Tema 22: Algoritmos sobre grafos

Informática (2013–14)

José A. Alonso Jiménez

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

Tema 22: Algoritmos sobre grafos

1. El TAD de los grafos
    - Definiciones y terminología sobre grafos
    - Signatura del TAD de los grafos
    - Implementación de los grafos como vectores de adyacencia
    - Implementación de los grafos como matrices de adyacencia

2. Recorridos en profundidad y en anchura
    - Recorrido en profundidad
    - Recorrido en anchura

3. Árboles de expansión mínimos
    - Árboles de expansión mínimos
    - El algoritmo de Kruskal
    - El algoritmo de Prim

Tema 22: Algoritmos sobre grafos

1. El TAD de los grafos
   - Definiciones y terminología sobre grafos
   - Signatura del TAD de los grafos
   - Implementación de los grafos como vectores de adyacencia
   - Implementación de los grafos como matrices de adyacencia

2. Recorridos en profundidad y en anchura

3. Árboles de expansión mínimos

Definiciones y terminología sobre grafos

- Un **grafo G** es un par (V, A) donde V es el conjunto de los **vértices** (o nodos) y A el de las **aristas**.
- Una **arista** del grafo es un par de vértices.
- Un **arco** es una arista dirigida.
- $|\mathbf{V}|$ es el número de vértices.
- $|\mathbf{A}|$ es el número de aristas.
- Un vértice v es **adjacente** a v' si vv' es una arista del grafo.
- Un **grafo ponderado** es un grafo cuyas aristas tienen un peso.

Tema 22: Algoritmos sobre grafos

1. El TAD de los grafos
   - Definiciones y terminología sobre grafos
   - Signatura del TAD de los grafos
   - Implementación de los grafos como vectores de adyacencia
   - Implementación de los grafos como matrices de adyacencia

2. Recorridos en profundidad y en anchura

3. Árboles de expansión mínimos

Signatura del TAD de los grafos

```
creaGrafo   :: (Ix v,Num p) => Orientacion -> (v,v) -> [(v,v,p)] ->
                               Grafo v p
dirigido    :: (Ix v,Num p) => (Grafo v p) -> Bool
adyacentes  :: (Ix v,Num p) => (Grafo v p) -> v -> [v]
nodos       :: (Ix v,Num p) => (Grafo v p) -> [v]
aristas     :: (Ix v,Num p) => (Grafo v p) -> [(v,v,p)]
aristaEn    :: (Ix v,Num p) => (Grafo v p) -> (v,v) -> Bool
peso        :: (Ix v,Num p) => v -> v -> (Grafo v p) -> p
```

Descripción de la signatura del TAD de grafos

- `(creaGrafo d cs as)` es un grafo (dirigido o no, según el valor de o), con el par de cotas cs y listas de aristas as (cada arista es un trío formado por los dos vértices y su peso).
  Ver un ejemplo en la siguiente transparencia.
- `(dirigido g)` se verifica si g es dirigido.
- `(nodos g)` es la lista de todos los nodos del grafo g.
- `(aristas g)` es la lista de las aristas del grafo g.
- `(adyacentes g v)` es la lista de los vértices adyacentes al nodo `v` en el grafo g.
- `(aristaEn g a)` se verifica si a es una arista del grafo g.
- `(peso v1 v2 g)` es el peso de la arista que une los vértices `v1` y `v2` en el grafo g.

Ejemplo de creación de grafos.

```
creaGrafo ND (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)]
```

crea el grafo

```
       12
  1 -------- 2
  | \78     /|
  |  \   32/ |
  |   \   /  |
34|     5    |55
  |   /   \  |
  |  /44   \ |
  | /     93\|
  3 -------- 4
       61
```

Tema 22: Algoritmos sobre grafos

1. El TAD de los grafos
   - Definiciones y terminología sobre grafos
   - Signatura del TAD de los grafos
   - Implementación de los grafos como vectores de adyacencia
   - Implementación de los grafos como matrices de adyacencia

2. Recorridos en profundidad y en anchura

3. Árboles de expansión mínimos

Los grafos como vectores de adyacencia

- Cabecera del módulo:

```
module GrafoConVectorDeAdyacencia
    (Orientacion (..),
     Grafo,
     creaGrafo,  -- (Ix v,Num p) => Orientacion -> (v,v) -> [(v,v,p)] ->
                 --                 Grafo v p
     dirigido,   -- (Ix v,Num p) => (Grafo v p) -> Bool
     adyacentes, -- (Ix v,Num p) => (Grafo v p) -> v -> [v]
     nodos,      -- (Ix v,Num p) => (Grafo v p) -> [v]
     aristas,    -- (Ix v,Num p) => (Grafo v p) -> [(v,v,p)]
     aristaEn,   -- (Ix v,Num p) => (Grafo v p) -> (v,v) -> Bool
     peso        -- (Ix v,Num p) => v -> v -> (Grafo v p) -> p
    ) where
```

- Librerías auxiliares.

```
import Data.Array
```

Los grafos como vectores de adyacencia

- `Orientacion` es D (dirigida) ó ND (no dirigida).

```
data Orientacion = D | ND
                   deriving (Eq, Show)
```

- `(Grafo v p)` es un grafo con vértices de tipo v y pesos de tipo p.

```
data Grafo v p = G Orientacion (Array v [(v,p)])
                 deriving (Eq, Show)
```

Los grafos como vectores de adyacencia

- (creaGrafo o cs as) es un grafo (dirigido o no según el valor de o), con el par de cotas cs y listas de aristas as (cada arista es un trío formado por los dos vértices y su peso). Ver un ejemplo a continuación.

```
creaGrafo :: (Ix v, Num p) =>
             Orientacion -> (v,v) -> [(v,v,p)] -> Grafo v p
creaGrafo o cs vs =
    G o (accumArray
         (\xs x -> xs++[x]) [] cs
         ((if o == D then []
           else [(x2,(x1,p))|(x1,x2,p) <- vs, x1 /= x2]) ++
          [(x1,(x2,p)) | (x1,x2,p) <- vs]))
```

Los grafos como vectores de adyacencia

- ejGrafoND es el grafo que de la página 8. Por ejemplo,

```
ghci> ejGrafoND
G ND array (1,5) [(1,[(2,12),(3,34),(5,78)]),
                  (2,[(1,12),(4,55),(5,32)]),
                  (3,[(1,34),(4,61),(5,44)]),
                  (4,[(2,55),(3,61),(5,93)]),
                  (5,[(1,78),(2,32),(3,44),(4,93)])])
```

```
ejGrafoND = creaGrafo ND (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)]
```

Los grafos como vectores de adyacencia

- ejGrafoD es el mismo grafo que ejGrafoND pero orientando las aristas de menor a mayor. Por ejemplo,

```
ghci> ejGrafoD
G D array (1,5) [(1,[(2,12),(3,34),(5,78)]),
                 (2,[(4,55),(5,32)]),
                 (3,[(4,61),(5,44)]),
                 (4,[(5,93)]),
                 (5,[])])
```

```
ejGrafoD = creaGrafo D (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)]
```

Los grafos como vectores de adyacencia

- (dirigido g) se verifica si g es dirigido. Por ejemplo,

```
dirigido ejGrafoD   ==  True
dirigido ejGrafoND  ==  False
```

```
dirigido :: (Ix v,Num p) => (Grafo v p) -> Bool
dirigido (G o _) = o == D
```

- (adyacentes g v) es la lista de los vértices adyacentes al nodo v en el grafo g. Por ejemplo,

```
adyacentes ejGrafoND 4  ==  [2,3,5]
adyacentes ejGrafoD  4  ==  [5]
```

```
adyacentes :: (Ix v,Num p) => (Grafo v p) -> v -> [v]
adyacentes (G _ g) v = map fst (g!v)
```

Los grafos como vectores de adyacencia

- (nodos g) es la lista de todos los nodos del grafo g. Por ejemplo,

```
nodos ejGrafoND  ==  [1,2,3,4,5]
nodos ejGrafoD   ==  [1,2,3,4,5]
```

```
nodos :: (Ix v,Num p) => (Grafo v p) -> [v]
nodos (G _ g) = indices g
```

- (peso v1 v2 g) es el peso de la arista que une los vértices v1 y v2 en el grafo g. Por ejemplo,

```
peso 1 5 ejGrafoND  ==  78
peso 1 5 ejGrafoD   ==  78
```

```
peso :: (Ix v,Num p) => v -> v -> (Grafo v p) -> p
peso x y (G _ g) = head [c | (a,c) <- g!x , a == y]
```

Los grafos como vectores de adyacencia

- (aristaEn g a) se verifica si a es una arista del grafo g. Por ejemplo,

```
aristaEn ejGrafoND (5,1)  ==  True
aristaEn ejGrafoND (4,1)  ==  False
aristaEn ejGrafoD  (5,1)  ==  False
aristaEn ejGrafoD  (1,5)  ==  True
```

```
aristaEn :: (Ix v,Num p) => (Grafo v p) -> (v,v) -> Bool
aristaEn g (x,y) = y `elem` adyacentes g x
```

Los grafos como vectores de adyacencia

- `(aristas g)` es la lista de las aristas del grafo g. Por ejemplo,

```
ghci> aristas ejGrafoND
[(1,2,12),(1,3,34),(1,5,78),(2,1,12),(2,4,55),(2,5,32),
 (3,1,34),(3,4,61),(3,5,44),(4,2,55),(4,3,61),(4,5,93),
 (5,1,78),(5,2,32),(5,3,44),(5,4,93)]
ghci> aristas ejGrafoD
[(1,2,12),(1,3,34),(1,5,78),(2,4,55),(2,5,32),(3,4,61),
 (3,5,44),(4,5,93)]
```

```
aristas :: (Ix v,Num p) => (Grafo v p) -> [(v,v,p)]
aristas (G o g) =
    [(v1,v2,w) | v1 <- nodos (G o g) , (v2,w) <- g!v1]
```

Tema 22: Algoritmos sobre grafos

1. El TAD de los grafos
   - Definiciones y terminología sobre grafos
   - Signatura del TAD de los grafos
   - Implementación de los grafos como vectores de adyacencia
   - Implementación de los grafos como matrices de adyacencia

2. Recorridos en profundidad y en anchura

3. Árboles de expansión mínimos

Los grafos como matrices de adyacencia

- Cabecera del módulo.

```
module GrafoConMatrizDeAdyacencia
    (Orientacion (..),
     Grafo,
     creaGrafo,  -- (Ix v,Num p) => Orientacion -> (v,v) -> [(v,v,p)] ->
                 --                  Grafo v p
     dirigido,   -- (Ix v,Num p) => (Grafo v p) -> Bool
     adyacentes, -- (Ix v,Num p) => (Grafo v p) -> v -> [v]
     nodos,      -- (Ix v,Num p) => (Grafo v p) -> [v]
     aristas,    -- (Ix v,Num p) => (Grafo v p) -> [(v,v,p)]
     aristaEn,   -- (Ix v,Num p) => (Grafo v p) -> (v,v) -> Bool
     peso        -- (Ix v,Num p) => v -> v -> (Grafo v p) -> p
    ) where
```

- Librerías auxiliares

```
import Data.Array
```

Los grafos como matrices de adyacencia

- `Orientacion` es `D` (dirigida) ó `ND` (no dirigida).

```
data Orientacion = D | ND
                   deriving (Eq, Show)
```

- `(Grafo v p)` es un grafo con vértices de tipo `v` y pesos de tipo `p`.

```
data Grafo v p = G Orientacion (Array (v,v) (Maybe p))
                 deriving (Eq, Show)
```

Los grafos como matrices de adyacencia

- (creaGrafo o cs as) es un grafo (dirigido o no, según el valor de o), con el par de cotas cs y listas de aristas as (cada arista es un trío formado por los dos vértices y su peso). Ver un ejemplo a continuación.

```
creaGrafo :: (Ix v, Num p) => Bool -> (v,v) -> [(v,v,p)]
                              -> Grafo v p
creaGrafo o cs@(l,u) as
    = G o (matrizVacia //
            ([((x1,x2),Just w) | (x1,x2,w) <- as] ++
             if o == D then []
             else [((x2,x1),Just w) | (x1,x2,w) <- as, x1 /= x2]))
      where
      matrizVacia = array ((l,l),(u,u))
                          [((x1,x2),Nothing) | x1 <- range cs,
                                               x2 <- range cs]
```

Los grafos como matrices de adyacencia

- ejGrafoND es el grafo que de la página 8. Por ejemplo,

```
ghci> ejGrafoND
G ND array ((1,1),(5,5))
           [((1,1),Nothing),((1,2),Just 12),((1,3),Just 34),
            ((1,4),Nothing),((1,5),Just 78),((2,1),Just 12),
            ((2,2),Nothing),((2,3),Nothing),((2,4),Just 55),
            ((2,5),Just 32),((3,1),Just 34),((3,2),Nothing),
            ((3,3),Nothing),((3,4),Just 61),((3,5),Just 44),
            ((4,1),Nothing),((4,2),Just 55),((4,3),Just 61),
            ((4,4),Nothing),((4,5),Just 93),((5,1),Just 78),
            ((5,2),Just 32),((5,3),Just 44),((5,4),Just 93),
            ((5,5),Nothing)]
```

```
ejGrafoND = creaGrafo ND (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)]
```

Los grafos como matrices de adyacencia

- ejGrafoD es el mismo grafo que ejGrafoND pero orientando las aristas de menor a mayor. Por ejemplo,

```
ghci> ejGrafoD
G D (array ((1,1),(5,5))
           [((1,1),Nothing),((1,2),Just 12),((1,3),Just 34),
            ((1,4),Nothing),((1,5),Just 78),((2,1),Nothing),
            ((2,2),Nothing),((2,3),Nothing),((2,4),Just 55),
            ((2,5),Just 32),((3,1),Nothing),((3,2),Nothing),
            ((3,3),Nothing),((3,4),Just 61),((3,5),Just 44),
            ((4,1),Nothing),((4,2),Nothing),((4,3),Nothing),
            ((4,4),Nothing),((4,5),Just 93),((5,1),Nothing),
            ((5,2),Nothing),((5,3),Nothing),((5,4),Nothing),
            ((5,5),Nothing)])
```

```
ejGrafoD = creaGrafo D (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)]
```

Los grafos como matrices de adyacencia

- (dirigido g) se verifica si g es dirigido. Por ejemplo,

```
dirigido ejGrafoD   ==  True
dirigido ejGrafoND  ==  False
```

```
dirigido :: (Ix v,Num p) => (Grafo v p) -> Bool
dirigido (G o _) = o == D
```

- (adyacentes g v) es la lista de los vértices adyacentes al nodo v en el grafo g. Por ejemplo,

```
adyacentes ejGrafoND 4  ==  [2,3,5]
adyacentes ejGrafoD  4  ==  [5]
```

```
adyacentes :: (Ix v,Num p) => (Grafo v p) -> v -> [v]
adyacentes (G o g) v =
    [v' | v' <- nodos (G o g), (g!(v,v')) /= Nothing]
```

Los grafos como matrices de adyacencia

- (nodos g) es la lista de todos los nodos del grafo g. Por ejemplo,

```
nodos ejGrafoND  ==  [1,2,3,4,5]
nodos ejGrafoD   ==  [1,2,3,4,5]
```

```
nodos :: (Ix v,Num p) => (Grafo v p) -> [v]
nodos (G _ g) = range (l,u)
    where ((l,_),(u,_)) = bounds g
```

- (peso v1 v2 g) es el peso de la arista que une los vértices v1 y v2 en el grafo g. Por ejemplo,

```
peso 1 5 ejGrafoND  ==  78
peso 1 5 ejGrafoD   ==  78
```

```
peso :: (Ix v,Num p) => v -> v -> (Grafo v p) -> p
peso x y (G _ g)  = w where (Just w) = g!(x,y)
```

Los grafos como matrices de adyacencia

- (aristaEn g a) se verifica si a es una arista del grafo g. Por ejemplo,

```
aristaEn ejGrafoND (5,1)  ==  True
aristaEn ejGrafoND (4,1)  ==  False
```

```
aristaEn :: (Ix v,Num p) => (Grafo v p) -> (v,v) -> Bool
aristaEn (G _o g) (x,y)= (g!(x,y)) /= Nothing
```

Los grafos como matrices de adyacencia

- `(aristas g)` es la lista de las aristas del grafo g. Por ejemplo,

```
ghci> aristas ejGrafoD
[(1,2,12),(1,3,34),(1,5,78),(2,4,55),(2,5,32),(3,4,61),
 (3,5,44),(4,5,93)]
ghci> aristas ejGrafoND
[(1,2,12),(1,3,34),(1,5,78),(2,1,12),(2,4,55),(2,5,32),
 (3,1,34),(3,4,61),(3,5,44),(4,2,55),(4,3,61),(4,5,93),
 (5,1,78),(5,2,32),(5,3,44),(5,4,93)]
```

```haskell
aristas :: (Ix v,Num p) => (Grafo v p) -> [(v,v,p)]
aristas g@(G o e) = [(v1,v2,extrae(e!(v1,v2)))
                    | v1 <- nodos g,
                      v2 <- nodos g,
                      aristaEn g (v1,v2)]
    where extrae (Just w) = w
```

Tema 22: Algoritmos sobre grafos

1. El TAD de los grafos

2. Recorridos en profundidad y en anchura
   - Recorrido en profundidad
   - Recorrido en anchura

3. Árboles de expansión mínimos

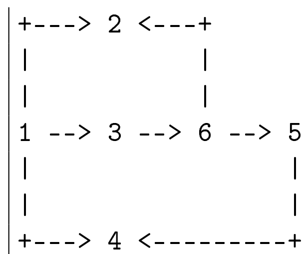
Recorrido en profundidad

- Importaciones de librerías auxiliares.

```
-- Nota: Elegir una implementación de los grafos.
import GrafoConVectorDeAdyacencia
-- import GrafoConMatrizDeAdyacencia
```

Recorrido en profundidad

- En los ejemplos se usará el grafo g



que se define por

```
g = creaGrafo D (1,6)
              [(1,2,0),(1,3,0),(1,4,0),(3,6,0),
               (5,4,0),(6,2,0),(6,5,0)]
```

Procedimiento elemental de recorrido en profundidad

- `(recorridoEnProfundidad i g)` es el recorrido en profundidad del grafo `g` desde el vértice `i`. Por ejemplo,

```
recorridoEnProfundidad 1 g  ==  [1,2,3,6,5,4]
```

```
recorridoEnProfundidad i g = rp [i] []
    where
      rp [] vis    = vis
      rp (c:cs) vis
          | c `elem` vis = rp cs vis
          | otherwise    = rp ((adyacentes g c)++cs)
                              (vis++[c])
```

Procedimiento elemental de recorrido en profundidad

- Traza del cálculo de (recorridoEnProfundidad 1 g)

```
recorridoEnProfundidad 1 g
= rp [1]     []
= rp [2,3,4] [1]
= rp [3,4]   [1,2]
= rp [6,4]   [1,2,3]
= rp [2,5,4] [1,2,3,6]
= rp [5,4]   [1,2,3,6]
= rp [4,4]   [1,2,3,6,5]
= rp [4]     [1,2,3,6,5,4]
= rp []      [1,2,3,6,5,4]
= [1,2,3,6,5,4]
```

Recorrido en profundidad con acumuladores

- (recorridoEnProfundidad' i g) es el recorrido en profundidad del grafo, usando la lista de los visitados como acumulador. Por ejemplo,

```
recorridoEnProfundidad' 1 g  ==  [1,2,3,6,5,4]
```

```
recorridoEnProfundidad' i g = reverse (rp [i] [])
    where
      rp [] vis     = vis
      rp (c:cs) vis
          | c `elem` vis = rp cs vis
          | otherwise    = rp ((adyacentes g c)++cs)
                              (c:vis)
```

Recorrido en profundidad con acumuladores

- Traza del cálculo de (recorridoEnProfundidad' 1 g)

```
recorridoEnProfundidad' 1 g
= reverse (rp [1]     [])
= reverse (rp [2,3,4] [1])
= reverse (rp [3,4]   [2,1])
= reverse (rp [6,4]   [3,2,1])
= reverse (rp [2,5,4] [6,3,2,1])
= reverse (rp [5,4]   [6,3,2,1])
= reverse (rp [4,4]   [5,6,3,2,1])
= reverse (rp [4]     [4,5,6,3,2,1])
= reverse (rp []      [4,5,6,3,2,1])
= reverse [4,5,6,3,2,1]
= [1,2,3,6,5,4]
```

Tema 22: Algoritmos sobre grafos

1. El TAD de los grafos

2. Recorridos en profundidad y en anchura
   - Recorrido en profundidad
   - Recorrido en anchura

3. Árboles de expansión mínimos

Recorrido en anchura

- Importaciones de librerías auxiliares.

```
-- Nota: Elegir una implementación de los grafos.
import GrafoConVectorDeAdyacencia
-- import GrafoConMatrizDeAdyacencia
```

Procedimiento elemental de recorrido en anchura

- `(recorridoEnAnchura i g)` es el recorrido en anchura del grafo g desde el vértice `i`. Por ejemplo,

```
recorridoEnAnchura 1 g  ==  [1,4,3,2,6,5]
```

```
recorridoEnAnchura i g = reverse (ra [i] [])
    where
      ra [] vis      = vis
      ra (c:cs) vis
          | c `elem` vis = ra cs vis
          | otherwise    = ra (cs ++ adyacentes g c)
                              (c:vis)
```

Procedimiento elemental de recorrido en anchura

- Traza del cálculo de (`recorridoEnAnchura 1 g`)

```
RecorridoEnAnchura 1 g
= ra [1]      []
= ra [2,3,4]  [1]
= ra [3,4]    [2,1]
= ra [4,6]    [3,2,1]
= ra [6]      [4,3,2,1]
= ra [2,5]    [6,4,3,2,1]
= ra [5]      [6,4,3,2,1]
= ra [4]      [5,6,4,3,2,1]
= ra []       [5,6,4,3,2,1]
= [1,2,3,4,6,5]
```

Tema 22: Algoritmos sobre grafos

1. El TAD de los grafos

2. Recorridos en profundidad y en anchura

3. Árboles de expansión mínimos
   Árboles de expansión mínimos
   El algoritmo de Kruskal
   El algoritmo de Prim

Árboles de expansión mínimos

- Sea $G = (V, A)$ un grafo conexo no orientado en el que cada arista tiene un peso no negativo. Un **árbol de expansión mínimo** de G es un subgrafo $G' = (V, A')$ que conecta todos los vértices de G y tal que la suma de sus pesos es mínima.
- **Aplicación:** Si los vértices representan ciudades y el coste de una arista $\{a, b\}$ es el construir una carretera de a a b, entonces un árbol de expansión mínimo representa el modo de enlazar todas las ciudades mediante una red de carreteras de coste mínimo.

Árboles de expansión mínimos

- Terminología de algoritmos voraces: Sea $G = (V, A)$ un grafo y T un conjunto de aristas de G.
  - T es una **solución** si es un grafo de expansión.
  - T es **completable** si no tiene ciclos.
  - T es **prometedor** si es completable y puede ser completado hasta llegar a una solución óptima.
  - Una arista **toca** un conjunto de vértices B si exactamente uno de sus extremos pertenece a B.
- **Teorema:** Sea $G = (V, A)$ un grafo conexo no orientado cuyas aristas tienen un peso asociado. Sea B un subjconjunto propio del conjunto de vértices V y T un conjunto prometedor de aristas tal que ninguna arista de T toca a B. Sea e una arista de peso mínimo de entre todas las que tocan a B. Entonces $(T \cup \{e\})$ es prometedor.

Tema 22: Algoritmos sobre grafos

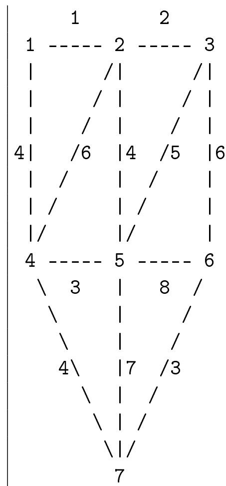
1. El TAD de los grafos

2. Recorridos en profundidad y en anchura

3. Árboles de expansión mínimos
   - Árboles de expansión mínimos
   - El algoritmo de Kruskal
   - El algoritmo de Prim

El algoritmo de Kruskal

Para los ejemplos se considera el siguiente grafo:



El algoritmo de Kruskal

- Aplicación del algoritmo de Kruskal al grafo anterior:

```
Etapa Arista Componentes conexas
0            {1} {2} {3} {4} {5} {6} {7}
1     {1,2}  {1,2} {3} {4} {5} {6} {7}
2     {2,3}  {1,2,3} {4} {5} {6} {7}
3     {4,5}  {1,2,3} {4,5} {6} {7}
4     {6,7}  {1,2,3} {4,5} {6,7}
5     {1,4}  {1,2,3,4,5} {6,7}
6     {2,5}  arista rechazada
7     {4,7}  {1,2,3,4,5,6,7}
```

- El árbol de expansión mínimo contiene las aristas no rechazadas:

```
{1,2}, {2,3}, {4,5}, {6,7}, {1,4} y {4,7}.
```

El algoritmo de Kruskal

- Librerías auxiliares.

```
-- Nota: Seleccionar una implementación del TAD grafo.
import GrafoConVectorDeAdyacencia
-- import GrafoConMatrizDeAdyacencia

-- Nota: Seleccionar una implementación del TAD tabla.
-- import TablaConFunciones
import TablaConListasDeAsociacion
-- import TablaConMatrices

import Data.List
import Data.Ix
```

El algoritmo de Kruskal

- Grafos usados en los ejemplos.

```
g1 :: Grafo Int Int
g1 = creaGrafo D (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)]

g2 :: Grafo Int Int
g2 = creaGrafo D (1,5) [(1,2,13),(1,3,11),(1,5,78),
                        (2,4,12),(2,5,32),
                        (3,4,14),(3,5,44),
                        (4,5,93)]
```

El algoritmo de Kruskal

- (kruskal g) es el árbol de expansión mínimo del grafo g calculado mediante el algoritmo de Kruskal. Por ejemplo,

```
kruskal g1  ==  [(55,2,4),(34,1,3),(32,2,5),(12,1,2)]
kruskal g2  ==  [(32,2,5),(13,1,2),(12,2,4),(11,1,3)]
```

```
kruskal :: (Ix v, Num p, Ord p) => Grafo v p -> [(p,v,v)]
kruskal g = kruskal' cola                          -- Cola de prioridad
                     (tabla [(x,x) | x <- nodos g]) -- Tabla de raices
                     []                             -- Árbol de expansión
                     ((length (nodos g)) - 1)       -- Aristas por
                                                    -- colocar
    where cola = sort [(p,x,y) | (x,y,p) <- aristas g]

kruskal' ((p,x,y):as) t ae n
    | n==0        = ae
    | actualizado = kruskal' as t' ((p,x,y):ae) (n-1)
    | otherwise   = kruskal' as t  ae           n
    where (actualizado,t') = buscaActualiza (x,y) t
```

El algoritmo de Kruskal

- (raiz t n) es la raíz de n en la tabla t. Por ejemplo,

```
> raiz (crea [(1,1),(3,1),(4,3),(5,4),(2,6),(6,6)]) 5
1
> raiz (crea [(1,1),(3,1),(4,3),(5,4),(2,6),(6,6)]) 2
6
```

```
raiz:: Eq n => Tabla n n -> n -> n
raiz t x | v == x      = v
         | otherwise = raiz t v
         where v = valor t x
```

El algoritmo de Kruskal

- `(buscaActualiza a t)` es el par formado por `False` y la tabla `t`, si los dos vértices de la arista `a` tienen la misma raíz en `t` y el par formado por `True` y la tabla obtenida añadiéndole a `t` la arista formada por el vértice de `a` de mayor raíz y la raíz del vértice de `a` de menor raíz. Por ejemplo,

```
ghci> let t = crea [(1,1),(2,2),(3,1),(4,1)]
ghci> buscaActualiza (2,3) t
(True,Tbl [(1,1),(2,1),(3,1),(4,1)])
ghci> buscaActualiza (3,4) t
(False,Tbl [(1,1),(2,2),(3,1),(4,1)])
```

```
buscaActualiza :: (Eq n, Ord n) => (n,n) -> Tabla n n
                                -> (Bool,Tabla n n)
buscaActualiza (x,y) t
    | x' == y'  = (False, t)
    | y' <  x'  = (True, modifica (x,y') t)
    | otherwise = (True, modifica (y,x') t)
    where x' = raiz t x
          y' = raiz t y
```

Tema 22: Algoritmos sobre grafos

1. El TAD de los grafos

2. Recorridos en profundidad y en anchura

3. Árboles de expansión mínimos
   - Árboles de expansión mínimos
   - El algoritmo de Kruskal
   - El algoritmo de Prim

El algoritmo de Prim

- (prim g) es el árbol de expansión mínimo del grafo g calculado mediante el algoritmo de Prim. Por ejemplo,

```
prim g1  ==  [(55,2,4),(34,1,3),(32,2,5),(12,1,2)]
prim g2  ==  [(32,2,5),(12,2,4),(13,1,2),(11,1,3)]
```

```
prim :: (Ix v, Num p, Ord p) => Grafo v p -> [(p,v,v)]
prim g = prim' [n]                -- Nodos colocados
               ns                 -- Nodos por colocar
               []                 -- Árbol de expansión
               (aristas g)        -- Aristas del grafo
         where (n:ns) = nodos g

prim' t [] ae as = ae
prim' t r  ae as = prim' (v':t) (delete v' r) (e:ae) as
    where e@(c,u', v') = minimum [(c,u,v)| (u,v,c) <- as,
                                           elem u t,
                                           elem v r]
```