

Ejercicio 1. *Backtracking*

[junio 2007]

Consideremos la siguiente ecuación:

$$C_n X_n + C_{n-1} X_{n-1} + \dots + C_1 X_1 + C_0 = 0$$
$$0 < X_i < d_i \forall i : 1..n$$

donde C_i y d_i ($C_i, d_i \in \mathbb{R}, \forall i : 1..n$) son conocidos. Diseña un algoritmo de vuelta atrás que muestre *todas* las soluciones sabiendo que $X_i \in \mathbb{Z}, \forall i : 1..n$. Detalla lo siguiente:

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

Solución ejercicio 1. *Backtracking*

- En el planteamiento de este problema mediante la técnica de backtracking, lo habitual es que la solución esté formada por una tupla de elementos $\langle x_1, \dots, x_n \rangle$.
- Cada elemento de la solución corresponde a una decisión con un conjunto de resultados posibles.
- En este problema, podemos utilizar las variables X_i de la ecuación como los elementos de la solución.
- Los valores posibles de cada variable X_i son los enteros contenidos en el intervalo abierto $(0, d_i)$.
- C_0 , C_i y d_i , $i \in \{1, \dots, n\}$, son valores conocidos del problema (datos de entrada)
- Restricciones explícitas: $0 < x_i < d_i$
- Restricciones implícitas: no hay

Solución ejercicio 1. *Backtracking* (cont.)

- Una posible implementación es la siguiente:

```
proc ecuacion(C[0..n],d[1..n],sol[1..n],valActIni,etapa)
  desde v  $\leftarrow$  1 hasta  $\lceil d[etapa] \rceil - 1$  hacer
    sol[etapa]  $\leftarrow$  v
    valAct  $\leftarrow$  valActIni + C[etapa]*v
    si etapa = n entonces
      si valAct = 0 entonces comunicar(sol)
    si no
      ecuacion(C,d,sol,valAct,etapa+1)
    fin si
  fin desde
fin proc
```

Solución ejercicio 1. *Backtracking* (cont.)

- El procedimiento llamador es el siguiente:

```
proc llamador_ecuacion(C[0..n],d[1..n])  
  crear sol[1..n]  
  ecuacion(C,d,sol,C[0],1)  
fin proc
```

- ¿Cómo sería el algoritmo para devolver solamente la primera solución?

```
fun ecuacion1(C[0..n],d[1..n],sol[1..n],valActIni,etapa)  
  desde v ← 1 hasta d[etapa] hacer  
    sol[etapa] ← v  
    valAct ← valActIni + C[etapa]*v  
    si etapa = n entonces  
      devolver (valAct = 0)  
    si no  
      si ecuacion1(C,d,sol,valAct,etapa+1) entonces devolver cierto  
    fin si  
  fin desde  
fin fun
```

Ejercicio 2. *Backtracking*

[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 vuelta atrás que resuelva el problema detallando lo siguiente:

1. El árbol de búsqueda: significado de las arista y de los niveles (0,5 puntos)
2. El código del procedimiento (2 puntos)
3. El programa llamador (0,5 puntos)

Solución ejercicio 2. *Backtracking*

- Este problema ya lo hemos visto en el tema de programación dinámica.
- También se puede resolver por el método voraz mediante el algoritmo de Dijkstra.
- Sin embargo, se pide diseñar un algoritmo de backtracking (aunque será el menos eficiente de los tres).
- La entrada es la matriz de adyacencia (diccionarios) del grafo de traducciones (la matriz será **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).
- El problema es de optimización, por lo que será necesario guardar la mejor solución hasta el momento.

Solución ejercicio 2. *Backtracking* (cont.)

- Los datos de entrada se pueden representar mediante una matriz cuadrada booleana $D[1..n,1..n]$ en la que las filas y las columnas representan los distintos idiomas, y $D[i,j]$ es cierto si existe un diccionario para traducir del idioma i al idioma j .
- La solución se puede representar mediante una tupla $\langle x_1, \dots, x_k \rangle$, donde x_1 es el idioma de origen y x_k el idioma de destino.
- Restricciones explícitas: Cada uno de los componentes de la solución es uno de los idiomas para los que se puede traducir: $1 \leq x_i \leq n$ (para $1 \leq i \leq k$)
- Restricciones implícitas:
 - ▶ Debe existir un diccionario entre dos elementos consecutivos de la solución: $D[x_i, x_{i+1}] = \text{cierto}, \forall i : i = 1, \dots, k - 1$
 - ▶ No deben repetirse elementos en la solución: $x_i \neq x_j, \forall i \neq j$

Solución ejercicio 2. *Backtracking* (cont.)

```
proc trad(D[1..n,1..n],dest,solAct[1..n],numActIni,solOpt[1..n],numOpt,etapa)
  desde v ← 1 hasta n hacer
    solAct[etapa] ← v
    si acceptable(solAct,etapa) ∧ numActIni < numOpt entonces
      numAct ← numActIni+1
      si solAct[etapa] = dest entonces
        si numAct < numOpt entonces numOpt ← numAct ; solOpt ←
          solAct
      si no
        trad(D,dest,solAct,numAct,solOpt,numOpt,etapa+1)
      fin si
    fin si
  fin desde
fin proc
```

Solución ejercicio 2. *Backtracking* (cont.)

- La función aceptable se puede definir como:

```
fun aceptable(solAct[1..n],etapa)
  si  $\neg D[\text{solAct}[\text{etapa}-1],\text{solAct}[\text{etapa}]]$  entonces devolver falso
  si no
    desde  $i \leftarrow 1$  hasta etapa-1 hacer
      si  $\text{solAct}[i] = \text{solAct}[\text{etapa}]$  entonces devolver falso
    fin desde
  devolver cierto
fin si
fin fun
```

- El procedimiento llamador:

```
fun llamador_trad(D[1..n,1..n],origen,dest,sol[1..n])
  crear solAct[1..n]
  solAct[1]  $\leftarrow$  origen
  numTrad  $\leftarrow \infty$ 
  trad(D,dest,solAct,0,sol,numTrad,2)
  devolver numTrad  $< \infty$ 
fin fun
```

Ejercicio 3. *Backtracking*

[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 vuelta atrás que determine si dada una posición inicial y un número máximo de movimientos es posible saltar todos los peones blancos. Es preciso detallar lo siguiente:

- 1 El árbol de búsqueda utilizado en el algoritmo (1 punto).
- 2 La primera llamada (1 punto).
- 3 El algoritmo completo (3 puntos).

Solución ejercicio 3. *Backtracking*

- Suponemos que realizamos una secuencia de movimientos con un solo peón de acuerdo a los movimientos posibles indicados.
- Suponemos que los movimientos del peón que se va a mover son consecutivos: 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: $Xini, Yini$
- Como resultado del algoritmo solo debemos indicar si se pueden saltar los peones blancos o no (no necesitamos proporcionar el camino)
- Por tanto, basta con encontrar la primera solución
- Sin embargo, es necesario mantener la solución parcial para que el algoritmo pueda determinar si se ha llegado al final

Solución ejercicio 3. *Backtracking* (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
- Una vez hecho un movimiento, debe ser posible deshacerlo para plantear otro recorrido
- Para comprobar si se han saltado todos los peones blancos, se puede utilizar un contador de peones blancos pendientes de saltar

Solución ejercicio 3. *Backtracking* (cont.)

$dX = (1, 0, -1, 0)$ //constantes globales

$dY = (0, 1, 0, -1)$ //constantes globales

fun peon($D[1..n,1..n], M, Xini, Yini, etapa, peonesPtesIni$)

si $etapa < M$ **entonces**

desde $v \leftarrow 1$ **hasta** 4 **hacer**

$X \leftarrow Xini + dX[v]$; $Y \leftarrow Yini + dY[v]$

$peonesPtes \leftarrow peonesPtesIni$

si $acceptable(D, etapa, X, Y, v, peonesPtes)$ **entonces**

si $peonesPtes = 0$ **entonces devolver** cierto

si $peon(D, M, X, Y, etapa+1, peonesPtes)$ **entonces devolver** cierto

si no si $peonesPtes \neq peonesPtesIni$ **entonces**

$D[Xini + dX[v], Yini + dY[v]] \leftarrow 2$ //deshace la etapa

fin si

fin si

fin desde

fin si

devolver falso

fin fun

Solución ejercicio 3. *Backtracking* (cont.)

```
fun aceptable(D[1..n,1..n],etapa,X,Y,v,peonesPtes)
  si  $1 \leq X \leq n \wedge 1 \leq Y \leq n$  entonces
    si  $D[X,Y] = 0$  entonces devolver cierto
    si  $D[X,Y] = 1$  entonces devolver falso
    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
        si  $D[X,Y] = 2$  entonces peonesPtes  $\leftarrow$  peonesPtes-1
         $D[X,Y] \leftarrow 3$ 
         $X \leftarrow X+dX[v]$  ;  $Y \leftarrow Y+dY[v]$ 
        devolver cierto
    si no
      devolver falso
  fin si
fin si
devolver falso
fin fun
```

- Esta función modifica los valores de los argumentos siguientes:
 $D[X, Y]$, X , Y y $peonesPtes$

Solución ejercicio 3. *Backtracking* (cont.)

- El procedimiento llamador debe inicializar `peonesPtes` y llamar con los argumentos correctos:

```
fun llamador_peon(D[1..n,1..n],M,X,Y)
  peonesPtes ← 0
  desde i ← 1 hasta n hacer
    desde j ← 1 hasta n hacer
      si D[i,j] = 2 entonces peonesPtes ← peonesPtes+1
    fin desde
  fin desde
  devolver peon(D,M,X,Y,1,peonesPtes)
fin fun
```

- ¿Hay alguna forma de mejorar este algoritmo?

Ejercicio 4. *Backtracking*

[Sep 2005] (4 puntos) 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 vuelta atrás recursivo la composición de canciones del disco de tal forma que maximice el número de canciones.
- Establecer el árbol de búsqueda (especificando el significado de las aristas y el contenido de cada nodo).
- Indicar la primera llamada al procedimiento.

Solución ejercicio 4. *Backtracking*

- Este problema es similar al de la mochila, pero en este caso tenemos **dos** mochilas (cada una de las caras del disco)
- Es un problema de optimización: debemos maximizar el número de canciones en el disco
- La solución se puede plantear como una tupla $\langle X_1, \dots, X_M \rangle$ 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 4. *Backtracking* (cont.)

```
proc disco_lp(D[1..M],cap[1..2],solAct[1..M],numActIni,sol[1..M],num,etapa)
  desde v ← 0 hasta 2 hacer
    si v=0 ∨ cap[v] ≥ D[etapa] entonces
      solAct[etapa] ← v
      si v>0 entonces
        cap[v] ← cap[v] - D[etapa]
        numAct ← numActIni + 1
      fin si
      si etapa = M entonces
        si numAct > num entonces num ← numAct ; sol ← solAct
      si no
        disco_lp(D,cap,solAct,numAct,sol,num,etapa+1)
      fin si
      si v>0 entonces
        cap[v] ← cap[v] + D[etapa] //deshace la etapa
      fin si
    fin si
  fin desde
fin proc
```

Solución ejercicio 4. *Backtracking* (cont.)

- El procedimiento llamador puede ser de la siguiente forma:

```
proc llamador_disco_lp(D[1..M],capacidad,sol[1..M])  
  crear solAct[1..M]  
  crear cap[1..2]  
  cap[1] ← capacidad/2 ; cap[2] ← capacidad/2  
  disco_lp(D,cap,solAct,0,sol,-1,1)  
fin proc
```

Ejercicio 5. *Backtracking*

[Junio 2005] (4 puntos) Nicanor Cienfuegos tienes dos grandes pasiones: su pueblo y los kioskos. Tal es su pasión por su pueblo que conoce perfectamente todas sus calles (con sus longitudes y sus intersecciones entre ellas). Él vive en la intersección de dos calles y quiere ir a casa de un amigo que también vive en la intersección de dos calles. Aunque podría ir directamente prefiere dar una vuelta y ojear los kioskos. Sin embargo, como nunca compra nada desea pasar por cada kiosko como mucho una vez. Nicanor sabe que los kioskos, en ese pueblo, solamente pueden estar en las intersecciones de dos calles y quiere visitarlos todos.

- Establecer las variables y/o tipos de variables que permitan representar la información que conoce Nicanor y el árbol de búsqueda.
- Establecer el camino más corto desde casa de Nicanor a casa del amigo que pase exactamente una vez por cada kiosko mediante un algoritmo de vuelta atrás iterativo.

Solución ejercicio 5. *Backtracking*

- Podemos representar el mapa del pueblo mediante un grafo en el que las intersecciones son los nodos y los fragmentos de calles las aristas (etiquetadas con su longitud)
- Algunos de los nodos tienen kiosko, y otros no. Los nodos que tienen kiosko sólo pueden visitarse una vez. **Suponemos que se puede construir el camino pedido visitando los demás nodos una sola vez.**
- Podemos representar el grafo como una matriz de adyacencia $D[1..n,1..n]$ con las distancias en cada elemento de la matriz (∞ si no hay un fragmento de calle directo entre dos intersecciones)
- A cada intersección le corresponde un número de fila/columna de la matriz
- Los kioskos pueden representarse mediante un vector booleano $K[1..n]$ que contiene *cierto* en $K[i]$ si en la intersección i se encuentra un kiosko
- M es el número total de kioskos
- Es un problema de optimización: deben buscarse **todas** las soluciones

Solución ejercicio 5. *Backtracking* (cont.)

```
proc kioskos(D[1..n,1..n],K[1..n],M,origen,destino,sol[1..n])
  crear solAct[1..n]
  l_sol  $\leftarrow \infty$  ; l_solAct  $\leftarrow 0$  ; numKioskos  $\leftarrow 0$ 
  solAct[1]  $\leftarrow$  origen ; nivel  $\leftarrow 2$  ; fin  $\leftarrow$  falso
  repetir
    si generar(D,nivel,solAct) entonces
      l_solAct  $\leftarrow$  l_solAct + 1
      si K[solAct[nivel]] entonces numKioskos  $\leftarrow$  numKioskos + 1
      si alcanzable(nivel,solAct) entonces
        si esSolucion(nivel,solAct,destino,numKioskos,M) entonces
          si l_solAct < l_sol entonces l_sol  $\leftarrow$  l_solAct ; sol  $\leftarrow$  solAct
          si no nivel  $\leftarrow$  nivel + 1
        fin si
      si K[solAct[nivel]] entonces numKioskos  $\leftarrow$  numKioskos - 1
    si no
      si nivel=2 entonces fin  $\leftarrow$  cierto si no retroceder(nivel,sol)
    fin si
  hasta fin
fin proc
```

Solución ejercicio 5. *Backtracking* (cont.)

```
//generar() también incluye la funcionalidad de hayMasHermanos()
fun generar(D[1..n,1..n],nivel,sol[1..n])
  repetir
    sol[nivel] ← sol[nivel]+1 //importante: hay dependencia entre llamadas a generar()
  hasta sol[nivel] > n ∨ D[sol[nivel-1],sol[nivel]] < ∞
  si sol[nivel] > n entonces devolver falso
  devolver cierto
fin fun

fun alcanzable(nivel,sol,K)
  desde i ← 1 hasta nivel-1 hacer
    si sol[i] = sol[nivel] entonces devolver falso
  fin desde
  devolver cierto
fin fun

fun esSolucion(nivel,sol[1..n],destino,numKioskos,M)
  devolver sol[nivel] = destino ∧ numKioskos = M
fin fun

proc retroceder(nivel,sol[1..n])
  sol[nivel] ← 0 // es necesario para reiniciar las siguientes llamadas a generar()
  nivel ← nivel-1
fin proc
```

Ejercicio 6. *Backtracking*

(basado en [GV00], p. 222).

Recorridos del rey de ajedrez (2)

En un tablero de ajedrez de tamaño $n \times n$ se coloca un rey en una casilla arbitraria (x_0, y_0) . Cada casilla (x, y) del tablero tiene asignado un peso $T(x, y)$, de tal forma que a cada recorrido $R = \langle (x_0, y_0), \dots, (x_k, y_k) \rangle$ se le puede asignar un valor que viene determinado por la siguiente expresión:

$$P(R) = \sum_{i=0}^k i \cdot T(x_i, y_i)$$

El problema consiste en diseñar un algoritmo que proporcione el recorrido de peso mínimo que visite todas las casillas del tablero sin repetir ninguna.

Solución ejercicio 6. *Backtracking*

$dX = [0, 1, 1, 1, 0, -1, -1, -1]$ // vectores globales

$dY = [-1, -1, 0, 1, 1, 1, 0, -1]$

proc rey($T[1..n,1..n]$,solAct[$1..n,1..n$],valActIni,sol[$1..n,1..n$],val,Xin,Yin,etapa)

desde $v \leftarrow 1$ **hasta** 8 **hacer**

$X \leftarrow Xin + dX[v]$

$Y \leftarrow Yin + dY[v]$

si $1 \leq X \leq n \wedge 1 \leq Y \leq n \wedge solAct[X, Y] = 0$ **entonces**

solAct[X,Y] \leftarrow etapa

valAct \leftarrow valActIni + $T[X,Y]*etapa$

si etapa = $n*n$ **entonces**

si valAct < val **entonces** sol \leftarrow solAct ; val \leftarrow valAct

si no si valAct < val **entonces**

rey(T ,solAct,valAct,sol,val,X,Y,etapa+1)

fin si

solAct[X,Y] \leftarrow 0 // deshace la etapa

fin si

fin desde

fin proc

Ejercicio 7. *Backtracking*

(basado en [GV00], p. 226).

El laberinto

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.

Solución ejercicio 7. *Backtracking*

- Por ejemplo, podemos considerar el siguiente laberinto:

	1	2	3	4	5
1	E				
2		∞	∞	∞	
3					
4		∞	∞	∞	∞
5					S

- ¿Cómo se generan los movimientos en el laberinto?
- ¿Cuál es el camino que va a seleccionar el algoritmo?

Solución ejercicio 7. *Backtracking* (cont.)

- Vamos a resolver este algoritmo mediante un algoritmo iterativo que utilice una pila para almacenar los nodos que están pendientes de procesar
- Cada elemento de la pila debe contener los datos necesarios para representar un nodo del árbol de expansión
- En este problema, los datos necesarios para representar un nodo son:
 - `sol` el estado de la solución parcial (el recorrido realizado hasta ahora)
 - `x, y` La última posición visitada
 - `etapa` La etapa del algoritmo
- Estos datos se representan mediante una estructura de datos `nodo`
- Las operaciones de la pila son `insertar()`, `desapilar()`, `vacia()` y `vaciar()`

Solución ejercicio 7. *Backtracking* (cont.)

```
dX = [ 1, 0, -1, 0] ; dY = [ 0, 1, 0, -1] // vectores globales
fun laberinto(L[1..n,1..n],sol[1..n,1..n])
  crear P // P es la pila
  crear u,v ; v.x ← 1 ; v.y ← 1 // u,v variables locales de tipo nodo
  desde i ← 1 hasta n hacer
    desde j ← 1 hasta n hacer v.sol[i,j] ← 0
  fin desde
  v.sol[1,1] ← 1 ; v.etapa ← 1 ; insertar(P,v)
  mientras ¬ vacia(P) hacer
    desapilar(P,v)
    u.sol ← v.sol ; u.etapa ← v.etapa+1
    desde m ← 1 hasta 4 hacer
      u.x ← v.x+dX[m] ; u.y ← v.y+dY[m]
      si 1 ≤ u.x ≤ n ∧ 1 ≤ u.y ≤ n ∧ L[u.x,u.y] = 0 ∧ u.sol[u.x,u.y] = 0 entonces
        u.sol[u.x,u.y] ← u.etapa
        si u.x = n ∧ u.y = n entonces vaciar(P) ; sol ← u.sol ; devolver cierto
        si no insertar(P,u)
        u.sol[u.x,u.y] ← 0 // deshace el movimiento
    fin si
  fin desde
fin mientras
devolver falso
fin fun
```