Vuelta atrás

Yolanda Ortega Mallén

Dpto. de Sistemas Informáticos y Programación

Universidad Complutense de Madrid

Sumario

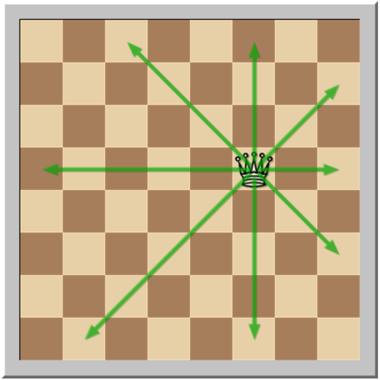
- Un ejemplo: el problema de las 8 reinas.
- Exploración exhaustiva.
- Espacios de soluciones y árboles de exploración.
- Vuelta atrás.
  - Esquema general: encontrar todas las soluciones.
  - Encontrar la primera solución.
  - Técnica de marcaje.
  - Encontrar la mejor solución.

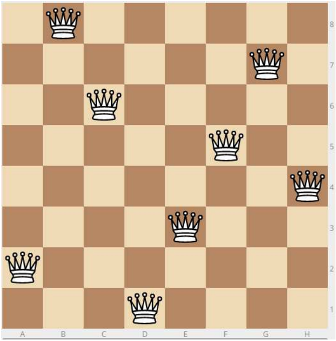
Bibliografía

- R. Neapolitan. *Foundations of Algorithms*. Quinta edición. Jones and Bartlett Publishers, 2015.
  Capítulo 5
- R. Peña. *Algoritmos y estructuras de datos*. Garceta Grupo Editorial, 2019.
  Sección 9.1
- N. Martí Oliet, Y. Ortega Mallén y J. A. Verdejo López. *Estructuras de datos y métodos algorítmicos. 213 Ejercicios resueltos*. Segunda edición. Garceta Grupo Editorial, 2013.
  Capítulo 14

Problema de las ocho reinas

Colocar ocho reinas en un tablero de ajedrez sin que se amenacen entre sí.





Fuerza bruta: Probar todas las posibilidades.

1. Las reinas en cualquier casilla:

$$\binom{64}{8} = 4{,}426{,}165{,}368.$$

2. Cada reina en una fila distinta:

$$8^8 = 16{,}777{,}216.$$

3. Cada reina en una fila y columna distintas:

$$8! = 40{,}320.$$

4. Por etapas y cuando una reina amenaza retroceder:
$15{,}721$ situaciones *parciales* analizadas,
$2{,}057$ situaciones prometedoras.

Exploración exhaustiva

- No siempre se pueden utilizar los métodos algorítmicos que producen soluciones *eficientes*.
- El último recurso es aplicar la fuerza bruta.
- Realizar una búsqueda exhaustiva por el espacio de posibles soluciones hasta encontrar una que satisfaga los criterios exigidos.
- Impracticable si el espacio de soluciones es muy grande.
- Estructurar el espacio a explorar para descartar en bloque posibles soluciones no satisfactorias.

- Construir las soluciones por etapas:
  n-tupla $(x_1, \ldots, x_n)$, $x_i \in S_i$ es la decisión tomada en la etapa i-ésima.
- Satisfacer / optimizar una cierta función criterio.
- Dos categorías de restricciones:

  Explícitas definen los conjuntos (finitos) de alternativas S_i;

  Implícitas relaciones entre las componentes de la tupla solución para satisfacer la función criterio.
- Espacio de soluciones: conjunto de tuplas (parciales / completas) que satisfacen las restricciones explícitas.

Ejemplo: Problema de las 8 reinas

Solución $(x_1, \ldots, x_8)$, $x_i =$ columna ocupada por la reina de la fila i-ésima.

Restricciones explícitas $x_i \in [1..8]$.

Restricciones implícitas dos reinas no comparten ni columna ni diagonales.

$$\forall i, j.(x_i \neq x_j) \wedge |x_i - x_j| \neq |i - j|$$

- El espacio de soluciones puede estructurarse como un árbol de exploración.
- En cada nivel se toma la decisión de la etapa correspondiente.

Nodo estado correspondiente a una tupla parcial o completa que satisface las restricciones explícitas;

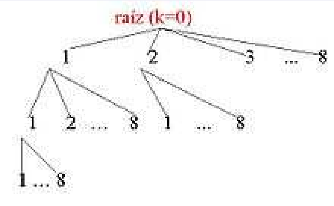
Nodo solución correspondiente a una tupla completa que satisface las restricciones explícitas e implícitas.

Ejemplo: Problema de las ocho reinas

Árbol de permutaciones:

Nodos estado $\sum_{i=1}^{8} 8^i = \frac{8^9-1}{7}$

Nodos solución solo en las hojas.



Poda del árbol: test de factibilidad para determinar si un estado parcial nunca va a conducir a un nodo solución

$\Rightarrow$ es inútil seguir buscando a partir de ese nodo.

Búsqueda en el espacio de soluciones

- Realizar un recorrido del árbol de exploración en cierto orden.
- Para cada nodo se irán generando sus sucesores.

  Nodo vivo todavía no se han generado todos sus hijos;

  Nodo en expansión sus hijos están siendo generados;

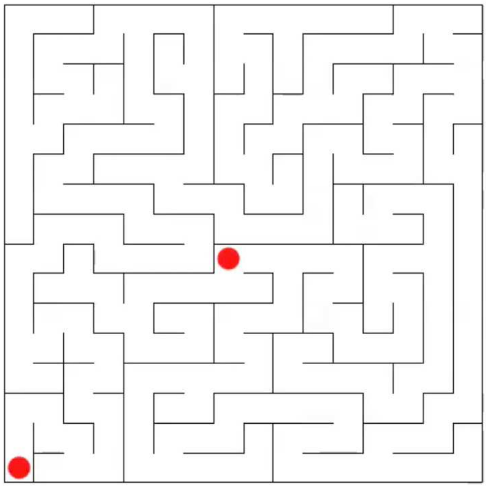
  Nodo muerto no puede ser expandido,
  - no supera el test de factibilidad, o
  - todos sus hijos ya han sido generados.

Vuelta atrás (backtracking): recorrido en profundidad;
los nodos vivos se gestionan mediante una pila.
Sencillo y eficiente en espacio.

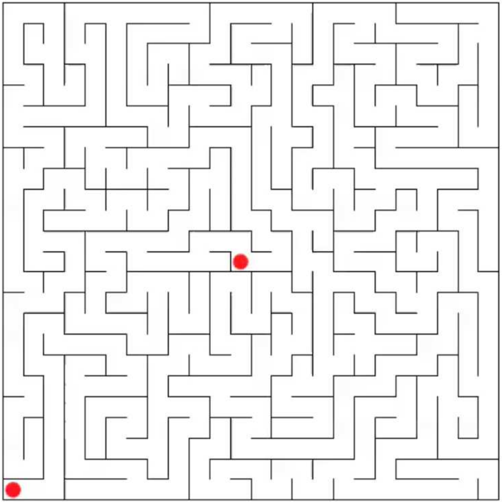
Ramificación y poda (branch & bound): búsqueda más "inteligente" que
expande el nodo vivo "más prometedor";
los nodos vivos se gestionan mediante una cola con prioridad.

- El coste en el caso peor está en el orden del tamaño del espacio de soluciones, que suele ser al menos exponencial.
- Su utilidad práctica depende de la efectividad de las funciones de poda:
  - Detectar muchos nodos no factibles y cuanto más arriba mejor.
  - El coste de aplicación debe compensar la poda.
- Difícil analizar teóricamente a priori; tomar medidas empíricas.

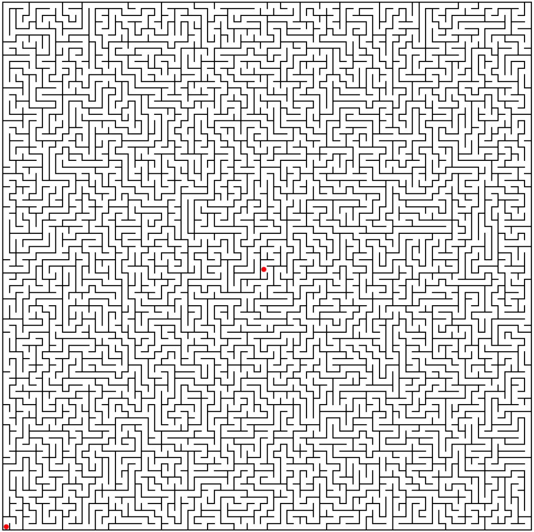
Salir del laberinto



Salir del laberinto

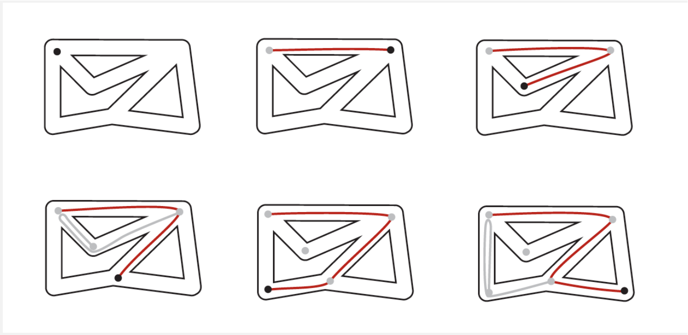


Salir del laberinto



Salir del laberinto

- Desenrollar una madeja de hilo a nuestro paso.
- Marcar cada nueva intersección y pasillo.
- Retroceder cuando no quedan opciones sin marcar.



Realizar una búsqueda en profundidad y al llegar a un nodo muerto, hay que deshacer la última decisión tomada, para optar por la siguiente alternativa.

Esquema general de vuelta atrás

proc `vuelta-atrás`$(sol : \mathit{tupla}, \textbf{e}\ k : \mathit{nat})$
  `preparar-recorrido-nivel`(k)
  mientras $\neg$`último-hijo-nivel`(k) **hacer**
    $sol[k] :=$ `siguiente-hijo-nivel`(k)
    si `es-solución?`(sol, k) **entonces**
      `tratar-solución`(sol)
    si no
      si `es-completable?`(sol, k) **entonces**
        `vuelta-atrás`$(sol, k+1)$
      fsi
    fsi
  fmientras
fproc

Obtiene todas las soluciones.

Los nodos solución están solo en las hojas.

proc `reinas-va1`$(sol[1..n]$ **de** $1..n$, **e** $k : 1..n)$
    para $columna = 1$ **hasta** n **hacer**
        $sol[k] := columna$
        si `no-jaque?`(sol, k) **entonces**
            si $k = n$ **entonces** `imprimir`(sol)
            si no `reinas-va1`$(sol, k+1)$
            fsi
        fsi
    fpara
fproc

$\{$ no hay jaque en $sol[1..k-1]$ $\}$
fun `no-jaque?`$(sol[1..n]$ **de** *nat*, $k : 1..n)$ **dev** $respuesta$: *bool*
    $i := 1$
    $respuesta :=$ `cierto`
    mientras $i \neq k \wedge respuesta$ **hacer**
        $respuesta := (sol[k] \neq sol[i]) \wedge (|sol[k] - sol[i]| \neq k - i)$
        $i := i + 1$
    fmientras
ffun

`reinas-va1`$(sol, 1)$

```
proc reinas-va2(sol[1..n] de 1..n, e k : 1..n, éxito : bool)
    columna := 1
    mientras ¬éxito ∧ columna ≤ n hacer
        sol[k] := columna
        si no-jaque?(sol, k) entonces
            si k = n entonces
                éxito := cierto ; imprimir(sol)
            si no reinas-va2(sol, k + 1, éxito)
            fsi
        fsi
        columna := columna + 1
    fmientras
fproc
```

```
éxito := falso
reinas-va2(sol, 1, éxito)
```

Los renos de Papá Noel

Los renos de Papá Noel (Variaciones)

Dados n renos, calcular todos los posibles equipos con m renos para tirar del trineo de Papá Noel. La posición dentro del equipo es importante.

Numeramos los renos: $\{1,\ldots,n\}$.

Soluciones $(x_1, x_2, \ldots, x_m)$, donde x_i es el reno que ocupa la posición i-ésima dentro del equipo.

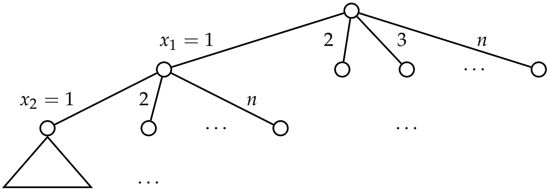
Restricciones explícitas utilizar renos válidos:

$$\forall\, i : 1 \leq i \leq m : x_i \in \{1,\ldots,n\}.$$

Restricciones implícitas que no haya renos repetidos:

$$\forall\, i,j : 1 \leq i,j \leq m : i \neq j \Rightarrow x_i \neq x_j.$$

Árbol de exploración, con m niveles:



```
proc variaciones-va1(e n : nat⁺, sol[1..m] de 1..n, e k : 1..m)
    para j = 1 hasta n hacer
        sol[k] := j
        si no-repetido?(sol, k) entonces
            si k = m entonces imprimir(sol)    { es una solución }
            si no variaciones-va1(n, sol, k+1)
            fsi
        fsi
    fpara
fproc
```

```
fun no-repetido?(sol[1..m] de nat, k : 1..n) dev respuesta : bool
    i := 1
    mientras sol[i] ≠ sol[k] hacer
        i := i+1
    fmientras
    respuesta := (i = k)
ffun
```

Ahorrar tiempo en el test de factibilidad asociando a cada nodo cierta cantidad de información correspondiente a "cálculos parciales" de dichos tests.

Marcadores: parámetros adicionales de entrada/salida (equivalen a variables globales) $\Rightarrow$ incremento del coste en espacio.

Esquema de vuelta atrás con marcadores

```
proc vuelta-atrás-marcadores(sol : tupla, e k : nat, m : marcador)
   preparar-recorrido-nivel(k)
   mientras ¬último-hijo-nivel(k) hacer
      sol[k] := siguiente-hijo-nivel(k)
      m := marcar(m, sol[k])
      si es-solución?(sol, k) entonces
         tratar-solución(sol)
      si no
         si es-completable?(sol, k, m) entonces
            vuelta-atrás-marcadores(sol, k + 1, m)
         fsi
      fsi
      m := desmarcar(m, sol[k])
   fmientras
fproc
```

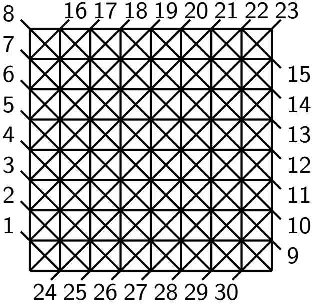
$$\forall i : 1 \le i \le n : usado[i] \Leftrightarrow i \text{ aparece en } sol[1..k].$$

proc variaciones-va2(**e** n : *nat*$^+$, $sol[1..m]$ **de** $1..n$, **e** $k : 1..m$, $usado[1..n]$ **de** *bool*)
  para $j = 1$ **hasta** n **hacer**
    si $\neg usado[j]$ **entonces**
      $sol[k] := j$
      $usado[j] :=$ cierto { marcar }
      si $k = m$ **entonces** imprimir(sol)
      si no variaciones-va2($n, sol, k+1, usado$)
      fsi
      $usado[j] :=$ falso { desmarcar }
    fsi
  fpara
fproc

proc variaciones(**e** n : *nat*$^+$)
var $sol[1..m]$ **de** $1..n$, $usado[1..n]$ **de** *bool*
  $usado[1..n] :=$ [falso]
  variaciones-va2($n, sol, 1, usado$)
fproc

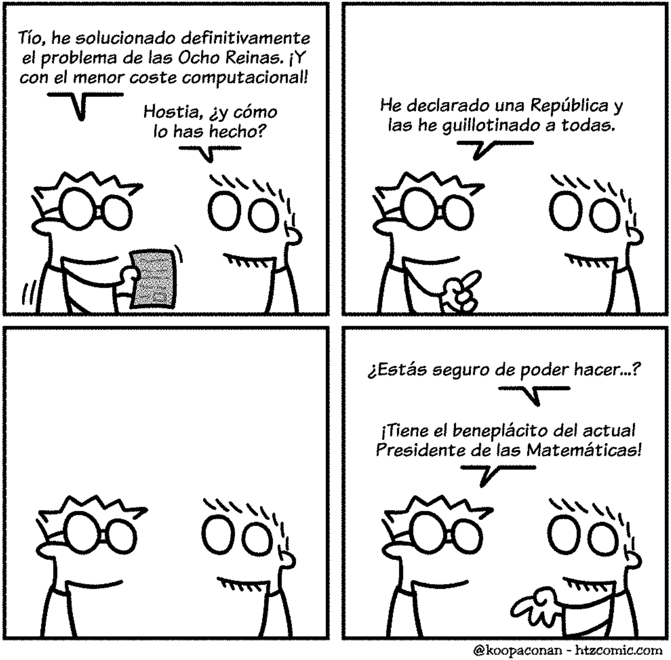
Cada posición en el tablero amenaza: 1 fila, 1 columna y 2 diagonales.

1. Un tablero con las posiciones amenazadas.
   Espacio $n \times n$
   Tiempo lineal respecto al tamaño del tablero (marcar las casillas)
   $\Rightarrow$ No hay mejora.
2. Dos vectores indicando las columnas y diagonales amenazadas.
   Numerar las diagonales:
   descendentes $\searrow$ de 1 a $2n-1$; ascendentes $\nearrow$ de $2n$ a $4n-2$.
   La reina en $\langle i, j \rangle$ amenaza la diagonal descendente $j - i + n$ y la diagonal ascendente $i + j + 2n - 2$.



```
proc reinas-va3(sol[1..n] de 1..n, e k : 1..n, C[1..n], D[1..4n − 2] de bool)
   para columna = 1 hasta n hacer
      sol[k] := columna
      si ¬C[sol[k]] ∧ ¬D[sol[k] − k + n] ∧ ¬D[k + sol[k] + 2n − 2] entonces
         { marcar }
         C[sol[k]] := cierto
         D[sol[k] − k + n] := cierto ; D[k + sol[k] + 2n − 2] := cierto
         si k = n entonces imprimir(sol)
         si no reinas-va3(sol, k + 1, C, D)
         fsi
         { desmarcar }
         C[sol[k]] := falso
         D[sol[k] − k + n] := falso ; D[k + sol[k] + 2n − 2] := falso
      fsi
   fpara
fproc

proc reinas(e n : nat⁺)
var sol[1..n] de 1..n, C[1..n], D[1..4n − 2] de bool
   C[1..n] := [falso] ; D[1..4n − 2] := [falso]
   reinas-va3(sol, 1, C, D)
fproc
```


Tío, he solucionado definitivamente el problema de las Ocho Reinas. ¡Y con el menor coste computacional!
Hostia, ¿y cómo lo has hecho?
He declarado una República y las he guillotinado a todas.
¿Estás seguro de poder hacer...?
¡Tiene el beneplácito del actual Presidente de las Matemáticas!
@koopaconan - htzcomic.com

Problema del viajante

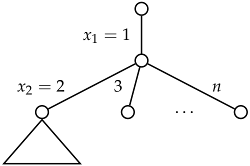
El representante de Rica-Cola tiene que controlar la venta de estos refrescos en n ciudades. Para ello, se ha informado sobre las posibles conexiones directas por ferrocarril entre las ciudades y desea conocer todos los circuitos en tren que recorran cada ciudad exactamente una vez y regresen a la ciudad de partida.

Encontrar los circuitos hamiltonianos en un grafo dirigido.

Soluciones $(x_1, \ldots, x_n)$, $x_i =$ vértice por el que se pasa en i-ésimo lugar.

- Utilizar vértices válidos, sin repeticiones y con arista de cada uno al siguiente, y con arista del último al primero.
- Evitar soluciones repetidas fijando el comienzo: $x_1 = 1$.

Árbol de exploración cada nodo, excepto la raíz, tiene $n-1$ hijos; n niveles. ?



proc `ciclo-hamiltoniano-va`(**e** G : *grafo[n]*, $sol[1..n]$ **de** $1..n$, **e** $k : 1..n$,
$usado[1..n]$ **de** *bool*)

```
   para vértice = 2 hasta n hacer
      si ¬usado[vértice] ∧ g-está-arista?(sol[k − 1], vértice, G) entonces
         sol[k] := vértice
         usado[vértice] := cierto  { marcar }
         si k = n entonces
            { falta comprobar que se cierra el ciclo }
            si g-está-arista?(sol[n], 1, G) entonces imprimir(sol) fsi
         si no ciclo-hamiltoniano-va(G, sol, k + 1, usado)
         fsi
         usado[vértice] := falso  { desmarcar }
      fsi
   fpara
fproc
```

```
proc ciclo-hamiltoniano(e G : grafo[n])
var sol[1..n] de 1..n, usado[1..n] de bool
   sol[1] := 1
   usado[1] := cierto ;  usado[2..n] := [falso]
   ciclo-hamiltoniano-va(G, sol, 2, usado)
ffun
```

- Características de los problemas de optimización para aplicar vuelta atrás:
  - solución expresable en forma de tupla: $(x_1, \ldots, x_n)$,
  - es posible determinar si una tupla es una solución factible,
  - es posible determinar si una tupla parcial puede ser completada hasta una solución factible.
- Almacenar la mejor solución encontrada hasta el momento.
- Almacenar también su valor asociado $\Rightarrow$ comparación más eficiente.
- Añadir como marcador el valor (parcial) de la tupla parcial $\Rightarrow$ facilitar el cálculo del valor de cada solución alcanzada.
- Mecanismo adicional de poda: cuando se puede asegurar que ninguno de los descendientes del nodo a expandir puede llegar a alcanzar una solución mejor que la mejor encontrada hasta ese momento.

Problema de minimización

Calcular una cota inferior (estimación) de la mejor solución alcanzable desde un nodo y podar si la estimación es ya mayor que el valor asociado a la mejor solución encontrada hasta el momento.

Esquema de vuelta atrás para optimización

proc `vuelta-atrás-opt`(sol : *tupla*, **e** k : *nat*,
    $valor$: *valor*, $sol\text{-}mejor$: *tupla*, $valor\text{-}mejor$: *valor*)
  `preparar-recorrido-nivel`(k)
  mientras $\neg$`último-hijo-nivel`(k) **hacer**
    $sol[k]$:= `siguiente-hijo-nivel`(k)
    $valor$:= `actualizar`($valor, sol, k$)
    si `es-solución?`(sol, k) **entonces**
      si `mejor`($valor, valor\text{-}mejor$) **entonces**
        $sol\text{-}mejor$:= sol ; $valor\text{-}mejor$:= $valor$
      fsi
    si no
      si `es-completable?`(sol, k)
        $\wedge$ `es-prometedor?`($sol, k, valor, valor\text{-}mejor$) **entonces**
        `vuelta-atrás-opt`($sol, k+1, valor, sol\text{-}mejor, valor\text{-}mejor$)
      fsi
    fsi
    $valor$:= `desactualizar`($valor, sol, k$)
  fmientras
fproc

Problema del viajante - Optimización

El representante de Rica-Cola se ha informado sobre las tarifas de conexión por tren entre cada par de ciudades y desea conocer un circuito en tren que recorra cada ciudad exactamente una vez y regrese a la ciudad de partida, y cuya tarifa total sea mínima.

Encontrar un circuito hamiltoniano de coste mínimo (grafo dirigido y valorado).

Guardar la mejor solución encontrada, junto con su coste correspondiente: $\langle$ *sol-mejor*, *coste-mejor* $\rangle$.

Marcador *coste* con el coste de la solución parcial (calcular de forma incremental).

Poda si para una solución parcial $\mathit{coste} \geq \mathit{coste\text{-}mejor}$.

Cota inferior el coste de las soluciones alcanzables desde $(x_1, \ldots, x_k)$ será

$$\underbrace{\sum_{i=2}^{k} \texttt{gv-valor}(x_{i-1}, x_i, G)}_{\text{fijo}} + \underbrace{\left(\sum_{i=k+1}^{n} \texttt{gv-valor}(x_{i-1}, x_i, G) \right) + \texttt{gv-valor}(x_n, x_1, G)}_{n-k+1 \text{ aristas}}$$

si $minG =$ valor mínimo de todas las aristas de G, entonces el coste de las últimas $n-k+1$ aristas se puede acotar con $(n-k+1) * minG$.

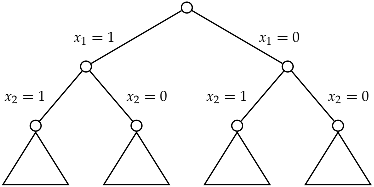
```
proc viajante-va(e G : grafo-val[n], e mínG : real, sol[1..n] de 1..n, e k : 1..n, coste : real,
                 usado[1..n] de bool, sol-mejor[1..n] de 1..n, coste-mejor : real∞)
  anterior := sol[k − 1]
  para vértice = 2 hasta n hacer
    si ¬usado[vértice] ∧ gv-está-arista?(anterior, vértice, G) entonces
      sol[k] := vértice
      usado[vértice] := cierto  { marcar }
      coste := coste + gv-valor(anterior, sol[k], G)
      si k = n entonces
        si gv-está-arista?(sol[n], 1, G) ∧c
          coste + gv-valor(sol[n], 1, G) < coste-mejor entonces
            sol-mejor := sol
            coste-mejor := coste + gv-valor(sol[n], 1, G)
        fsi
      si no   { k ≠ n }
        coste-estimado := coste + (n − k + 1) ∗ mínG
        si coste-estimado < coste-mejor entonces  { se puede mejorar sol-mejor }
          viajante-va(G, mínG, sol, k + 1, coste, usado, sol-mejor, coste-mejor)
        fsi
      fsi
      usado[vértice] := falso  { desmarcar }
      coste := coste − gv-valor(anterior, sol[k], G)
    fsi
  fpara
fproc
```

fun viajante$(G : \textit{grafo-val}[n])$ **dev** $\langle \textit{sol-mejor}[1..n]$ **de** $1..n, \textit{coste-mejor} : \textit{real}_\infty \rangle$
var $\textit{sol}[1..n]$ **de** $1..n, \textit{usado}[1..n]$ **de** *bool*
$\quad \textit{mínG} := \texttt{cálculo-mínimo}(G)$
$\quad \textit{sol}[1] := 1$
$\quad \textit{coste} := 0\ ;\ \textit{usado}[1] := \texttt{cierto}\ ;\ \textit{usado}[2..n] := [\texttt{falso}]$
$\quad \textit{coste-mejor} := +\infty$
$\quad \texttt{viajante-va}(G, \textit{mínG}, \textit{sol}, 2, \textit{coste}, \textit{usado}, \textit{sol-mejor}, \textit{coste-mejor})$
ffun

n objetos, con un peso $p_i > 0$ y un valor $v_i > 0$, y un peso total máximo $M > 0$.
Maximizar el valor total de los objetos metidos en la mochila.

Problema de la mochila: Espacio de soluciones y árbol de exploración

1. Etapa i-ésima: ¿Qué objeto meter (tras haber introducido $i-1$ objetos)?
$(x_1, x_2, \ldots, x_k)$ con $0 \leq k \leq n$, $x_i \in \{1, \ldots, n\}$ y $\forall i, j.(x_i \neq x_j)$ y $\sum_{i=1}^{k} p_{x_i} \leq M$.
$\Rightarrow$ Todos los nodos estado que lo verifiquen son nodos solución.
2. Etapa i-ésima: ¿Metemos el objeto i-ésimo en la mochila?
$(x_1, x_2, \ldots, x_n)$ con $x_i \in \{0, 1\}$ y $\sum_{i=1}^{n} x_i p_i \leq M$.
$\Rightarrow$ Árbol binario completo con los nodos solución solo en las hojas.



Objetivo Maximizar $\sum_{i=1}^{n} x_i v_i$.

Marcadores *peso* y *beneficio* (peso y beneficio de la solución parcial).

Poda no es factible (excede el peso máximo) o no interesa (cualquier extensión va a ser peor que la solución mejor actual).

Cota superior (de la mejor solución alcanzable) rellenar lo que se pueda con los objetos restantes y fraccionar algún objeto si fuera necesario.

fun `c-estimación`$(P[1..n], V[1..n]$ **de** *real*$^+$$, M :$ *real*$^+$$, k : 1..n, peso, beneficio :$ *real*$)$
dev $estimación$: *real*

$hueco := M - peso$
$estimación := beneficio$
$j := k + 1$
mientras $j \leq n \ \wedge \ P[j] \leq hueco$ **hacer**
{ podemos coger el objeto j entero }
$hueco := hueco - P[j]$
$estimación := estimación + V[j]$
$j := j + 1$
fmientras
si $j \leq n$ **entonces** { quedan objetos por probar }
{ fraccionamos el objeto j }
$estimación := estimación + (hueco/P[j]) * V[j]$
fsi
ffun

¿Interesa seleccionar los objetos en algún orden?

1. Objeto más valioso: incrementar el valor total lo más rápido posible.
2. Objeto más ligero: agotar el peso lentamente para que quepan más objetos.

$M = 10$, $n = 5$

p_i	1	2	3	4	5
v_i	20	30	66	40	60
$\frac{v_i}{p_i}$	2	1,5	2,2	1	1,2

Seleccionar	x_i					valor
máx v_i	0	0	1	0,5	1	146
mín p_i	1	1	1	1	0	156
máx $\frac{v_i}{p_i}$	1	1	1	0	0,8	164

1. Objetos muy valiosos pero muy pesados.
2. Objetos muy ligeros pero poco valiosos.

Solución: maximizar la relación valor por unidad de peso.

$\{ \frac{V[1]}{P[1]} \geq \frac{V[2]}{P[2]} \geq \ldots \geq \frac{V[n]}{P[n]} \}$

proc `mochila-va`(**e** $P[1..n], V[1..n]$ **de** *real*$^+$, **e** M : *real*, $sol[1..n]$ **de** $0..1$, **e** $k : 1..n$,
$\quad$ *peso*, *beneficio* : *real*, *sol-mejor*$[1..n]$ **de** $0..1$, *beneficio-mejor* : *real*)

$\quad$ { hijo izquierdo — coger objeto, no hacemos estimación }
$\quad$ $sol[k] := 1$
$\quad$ $peso := peso + P[k]$; $beneficio := beneficio + V[k]$ $\quad$ { marcar }
$\quad$ **si** $peso \leq M$ **entonces**
$\quad\quad$ **si** $k = n$ **entonces**
$\quad\quad\quad$ *sol-mejor* := *sol* ; *beneficio-mejor* := *beneficio*
$\quad\quad$ **si no**
$\quad\quad\quad$ `mochila-va`($P, V, M, sol, k+1$, *peso*, *beneficio*, *sol-mejor*, *beneficio-mejor*)
$\quad\quad$ **fsi**
$\quad$ **fsi**
$\quad$ $peso := peso - P[k]$; $beneficio := beneficio - V[k]$ $\quad$ { desmarcar }
$\quad$ { hijo derecho — no coger objeto, no se marca pero sí se hace estimación }
$\quad$ $sol[k] := 0$
$\quad$ *beneficio-estimado* := `c-estimación`(P, V, M, k, *peso*, *beneficio*)
$\quad$ **si** *beneficio-estimado* $>$ *beneficio-mejor* **entonces**
$\quad\quad$ **si** $k = n$ **entonces**
$\quad\quad\quad$ *sol-mejor* := *sol* ; *beneficio-mejor* := *beneficio*
$\quad\quad$ **si no**
$\quad\quad\quad$ `mochila-va`($P, V, M, sol, k+1$, *peso*, *beneficio*, *sol-mejor*, *beneficio-mejor*)
$\quad\quad$ **fsi**
$\quad$ **fsi**

fproc

fun `mochila-principal`$(P[1..n], V[1..n]$ **de** *real*$^+$$, M :$ *real*$^+)$
dev $\langle$ *sol-mejor*$[1..n]$ **de** $0.,1$, *beneficio-mejor* : *real* $\rangle$
var *sol*$[1..n]$ **de** $0.,1$
peso $:= 0$; *beneficio* $:= 0$
beneficio-mejor $:= -1$ { peor que cualquier solución }
`mochila-va`$(P, V, M,$ *sol*$, 1,$ *peso*$,$ *beneficio*$,$ *sol-mejor*$,$ *beneficio-mejor*$)$
ffun