Tema 21: Algoritmos de exploración de grafos

Informática (2009–10)

José A. Alonso Jiménez

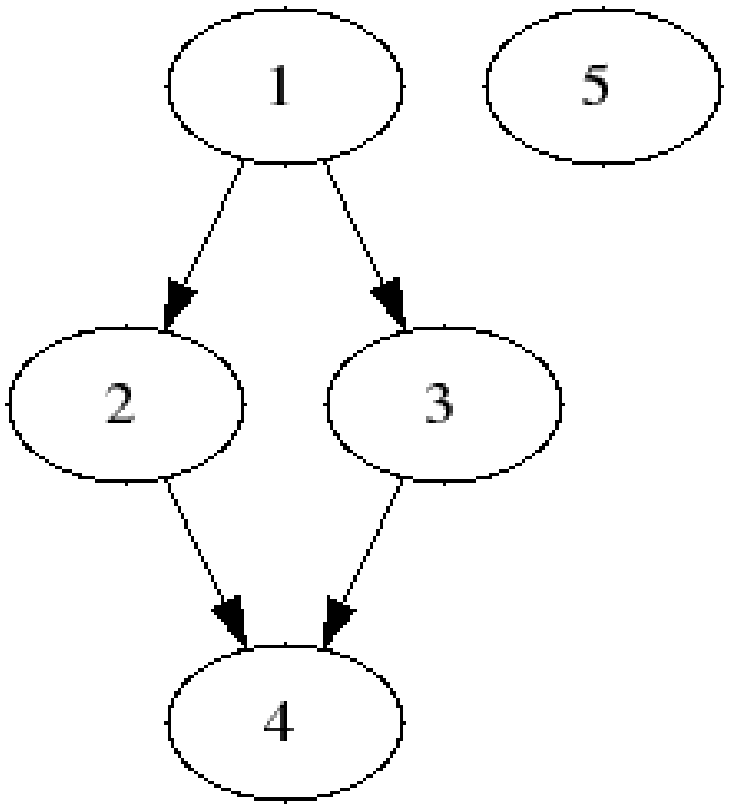
Grupo de Lógica Computacional
Departamento de Ciencias de la Computación e I.A.
Universidad de Sevilla

Tema 21: Algoritmos de exploración de grafos

Tema 21: Algoritmos de exploración de grafos

Representación de grafos

- Ejemplo de grafo



Representación de grafos

- Un grafo de tipo `v` es un tipo de datos compuesto por la lista de vértices y la función que asigna a cada vértice la lista de sus sucesores.

```
data Grafo v = G [v] (v -> [v])
```

- El grafo anterior se representa por

```
ej_grafo = G [1..5] suc
    where suc 1 = [2,3]
          suc 2 = [4]
          suc 3 = [4]
          suc _ = []
```

Búsqueda en profundidad

- `(camino g u v)` es un camino (i.e una lista de vértices tales que cada uno es un sucesor del anterior) en el grafo `g` desde el vértice `u` al `v`. Por ejemplo,

```
camino ej_grafo 1 4 ⇝ [4,2,1]
```

```
camino :: Eq a => Grafo a -> a -> a -> [a]
camino g u v = head (caminosDesde g u (== v) [])
```

Búsqueda en profundidad

- `(caminosDesde g o te vis)` es la lista de los caminos en el grafo g desde el vértice origen o hasta vértices finales (i.e los que verifican el test de encontrado `te`) sin volver a pasar por los vértices visitados `vis`. Por ejemplo,

```
caminosDesde ej_grafo 1 (==4) [] ⇝ [[4,2,1],[4,3,1]]
```

```
caminosDesde :: Eq a => Grafo a -> a -> (a -> Bool) -> [a]
caminosDesde g o te vis
    | te o      = [o:vis]
    | otherwise = concat [caminosDesde g o' te (o:vis)
                          | o' <- suc o,
                            notElem o' vis]
    where G _ suc = g
```

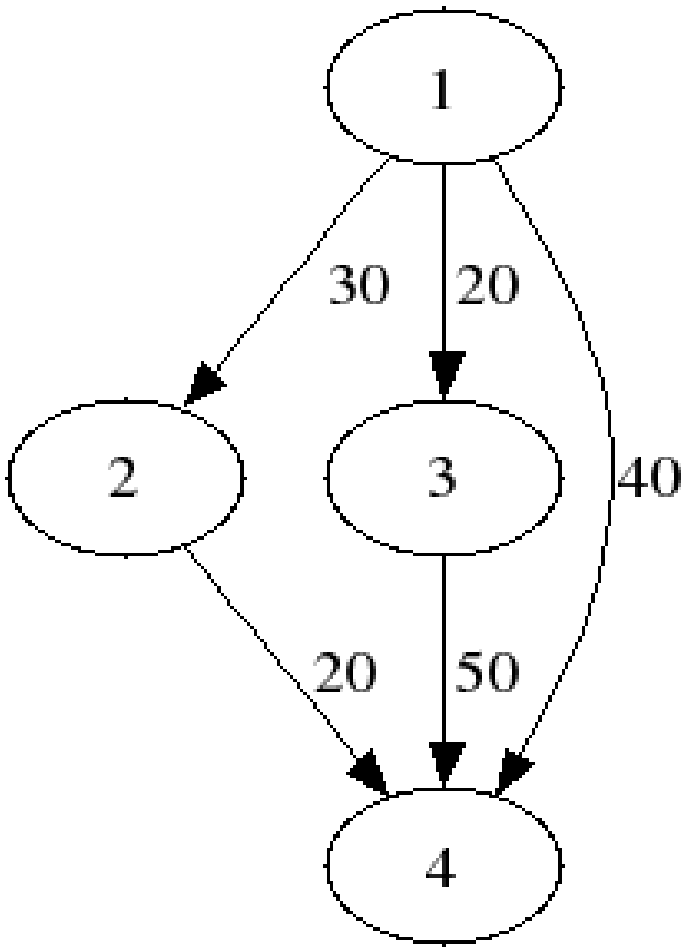
Búsqueda en anchura

- Para hacer la búsqueda en anchura basta cambiar en la definición de `caminosDesde` la expresión `o:vis` por `vis++[o]`.

Tema 21: Algoritmos de exploración de grafos

Representación de grafos con pesos

- Ejemplo de grafo con peso



Representación de grafos con pesos

- Un grafo con pesos de tipo `v` es un tipo de datos compuesto por la lista de vértices y la función que asigna a cada vértice la lista de sus sucesores junto con el coste de la transición.

```
data Grafo v p = G [v] (v -> [(v,p)])
```

- El grafo con pesos del ejemplo se representa por

```
ej_grafo = G [1..4] suc
    where suc 1 = [(2,30), (3,20), (4,40)]
          suc 2 = [(4,20)]
          suc 3 = [(4,50)]
          suc _ = []
```

Búsqueda acotada en grafos con pesos

- `(caminos g u v p)` es la lista de caminos en el grafo `g` desde el vértice `u` al `v` de coste menor o igual que `p`. Por ejemplo,

```
caminos ej_grafo 1 4 40 ⇝ [[4,1]]
caminos ej_grafo 1 4 50 ⇝ [[4,2,1],[4,1]]
caminos ej_grafo 1 4 70 ⇝ [[4,2,1],[4,3,1],[4,1]]
```

```
caminos :: (Eq a, Num b, Ord b) =>
           Grafo a b -> a -> a -> b -> [[a]]
caminos g u v pt =
    caminosDesde g u (\x _ -> x==v) (> pt) [] 0
```

Búsqueda acotada en grafos con pesos

- `(caminosDesde g o te tr vis p)` es la lista de los caminos en el grafo `g` desde el vértice origen `o` hasta vértices finales (i.e los que verifican el test de encontrado `te`) que no están podados por el test de retorno `tr` teniendo en cuenta los vértices visitados `vis` y el peso consumido `p`. Por ejemplo,

```
> caminosDesde ej_grafo 1 (\x _ -> x==4) (>50) [] 0
[[4,2,1],[4,1]]
```

Búsqueda acotada en grafos con pesos

```
caminosDesde :: (Eq a, Num b) =>
                Grafo a b -> a -> (a -> b -> Bool) ->
                (b -> Bool) -> [a] -> b -> [[a]]
caminosDesde g o te tr vis p
    | te o p    = [o:vis]
    | otherwise = concat [caminosDesde g o' te tr (o:vis) np |
                          (o',p') <- suc o,
                          notElem o' vis,
                          let np = p+p',
                          not(tr np)]
    where G _ suc = g
```

Coste de un camino

- `(costeCamino g c)` es el coste del camino c en el grafo g. Por ejemplo,

```
costeCamino ej_grafo [4,1]   ⇝ 40
costeCamino ej_grafo [4,2,1] ⇝ 50
costeCamino ej_grafo [4,3,1] ⇝ 70
```

```
costeCamino :: (Num a, Eq b) => Grafo b a -> [b] -> a
costeCamino _ [_]      = 0
costeCamino g (u:v:xs) = p + costeCamino g (v:xs)
                         where G _ suc = g
                               Just p  = lookup u (suc v)
```

Inserción de un camino

- `(insertaCamino g c l)` inserta el camino c del grafo g en la lista de caminos `l` delante del primer elemento de `l` de coste mayor o igual que e. Por ejemplo,

```
Main*> insertaCamino ej_grafo [4,2,1] [[4,1],[4,3,1]]
[[4,1],[4,2,1],[4,3,1]]
```

```
insertaCamino :: (Num a, Eq b, Ord a) =>
                 Grafo b a -> [b] -> [[b]] -> [[b]]
insertaCamino g c []     = [c]
insertaCamino g c (x:xs)
    | costeCamino g c <= costeCamino g x = c:x:xs
    | otherwise = x : insertaCamino g c xs
```

Ordenación de caminos por coste

- `(ordenaCaminos l)` es la lista `l` ordenada mediante inserción,
  Por ejemplo,

```
Main*> ordenaCaminos ej_grafo [[4,2,1],[4,3,1],[4,1]]
[[4,1],[4,2,1],[4,3,1]]
```

```
ordenaCaminos :: (Ord a, Eq b, Num a) =>
                 Grafo b a -> [[b]] -> [[b]]
ordenaCaminos g  = foldr (insertaCamino g) []
```

Búsqueda de caminos de menor coste

- `(mejorCamino g u v)` es el camino de menor coste en el grafo g desde u hasta v. Por ejemplo.

```
mejorCamino ej_grafo 1 4 ⇝ [4,1]
```

```
mejorCamino :: (Eq a, Num b, Ord b) =>
               Grafo a b -> a -> a -> [a]
mejorCamino g u v =
    head(ordenaCaminos g (caminosDesde g
                                       u
                                       (\x _ -> x==v)
                                       (\_ -> False)
                                       []
                                       0))
```

Tema 21: Algoritmos de exploración de grafos

La clase de los grafos con búsqueda en profundidad

- La clase de los grafos con búsqueda en profundidad de tipo `v` consta de los siguientes métodos primitivos:
  1. `vértices` es la lista de los vértices del grafo.
  2. `(suc x)` es la lista de los sucesores del vértice `x`.

  y de los siguientes métodos definidos:
  1. `(camino u v)` es un camino (i.e una lista de vértices tales que cada uno es un sucesor del anterior) desde el vértice `u` al `v`.
  2. `(caminosDesde o te vis)` es la lista de los caminos desde el vértice origen `o` hasta vértices finales (i.e los que verifican el test de encontrado `te`) a partir de los vértices visitados `vis`.
  3. `(tv x ys)` se verifica si el vértice `x` cumple el test de visitabilidad respecto de la lista de vértices `ys`. El test de visitabilidad por defecto es que `x` no pertenezca a `ys` (para evitar ciclos en el grafo).

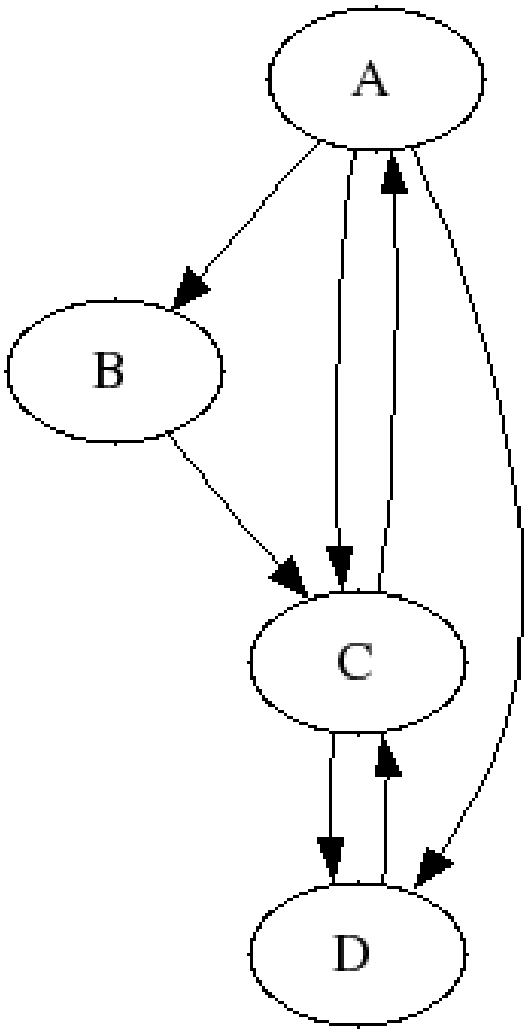
  Para definir un grafo basta determinar los vértices y sucesores.

  Los tres restantes métodos están definidos a partir de ellos.

La clase de los grafos con búsqueda en profundidad

```
class Eq v => Grafo v where
  vértices     :: [v]
  suc          :: v -> [v]
  camino       :: v -> v -> [v]
  caminosDesde :: v -> (v -> Bool) -> [v] -> [[v]]
  tv           :: v -> [v] -> Bool
  -- Métodos por defecto:
  camino u v = head (caminosDesde u (== v) [])
  caminosDesde o te vis
      | te o      = [o:vis]
      | otherwise = concat [caminosDesde o' te (o:vis) |
                            o' <- suc o,
                            tv o' vis]
  tv = notElem
```

Primer ejemplo de grafo con búsqueda en profundidad



Primer ejemplo de grafo con búsqueda en profundidad

- El grafo anterior se representa por

```
data Vértice = A | B | C | D | E
               deriving (Show,Eq,Enum)

instance Grafo Vértice where
    vértices = [A .. E]
    suc A = [B,C,D]
    suc B = [C]
    suc C = [A,D]
    suc D = [C]
    suc E = []
```

Primer ejemplo de grafo con búsqueda en profundidad

- Con los métodos definidos podemos calcular

```
camino A D                    ⇝ [D,C,B,A]
caminosDesde A (== D) []      ⇝ [[D,C,B,A],[D,C,A],[D,A]]
tv B [A,B,C]                  ⇝ False
tv D [A,B,C]                  ⇝ True
```

Segundo ejemplo de grafo con búsqueda en profundidad

- El grafo anterior, suponiendo que el vértice `B` no es visitable, se representa por

```
data Vértice = A | B | C | D | E
               deriving (Show,Eq,Enum)

instance Grafo Vértice where
    vértices = [A .. E]
    suc A = [B,C,D]
    suc B = [C]
    suc C = [A,D]
    suc D = [C]
    suc E = []
    tv x ys = notElem x ys && x /= B
```

Segundo ejemplo de grafo con búsqueda en profundidad

- Con los métodos definidos podemos calcular

```
camino A D                      ⇝ [D,C,A]
caminosDesde A (== D) []        ⇝ [[D,C,A],[D,A]]
tv B [A,B,D]                    ⇝ False
tv D [A,B,C]                    ⇝ True
tv B [A,C,E]                    ⇝ False
```

Tercer ejemplo de grafo con búsqueda en profundidad

- El grafo anterior, donde cada vértice es visitable hasta dos veces se representa por

```
data Vértice = A | B | C | D | E
               deriving (Show,Eq,Enum)

instance Grafo Vértice where
    vértices = [A .. E]
    suc A = [B,C,D]
    suc B = [C]
    suc C = [A,D]
    suc D = [C]
    suc E = []
    tv x ys = notElem x (ys \\ [x])
```

Tercer ejemplo de grafo con búsqueda en profundidad

- Con los métodos definidos podemos calcular

```
camino A D                    ⇝ [D,C,B,A,C,B,A]
caminosDesde A (== D) []      ⇝ [[D,C,B,A,C,B,A],
                                 [D,C,A,C,B,A],
                                 [D,A,C,B,A],
                                 [D,C,B,A],
                                 [D,C,B,A,C,A],
                                 [D,C,A,C,A],
                                 [D,A,C,A],
                                 [D,C,A],
                                 [D,A]]
tv B [A,B,D]                  ⇝ True
tv B [A,B,D,B]                ⇝ False
tv D [A,B,C]                  ⇝ True
```

Tema 21: Algoritmos de exploración de grafos

La clase grafo de búsqueda con pesos

- Un `Arco` es un par formado por un vértice y un peso.

```
data Arco  v p = Arc v p
type Arcos v p = [Arco v p]
```

La clase grafo de búsqueda con pesos

- La clase de los grafos con búsqueda con pesos de tipo `v` consta de los siguientes métodos primitivos:
  1. `vértices` es la lista de los vértices del grafo.
  2. `(suc x)` es la lista de los sucesores del vértice `x`, donde cada sucesor de un vérrtice `x` es un `Arco y p` donde `y` es el vértice sucesor de `x` y `p` es el coste de ir de `x` a `y`.

  y de los siguientes métodos definidos:
  1. `(camino u v p)` es la lista de caminos desde el vértice `u` al `v` de coste menor o igual que `p`.
  2. `(caminosDesde o te tr vis p)` es la lista de los caminos desde el vértice origen `o` hasta vértices finales (i.e los que verifican el test de encontrado `te`) que no están podados por el test de retorno `tr` teniendo en cuenta los vértices visitados `vis` y el peso consumido `p`.
  3. `(tv a ys)` se verifica si el arco `a` cumple el test de visitabilidad respecto de la lista de vértices `ys`. El test de visitabilidad por defecto es que el vértice de `a` no pertenezca a `ys` (para evitar ciclos en el grafo).

  Para definir un grafo basta determinar los vértices y sucesores. Los tres restantes métodos están definidos a partir de ellos.

La clase grafo de búsqueda con pesos

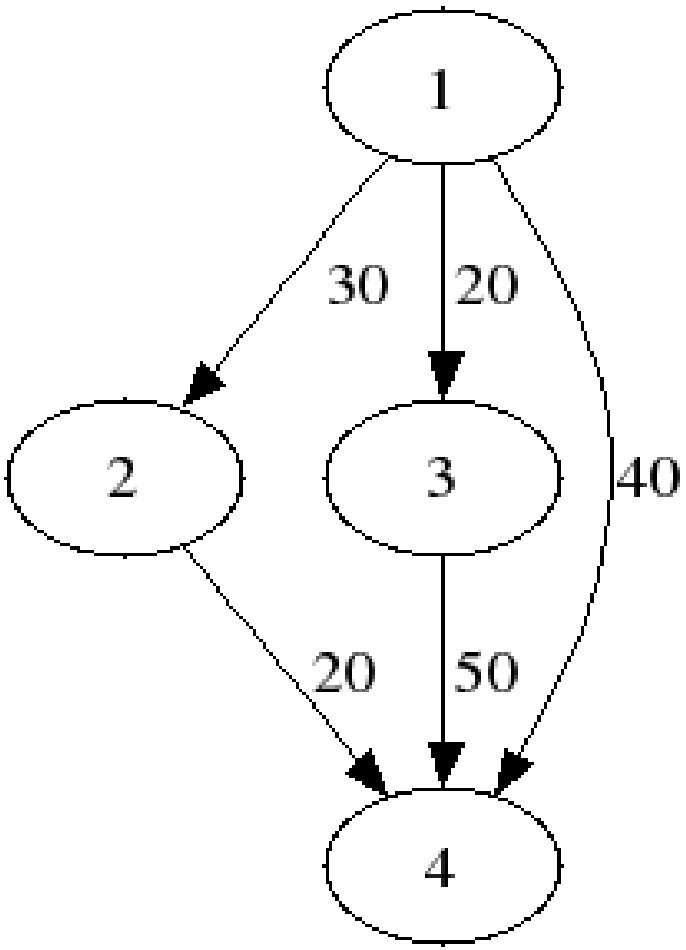
```
class Eq v => GrafoPesos v where
  vértices     :: [v]
  suc          :: Num p => v -> Arcos v p
  caminos      :: (Num p,Ord p) => v -> v -> p -> [[v]]
  caminosDesde :: Num p => v -> (v->p->Bool) -> (p->Bool) ->
                           [v] -> p -> [[v]]
  tv           :: Num p => (v,p) -> [v] -> Bool
  -- Métodos por defecto:
  caminos u v pt = caminosDesde u (\x _ -> x==v) (> pt) [] 0

  tv (x,_) ys = notElem x ys

  caminosDesde o te tr vis p
      | te o p    = [o:vis]
      | otherwise = concat [caminosDesde o' te tr (o:vis) np |
                            Arc o' p' <- suc o,
                            tv (o',p') vis,
                            let np = p+p',
                            not(tr np)]
```

Ejemplo de grafo de búsqueda con pesos

- Ejemplo de grafo de búsqueda con pesos



Ejemplo de grafo de búsqueda con pesos

- El grafo anterior se representa por

```
instance GrafoPesos Int where
   suc 1 = [Arco 2 30, Arco 3 20, Arco 4 40]
   suc 3 = [Arco 4 50]
   suc 2 = [Arco 4 20]
```

- Con los métodos definidos podemos calcular

```
caminos 1 4 40 :: [[Int]] ⇝ [[4,1]]
caminos 1 4 50 :: [[Int]] ⇝ [[4,2,1],[4,1]]
caminos 1 4 70 :: [[Int]] ⇝ [[4,2,1],[4,3,1],[4,1]]
```

Notar que se ha indicado el tipo del resultado para resolver las ambiguedades de tipo.

Tema 21: Algoritmos de exploración de grafos

Tema 21: Algoritmos de exploración de grafos

Enunciado del problema de las jarras

- En el problema de las jarras `X-Y-Z` se dispone de una jarra de capacidad `X` litros, otra jarra de capacidad `Y` litros, de un grifo que permite llenar las jarras de agua (las jarras no tienen marcas de medición) y de un lavabo donde vaciar las jarras. El problema consiste en averiguar cómo se puede lograr tener `Z` litros de agua en una de las jarras empezando con las dos jarras vacías. Los 8 tipos de acciones que se permiten son llenar una de las jarras con el grifo, llenar una jarra con la otra jarra, vaciar una de las jarras en el lavabo y vaciar una jarra en la otra jarra. La soluciones del problema de las jarras `X-Y-Z` pueden representarse mediante listas de pares `(V a b)` donde `a` es el contenido de la jarra de `X` litros y `b` el de la de `Y` litros. Por ejemplo, una solución del problema `4-3-2` es

  ```
  [J 4 2,J 3 3,J 3 0,J 0 3,J 0 0]
  ```

  es decir, se comienza con las jarras vacías (`J 0 0`), se llena la segunda con el grifo (`J 0 3`), se vacia la segunda en la primera (`J 3 0`), se llena la segunda con el grifo (`J 3 3`) y se llena la primera con la segunda (`J 4 2`).

Diseño de la solución del problema de las jarras

- Para resolver el problema basta crear una instancia de la clase grafo de búsqueda. Para lo cual basta definir los estados y la relación sucesor. Una vez creada la instancia, usando el método por defecto `caminosDesde`, se define la función `jarras` que calcula las soluciones.
- El problema se representa en el módulo `Jarras`, que importa el módulo del grafo de búsqueda en profundidad y las funciones de diferencia de conjuntos (`\\`) y eliminación de duplicados (`nub`):

```
module Jarras where
import Grafo_busq_prof
import Data.List ((\\), nub)
```

Representación del problema de las jarras

- Un problema de las jarras es un par (`(PJ mx my)` donde `mx` es la capacidad de la primera jarra y `my` la de la segunda.

```
data PJarras = PJ Int Int
               deriving (Show,Eq)
```

- Un estado en el problema de las jarras es un par (`J x y`) donde `x` es el contenido de la primera jarra e `y` el de la segunda.

```
data Jarras = J Int Int
              deriving (Show,Eq)
```

Representación del problema de las jarras

Los operadores de un problema de las jarras son

- `llX`: Llenar la primera jarra con el grifo.
- `llY`: Llenar la segunda jarra con el grifo.
- `vaX`: Vaciar la primera jarra en el lavabo.
- `vaY`: Vaciar la segunda jarra en el lavabo.
- `voXY`: Volcar la primera jarra sobre la segunda, quedando la primera vacía o la segunda llena.
- `voYX`: Volcar la segunda jarra sobre la primera, quedando la segunda vacía o la primera llena.

Representación del problema de las jarras

```
ops :: PJarras -> [Jarras -> Jarras]
ops (PJ mx my) =
    [llX, llY, vaX, vaY, voXY, voYX]
    where llX  (J _ y) = J mx  y
          llY  (J x _) = J x   my
          vaX  (J _ y) = J 0   y
          vaY  (J x _) = J x   0
          voXY (J x y) = if s <= my then J 0 (s)
                                    else J (s-my) my
                         where s=x+y
          voYX (J x y) = if s <= mx then J (s) 0
                                    else J mx (s-mx)
                         where s=x+y
```

Solución del problema de las jarras

- Declaración de `Jarras` como instancia de `Grafo`:

```
instance Grafo Jarras where
    suc c = nub [f c | f <- ops (PJ 4 3)] \\ [c]
```

Para cambiar el problema basta modificar la expresión `PJ 4 3`.

- `(jarras z)` es la lista de soluciones del problema de las jarras para obtener `z` litros en alguna de las jarras. Por ejemplo,

```
Jarras> jarras 2
[[J 4 2,J 3 3,J 3 0,J 0 3,J 4 3,J 4 0,J 0 0],
 [J 4 2,J 3 3,J 3 0,J 0 3,J 4 3,J 1 3,J 4 0,J 0 0],
 ...
```

```
jarras :: Int -> [[Jarras]]
jarras z = caminosDesde (J 0 0) test []
    where test (J x y) = x==z || y==z
```

Solución más corta del problema de las jarras

- `(másCorta l)` es la lista más corta de la lista de listas `l`. Por ejemplo,

```
Main*> másCorta [[1,3],[5],[2,3,4]]
[5]
Main*> másCorta (jarras 2)
[J 4 2,J 3 3,J 3 0,J 0 3,J 0 0]
```

```
másCorta :: [[a]] -> [a]
másCorta =
    foldr1 (\x y -> if length x < length y then x else y)
```

Tema 21: Algoritmos de exploración de grafos

Enunciado del problema de los misioneros

- Enunciado: Hay 3 misioneros y 3 caníbales en la orilla izquierda de un río y tienen una barca en la que caben a lo sumo 2 personas. El problema consiste en diseñar un plan para pasar los 6 a la orilla derecha sin que en ningún momento el número de caníbales que haya en cualquiera de las orillas puede superar al número de misioneros.
- Diseño de la solución: Para resolver el problema basta crear una instancia de la clase grafo de búsqueda. Para lo cual basta definir los estados y la relación sucesor. Una vez creada la instancia, usando el método por defecto `caminosDesde`, se define la función `misioneros` que calcula las soluciones.

Representación del problema de los misioneros

- El problema se representa en el módulo `Misioneros`, que importa el módulo del grafo de búsqueda en profundidad y la función `másCorta` del módulo `Jarras`:

```
module Misioneros where
import Grafo_busq_prof
import Jarras (másCorta)
```

- El número de caníbales y de misioneros son números enteros:

```
type Caníbales     = Int
type Misioneros    = Int
```

Representación del problema de los misioneros

- La barca puede estar en la derecha o en la izquierda:

```
data PosiciónBarca = Der | Izq
                   deriving (Show,Eq)
```

- Los estados del problema de los misioneros son triples de la forma `(EM m c b)` donde `m` es el número de misioneros en la orilla izquierda, `c` es el número de caníbales en la orilla izquierda y `b` es la posición de la barca.

```
data EMisioneros = EM Caníbales Misioneros PosiciónBarca
                 deriving (Show,Eq)
```

Representación del problema de los misioneros

- `viajes` es la lista de las distintas formas de viajar los misioneros y caníbales suponiendo que la barca no puede viajar vacía ni llevar a más de dos personas.

```
viajes :: [(Misioneros,Caníbales)]
viajes = [(x,y) | x <- [0..2], y <- [0..2],
                  0 < x+y, x+y <= 2]
```

Para el presente caso

`viajes` $\rightsquigarrow$ `[(0,1),(0,2),(1,0),(1,1),(2,0)]`

Para aumentar la capacidad de la barca basta sustituir 2 por la nueva capacidad.

Representación del problema de los misioneros

- `(esSegura c m)` se verifica si estando `c` caníbales y `m` misioneros en una orilla, los caníbales no se comen a los misioneros.

```
esSegura :: Caníbales -> Misioneros -> Bool
esSegura _ 0 = True
esSegura c m = m >= c
```

Solución del problema de los misioneros

- Declaración de `EMisioneros` como instancia de `Grafo`:

```
instance Grafo EMisioneros where
   suc (EM m c Izq) = [EM m' c' Der |
                       (mb,cb)<- viajes,
                       let m'=m-mb, m'>=0,
                       let c'=c-cb, c'>=0,
                       esSegura c' m',
                       esSegura (3-c') (3-m') ]
   suc (EM m c Der) = [EM m' c' Izq |
                       (mb,cb)<- viajes,
                       let m'=m+mb, m'<=3,
                       let c'=c+cb, c'<=3,
                       esSegura c' m',
                       esSegura (3-c') (3-m') ]
```

Solución del problema de los misioneros

- `misioneros` es la lista de soluciones del problema de los misioneros. Por ejemplo,

```
Misioneros> misioneros
[[EM 0 0 Der,EM 0 2 Izq,EM 0 1 Der,EM 0 3 Izq,
  EM 0 2 Der,EM 2 2 Izq,EM 1 1 Der,EM 3 1 Izq,
  EM 3 0 Der,EM 3 2 Izq,EM 3 1 Der,EM 3 3 Izq],
 [EM 0 0 Der,EM 1 1 Izq,EM 0 1 Der,EM 0 3 Izq,
  EM 0 2 Der,EM 2 2 Izq,EM 1 1 Der,EM 3 1 Izq,
  EM 3 0 Der,EM 3 2 Izq,EM 3 1 Der,EM 3 3 Izq],
 ...
```

```
misioneros :: [[EMisioneros]]
misioneros =
    caminosDesde (EM 3 3 Izq) ((==) (EM 0 0 Der)) []
```

Solución más corta del problema de los misioneros

- `misionerosMásCorta` es la solución más corta del problema de los misioneros. Por ejemplo,

```
Misioneros> misionerosMásCorta
[EM 0 0 Der,EM 1 1 Izq,EM 0 1 Der,EM 0 3 Izq,
 EM 0 2 Der,EM 2 2 Izq,EM 1 1 Der,EM 3 1 Izq,
 EM 3 0 Der,EM 3 2 Izq,EM 2 2 Der,EM 3 3 Izq]
```

```
misionerosMásCorta :: [EMisioneros]
misionerosMásCorta = másCorta misioneros
```