Tema 6

Grafos y Estructuras de Partición

Objetivos

- Estudio de la representación de una relación binaria entre los datos de una colección mediante la estructura ***Grafo*** y algunas de sus aplicaciones más significativas
- Reutilización de modelos ya estudiados para representar grafos y para explorarlos

- Desarrollo de estructuras de datos eficientes para agrupar *n* elementos distintos en una colección de *k* conjuntos disjuntos $S = \{S_1, S_2, \ldots, S_k\}$ con dos tipos de operaciones:
  - Unión de dos conjuntos disjuntos
  - Búsqueda para saber a qué conjunto pertenece un elemento

Bibliografía

Michael T. Goodrich and Roberto Tamassia. "*Data Structures & Algorithms in Java*" (4th edition), John Wiley & Sons, 2005

(capítulo 13 y apartado 6 del capítulo 11)

Contenidos

1. Introducción
2. Representación de grafos
3. Recorridos sobre grafos
4. Árbol de recubrimiento de coste mínimo (Kruskal)
5. Estructuras de partición
6. Implementación de un Grafo Dirigido mediante listas de adyacencia
7. Caminos de mínimo peso (Dijkstra)
8. Órdenes topológicos

1. Introducción

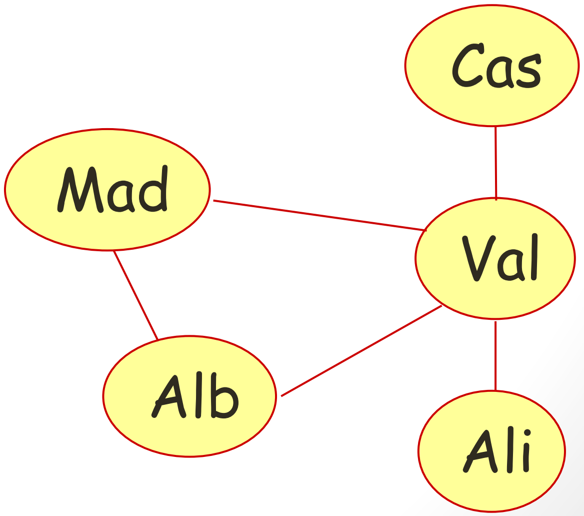
Relaciones entre los datos de la colección

- Relación binaria entre los datos de la colección:
  - Una relación R sobre un conjunto S se define como un conjunto de pares (a, b) / $a, b \in S$
  - Si $(a, b) \in R$, se escribe "$a\ R\ b$" y denota que a está relacionado con b

Ejemplo: grafo cuyos ***vértices*** (S) se relacionan vía ***aristas*** (R)

S = {Cas, Val, Ali, Mad, Alb}

R = {(Cas,Val), (Val,Ali), (Val,Alb),
(Mad,Alb), (Mad,Val)}

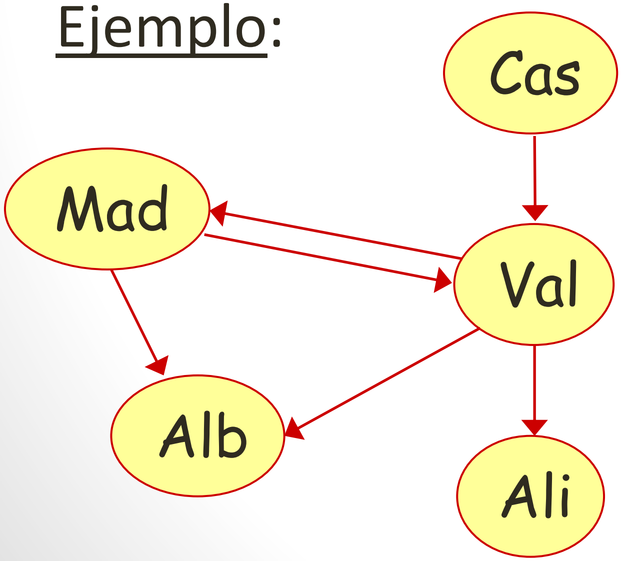


1. Introducción

Grafos dirigidos (Digrafos)

- Un **grafo dirigido** (*gd*) es un par $G = (V, A)$
  - V es un conjunto finito de ***vértices*** (o nodos o puntos)
  - A es un conjunto de ***aristas*** (o arcos) dirigidas, donde una a*rista* es un par ordenado de vértices (u, v): $u \rightarrow v$

<u>Ejemplo</u>:



$V = \{\text{Cas, Val, Ali, Alb, Mad}\}$

$|V| = 5$

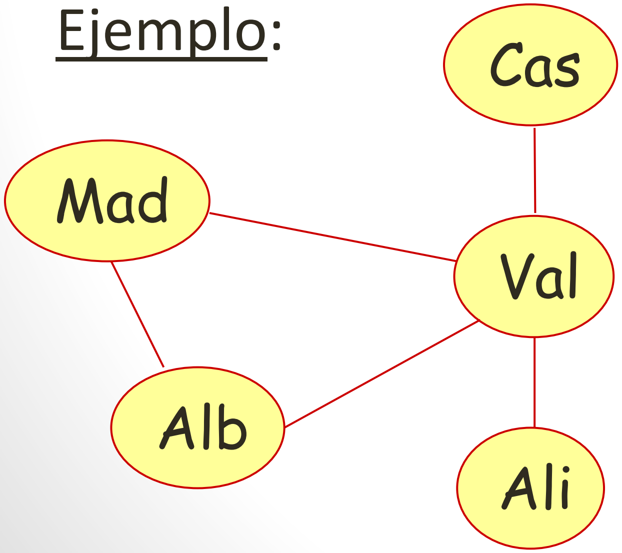
$A = \{(\text{Cas,Val}), (\text{Val,Mad}),(\text{Val,Alb}), (\text{Val,Ali}), (\text{Mad,Val}), (\text{Mad,Alb})\}$

$|A| = 6$

1. Introducción
Grafos no dirigidos (Grafos)

- Un **grafo no dirigido** (*gnd*) es un par $G = (V, A)$
  - V es un conjunto finito de ***vértices***
  - A es un conjunto de ***aristas*** (o arcos) no dirigidas, donde una a*rista* es un par no ordenado de vértices $(u,v) = (v,u)$, $u \neq v$: u —— v

Ejemplo:

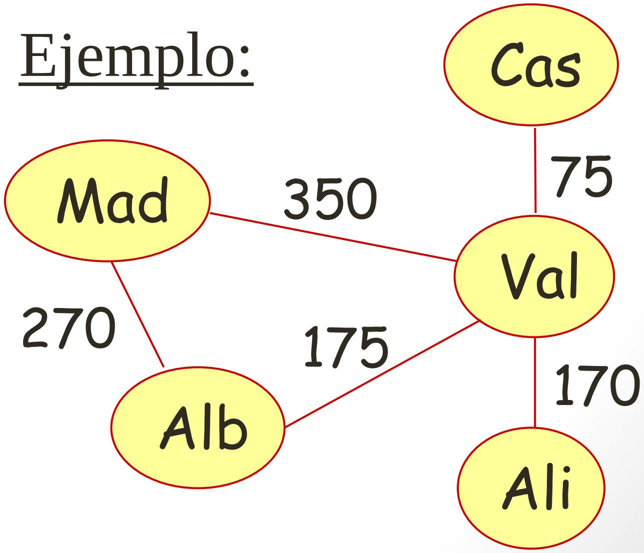


V = {Cas, Val, Ali, Alb, Mad}
$|V| = 5$
A = {(Cas,Val), (Val,Ali), (Val,Mad),
(Val,Alb), (Mad,Alb)}
$|A| = 5$

1. Introducción
Grafos etiquetados

- Un **grafo *etiquetado*** es un grafo $G = (V, A)$ sobre el que se define una función f: $A \rightarrow E$, donde E es un conjunto cuyas componentes se llaman ***etiquetas***
  - *Nota*: la función de etiquetado se puede definir también sobre V, el conjunto de vértices
- Un **grafo *ponderado*** es un grafo etiquetado con números reales ($A \equiv \Re$)

<u>Ejemplo:</u>



1. Introducción

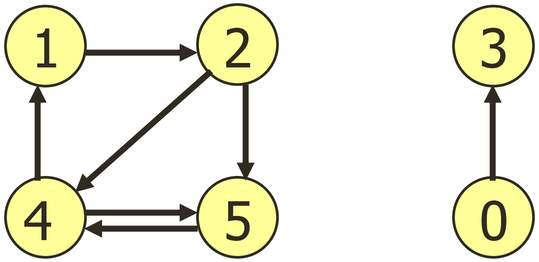
Relaciones de adyacencia

- Sea $G = (V, A)$ un grafo. Si $(u, v) \in A$, decimos que el vértice u es adyacente al vértice v

Ejemplo con el vértice 1:

1 es adyacente al 2

1 no es adyacente al 4



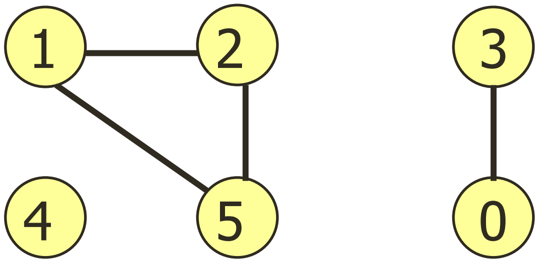
- En un grafo no dirigido la relación es simétrica

1. Introducción

Grado de un vértice

- El ***grado de un vértice*** en un grafo no dirigido es el número de aristas que inciden sobre él (o de vértices adyacentes)

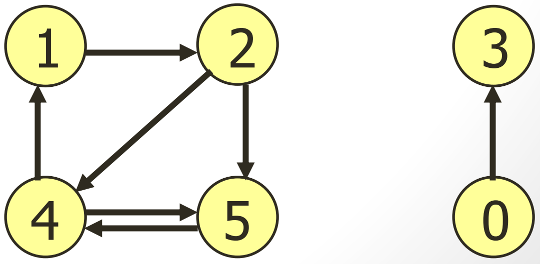
<u>Ejemplo</u>: el grado del vértice 2 es 2



- El grado de un vértice en un grafo dirigido es la suma de:
  - El número de aristas que salen de él (*grado de salida*)
  - El número de aristas que entran en él (*grado de entrada*)

<u>Ejemplo</u>: el grado de entrada de 2 es 1
\+ el grado de salida de 2 es 2

el grado del vértice 2 es 3

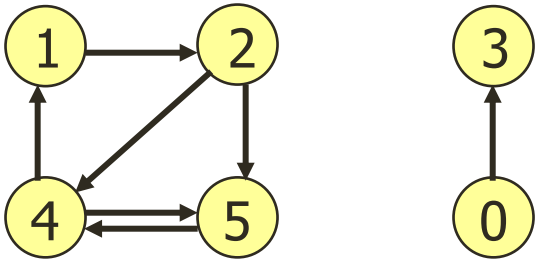


1. Introducción

Grado de un grafo

- El ***grado de un grafo*** es el de su vértice de grado máximo

Ejemplo:



El grado de este grafo es 4 (el grado del vértice 4)

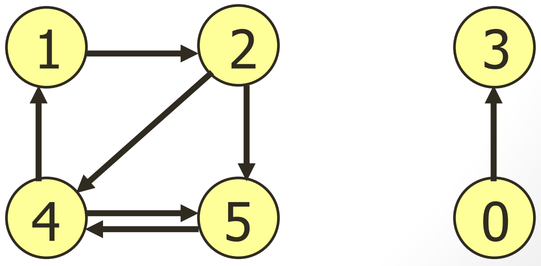
1. Introducción
Caminos

- Un ***camino*** de longitud k desde u a u' en un grafo $G = (V, A)$ es una secuencia de vértices $\langle v_0, v_1, ..., v_k\rangle$ tal que:
  - $v_0 = u$ y $v_k = u'$
  - $\forall\ i : 1...k : (v_{i-1}, v_i) \in A$
  - La longitud k del camino es el número de aristas
  - La longitud del camino con pesos es la suma de los pesos de las aristas que forman el camino
- Si hay un camino P desde u hasta u', decimos que u' es ***alcanzable*** desde u vía P

1. Introducción

Caminos simples y ciclos

- Un ***ciclo*** es un camino $\langle v_0, v_1, ..., v_k \rangle$ que:
  - Empieza y acaba en el mismo vértice ($v_0 = v_k$)
  - Contiene al menos una arista
- Un camino o ciclo es ***simple*** si todos sus vértices son distintos
- Un ***bucle*** es un ciclo de longitud 1. No se admiten bucles en grafos simples.
- Un grafo es ***acíclico*** si no contiene ciclos



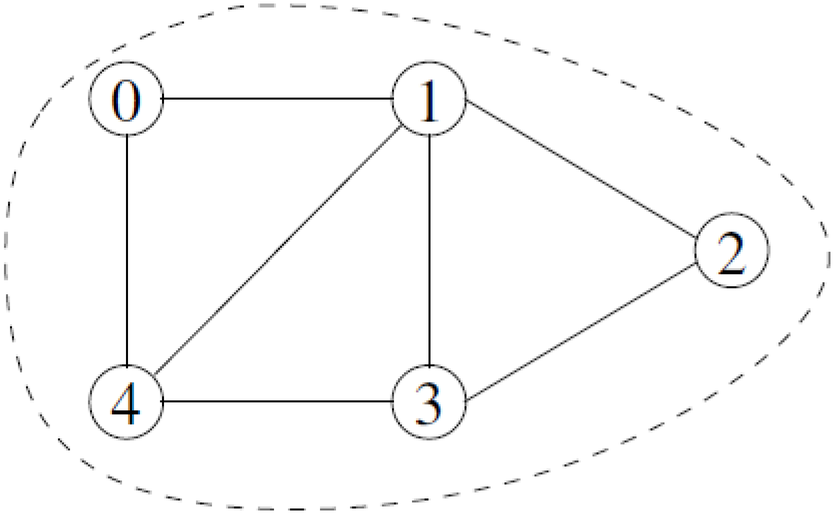
Ejemplo:

$\langle 1, 2, 5, 4, 1 \rangle$ es un ciclo de longitud 4

1. Introducción

Componentes conexas

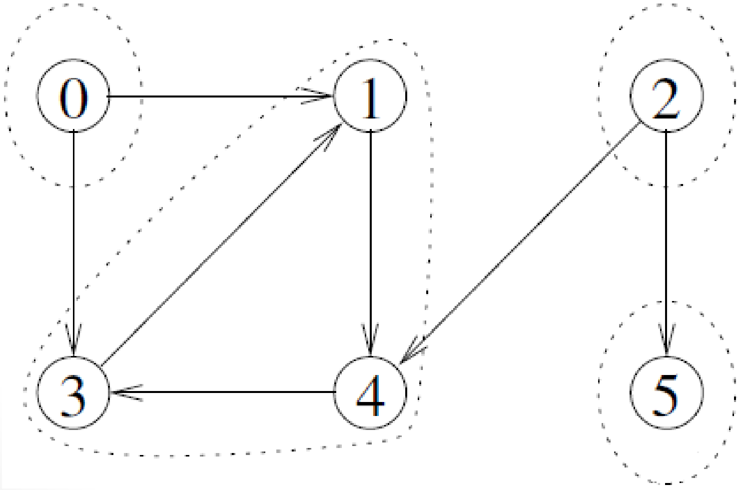
- Las ***componentes conexas*** en un grafo no dirigido son las clases de equivalencia de vértices según la relación "*ser alcanzable*"
  - Un grafo no dirigido es conexo si $\forall u, v \in V$, v es alcanzable desde u. Es decir, si tiene una única componente conexa



Ejemplo: grafo no dirigido conexo

1. Introducción
Componentes conexas

- Las ***componentes fuertemente conexas*** en un grafo dirigido son las clases de equivalencia de vértices según la relación "*ser mutuamente alcanzable*"
  - Un grafo dirigido es fuertemente conexo si $\forall$u, v $\in$ *V*, *v* es alcanzable desde *u*



Ejemplo: grafo dirigido con 4 componentes fuertemente conexas

2. Representación de grafos
Representaciones

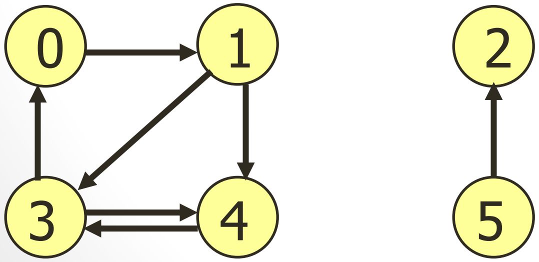
- Existen dos formas fundamentales de representar un grafo:
  - Si el grafo es ***disperso*** ($|A| <<< |V|^2$): <u>listas de adyacencia</u>
  - Si el grafo es ***denso*** ($|A| \approx |V|^2$): <u>matriz de adyacencias</u>

2. Representación de grafos

Matriz de adyacencias

- Un grafo $G = (V, A)$ se representa como una ***matriz*** de $|V| \times |V|$ elementos de tipo *boolean*
  - Si $(u, v) \in A \rightarrow G[u, v] = true$ (si no $G[u, v] = false$)
  - Coste espacial $O(|V|^2)$
  - Tiempo de acceso $O(1)$

Ejemplo:



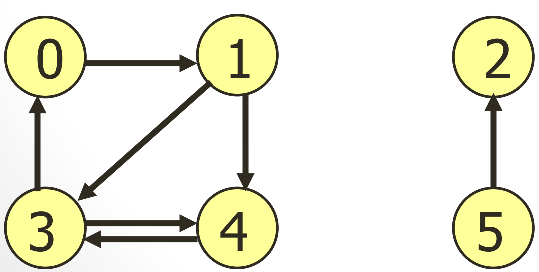
	0	1	2	3	4	5
0	false	true	false	false	false	false
1	false	false	false	true	true	false
2	false	false	false	false	false	false
3	true	false	false	false	true	false
4	false	false	false	true	false	false
5	false	false	true	false	false	false

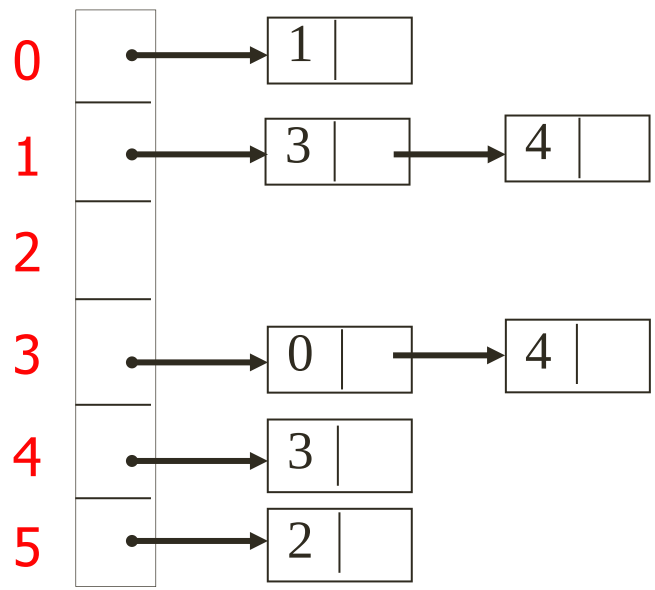
2. Representación de grafos

Listas de adyacencia

- Un grafo $G = (V, A)$ se representa como un ***array*** de $|V|$ ***listas*** de vértices
  - $G[v]$, $v \in V$, es la lista de los vértices adyacentes a v
  - Coste espacial $O(|V| + |A|)$
  - Tiempo de acceso O(grado de G)

Ejemplo:





2. Representación de grafos
Funcionalidad básica de un grafo

- Vamos a crear la clase abstracta *Grafo* para que defina la funcionalidad básica de un grafo
  - No utilizamos una *interfaz* ya que escribiremos el código de algunos métodos, como los recorridos, que son independientes de la implementación utilizada y del tipo de grafo
- La funcionalidad básica incluye:
  - Modificadores: inserción de aristas (con o sin pesos)
  - Consultores: número de vértices/aristas, búsqueda de aristas
  - Recorridos: en profundidad y en anchura

2. Representación de grafos
La clase Grafo: consultores

```
public abstract class Grafo {
  // Devuelve el número de vértices del grafo
  public abstract int numVertices();

  // Devuelve el número de aristas del grafo
  public abstract int numAristas();

  // Comprueba la existencia de la arista (i,j)
  public abstract boolean existeArista(int i, int j);

  // Recupera el peso de la arista (i,j)
  public abstract double pesoArista(int i, int j);

  // Devuelve una lista con los adyacentes del vértice i
  public abstract ListaConPI<Adyacente> adyacentesDe(int i);
```

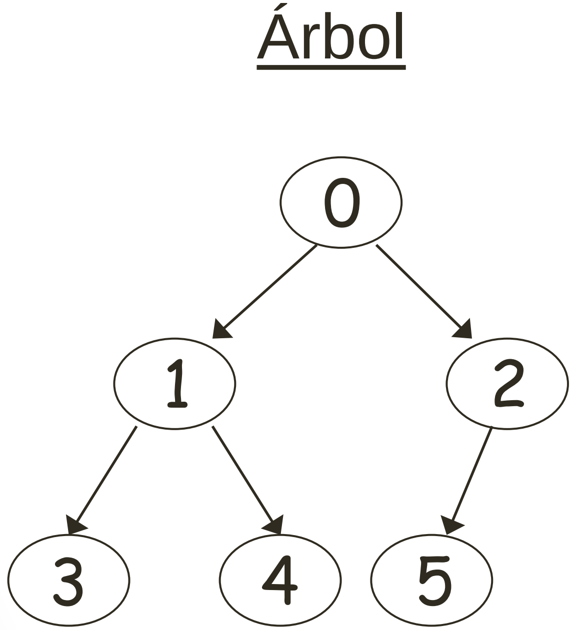
2. Representación de grafos
La clase Grafo: modificadores

```
public abstract class Grafo {
  ...

  // Añade la arista (i,j) a un grafo sin pesos
  public abstract void insertarArista(int i, int j);

  // Añade la arista (i,j) con peso p a un grafo ponderado
  public abstract void insertarArista(int i, int j,
                                      double p);
```

- El método para insertar aristas está sobrecargado para permitir la inserción de aristas tanto en un grafo sin pesos como en uno ponderado

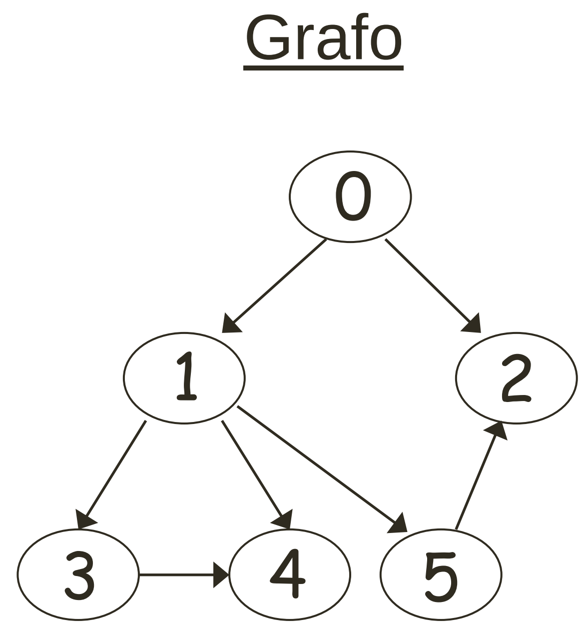
3. Recorridos sobre grafos
Recorrido en profundidad o DFS

- Generalización del recorrido en *PreOrden* de un árbol:



PreOrden: Padre, Izq, Der

0, 1, 3, 4, 2, 5



0, 1, 3, 4, 5, 2

Precaución para no repetir ningún vértice

3. Recorridos sobre grafos
Implementación del recorrido DFS (1/2)

```
public abstract class Grafo {
  // El recorrido en profundidad necesita dos atributos
  protected int visitados[];  // Para no repetir vértices
  protected int ordenVisita;  // Orden de visita de los
                              // vértices
  // Recorrido en profundidad (DFS): devuelve un array con
  // los códigos de los vértices recorridos según DFS
  public int[] toArrayDFS() {
    int res[] = new int[numVertices()];
    visitados = new int[numVertices()];
    ordenVisita = 0;
    for (int  i = 0; i < numVertices(); i++)
      if (visitados[i] == 0) toArrayDFS(i, res);
    return res;
  }
```

Se inicializa automáticamente a cero

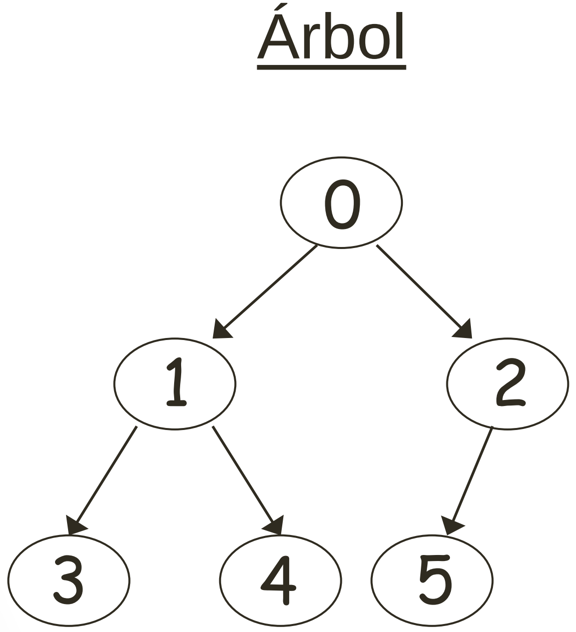
3. Recorridos sobre grafos

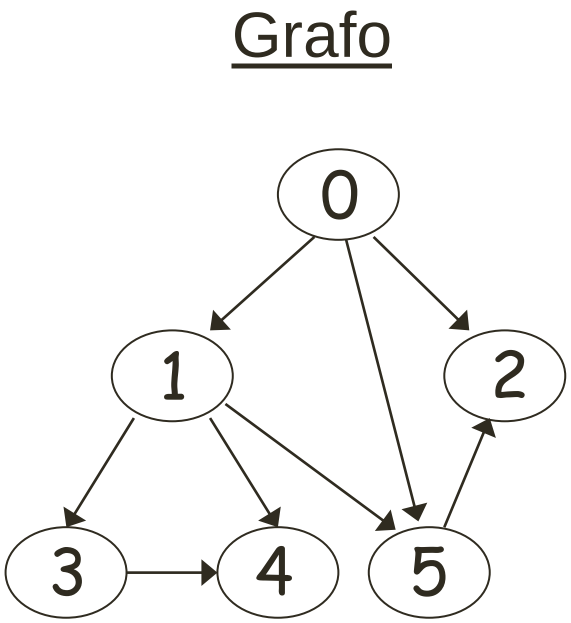
Implementación del recorrido DFS (2/2)

```
// Método recursivo para el recorrido en profundidad
protected void toArrayDFS(int origen, int res[]) {
  // Añadimos el vértice origen y lo marcamos como visitado
  res[ordenVisita++] = origen;
  visitados[origen] = 1;
  // Recorremos los adyacentes del vértice origen
  ListaConPI<Adyacente> l = adyacentesDe(origen);
  for (l.inicio(); !l.esFin(); l.siguiente()) {
    Adyacente a = l.recuperar();
    if (visitados[a.destino] == 0)
       toArrayDFS(a.destino, res);
  }
}
```

3. Recorridos sobre grafos
Recorrido en anchura o BFS

- Generalización del recorrido por niveles de un árbol:





Por niveles

0, 1, 2, 3, 4, 5

0, 1, 5, 2, 3, 4

3. Recorridos sobre grafos
Implementación del recorrido BFS (1/2)

```
public abstract class Grafo {
  ... // Además de los atributos visitados y ordenVisita, el
      // recorrido BFS requiere una Cola auxiliar pues el
      // recorrido es iterativo
  protected Cola<Integer> q;

  // Recorrido en anchura (BFS)
  public int[] toArrayBFS() {
    int res[] = new int[numVertices()];
    visitados = new int[numVertices()];
    ordenVisita = 0;
    q = new ArrayCola<Integer>();
    for (int  i = 0; i < numVertices(); i++)
      if (visitados[i] == 0) toArrayBFS(i, res);
    return res;
  }
```

3. Recorridos sobre grafos

Implementación del recorrido BFS (2/2)

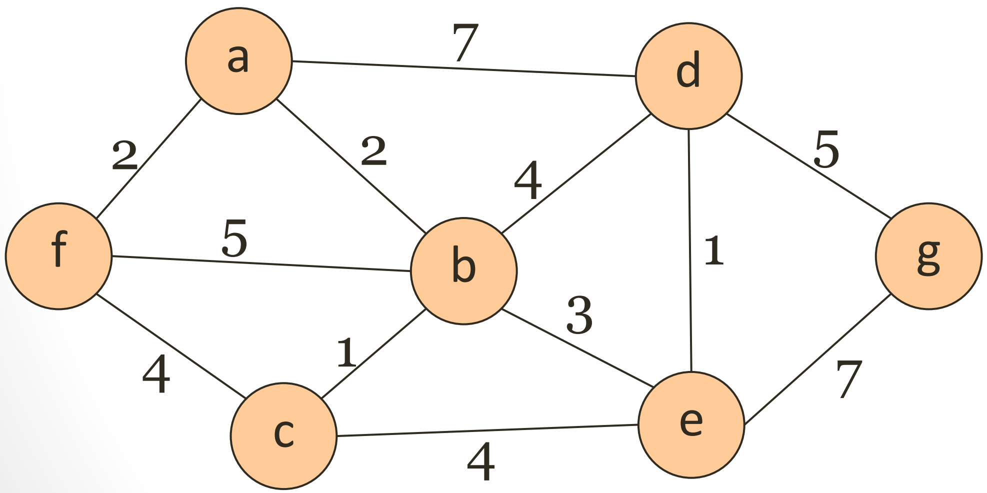
```
protected void toArrayBFS(int origen, int res[]) {
  res[ordenVisita++] = origen;
  visitados[origen] = 1;
  q.encolar(origen);
  while (!q.esVacia()) {
    int u = q.desencolar().intValue();
    ListaConPI<Adyacente> l = adyacentesDe(u);
    for (l.inicio(); !l.esFin(); l.siguiente()) {
      Adyacente a = l.recuperar();
      if (visitados[a.destino] == 0) {
        res[ordenVisita++] = a.destino;
        visitados[a.destino] = 1;
        q.encolar(a.destino);
      }
    }
  }
}
```

4. Árbol generador minimal
Introducción

- Un grafo no dirigido es **conexo** si cualquier par de vértices está conectado por un camino
- Un grafo no dirigido acíclico y conexo es un **árbol**
- Un árbol generador (o árbol de recubrimiento) de un grafo (V, A) es un árbol (V', A') tal que:
  - $V' = V$
  - $A' \subseteq A$
- El problema de obtener el árbol generador minimal es muy importante por sus numerosas aplicaciones (diseño de redes, trazado de carreteras, astronomía, medicina, etc.)

4. Árbol generador minimal
Ejemplo

- Los vértices del siguiente grafo representan los puntos de luz en una fábrica, y las aristas la longitud de cable necesaria para unir dos puntos de luz:



Problema:

¿Cómo unir todos los puntos de luz utilizando la menor cantidad posible de cable?

4. Árbol generador minimal
Algoritmo de Kruskal

Paso 1: guardar las aristas en una cola de prioridad

(una arista será menor que otra si tiene menor coste)

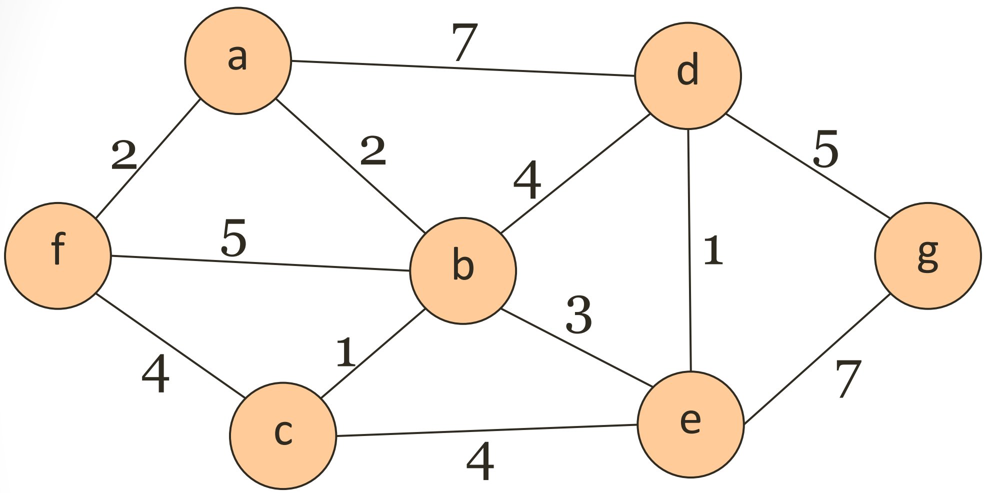
Paso 2: partimos de un grafo sin aristas (sólo con los vértices)

Paso 3: *mientras* |A| < |V| - 1 *hacer*:

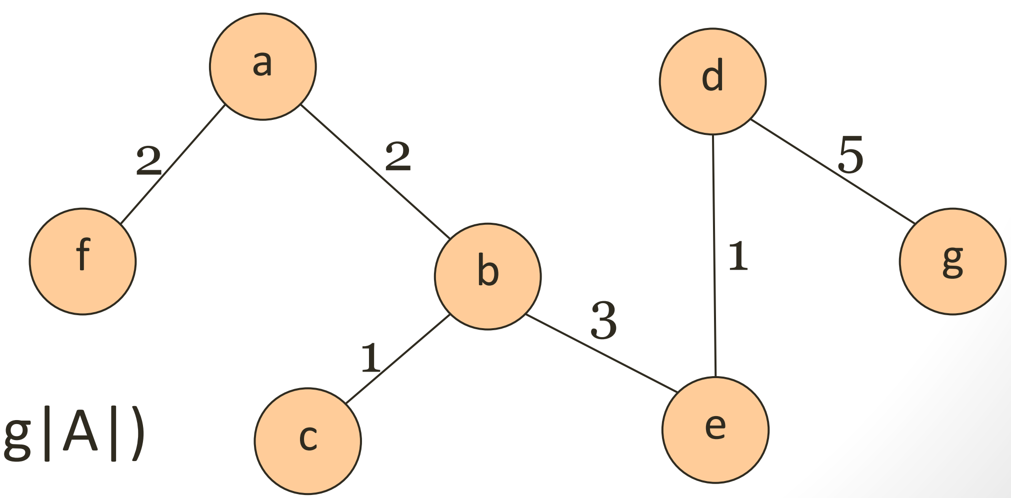
- Recuperar y eliminar la arista menor coste de la cola de prioridad
- Incluir la arista en el grafo si no provoca ciclos

4. Árbol generador minimal

Algoritmo de Kruskal

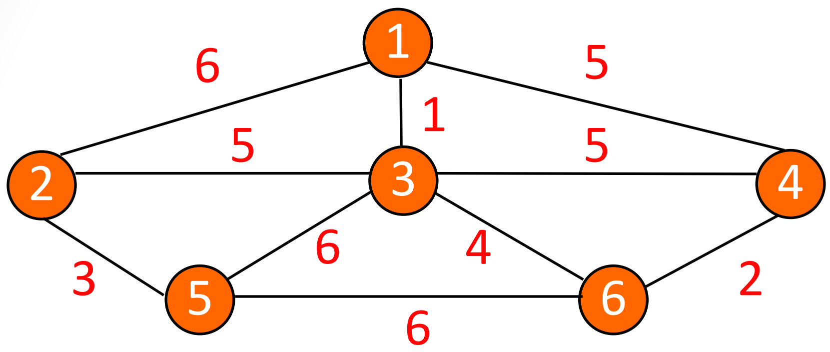


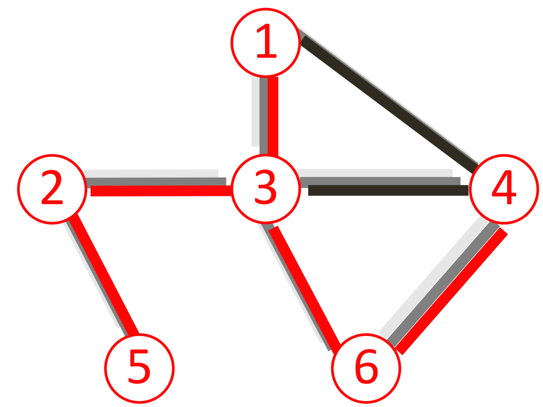
Resultado de aplicar *Kruskal*



$T_{kruskal}(|V|,|A|) \in O(|A|\cdot\log|A|)$

- *Algoritmo de Kruskal: un detalle para su implementación eficiente*





Paso 0:

V = V'; E' = ∅

aristasFactibles = { (1,3,1), (4,6,2), (2,5,3), (3,6,4), (4,1,5), (4,3,5), (3,2,5), ~~(1,2,6)~~, ~~(3,5,6)~~, ~~(6,5,6)~~ }

Paso i:

(*v*, *w*) = min(aristasFactibles);

if *(v, w)* **NO origina un ciclo entonces** E' = E' ∪ *(v, w)*

¿Cómo comprobarlo EFICIENTEMENTE?

Terminación: |E'| = |V| - 1

5. Estructuras de partición

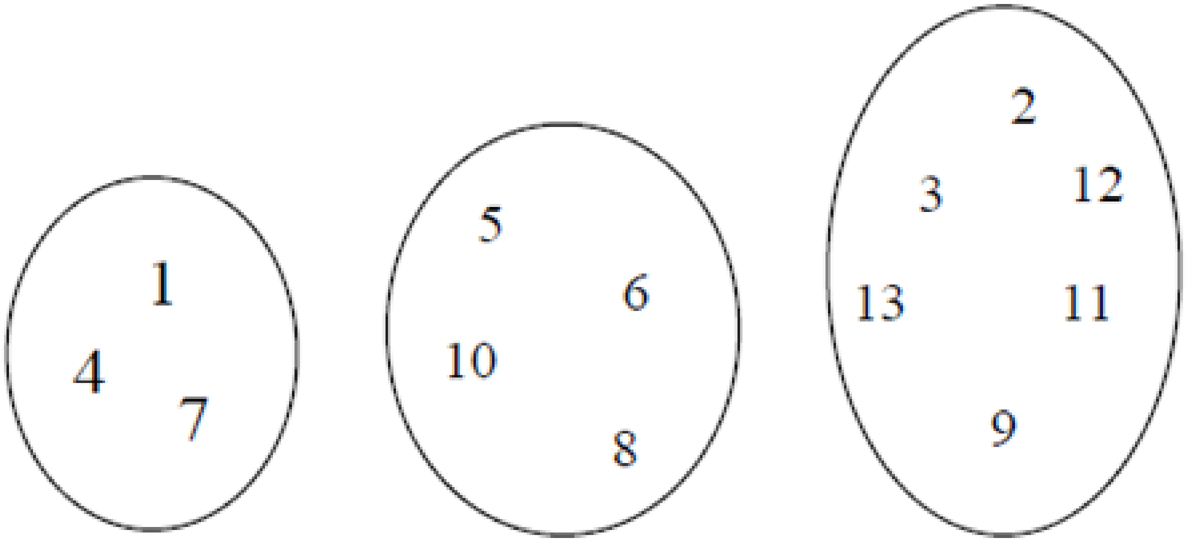
Relaciones de equivalencia

- Para una implementación eficiente de Kruskal, concretamente para comprobar si la inclusión de una arista provoca ciclos, necesitamos usar estructuras de partición
- Una relación *R* definida en un conjunto *C* es un subconjunto del producto cartesiano $C \times C$ de manera que *aRb* denota que (a, b) ∈ R
- *R* es una <u>relación de equivalencia</u> si cumple las siguientes propiedades:
  - ***Reflexiva***: *aRa* para todo $a \in C$.
  - ***Simétrica***: *aRb* si y solo si *bRa*, para todo $a, b \in$ C.
  - ***Transitiva***: *aRb* y *bRc* implica *aRc*, para todo $a, b, c \in C$.
- Un conjunto de elementos se puede particionar en clases de equivalencia a partir de la definición de una relación de equivalencia

5. *UF-Sets*

- Los UF-Sets (*Union-Find Set*) son unas estructuras eficientes para determinar las posibles particiones de un conjunto
  - Los elementos están organizados en subconjuntos disjuntos
  - El numero de elementos es fijo (no se añaden ni se borran)
- Sus operaciones características son:
  - **Unión** (*union*) de dos conjuntos disjuntos
  - **Búsqueda** (*find*): dado un elemento debe determinar a que conjunto pertenece

5. Aplicaciones de UF-Sets



Cada subconjunto se puede identificar por uno de sus miembros

- Aplicaciones:
  - Obtención del árbol de recubrimiento de mínimo peso en un grafo no dirigido (Kruskal)
  - Componentes conexas de un grafo no dirigido
  - Equivalencia entre autómatas finitos

5. Representación de *UF-Sets*

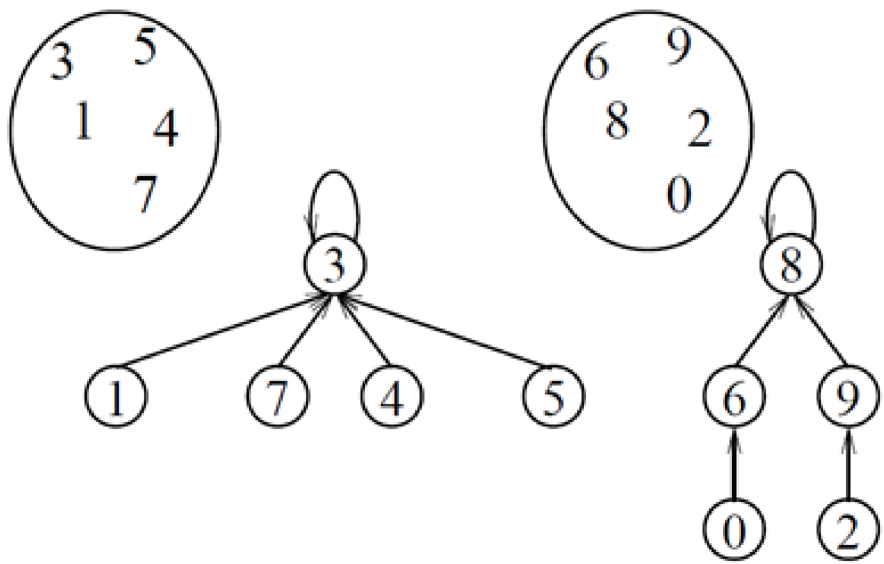
Operaciones sobre UF-Sets

```
public interface UFSet
{
  /** Devuelve el identificador del conjunto
   *  al que pertence el elemento x
   */
  int find(int x);

  /** Une dos conjuntos identificados por
   *  x e y
   */
  void union(int x, int y);
}
```

5. Representación de *UF-Sets*

Representación en bosque

- Cada subconjunto se guarda como un árbol:
  - Los nodos del árbol son los elementos del subconjunto
  - En cada nodo guardamos una referencia al padre
  - El elemento raíz del árbol se usa para representar el subconjunto



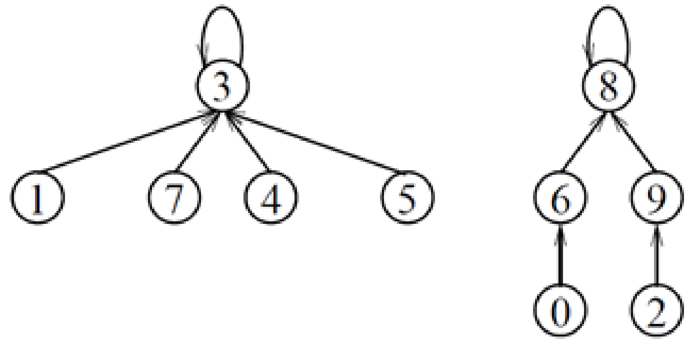
0	1	2	3	4	5	6	7	8	9
6	3	9	3	3	3	8	3	8	8

elArray

- *elArray*[*i*] es el padre del elemento *i*
- Si *elArray*[*i*]=*i*, *i* es la raíz de un árbol

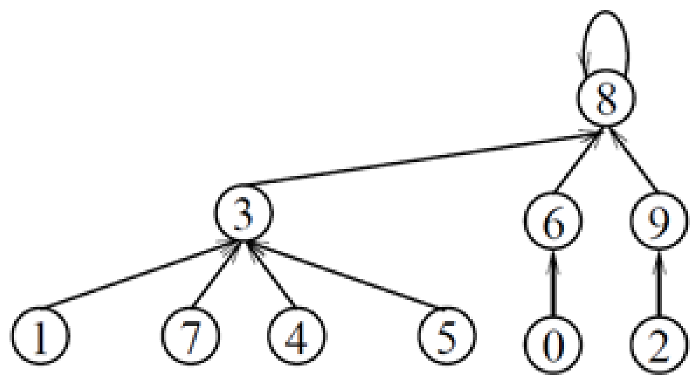
5. Representación de *UF-Sets*

Operaciones sobre UF-Sets



0	1	2	3	4	5	6	7	8	9
6	3	9	3	3	3	8	3	8	8

find(0) = 8
find(9) = 8
find(4) = 3
find(3) = 3



0	1	2	3	4	5	6	7	8	9
6	3	9	**8**	3	3	8	3	8	8

union(3, 8)

5. Mejoras en la eficiencia
Introducción

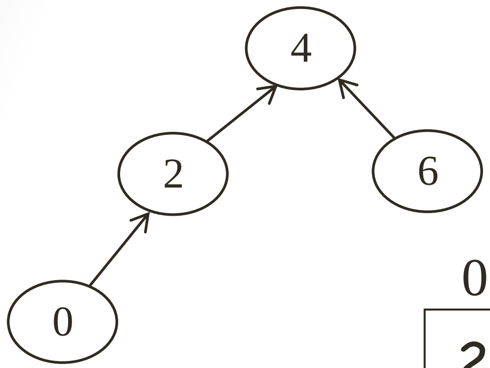
- El método *find* puede tener un coste lineal en función del número de nodos si los árboles están desequilibrados (árboles como listas)
- Las mejoras en el coste se basan en reducir la altura de los árboles:
  - Combinar por rango
  - Compresión de caminos
- Con estas mejoras el coste amortizado de las operaciones es prácticamente constante
  - El coste es una inversa de la función de Ackermann, que crece muy lentamente (ejemplo: $\alpha(2^{65536}) = 5$)

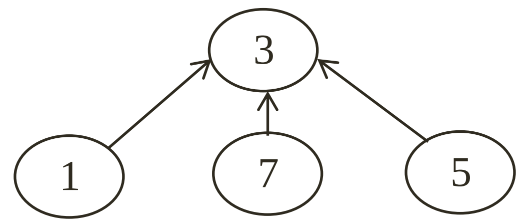
5. Mejoras en la eficiencia
Combinación por rango

- En la operación *union* hacemos que la raíz del árbol de menor altura apunte a la raíz del de mayor altura
  - Si las alturas de los dos árboles a unir son distintas, la altura del árbol resultante será la del árbol de mayor altura
  - Si ambos árboles tienen la misma altura, la altura del árbol resultante será una unidad mayor que la de los árboles a unir
- Para ello es necesario guardar la altura de cada árbol
  - Este valor se puede mantener en el propio vector, en el nodo asociado a la raíz de cada árbol, pero con signo negativo
  - Entonces, si elArray[i]<0, i es la raíz de un árbol y, además |elArray[i]| – 1 será la altura de dicho árbol

5. Mejoras en la eficiencia

Combinación por rango – Ejemplo:

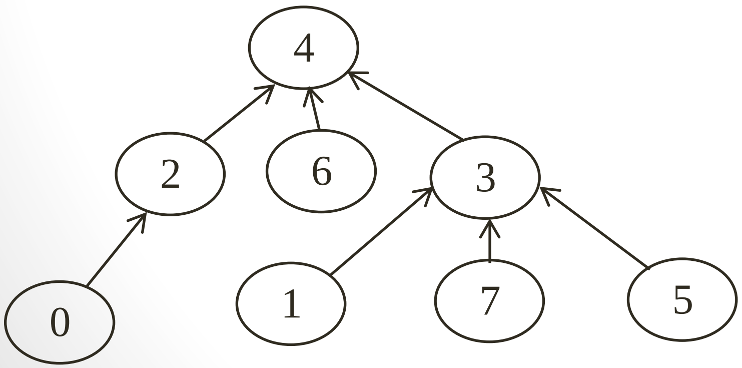




0	1	2	3	4	5	6	7
2	3	4	-2	-3	3	4	3

El 3 es la raíz de un árbol de altura 1

El 4 es la raíz de un árbol de altura 2



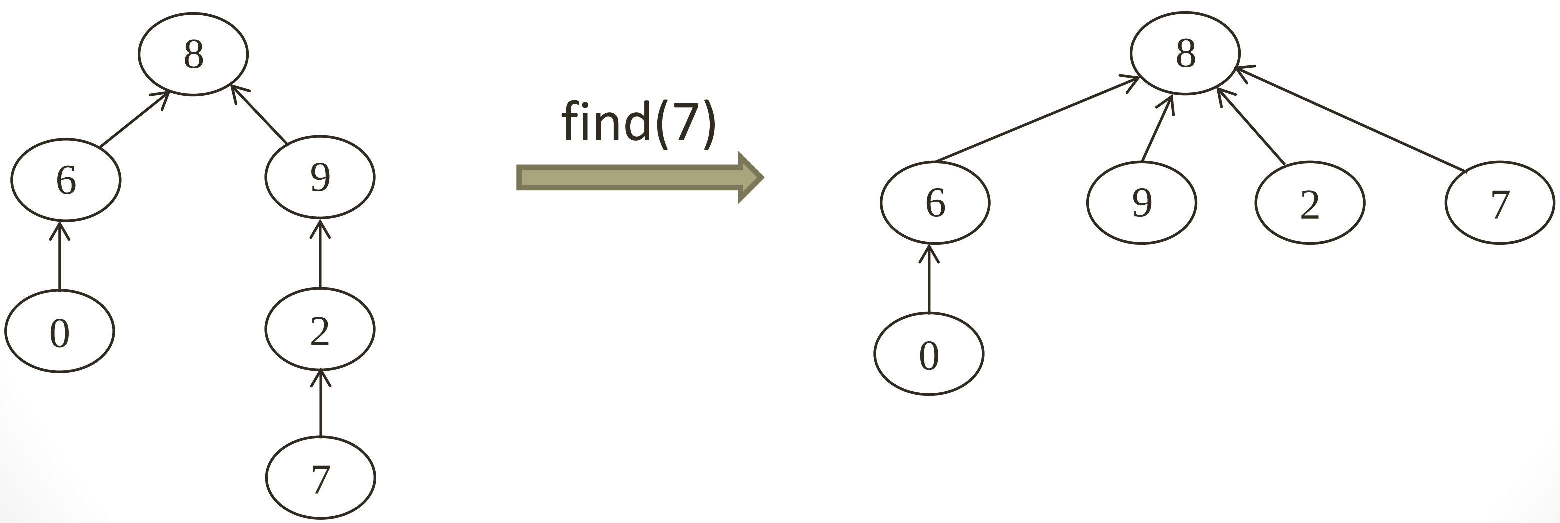
- Al unir ambos árboles, colgamos el de menor altura (el 3) del más alto (el 4)

0	1	2	3	4	5	6	7
2	3	4	4	-3	3	4	3

5. Mejoras en la eficiencia
Compresión de caminos

- En las operaciones de búsqueda (*find*) hacemos que cada nodo por el que pasemos apunte directamente a la raíz del árbol

Ejemplo:



5. Mejoras en la eficiencia
Combinación de las dos estrategias

- La compresión de caminos no es totalmente compatible con la combinación por rango, ya que la compresión de caminos puede reducir la altura del árbol
- En tal caso, la altura almacenada en el vector ya no coincide necesariamente con la altura real del árbol, pero es una cota pesimista de la altura real, con lo que se sigue manteniendo una cota de O(log n), que en la practica será mucho menor por la compresión de caminos

5. Implementación

Atributos y constructor

```
public class ForestUFSet implements UFSet {
  // Array de int que representa un bosque de
  // árboles, de forma que si elArray[i] < 0:
  // (a) i es el identificador de una clase
  // (b) i es la raíz del árbol que representa a la clase
  // (c) |elArray[i]| - 1 es su altura
  protected int elArray[];

  // Crea un UFSet de talla n. Al principio se crean n
  // árboles distintos de un solo elemento (altura 0)
  public ForestUFSet(int n) {
    elArray = new int[n];
    for (int i = 0; i < n; i++)
      elArray[i] = -1;                  // Altura 0
  }
```

5. Implementación

Búsqueda de elementos

```
/** Devuelve el identificador del conjunto al que pertence
  * el elemento x, ademas de enlazar todo los elementos
  * del camino visitado directamente con la raíz */
  public int find(int x) {
    if (elArray[x] < 0) return x; // Raíz del conjunto
    // Compresión del camino
    elArray[x] = find(elArray[x]);
    return elArray[x];
 }
```

5. Implementación

Unión de conjuntos

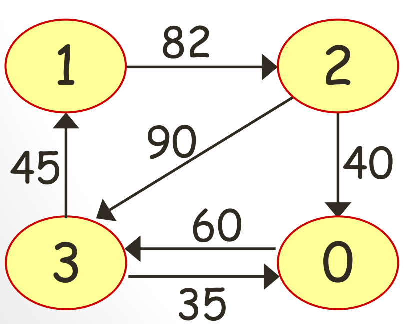
```
/** Une los conjuntos identificados por x e y.
    Los elementos x e y han de ser identificadores de sus
respectivas clases.
*/
public void union(int x, int y) {
  if (elArray[x] == elArray[y]) { // Alturas iguales
    elArray[x] = y;               // Colgamos x de y
    elArray[y]--;    // Incrementamos la altura de y
  } else if (elArray[x] < elArray[y]) {
    elArray[y] = x;               // Colgamos y de x
  } else {
    elArray[x] = y;               // Colgamos x de y
  }
}
```

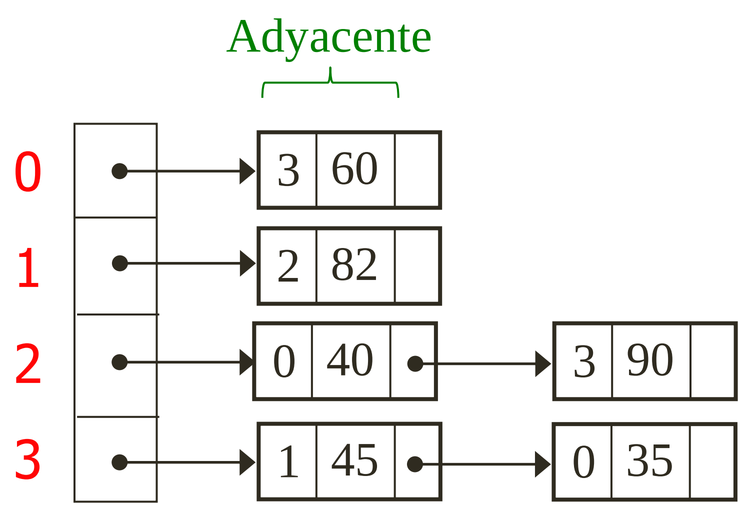
6. Implementación de un Grafo Dirigido mediante listas de adyacencia

La clase Adyacente

```
class Adyacente {
   int destino;          // Vértice destino de la arista
   double peso;          // Peso de la arista
   public Adyacente(int d, double p){ destino = d; peso = p; }
   public String toString(){ return destino+“(”+peso+“)”; }
}
```

Ejemplo:





6. Implementación

La clase GrafoDirigido (1/3)

```
// Implementación de un Grafo Dirigido
public class GrafoDirigido extends Grafo {

  // Número de vértices y aristas
  protected int numV, numA;
   // El array de listas con los adyacentes de cada vértice
  protected ListaConPI<Adyacente> elArray[];

  // Construye un Grafo con un número de vértices dado
  @SuppressWarnings("unchecked")
  public GrafoDirigido(int numVertices) {
    numV = numVertices;
    numA = 0;
    elArray = new ListaConPI[numVertices];
    for (int i = 0; i < numV; i++)
      elArray[i] = new LEGListaConPI<Adyacente>();
  }
```

6. Implementación

La clase GrafoDirigido (2/3)

```
// Consultores
public int numVertices() { return numV; }
public int numAristas()  { return numA; }

public ListaConPI<Adyacente> adyacentesDe(int i) {
  return elArray[i];
}

public boolean existeArista(int i, int j) {
  ListaConPI<Adyacente> l = elArray[i];
  boolean esta = false;
  for (l.inicio(); !l.esFin()&& !esta; l.siguiente())
    if (l.recuperar().destino == j) esta = true;
  return esta;
}
```

6. Implementación

La clase GrafoDirigido (3/3)

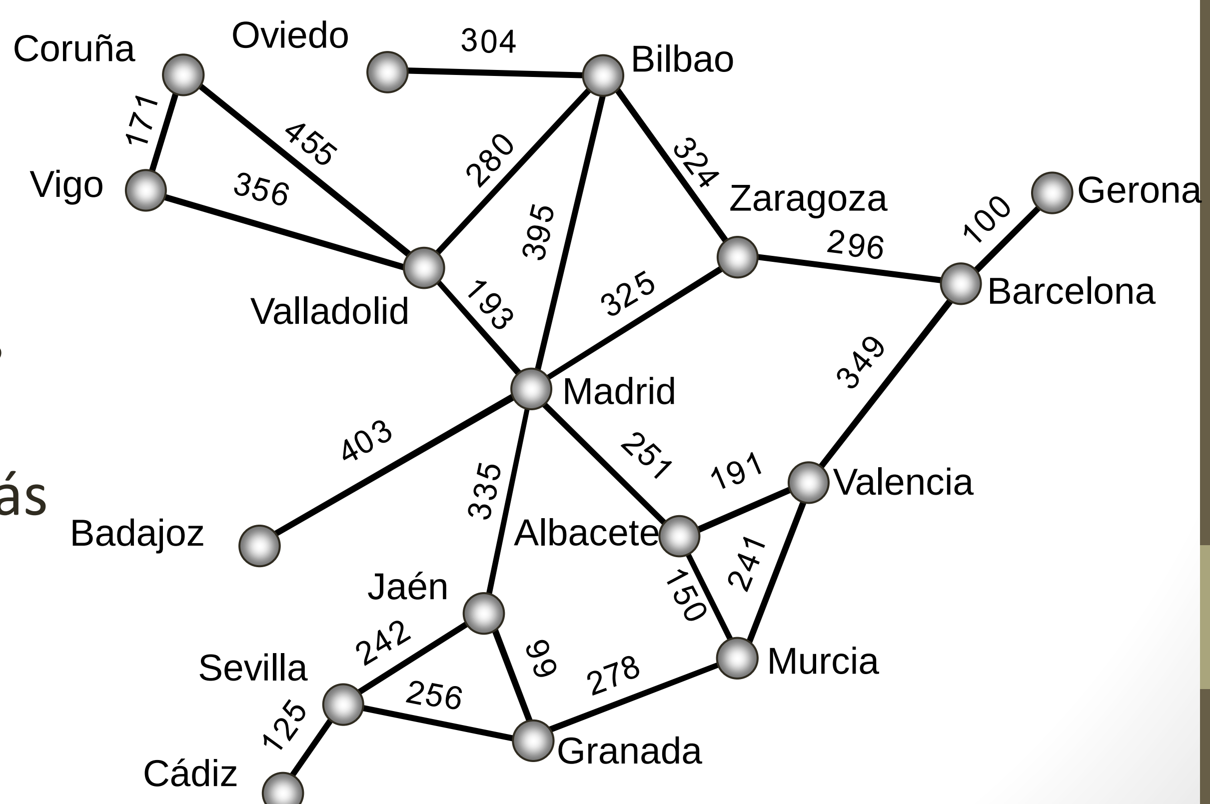
```
public double pesoArista(int i, int j) {
  ListaConPI<Adyacente> l = elArray[i];
  for (l.inicio(); !l.esFin(); l.siguiente())
    if (l.recuperar().destino == j)
      return l.recuperar().peso;
  return 0.0;
}

// Inserción de aristas
public void insertarArista(int i, int j) {
  insertarArista(i, j, 1.0);  // El peso por defecto es 1.0
}
public void insertarArista(int i, int j, double p) {
  if (!existeArista(i,j)) {
    elArray[i].insertar(new Adyacente(j,p));
    numA++;
  }
}
```

7. Caminos de mínimo peso

Definición del problema

- ***Peso de un camino***: suma de los pesos de las aristas por las que pasa: $p(v_0, v_1, \dots, v_k) = \sum_{i=1}^{k} p(v_{i-1}, v_i)$

- ***Problema***: calcular el camino de mínimo peso entre dos nodos o entre un nodo y todos los demás



7. Caminos de mínimo peso
Dijkstra

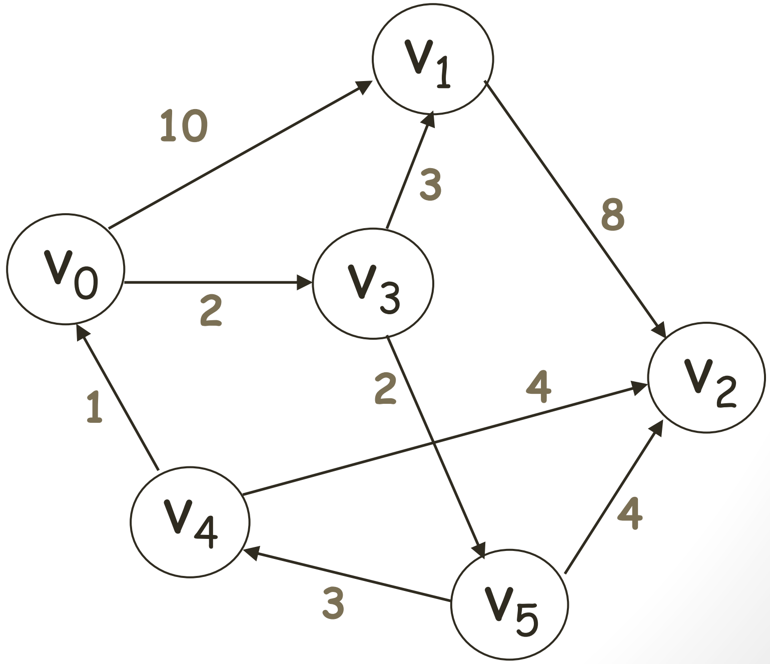
- ***Dijkstra***: calcula los caminos mínimos de un vértice dado al resto de vértices. Requiere que los pesos de las aristas sean positivos.
- Almacena la información en dos arrays:
  - ***distanciaMin***: guarda distancia mínima del vértice origen al resto de vértices
  - ***caminoMin***: para cada vértice guarda el vértice anterior en el camino más corto desde el vértice origen

7. Caminos de mínimo peso

Ejemplo de Dijkstra: caminos mínimos desde v_0

- Queremos calcular los caminos mínimos desde el vértice v_0, por ejemplo, al resto de vértices
- <u>Primer paso</u>: ¿Cuál es la información para el vértice v_0?

```
// Llegar a v0 desde v0 no cuesta nada
distanciaMin[0] = 0
// No hay vértice anterior, el camino
// comienza en v0
caminoMin[0]   = -1
```



7. Caminos de mínimo peso

Ejemplo de Dijkstra: caminos mínimos desde v_0

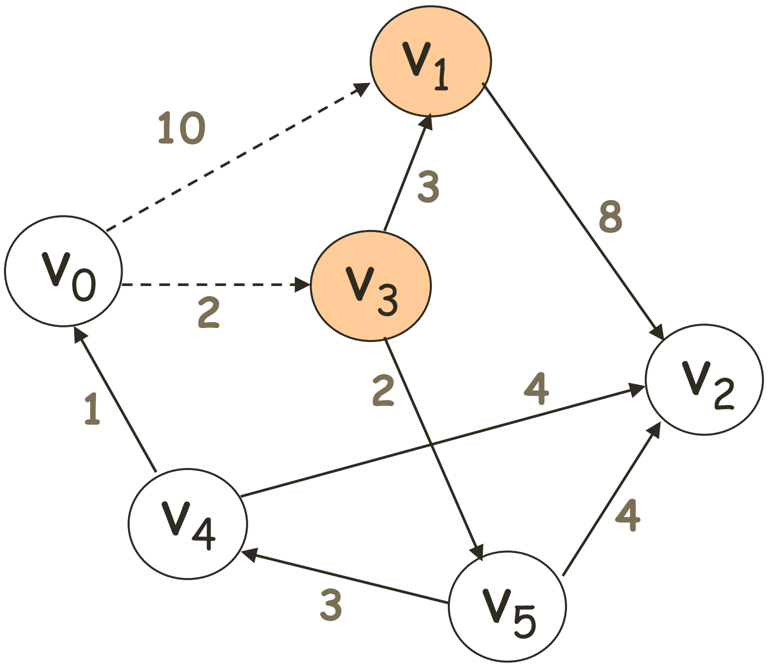
- <u>Segundo paso</u>: calculamos la distancia a los vértices adyacentes a v_0

distanciaMin[1] = 10

caminoMin[1] = 0

distanciaMin[3] = 2

caminoMin[3] = 0



¿Por qué vértice continuamos: v_1 ó v_3?

$\Rightarrow$ Por v_3, que es el que tiene la menor distancia

7. Caminos de mínimo peso

Ejemplo de Dijkstra: caminos mínimos desde v_0

- Tercer paso: calculamos la distancia a los vértices adyacentes a v_3

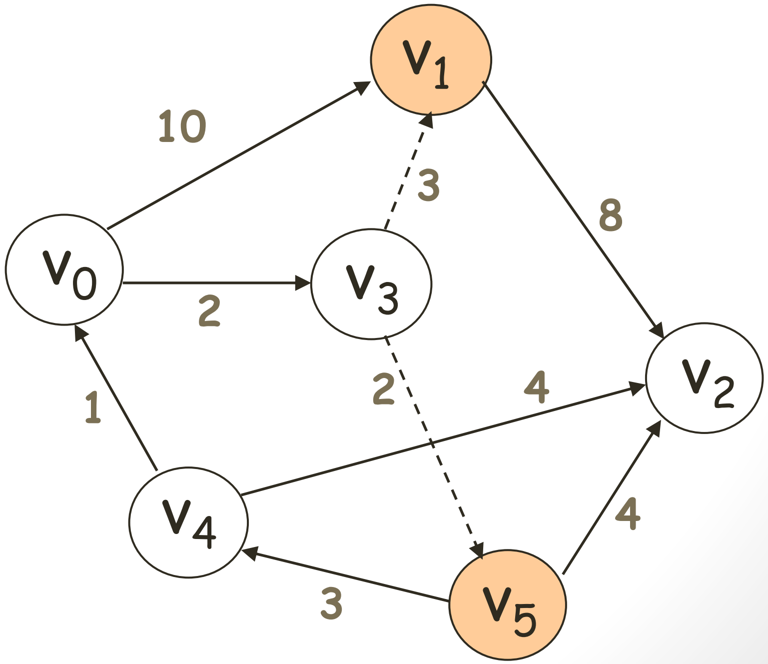
El vértice v_1 ya lo habíamos alcanzado antes:

distanciaMin[1] = *min*(distanciaMin[1], distanciaMin[3] + 3) = min(10, 2 + 3) = 5

caminoMin[1] = 3

distanciaMin[5] = distanciaMin[3] + 2 = 4

caminoMin[5] = 3



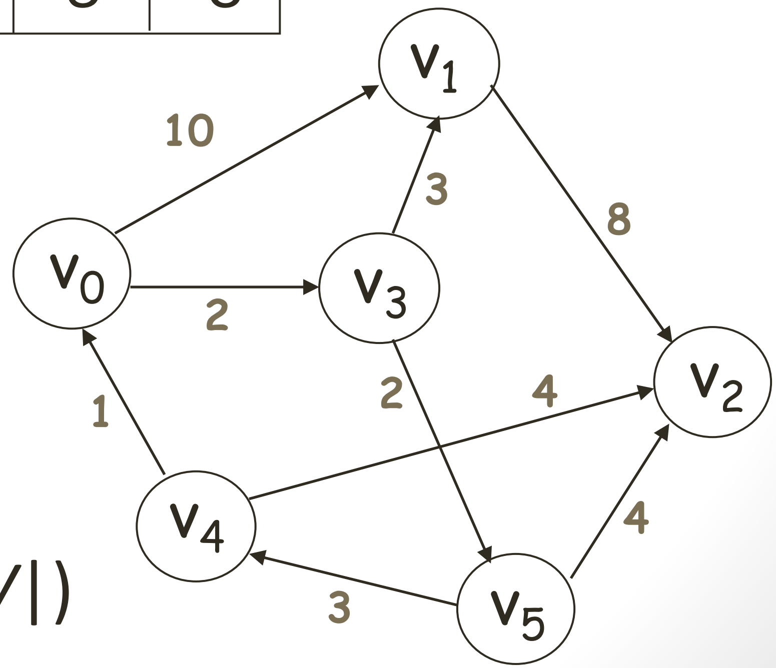
- Etc…

7. Caminos de mínimo peso

Ejemplo de Dijkstra: caminos mínimos desde v_0

- Resultado final del proceso:

	v_0	v_1	v_2	v_3	v_4	v_5
distanciaMin	0	5	8	2	7	4
caminoMin	-1	*3*	*5*	*0*	*5*	*3*



$T_{dijkstra}(|V|, |A|) \in O(|A|\cdot\log|V|)$

7. Caminos de mínimo peso
Decodificación del camino mínimo

- ¿Cómo calculamos ahora el camino mínimo entre v_0 y otro vértice, por ejemplo v_4?

	v_0	v_1	v_2	v_3	v_4	v_5
caminoMin	-1	3	5	0	5	3

1.- La forma más corta de llegar a $\boldsymbol{v_4}$ es por $\boldsymbol{v_5}$

2.- La forma más corta de llegar a $\boldsymbol{v_5}$ es por $\boldsymbol{v_3}$

3.- La forma más corta de llegar a $\boldsymbol{v_3}$ es por $\boldsymbol{v_0}$

4.- $\boldsymbol{v_0}$ es el origen, pues *caminoMin*[*0*] = -1

$\langle \boldsymbol{v_4}, \boldsymbol{v_5}, \boldsymbol{v_3}, \boldsymbol{v_0} \rangle$ Ojo: hay que invertir el camino

$\langle \boldsymbol{v_0}, \boldsymbol{v_3}, \boldsymbol{v_5}, \boldsymbol{v_4} \rangle$

7. Caminos de mínimo peso

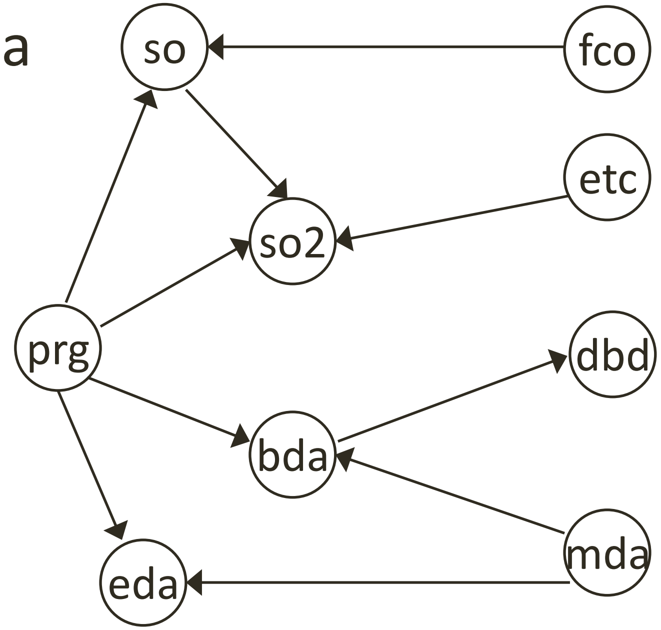
Algoritmo de Dijkstra (pseudocódigo)

```
void dijkstra(int vOrigen) {
  caminoMin[v] = -1,   ∀v∈V                    // Inicializaciones
  distanciaMin[v] = ∝, ∀v∈V
  distanciaMin[vOrigen] = 0
  qPrior ← (vOrigen,0)
  while qPrior ≠ ∅ {                // Mientras haya vértices por explorar
    v ← qPrior        // El siguiente vértice a explorar es el de menor distancia
    if !visitado[v] {                          // Evitamos repeticiones
      visitado[v] = true
      for each a ∈ adyacentesDe(v) {        // Recorremos los vértices
        w = a.destino                       // adyacentes de v
        pesoW = a.peso
        // Vemos si la mejor forma de alcanzar w  es a través de v
        if distanciaMin[w] > distanciaMin[v] + pesoW {
          distanciaMin[w] = distanciaMin[v] + pesoW;
          caminoMin[w] = v;
          qPrior ← (w, distanciaMin[w])
}}}}}
```

8. Órdenes topológicos

Introducción

- <u>Ejemplo</u>: el siguiente grafo representa los prerrequisitos entre asignaturas. Una arista (u, w) indica que la asignatura u debe ser aprobada para poder matricularse en w

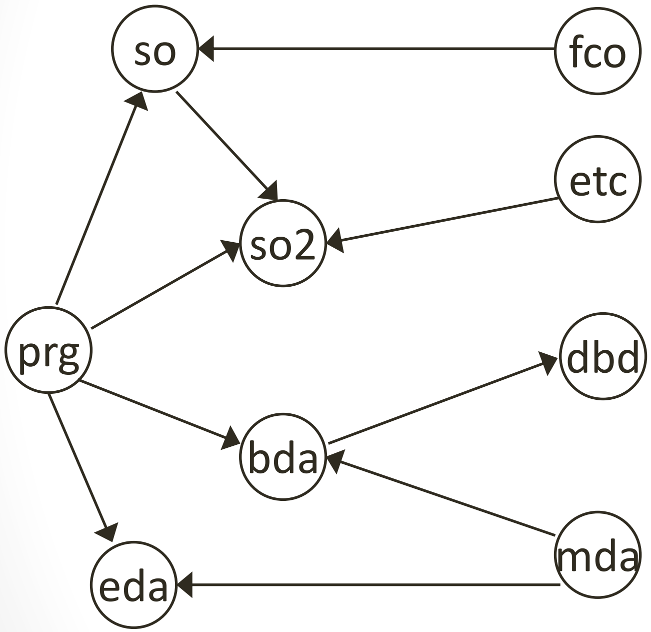


- ⟨prg, so, so2⟩, ⟨prg, bda, dbd⟩, ⟨mda, bda, dbd⟩, ⟨mda, eda⟩, etc. son ***órdenes topológicos***

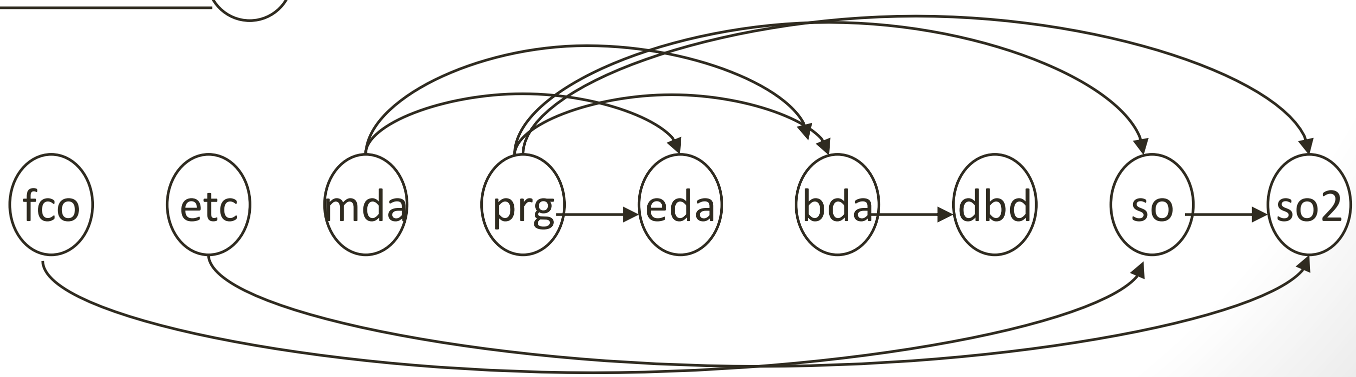
- Una ***ordenación topológica*** es una ordenación lineal de los vértices de un grafo <u>acíclico</u> dado, conservando la ordenación parcial original

8. Órdenes topológicos
Introducción

<u>Ejemplo</u>: encontrar un orden para poder estudiar TODAS las asignaturas:



Recorrido en profundidad
+
uso de una pila de vértices



8. Órdenes topológicos

Método lanzadera

```
// Devuelve un array con los códigos de los vértices en orden
// topológico
public int[] toArrayTopologico() {
  visitados = new int[numVertices()];
  Pila<Integer> pVRecorridos = new ArrayPila<Integer>();
  // Recorrido de los vértices
  for (int vOrigen = 0; vOrigen < numVertices(); vOrigen++)
    if (visitado[vOrigen] == 0)
        ordenacionTopologica(vOrigen, pVRecorridos);
  // Copia el resultado de la ordenación a un array
  int res[] = new int[numVertices()];
  for (int i = 0; i < numVertices(); i++)
    res[i] = pVRecorridos.desapilar();
  return res;
}
```

8. Órdenes topológicos

Método recursivo

```
protected void ordenacionTopologica(int origen,
                    Pila<Integer> pVRecorridos) {
  visitados[origen] = 1;
  // Recorremos los vértices adyacentes
  ListaConPI<Adyacente> aux = adyacentesDe(origen);
  for (aux.inicio(); !aux.esFin(); aux.siguiente()) {
    int destino = aux.recuperar().destino;
    if (visitados[destino] == 0)
       ordenacionTopologica(destino, pVRecorridos);
  }
  // Apilamos el vértice
  pVRecorridos.apilar(origen);
}
```

$$T_{\text{ordenacionTopologica}}(|V|, |A|) \in O(|V| + |A|)$$