del grafo que se han considerado hasta ese momento.

Solución Ejercicio 8

- Como parámetros de entrada tenemos un grafo G y un conjunto de nodos T.
- La solución será un árbol A representado por un conjunto de nodos (al que llamaremos nodos *visitados*) y un conjunto de aristas (al que llamaremos *Sol*).
- El grafo de entrada se puede representar mediante un **vector de aristas**. Cada elemento de este vector contiene tres atributos para representar el origen, destino y coste de la arista.
- El conjunto de nodos que deben incluirse en el árbol de Steiner (A) se puede representar mediante un vector booleano de N elementos, indicando si el vértice i-ésimo pertenece o no a T.
- El conjunto de aristas del árbol de Steiner será un vector de (como máximo) $N - 1$ aristas. Cada elemento de este vector contiene el número de arista en el vector de aristas.
- ¿Se puede utilizar otra representación para el conjunto de aristas solución?

Solución Ejercicio 8 (cont.)

- El nodo del árbol de expansión tiene la siguiente estructura:

Solución parcial: Vector de aristas
Etapa
Coste real acumulado
Cota inferior
Vértices visitados
Núm. nodos de T pendientes

Solución Ejercicio 8 (cont.)

- En cada etapa, el algoritmo toma una arista de entre las posibles.
- Si el grafo G tiene M aristas, el nodo raiz tendrá tantos hijos como aristas tenga el grafo G, es decir M.
- Dado un nodo en la etapa k, el procedimiento de generación de hijos ha de tener en cuenta la última arista seleccionada D[k].
- Este nodo tendrá como hijos las aristas de la forma D[k+1], ..., D[M] que sean factibles (para evitar repetir soluciones, ya que el conjunto de aristas $\{(1,2),(1,4)\}$ es igual al conjunto $\{(1,4),(1,2)\}$).
- El árbol de expansión tendrá como máximo N-1 niveles (si todos los vértices forman parte del árbol). Las soluciones pueden estar en cualquier nivel del árbol, no solamente en el nivel N-1.

Solución Ejercicio 8 (cont.)

- Necesitamos comprobar si un conjunto de aristas y el conjunto de vértices asociado a esas aristas forman un árbol (A es un árbol si el número de nodos es igual al número de aristas + 1 y no se forman ciclos)
- No todos los nodos llevan a una solución (puede haber nodos hoja del árbol que tengan como solución conjuntos de aristas del grafo que no sean conexas, y por tanto no sean un árbol), por lo que **se utiliza un esquema de poda basado en una sola cota.**
- La cota inferior del nodo *i* puede ser la suma de:
  - El coste real acumulado (coste de las aristas consideradas hasta ese momento).
  - La suma de las aristas de menor peso que salen de los vértices de T no visitados todavía.

Solución Ejercicio 8 (cont.)

```
fun calcularCotaInf(nodo,D[1..M],T[1..N])
   cotaInferior ← nodo.coste
   desde i ← 1 hasta N hacer
      si ¬nodo.visitado[i] ∧ T[i] entonces
         minFila ←∞
         desde j ← 1 hasta M hacer
            si i = D[j].origen ∨ i = D[j].destino entonces
               minFila ← mín{minFila, D[j]}
            fin si
         fin desde
         cotaInferior ← cotaInferior + minFila
      fin si
   fin desde
   devolver cotaInferior
fin fun
```

Solución Ejercicio 8 (cont.)

```
proc steiner(D[1..M],T[1..N],sol[1..N-1])
   crear Lnv // montículo ordenado por cota inferior
   nodoRaiz(raiz,D,T)
   Cota ← ∞
   introducir(Lnv, raiz)
   mientras ¬vacia(Lnv) hacer
      sacar(Lnv,x)
      si x.Cinf < Cota entonces
         generarHijos(x, hijos, numhijos, D, T)
         añadirHijosLNV(hijos, numhijos, sol, Lnv, Cota)
      si no
         vaciar(Lnv)
      fin si
   fin mientras
fin proc
```

Solución Ejercicio 8 (cont.)

```
proc generarHijos(padre, hijos[1..N], numHijos, D[1..M],T[1..N])
  numHijos ← 0 ; newEtapa ← padre.etapa +1
  desde i ← padre.sol[padre.etapa] + 1  hasta M hacer
    si ¬padre.visitado[D[i].origen] ∨ ¬padre.visitado[D[i].destino] ∨
      ¬formaCiclo(padre,D,i) entonces
      numHijos ← numHijos +1 ; crear hijos[numHijos]
      hijos[numHijos].sol ← padre.sol ; hijos[numHijos].sol[newEtapa] ← i
      si T[D[i].origen] ∧¬padre.visitado[D[i].origen] entonces
        hijos[numHijos].numPendientes ← padre.numPendientes - 1
      fin si
      si T[D[i].destino] ∧¬padre.visitado[D[i].destino] entonces
        hijos[numHijos].numPendientes ← padre.numPendientes - 1
      fin si
      hijos[numHijos].visitado ← padre.visitado
      hijos[numHijos].visitado[D[i].origen] ← cierto
      hijos[numHijos].visitado[D[i].destino] ← cierto
      hijos[numHijos].etapa ← newEtapa
      hijos[numHijos].coste ← padre.coste + D[i].coste
      hijos[numHijos].Cinf ← calcularCotaInf(hijos[numHijos],D,T)
    fin si
  fin desde

fin proc
```

Solución Ejercicio 8 (cont.)

```
fun formaCiclo(nodo,D[1..M],i) //Devuelve cierto si al añadir la arista i se forma un ciclo
  crear v_visitados[1..N] ; crear a_visitadas[1..nodo.etapa]
  desde j ← 1 hasta N hacer v_visitados[j] ← falso
  desde j ← 1 hasta nodo.etapa hacer a_visitadas[j] ← falso
  v_visitados[D[i].origen] ← cierto ; v_visitados[D[i].destino] ← cierto
  hayCiclo ← falso ; hayCambios ← cierto
  mientras ¬hayCiclo ∧ hayCambios hacer
    hayCambios ← falso ; j ← 1
    mientras j ≤ nodo.etapa ∧¬hayCiclo hacer
      si ¬a_visitadas[j] entonces
        origen ← D[nodo.sol[j]].origen ; dest ← D[nodo.sol[j]].destino
        si v_visitados[origen] ∧ v_visitados[dest] entonces
          hayCiclo ← cierto
        si no si v_visitados[origen] ∨ v_visitados[dest] entonces
          v_visitados[origen] ← cierto ; v_visitados[dest] ← cierto
          hayCambios ← cierto ; a_visitadas[j] ← cierto
        fin si
      fin si
      j ← j + 1
    fin mientras
  fin mientras
  devolver hayCiclo
fin fun
```

Solución Ejercicio 8 (cont.)

```
proc añadirHijosLNV(hijos[1..N], numhijos, mejorSol[1..N-1], Lnv, Cota)
  desde i ← 1  hasta numhijos hacer
    si hijos[i].Cinf < Cota entonces
      si hijos[i].numPendientes = 0 ∧ esArbol(hijos[i]) entonces
        mejorSol ← hijos[i].sol ; Cota ← hijos[i].coste //nuevo valor de poda
      si no
        introducir(Lnv,hijos[i],hijos[i].Cinf) //montículo ordenado por la cota inferior
      fin si
    fin si
  fin desde
fin proc

fun esArbol(nodo)
  // Como no tiene ciclos, solo hay que comprobar que núm.vertices = núm.aristas+1
  numVertices ←0 ; numAristas ←nodo.etapa
  desde i ← 1 hasta N hacer
    si nodo.visitados[i] entonces  numVertices ←numVertices + 1
  fin desde
  devolver numVertices = numAristas + 1
fin fun
```

Solución Ejercicio 8 (cont.)

```
proc nodoRaiz(raiz,D[1..M],T[1..N], Min)
   crear raiz
   numPtes ← 0
   desde i ← 1 hasta N hacer
      raiz.visitado[i] ← falso
      si T[i] entonces numPtes ← numPtes + 1
   fin desde
   desde i ← 1 hasta N − 1 hacer
      raiz.sol[i] ← 0
   fin desde
   raiz.numPendientes ← numPtes
   raiz.etapa ← 0
   raiz.numVertices ← 0
   raiz.coste ← 0
   raiz.Cinf ← calcularCotaInf(raiz,D,T)
fin proc
```