

1. Introducción
2. Estructuras de datos lineales
3. Estructuras de datos jerárquicas
- 4. Grafos y caminos**
5. Implementación de listas, colas, y pilas
6. Implementación de mapas, árboles, y grafos

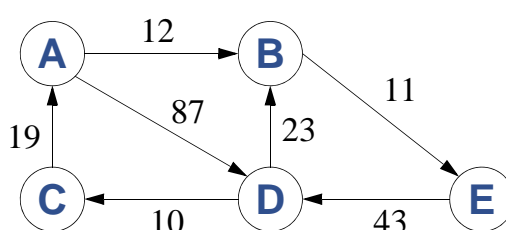
4. Grafos y Caminos

- 4.1. Concepto de grafo
- 4.2. Definiciones
- 4.3. La interfaz de las aristas
- 4.4. La interfaz de los grafos
- 4.5. Cálculo de caminos mínimos sin pesos
- 4.6. Cálculo de caminos mínimos con pesos positivos
- 4.7. Cálculo de caminos mínimos con pesos negativos
- 4.8. Cálculo de caminos en grafos acíclicos

4.1 Grafos

Un **grafo** es una estructura de datos que almacena datos de dos tipos:

- **vértices** o nudos, con un valor almacenado
- **aristas** o arcos: cada una conecta a un vértice con otro, y puede tener un valor almacenado
 - una arista es un par de vértices (v,w)
 - si el par está ordenado, se dice que el grafo es **dirigido** o que es un **digrafo**



Notas:

Los **grafos**, constituyen estructuras de datos en las que se pueden expresar relaciones de conexión entre diversos elementos denominados vértices. Cada conexión se representa por un dato llamado arista

Los grafos tienen gran cantidad de aplicaciones; por ejemplo:

- Representación de circuitos electrónicos analógicos y digitales
- Representación de caminos o rutas de transporte entre localidades
- Representación de redes de computadores.

Uno de los problemas más importantes en los grafos es el de encontrar el camino de coste mínimo.

4.2. Definiciones

- **Adyacente:** se dice que w es adyacente a v si existe la arista (v, w)
 - en un grafo dirigido, no es lo mismo que v sea adyacente a w que al revés
- **Camino:** secuencia de vértices tales que cada uno es adyacente al anterior
- **Peso o coste:** las aristas pueden contener datos y uno de ellos puede ser el coste o peso asociado a esa arista.
 - se usa para determinar el coste de recorrer el camino
- **Longitud del camino:** n^0 de aristas que tiene
- **Coste de un camino:** la suma de los pesos de sus aristas
- **Camino simple:** es aquel en que todos los vértices son distintos, excepto quizás el primero y el último

Definiciones (cont.)

- **Ciclo:** es un camino de longitud al menos 1 que empieza y acaba en el mismo vértice
- **Grafo dirigido acíclico:** es un grafo dirigido sin ciclos
- **Grafo denso:** es aquel que tiene un gran número de aristas
 - cercano al número de vértices, V , al cuadrado
- **Grafo disperso:** es aquel en que el número de aristas E es pequeño $E \ll V^2$
 - es el tipo de grafo más habitual
 - intentaremos optimizar las operaciones del grafo para este caso

4.3. La interfaz de las aristas

Los vértices del grafo se suelen numerar para trabajar con ellos más cómodamente

- disponen de un identificador entero

Las aristas son objetos que contienen

- el identificador del vértice origen
 - en ocasiones este dato no se almacena, pues se averigua a partir del grafo
- el identificador del vértice destino
- un contenido
 - el peso, en el caso de los grafos con pesos
 - opcionalmente, más datos

4.3. La interfaz de las aristas

Operaciones

operación	argumentos	retorna	errores
constructor	idOrigen, idDestino, peso, ...	Arista	
idOrigen		entero	
idDestino		entero	
contenido		Contenido	

La clase Arista

```
package adts;

public class Arista<A>
{
    public Arista (int origen, int destino,
                  A contenido) {...}

    public int destino() {...}

    public int origen() {...}

    public A contenido() {...}
}
```

4.4. La interfaz abstracta de los grafos

Operaciones del grafo:

operación	argumentos	retorna	errores
constructor		Grafo	
nuevaArista	elemOrigen, elemDestino, contArista		
idVertice	elemento	entero	NoExiste
contenido	entero	Elemento	IdIncorrecto
listaAristas	entero	List<Arista>	IdIncorrecto
numVertices		entero	
numAristas		entero	

El elemento que se guarda en los vértices debe implementar

- `hashCode()` e `equals()`

- *constructor*: Crea el grafo vacío
- *nuevaArista*: Inserta una nueva arista con peso a partir de las descripciones de sus vértices. Si los vértices son nuevos, los añade al grafo
- *idVertice*: Retorna el identificador del vértice indicado. Lanza **NoExiste** si el vértice no pertenece al grafo
- *contenido*: Retorna el contenido del vértice cuyo identificador se indica. Lanza **IdIncorrecto** si ese identificador no pertenece al grafo
- *listaAristas*: Retorna la lista de aristas del vértice de identificador **idVertice**. Lanza **IdIncorrecto** si ese identificador no pertenece al grafo
- *numVertices*: Retorna el número de vértices
- *numAristas*: Retorna el número de aristas

La interfaz Java de los grafos

```
package adts;
import java.util.*;
public interface Grafo<V,A>
{
    Arista nuevaArista (V vertice1, V vertice2,
                       A contenidoArista);
    int idVertice(V vertice)
        throws NoExiste;
    V contenido(int idVertice)
        throws IdIncorrecto;
    List<Arista<A>> listaAristas(int idVertice)
        throws IdIncorrecto;
    int numVertices();
    int numAristas();
}
```

Ejemplo con grafos

1. Escribir un método para recorrer todos los arcos de un grafo y mostrarlos en la pantalla
2. Escribir un método al que se le pase un camino, como una lista de valores de vértices, y nos calcule su coste o peso total

4.5. Cálculo de caminos mínimos sin pesos

El problema del cálculo del camino mínimo se define como

- Encontrar el camino de menor coste desde un vértice dado Origen (O) hasta cualquier otro vértice del grafo
- Cuando no hay pesos, el coste es la longitud del camino

Tiene muchas aplicaciones. Por ejemplo:

- ruta más rápida para un transporte
- camino más corto para un e-mail en una red de computadores

Resolución del problema

Resolveremos el problema calculando los caminos mínimos a otros vértices

- el camino desde O a sí mismo es de longitud 0
- **procesar O** : buscamos los vértices cuyo camino mínimo desde O es de longitud 1
 - son los vértices adyacentes a O
- **procesar vértices adyacentes a O** : buscamos los vértices cuyo camino mínimo desde O es de longitud 2
 - son los vértices aún no visitados que son adyacentes a los vértices del paso anterior
- en el paso i , buscamos los vértices cuyo camino mínimo desde O es de longitud i
 - vértices aún no visitados adyacentes a los del paso anterior

Resolución del problema

En este algoritmo se garantiza que la primera vez que se visita un vértice se ha hecho por el camino mínimo

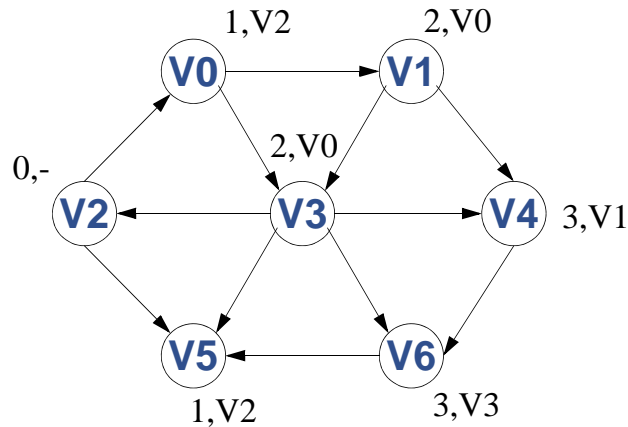
Organización de los datos

- Necesitamos guardar en cada vértice el hecho de que lo hemos **visitado** o no
- Para **recordar el camino**, podemos anotar en cada vértice cuál es el vértice desde el que procede el camino
- Necesitamos almacenar el conjunto de vértices a **procesar**
 - podemos usar una cola

Como cada arista se recorre como máximo una vez, el coste de este algoritmo en el peor caso es $O(E)$, siendo E el número de aristas

Ejemplo

En el grafo siguiente se han anotado los sucesivos valores de longitud del camino y vértice anterior, suponiendo el origen = **V2**



Pseudocódigo del camino mínimo

método caminoMinimoSinPesos (Grafo g,
idVertice origen, destino)
retorna Camino

// inicializaciones

```
Cola<idVertice> procesar := nueva Cola vacía;  
idVertice anterior[] := nuevo array[numVertices]  
boolean visitado[] := nuevo array[numVertices] =  
false para todas sus casillas
```

```
procesar.inserta(origen)
```

```
visitado[origen] := true
```

```
anterior[origen] := -1 // para indicar que no tiene
```

Pseudocódigo del camino mínimo (cont.)

```
mientras procesar no esta vacío hacer
  idVertice v:=procesar.extraer()
  Lista adj:=g.listaAristas(v)
  para cada arista a de adj hacer
    idVertice dest:=a.destino;
    si no visitado[dest] entonces
      anterior[dest]:=v
      si dest==destino entonces
        retorna caminoDe(origen,destino,anterior);
      fsi
    visitado[dest]:=true;
    procesar.inserta(dest);
  fsi
fpara
fmientras
```

Pseudocódigo del camino mínimo (cont.)

indicar error (el destino no ha sido hallado)
fmétodo

```
método caminoDe(idVertice origen, destino,
                 idVertice[] anterior)
retorna Camino
  Camino cam:= nuevo camino vacío
  idVertice v:=destino
  mientras anterior[v]!=-1 hacer
    cam.inserta(v)
    v:=anterior[v];
  fmientras
  cam.inserta(origen);
  retorna cam;
fmétodo
```

La clase CaminoSinPesos

Operaciones del camino:

operación	argumentos	retorna	errores
constructor	Grafo	Camino	
inserta	idVertice		
vertice	entero	idVertice	NoExiste
longitud		entero	
muestra			

- **constructor**: retorna un camino vacío
- **inserta**: inserta los vértices del camino empezando por el final
- **vertice**: retorna el vértice *i*-ésimo
- **longitud**: retorna la longitud del camino

Implementación en Java

```
import adts.*;
import java.util.*;
/**
 * Clase que contiene algoritmos para la resolución
 * del problema de caminos mínimos en grafos
 */
public class Caminos
{
    public static <V,A> CaminoSinPesos
        caminoMinimoSinPesos
            (Grafo<V,A> g, int origen, int destino)
                throws NoExiste, IdIncorrecto
    {
```

Implementación en Java (cont.)

```
// inicializaciones
LinkedList<Integer> procesar=
    new LinkedList<Integer>();
int[] anterior= new int[g.numVertices()];
boolean[] visitado=
    new boolean[g.numVertices()];
// todas las casillas valen vale false por omisión
// en Java

procesar.addLast(new Integer(origen));
visitado[origen]=true;
anterior[origen]=-1; // para indicar que no tiene
```

Implementación en Java (cont.)

```
while (!procesar.isEmpty()) {
    int v=procesar.removeFirst().intValue();
    List<Arista<A>> adj=g.listaAristas(v);
    for (Arista<A> a:adj) {
        int dest=a.destino();
        if (!visitado[dest]) {
            anterior[dest]=v;
            if (dest==destino) {
                // hemos encontrado el camino mínimo
                return caminoDe
                    (origen,destino,anterior,g);
            }
            visitado[dest]=true;
            procesar.addLast(dest);
        } // if
    }
}
```

Implementación en Java (cont.)

```

    } // for
  } // while
  // el destino no ha sido hallado)
  throw new adts.NoExiste();
} // método

```

Implementación en Java (cont.)

```

private static CaminoSinPesos caminoDe
  (int origen, int destino, int[] anterior, Grafo g)
{
  CaminoSinPesos cam= new CaminoSinPesos(g);
  int v=destino;
  while (anterior[v]!=-1) {
    cam.inserta(v);
    v=anterior[v];
  }
  cam.inserta(origen);
  return cam;
} // método
} // clase

```

4.6. Cálculo de caminos mínimos con pesos positivos

En este caso tendremos en cuenta el coste de recorrer cada arista

- el camino mínimo ya no es necesariamente el de menor longitud
- al ser los pesos positivos, no es posible rebajar el coste recorriendo aristas adicionales a las ya recorridas en el camino

Resolveremos el problema con el algoritmo de *Dijkstra*

Es como en el caso anterior

- recorreremos los vértices adyacentes, en el orden de camino mínimo
- pero cuando hemos visitado un vértice, si encontramos un camino de menor coste hay que anotar el nuevo camino

Cálculo de caminos mínimos con pesos positivos (cont.)

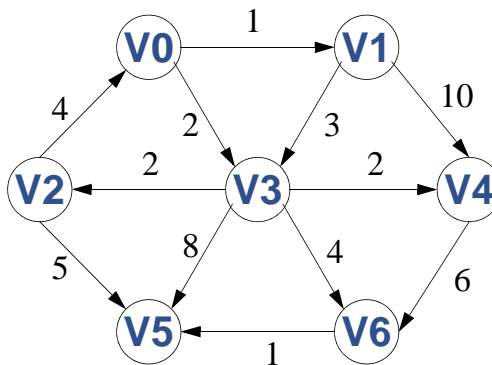
En este algoritmo ya no se garantiza que la primera vez que se visita un vértice se ha hecho por el camino mínimo

- debemos procesar todos los vértices antes de dar la solución

Por ejemplo, el camino

$\{V2, V0, V1, V4\}$ tiene un peso de 15

- pero el camino $\{V2, V0, V3, V4\}$, que se encuentra posteriormente, tiene un peso de 8



Organización de los datos

- Necesitamos guardar en cada vértice el peso del camino mínimo encontrado hasta el momento (D ; infinito si no se ha visitado)
- Igual que antes, para recordar el camino, podemos anotar en cada vértice cuál es el vértice desde el que procede el camino
- Necesitamos saber para cada vértice si ha sido procesado o no
- Necesitamos almacenar el conjunto de vértices a procesar
 - deben procesarse en orden de peso del camino mínimo encontrado hasta el momento (D), comenzando por el menor
 - una cola de prioridad, donde la prioridad es D , es eficiente
 - podemos usar una cola de prioridad con posibilidad de modificar la prioridad de un elemento
 - o podemos volver a insertar el elemento, con el nuevo coste, y descartar los elementos ya procesados

Eficiencia

El tamaño de la cola de prioridad puede ser como máximo E (el número de aristas)

Como mucho hay E inserciones y eliminaciones

Las operaciones en la cola de prioridad son $O(\log n)$

La eficiencia total es $O(E \log E)$,

Teniendo en cuenta que $E \leq V^2$ el coste es a $O(E \log V) = O(E \log V)$, siendo V el número de vértices

Implementación del camino mínimo con pesos



Objeto almacenado en la cola de prioridad

```
clase SubCamino
  atributos
    IdVertice dest;
    real peso;
  fatributos

  método constructor (IdVertice dest, real peso)
    this.dest=dest; this.peso=peso;
  fmétodo
  método compareTo(Subcamino otro) retorna entero
    retorna this.peso.compareTo(otro.peso)
  fmétodo
fclase
```

Pseudocódigo del camino mínimo con pesos



```
método caminoMinimoConPesos (Grafo g,
  idVertice origen, destino)
retorna Camino
  // inicializaciones
  ColaPrioridad<SubCamino> procesar:=
    nueva ColaPrioridad vacía;
  idVertice anterior[]:= nuevo array[numVertices]
  real peso[]:= nuevo array[numVertices]=
    infinito para todas sus casillas
  boolean procesado[]:= nuevo array[numVertices]=
    false para todas sus casillas
  procesar.inserta(origen)
  peso[origen]:=0; // para no hacer ciclos
  anterior[origen]:=-1 // para indicar que no tiene
```


Pseudocódigo del camino mínimo con pesos (cont.)



```
mientras procesar no esta vacio hacer
  idVertice v:=procesar.extraer()
  si no procesado[v] entonces
    procesado[v]:=true;
    Lista adj:=g.listaAristas(v)
    para cada arista a de adj hacer
      idVertice dest:=a.destino;
      real p:=a.peso;
      si peso[dest]>peso[v]+p entonces
        anterior[dest]:=v
        peso[dest]:=peso[v]+p;
        procesar.inserta
          (nuevo Subcamino(dest,peso[dest]));
    fsi
  fpara
```

Pseudocódigo del camino mínimo (cont.)



```
fsi
fmientras
si peso[destino]==infinito entonces
  indicar error (el destino no ha sido hallado)
si no
  retorna caminoDe(origen,destino,anterior,peso)
fsi
fmétodo

// CaminoDe es parecido al anterior,
// pero anota también el peso del camino
```

Implementación en Java

```

/**
 * Algoritmo del calculo del camino mínimo con pesos
 */
public static <V> CaminoConPesos caminoMinimoConPesos
    (Grafo<V,Double> g, int origen, int destino)
    throws NoExiste, IdIncorrecto
{
    // inicializaciones
    PriorityQueue<SubCamino> procesar=
        new PriorityQueue<SubCamino>();
    int[] anterior= new int[g.numVertices()];
    double[] peso= new double[g.numVertices()];
    boolean[] procesado= new boolean[g.numVertices()];
    // valor por omisión false para todas sus casillas

```

Implementación en Java (cont.)

```

// dar valor infinito a todas las casillas de peso
for (int i=0; i<peso.length; i++) {
    peso[i]=Double.POSITIVE_INFINITY;
}

procesar.add(new SubCamino(origen,0.0));
peso[origen]=0.0; // para no hacer ciclos
anterior[origen]=-1; // para indicar que no tiene

// bucle para procesar vértices
while (! procesar.isEmpty()) {
    int v=procesar.remove().dest;
    if (! procesado[v]) {
        procesado[v]=true;

```

Implementación en Java (cont.)

```

List<Arista<Double>> adj=g.listaAristas(v);
for (Arista<Double> a: adj) {
    int dest=a.destino();
    double p=a.contenido().doubleValue();
    if (peso[dest]>peso[v]+p) {
        anterior[dest]=v;
        peso[dest]=peso[v]+p;
        procesar.add
            (new SubCamino(dest,peso[dest]));
    }
} // for
} // if
} // while

```

Implementación en Java (cont.)

```

if (Double.isInfinite(peso[destino])) {
    throw new NoExiste();
} else {
    return caminoDe
        (origen,destino,anterior,peso[destino],g);
}
}

```

Implementación en Java (cont.)

```

/**
 * Método para formar el camino con pesos
 */
private static CaminoConPesos caminoDe(int origen,
    int destino, int[] anterior, double peso, Grafo g)
{
    CaminoConPesos cam= new CaminoConPesos(g,peso);
    int v=destino;
    while (anterior[v]!=-1) {
        cam.inserta(v);
        v=anterior[v];
    }
    cam.inserta(origen);
    return cam;
}

```

4.7. Cálculo de caminos mínimos con pesos negativos

Cuando las aristas tienen pesos negativos el algoritmo de *Dijkstra* no funciona correctamente

- una vez que un vértice v ha sido procesado puede existir un camino de vuelta desde otro vértice u con coste negativo
- en ese caso el nuevo camino puede ser de coste menor
- tendríamos que procesar v de nuevo, porque los vértices afectados desde v también variarían

Además, pueden aparecer ciclos de coste negativo

- en ese caso el camino más corto está indefinido
- el problema no tiene solución
- el algoritmo debe detectar este caso

Solución al problema

Usaremos una solución similar a las anteriores

- será bastante menos eficiente
- podemos usar una cola simple, en lugar de una cola de prioridad

Cuando se modifica el camino mínimo de un vértice, deberá procesarse de nuevo

Para detectar ciclos negativos, anotaremos el número de veces que hemos visitado cada vértice

- si se visita más de V (número de vértices) veces, esto indica que hay un ciclo negativo

La eficiencia será $O(E V)$, porque hay que recorrer cada arista un máximo de V veces

Organización de los datos

- Necesitamos guardar en cada vértice el **peso** del camino mínimo encontrado hasta el momento (D ; infinito si no se ha visitado)
- Para recordar el camino anotaremos en cada vértice cuál es el vértice **anterior** en el camino
- Necesitamos saber para cada vértice cuántas veces ha sido **visitado**
- Necesitamos almacenar el conjunto de vértices a **procesar**
 - podemos usar una cola sencilla
- Por eficiencia, podemos anotar si un vértice **está en la cola** o no
 - así, si ya está en la cola no lo volveremos a añadir cuando modificamos su camino mínimo

Pseudocódigo del camino mínimo con pesos negativos



```
método caminoMinimoConPesosNegativos (Grafo g,  
    idVertice origen, destino)  
retorna Camino  
    // inicializaciones  
Cola<IdVertice> procesar:= nueva Cola vacía;  
idVertice anterior[]:= nuevo array[numVertices];  
real peso[]:= nuevo array[numVertices]=  
    infinito para todas sus casillas  
entero visitado[]:= nuevo array[numVertices]=  
    0 para todas sus casillas  
boolean estaEnLaCola[]:= nuevo array[numVertices]=  
    false para todas sus casillas  
procesar.inserta(origen)  
peso[origen]:=0; // para no hacer ciclos  
anterior[origen]:=-1 // para indicar que no tiene
```

Pseudocódigo del camino mínimo con pesos negativos (cont.)



```
mientras procesar no esta vacio hacer  
    idVertice v:=procesar.extraer()  
    estaEnLaCola[v]:=false;  
    Lista adj:=g.listaAristas(v)  
    para cada arista a de adj hacer  
        idVertice dest:=a.destino;  
        real p:=a.peso;  
        si peso[dest]>peso[v]+p entonces  
            visitado[dest]++;  
            si (visitado[dest]>g.numVertices()) entonces  
                lanza CicloNegativo  
        fsi  
        anterior[dest]:=v  
        peso[dest]:=peso[v]+p;
```

Pseudocódigo del camino mínimo con pesos negativos (cont.)



```
    si no estaEnLaCola(dest) entonces
        estaEnLaCola[dest]:=true;
        procesar.inserta(dest);
    fsi
fpara
fmientras
si peso[destino]==infinito entonces
    indicar error (el destino no ha sido hallado)
si no
    retorna caminoDe(origen,destino,anterior,peso)
fsi
fmétodo

// CaminoDe es igual al anterior
```

Implementación en Java



```
/**
 * Algoritmo del camino mínimo con pesos negativos
 */
public static <V> CaminoConPesos
    caminoMinimoConPesosNegativos
    (Grafo<V,Double> g, int origen, int destino)
    throws NoExiste, IdIncorrecto, CicloNegativo
{
    // inicializaciones
    LinkedList<Integer> procesar=
        new LinkedList<Integer>();
    int numVertices=g.numVertices();
    int[] anterior= new int[numVertices];
    double[] peso= new double[numVertices];
```

Implementación en Java (cont.)

```
int[] visitado= new int[numVertices];
// valor por omisión es 0 para todas sus casillas
boolean[] estaEnLaCola= new boolean[numVertices];
// valor por omisión false para todas sus casillas

// dar valor infinito a todas las casillas de peso
for (int i=0; i<peso.length; i++) {
    peso[i]=Double.POSITIVE_INFINITY;
}
procesar.addLast(new Integer(origen));
peso[origen]=0.0; // para no hacer ciclos
anterior[origen]=-1; // para indicar que no tiene
visitado[origen]++;
```

Implementación en Java (cont.)

```
// bucle para procesar vértices
while (! procesar.isEmpty()) {
    int v=procesar.removeFirst().intValue();
    estaEnLaCola[v]=false;
    List<Arista<Double>> adj=g.listaAristas(v);
    for (Arista<Double> a: adj) {
        int dest=a.destino();
        double p=a.contenido().doubleValue();
        if (peso[dest]>peso[v]+p) {
            visitado[dest]++;
            if (visitado[dest]>numVertices) {
                throw new CicloNegativo();
            }
            anterior[dest]=v;
            peso[dest]=peso[v]+p;
        }
    }
}
```



```
        if (!estaEnLaCola[dest]) {
            estaEnLaCola[dest]=true;
            procesar.addLast(new Integer(dest));
        }
    } // for
} // while
if (Double.isInfinite(peso[destino])) {
    throw new NoExiste();
} else {
    return caminoDe
        (origen, destino, anterior, peso[destino], g);
}
}
```

4.8. Cálculo de caminos en grafos acíclicos

Los grafos acíclicos son aquellos que no contienen ciclos

- En este caso el problema del camino mínimo es más sencillo
 - incluso con pesos negativos
- Si seguimos el orden apropiado, una vez que visitamos un vértice para asignarle un camino mínimo, ya no hace falta visitarlo más
 - el orden apropiado para este caso es el orden topológico

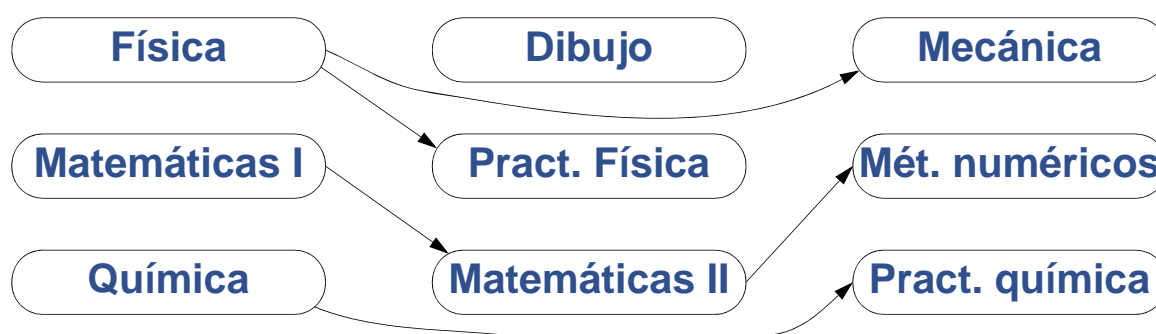
El orden topológico

Un **orden topológico** ordena los vértices del grafo de modo que si hay un camino entre dos vértices u y v , entonces v aparece después de u en la ordenación

- no puede haber un orden topológico si el grafo tiene ciclos
 - entre dos vértices u y v que pertenecen a un ciclo, siempre hay un camino de u a v y de v a u
 - ello hace imposible la ordenación topológica

Ejemplo de orden topológico

El siguiente grafo muestra la estructura de asignaturas "llave" en un plan de estudios



Para matricularse hay que seguir un orden topológico

- por ejemplo: Física, Matemáticas I, Dibujo, Prácticas de Física, Química, Mecánica, Métodos Numéricos, Prácticas de química

Estrategia para hallar un orden topológico

Repetir estos pasos:

- Elegir un vértice que no tenga aristas de entrada y listarlo
- Borrar (imaginariamente) ese vértice y todas sus aristas del grafo

Este algoritmo funciona pues si no hay vértices sin aristas de entrada es porque hay ciclos

Implementación del orden topológico

Para implementar este algoritmo definimos el **grado de entrada** de un vértice como el número de aristas de entrada que tiene

- Calcularemos el grado de entrada de cada vértice
- Cuando "borramos" una arista, lo que hacemos en realidad es restar una unidad al grado de entrada de los vértices destino

Implementación del orden topológico cont.)

Organización de los datos

- Metemos en una cola todos los vértices no procesados de grado de entrada cero
- Sacamos los vértices de la cola uno por uno, los listamos, y "borramos sus aristas"
- Cuando se reduce a cero el grado de entrada de un vértice lo metemos en la cola
- Si la cola se queda vacía sin haber procesado todos los vértices es que hay un ciclo

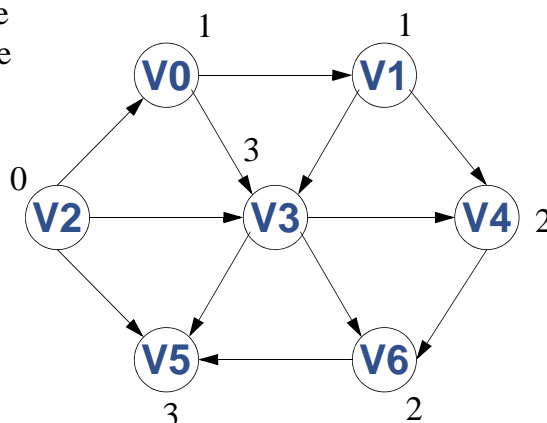
Eficiencia

- El tiempo depende del número de aristas, pues como máximo recorreremos cada una sólo una vez

Ejemplo

Aplicar el algoritmo anterior para calcular un orden topológico en este ejemplo

Se muestra el grado de entrada de cada vértice



Pseudocódigo del orden topológico

```
método calculaGradoEntrada(Grafo g)
retorna entero[]
    entero[] grado:= nuevo array[g.numVertices()]:=
        0 para todas sus casillas
    para v desde 0 hasta g.numVertices()-1 hacer
        Lista<Arista> ady:=g.listaAristas(v);
        para cada Arista a de ady hacer
            grado[a.destino()]++;
        fpara
    fpara
    retorna grado;
fmétodo
```

Pseudocódigo del orden topológico (cont.)

```
método ordenTopologico(Grafo g) lanza HayCiclos
    entero[] grado:= calculaGradoEntrada(g);
    Cola<IdVertice> procesar := nueva Cola vacía
    entero procesados:=0;
    // insertar en la cola vértices de grado cero
    para v desde 0 hasta g.numVertices()-1 hacer
        si grado[v]==0 entonces
            procesar.inserta(v);
        fsi
    fpara
    // bucle principal
    mientras procesar no esta vacío hacer
        procesados++;
        IdVertice v=procesar.extraer();
        listar v;
```

Pseudocódigo del orden topológico (cont.)



```
// procesar las aristas de v
Lista<Arista> ady:=g.listaAristas(v);
para cada Arista a de ady hacer
    idvertice dest=a.destino;
    grado[dest]--;
    si grado[dest]==0 entonces
        procesar.inserta(dest);
    fsi
fpara
fmientras
si procesados!=g.numVertices() entonces
    lanza HayCiclos
fsi
fmétodo
```

Aplicación del orden topológico al cálculo de caminos mínimos



Aplicaremos un algoritmo similar al de *Dijkstra*, pero procesando los vértices en un orden topológico

- cuando procesamos un vértice sabemos que su peso de camino mínimo no va a cambiar
 - por el orden topológico, no hay aristas que lleguen al vértice desde otros vértices no procesados
- los vértices procesados antes de llegar al origen son inalcanzables desde éste
 - por ello, no influyen en la distancia

Organización de los datos

- Necesitamos guardar en cada vértice el peso del camino mínimo encontrado hasta el momento (D ; infinito si no se ha visitado)
- Para recordar el camino, podemos anotar en cada vértice cuál es el vértice desde el que procede el camino
- No necesitamos almacenar el conjunto de vértices a procesar
 - bastará seguir el orden topológico

Eficiencia

- La de calcular el orden topológico + recorrer todas las aristas = $O(E)$
- Igual a la del cálculo de caminos mínimos sin pesos

Pseudocódigo del camino mínimo en grafos acíclicos

```

método caminoMinimoAciclico (Grafo g,
    idVertice origen, destino)
lanza HayCiclos, NoExiste, IdIncorrecto
retorna Camino
    // inicializaciones
    idVertice anterior[]:= nuevo array[numVertices]
    real peso[]:= nuevo array[numVertices]=
        infinito para todas sus casillas
    peso[origen]:=0; // para no hacer ciclos
    anterior[origen]:=-1 // para indicar que no tiene

    entero[] grado:= calculaGradoEntrada(g);
    Cola<IdVertice> procesar := nueva Cola vacía
    entero procesados:=0;
  
```

Pseudocódigo del camino mínimo en grafos acíclicos

```
// insertar en la cola los vértices de grado cero
para v desde 0 hasta g.numVertices()-1 hacer
    si grado[v]==0 entonces
        procesar.inserta(v);
    fsi
fpara
// bucle principal
mientras procesar no esta vacío hacer
    procesados++;
    IdVertice v=procesar.extraer();
    // procesar las aristas de v
    Lista<Arista> ady:=g.listaAristas(v);
    para cada Arista a de ady hacer
        idVertice dest=a.destino;
        grado[dest]--;
```

Pseudocódigo del camino mínimo en grafos acíclicos (cont.)

```
    si grado[dest]==0 entonces
        procesar.inserta(dest);
    fsi

    si peso[v]!=infinito entonces
        real p:=a.peso;
        si peso[dest]>peso[v]+p entonces
            anterior[dest]:=v
            peso[dest]:=peso[v]+p;
        fsi
    fsi

fpara
fmientras
```


Pseudocódigo del camino mínimo en grafos acíclicos (cont.)

```
si procesados!=g.numVertices() entonces
    lanza HayCiclos
fsi
```

```
si peso[destino]==infinito entonces
    lanza NoExiste
si no
    retorna caminoDe(origen,destino,anterior,peso)
fsi
```

fmétodo

```
// CaminoDe es igual al de Dijkstra
```

Implementación en Java

```
/**
 * Calcular el camino mínimo en un grafo acíclico
 */
public static <V> CaminoConPesos caminoMinimoAciclico
    (Grafo<V,Double> g, int origen, int destino)
    throws NoExiste, IdIncorrecto, HayCiclos
{
    int anterior[]= new int[g.numVertices()];
    double[] peso= new double[g.numVertices()];
    // dar valor infinito a todas las casillas de peso
    for (int i=0; i<peso.length; i++) {
        peso[i]=Double.POSITIVE_INFINITY;
    }
    peso[origen]=0.0; // para no hacer ciclos
    anterior[origen]=-1; // para indicar que no tiene
```

Implementación en Java (cont.)

```

int[] grado= calculaGradoEntrada(g);
LinkedList<Integer> procesar =
    new LinkedList<Integer>();
int procesados=0;
// insertar en la cola vértices de grado cero
for (int v=0; v<g.numVertices(); v++) {
    if (grado[v]==0) {
        procesar.addLast(new Integer(v));
    }
}
// bucle principal
while (!procesar.isEmpty()) {
    procesados++;
    int v=procesar.removeFirst().intValue();

```

Implementación en Java (cont.)

```

// procesar las aristas de v
List<Arista<Double>> ady=g.listaAristas(v);
for (Arista<Double> a: ady) {
    int dest=a.destino();
    grado[dest]--;
    if (grado[dest]==0) {
        procesar.addLast(new Integer(dest));
    }
    if (!Double.isInfinite(peso[v])) {
        double p=a.contenido().doubleValue();
        if (peso[dest]>peso[v]+p) {
            anterior[dest]=v;
            peso[dest]=peso[v]+p;
        }
    }
}

```

Implementación en Java (cont.)

```
    }  
  }  
  if (procesados!=g.numVertices()) {  
    throw new HayCiclos();  
  }  
  if (Double.isInfinite(peso[destino])) {  
    throw new NoExiste();  
  } else {  
    return caminoDe  
      (origen,destino,anterior,peso[destino],g);  
  }  
}
```