Métodos de exploración exhaustiva

Yolanda Ortega Mallén

Dpto. de Sistemas Informáticos y Programación

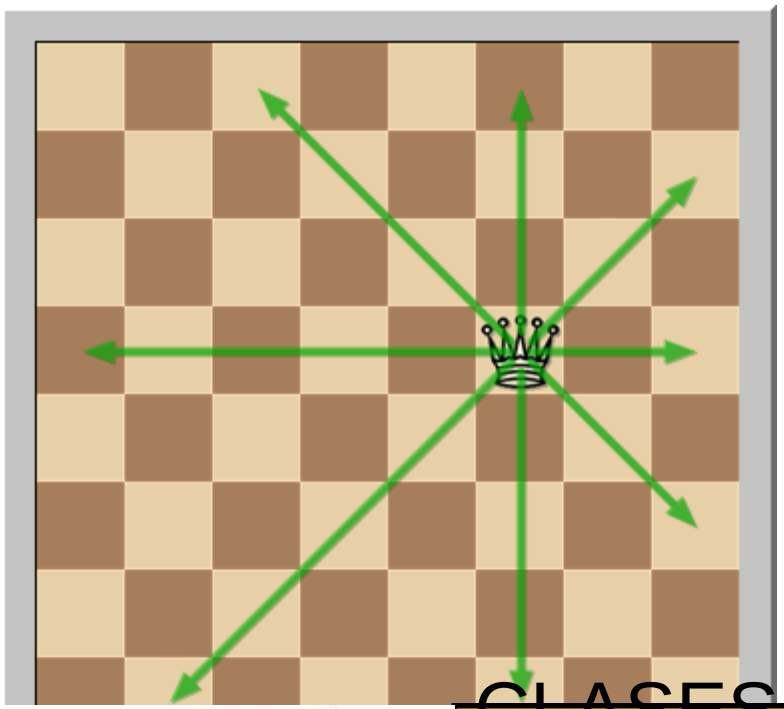
Universidad Complutense de Madrid

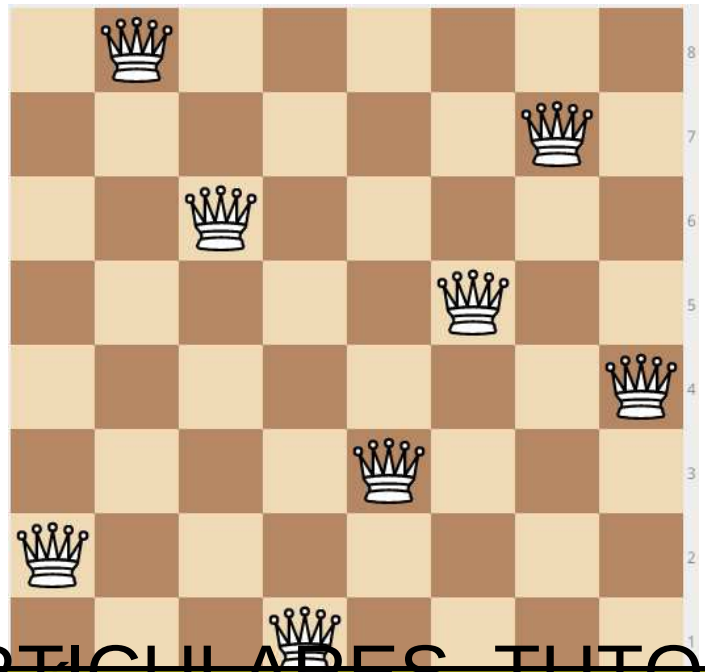
Sumario

- Un ejemplo: el problema de las ocho 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.
- Ramificación y poda.
  - Esquema general.
  - Esquema *optimista/pesimista*.

Problema de las ocho reinas

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





Problema de las ocho reinas

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 [illegible]

Exploración exhaustiva

- No siempre se pueden utilizar las técnicas vistas hasta ahora para lograr 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.

Espacio de soluciones

- 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 ocho reinas

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

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

Árbol de exploración

- 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;

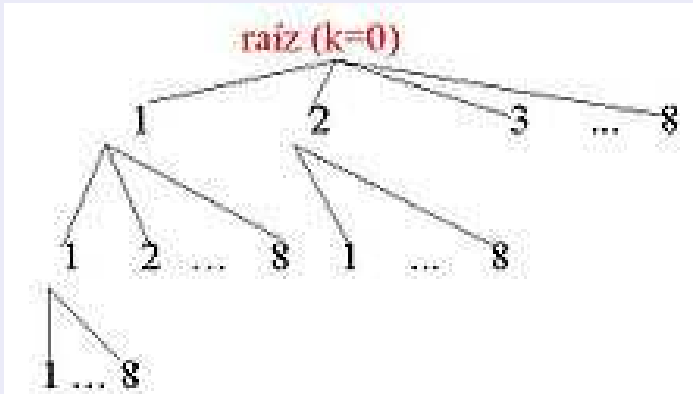
Nodo solución correspondiente a una tupla completa que satisface las restricciones explícitas e implícitas.

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.



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 muy grande.
- Difícil analizar teóricamente a priori: tomar medidas empíricas.

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 : tupla, e k : nat)
    preparar-recorrido-nivel(k)
    mientras ¬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
```

Los nodos solución están solo en las hojas.

Problema de las n reinas

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

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

Problema de las n reinas: Primera solución

```
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
```

Alineaciones de fútbol

Dado un equipo de fútbol con n jugadores, y suponiendo que todos pueden jugar en cualquier posición, calcular todas las posibles alineaciones del equipo.

Alineaciones de fútbol (Variaciones)

Generalizamos a alineaciones de m posiciones.

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

Soluciones $(x_1, x_2, \ldots, x_m)$, donde x_i es el jugador que ocupa la posición i-ésima de la alineación.

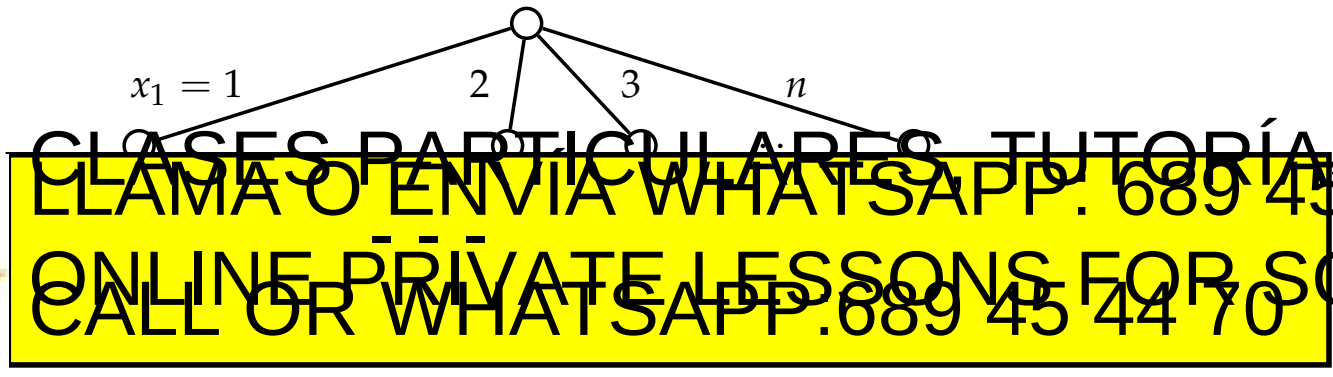
Restricciones explícitas utilizar jugadores válidos:

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

Restricciones implícitas que no haya jugadores 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:



Variaciones

```
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
```

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
    fmientras
fproc
```

Variaciones con marcadores

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

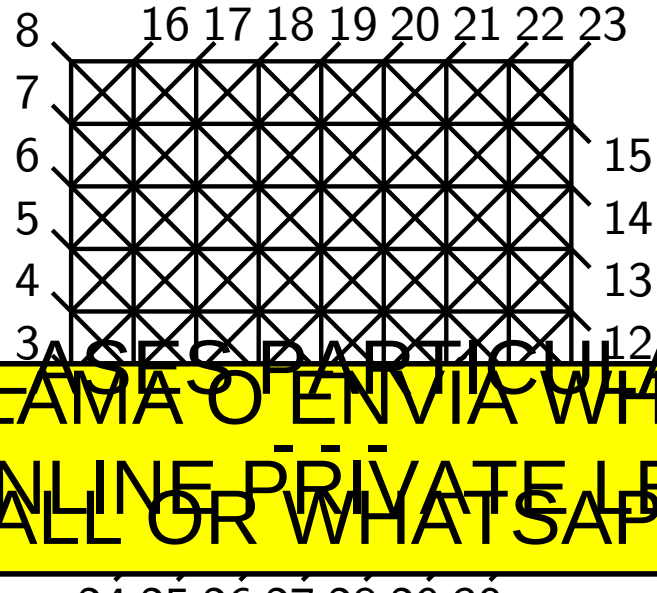
```
proc variaciones-va2(e n : nat+, sol[1..m] de 1..n, e k : 1..m, usado[1..n] de bool)
    para j = 1 hasta n hacer
        si ¬usado[j] entonces
            sol[k] := j
            usado[j] := cierto    { marcar }
            si k = m entonces imprimir(sol)
            si no variaciones-va2(n, sol,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

proc variaciones(e n : nat+)
```

Problema de las n reinas con marcadores

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
```

Problema del viajante

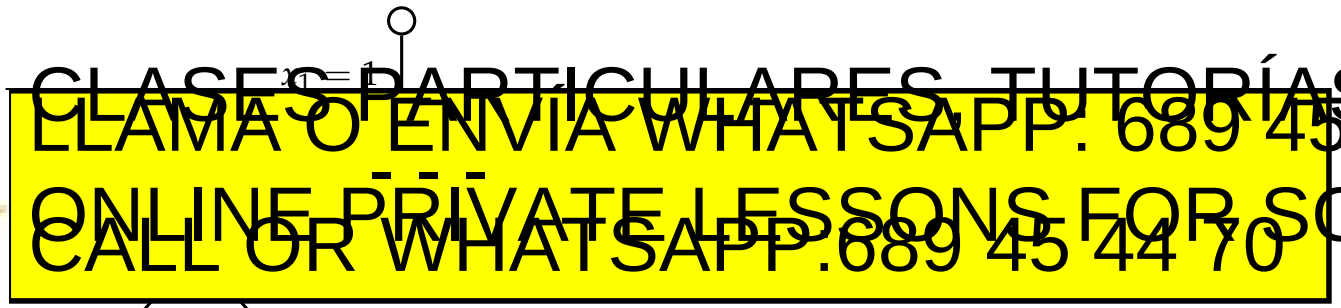
El representante de Rica-Cola tiene que controlar la venta de estos refrescos en n ciudades. Para ello, se ha informado sobre las posibles conexiones directas por ferrocarril entre las ciudades y desea conocer todos los circuitos en tren que recorran cada ciudad exactamente una vez y regresen a la ciudad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

Vuelta atrás y optimización

- 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

Esquema de vuelta atrás para optimización

```
proc vuelta-atrás-opt(sol : tupla, e k : nat,
                      valor : valor, sol-mejor : tupla, valor-mejor : valor)
    preparar-recorrido-nivel(k)
    mientras ¬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-mejor) entonces
                sol-mejor := sol ; valor-mejor := valor
            fsi
        si no
            si es-completable?(sol, k)
                ∧ es-prometedor?(sol, k, valor, valor-mejor) entonces
                vuelta-atrás-opt(sol, k + 1, valor, sol-mejor, valor-mejor)
            fsi
        fsi
```

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: $\langle \textit{sol-mejor}, \textit{coste-mejor} \rangle$.

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

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

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

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

últimas $n - k$ + 1 aristas se puede acotar con $(n - k + 1) * minG$.

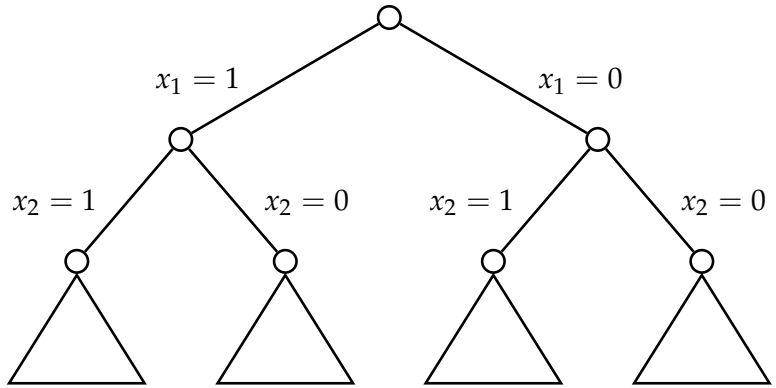
```
proc viajante-va(e G : grafo-val[n], e mínG : real, sol[1..n] de 1..n, e k : 1..n, coste : real,
                 usado[1..n] de bool, sol-mejor[1..n] de 1..n, coste-mejor : real∞)
    anterior := sol[k − 1]
    para vértice = 2 hasta n hacer
        si ¬usado[vértice] ∧ gv-está-arista?(anterior, 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
```

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*
    $\textit{mínG} :=$ `cálculo-mínimo`(G)
    $\textit{sol}[1] := 1$
    $\textit{coste} := 0$; $\textit{usado}[1] :=$ `cierto` ; $\textit{usado}[2..n] := [$`falso`$]$
    $\textit{coste-mejor} := +\infty$
    `viajante-va`$(G, \textit{mínG}, \textit{sol}, 2, \textit{coste}, \textit{usado}, \textit{sol-mejor}, \textit{coste-mejor})$
ffun

Problema de la mochila (versión entera)

n objetos, con un peso $p_i > 0$ y un valor $v_i > 0$, y un peso total máximo $M > 0$.

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.



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

$\Rightarrow$ objetos ordenados decrecientemente por valor por unidad de peso

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

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

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 (solución voraz) }
$estimación := estimación + (hueco / P[j]) * V[j]$
fsi
ffun

fun `mochila-principal`$(P[1..n], V[1..n]$ **de** $real^+, M : real^+)$
dev $\langle sol\text{-}mejor[1..n]$ **de** $0.,1, beneficio\text{-}mejor : real \rangle$

var $sol[1..n]$ **de** $0.,1$

Ramificación y poda (Branch & Bound)

- Gestión de los nodos vivos mediante una cola con prioridad, expandiendo en cada momento el más prometedor.
- Necesario si existen ramas de profundidad no acotada.
- Problemas de optimización: se espera encontrar la solución óptima de forma más rápida que con vuelta atrás.
- Función `valor-estimado`: dada una tupla parcial $X = (x_1, \ldots, x_k)$, proporciona una cota del valor de la mejor solución alcanzable desde X.

  Minimización:

  $\texttt{coste-real}(X) =$ coste de la mejor solución alcanzable desde X,

  $$\texttt{coste-estimado}(X) \leq \texttt{coste-real}(X)$$

  Maximización: $\texttt{beneficio-real}(X) =$ beneficio de la mejor solución alcanzable desde X,

  $$\texttt{beneficio-estimado}(X) \geq \texttt{beneficio-real}(X)$$

esa rama del arbol.

Esquema general de ramificación y poda (minimización)

```
fun ramificación-y-poda-mín(T : árbol-de-estados)
     dev ⟨ sol-mejor : tupla, coste-mejor : valor ⟩
var X, Y : nodo, C : colapr[nodo]
    Y := raíz(T)
    C := cp-vacía() ; añadir(C, Y)
    coste-mejor := +∞
    mientras ¬es-cp-vacía?(C) ∧ c-estimado(mínimo(C)) < coste-mejor hacer
        Y := mínimo(C) ; eliminar-mín(C)
        para todo hijo X de Y hacer
            si es-solución?(X) entonces
                si coste-real(X) < coste-mejor entonces
                    coste-mejor := coste-real(X) ; sol-mejor := solución(X)
                fsi
            si no
                si es-completable?(X) ∧ c-estimado(X) < coste-mejor entonces
                    añadir(C, X)
                fsi
            fsi
        fpara
```

- El orden de generación de los hijos no influye.

Problema del viajante

tipos
  nodo = **reg**
    $sol[1..n]$ **de** $1..n$
    $k : 1..n$
    coste : *real*
    coste-estimado : *real* { prioridad }
    $usado[1..n]$ **de** *bool* { marcador }
  freg
ftipos

fun `viajante-rp`$(G$: *grafo-val*$[n])$ **dev** $\langle$ *sol-mejor*$[1..n]$ **de** $1..n$, *coste-mejor* : $real_\infty \rangle$
var X, Y : *nodo*, C : *colapr*[*nodo*]
  mínG := `cálculo-mínimo`(G)
  { generamos la raíz }
  $Y.k := 1$; $Y.sol[1] := 1$

mientras $\neg$ `es-cp-vacía?`(C) $\wedge$ `mínimo`(C).*coste-estimado* $<$ *coste-mejor* **hacer**
  Y := `mínimo`(C) ; `eliminar-mín`(C)
  { generamos los hijos de Y }
  $X.k := Y.k + 1$; $X.sol := Y.sol$; $X.usado := Y.usado$
  $anterior := X.sol[X.k - 1]$
  para *vértice* $= 2$ **hasta** n **hacer**
    si $\neg X.usado[vértice]$ $\wedge$ `gv-está-arista?`$(anterior, vértice, G)$ **entonces**
      $X.sol[X.k] := vértice$; $X.usado[vértice] :=$ `cierto` { marcar }
      $X.coste := Y.coste +$ `gv-valor`$(anterior, vértice, G)$
      si $X.k = n$ **entonces**
        si `gv-está-arista?`$(sol[n], 1, G)$ $\wedge_c$
          $X.coste +$ `gv-valor`$(X.sol[n], 1, G) <$ *coste-mejor* **entonces**
          sol-mejor := $X.sol$; *coste-mejor* := $X.coste +$ `gv-valor`$(X.sol[n], 1, G)$
        fsi
      si no
        $X.coste\text{-}estimado := X.coste + (n - X.k + 1) * mínG$
        si $X.coste\text{-}estimado <$ *coste-mejor* **entonces**
          `añadir`(C, X)
        fsi
      fsi
      $X.usado[$ [illegible] $]$:= falso
fin

Ramificación y poda: Esquema optimista/pesimista (minimización)

coste-mejor almacena el coste de la mejor solución obtenida hasta el momento.

- Solo se modifica al encontrar una solución mejor.
- Al principio tiene el peor valor posible ($+\infty$).
- Poda no efectiva hasta encontrar una solución. Después mejora despacio.

Poda más efectiva si se dispone de una función que para un nodo X calcule una cota superior del coste de la mejor solución alcanzable desde X.
Actualizar *coste-mejor* cuando se encuentre un nodo factible con una cota superior menor que el valor actual de *coste-mejor*.

$\texttt{coste-optimista}(X)$	$=$	cota inferior del coste de la mejor solución alcanzable desde X (antes $\texttt{coste-estimado}(X)$).
$\texttt{coste-pesimista}(X)$	$=$	cota superior del coste de la mejor solución alcanzable desde X.

$$\texttt{coste-optimista}(X) \leq \texttt{coste-real}(X) \leq \texttt{coste-pesimista}(X).$$

Poda: Si $\texttt{coste-optimista}(X) \geq$ *coste-mejor* no explorar X.

No necesariamente se tiene que cumplir que $\texttt{coste-real}(X) \leq \texttt{coste-pesimista}(X)$.

Esquema optimista/pesimista de ramificación y poda (minimización)

```
fun rp-opt-pes-mín(T : árbol-de-estados) dev ⟨sol-mejor : tupla, coste-mejor : valor⟩
var X, Y : nodo, C : colapr[nodo]
    Y := raíz(T) ; C := cp-vacía() ; añadir(cola, Y)
    coste-mejor := coste-pesimista(Y)
    mientras ¬es-cp-vacía?(C) ∧ c-optimista(mínimo(C)) ≤ coste-mejor hacer
        Y := mínimo(C) ; eliminar-mín(C)
        para todo hijo X de Y hacer
            si es-solución?(X) entonces
                si coste-real(X) ≤ coste-mejor entonces
                    coste-mejor := coste-real(X) ; sol-mejor := X
                fsi
            si no
                si es-completable?(X) ∧ c-optimista(X) ≤ coste-mejor entonces
                    añadir(C, X)
                    si coste-pesimista(X) < coste-mejor entonces
                        coste-mejor := coste-pesimista(X)
                    fsi
                fsi
```

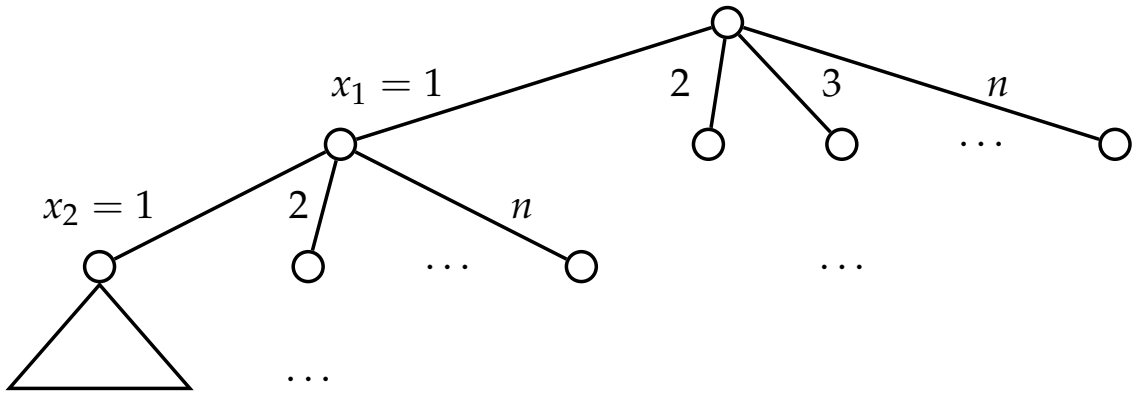
Haciendo trabajar al Ministerio

El Ministro de Hacienda-somos-todos se ha propuesto hacer trabajar en firme a los n funcionarios de su Ministerio y se ha sacado de la manga n trabajos.
A pesar de su tradicional (pero infundada) ineficacia, todos los funcionarios son capaces de hacer cualquier trabajo, aunque unos tardan más que otros.
La información al respecto se recoge en la tabla $T[1..n, 1..n]$, donde $T[i,j]$ representa el tiempo que el funcionario i tarda en realizar el trabajo j.
Su Excelencia el Sr. Ministro desea conocer la asignación óptima de trabajos a funcionarios de modo que la suma total de tiempos sea mínima.

Haciendo trabajar al Ministerio

Soluciones $(x_1, x_2, \ldots, x_n)$ donde x_i = trabajo asignado al funcionario i.
Para ser factibles deben ser permutaciones de los n trabajos.

Árbol de exploración hay n niveles y cada nodo tiene n hijos.



Marcador llevar cuenta de los trabajos ya asignados en un vector $asignado[1..n]$ de booleanos.

Haciendo trabajar al Ministerio: Cota optimista

¿Cota inferior del coste total a partir del coste de una solución parcial?

Para $(x_1, \ldots, x_k)$, el tiempo hasta el momento es $tiempo = \sum_{i=1}^{k} T[i, x_i]$, y hay que estimar el tiempo del resto de la solución.

1. Opción más sencilla (y más optimista): aproximar con 0 y utilizar *tiempo* como estimación.
2. Calcular un mínimo global de la matriz T,

$$mínT = \min\{T[i,j] \mid 1 \leq i \leq n \ \wedge \ 1 \leq j \leq n\},$$

que sirve como cota inferior del tiempo de realización de cada trabajo por los funcionarios $\Rightarrow (n-k)\, mínT \leq$ el tiempo del resto de la solución.

3. Tener calculado un mínimo por cada fila: para cada funcionario, cuánto tarda en realizar el trabajo que realiza más rápido:

$$rápido[i] = \min\{T[i,j] \mid 1 \leq j \leq n\}$$

$\Rightarrow \sum_{i=k+1}^{n} rápido[i] \leq$ el tiempo del resto de la solución.

4. Calcular *rápido* dinámicamente entre los trabajos no repartidos todavía:

Haciendo trabajar al Ministerio: Cota pesimista

1. Derivar una solución cualquiera a partir de la solución parcial, asignando trabajos libres a los funcionarios restantes siguiendo el orden establecido.
2. Calcular el máximo global de la matriz T,

$$máxT = \text{máx}\{T[i,j] \mid 1 \leq i \leq n \ \wedge \ 1 \leq j \leq n\},$$

   $\Rightarrow (n-k)\, máxT \geq$ el tiempo del resto de la solución.
3. Se puede mejorar calculando un máximo por cada fila:

$$lento[i] = \text{máx}\{T[i,j] \mid 1 \leq j \leq n\}$$

   $\Rightarrow \sum_{i=k+1}^{n} lento[i] \geq$ el tiempo del resto de la solución.

   Calcularemos inicialmente las sumas

$$pes[k] = \sum_{i=k+1}^{n} lento[i].$$

Haciendo trabajar al Ministerio: Implementación

tipos
  nodo = **reg**
    $sol[1..n]$ **de** $1..n$
    $k : 0..n$
    tiempo : *real*
    tiempo-opt : *real* { prioridad }
    $asignado[1..n]$ **de** *bool*
  freg
ftipos

fun `funcionarios-mín-rp`$(T[1..n, 1..n]$ **de** $real^+)$
  dev $\langle$ *sol-mejor*$[1..n]$ **de** $1..n$, *tiempo-mejor* : *real* $\rangle$
var X, Y : *nodo*, C : *colapr[nodo]*, $opt[0..n], pes[0..n]$ **de** *real*
  $\langle opt, pes \rangle$:= `pre-cálculo-estim`(T)
  { generamos la raíz }

mientras $\neg$`es-cp-vacía?`(C) $\wedge$ `mínimo`$(C).$*tiempo-opt* $\leq$ *tiempo-mejor* **hacer**
  $Y :=$ `mínimo`(C) ; `eliminar-mín`(C)
  { generamos los hijos de Y }
  $X.k := Y.k + 1$; $X.sol := Y.sol$; $X.asignado := Y.asignado$
  para $t = 1$ **hasta** n **hacer**
    si $\neg X.asignado[t]$ **entonces**
      $X.sol[X.k] := t$; $X.asignado[t] :=$ `cierto`
      $X.tiempo := Y.tiempo + T[X.k, t]$
      $X.tiempo\text{-}opt := X.tiempo + opt[X.k]$
      si $X.tiempo\text{-}opt \leq tiempo\text{-}mejor$ **entonces**
        si $X.k = n$ **entonces**
          $sol\text{-}mejor := X.sol$; $tiempo\text{-}mejor := X.tiempo$
        si no
          `añadir`(C, X)
          $tiempo\text{-}mejor :=$ `mín`$(tiempo\text{-}mejor, X.tiempo + pes[X.k])$
        fsi
      fsi
      $X.asignado[t] :=$ `falso`
  fsi

Haciendo trabajar al Ministerio: Implementación

fun `pre-cálculo-estim`$(T[1..n,1..n]$ **de** *real*$^+)$ **dev** $\langle opt[0..n], pes[0..n]$ **de** *real*$\rangle$
var $rápido[1..n], lento[1..n]$ **de** *real*
  { cálculo de los mínimos y máximos por filas }
  para $i = 1$ **hasta** n **hacer**
    $rápido[i] := T[i,1]$
    $lento[i] := T[i,1]$
    para $j = 2$ **hasta** n **hacer**
      $rápido[i] :=$ `mín`$(rápido[i], T[i,j])$
      $lento[i] :=$ `máx`$(lento[i], T[i,j])$
    fpara
  fpara
  { cálculo de las estimaciones }
  $opt[n] := 0$; $pes[n] := 0$
  para $i = n-1$ **hasta** 0 **paso** -1 **hacer**
    $opt[i] := opt[i+1] + rápido[i+1]$
    $pes[i] := pes[i+1] + lento[i+1]$
  f

Problema de la mochila (versión entera): Cotas

Cota optimista Utilizar el algoritmo voraz que resolvía el problema cuando los objetos se podían fraccionar $(0 \leq x_i \leq 1)$.
Necesitamos los objetos en orden decreciente de valor por unidad de peso, v_i/p_i.

Cota pesimista

1. Una cota inferior es el valor de los objetos que ya se han cogido.
2. Mejor probar una posible solución: incorporar a la mochila todos los objetos restantes que se pueda, considerándolos en el orden establecido.
   Para un nodo en el que el último objeto considerado se ha metido en la mochila, la cota pesimista coincide con la de su padre
   $\Rightarrow$ no se podrá mejorar *beneficio-mejor*.

Problema de la mochila: Implementación

tipos
  nodo = **reg**
    $sol[1..n]$ **de** $0..1$
    $k : 0..n$
    peso, *beneficio* : *real*
    beneficio-opt : *real* { prioridad }
  freg
ftipos

$\{ \frac{V[1]}{P[1]} \geq \frac{V[2]}{P[2]} \geq \ldots \geq \frac{V[n]}{P[n]} \}$
fun `mochila-rp`$(P[1..n], V[1..n]$ **de** *real*$^+, M$: *real*$^+)$
  dev $\langle sol\text{-}mejor[1..n]$ **de** $0..1, beneficio\text{-}mejor$: *real*$\rangle$
var X, Y : *nodo*, C : *colapr[nodo]*
  { generamos la raíz }
  $Y.k := 0$; $Y.peso := 0$; $Y.beneficio := 0$; ...

mientras $\neg$`es-cp-vacía?`$(C) \wedge$ `máximo`$(C).beneficio\text{-}opt \geq beneficio\text{-}mejor$ **hacer**
$Y :=$ `máximo`(C) ; `eliminar-máx`(C)
$X.k := Y.k + 1$; $X.sol := Y.sol$
{ probamos a meter el objeto en la mochila }
si $Y.peso + P[X.k] \leq M$ **entonces**
{ es factible y, por tanto, las estimaciones coinciden con las de Y }
{ $beneficio\text{-}opt(X) = beneficio\text{-}opt(Y) \geq beneficio\text{-}mejor$ }
$X.sol[X.k] := 1$; $X.peso := Y.peso + P[X.k]$
$X.beneficio := Y.beneficio + V[X.k]$; $X.beneficio\text{-}opt := Y.beneficio\text{-}opt$
si $X.k = n$ **entonces** { $beneficio(X) = beneficio\text{-}opt(X) \geq beneficio\text{-}mejor$ }
$sol\text{-}mejor := X.sol$; $beneficio\text{-}mejor := X.beneficio$
si no `añadir`(C, X) { no se puede mejorar $beneficio\text{-}mejor$ }
fsi
fsi
{ probamos a no meter el objeto (siempre es factible) }
$\langle X.beneficio\text{-}opt, pes \rangle :=$ `cálculo-estimaciones`$(P, V, M, X.k, Y.peso, Y.beneficio)$
si $X.beneficio\text{-}opt \geq beneficio\text{-}mejor$ **entonces**
$X.sol[X.k] := 0$; $X.peso := Y.peso$; $X.beneficio := Y.beneficio$
si $X.k = n$ **entonces**
$sol\text{-}mejor := X.sol$; $beneficio\text{-}mejor := X.beneficio$
si no
fsi
mientras

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

fun `cálculo-estim`$(P[1..n], V[1..n]$ **de** $real^+, M : real^+, k : 0..n, peso, beneficio : real)$

dev $\langle opt, pes : real \rangle$

$hueco := M - peso$; $pes := beneficio$; $opt := beneficio$

$j := k+1$

mientras $j \leq n \ \wedge \ P[j] \leq hueco$ **hacer**

{ podemos coger el objeto j entero }

$hueco := hueco - P[j]$; $opt := opt + V[j]$; $pes := pes + V[j]$

$j := j+1$

fmientras

si $j \leq n$ **entonces** { quedan objetos por probar y $P[j] > hueco$ }

{ fraccionamos el objeto j (solución voraz) }

$opt := opt + (hueco / P[j]) * V[j]$

{ extendemos a una solución en la versión 0/1 }

$j := j+1$

mientras $j \leq n \ \wedge \ hueco > 0$ **hacer**

si $P[j] \leq hueco$ **entonces**

$hueco := hueco - P[j]$; $pes := pes + V[j]$

fsi

...