

Tema 6 – S1

Grafo y *UF-Set*: Jerarquía Java y Aplicaciones

Contenidos

1. Implementación de un Grafo: la jerarquía Java Grafo
 - La clase abstracta Grafo y la clase Adyacente
 - Implementación mediante Listas de Adyacencia: las clases GrafoDirigido y GrafoNoDirigido como Derivadas de Grafo
 - Ejemplos
 - Ejercicios

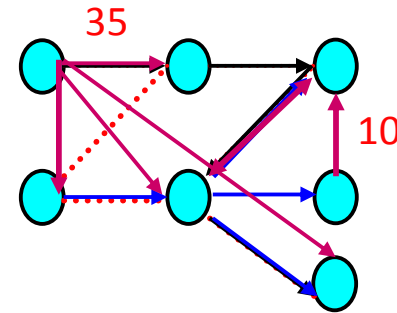
¿Por qué nos interesan los Grafos?

Sea una Colección de ...

- Ciudades
- Aeropuertos
- Ordenadores
- Puntos del plano de una ciudad
- Carreteras, aeropuertos, puertos, ...
- Vuelos
- Enlaces
- Calles

Queremos modelar...

- Una red de comunicaciones entre ciudades
- Rutas aéreas
- El envío de correo electrónico a través de una Red de ordenadores
- Centro de Información turístico
- ...



- Un **Grafo** es, precisamente, una representación de un conjunto de **datos** en el que algunos pares de éstos están conectados mediante **enlaces**
- Un **Grafo** está compuesto por un conjunto de **vértices** V y un conjunto de **aristas** E que conectan algunos de esos vértices entre sí. Esto es, un Grafo G es un Par $G = (V, E)$ en el que cada una de sus aristas es un par (v, w) tal que $v, w \in V$ (**Relación Binaria**)

¿Qué tipos de Grafos nos interesan?

Existen muchos y muy variados tipos de Grafos. Entre todos ellos, nosotros estamos interesados en los siguientes:

- Grafos Simples
- Grafos No-Dirigidos y Dirigidos
- Grafos Etiquetados, en sus aristas y vértices
- Grafos Dispersos

1. Implementación de un Grafo

El problema: hay miríadas de tipos de grafos

- Se podría decir que, los grafos son objetos **polimórficos**. Por ello, las palabras que se necesitan para representar dicho polimorfismo son:
 - **Reutilización** de código
 - **Herencia**, el mecanismo fundamental para la Reutilización del Software
- Claves para el diseño de la Jerarquía Java de un Grafo:
 - La Raíz de la Jerarquía, la clase `Grafo`, debe ser la clase **Factor Común** (tanto a nivel de estructura como de funcionalidad) de todos los grafos que se pueden definir, mientras que cada una de sus clases Derivadas la especializará para representar a cada tipo específico de grafo
 - La Representación con una Matriz de Adyacencias de un Grafo **ES UN** Grafo
 - La Representación con Listas de Adyacencia de un Grafo **ES UN** Grafo
 - Un Grafo No-Dirigido **ES UN** Grafo Dirigido en el que existe la arista (j, i) siempre que exista la arista (i, j)
 - Un Grafo No-Ponderado **ES UN** Grafo Ponderado en el que el que cada arista tiene un peso 1

1. Implementación de un Grafo

Las clases Grafo y Adyacente

La clase `Grafo` representa al Factor Común de todos los grafos

○ Estructura

- Vértices, o valores de tipo `int`
- Aristas, pares de vértices adyacentes (pares de `int` + su peso (valor `double`))

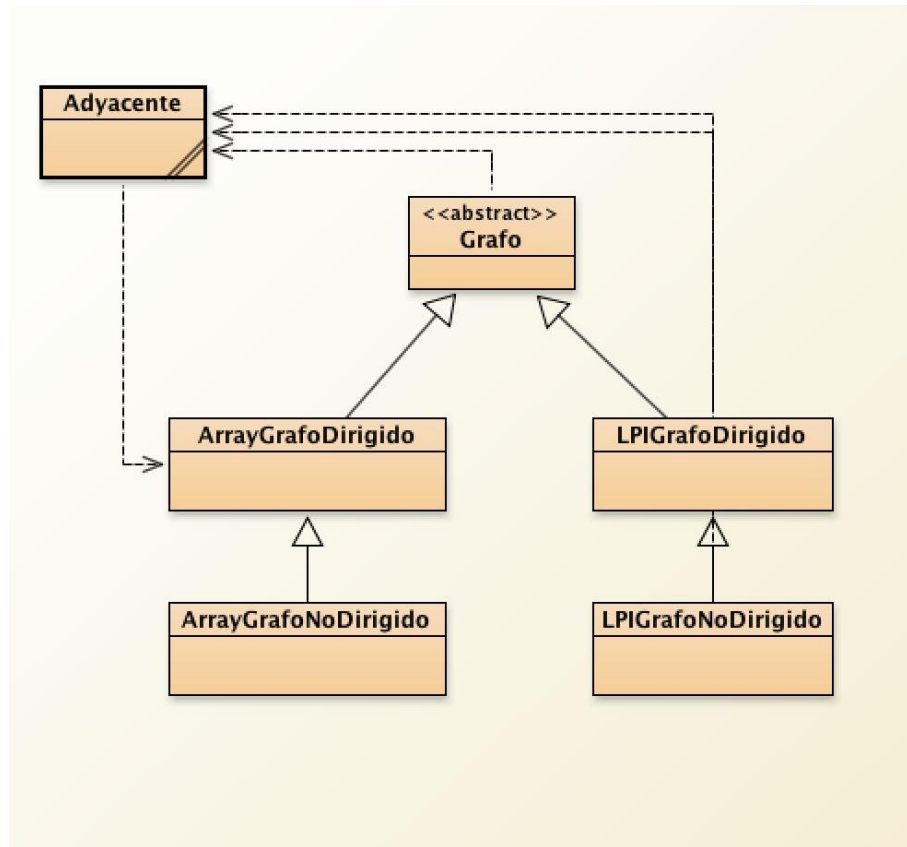
Clase `Adyacente`, una representación *implícita* de una arista; en concreto, representa a un elemento de la Lista de Adyacencias de un vértice: uno de sus vértices adyacentes y el peso de la arista que los une

○ Funcionalidad (operaciones)

- `existeArista(i, j)`, `insertarArista(i, j)`, ...
- `adyacentesDe(i)`: Lista con un `Iterador`, o `ListaConPI`, que permite visitar cada uno de los vértices adyacentes a `i`
- `gradoSalida(i)`, `gradoDeEntrada(i)`, ...
- `toString()`, ...

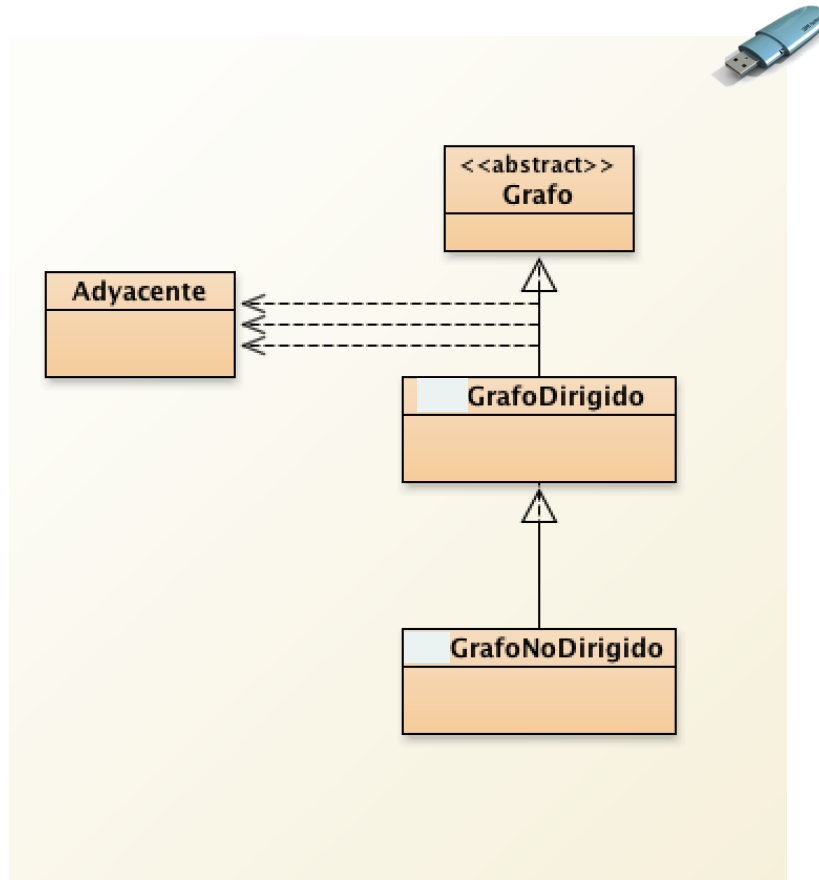
1. Implementación de un Grafo

Una posible Jerarquía Java de un Grafo




1. Implementación de un Grafo

Nuestra Jerarquía Java de un Grafo



1. Implementación de un Grafo

La clase Raíz de nuestra Jerarquía Java de un Grafo




```
package librerias.estructurasDeDatos.grafos;
...
public abstract class Grafo {
    public abstract int numVertices();
    public abstract int numAristas();
    public abstract boolean existeArista(int i, int j);
    public abstract double pesoArista(int i, int j);
    public abstract void insertarArista(int i, int j);
    public abstract void insertarArista(int i, int j, double p);
    public abstract ListaConPI<Adyacente> adyacentesDe(int i);
    public String toString(){ ...}
    // Conforme avance el tema se irán añadiendo más métodos
    ...
}
```

```
public class Adyacente {
    protected int destino; protected double peso;
    public Adyacente(int v, double peso) { destino = v; peso = peso; }
    public int getDestino() { return this.destino; }
    public double getPeso() { return this.peso; }
    public String toString() { return destino + "(" + peso + ")"; }
}
```


1. Implementación de un Grafo

Un ejemplo: el método toString de la clase Grafo



```
/** Devuelve un String con los vértices de un Grafo
 * y sus Adyacentes en orden de inserción */
public String toString() {
    String res = "";

    for (int i = 0 ; i < numVertices(); i++) {
        res += "Vertice: " + i;
        ListaConPI<Adyacente> l = adyacentesDe(i);
        res += (l.esVacia()) ? " sin Adyacentes " : " con Adyacentes: ";

        for (l.inicio(); !l.esFin(); l.siguiente()) {
            res += l.recuperar() + " ";
        }
        res += "\n";
    }
    return res;
}
```

$$T_{\text{toString}}(|V|, |E|) \in O(|V| + |E|)$$

1. Implementación de un Grafo

Las subclases GrafoDirigido y GrafoNoDirigido

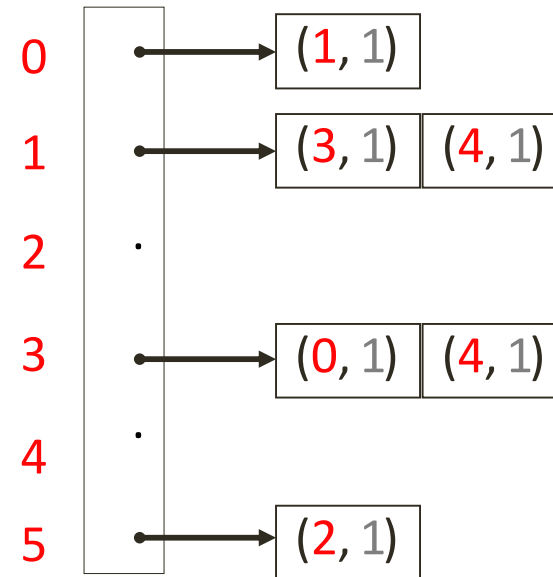
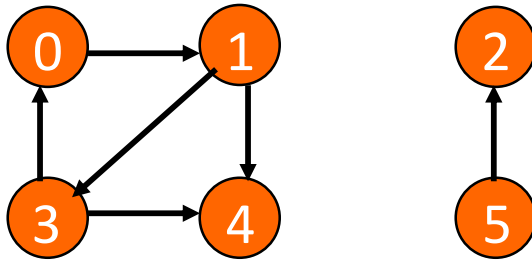
En la clase GrafoDirigido del paquete grafos...



- (a) Analiza la Representación de un Grafo Dirigido (atributos) y su coste
- (b) Analiza las Implementaciones de los métodos de Grafo en un Grafo Dirigido (en base a la Representación vista en (a)) y sus costes: `numVertices()`, `numAristas()`, `existeArista(i, j)`, `pesoArista(i, j)`, `insertarArista(i, j)` y `adyacentesDe(i)`



Ejemplo:



elArray numV = 6 numA = 6

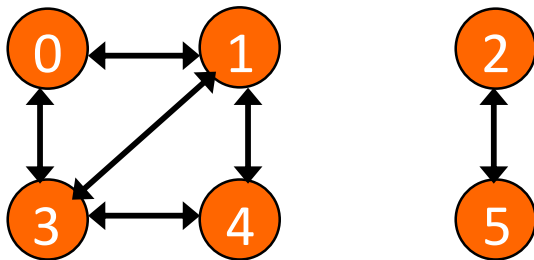
1. Implementación de un Grafo

Las subclases *GrafoDirigido* y *GrafoNoDirigido*

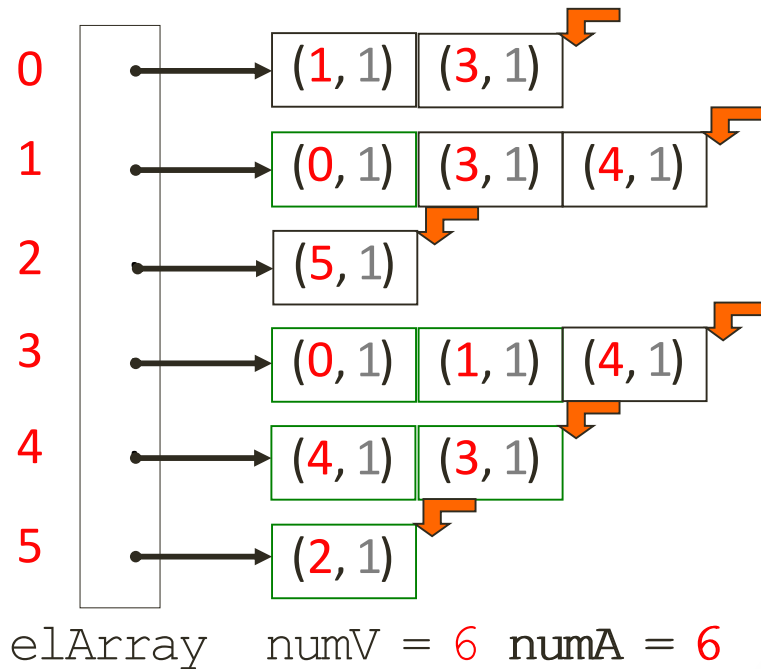
En la clase **GrafoNoDirigido** del paquete `grafos...`

- (a) Analiza la Representación de un Grafo **No** Dirigido (atributos) y su coste
- (b) Analiza las Implementaciones de los métodos de **Grafo** en un Grafo **No** Dirigido (en base a la Representación vista en (a)) y sus costes:
`insertarArista(i, j)`

Ejemplo:



¿En qué orden han sido insertadas las aristas del Grafo?



Complejidad Espacial: $\Theta(|V| + |E|)$

1. Implementación de un Grafo

Ejemplos



En la clase GrafoDirigido,...

Ejemplo 1: completa el código del método `gradoSalida` que, **siempre en tiempo constante** ($\Theta(1)$), devuelve el grado de Salida del vértice `i` de un Grafo Dirigido

```
public int gradoSalida(int i) {  
  
}
```

Ejemplo 2: completa el código del método `gradoSalida` que, **siempre en tiempo lineal con el nº de vértices** ($\Theta(|V|)$), devuelve el grado de Salida de un **Digrafo**

```
public int gradoSalida() {
```

¿Se debería sobrescribir alguno de ellos en GrafoNoDirigido?

Ejemplos

En la clase GrafoDirigido, ...

Ejemplo 3: completa el código del método `gradoEntrada` que, **en tiempo lineal con la talla del problema** ($O(|V|+|E|)$), devuelve el grado de Entrada del vértice `i` de un Grafo Dirigido

NOTA: usa el método `existeArista` para hacer más legible tu código

```
public int gradoEntrada(int i) {
```

}

¿Se debería sobrescribir este método en GrafoNoDirigido?

1. Implementación de un Grafo

Ejemplos

En la clase GrafoDirigido, ...

Ejemplo 4: la siguiente es una versión ineficiente del método que devuelve el grado de Entrada de un Grafo Dirigido. **Usa la pista que te damos** para mejorar su eficiencia

```
public int gradoEntrada() {  
    int gradoMax = gradoEntrada(0); //  $O(|V| + |E|)$   
    for (int i = 1; i < numV; i++) {  
        int grado = gradoEntrada(i); //  $O(|V| + |E|)$   
        if (grado > gradoMax) gradoMax = grado;  
    }  
    return gradoMax;  
}
```

PISTA: el coste de este método es $O(|V| * |E|)$ porque cada iteración del bucle for “cuesta” $O(|V| + |E|)$, el coste de ejecutar `gradoEntrada(i)` una vez por vértice. Para evitar esta llamada, y reducir el coste del método a $O(|V| + |E|)$, **“llevar la cuenta” del valor del grado de Entrada de cada vértice en un array auxiliar de contadores** (`gradoEntrada[i]`, con $0 \leq i < \text{numV}$, “cuenta” el grado de Entrada del vértice i)

(LA PISTA SIGUE EN LA SIGUIENTE PÁGINA)

1. Implementación de un Grafo

Ejemplos

Ejemplo 4 [PISTA: “llevar la cuenta” de los grados de Entrada en un array de contadores]

El esqueleto de la versión eficiente del método sería...

```
public int gradoEntrada() {  
    // crear e inicializar el array de contadores  
    /* COMPLETAR */  
    for (int i = 0; i < numV; i++) {  
        // actualizar el contador del grado de Entrada  
        // de cada vértice de la Lista adyacentesDe(i)  
        /* COMPLETAR */  
    }  
    //  $\forall i: 0 \leq i < \text{numV}$ : gradoEntrada[i] es el grado de Entrada del vértice i  
    int gradoMax = gradoEntrada[0];  
    for (int i = 1; i < numV; i++) {  
        int grado = gradoEntrada[i];  
        if (grado > gradoMax) { gradoMax = grado; }  
    }  
    return gradoMax;  
}
```

¿Se debería sobrescribir este método en GrafoNoDirigido?

1. Implementación de un Grafo

Ejercicios

En la clase `GrafoDirigido`, y con el menor coste Temporal posible, ...

Ejercicio 1: Grado de un Grafo Dirigido (GD) y su coste



(clave: CCDLG00Z)

Completa el código del método `grado` para que devuelva el grado de un GD

Ejercicio 2: Arista de mayor peso en un Grafo Dirigido



(clave: CCDLN00Z)

Completa el código del método `aristaMayorPeso` para que devuelva el peso máximo de las aristas de un Grafo Dirigido; supón que el n^o de aristas del Grafo es mayor que cero y que los pesos de las aristas son no negativos

1. Implementación de un Grafo

Ejercicios

En la correspondiente clase (o clases) de la Jerarquía Grafo, implementa con el menor coste posible los siguientes métodos:

Ejercicio 3: `getVerticeReceptivo`, que devuelve el primer vértice (el de menor índice) de un Grafo con grado de Entrada $|V|-1$, o -1 si no existe

Ejercicio 4: `esSumidero`, que comprueba si un vértice v de un Grafo es un Sumidero

Nota: Un Sumidero (*Sink* en Inglés) es un vértice con grado de Entrada mayor que cero y grado de Salida 0

Ejercicio 5: `getSumideroU`, que devuelve el primer Sumidero Universal de un Grafo, o -1 si no existe

Nota: Un Sumidero Universal (*Super Sink* en Inglés) es un vértice con grado Entrada $|V|-1$ y grado de Salida 0

1. Implementación de un Grafo

Ejercicios

En la correspondiente clase (o clases) de la Jerarquía Grafo, implementa con el menor coste posible los siguientes métodos:

Ejercicio 6: `getFuenteU`, que devuelve la primera Fuente Universal de un Grafo, o -1 si no existe

Nota: Una Fuente Universal (*Source* en Inglés) es un vértice con grado de Entrada 0 y grado de Salida $|V|-1$

Ejercicio 7: método `esCompleto`, que comprueba si un Grafo es Completo

Nota: Un grafo Completo es un grafo Simple donde cada par de vértices distintos están conectados por una única arista (bidireccional en el caso de Digrafos). Equivalentemente, un grafo Simple es Completo si cada uno de sus Vértices es Adyacente al resto

Pista: lema *Handshake*

Tema 6 – S2

Grafo: Jerarquía Java y Aplicaciones

Contenidos

2. Exploración de un Grafo en Profundidad (Depth First Search, DFS) y en Anchura (*Breadth First Search*, BFS)
 - Conceptos previos: Alcanzabilidad
 - Exploración DFS o en Profundidad de un Grafo: propiedades, implementación, análisis de su coste y ejemplos
 - Cuestiones y ejercicios

2. Estrategias DFS and BFS

Conceptos previos: Alcanzabilidad

- Un gran número de aplicaciones sobre grafos se basan en responder a la siguiente pregunta:

Dado un grafo G y un vértice v de G , **¿qué vértices de G son alcanzables desde v ?**

- ① Un vértice u es alcanzable desde v SI existe un **camino** de v a u
- ② Responder a la pregunta requiere visitar todos los vértices alcanzables desde v

- Las dos estrategias estándar para buscar tales vértices de un Grafo son la **Exploración en Profundidad** (o **DFS**, del Inglés *Depth First Search*) y la **Exploración en Amplitud** (o **BFS**, del Inglés *Breadth First Search*)

- ① La estrategia **DFS** puede ser considerada una generalización de la exploración en **Pre-Orden** de un Árbol
- ② La estrategia **BFS** puede ser considerada una generalización de la exploración **Por Niveles** de un Árbol

2. Estrategias DFS y BFS

DFS: generalizar el Recorrido en Pre-Orden de un Árbol

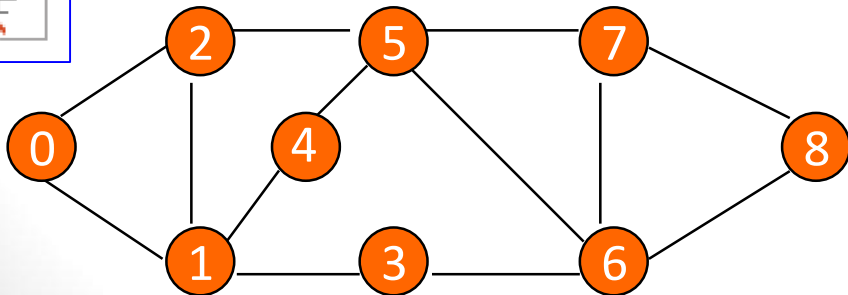
```
preOrden(Árbol t) { preOrden(raízT); }
```

```
preOrden(Nodo n) {  
    visitar(n);  
     $\forall h \in \text{hijosDe}(n)$  preOrden(h);  
}
```

Generalización: considerar que (1) cada vértice v es la Raíz de un Árbol y que (2) cada vértice w adyacente a v es un Hijo de v . Así:

```
dFS(Grafo g) {  $\forall v \in V$  dFS(v); }
```

```
dFS(Vértice v) {  
    visitar(v);  
     $\forall w \in \text{adyacentesDe}(v)$  dFS(w);  
}
```



```
Vertice: 0 con Adyacentes 2 1  
Vertice: 1 con Adyacentes 0 2 4 3  
Vertice: 2 con Adyacentes 1 0 5  
Vertice: 3 con Adyacentes 6 1  
...
```

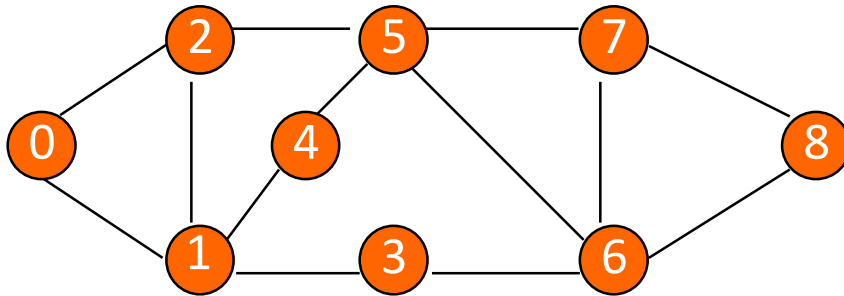
2. Estrategias DFS y BFS

DFS: 3 cuestiones sobre 2 tipos de Grafos



Traza la ejecución del algoritmo **DFS** sobre el Grafo ejemplo, representado mediante Listas de Adyacencia como se indica, y responde:

- ¿Cuántos vértices se alcanzan desde el 0?
- ¿Por qué?
- ¿Cuántas aristas de G se usan en el DFS del vértice 0?



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

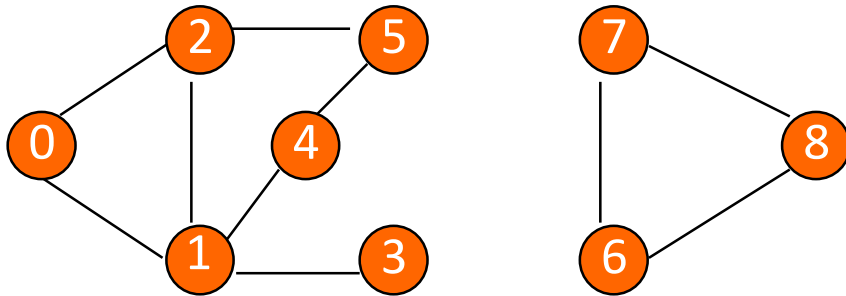
```
Vertice: 0 con Adyacentes 2 1
Vertice: 1 con Adyacentes 0 2 4 3
Vertice: 2 con Adyacentes 1 0 5
Vertice: 3 con Adyacentes 6 1
Vertice: 4 con Adyacentes 1 5
Vertice: 5 con Adyacentes 4 6 7 2
Vertice: 6 con Adyacentes 8 3 7 5
Vertice: 7 con Adyacentes 8 6 5
Vertice: 8 con Adyacentes 6 7
```

2. Estrategias DFS y BFS

DFS: 3 cuestiones sobre 2 tipos de Grafos

Traza la ejecución del algoritmo DFS sobre el Grafo ejemplo, representado mediante Listas de Adyacencia como se indica, y responde:

- ¿Cuántos vértices se alcanzan desde el 0?
- ¿Por qué?
- ¿Cuántas aristas de G se usan en el DFS del vértice 0?



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

Vertice: 0 con Adyacentes 2 1
Vertice: 1 con Adyacentes 0 2 4 3
Vertice: 2 con Adyacentes 1 0 5
Vertice: 3 con Adyacentes 1
Vertice: 4 con Adyacentes 1 5
Vertice: 5 con Adyacentes 4 2
Vertice: 6 con Adyacentes 8 7
Vertice: 7 con Adyacentes 8 6
Vertice: 8 con Adyacentes 6 7

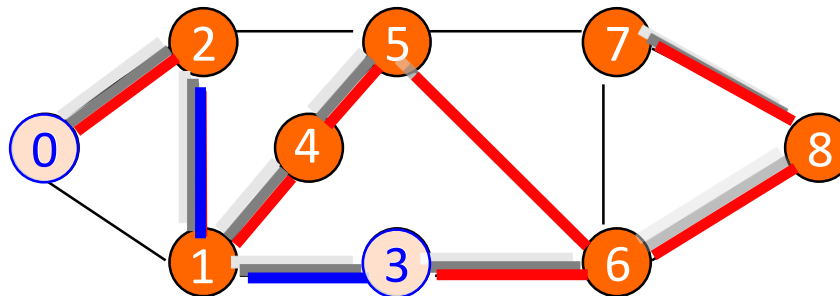
2. Estrategias DFS y BFS

DFS: Propiedades

- El DFS de un vértice v alcanza **primero** (*First*) tanta **profundidad** (*Depth*) como puede – de ahí su nombre en Inglés: *Depth First*

Durante el DFS del vértice v , el siguiente DFS se realiza en el siguiente nivel de adyacencia. Solo cuando ya no quedan vértices adyacentes alcanzables en el siguiente nivel vuelve hacia atrás (recursivamente) para visitar otros vértices, hasta alcanzar de nuevo su nivel de adyacencia (i.e. el de los adyacentes a v)

- En el DFS de v se visitan **todos los vértices** w **alcanzables desde** v
- El DFS del vértice v **no** siempre encuentra el camino sin pesos más corto entre v y otro vértice destino dado w (porque v y w pueden compartir vértices adyacentes)



2. Estrategias DFS y BFS

DFS: método toArrayDFS de la clase Grafo

(diferencias con el algoritmo dFS en verde)

```
protected int[] visitados; // atributo "auxiliar"
protected int ordenVisita; // atributo "auxiliar"
public int[] toArrayDFS() {
    int[] res = new int[numVertices()]; ordenVisita = 0;
    visitados = new int[numVertices()];
    for (int v = 0; v < numVertices(); v++) {
        if (visitados[v] == 0) { toArrayDFS(v, res); }
    }
    return res;
}
protected void toArrayDFS(int v, int[] res) {
    visitados[v] = 1;
    res[ordenVisita] = v; ordenVisita++;
    ListaConPI<Adyacente> l = adyacentesDe(v);
    for (l.inicio(); !l.esFin(); l.siguiente()) {
        int w = l.recuperar().getDestino();
        if (visitados[w] == 0) { toArrayDFS(w, res); }
    }
}
```

2. Estrategias DFS y BFS

DFS: Análisis (c. temporal) del método toArrayDFS()

```
public int[] toArrayDFS() {  
    int[] res = new int[numVertices()]; ordenVisita = 0;  
    visitados = new int[numVertices()];  
    for (int v = 0; v < numVertices(); v++) {  
        if (visitados[v] == 0) { toArrayDFS(v, res); }  
    }  
    return res;  
}  
  
protected void toArrayDFS(int v, int[] res) {  
    res[ordenVisita] = v; ordenVisita++; visitados[v] = 1;  
    ListaConPI<Adyacente> l = adyacentesDe(v);  
    for (l.inicio(); !l.esFin(); l.siguiente()) {  
        int w = l.recuperar().getDestino();  
        if (visitados[w] == 0) { toArrayDFS(w, res); }  
    }  
}
```

El DFS se realiza (¡solo!)
para aquellos vértices no alcanzables desde v
(al menos 1 vez y como máximo $|V|$)

En el DFS de v se visitan (¡exactamente una vez!)
todos los vértices w alcanzables desde v
(conectados por al menos $|V|-1$ aristas y como máximo $|E|$)

$$T_{\text{toArrayDFS}}(|V|, |E|) \in \Theta(|V| + |E|)$$

2. Estrategias DFS y BFS

DFS: Ejercicio

Para que te familiarices y pongas en práctica los conceptos sobre Exploración DFS introducidos hasta el momento hemos preparado el siguiente ejercicio:

Ejercicio 8: Examen poli [formaT]

“Tema 6 – S2: Cuestiones sobre la traza del Recorrido DFS de un Grafo”

Tema 6 – S3 y S4

Grafo: Jerarquía Java y Aplicaciones

Contenidos

2. Exploración de un Grafo en Profundidad (*Depth First Search*, DFS) y en Anchura (*Breadth First Search*, BFS)
3. Aplicaciones BFS y DFS
 - Árbol de Recubrimiento
 - Conceptos previos: Subgrafo, Árbol y Árbol de Recubrimiento
 - Árboles de Recubrimiento DFS y BFS (**Práctica 6 - S1**)
 - Árbol de Recubrimiento Mínimo: algoritmo de Kruskal (**Práctica 6 - S2**)
 - Caminos desde un vértice dado
 - Conceptos previos: camino entre 2 vértices (simple, ciclo, bucle) y su longitud
 - Caminos mínimos (BFS) SIN pesos
 - Caminos mínimos (BFS) CON pesos: algoritmo de Dijkstra

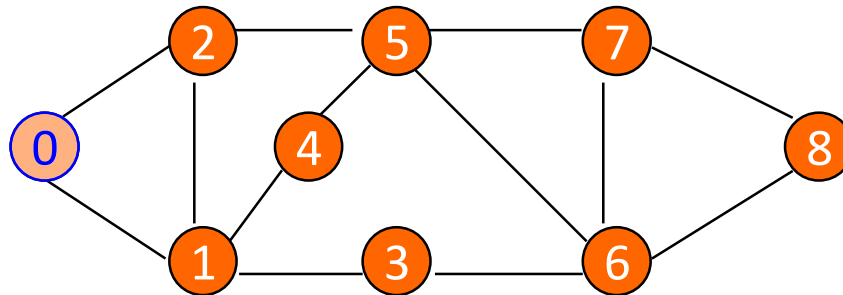
Ejercicios

- Ordenación Topológica y Test de Aciclicidad como Recorridos DFS

2. Estrategias DFS y BFS

BFS: una forma/orden alternativo de explorar un Grafo

- El dFS/bFS de v alcanza **primero** tanta profundidad/amplitud como puede
- El siguiente dFS/bFS se realiza en el siguiente/mismo nivel de adyacencia (dist. a v)



Vertice: 0 con Adyacentes 2 1

Vertice: 1 con Adyacentes 0 2 4 3

Vertice: 2 con Adyacentes 1 0 5

Vertice: 3 con Adyacentes 6 1

Vertice: 4 con Adyacentes 1 5

Vertice: 5 con Adyacentes 4 6 7 2

Vertice: 6 con Adyacentes 8 3 7 5

Vertice: 7 con Adyacentes 8 6 5

Vertice: 8 con Adyacentes 6 7

2. Estrategias DFS y BFS

BFS: generalizar el Recorrido Por Niveles de un Árbol

```
porNiveles (Árbol t) { porNiveles(raízT); }
```

```
porNiveles(Nodo n) {  
    Cola<Nodo> q = new ArrayCola<Nodo>(); q.encolar(n);  
    while (!q.esVacia()) {  
        Nodo n = q.desencolar(); visitar(n);  
         $\forall h \in \text{HijosDe}(n)$  q.encolar(h);  
    }  
}
```

```
bFS(Grafo g) {
```

```
    Cola<Vértice> q = new ArrayCola<Vértice>(); // Vértice es Integer
```

```
     $\forall v \in V$  if (visitados[v] == 0) bFS(v);  
}
```

```
bFS(Vértice v) {  
    q.encolar(v); visitar(v); visitados[v] = 1;  
    while (!q.esVacia()) {  
        Vértice v = q.desencolar();  
         $\forall w \in \text{adyacentesDe}(v)$   
        if (visitados[w] == 0) {  
            visitar(w); visitados[w] = 1; q.encolar(w);  
        }  
    }  
}
```

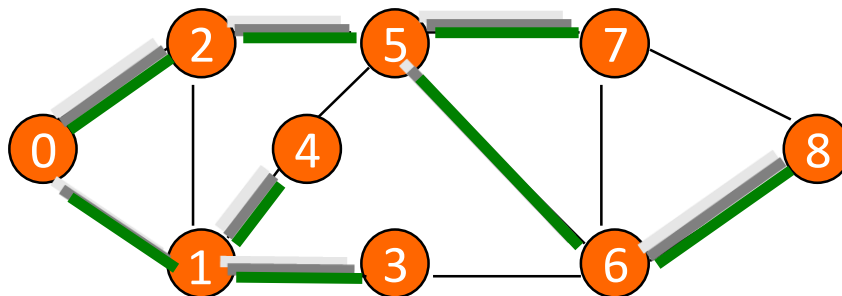
2. Estrategias DFS y BFS

BFS: Propiedades

- El BFS de un vértice v alcanza **primero** (*First*) tanta **anchura** (*breadth*) como puede – de ahí su nombre en Inglés: *Breadth First*

Durante el BFS del vértice v , el siguiente BFS se realiza en el mismo nivel de adyacencia (o a la misma distancia i de v , por tanto iterativamente). Solo cuando ya no quedan vértices adyacentes alcanzables en dicho nivel visita otros vértices a más profundidad, i.e. en el siguiente nivel de adyacencia (o a una distancia $i+1$ de v), hasta el máximo nivel de adyacencia del Grafo donde queden vértices alcanzables aún no visitados.

- En el BFS de v se visitan todos los vértices w alcanzables desde v
- El BFS del vértice v **sí** encuentra siempre el camino sin pesos más corto entre v y otro vértice destino dado w (porque los vértices son visitados en orden de su distancia a v , siendo cada camino de longitud i examinado antes que los caminos de longitud $i+1$)



2. Estrategias DFS y BFS

*BFS: método toArrayBFS de la clase Grafo
(diferencias con toArrayDFS en verde)*

```
protected int[] visitados; protected int ordenVisita; // atributos "auxiliares"
protected Cola<Integer> // atributo "auxiliar"

public int[] toArrayBFS() {
    int[] res = new int[numVertices()]; visitados = new int[numVertices()];
    ordenVisita = 0; q = new ArrayCola<Integer>();
    for (int v = 0; v < numVertices(); v++) { if (visitados[v] == 0) { toArrayBFS(v, res); } }
    return res;
}

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


2. Estrategias DFS y BFS

BFS: Análisis (c. Temporal) del método toArrayBFS()

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

El BFS se ejecuta (solo!)
para aquellos vértices no alcanzables desde v

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

En el BFS de v se visitan
(¡exactamente una vez!)
todos los vértices w
alcanzables desde v

$$T_{\text{toArrayBFS}}(|V|, |E|) \in \Theta(|V| + |E|)$$

2. Estrategias DFS y BFS

BFS: Ejercicio

Para que te familiarices y pongas en práctica los conceptos sobre Exploración BFS introducidos hasta el momento hemos preparado el siguiente ejercicio:

Ejercicio 9: Examen poli [format]

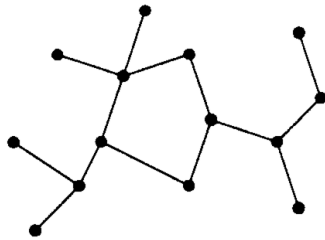
“Tema 6 – S3: Cuestiones sobre la traza del Recorrido BFS de un Grafo”

3. Aplicaciones DFS y BFS

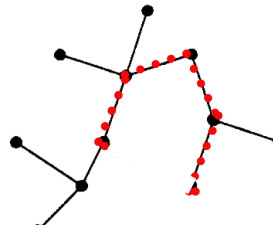
Árbol de Recubrimiento: conceptos previos

Algunas definiciones de interés:

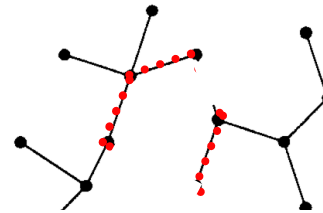
- Un Grafo H es un **Subgrafo** de un Grafo $G = (V, T)$ SII sus vértices y aristas son subconjuntos de los de G
- Un **Árbol** es un Grafo Conexo Acíclico y No-Dirigido
- Un **Bosque** es un Grafo Acíclico y No-Dirigido
i.e. una Componente Conexa de tal Grafo es un Árbol



Grafo



Árbol



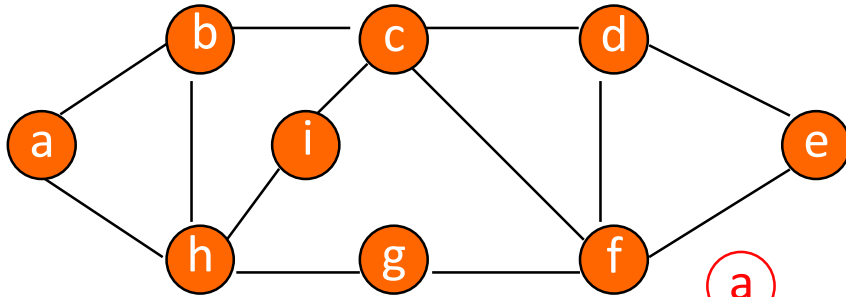
Bosque

- Un Subgrafo T de G que contiene a todos los vértices de G y es un Árbol (Acíclico) es un **Árbol de Recubrimiento** (*Spanning Tree* en inglés)
 - ➔ Cualquier par de vértices de T (V) están conectados por un único camino Simple
 - ➔ Un Grafo No-Dirigido Conexo debe tener al menos $|V| - 1$ aristas... **¿Cuáles?**
 - ➔ Las que se usan en el DFS o BFS de un vértice dado para alcanzar cada uno de los restantes vértices del Grafo

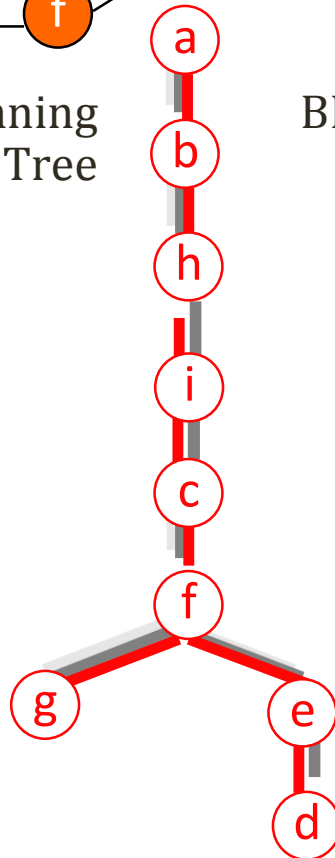
3. Aplicaciones DFS y BFS

Árboles de Recubrimiento DFS y BFS

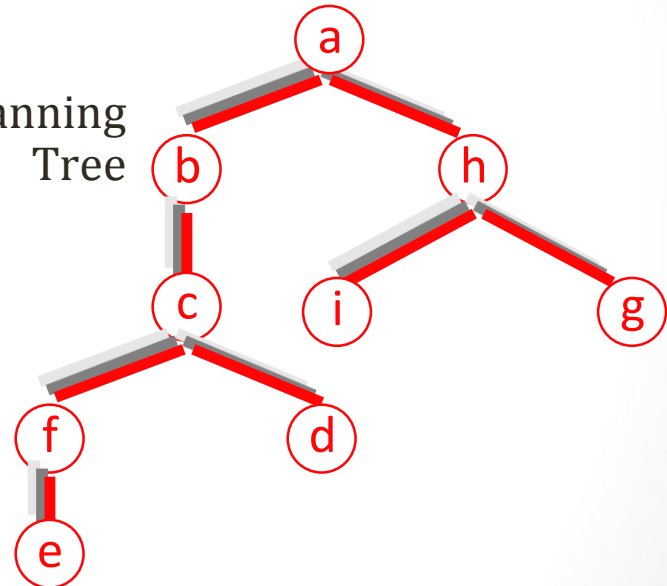
Sea el siguiente Grafo:



DFS Spanning Tree



BFS Spanning Tree



PRÁCTICA 6 - S1

3. Aplicaciones DFS y BFS

Árbol de Recubrimiento DFS

Para que te familiarices y pongas en práctica el concepto de Árbol de Recubrimiento hemos preparado el siguiente ejercicio:

Ejercicio 10: en la clase Grafo, diseña un método `spanningTreeDFS` que, con el menor coste posible, devuelva un *Spanning Tree* DFS de un Grafo No Dirigido, o `null` si no existe ninguno. En concreto, si encuentra un *Spanning Tree*, este método debe devolver un array de `String`, cada uno de los cuales representa una arista (v, w) de las que lo componen

Consejo: en lugar de empezar el diseño de `spanningTreeDFS` desde cero, modifica donde creas necesario el método `toArrayDFS` de la misma clase que aparece en la siguiente transparencia

3. Aplicaciones DFS y BFS

Ejercicio 10: modifica el método `toArrayDFS()` que figura a continuación para convertirlo en `spanningTreeDFS()`



```
/* devuelve un array cuyos elementos son los vértices
 * de un Grafo en Pre-Orden DFS */
public int[] toArrayDFS() {
    int[] res = new int[numVertices()]; ordenVisita = 0;
    visitados = new int[numVertices()];
    for (int v = 0; v < numVertices(); v++) {
        if (visitados[v] == 0) { toArrayDFS(v, res); }
    }
    return res;
}

protected void toArrayDFS(int v, int[] res) {
    visitados[v] = 1;
    res[ordenVisita] = v; ordenVisita++;
    ListaConPI<Adyacente> l = adyacentesDe(v);
    for (l.inicio(); !l.esFin(); l.siguiente()) {
        int w = l.recuperar().getDestino();
        if (visitados[w] == 0) { toArrayDFS(w, res); }
    }
}
```

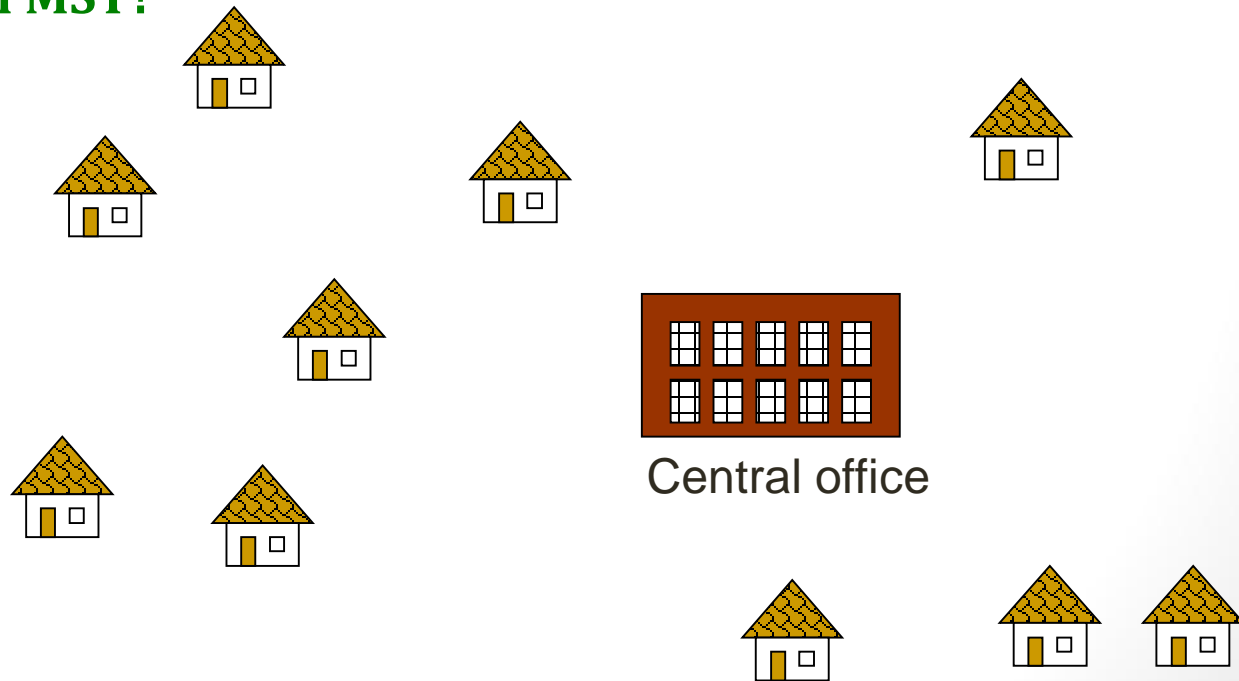
3. Aplicaciones DFS y BFS

Árbol de Recubrimiento Mínimo, o problema MST (Minimum Spanning Tree)

▪ *Ejemplo 1: Tendido de la línea telefónica*

Una compañía telefónica tiene que tender las líneas de teléfono necesarias para prestar sus servicios a una zona con $|V|$ casas (incluida su central). Un solo cable debe conectar cada par de casas y todas las casas deben estar interconectadas a un cable

¿Dónde está el MST?



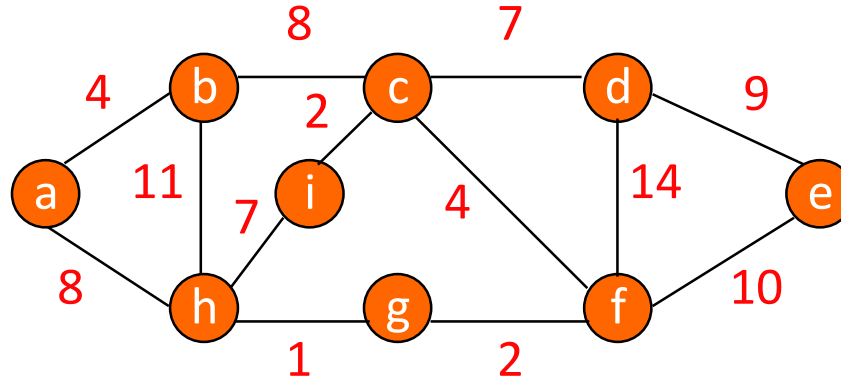
3. Aplicaciones DFS y BFS

Árbol de Recubrimiento Mínimo, o problema MST

- **Ejemplo 2:** Conectar entre sí varios pins para conseguir su equivalencia eléctrica

Interconectar un conjunto de $|V|$ pins usando $|V|-1$ cables, cada uno de los cuales conecta dos pins, **Y LA MENOR CANTIDAD DE CABLE POSIBLE**

¿Dónde está el MST?



3. Aplicaciones DFS y BFS

Árbol de Recubrimiento Mínimo, o problema MST

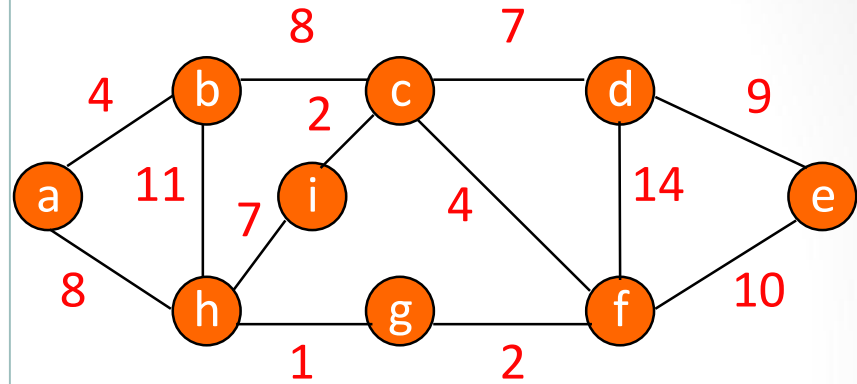
- *¿Cómo construir el MST? Un primer algoritmo*



Algoritmo de Kruskal

Idea: dado un Grafo $G = (V, E)$, construir *paso a paso, arista a arista por orden creciente de peso*, un MST de G , $T = (V', E')$

- **Paso 0**, inicialización de la iteración: por definición de MST, los nodos del T son los vértices del Grafo: $V' = V$; el conjunto de aristas del MST está vacío: $E' = \emptyset$; en principio, cualquier arista del Grafo G es susceptible de formar parte de E' , o es una arista Factible: **aristasFactibles** = E ;
- **Paso i** de la iteración:
 - i.1.** Seleccionar aquella arista (v, w) de **aristasFactibles** tal que su peso sea mínimo: $(v, w) = \min(\text{aristasFactibles})$;
 - i.2.** Si la arista (v, w) NO ORIGINA UN CICLO, entonces $E' = E' \cup (v, w)$
- **Terminación:** $|E'| = |V| - 1$



3. Aplicaciones DFS y BFS

Árbol de Recubrimiento Mínimo, o problema MST

- ¿Cómo construir el MST? Un segundo algoritmo

Algoritmo de Prim

Idea: dado un Grafo $G = (V, E)$, construir paso a paso, **Vértice a Vértice por niveles de adyacencia**, un MST de G , $T = (V', E')$

- **Paso 0:** la Raíz del Árbol T es un vértice (cualquiera) del Grafo G : por ejemplo, $V' = \{a\}$; $V = V - \{a\}$; el conjunto de aristas del MST está vacío: $E' = \emptyset$;

- **Paso i:** dado un vértice $u \in V'$

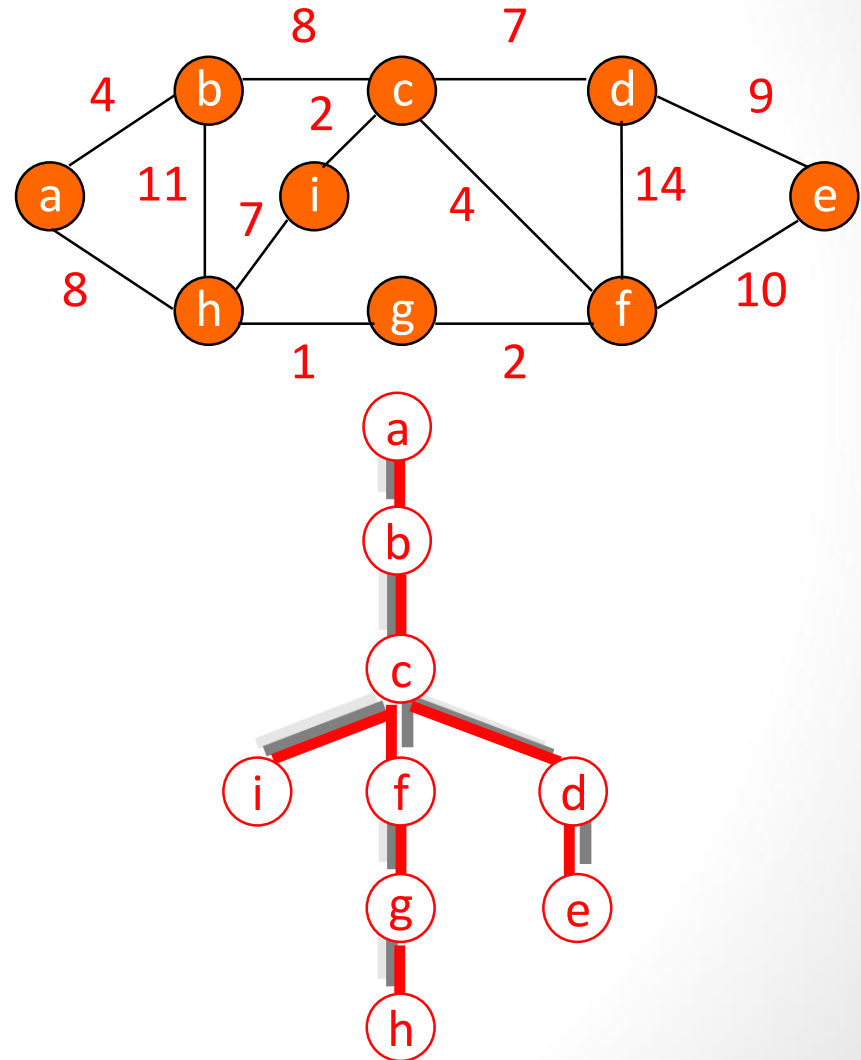
i.1. Seleccionar su Adyacente $w \notin V'$ tal que el peso de la arista (v, w) sea mínimo:

$w = \min(\text{verticesFactibles})$; $V = V - \{w\}$

i.2. $V' = V' \cup w$; $E' = E' \cup (v, w)$

- **Terminación:** $|E'| = |V| - 1$

$T_{\text{prim}}(|V|, |E|) \in O(|E| \cdot \log|E|)$



3. Aplicaciones DFS y BFS

Camino desde un vértice dado: conceptos previos

Nos interesan los siguiente conceptos:

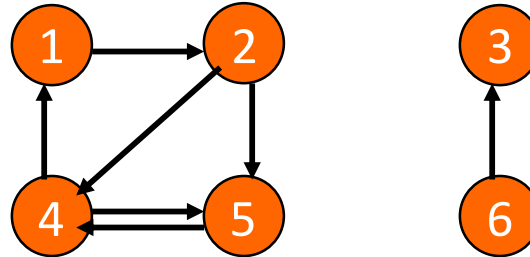
- Camino
 - Longitud
 - Camino Simple
 - Bucle y Ciclo
-
- [http://en.wikipedia.org/wiki/Path \(graph theory\)](http://en.wikipedia.org/wiki/Path_(graph_theory))

3. Aplicaciones DFS y BFS

Camino desde un vértice dado: cuestión



Sea $G = (V, E)$ el siguiente Digrafo



Completa las siguientes frases:

- El vértice 5 es alcanzable desde los vértices , y ; sin embargo, el vértice no es alcanzable desde ningún otro vértice
- $\langle 1, , , \rangle$ es un camino Simple de longitud 3 desde el vértice 1
- $\langle 2, , , \rangle$ es un camino No Simple de longitud 3 desde el vértice 2
- $\langle 1, , , , \rangle$ es un Ciclo de longitud 4 desde el vértice 1

3. Aplicaciones DFS y BFS

*Caminos mínimos **sin pesos** desde un vértice dado, como modificación de `toArrayBFS()`*

Enunciado: encontrar el camino más corto (medido en nº de aristas) desde un vértice origen v al resto de vértices de un Grafo

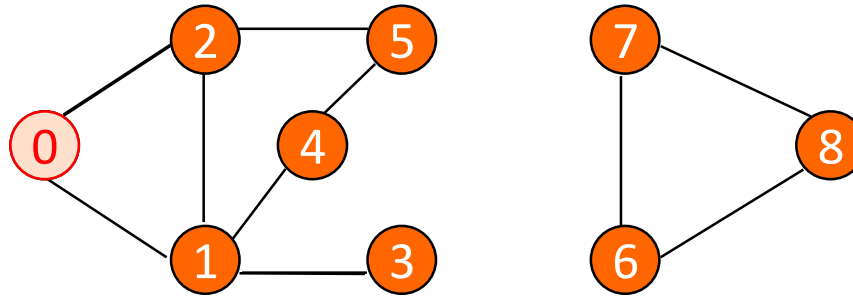
Ideas básicas para modificar el código de `toArrayBFS(v)`

- 1) En el BFS de v se visitan todos los vértices w alcanzables desde v
- 2) La distancia (longitud de camino) se mide en nº de aristas
- 3) El resultado del método tiene que cambiar: para representar el camino más corto de v a w se usan **dos atributos auxiliares** - ambos de tipo array, con longitud `numVertices()` y que se inicializan DENTRO del BFS de v
 - `double[] distanciaMin`, tal que `distanciaMin[w]` es la distancia mínima de v a w , $\forall w$. Con el valor ∞ se indica que w no ha sido visitado en el BFS de v , i.e. no hay camino de v a w
 - `int[] caminoMin`, tal que `caminoMin[w]` es el vértice del camino mínimo entre v y w desde el que se alcanza w , $\forall w$. Con el valor **-1** se indica que w no visitado en el BFS de v , i.e. no hay camino de v a w

3. Aplicaciones DFS y BFS

Caminos mínimos sin pesos desde un vértice dado, como modificación de `toArrayBFS()`

Traza ejemplo: encontrar el camino más corto (medido en nº de aristas) desde $v = 0$ al resto de vértices de un Grafo



3. Aplicaciones DFS y BFS

*El método caminosMinimosSinPesos de la clase Grafo
(cambios con respecto a toArrayBFS() en verde)*

```
protected Cola<Integer>; // atributo "auxiliar"
protected double[] distanciaMin; protected int[] caminoMin; // atributos "auxiliares"
protected static final double INFINITO = Double.POSITIVE_INFINITY; // atributo "auxiliar"
public void caminosMinimosSinPesos(int v) {
    caminoMin = new int[numVertices()]; distanciaMin = new double[numVertices()];
    for (int i = 0; i < numVertices(); i++) {
        distanciaMin[i] = INFINITO;
        caminoMin[i] = -1;
    }
    distanciaMin[v] = 0;
    q = new ArrayCola<Integer>();
    caminosBFS(v);
}
protected void caminosBFS(int v) {
    q.encolar(new Integer(v));
    while (!q.esVacia()) {
        int u = q.desencolar().intValue();
        ListaConPI<Adyacente> l = adyacentesDe(u);
        for (l.inicio(); !l.esFin(); l.siguiente()) {
            int w = l.recuperar().getDestino();
            if (distanciaMin[w] == INFINITO) {
                distanciaMin[w] = distanciaMin[u] + 1; caminoMin[w] = u;
                q.encolar(new Integer(w));
            }
        }
    }
}
```

3. Aplicaciones DFS y BFS

Simplificación del método caminosMinimosSinPesos

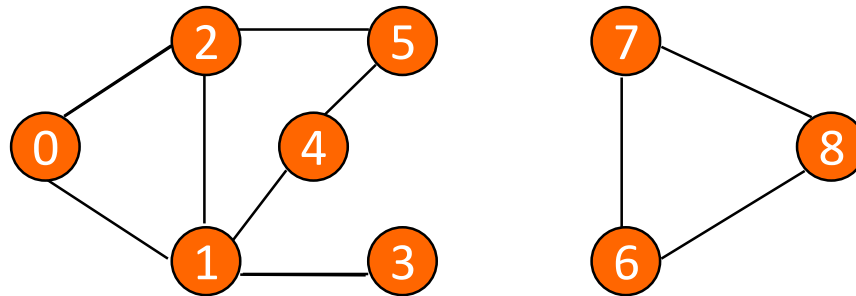
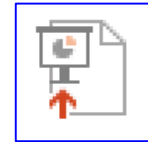
```
protected Cola<Integer>; // atributo "auxiliar"
protected double[] distanciaMin; // atributo "auxiliar"
protected int[] caminoMin; // atributo "auxiliar"
protected static final double INFINITO = Double.POSITIVE_INFINITY; // atributo "auxiliar"
public void caminosMinimosSinPesos(int v) {
    caminoMin = new int[numVertices()]; distanciaMin = new double[numVertices()];
    for (int i = 0; i < numVertices(); i++) {
        distanciaMin[i] = INFINITO;
        caminoMin[i] = -1;
    }
    distanciaMin[v] = 0;
    q = new ArrayCola<Integer>();
    // código de caminosBFS, en lugar de invocación
    q.encolar(new Integer(v));
    while (!q.esVacía()) {
        int u = q.desencolar().intValue();
        ListaConPI<Adyacente> l = adyacentesDe(u);
        for (l.inicio(); !l.esFin(); l.siguiente()) {
            int w = l.recuperar().getDestino();
            if (distanciaMin[w] == INFINITO) {
                distanciaMin[w] = distanciaMin[u] + 1; caminoMin[w] = u;
                q.encolar(new Integer(w));
            }
        }
    }
}
```


3. Aplicaciones DFS y BFS

Camino mínimo sin pesos entre 2 vértices dados reutilizando caminos
`MinimosSinPesos(v)`

Enunciado: encontrar el camino más corto (medido en nº de aristas) entre los vértices v y w

Traza ejemplo 1: encontrar el camino más corto entre $v = 0$ y $w = 5$



Tras finalizar `caminosMinimosSinPesos` ($v=0$) ...

`distanciaMin` almacena la distancia mínima de v a w , $\forall w$

0	1	1	2	2	2	∞	∞	∞
0	1	2	3	4	5	6	7	8

¿Cuál es la longitud del camino mínimo de 0 a 5?

`caminoMin` almacena (*codificado*) el camino mínimo de v a w , $\forall w$

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

¿Cuál es el camino mínimo de 0 a 5?

3. Aplicaciones DFS y BFS

*Caminos mínimos sin pesos entre 2 vértices dados,
reutilizando caminosMinimosSinPesos(v)*

```
/** SII v != w AND 0 <= v < numVertices() AND 0 <= w < numVertices()
 * devuelve una ListaConPI con los vértices del camino mínimo
 * SIN pesos entre v y w, o una lista vacía si tal camino no existe
 */
public ListaConPI<Integer> caminoMinimoSinPesos(int v, int w) {
    caminosMinimosSinPesos(v);
    return decodificarCaminoHasta(w);
}

// SII 0 <= w < numVertices()
// devuelve una ListaConPI con los vértices del camino hasta w
protected ListaConPI<Integer> decodificarCaminoHasta(int w) {
    ListaConPI<Integer> res = new LEGListaConPI<Integer>();
    // PISTAS:
    // SI distanciaMin[w] == INFINITO devolver lista vacía
    // SINO decodificar el camino de caminoMin que acaba en w:
    //     1.- El camino de caminoMin que acaba en w empieza
    //         en el vértice v tal que caminoMin[v] = -1
    //     2.- Se alcanza w desde el vértice caminoMin[w]
    ...
    return res;
}
```

3. Aplicaciones DFS y BFS

*Caminos mínimos CON pesos - Algoritmo de Dijkstra,
como modificación de caminosMinimosSinPesos*

```
/** SII v != w AND 0 <= v < numVertices() AND 0 <= w < numVertices()  
 * devuelve una ListaConPI con los vértices del camino  
 * mínimo CON pesos entre v y w, o una lista vacía si  
 * tal camino no existe */
```

```
public ListaConPI<Integer> caminoMinimo(int v, int w) {  
    dijkstra(v);  
    return decodificarCaminoHasta(w);  
}
```

```
public void dijkstra(int v) {
```



```
// MODIFICAR "ADUECUADAMENTE" caminosMinimosSinPesos(int v)  
...  
}
```

3. Aplicaciones DFS y BFS

Caminos mínimos CON pesos - Algoritmo de Dijkstra

```
protected void dijkstra(int s) {
    distanciaMin = new double[numVertices()]; caminoMin = new int[numVertices()];
    for (int i = 0; i < numVertices(); i++) {
        distanciaMin[i] = INFINITO;
        caminoMin[i] = -1;
    }
    distanciaMin[s] = 0;
    ColaPrioridad qP = new PriorityQColaPrioridad<AlcanzableDijkstra>();
    visitados = new int[numVertices()];
    // caminosBFS(s);
    qP.insertar(new AlcanzableDijkstra(s, distanciaMin[s]));
    while (!qP.esVacia()) {
        int u = qP.eliminarMin().vAlcanzable;
        if (visitados[u] == 0) {
            visitados[u] = 1;
            ListaConPI<Adyacente> l = adyacentesDe(u);
            for (l.inicio(); !l.esFin(); l.siguiente()) {
                int w = l.recuperar().getDestino();
                double pesoW = l.recuperar().getPeso();
                if (distanciaMin[w] > distanciaMin[u] + pesoW) {
                    distanciaMin[w] = distanciaMin[u] + pesoW;
                    caminoMin[w] = u;
                    qP.insertar(new AlcanzableDisjksstra(w, distanciaMin[w]));
                }
            }
        }
    }
}
```

3. Aplicaciones DFS y BFS

Ejercicios propuestos

Asumiendo como Precondición que el Grafo sobre el que se aplican es No Dirigido, implementa en la clase Grafo con el menor coste posible los siguientes dos métodos:

Ejercicio 11: esConexo, que comprueba si un Grafo es Conexo

Ejercicio 12: toStringCC , que devuelve un String con los vértices de las Componentes Conexas de un Grafo

Ejercicio 13: Vértice Raíz de un Grafo Acíclico y su coste



(clave: CCDL000Z)

Completa en la clase Grafo el código del método verticeRaiz para que, con coste mínimo, devuelva el primer vértice Raíz que encuentre en un Grafo Acíclico, o -1 si tal vértice no existe

Nota: dado un Grafo G , un vértice v es una Raíz de G si cada uno de los vértices de G es alcanzable desde v

Tema 6 – S5 Y S6

Grafo: Jerarquía Java y Aplicaciones

Contenidos

3. Aplicaciones BFS y DFS

- Árbol de Recubrimiento

- Conceptos previos: Subgrafo, Árbol y Árbol de Recubrimiento
- Árbol de Recubrimiento BFS (Práctica 6 - S1)
- Árbol de Recubrimiento Mínimo: algoritmo de Kruskal (Práctica 6 - S2)

- Camino desde un vértice dado

- Conceptos previos: camino entre 2 vértices, Camino Simple
- Caminos BFS: caminos mínimos SIN pesos
- Caminos mínimos (BFS) CON pesos: algoritmo de Dijkstra

- Resolución de ejercicios propuestos (11, 12 y planteamiento del 13)
- Ordenación Topológica y Test de Aciclicidad como Recorridos DFS



3. Aplicaciones DFS y BFS

La estrategia DFS Post-Orden (I)

Antes de hablar de Ordenación Topológica y Test de Aciclicidad es necesario hablar sobre la implementación de un DFS **Post-Orden**. Para ello, se te propone el siguiente ejercicio:

Ejercicio 14: en la clase Grafo, diseña un método `finDelDFS` que, con coste mínimo, devuelva un array cuyos elementos sean los vértices de un Grafo en el orden en el que finaliza su DFS

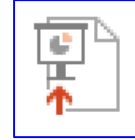
Consejo: en lugar de empezar el diseño de `finDelDFS` desde cero, modifica donde creas necesario el método `toArrayDFS` de la misma clase que aparece en la siguiente transparencia

PISTA: marca el final del DFS de un vértice v poniendo un 2 en `visitados[v]`

.

3. Aplicaciones DFS y BFS

La estrategia DFS Post-Orden (II)



```
/* devuelve un array cuyos elementos son los vértices
 * de un Grafo en Pre-Orden DFS */
public int[] toArrayDFS() {
    int[] res = new int[numVertices()]; ordenVisita = 0;
    visitados = new int[numVertices()];
    for (int v = 0; v < numVertices(); v++) {
        if (visitados[v] == 0) { toArrayDFS(v, res); }
    }
    return res;
}

protected void toArrayDFS(int v, int[] res) {
    visitados[v] = 1;
    res[ordenVisita] = v; ordenVisita++;
    ListaConPI<Adyacente> l = adyacentesDe(v);
    for (l.inicio(); !l.esFin(); l.siguiente()) {
        int w = l.recuperar().getDestino();
        if (visitados[w] == 0) { toArrayDFS(w, res); }
    }
}
```


3. Aplicaciones DFS y BFS

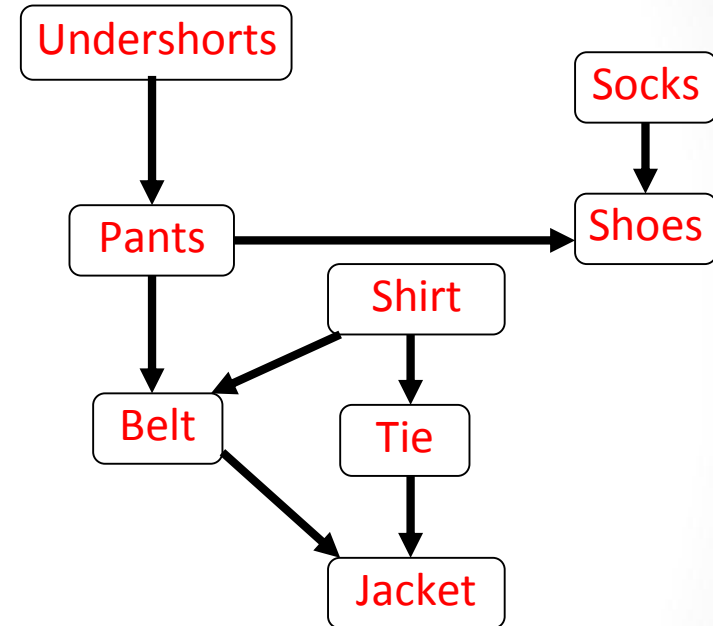
Ordenación Topológica y Test de Aciclicidad (DAG)

- *Qué tienen que ver un DAG y la planificación factible de tareas*

Ejemplo 1:

El DAG (*Directed Acyclic Graph*) de la figura representa, o visualiza, las diferentes **planificaciones**, o caminos, **que un hombre puede seguir para conseguir vestirse**:

- **V** es el conjunto de prendas a usar
- **E** es el conjunto de prerequisites entre pares de prendas (v, w): **la prenda v debe ponerse ANTES que la w**



Una planificación factible sería...

[Undershorts, Socks, Shirt, Pants, Belt, Tie, Jacket, Shoes]

3. Aplicaciones DFS y BFS

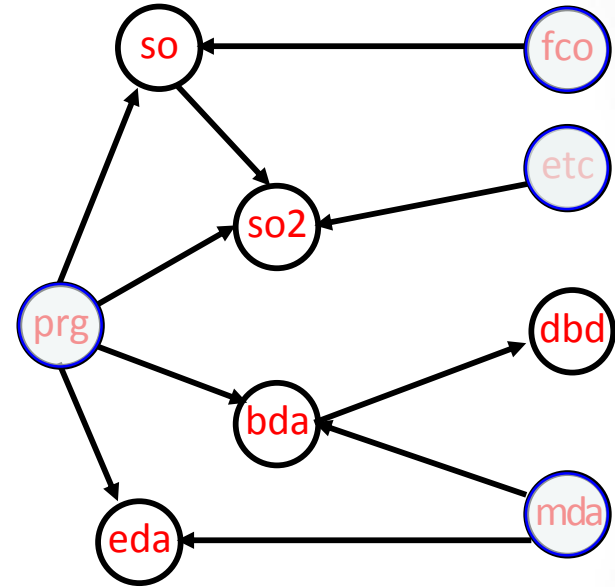
Ordenación Topológica y Test de Aciclicidad (DAG)

- *Qué tienen que ver un DAG y la planificación factible de tareas*

Ejemplo 2:

El DAG (*Directed Acyclic Graph*) de la figura representa, o visualiza, las diferentes **planificaciones**, o caminos, **que un estudiante puede seguir para conseguir graduarse**:

- **V** es el conjunto de asignaturas a cursar
- **E** es el conjunto de prerequisites entre pares de asignaturas (v, w) : **la asignatura v debe ser cursada ANTES que la w**



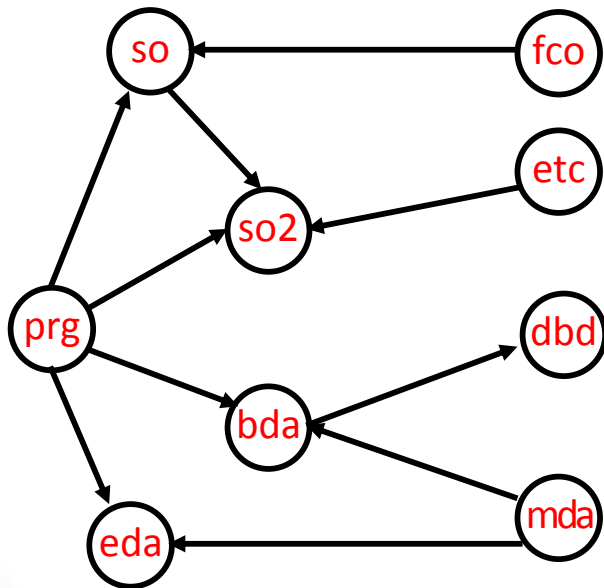
Una planificación factible sería [fco, etc, mda, prg, eda, bda, dbd, so, so2]

3. Aplicaciones DFS y BFS

Ordenación Topológica (DAG) - Definición del problema

Sea G un DAG:

- Un **Orden Topológico (OT)**, o Secuencia Topológica, es una ordenación de todos los vértices de G v_1, v_2, \dots, v_n tal que para cada arista (v_i, v_k) de E , $i < k$
Gráficamente, toda arista de la Secuencia está orientada hacia la derecha
- Una **Ordenación Topológica** es el proceso de obtención de un Orden Topológico de G



- Para un DAG dado suelen existir bastantes órdenes topológicos factibles:

[fco, etc, mda, prg, eda, bda, dbd, so, so2]

[prg, fco, so, etc, so2, mda, bda, dbd, eda]

[fco, prg, etc, mda, eda, bda, dbd, so, so2]

...

- Cada OT constituye una **planificación factible**

3. Aplicaciones DFS y BFS

Ordenación Topológica - Resolución en 2 pasos



Paso 1: ¿elegir el DFS “apropiado?” (“la” Ordenación Topológica) para obtener un Orden topológico del DAG, p.ej. [fco, etc, mda, prg, eda, bda, dbd, so, so2]

Pista: prueba con un DFS Post-Orden (`finDelDfs()`) y observa en qué se diferencia del Orden Topológico que quieres obtener

prg: so, so2, bda, eda

so: so2

so2: \emptyset

bda: dbd

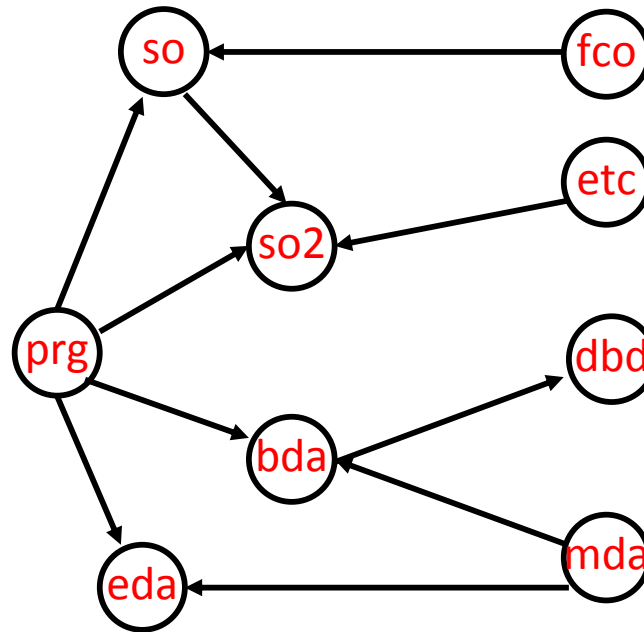
dbd: \emptyset

eda: \emptyset

mda: eda, bda

etc: so2

fco: so



3. Aplicaciones DFS y BFS

Ordenación Topológica - Resolución en 2 pasos

Paso 2: obtener el **INVERSO** del DFS Post-Orden del DAG, pues es el Orden Topológico deseado (cambios respecto a `finDelDFS()` en **verde**)

```
/* SII el Grafo es un DAG */
public int[] ordenTopologicoDFS() {
    int[] res = new int[numVertices()]; ordenVisita = 0;
    visitados = new int[numVertices()];
    for (int v = 0; v < numVertices(); v++) {
        if (visitados[v] == 0) { ordenTopologicoDFS(v, res); }
    }
    return res;
}

protected void ordenTopologicoDFS(int v, int[] res) {
    visitados[v] = 1;
    ListaConPI<Adyacente> l = adyacentesDe(v);
    for (l.inicio(); !l.esFin(); l.siguiente()) {
        int w = l.recuperar().getDestino();
        if (visitados[w] == 0) { ordenTopologicoDFS(w, res); }
    }
    visitados[v] = 2;
    res[numVertices() - 1 - ordenVisita] = v; ordenVisita++;
}

¡Invertir SÓLO la numeración!
```

3. Aplicaciones DFS y BFS

Test de Aciclicidad - ¿Por qué SOLO se puede realizar una Ordenación Topológica para un DAG?

Ejemplo:

Sea el Grafo Dirigido **CON** un ciclo de la figura...

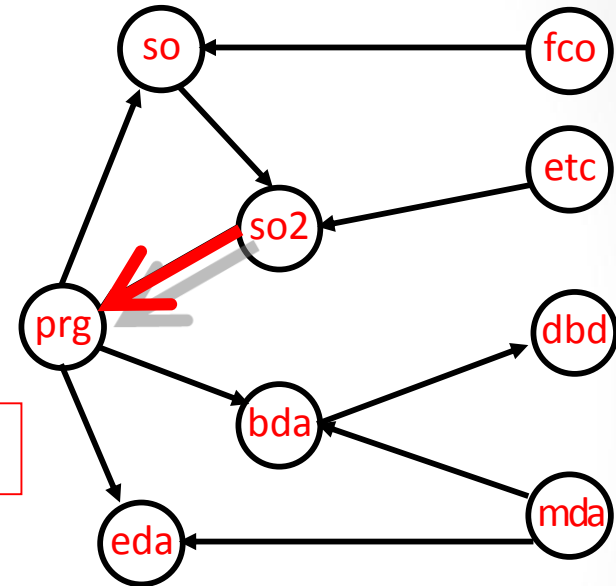
¿Puede un estudiante trazar en él un camino -al menos- que le permita “alcanzar” la graduación y, al mismo tiempo, sea compatible con el plan de estudios?

¡Imposible!

Si v (**prg**) y w (**so2**) son vértices de un ciclo, existe un camino de v a w y de w a v ...

¿Qué asignatura debe cursar **antes**, **prg** o **so2**?

Resulta imposible saberlo, pues en un camino **prg** precede a **so2** mientras que en el otro es **so2** la que precede a **prg**



3. Aplicaciones DFS y BFS

Test de Aciclicidad – Estrategia de resolución

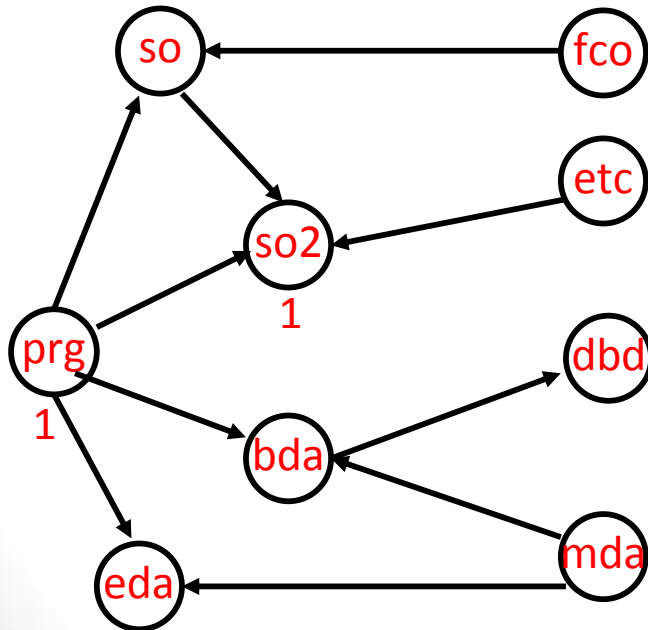


Detectar el primer ciclo de un Grafo Dirigido es encontrar su primera **arista Hacia Atrás (HA)**; si tal arista no existe, el Grafo es un DAG

Una **arista HA** (w, v) es una arista donde **v ya ha sido visitado** y es un **antecesor de w** (v precede a w en un Orden Topológico)

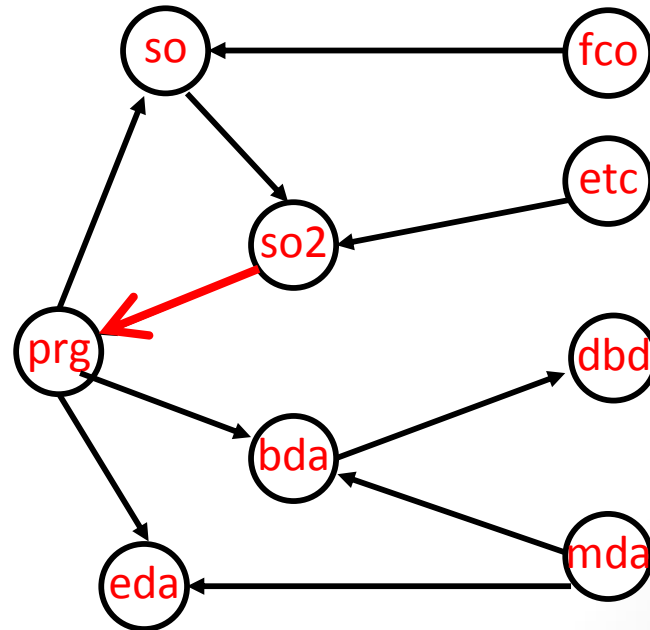
¿Es Acíclico el Grafo?

SÍ



¿Es Acíclico el Grafo?

NO



3. Aplicaciones DFS y BFS

Test de Aciclicidad – Método hayCicloDFS()

(cambios con respecto a toArrayDFS() en verde)

```
// SII el Grafo es un Digrafo...

public boolean hayCicloDFS() {
    boolean ciclo = false;
    visitados = new int[numVertices()];
    for (int v = 0; v < numVertices() && !ciclo; v++) {
        if (visitados[v] == 0) { ciclo = hayAristaHADFS(v); }
    }
    return ciclo;
}

protected boolean hayAristaHADFS(int v) {
    boolean aristaHA = false; visitados[v] = 1;
    ListaConPI<Adyacente> l = adyacentesDe(v);
    for (l.inicio(); !l.esFin() && !aristaHA; l.siguiente()) {
        int w = l.recuperar().getDestino();
        if (visitados[w] == 0) { aristaHA = hayAristaHADFS(w); }
        else if (visitados[w] == 1) { aristaHA = true; }
    }
    visitados[v] = 2;
    return aristaHA;
}
```


3. Aplicaciones DFS y BFS

Test de Aciclicidad JUNTO CON Ordenación Topológica

(cambios con respecto al primer método presentado en verde)

```
// SII el Grafo es Dirigido...
```

```
public int[] ordenTopologicoDFS() {
```

```
    int[] res = new int[numVertices()]; ordenVisita = 0;
```

```
    visitados = new int[numVertices()]; boolean ciclo = false;
```

```
    for (int v = 0; v < numVertices() && !ciclo; v++) {
```

```
        if (visitados[v] == 0) { ciclo = ordenTopologicoDFS(v, res); }
```

```
    }
```

```
    if (!ciclo) { return res; }
```

```
    return null;
```

```
}  
protected boolean ordenTopologicoDFS(int v, int[] res) {
```

```
    boolean aristaHA = false; visitados[v] = 1;
```

```
    ListaConPI<Adyacente> l = adyacentesDe(v);
```

```
    for (l.inicio(); !l.esFin() && !aristaHA; l.siguiente()) {
```

```
        int w = l.recuperar().getDestino();
```

```
        if (visitados[w] == 0) { aristaHA = ordenTopologicoDFS(w, res); }
```

```
        else if (visitados[w] == 1) { aristaHA = true; }
```

```
    }
```

```
    visitados[v] = 2;
```

```
    res[numVertices() - 1 - ordenVisita] = v; ordenVisita++;
```

```
    return aristaHA;
```

```
}
```