Algoritmos de vuelta atrás

January 25, 2011

- En muchos casos las técnicas anteriores no son aplicables.
- Los algortimos de vuelta atrás resuelven el problema mediante una búsqueda exhaustiva por el espacio de posibles soluciones hasta encontrar una que satisfaga los criterios exigidos o bien determine que no existe.
- Si el número de posibles soluciones es muy grande, esta técnica puede resultar impracticable.
- Es imprescindible intentar estructurar el espacio de búsqueda trantado de descartar posibles soluciones no satistactorias.
- El espacio de soluciones tiene una estructura de árbol. Es recorrido en profundidad.
- Son típicamente recursivos.

- El árbol es implícito: en un instante solamente existe un subconjunto de nodos.
- Cada nodo representa un estado de cómputo.
- Cada arista representa una decisión:
  - Se le asignan a un conjunto de variables un conjunto de valores.
  - Las condiciones que deben satisfacer estos valores son las *restricciones explícitas*.
  - Las condiciones que deben satisfacer los valores de un cojunto de variables para que constituya una solución se denominan *restricciones implícitas*.
- La eficiencia viene dada por el número del espacio de posibles soluciones. Suele ser exponencial.
- La solución suele tener forma de n-tupla.

Problema

Dados N empleados y N tareas, determina la asignación de tareas de mínimo coste:

- *El coste de realizar una tarea depende del empleado que la realice de acuerdo con los valores de la tabla:* $T[1..N, 1..N] \to T[I, J]$*: coste de realización de la tarea J por el empleado I.*
- *Es preciso que se realicen todas las tareas.*
- *Un empleado no puede hacer 2 o más tareas.*

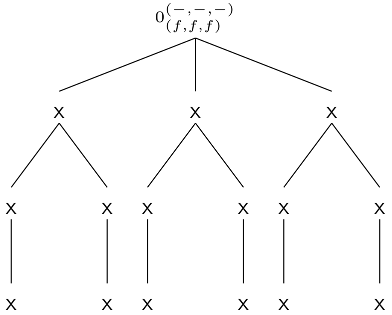
Planteamiento

- Árbol de búsqueda:
  - Cada nivel representa a un empleado.
  - Cada arista representa la asignación a un empleado de una tarea.
- Una tarea no puede volver a realizarse $\rightarrow$ Es preciso almacenar la información de las tareas ya realizadas en un array. $Asignada[1..N] \rightarrow Asignada[k] = cierto$ si la tarea k ya ha sido asignada.
- Información de las planificación, quizá parcial, obtenida en un momento dado: $Fun[1..N] \rightarrow Fun[q]$: tarea asignada al funcionario q.
- $Coste$ almacena el coste de la planificación, quizá parcial, en cada momento.
- $Fun\_min[1..N]$ y $Coste\_min$ contienen la mejor planificación global alcanzada hasta ese momento y su coste correspondiente.

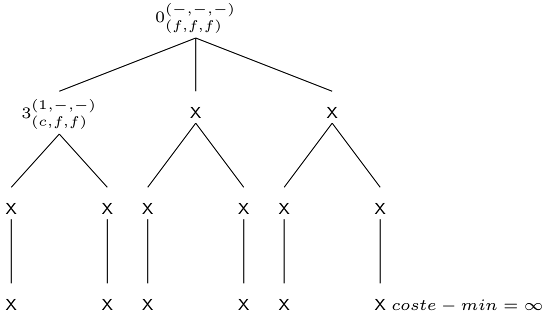
Seguimiento:

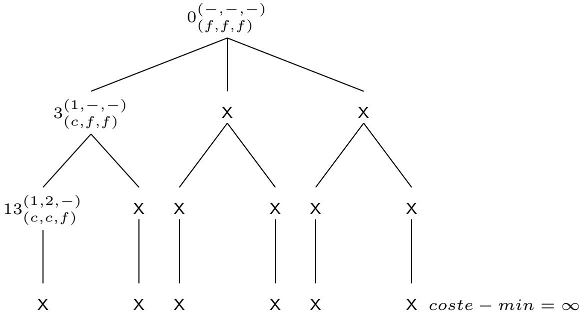
Sea T:

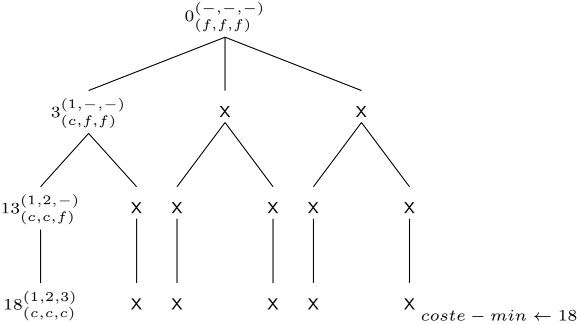
$$\begin{bmatrix} 3 & 5 & 1 \\ 10 & 10 & 1 \\ 8 & 5 & 5 \end{bmatrix}$$

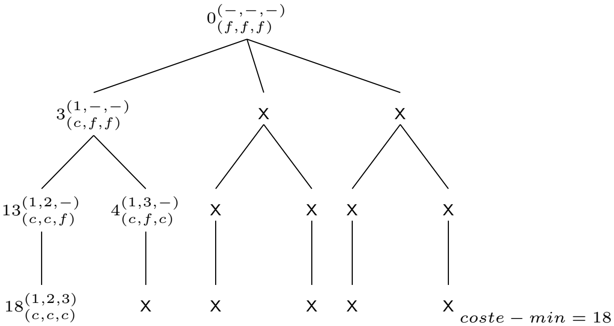


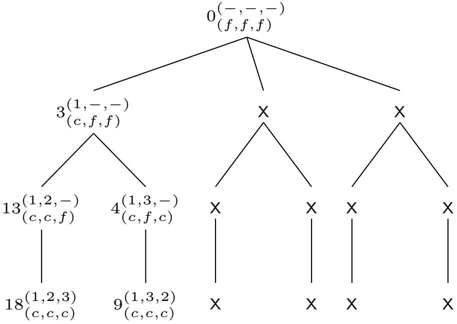
$coste - min = \infty$


$0^{(-,-,-)}_{(f,f,f)}$
$3^{(1,-,-)}_{(c,f,f)}$
X
X
X
X
X
X
X
X
X
X
X
X
X
X $coste - min = \infty$

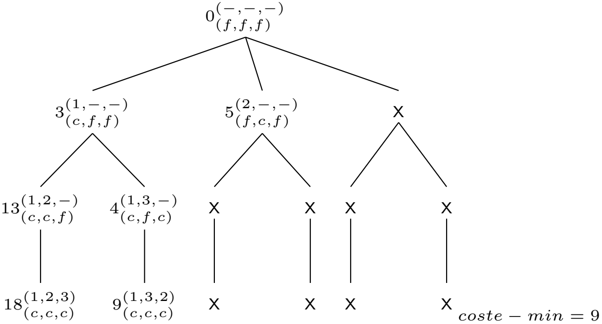

$0^{(-,-,-)}_{(f,f,f)}$
$3^{(1,-,-)}_{(c,f,f)}$
X
X
$13^{(1,2,-)}_{(c,c,f)}$
X
X
X
X
X
X
X
X
X
X
X
$coste - min = \infty$

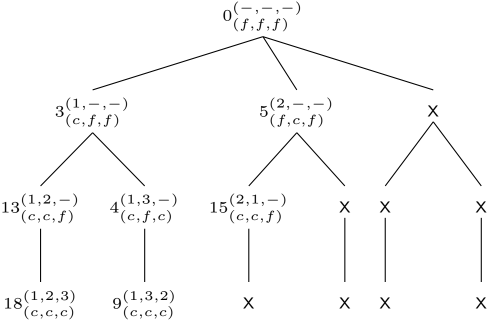

$0^{(-,-,-)}_{(f,f,f)}$
$3^{(1,-,-)}_{(c,f,f)}$
X
X
$13^{(1,2,-)}_{(c,c,f)}$
X
X
X
X
X
$18^{(1,2,3)}_{(c,c,c)}$
X
X
X
X
X
$coste-min \leftarrow 18$


$0^{(-,-,-)}_{(f,f,f)}$
$3^{(1,-,-)}_{(c,f,f)}$
X
X
$13^{(1,2,-)}_{(c,c,f)}$
$4^{(1,3,-)}_{(c,f,c)}$
X
X
X
X
$18^{(1,2,3)}_{(c,c,c)}$
X
X
X
X
X
$coste - min = 18$

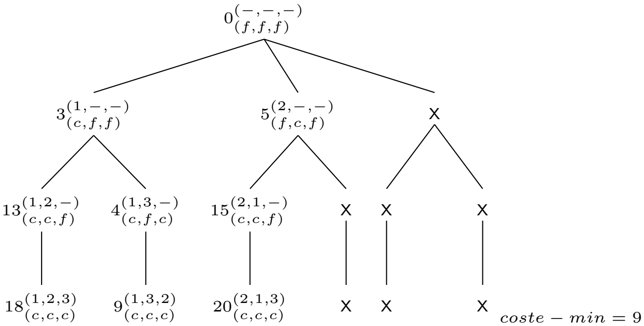


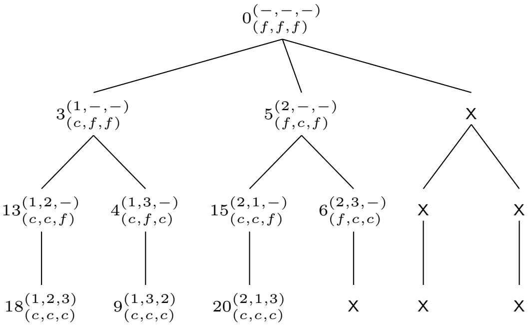
$coste - min \leftarrow 9$


$0^{(-,-,-)}_{(f,f,f)}$
$3^{(1,-,-)}_{(c,f,f)}$
$5^{(2,-,-)}_{(f,c,f)}$
X
$13^{(1,2,-)}_{(c,c,f)}$
$4^{(1,3,-)}_{(c,f,c)}$
X
X
X
X
$18^{(1,2,3)}_{(c,c,c)}$
$9^{(1,3,2)}_{(c,c,c)}$
X
X
X
X
$coste - min = 9$

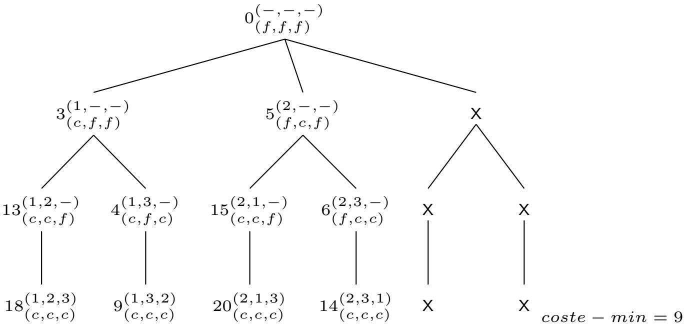

$0^{(-,-,-)}_{(f,f,f)}$
$3^{(1,-,-)}_{(c,f,f)}$
$5^{(2,-,-)}_{(f,c,f)}$
X
$13^{(1,2,-)}_{(c,c,f)}$
$4^{(1,3,-)}_{(c,f,c)}$
$15^{(2,1,-)}_{(c,c,f)}$
X
X
X
$18^{(1,2,3)}_{(c,c,c)}$
$9^{(1,3,2)}_{(c,c,c)}$
X
X
X
X

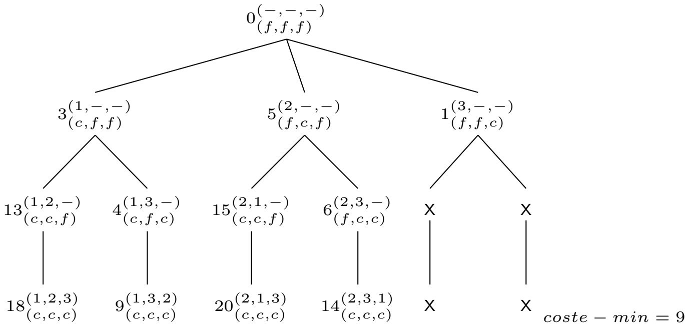
$coste - min = 9$

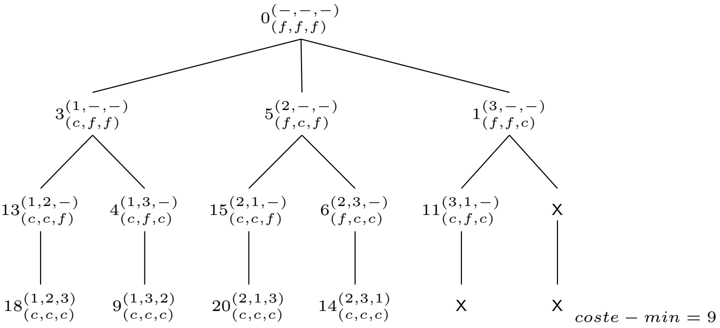

$0^{(-,-,-)}_{(f,f,f)}$
$3^{(1,-,-)}_{(c,f,f)}$
$5^{(2,-,-)}_{(f,c,f)}$
X
$13^{(1,2,-)}_{(c,c,f)}$
$4^{(1,3,-)}_{(c,f,c)}$
$15^{(2,1,-)}_{(c,c,f)}$
X
X
X
$18^{(1,2,3)}_{(c,c,c)}$
$9^{(1,3,2)}_{(c,c,c)}$
$20^{(2,1,3)}_{(c,c,c)}$
X
X
X
$coste-min = 9$

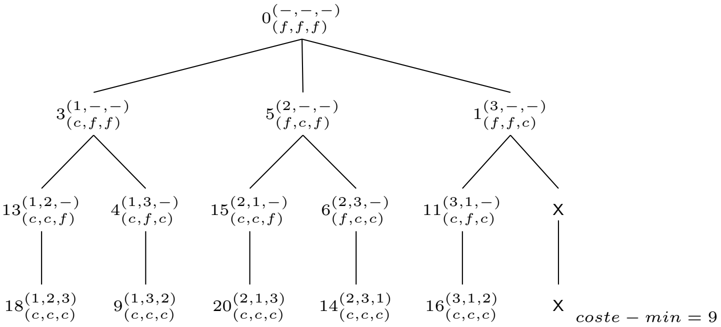

$0^{(-,-,-)}_{(f,f,f)}$
$3^{(1,-,-)}_{(c,f,f)}$
$5^{(2,-,-)}_{(f,c,f)}$
X
$13^{(1,2,-)}_{(c,c,f)}$
$4^{(1,3,-)}_{(c,f,c)}$
$15^{(2,1,-)}_{(c,c,f)}$
$6^{(2,3,-)}_{(f,c,c)}$
X
X
$18^{(1,2,3)}_{(c,c,c)}$
$9^{(1,3,2)}_{(c,c,c)}$
$20^{(2,1,3)}_{(c,c,c)}$
X
X
X

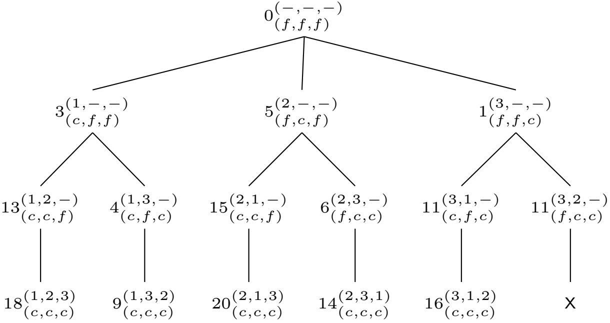
$coste - min = 9$


$0^{(-,-,-)}_{(f,f,f)}$
$3^{(1,-,-)}_{(c,f,f)}$
$5^{(2,-,-)}_{(f,c,f)}$
X
$13^{(1,2,-)}_{(c,c,f)}$
$4^{(1,3,-)}_{(c,f,c)}$
$15^{(2,1,-)}_{(c,c,f)}$
$6^{(2,3,-)}_{(f,c,c)}$
X
X
$18^{(1,2,3)}_{(c,c,c)}$
$9^{(1,3,2)}_{(c,c,c)}$
$20^{(2,1,3)}_{(c,c,c)}$
$14^{(2,3,1)}_{(c,c,c)}$
X
X
$coste - min = 9$

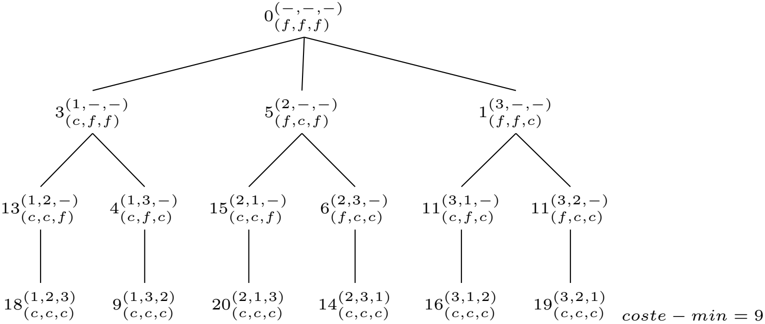

$0^{(-,-,-)}_{(f,f,f)}$
$3^{(1,-,-)}_{(c,f,f)}$
$5^{(2,-,-)}_{(f,c,f)}$
$1^{(3,-,-)}_{(f,f,c)}$
$13^{(1,2,-)}_{(c,c,f)}$
$4^{(1,3,-)}_{(c,f,c)}$
$15^{(2,1,-)}_{(c,c,f)}$
$6^{(2,3,-)}_{(f,c,c)}$
X
X
$18^{(1,2,3)}_{(c,c,c)}$
$9^{(1,3,2)}_{(c,c,c)}$
$20^{(2,1,3)}_{(c,c,c)}$
$14^{(2,3,1)}_{(c,c,c)}$
X
X
$coste - min = 9$


$0^{(-,-,-)}_{(f,f,f)}$
$3^{(1,-,-)}_{(c,f,f)}$
$5^{(2,-,-)}_{(f,c,f)}$
$1^{(3,-,-)}_{(f,f,c)}$
$13^{(1,2,-)}_{(c,c,f)}$
$4^{(1,3,-)}_{(c,f,c)}$
$15^{(2,1,-)}_{(c,c,f)}$
$6^{(2,3,-)}_{(f,c,c)}$
$11^{(3,1,-)}_{(c,f,c)}$
X
$18^{(1,2,3)}_{(c,c,c)}$
$9^{(1,3,2)}_{(c,c,c)}$
$20^{(2,1,3)}_{(c,c,c)}$
$14^{(2,3,1)}_{(c,c,c)}$
X
X
$coste - min = 9$


$0^{(-,-,-)}_{(f,f,f)}$
$3^{(1,-,-)}_{(c,f,f)}$
$5^{(2,-,-)}_{(f,c,f)}$
$1^{(3,-,-)}_{(f,f,c)}$
$13^{(1,2,-)}_{(c,c,f)}$
$4^{(1,3,-)}_{(c,f,c)}$
$15^{(2,1,-)}_{(c,c,f)}$
$6^{(2,3,-)}_{(f,c,c)}$
$11^{(3,1,-)}_{(c,f,c)}$
X
$18^{(1,2,3)}_{(c,c,c)}$
$9^{(1,3,2)}_{(c,c,c)}$
$20^{(2,1,3)}_{(c,c,c)}$
$14^{(2,3,1)}_{(c,c,c)}$
$16^{(3,1,2)}_{(c,c,c)}$
X
$coste - min = 9$


$0^{(-,-,-)}_{(f,f,f)}$
$3^{(1,-,-)}_{(c,f,f)}$
$5^{(2,-,-)}_{(f,c,f)}$
$1^{(3,-,-)}_{(f,f,c)}$
$13^{(1,2,-)}_{(c,c,f)}$
$4^{(1,3,-)}_{(c,f,c)}$
$15^{(2,1,-)}_{(c,c,f)}$
$6^{(2,3,-)}_{(f,c,c)}$
$11^{(3,1,-)}_{(c,f,c)}$
$11^{(3,2,-)}_{(f,c,c)}$
$18^{(1,2,3)}_{(c,c,c)}$
$9^{(1,3,2)}_{(c,c,c)}$
$20^{(2,1,3)}_{(c,c,c)}$
$14^{(2,3,1)}_{(c,c,c)}$
$16^{(3,1,2)}_{(c,c,c)}$
X

$coste - min = 9$


$0^{(-,-,-)}_{(f,f,f)}$
$3^{(1,-,-)}_{(c,f,f)}$
$5^{(2,-,-)}_{(f,c,f)}$
$1^{(3,-,-)}_{(f,f,c)}$
$13^{(1,2,-)}_{(c,c,f)}$
$4^{(1,3,-)}_{(c,f,c)}$
$15^{(2,1,-)}_{(c,c,f)}$
$6^{(2,3,-)}_{(f,c,c)}$
$11^{(3,1,-)}_{(c,f,c)}$
$11^{(3,2,-)}_{(f,c,c)}$
$18^{(1,2,3)}_{(c,c,c)}$
$9^{(1,3,2)}_{(c,c,c)}$
$20^{(2,1,3)}_{(c,c,c)}$
$14^{(2,3,1)}_{(c,c,c)}$
$16^{(3,1,2)}_{(c,c,c)}$
$19^{(3,2,1)}_{(c,c,c)}$
$coste - min = 9$

Algoritmo

proc *funcionarios(q, fun, fun-mejor, asignada, coste, coste-min, T)*
  //Asignación de tareas al funcionario q
  desde $ntarea \leftarrow 1$ **hasta** n **hacer**
    //Asignación de la tarea ntarea al funcionario q
    si $\neg asignada[ntarea]$ **entonces**
      $Fun[q] \leftarrow ntarea$
      $coste \leftarrow coste + t[q, ntarea]$
      $asignada[ntarea] \leftarrow cierto$
      si $q = n$ **entonces**
        //Tenemos una planificación global
        si $coste < coste - min$ **entonces**
          $coste - min \leftarrow coste$
          $fun - mejor \leftarrow fun$
        fin si
      si no
        //Seguimos asignando tareas
        si $coste < coste - min$ **entonces**
          $funcionarios(q + 1, fun, fun - mejor, asignada, coste, coste - min, T)$
        fin si
      fin si
      //Deshacemos la asignación para probar otras asignaciones
      $coste \leftarrow coste - t[q, ntarea]$
      $asignada[ntarea] \leftarrow falso$
    fin si
  fin desde
fin proc

El programa llamador puede ser:

Algoritmo

proc *tareas-funcionarios(T,fun-mejor)*
  desde $ntarea \leftarrow 1$ **hasta** n **hacer**
    $asignada[ntarea] \leftarrow falso$
  fin desde
  $coste - min \leftarrow \infty$
  $coste \leftarrow 0$
  $funcionarios(1, fun, fun - mejor, asignada, coste, coste - min, T)$
fin proc

Complejidad: $\theta(n!)$

Problema

Sean N tipos de objetos cuyo valor y peso son conocidos. Sea una mochila cuya capacidad, en peso, es conocida. Para cada tipo de objeto tenemos un número de copias conocidas. Determina la composición que maximiza el valor de la mochila.

Planteamiento:

- Datos:
  - $v[1..n] \rightarrow v[i]$: valor del tipo de objeto i.
  - $w[1..n] \rightarrow w[i]$: peso del tipo de objeto i.
  - $c[1..n] \rightarrow c[i]$: copias del tipo de objeto i.
- Como no queremos repetir combinaciones de elementos, pues el orden no importa, imponemos la condición de que solamente se pueden introducir objetos del mismo tipo que el último introducido o posteriores.
- Arbol de búsqueda:
  - Cada nivel representa el número de objetos introducidos.
  - Cada arista representa la introducción de un objeto en la mochila.

Algoritmo

proc *llamar-mochila (v[1..n],w[1..n],c[1..n],W,comp-op)*

$composicion \leftarrow \emptyset$

$valor \leftarrow 0$

$peso \leftarrow 0$

mochila (1,v[1..n],w[1..n],c[1..n],W,comp-op, composicion, valor, valor-op, peso)

fin proc

Algoritmo

proc *mochila (q,v[1..n],w[1..n],c[1..n],W,comp-op,composicion, valor, valor-op, peso)*
  desde $I \leftarrow q$ **hasta** N **hacer**
    si $c[I] > 0 \wedge peso + v[I] \leq W$ **entonces**
      $composicion \leftarrow composicion \cup \{I\}$
      $valor \leftarrow valor + v[I]$
      $peso \leftarrow peso + w[I]$
      $c[I] - -$
      si $valor > valor - op$ **entonces**
        $valor - op \leftarrow valor$
        $comp - op \leftarrow composicion$
      fin si
      mochila(I,v[1..n],w[1..n],c[1..n],W,comp-op, composicion, valor, valor-op, peso)
      $composicion \leftarrow composicion \setminus \{I\}$
      $valor \leftarrow valor - v[I]$
      $peso \leftarrow peso - w[I]$
      $c[I] + +$
    fin si
  fin desde
fin proc

Observaciones:

- ¿Por qué no aparece el nivel en el algoritmo?.
- La mochila con N tipos de objertos con número infinito de copias $\Leftrightarrow$ quitar la condición: $c[i] > 0$.
- La mochila con N objetos $\Leftrightarrow$ $c[i] = 1$.

Arbol alternativo:

- El nivel representa el tipo de objeto procesado.
- En cada arista decidimos introducir una cantidad de 0 al número de copias de cada tipo objeto.
- La solución necesariamente tiene que estar en el nivel n.

Algoritmo:

Algoritmo

proc *mochila (q,v[1..n],w[1..n],c[1..n],W,comp-op,composicion, valor, valor-op, peso)*
  desde $I \leftarrow 0$ **hasta** $min\{c[q], (W - peso)/w[q]\}$ **hacer**
    $composicion \leftarrow composicion \cup \{(I, q)\}$
    $valor \leftarrow valor + v[q] * I$
    $peso \leftarrow peso + w[q] * I$
    si $q = n$ **entonces**
      si $valor > valor - op$ **entonces**
        $valor - op \leftarrow valor$
        $comp - op \leftarrow composicion$
      fin si
    fin si
    mochila(q+1,v[1..n],w[1..n],c[1..n],W,comp-op, composicion, valor, valor-op, peso)
    $composicion \leftarrow composicion \setminus \{(I, q)\}$
    $valor \leftarrow valor - v[q] * I$
    $peso \leftarrow peso - w[q] * I$
  fin desde
fin proc

Programa llamador:

Algoritmo

proc *llamar-mochila (v[1..n],w[1..n],c[1..n],W,comp-op)*
$composicion \leftarrow vacio$
$valor \leftarrow 0$
$peso \leftarrow 0$
mochila (1,v[1..n],w[1..n],c[1..n],W,comp-op, composicion, valor, valor-op, peso)
fin proc

Problema

Consideremos una postal de M sellos como máximo (M "huecos"). Tenemos un conjunto de copias conocidas de N tipos de sellos. La postal tiene una tarifa mínima para poder ser enviada (es posible sobrepasarla!!!). Además, se supone que el orden de los sellos en la postal debe de ser tal que su valor sea creciente. Determinar la forma de franquear la postal que minimice el coste.

Planteamiento:

- $valor[1..N] \rightarrow valor[i]$: representa el valor los sellos de tipo i. Se supone que está ordenado crecientemente.
- $cantidad[1..N] \rightarrow cantidad[i]$: representa la cantidad de sellos del tipo i.
- $postal[1..M] \rightarrow postal[i]$: representa la posición i de la postal.
- Árbol de búsqueda:
    - Cada arista representa el pegado de un sello en una posición.
    - Cada nivel representa el franqueo de una posición.

Algoritmo

proc *franqueo(K, valor[1..N], cantidad[1..N], Tarifa, coste, postal[1..M], coste-min, postal-mejor[1..M])*
  //K: número de posición franqueada en la postal
  //T: último tipo de sello utilizado
  si $K \leq M$ **entonces**
    desde $ntiposello \leftarrow 1$ **hasta** N **hacer**
      si $cantidad[ntiposello] > 0 \wedge_c (k = 1 \vee_c valor[postal[k-1]] \leq valor[ntiposello])$
      entonces
        $cantidad[ntiposello] - -$
        $postal[K] \leftarrow ntiposello$
        $coste \leftarrow coste + valor[ntiposello]$
        si $coste \geq tarifa$ **entonces**
          si $coste < coste - min$ **entonces**
            $coste - min \leftarrow coste$
            $postal - mejor \leftarrow postal$
          fin si
        si no
          franqueo(K+1, ntiposello,...)
        fin si
        $cantidad[ntiposello] + +$
        $coste \leftarrow coste - valor[ntiposello]$
      fin si
    fin desde
  fin si
fin proc

La primera llamada será:

Algoritmo

proc *franquear(valor[1..N], cantidad[1..N], Tarifa, postal-mejor[1..M])*

$coste \leftarrow 0$

$coste - min \leftarrow \infty$

$franqueo(1, 1, valor, cantidad, Tarifa, coste, postal, coste - min, postal - mejor)$

fin proc

Problema

Dado un mapa con N países y M colores disponibles, determina un algoritmo que puestre todas las formas posibles de colorearlo suponiendo que dos países adyacentes no pueden tener el mismo color.

Planteamiento:

- Representamos el mapa mediante un grafo $\rightarrow$ Matriz de adyacencia: *G[I,J]: cierto*, si existe frontera entre I y J.
- Arbol de búsqueda:
  - Cada nivel representa el procesamiento de un país.
  - Cada arista representa la decisión de pintar un país con un color.
  - Cada nodo contiene un coloreado parcial del mapa.

Algoritmo

proc *colorear-mapa(K,G[1..N,1..N],pais[1..N],color[1..M])*
  //Coloreamos el pais K
  desde $J \leftarrow 1$ **hasta** M **hacer**
    //Utilizamos el color J
    $OK \leftarrow cierto$
    //Comprobamos si algún país adyacente tiene ya ese color
    desde $Q \leftarrow 1$ **hasta** n **hacer**
      si $G[K,Q] = cierto$ **entonces**
        si $pais[Q] = color[J]$ **entonces**
          $OK \leftarrow falso$
        fin si
      fin si
    fin desde
    si $OK = cierto$ **entonces**
      $pais[K] \leftarrow color[J]$
      si $K = N$ **entonces**
        imprimir(pais)
      si no
        $colorear - mapa(K+1, G, pais, color)$
      fin si
      $pais[K] \leftarrow 0$
    fin si
  fin desde
fin proc

Problema

Dado un mapa con N países y M colores disponibles, determina un algoritmo que el coloreado del mapa que utilice el mínimo número de colores suponiendo que dos países adyacentes no pueden tener el mismo color.

Algoritmo

proc *colorear-mapa(K,G[1..N,1..N],pais[1..N],color[1..M],num,pais-op[1..N])*
  //Coloreamos el pais K
  desde $J \leftarrow 1$ **hasta** M **hacer**
    $chequear(J, G, pais, color)$
    si $OK = cierto$ **entonces**
      $pais[K] \leftarrow color[J]$
      $num - ant \leftarrow num$
      $z \leftarrow 1$
      $fin \leftarrow falso$
      mientras $z \leq K - 1 \wedge \neg fin$ **hacer**
        si $pais[z] = color[J]$ **entonces**
          $fin \leftarrow true$
        fin si
        $z \leftarrow z + 1$
      fin mientras
      ...
    fin si
  fin desde
fin proc

Algoritmo

```
proc colorear-mapa(K,G[1..N,1..N],pais[1..N],color[1..M],num,pais-op[1..N])
  //Coloreamos el pais K
  desde J ← 1 hasta M hacer
    chequear(J, G, pais, color)
    si OK = cierto entonces
      ...
      si ¬fin entonces
        num ← num + 1
      fin si
      si K = N entonces
        si num − op > num entonces
          num − op ← num
          pais − op ← pais
        fin si
      si no
        colorear − mapa(K + 1, G, pais, color, num, pais − op[1..N])
      fin si
      pais[K] ← 0
      num ← num − ant
    fin si
  fin desde
fin proc
```

Algoritmo

fun *chequear(J:ent,G[1..N,1..N],pais[1..N],color[1..M])***dev** *OK:bool*
  $OK \leftarrow cierto$
  desde $Q \leftarrow 1$ **hasta** N **hacer**
    si $G[K, Q] = cierto$ **entonces**
      si $pais[Q] = color[J]$ **entonces**
        $OK \leftarrow falso$
      fin si
    fin si
  fin desde
fin fun

Problema

Un camino hamiltoniano en un grafo es un camino, es decir, una sucesión de aristas adyacentes, que visita todos los vértices del grafo una sola vez. Si además el último vértice visitado es adyacente al primero, el camino es un ciclo hamiltoniano. Diseña un algoritmo, para un nodo dado, determine:

1. *Todos los ciclos hamiltonianos que existen.*
2. *El ciclo hamiltoniano de menor coste.*

Planteamiento:

- La solución tiene estructura de N-tupla. El I-ésimo nodo contiene el nodo visitado en el movimiento I.
- Árbol de búsqueda:
  - Cada arista representa el movimiento a un nodo adyacente no visitado.
  - Cada nivel representa el número de movimiento realizado.

Algoritmo 1:

Algoritmo

proc *CH(G[1..n,1..n],sol[1..n],k,visitado[1..n])*
  desde $vertice \leftarrow 2$ **hasta** n **hacer**
    si $\neg visitado[vertice] \wedge G[sol[k-1]], vertice] < \infty$ **entonces**
      $sol[k] \leftarrow vertice$
      $visitado[vertice] \leftarrow cierto$
      si $k = n$ **entonces**
        si $G[sol[n], 1] < \infty$ **entonces**
          $imprimir(sol)$
        fin si
      si no
        $CH(G[1..n, 1..n], sol[1..n], k+1, visitado[1..n])$
      fin si
      $visitado[vertice] \leftarrow falso$
    fin si
  fin desde
fin proc

Algoritmo 2:

Algoritmo

proc *CH(G[1..n,1..n],sol[1..n],k,visitado[1..n], coste,sol-op, coste-min)*
  desde $vertice \leftarrow 2$ **hasta** n **hacer**
    si $\neg visitado[vertice] \wedge G[sol[k-1]], vertice] < \infty$ **entonces**
      $sol[k] \leftarrow vertice$
      $visitado[vertice] \leftarrow cierto$
      $coste \leftarrow coste + G[sol[k-1], vertice]$
      si $k = n$ **entonces**
        si $G[sol[n], 1] < \infty$ **entonces**
          $coste \leftarrow coste + G[sol[k-1], 1]$
          si $coste < coste - min$ **entonces**
            $coste - min \leftarrow coste$
            $sol - op \leftarrow sol$
          fin si
          $coste \leftarrow coste - G[sol[k-1], 1]$
        fin si
      si no
        $CH(G[1..n, 1..n], sol[1..n], k+1, visitado[1..n], coste, sol - op, coste - min)$
      fin si
      $visitado[vertice] \leftarrow falso$
      $coste \leftarrow coste - G[sol[k-1], vertice]$
    fin si
  fin desde
fin proc

Problema

Dado una tablero de ajedrez de dimensión n, determina todas las formas posibles colocar en él n reinas sin que se amenacen entre sí.

Planteamiento:

- Arbol de búsqueda:
  - El nivel representa el número de reinas colocadas,
  - Cada arista representa el hecho de colocar una reina en una posición válida.
  - Cada nodo contiene una colocación (quizás parcial de las reinas).
- La solución tiene formato de n-tupla: $sol[1..n] \rightarrow sol[i]$: fila de la reina i.
- Información adicional de cada nodo:
  - Filas ya ocupadas: $Fil = \{sol[i] \ \forall i : 1..k\}$.
  - Diagonales de 45° ya ocupadas: $Diag45 = \{sol[i] - i + 1 \ \forall i : 1..k\}$.
  - Diagonales de 135° ya ocupadas: $Diag135 = \{sol[i] + i - 1 \ \forall i : 1..k\}$.

Algoritmo

proc *reina(k, fil, diag-45, diag-135)*
  //tenemos k reinas ya colocadas, colocamos la reina k+1
  si $k = n$ **entonces**
    $imprimir(sol)$
  si no
    desde $J \leftarrow 1$ **hasta** n **hacer**
      si $J \notin Fil \wedge J - k \notin diag45 \wedge J + k \notin diag135$ **entonces**
        //colocamos la reina
        $sol[k+1] \leftarrow J$
        $reina(k+1, fil \cup \{J\}, diag45 \cup \{J-k\}, \{J+k\} \cup diag135)$
      fin si
    fin desde
  fin si
fin proc

Problema

Consideremos un caballo de ajedrez colocado en una posición, X, Y de un tablero de dimensiones $N \times N$. Diseñad u algoritmo que determine si es posible visitar todas las casillas pasando exactamente una vez por cada una de ellas.

Planteamiento:
La solución al problema tiene la siguiente forma:

Algoritmo

proc *caballo*
  repetir
    "seleccionar un movimiento del caballo"
    si *"esta en el tablero"* $\wedge$ *"no ha pasado por esta casilla"* **entonces**
      "anotamos movimiento"
      si *"no hemos completado tablero"* **entonces**
        "seguimos moviendo"
        si *"no se alcanzó la solución"* **entonces**
          "borrar anotación anterior"
        fin si
      fin si
    fin si
  hasta *"completado tablero"* $\vee$ *"agotado los 8 movimientos"*
fin proc

Podemos generar el algoritmo codificando las situaciones anteriores.

- Utilizaremos un array de dos dimensiones, $T[1..N, 1..N]$, para representar el tablero:

$$T[X,Y] = \begin{cases} 0 & \text{si no hemos pasado por esa posición} \\ I & \text{Si hemos pasado por ella en el movimiento I} \end{cases}$$

- Codificación de los 8 movimientos posibles del caballo. Matriz de desplazamientos relativos:

$$\begin{bmatrix} 2 & 1 & -1 & -2 & -2 & -1 & 1 & 2 \\ 1 & 2 & 2 & 1 & -1 & -2 & -2 & -1 \end{bmatrix}$$

- Cuestión: Hasta qué punto influye la numeración de los movimientos del caballo.
- Las coordenadas de un nuevo movimiento, (X, Y), están en el tablero si: $X \in [1..N]$ e $Y \in [1..N]$.
- Una casilla, (X, Y), no ha sido visitada si $T[X, Y] = 0$.
- Hemos completado el tablero si $I = N \times N$.

Algoritmo

proc *caballo(I,X,Y,S,T,D)*
  //I: numero de movimientos realizados
  $S \leftarrow falso$
  $K \leftarrow 0$ *//Número de movimiento del caballo*
  repetir
    $K \leftarrow k + 1$
    // Nuevas coordenadas
    $Nx \leftarrow X + D[1, K]$
    $Ny \leftarrow Y + D[2, K]$
    si $Nx \in [1..N] \wedge Ny \in [1..N]$ **entonces**
      si $T[Nx, Ny] = 0$ **entonces**
        //No ha sido visitada
        $T[Nx, Ny] \leftarrow I$
        si $I < N^2$ **entonces**
          $caballo(I + 1, Nx, Ny, S, T, D)$ *//Aún quedan casillas por visitar*
          si $\neg S$ **entonces**
            //No hemos encontrado la solución en el resto de movimientos
            $T[Nx, Ny] \leftarrow 0$
          fin si
        si no
          $S \leftarrow cierto$
        fin si
      fin si
    fin si
  hasta $S \vee K = 8$
fin proc

Cambios respecto de las versiones recursivas:

- Se hace explícita la pila que utiliza el compilador cuando utilizamos algoritmos recursivos.
  - Cada llamada recursiva se corresponderá con la operación de apilar.
  - Cada retorno de la función recursiva se corresponderá con la operación de desapilar.
- El conjunto de variables del procedimiento recursivo (variables locales y parámetros formales) determinan cada uno de los estados de nuestro árbol de búsqueda en backtraking recursivo.
- Es preciso establecer qué variables van a formar parte de cada estado del árbol de busqueda.

Planteamiento:

- Árbol de búsqueda:
  - Cada nivel representa la asignación de una tarea a un empleado.
  - Cada arista representa la asignación de una tarea concreta a un empleado.
- Variables utilizadas en la versión recursiva:
  - q: número de empleado procesado.
  - fun: planificacion posible (quizá parcial) (array).
  - fun_mejor: mejor planificación disponible en un momento dado (array).
  - $asignada$: tareas ya asignadas en un momento dado (array).
  - $coste$: coste de una planificación posible.
  - $coste\_min$: coste de la mejor planificación disponible.

- Variables utilizadas en la versión iterativa:
  - Relativas a cada nodo:
    - q: número de empleado procesado.
    - fun: planificacion posible (quizá parcial) (array).
    - $asignada$: tareas ya asignadas en un momento dado (array).
    - $coste$: coste de una planificación posible.
  - Independientes de un nodo en concreto:
    - fun_mejor: mejor planificación disponible en un momento dado (array).
    - $coste\_min$: coste de la mejor planificación disponible.

Algoritmo

proc *funcionarios (T[1..N,1..N], fun-mejor, coste-min)*
  $v.q \leftarrow 0$
  $v.coste \leftarrow 0$
  $coste - min \leftarrow \infty$
  desde $J \leftarrow 1$ **hasta** N **hacer**
    $v.asignada[J] \leftarrow falso$
  fin desde
  $insertar(P, v)$
  mientras $\neg vacia(P)$ **hacer**
    desapilar (P,v)
    si $v.q = N$ **entonces**
      si $v.coste < coste - min$ **entonces**
        $coste - min \leftarrow v.coste$
        $fun - mejor \leftarrow v.fun$
      fin si
    si no
      $u.q \leftarrow v.q + 1$
      desde $K \leftarrow 1$ **hasta** N **hacer**
        si $\neg v.asignada[K]$ **entonces**
          $u.fun \leftarrow v.fun$
          $u.fun[u.q] \leftarrow K$
          $u.coste \leftarrow v.coste + T[u.q, K]$
          $u.asignada \leftarrow v.asignada$
          $u.asignada[K] \leftarrow cierto$
          $insertar(P, u)$
        fin si
      fin desde
    fin si

Algoritmo

```
proc colorear-mapa (G[1..N,1..N],M)
  v.nivel ← 0
  apilar(P, v)
  mientras ¬vacia(P) hacer
    desapilar(P, v)
    si v.nivel = N entonces
      imprimir(v.pais)
    si no
      u.nivel ← v.nivel + 1
      desde J ← 1 hasta M hacer
        // Probamos el color J
        ok ← cierto
        desde q ← 1 hasta N hacer
          //Miramos el color de los adyacentes
          si G[u.nivel, q] entonces
            si v.pais[q] = J entonces
              ok ← falso
            fin si
          fin si
        fin desde
        si ok = cierto entonces
          u.pais ← v.pais
          u.pais[u.nivel] ← J
          apilar(P, u)
        fin si
      fin desde
    fin si
  fin mientras
```

Problema

Consideremos la siguiente ecuación:

$$C_nX_n + C_{n-1}X_{n-1} + ... + C_1X_1 + C_0 = 0$$
$$0 < X_i < d_i \ \forall i : 1..n$$

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* $X_i \in \mathbb{N}$ $\forall i : 1..n$.

Algoritmo

proc *resolver-ecuacion(C[0..N], d[1..N])*
  $Acum \leftarrow C[0]$
  $ecuacion(1, C[0..N], X[1..N], d[1..N], Acum)$
fin proc
proc *ecuacion (q, C[0..N], X[1..N], d[1..N], Acum)*
  desde $J \leftarrow 1$ **hasta** $hastaparte - entera(d[i])$ **hacer**
    $X[q] \leftarrow J$
    $Acum \leftarrow C[q] * X[q] + Acum$
    si $q = N$ **entonces**
      si $Acum = 0$ **entonces**
        $imprimir(X)$
      fin si
    si no
      $ecuacion(q + 1, C[0..N], X[1..N], d[1..N], Acum)$
    fin si
    $Acum \leftarrow C[q] * X[q] - Acum$
  fin desde
fin proc

Problema

Dadas n tareas y m empleados, deseamos realizar las n tareas distribuyéndolas entre los m empleados de tal forma que el tiempo de la planificación sea mínimo. El tiempo de realización de cada tarea por cada empleado es conocido. La asignación de tareas debe cumplir las siguientes condiciones:

- *Todas las tareas tienen que ser realizadas.*
- *Dos tareas no pueden ser realizadas al mismo tiempo por el mismo empleado.*

Algoritmo

proc *tareas(k,T[1..n,1..m], Tmin, Asig-mejor[1..n], Asig[1..n],Trab[1..m])*
  //k: tarea a procesar
  //Trab[J]: tiempo que tarda el trabajador J para realizar sus tareas asignadas
  desde $J \leftarrow 1$ **hasta** m **hacer**
    //J: empleado a considerar
    $Asig[k] \leftarrow J$
    $Trab[J] \leftarrow T[k, J] + Trab[J]$
    si $k = n$ **entonces**
      si $Tmin > maximo(Trab)$ **entonces**
        $Tmin \leftarrow maximo(Trab)$
        $Asig - mejor \leftarrow Asig$
      fin si
    si no
      $tareas(k + 1, T[1..n, 1..m], Tmin, Asig - mejor[1..n], , Asig[1..n], Trab[1..m])$
    fin si
    $Trab[J] \leftarrow Trab[J] - T[k, J]$
  fin desde
fin proc

Algoritmo

proc *llamada(T[1..n,1..m], Tmin, Asig-mejor[1..m])*
  desde $q \leftarrow 1$ **hasta** m **hacer**
    $Trab[q] \leftarrow 0$
  fin desde
  $Tmin \leftarrow +\infty$
  $tareas(1, T, Tmin, Asig - mejor[1..m], , Asig[1..m], Trab[1..m])$
fin proc

Problema

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.

Ejemplo 1: *Traducir del latín al arameo disponiendo de los siguientes diccionarios:*

latín-griego, griego-etrusco, griego-demótico, demótico-arameo

Solución: Sí es posible la traducción: latín-griego-demótico-arameo, número de traducciones: 3.

Ejemplo 2: *Traducir del latín al arameo disponiendo de los siguientes diccionarios:*

latín-griego, arameo-etrusco, griego-demótico, demótico-hebreo

Solución: No es posible la traducción.

Algoritmo

proc *traducir(K, final, Diccionario[1..N,1,,N], camino-min[1..N] ,trad-min traducido[1..N],camino[1..N],trad)*
  desde $J \leftarrow 1$ **hasta** N **hacer**
    si $Diccionario[camino[K-1], J] \wedge \neg traducido[J]$ **entonces**
      $camino[K] \leftarrow J$
      $trad \leftarrow trad + 1$
      $traducido[J] \leftarrow cierto$
      si $J = final$ **entonces**
        si $trad < trad - min$ **entonces**
          $camino - min \leftarrow camino$
          $trad - min \leftarrow trad$
        fin si
      si no
        *traducir(K+1, final, Diccionario[1..N,1,,N], camino-min[1..N*N], trad-min, traducido[1..N],camino[1..N*N],trad)*
      fin si
      $traducido[J] \leftarrow falso$
      $trad \leftarrow trad - 1$
    fin si
  fin desde
fin proc

Algoritmo

proc *llamar-traducir(inicio, final, Diccionario[1..N,1,,N], camino-min[1..N*N] ,trad-min)*
$trad - min \leftarrow \infty$
$trad \leftarrow 0$
desde $J \leftarrow 1$ **hasta** N **hacer**
$traducido[J] \leftarrow falso$
fin desde
$camino[1] \leftarrow inicio$
*traducir(2, final, Diccionario[1..N,1,,N], camino-min[1..N*N] ,trad-min traducido[1..N],camino[1..N*N],trad)*
fin proc

Problema

Queremos ir de una ciudad a otra de la misma región. Para ello disponemos de los horarios de todos los trenes que comunican las ciudades de esa región. Además de la hora de salida también conocemos la duración de cada viaje. Deseamos encontrar el camino seguido y el tiempo mínimo para viajar entre dos ciudades dadas mediante un algoritmo de vuelta atrás. Comentad la estrategia seguida.

Nota: *Se supone que, a lo sumo, hay un tren para cada par de ciudades.*

Árbol de búsqueda:

- Cada arista representa el desplazamiento a una ciudad.
- Cada nivel representa la k-ésimo movimiento.
- El nodo de cada árbol representa el estado de un viaje parcial.

Algoritmo

proc *Trenes (k, Tren[1..N,1..N], destino, visitada[1..N], tiempo, tiempo-min, camino[1..N], camino-min)*
// K: número de viaje
//camino[k-1]: ciudad de partida del K-ésimo viaje
desde $J \leftarrow 1$ **hasta** N **hacer**
si $\neg visitada[J] \wedge tiempo \leq Tren[camino[k-1], J].salida \wedge Tren[camino[k-1], J].existe\ entonces$ **entonces**
$visitada[J] \leftarrow cierto$
$tiempo - ant \leftarrow tiempo$
$tiempo \leftarrow Tren[camino[k-1], J].salida + Tren[camino[k-1], J].duracion$
$camino[K] \leftarrow J$
si $J = destino$ **entonces**
si $tiempo < tiempo - min$ **entonces**
$tiempo - min \leftarrow tiempo$
$camino - min \leftarrow camino$
fin si
si no
Trenes(k+1, Tren[1..N,1..N], destino, visitada[1..N], tiempo, tiempo-min, camino[1..N], camino-min)
fin si
$visitada[J] \leftarrow falso$
$tiempo \leftarrow tiempo - ant$
fin si
fin desde
fin proc

Algoritmo

proc *llamadaTrenes(hora-inic, origen, destino, Tren[1..N,1..N])*

$tiempo \leftarrow hora - inic$

$tiempo - min \leftarrow \infty$

desde $q \leftarrow 1$ **hasta** N **hacer**

$visitado[q] \leftarrow falso$

fin desde

$camino[1] \leftarrow origen$

$visitado[origen] \leftarrow cierto$

Trenes(2, Tren[1..N,1..N], destino, visitada, tiempo, tiempo-min, camino, camino-min)

fin proc