Tema 23: Técnicas de diseño descendente de algoritmos

Informática (2010–11)

José A. Alonso Jiménez

Grupo de Lógica Computacional
Departamento de Ciencias de la Computación e I.A.
Universidad de Sevilla

Tema 23: Técnicas de diseño descendente de algoritmos

Tema 23: Técnicas de diseño descendente de algoritmos

La técnica de divide y vencerás

La técnica **divide y vencerás** consta de los siguientes pasos:

1. *Dividir* el problema en subproblemas menores.
2. *Resolver* por separado cada uno de los subproblemas:
    - si los subproblemas son complejos, usar la misma técnica recursivamente;
    - si son simples, resolverlos directamente.
3. *Combinar* todas las soluciones de los subproblemas en una solución simple.

La técnica de divide y vencerás

- `(divideVenceras ind resuelve divide combina pbInicial)` resuelve el problema `pbInicial` mediante la técnica de divide y vencerás, donde
  - `(ind pb)` se verifica si el problema `pb` es indivisible,
  - `(resuelve pb)` es la solución del problema indivisible `pb`,
  - `(divide pb)` es la lista de subproblemas de `pb`,
  - `(combina pb ss)` es la combinación de las soluciones `ss` de los subproblemas del problema `pb` y
  - `pbInicial` es el problema inicial.

```
divideVenceras :: (p -> Bool) -> (p -> s) -> (p -> [p])
                  -> (p -> [s] -> s) -> p -> s
divideVenceras ind resuelve divide combina pbInicial =
    dv' pbInicial where
    dv' pb
        | ind pb    = resuelve pb
        | otherwise = combina pb [dv' sp | sp <- divide pb]
```

La técnica de divide y vencerás

- `(divideVenceras ind resuelve divide combina pbInicial)` resuelve el problema `pbInicial` mediante la técnica de divide y vencerás, donde
  - `(ind pb)` se verifica si el problema `pb` es indivisible,
  - `(resuelve pb)` es la solución del problema indivisible `pb`,
  - `(divide pb)` es la lista de subproblemas de `pb`,
  - `(combina pb ss)` es la combinación de las soluciones `ss` de los subproblemas del problema `pb` y
  - `pbInicial` es el problema inicial.

```haskell
divideVenceras :: (p -> Bool) -> (p -> s) -> (p -> [p])
                  -> (p -> [s] -> s) -> p -> s
divideVenceras ind resuelve divide combina pbInicial =
    dv' pbInicial where
    dv' pb
        | ind pb    = resuelve pb
        | otherwise = combina pb [dv' sp | sp <- divide pb]
```

Tema 23: Técnicas de diseño descendente de algoritmos

La ordenación por mezcla como ejemplo de DyV

- (ordenaPorMezcla xs) es la lista obtenida ordenando xs por el procedimiento de ordenación por mezcla. Por ejemplo,

```
ghci> ordenaPorMezcla [3,1,4,1,5,9,2,8]
[1,1,2,3,4,5,8,9]
```

```
ordenaPorMezcla :: Ord a => [a] -> [a]
ordenaPorMezcla xs =
    divideVenceras ind id divide combina xs
    where
      ind xs              = length xs <= 1
      divide xs           = [take n xs, drop n xs]
                            where n = length xs `div` 2
      combina _ [l1,l2] = mezcla l1 l2
```

La ordenación por mezcla como ejemplo de DyV

- (ordenaPorMezcla xs) es la lista obtenida ordenando xs por el procedimiento de ordenación por mezcla. Por ejemplo,

```
ghci> ordenaPorMezcla [3,1,4,1,5,9,2,8]
[1,1,2,3,4,5,8,9]
```

```haskell
ordenaPorMezcla :: Ord a => [a] -> [a]
ordenaPorMezcla xs =
    divideVenceras ind id divide combina xs
    where
      ind xs            = length xs <= 1
      divide xs         = [take n xs, drop n xs]
                          where n = length xs `div` 2
      combina _ [l1,l2] = mezcla l1 l2
```

La ordenación por mezcla como ejemplo de DyV

- (mezcla xs ys) es la lista obtenida mezclando xs e ys. Por ejemplo,

```
mezcla [1,3] [2,4,6]  ⇝  [1,2,3,4,6]
```

```
mezcla :: Ord a => [a] -> [a] -> [a]
mezcla [] b = b
mezcla a [] = a
mezcla a@(x:xs) b@(y:ys)
    | x <= y    = x : (mezcla xs b)
    | otherwise = y : (mezcla a ys)
```

La ordenación por mezcla como ejemplo de DyV

- `(mezcla xs ys)` es la lista obtenida mezclando `xs` e `ys`. Por ejemplo,

```
mezcla [1,3] [2,4,6]  ⇝  [1,2,3,4,6]
```

```haskell
mezcla :: Ord a => [a] -> [a] -> [a]
mezcla [] b = b
mezcla a [] = a
mezcla a@(x:xs) b@(y:ys)
    | x <= y     = x : (mezcla xs b)
    | otherwise = y : (mezcla a ys)
```

Tema 23: Técnicas de diseño descendente de algoritmos

1. La técnica de divide y vencerás
   - La técnica de divide y vencerás
   - La ordenación por mezcla como ejemplo de DyV
   - La ordenación rápida como ejemplo de DyV
2. Búsqueda en espacios de estados
3. Búsqueda por primero el mejor
4. Búsqueda en escalada

La ordenación rápida como ejemplo de DyV

- `(ordenaRapida xs)` es la lista obtenida ordenando `xs` por el procedimiento de ordenación rápida. Por ejemplo,

```
ghci> ordenaRapida [3,1,4,1,5,9,2,8]
[1,1,2,3,4,5,8,9]
```

```
ordenaRapida :: Ord a => [a] -> [a]
ordenaRapida xs =
    divideVenceras ind id divide combina xs
    where
      ind xs                 = length xs <= 1
      divide (x:xs)          = [[ y | y <- xs, y <= x],
                                [ y | y <- xs, y > x]]
      combina (x:_) [l1,l2] = l1 ++ [x] ++ l2
```

La ordenación rápida como ejemplo de DyV

- `(ordenaRapida xs)` es la lista obtenida ordenando `xs` por el procedimiento de ordenación rápida. Por ejemplo,

```
ghci> ordenaRapida [3,1,4,1,5,9,2,8]
[1,1,2,3,4,5,8,9]
```

```
ordenaRapida :: Ord a => [a] -> [a]
ordenaRapida xs =
    divideVenceras ind id divide combina xs
    where
      ind xs                = length xs <= 1
      divide (x:xs)         = [[ y | y <- xs, y <= x],
                                [ y | y <- xs, y > x]]
      combina (x:_) [l1,l2] = l1 ++ [x] ++ l2
```

Tema 23: Técnicas de diseño descendente de algoritmos

Descripción de los problemas de espacios de estados

Las características de los problemas de espacios de estados son:

- un conjunto de las posibles situaciones o **nodos** que constituye el **espacio de estados** (estos son las potenciales soluciones que se necesitan explorar),
- un conjunto de movimientos de un nodo a otros nodos, llamados los **sucesores** del nodo,
- un **nodo inicial** y
- un **nodo objetivo** que es la solución.

Descripción de los problemas de espacios de estados

- En estos problemas usaremos las siguientes librerías auxiliares:

```
-- Nota: Hay que elegir una implementación de las pilas.
import PilaConListas
-- import PilaConTipoDeDatoAlgebraico

import Data.Array
import Data.List (sort)
```

El patrón de búsqueda en espacios de estados

- Se supone que el grafo implícito de espacios de estados es acíclico.
- `(buscaEE s o e)` es la lista de soluciones del problema de espacio de estado definido por la función sucesores (`s`), el objetivo (`o`) y estado inicial (`e`).

```
buscaEE:: (Eq node) => (node -> [node])  -> (node -> Bool)
                    -> node -> [node]
buscaEE sucesores esFinal x = busca' (apila x vacia)
 where busca' p
    | esVacia p         = []
    | esFinal (cima p) = cima p : busca' (desapila p)
    | otherwise        = busca' (foldr apila (desapila p)
                                            (sucesores x))
                         where x = cima p
```

El patrón de búsqueda en espacios de estados

- Se supone que el grafo implícito de espacios de estados es acíclico.
- `(buscaEE s o e)` es la lista de soluciones del problema de espacio de estado definido por la función sucesores (`s`), el objetivo (`o`) y estado inicial (`e`).

```
buscaEE:: (Eq node) => (node -> [node])  -> (node -> Bool)
                      -> node -> [node]
buscaEE sucesores esFinal x = busca' (apila x vacia)
 where busca' p
    | esVacia p        = []
    | esFinal (cima p) = cima p : busca' (desapila p)
    | otherwise        = busca' (foldr apila (desapila p)
                                      (sucesores x))
                         where x = cima p
```

Tema 23: Técnicas de diseño descendente de algoritmos

El problema del 8 puzzle

Para el 8–puzzle se usa un cajón cuadrado en el que hay situados 8 bloques cuadrados. El cuadrado restante está sin rellenar. Cada bloque tiene un número. Un bloque adyacente al hueco puede deslizarse hacia él. El juego consiste en transformar la posición inicial en la posición final mediante el deslizamiento de los bloques. En particular, consideramos el estado inicial y final siguientes:

```
+---+---+---+                   +---+---+---+
| 2 | 6 | 3 |                   | 1 | 2 | 3 |
+---+---+---+                   +---+---+---+
| 5 |   | 4 |                   | 8 |   | 4 |
+---+---+---+                   +---+---+---+
| 1 | 7 | 8 |                   | 7 | 6 | 5 |
+---+---+---+                   +---+---+---+
Estado inicial                  Estado final
```

El problema del 8 puzzle

- Una posición es un par de enteros.

```
type Posicion = (Int,Int)
```

- Un tablero es un vector de posiciones, en el que el índice indica el elemento que ocupa la posición.

```
type Tablero  = Array Int Posicion
```

El problema del 8 puzzle

- Una posición es un par de enteros.

```
type Posicion = (Int,Int)
```

- Un tablero es un vector de posiciones, en el que el índice indica el elemento que ocupa la posición.

```
type Tablero  = Array Int Posicion
```

El problema del 8 puzzle

- Una posición es un par de enteros.

```
type Posicion = (Int,Int)
```

- Un tablero es un vector de posiciones, en el que el índice indica el elemento que ocupa la posición.

```
type Tablero  = Array Int Posicion
```

El problema del 8 puzzle

- `inicial8P` es el estado inicial del 8 puzzle. En el ejemplo es

```
+---+---+---+
| 2 | 6 | 3 |
+---+---+---+
| 5 |   | 4 |
+---+---+---+
| 1 | 7 | 8 |
+---+---+---+
```

```haskell
inicial8P :: Tablero
inicial8P = array (0,8) [(2,(1,3)),(6,(2,3)),(3,(3,3)),
                         (5,(1,2)),(0,(2,2)),(4,(3,2)),
                         (1,(1,1)),(7,(2,1)),(8,(3,1))]
```

El problema del 8 puzzle

- `inicial8P` es el estado inicial del 8 puzzle. En el ejemplo es

```
+---+---+---+
| 2 | 6 | 3 |
+---+---+---+
| 5 |   | 4 |
+---+---+---+
| 1 | 7 | 8 |
+---+---+---+
```

```haskell
inicial8P :: Tablero
inicial8P = array (0,8) [(2,(1,3)),(6,(2,3)),(3,(3,3)),
                         (5,(1,2)),(0,(2,2)),(4,(3,2)),
                         (1,(1,1)),(7,(2,1)),(8,(3,1))]
```

El problema del 8 puzzle

- `final8P` es el estado final del 8 puzzle. En el ejemplo es

```
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 8 |   | 4 |
+---+---+---+
| 7 | 6 | 5 |
+---+---+---+
```

```
final8P :: Tablero
final8P = array (0,8) [(1,(1,3)),(2,(2,3)),(3,(3,3)),
                       (8,(1,2)),(0,(2,2)),(4,(3,2)),
                       (7,(1,1)),(6,(2,1)),(5,(3,1))]
```

El problema del 8 puzzle

- `final8P` es el estado final del 8 puzzle. En el ejemplo es

```
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 8 |   | 4 |
+---+---+---+
| 7 | 6 | 5 |
+---+---+---+
```

```haskell
final8P :: Tablero
final8P = array (0,8) [(1,(1,3)),(2,(2,3)),(3,(3,3)),
                       (8,(1,2)),(0,(2,2)),(4,(3,2)),
                       (7,(1,1)),(6,(2,1)),(5,(3,1))]
```

El problema del 8 puzzle

- `(distancia p1 p2)` es la distancia Manhatan entre las posiciones p1 y p2. Por ejemplo,

```
distancia (2,7) (4,1)  ⇝  8
```

```
distancia :: Posicion -> Posicion -> Int
distancia (x1,y1) (x2,y2) = abs (x1-x2) + abs (y1-y2)
```

- `(adyacente p1 p2)` se verifica si las posiciones p1 y p2 son adyacentes. Por ejemplo,

```
adyacente (3,2) (3,1)  ⇝  True
adyacente (3,2) (1,2)  ⇝  False
```

```
adyacente :: Posicion -> Posicion -> Bool
adyacente p1 p2 = distancia p1 p2 == 1
```

El problema del 8 puzzle

- (distancia p1 p2) es la distancia Manhatan entre las posiciones p1 y p2. Por ejemplo,

```
distancia (2,7) (4,1)  ⇝  8
```

```haskell
distancia :: Posicion -> Posicion -> Int
distancia (x1,y1) (x2,y2) = abs (x1-x2) + abs (y1-y2)
```

- (adyacente p1 p2) se verifica si las posiciones p1 y p2 son adyacentes. Por ejemplo,

```
adyacente (3,2) (3,1)  ⇝  True
adyacente (3,2) (1,2)  ⇝  False
```

```haskell
adyacente :: Posicion -> Posicion -> Bool
adyacente p1 p2 = distancia p1 p2 == 1
```

El problema del 8 puzzle

- (distancia p1 p2) es la distancia Manhatan entre las posiciones p1 y p2. Por ejemplo,

```
distancia (2,7) (4,1)  ⇝  8
```

```haskell
distancia :: Posicion -> Posicion -> Int
distancia (x1,y1) (x2,y2) = abs (x1-x2) + abs (y1-y2)
```

- (adyacente p1 p2) se verifica si las posiciones p1 y p2 son adyacentes. Por ejemplo,

```
adyacente (3,2) (3,1)  ⇝  True
adyacente (3,2) (1,2)  ⇝  False
```

```haskell
adyacente :: Posicion -> Posicion -> Bool
adyacente p1 p2 = distancia p1 p2 == 1
```

El problema del 8 puzzle

- `(todosMovimientos t)` es la lista de los tableros obtenidos aplicándole al tablero `t` todos los posibles movimientos; es decir, intercambiando la posición del hueco con sus adyacentes.

```
todosMovimientos :: Tablero -> [Tablero]
todosMovimientos t =
    [t//[(0,t!i),(i,t!0)] | i<-[1..8],
                            adyacente (t!0) (t!i)]
```

- Los nodos del espacio de estados son listas de tableros $[t_n, \dots, t_1]$ tal que t_i es un sucesor de t_{i-1}.

```
data Tableros = Est [Tablero] deriving (Eq, Show)
```

El problema del 8 puzzle

- `(todosMovimientos t)` es la lista de los tableros obtenidos aplicándole al tablero `t` todos los posibles movimientos; es decir, intercambiando la posición del hueco con sus adyacentes.

```
todosMovimientos :: Tablero -> [Tablero]
todosMovimientos t =
    [t//[(0,t!i),(i,t!0)] | i<-[1..8],
                            adyacente (t!0) (t!i)]
```

- Los nodos del espacio de estados son listas de tableros $[t_n, \dots, t_1]$ tal que t_i es un sucesor de t_{i-1}.

```
data Tableros = Est [Tablero] deriving (Eq, Show)
```

El problema del 8 puzzle

- `(sucesores8P e)` es la lista de sucesores del estado e.

```
sucesores8P :: Tableros -> [Tableros]
sucesores8P (Est(n@(t:ts))) =
    filter (noEn ts)
           [Est (t':n) | t' <- todosMovimientos t]
    where
      noEn ts (Est(t:_)) =
         not (elem (elems t) (map elems ts))
```

El problema del 8 puzzle

- `(sucesores8P e)` es la lista de sucesores del estado e.

```
sucesores8P :: Tableros -> [Tableros]
sucesores8P (Est(n@(t:ts))) =
    filter (noEn ts)
           [Est (t':n) | t' <- todosMovimientos t]
    where
      noEn ts (Est(t:_)) =
         not (elem (elems t) (map elems ts))
```

Solución del 8 puzzle por búsqueda en espacios de estados

- (esFinal8P e) se verifica si e es un estado final del 8 puzzle.

```
esFinal8P :: Tableros -> Bool
esFinal8P (Est (n:_)) = elems n == elems final8P
```

- (buscaEE8P) es la lista de las soluciones del problema del 8 puzzle.

```
buscaEE8P :: [[Posicion]]
buscaEE8P = map elems ls
    where ((Est ls):_) = buscaEE sucesores8P
                                 esFinal8P
                                 (Est [inicial8P])
```

- Nota: No termina.

Solución del 8 puzzle por búsqueda en espacios de estados

- `(esFinal8P e)` se verifica si e es un estado final del 8 puzzle.

```
esFinal8P :: Tableros -> Bool
esFinal8P (Est (n:_)) = elems n == elems final8P
```

- `(buscaEE8P)` es la lista de las soluciones del problema del 8 puzzle.

```
buscaEE8P :: [[Posicion]]
buscaEE8P = map elems ls
    where ((Est ls):_) = buscaEE sucesores8P
                                 esFinal8P
                                 (Est [inicial8P])
```

- Nota: No termina.

Solución del 8 puzzle por búsqueda en espacios de estados

- (esFinal8P e) se verifica si e es un estado final del 8 puzzle.

```
esFinal8P :: Tableros -> Bool
esFinal8P (Est (n:_)) = elems n == elems final8P
```

- (buscaEE8P) es la lista de las soluciones del problema del 8 puzzle.

```
buscaEE8P :: [[Posicion]]
buscaEE8P = map elems ls
    where ((Est ls):_) = buscaEE sucesores8P
                                 esFinal8P
                                 (Est [inicial8P])
```

- Nota: No termina.

Tema 23: Técnicas de diseño descendente de algoritmos

El problema de las n reinas

- El problema de las n reinas consiste en colocar n reinas en un tablero cuadrado de dimensiones n por n de forma que no se encuentren más de una en la misma línea: horizontal, vertical o diagonal.
- Las posiciones de las reinas en el tablero se representan por su columna y su fila.

```
type Columna = Int
type Fila    = Int
```

- Una solución del problema de las n reinas es una lista de posiciones.

```
type SolNR = [(Columna,Fila)]
```

El problema de las n reinas

- (valida sp p) se verifica si la posición p es válida respecto de la solución parcial sp; es decir, la reina en la posición p no amenaza a ninguna de las reinas de la sp (se supone que están en distintas columnas). Por ejemplo,

```
valida [(1,1)] (2,2)  ⇝  False
valida [(1,1)] (2,3)  ⇝  True
```

```
valida :: SolNR -> (Columna,Fila) -> Bool
valida solp (c,r) = and [test s | s <- solp]
    where test (c',r') = and [c'+r'/=c+r,
                              c'-r'/=c-r,
                              r'/=r]
```

El problema de las n reinas

- `(valida sp p)` se verifica si la posición `p` es válida respecto de la solución parcial `sp`; es decir, la reina en la posición `p` no amenaza a ninguna de las reinas de la `sp` (se supone que están en distintas columnas). Por ejemplo,

```
valida [(1,1)] (2,2)  ⇝  False
valida [(1,1)] (2,3)  ⇝  True
```

```haskell
valida :: SolNR -> (Columna,Fila) -> Bool
valida solp (c,r) = and [test s | s <- solp]
    where test (c',r') = and [c'+r'/=c+r,
                              c'-r'/=c-r,
                              r'/=r]
```

El problema de las n reinas

- Los nodos del problema de las n reinas son ternas formadas por la columna de la última reina colocada, el número de columnas del tablero y la solución parcial de las reinas colocadas anteriormente.

```
type NodoNR = (Columna,Columna,SolNR)
```

- (sucesoresNR e) es la lista de los sucesores del estado e en el problema de las n reinas. Por ejemplo,

```
ghci> sucesoresNR (1,4,[])
[(2,4,[(1,1)]),(2,4,[(1,2)]),(2,4,[(1,3)]),(2,4,[(1,4)])]
```

```
sucesoresNR :: NodoNR -> [NodoNR]
sucesoresNR (c,n,solp)
    = [(c+1,n,solp++[(c,r)]) | r <- [1..n],
                               valida solp (c,r)]
```

El problema de las n reinas

- Los nodos del problema de las n reinas son ternas formadas por la columna de la última reina colocada, el número de columnas del tablero y la solución parcial de las reinas colocadas anteriormente.

```
type NodoNR = (Columna,Columna,SolNR)
```

- `(sucesoresNR e)` es la lista de los sucesores del estado e en el problema de las n reinas. Por ejemplo,

```
ghci> sucesoresNR (1,4,[])
[(2,4,[(1,1)]),(2,4,[(1,2)]),(2,4,[(1,3)]),(2,4,[(1,4)])]
```

```
sucesoresNR :: NodoNR -> [NodoNR]
sucesoresNR (c,n,solp)
    = [(c+1,n,solp++[(c,r)]) | r <- [1..n],
                               valida solp (c,r)]
```

El problema de las n reinas

- (esFinalNQ e) se verifica si e es un estado final del problema de las n reinas.

```
esFinalNQ :: NodoNR -> Bool
esFinalNQ (c,n,solp) = c > n
```

Solución del problema de las n reinas por EE

- (buscaEE_NQ n) es la primera solución del problema de las n reinas, por búsqueda en espacio de estados. Por ejemplo,

```
ghci> buscaEE_NQ 8
[(1,1),(2,5),(3,8),(4,6),(5,3),(6,7),(7,2),(8,4)]
```

```
buscaEE_NQ :: Columna -> SolNR
buscaEE_NQ n = s
    where ((_,_,s):_) = buscaEE sucesoresNR
                                esFinalNQ
                                (1,n,[])
```

Solución del problema de las n reinas por EE

- (buscaEE_NQ n) es la primera solución del problema de las n reinas, por búsqueda en espacio de estados. Por ejemplo,

```
ghci> buscaEE_NQ 8
[(1,1),(2,5),(3,8),(4,6),(5,3),(6,7),(7,2),(8,4)]
```

```
buscaEE_NQ :: Columna -> SolNR
buscaEE_NQ n = s
    where ((_,_,s):_) = buscaEE sucesoresNR
                                esFinalNQ
                                (1,n,[])
```

Solución del problema de las n reinas por EE

- (nSolucionesNQ n) es el número de soluciones del problema de las n reinas, por búsqueda en espacio de estados. Por ejemplo,

  ```
  nSolucionesNQ 8  ⇝  92
  ```

  ```
  nSolucionesNQ :: Columna -> Int
  nSolucionesNQ n =
      length (buscaEE sucesoresNR
                      esFinalNQ
                      (1,n,[]))
  ```

Solución del problema de las n reinas por EE

- (nSolucionesNQ n) es el número de soluciones del problema de las n reinas, por búsqueda en espacio de estados. Por ejemplo,

```
nSolucionesNQ 8  ⇝  92
```

```
nSolucionesNQ :: Columna -> Int
nSolucionesNQ n =
    length (buscaEE sucesoresNR
                    esFinalNQ
                    (1,n,[]))
```

Tema 23: Técnicas de diseño descendente de algoritmos

El problema de la mochila

- Se tiene una mochila de capacidad de peso p y una lista de n objetos para colocar en la mochila. Cada objeto i tiene un peso w_i y un valor v_i. Considerando la posibilidad de colocar el mismo objeto varias veces en la mochila, el problema consiste en determinar la forma de colocar los objetos en la mochila sin sobrepasar la capacidad de la mochila colocando el máximo valor posible.
- Los pesos son número enteros.

```
type Peso = Int
```

- Los valores son números reales.

```
type Valor = Float
```

El problema de la mochila

- Los objetos son pares formado por un peso y un valor.

```
type Objeto = (Peso,Valor)
```

- Una solución del problema de la mochila es una lista de objetos.

```
type SolMoch = [Objeto]
```

El problema de la mochila

- Los estados del problema de la mochila son 5–tuplas de la forma `(v,p,l,o,s)` donde
  - `v` es el valor de los objetos colocados,
  - `p` es el peso de los objetos colocados,
  - `l` es el límite de la capacidad de la mochila,
  - `o` es la lista de los objetos colocados (ordenados de forma creciente según sus pesos) y
  - `s` es la solución parcial.

```
type NodoMoch = (Valor,Peso,Peso,[Objeto],SolMoch)
```

El problema de la mochila

- (sucesoresMoch e) es la lista de los sucesores del estado e en el problema de la mochila.

```
sucesoresMoch :: NodoMoch -> [NodoMoch]
sucesoresMoch (v,p,limite,objetos,solp)
    = [( v+v',
         p+p',
         limite,
         [o | o@(p'',_) <- objetos,(p''>=p')],
         (p',v'):solp )
       | (p',v') <- objetos,
         p+p' <= limite]
```

El problema de la mochila

- (sucesoresMoch e) es la lista de los sucesores del estado e en el problema de la mochila.

```
sucesoresMoch :: NodoMoch -> [NodoMoch]
sucesoresMoch (v,p,limite,objetos,solp)
    = [( v+v',
         p+p',
         limite,
         [o | o@(p'',_) <- objetos,(p''>=p')],
         (p',v'):solp )
       | (p',v') <- objetos,
         p+p' <= limite]
```

El problema de la mochila

- `(esObjetivoMoch e)` se verifica si e es un estado final el problema de la mochila.

```
esObjetivoMoch :: NodoMoch -> Bool
esObjetivoMoch (_,p,limite,((p',_):_),_) =
   p+p'>limite
```

El problema de la mochila

- `(esObjetivoMoch e)` se verifica si e es un estado final el problema de la mochila.

```
esObjetivoMoch :: NodoMoch -> Bool
esObjetivoMoch (_,p,limite,((p',_):_),_) =
   p+p'>limite
```

Solución del problema de la mochila por EE

- (buscaEE_Mochila os l) es la solución del problema de la mochila para la lista de objetos os y el límite de capacidad l. Por ejemplo,

```
> buscaEE_Mochila [(2,3),(3,5),(4,6),(5,10)] 8
([(5,10.0),(3,5.0)],15.0)
> buscaEE_Mochila [(2,3),(3,5),(5,6)] 10
([(3,5.0),(3,5.0),(2,3.0),(2,3.0)],16.0)
> buscaEE_Mochila [(2,2.8),(3,4.4),(5,6.1)] 10
([(3,4.4),(3,4.4),(2,2.8),(2,2.8)],14.4)
```

```
buscaEE_Mochila :: [Objeto] -> Peso -> (SolMoch,Valor)
buscaEE_Mochila objetos limite = (sol,v)
    where
      (v,_,_,_,sol) =
          maximum (buscaEE sucesoresMoch
                           esObjetivoMoch
                           (0,0,limite,sort objetos,[]))
```

Solución del problema de la mochila por EE

- (buscaEE_Mochila os l) es la solución del problema de la mochila para la lista de objetos os y el límite de capacidad l. Por ejemplo,

```
> buscaEE_Mochila [(2,3),(3,5),(4,6),(5,10)] 8
([(5,10.0),(3,5.0)],15.0)
> buscaEE_Mochila [(2,3),(3,5),(5,6)] 10
([(3,5.0),(3,5.0),(2,3.0),(2,3.0)],16.0)
> buscaEE_Mochila [(2,2.8),(3,4.4),(5,6.1)] 10
([(3,4.4),(3,4.4),(2,2.8),(2,2.8)],14.4)
```

```
buscaEE_Mochila :: [Objeto] -> Peso -> (SolMoch,Valor)
buscaEE_Mochila objetos limite = (sol,v)
    where
      (v,_,_,_,sol) =
          maximum (buscaEE sucesoresMoch
                           esObjetivoMoch
                           (0,0,limite,sort objetos,[]))
```

Tema 23: Técnicas de diseño descendente de algoritmos

El patrón de búsqueda por primero el mejor

- `(buscaPM s o e)` es la lista de soluciones del problema de espacio de estado definido por la función sucesores (`s`), el objetivo (`o`) y estado inicial (`e`), obtenidas buscando por primero el mejor.

```
import ColaDePrioridadConMonticulos

buscaPM :: (Ord nodo) => (nodo -> [nodo]) -> (nodo -> Bool)
                         -> nodo -> [(nodo,Int)]
buscaPM sucesores esFinal x = busca' (inserta x vacia) 0
 where
   busca' c t
    | esVacia c = []
    | esFinal (primero c)
        = ((primero c),t+1):(busca' (resto c)(t+1))
    | otherwise
        = busca' (foldr inserta (resto c) (sucesores x)) (t+1)
          where x = primero c
```

El patrón de búsqueda por primero el mejor

- (buscaPM s o e) es la lista de soluciones del problema de espacio de estado definido por la función sucesores (s), el objetivo (o) y estado inicial (e), obtenidas buscando por primero el mejor.

```
import ColaDePrioridadConMonticulos

buscaPM :: (Ord nodo) => (nodo -> [nodo]) -> (nodo -> Bool)
                         -> nodo -> [(nodo,Int)]
buscaPM sucesores esFinal x = busca' (inserta x vacia) 0
 where
   busca' c t
    | esVacia c = []
    | esFinal (primero c)
        = ((primero c),t+1):(busca' (resto c)(t+1))
    | otherwise
        = busca' (foldr inserta (resto c) (sucesores x)) (t+1)
          where x = primero c
```

Tema 23: Técnicas de diseño descendente de algoritmos

El problema del 8 puzzle por BPM

- `(heur1 t)` es la suma de la distancia Manhatan desde la posición de cada objeto del tablero `t` a su posición en el estado final. Por ejemplo,

  ```
  heur1 inicial8P  ~>  12
  ```

```
heur1 :: Tablero  -> Int
heur1 b =
    sum [distancia (b!i) (final8P!i) | i <- [0..8]]
```

- Dos estados se consideran iguales si tienen la misma heurística.

```
instance Eq Tableros
    where Est(t1:_) == Est(t2:_) = heur1 t1 == heur1 t2
```

El problema del 8 puzzle por BPM

- `(heur1 t)` es la suma de la distancia Manhatan desde la posición de cada objeto del tablero `t` a su posición en el estado final. Por ejemplo,

  ```
  heur1 inicial8P  ⇝  12
  ```

```
heur1 :: Tablero  -> Int
heur1 b =
    sum [distancia (b!i) (final8P!i) | i <- [0..8]]
```

- Dos estados se consideran iguales si tienen la misma heurística.

```
instance Eq Tableros
    where Est(t1:_) == Est(t2:_) = heur1 t1 == heur1 t2
```

El problema del 8 puzzle por BPM

- Un estado es menor o igual que otro si tiene una heurística menor o igual.

```
instance Ord Tableros where
    Est (t1:_) <= Est (t2:_) = heur1 t1 <= heur1 t2
```

- (buscaPM_8P) es la lista de las soluciones del 8 puzzle por búsqueda primero el mejor.

```
buscaPM_8P = buscaPM sucesores8P
                     esFinal8P
                     (Est [inicial8P])
```

El problema del 8 puzzle por BPM

- Un estado es menor o igual que otro si tiene una heurística menor o igual.

```
instance Ord Tableros where
    Est (t1:_) <= Est (t2:_) = heur1 t1 <= heur1 t2
```

- (buscaPM_8P) es la lista de las soluciones del 8 puzzle por búsqueda primero el mejor.

```
buscaPM_8P = buscaPM sucesores8P
                     esFinal8P
                     (Est [inicial8P])
```

El problema del 8 puzzle por BPM

- (nSolucionesPM_8P) es el número de soluciones del 8 puzzle por búsqueda primero el mejor. Por ejemplo,

```
nSolucionesPM_8P  ~>  43
```

```
nSolucionesPM_8P = length ls
    where (((Est ls),_):_) = buscaPM sucesores8P
                                     esFinal8P
                                     (Est [inicial8P])
```

El problema del 8 puzzle por BPM

- (nSolucionesPM_8P) es el número de soluciones del 8 puzzle por búsqueda primero el mejor. Por ejemplo,

```
nSolucionesPM_8P  ⇝  43
```

```
nSolucionesPM_8P = length ls
    where (((Est ls),_):_) = buscaPM sucesores8P
                                     esFinal8P
                                     (Est [inicial8P])
```

Tema 23: Técnicas de diseño descendente de algoritmos

El patrón de búsqueda en escalada

- `(buscaEscalada s o e)` es la lista de soluciones del problema de espacio de estado definido por la función sucesores (`s`), el objetivo (`o`) y estado inicial (`e`), obtenidas buscando por escalada.

```
buscaEscalada :: Ord nodo => (nodo -> [nodo])
                 -> (nodo -> Bool) -> nodo -> [nodo]
buscaEscalada sucesores esFinal x =
    busca' (inserta x vacia) where
    busca' c
        | esVacia c           = []
        | esFinal (primero c) = [primero c]
        | otherwise           =
            busca' (foldr inserta vacia (sucesores x))
            where x = primero c
```

El patrón de búsqueda en escalada

- `(buscaEscalada s o e)` es la lista de soluciones del problema de espacio de estado definido por la función sucesores (`s`), el objetivo (`o`) y estado inicial (`e`), obtenidas buscando por escalada.

```
buscaEscalada :: Ord nodo => (nodo -> [nodo])
                 -> (nodo -> Bool) -> nodo -> [nodo]
buscaEscalada sucesores esFinal x =
    busca' (inserta x vacia) where
    busca' c
        | esVacia c            = []
        | esFinal (primero c) = [primero c]
        | otherwise            =
          busca' (foldr inserta vacia (sucesores x))
          where x = primero c
```

Tema 23: Técnicas de diseño descendente de algoritmos

1. La técnica de divide y vencerás
2. Búsqueda en espacios de estados
3. Búsqueda por primero el mejor
4. Búsqueda en escalada
   - El patrón de búsqueda en escalada
   - El problema del cambio de monedas por escalada
   - El algoritmo de Prim del árbol de expansión mínimo por escalada

El problema del cambio de monedas por escalada

- El problema del cambio de monedas consiste en determinar cómo conseguir una cantidad usando el menor número de monedas disponibles.
- Las monedas son números enteros.

```
type Moneda = Int
```

- `monedas` es la lista del tipo de monedas disponibles. Se supone que hay un número infinito de monedas de cada tipo.

```
monedas :: [Moneda]
monedas = [1,2,5,10,20,50,100]
```

- Las soluciones son listas de monedas.

```
type Soluciones = [Moneda]
```

El problema del cambio de monedas por escalada

- Los estados son pares formados por la cantidad que falta y la lista de monedas usadas.

```haskell
type NodoMonedas = (Int, [Moneda])
```

- (sucesoresMonedas e) es la lista de los sucesores del estado e en el problema de las monedas. Por ejemplo,

```
ghci> sucesoresMonedas (199,[])
[(198,[1]),(197,[2]),(194,[5]),(189,[10]),
 (179,[20]),(149,[50]),(99,[100])]
```

```haskell
sucesoresMonedas :: NodoMonedas -> [NodoMonedas]
sucesoresMonedas (r,p) =
    [(r-c,c:p) | c <- monedas, r-c >= 0]
```

El problema del cambio de monedas por escalada

- Los estados son pares formados por la cantidad que falta y la lista de monedas usadas.

```
type NodoMonedas = (Int, [Moneda])
```

- `(sucesoresMonedas e)` es la lista de los sucesores del estado e en el problema de las monedas. Por ejemplo,

```
ghci> sucesoresMonedas (199,[])
[(198,[1]),(197,[2]),(194,[5]),(189,[10]),
 (179,[20]),(149,[50]),(99,[100])]
```

```
sucesoresMonedas :: NodoMonedas -> [NodoMonedas]
sucesoresMonedas (r,p) =
    [(r-c,c:p) | c <- monedas, r-c >= 0]
```

El problema del cambio de monedas por escalada

- `(esFinalMonedas e)` se verifica si e es un estado final del problema de las monedas.

```
esFinalMonedas :: NodoMonedas -> Bool
esFinalMonedas (v,_) = v==0
```

- `(cambio n)` es la solución del problema de las monedas por búsqueda en escalada. Por ejemplo,

```
cambio 199  ~>  [2,2,5,20,20,50,100]
```

```
cambio :: Int -> Soluciones
cambio n =
    snd (head (buscaEscalada sucesoresMonedas
                             esFinalMonedas
                             (n,[])))
```

El problema del cambio de monedas por escalada

- `(esFinalMonedas e)` se verifica si e es un estado final del problema de las monedas.

```
esFinalMonedas :: NodoMonedas -> Bool
esFinalMonedas (v,_) = v==0
```

- `(cambio n)` es la solución del problema de las monedas por búsqueda en escalada. Por ejemplo,

```
cambio 199  ⇝  [2,2,5,20,20,50,100]
```

```
cambio :: Int -> Soluciones
cambio n =
    snd (head (buscaEscalada sucesoresMonedas
                             esFinalMonedas
                             (n,[])))
```

El problema del cambio de monedas por escalada

- (esFinalMonedas e) se verifica si e es un estado final del problema de las monedas.

```
esFinalMonedas :: NodoMonedas -> Bool
esFinalMonedas (v,_) = v==0
```

- (cambio n) es la solución del problema de las monedas por búsqueda en escalada. Por ejemplo,

```
cambio 199  ⇝  [2,2,5,20,20,50,100]
```

```
cambio :: Int -> Soluciones
cambio n =
    snd (head (buscaEscalada sucesoresMonedas
                             esFinalMonedas
                             (n,[])))
```

Tema 23: Técnicas de diseño descendente de algoritmos

El algoritmo de Prim del árbol de expansión mínimo por escalada

- Ejemplo de grafo.

```
g1 :: Grafo Int Int
g1 = creaGrafo True (1,5) [(1,2,12),(1,3,34),(1,5,78),
                           (2,4,55),(2,5,32),
                           (3,4,61),(3,5,44),
                           (4,5,93)]
```

- Una arista esta formada dos nodos junto con su peso.

```
type Arista a b = (a,a,b)
```

El algoritmo de Prim del árbol de expansión mínimo por escalada

- Un nodo `(NodoAEM (p,t,r,aem))` está formado por
  - el peso `p` de la última arista añadida el árbol de expansión mínimo (`aem`),
  - la lista `t` de nodos del grafo que están en el `aem`,
  - la lista `r` de nodos del grafo que no están en el `aem` y
  - el `aem`.

```
type NodoAEM a b = (b,[a],[a],[Arista a b])
```

El algoritmo de Prim del árbol de expansión mínimo por escalada

- `(sucesoresAEM g n)` es la lista de los sucesores del nodo `n` en el grafo `g`. Por ejemplo,

```
ghci> sucesoresAEM g1 (0,[1],[2..5],[])
[(12,[2,1],[3,4,5],[(1,2,12)]),
 (34,[3,1],[2,4,5],[(1,3,34)]),
 (78,[5,1],[2,3,4],[(1,5,78)])]
```

```
sucesoresAEM :: (Ix a,Num b) => (Grafo a b) -> (NodoAEM a
                                -> [(NodoAEM a b)]
sucesoresAEM g (_,t,r,aem)
        = [(peso x y g, (y:t), delete y r, (x,y,peso x y g
            | x <- t , y <- r, aristaEn g (x,y)]
```

El algoritmo de Prim del árbol de expansión mínimo por escalada

- (sucesoresAEM g n) es la lista de los sucesores del nodo n en el grafo g. Por ejemplo,

```
ghci> sucesoresAEM g1 (0,[1],[2..5],[])
[(12,[2,1],[3,4,5],[(1,2,12)]),
 (34,[3,1],[2,4,5],[(1,3,34)]),
 (78,[5,1],[2,3,4],[(1,5,78)])]
```

```
sucesoresAEM :: (Ix a,Num b) => (Grafo a b) -> (NodoAEM a b)
                                -> [(NodoAEM a b)]
sucesoresAEM g (_,t,r,aem)
        = [(peso x y g, (y:t), delete y r, (x,y,peso x y g)
          | x <- t , y <- r, aristaEn g (x,y)]
```

El algoritmo de Prim del árbol de expansión mínimo por escalada

- `(esFinalAEM n)` se verifica si `n` es un estado final; es decir, si no queda ningún elemento en la lista de nodos sin colocar en el árbol de expansión mínimo.

```
esFinalAEM (_,_,[],_) = True
esFinalAEM _           = False
```

El algoritmo de Prim del árbol de expansión mínimo por escalada

- (esFinalAEM n) se verifica si n es un estado final; es decir, si no queda ningún elemento en la lista de nodos sin colocar en el árbol de expansión mínimo.

```
esFinalAEM (_,_,[],_) = True
esFinalAEM _          = False
```

El algoritmo de Prim del árbol de expansión mínimo por escalada

- `(prim g)` es el árbol de expansión mínimo del grafo g, por el algoritmo de Prim como búsqueda en escalada. Por ejemplo,

  `prim g1` ⇝ `[(2,4,55),(1,3,34),(2,5,32),(1,2,12)]`

```
prim g = sol
    where [(_,_,_,sol)] = buscaEscalada (sucesoresAEM g)
                                        esFinalAEM
                                        (0,[n],ns,[])
          (n:ns) = nodos g
```

El algoritmo de Prim del árbol de expansión mínimo por escalada

- (prim g) es el árbol de expansión mínimo del grafo g, por el algoritmo de Prim como búsqueda en escalada. Por ejemplo,

```
prim g1 ⇝ [(2,4,55),(1,3,34),(2,5,32),(1,2,12)]
```

```
prim g = sol
    where [(_,_,_,sol)] = buscaEscalada (sucesoresAEM g)
                                         esFinalAEM
                                         (0,[n],ns,[])
          (n:ns) = nodos g
```