

# GRAFOS NO DIRIGIDOS

Grafos donde las aristas son pares de vértices no ordenados

Vamos a ver cómo podemos representarlos en memoria de tal forma que podemos implementar de manera eficiente algoritmos sobre grafos.



UNIVERSIDAD  
**COMPLUTENSE**  
MADRID

**ALBERTO VERDEJO**

# Grafos como modelo abstracto

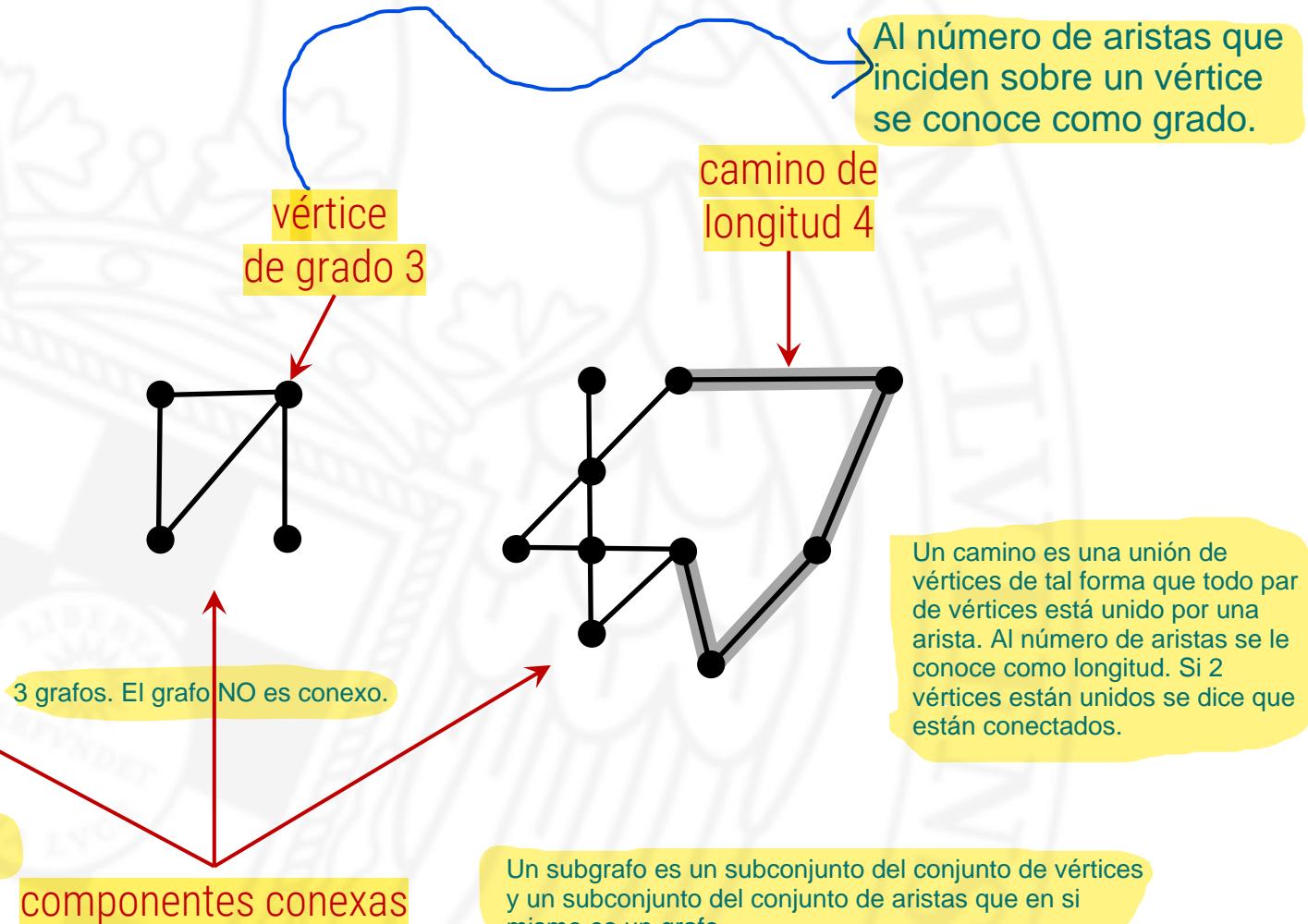
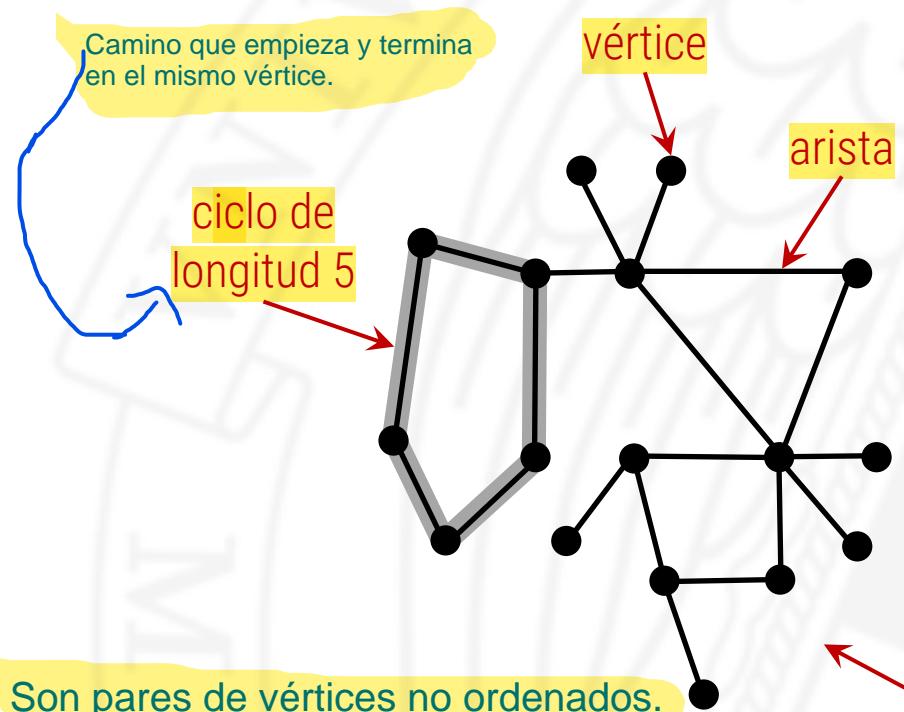
- ▶ Los grafos sirven para representar elementos y conexiones uno a uno entre ellos.

aplicación	elemento	conexión
mapa	intersección	calle, carretera
internet	subred clase C	cable de red
web	página	enlace
red social	persona	amistad
juego	estado del tablero	movimiento legal
circuito	puerta lógica, transistor	cable
red de metro	estación	vía

Ejemplos de dónde podemos utilizar GRAFOS

# Grafos no dirigidos: terminología

- ▶ Un grafo es un conjunto de **vértices** y un conjunto de **aristas** que conectan pares de vértices.



# Problemas sobre grafos

Dados dos vértices decidir si les conecta o no

EN ESTE CURSO VEREMOS CÓMO  
RESOLVER MUCHOS DE ESTOS  
PROBLEMAS..

problema	descripción
camino $s - t$	¿Existe un camino entre $s$ y $t$ ?
camino más corto $s - t$	¿Cuál es el camino más corto (menos aristas) entre $s$ y $t$ ?
grafo conexo	¿Existe un camino entre todo par de vértices? <span style="background-color: yellow; padding: 2px;">Si un grafo es conexo y acíclico se le denomina árbol libre.</span>
ciclo	¿Existe un ciclo en el grafo?
ciclo euleriano	¿Existe un ciclo que utiliza cada arista del grafo exactamente una vez?
ciclo hamiltoniano	¿Existe un ciclo que pasa por cada vértice del grafo exactamente una vez?
grafo bipartito	¿Se pueden repartir los vértices en dos conjuntos de tal forma que las aristas siempre conecten vértices en conjuntos distintos?
grafo planar	¿Puede dibujarse el grafo en un plano sin que haya aristas que se crucen?
grafos isomorfos	¿Existe un isomorfismo entre los grafos?

Qué problemas nos suelen pedir de grafos?

Si existe una biyección entre los vértices que asocia cada vértice de un grafo.

Interesante si el grafo representa un circuito que va a ser construido sobre una placa, no queremos que los cables se crucen

# Representación de un grafo

- En general, los nombres de los vértices no son importantes, pero hay que distinguirlos. Los numeramos de 0 a  $V - 1$ .

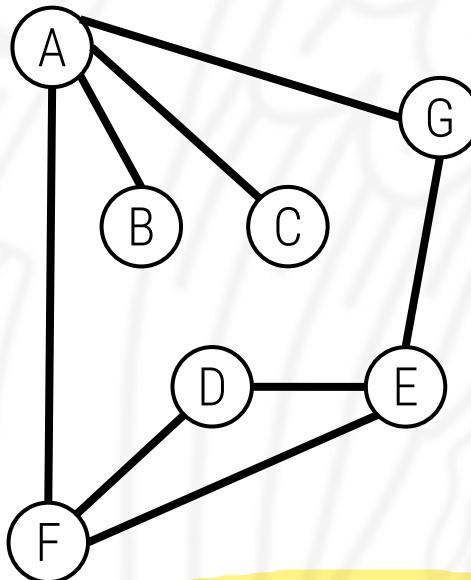
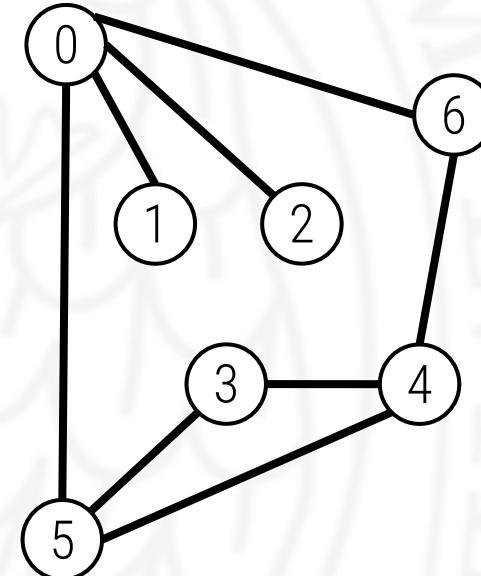


tabla de  
símbolos

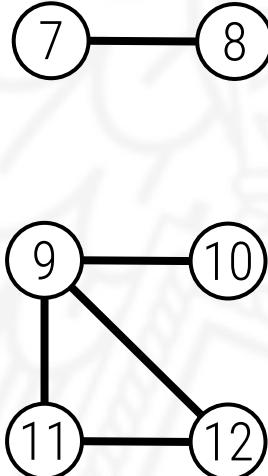
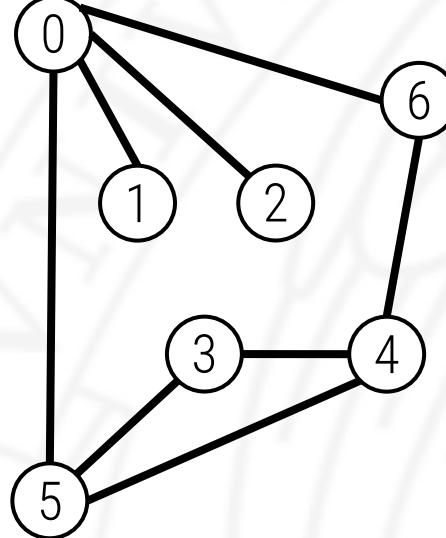


Si el grafo tiene  $V$  vértices lo vamos a numerar desde el 0 hasta el  $V-1$ .

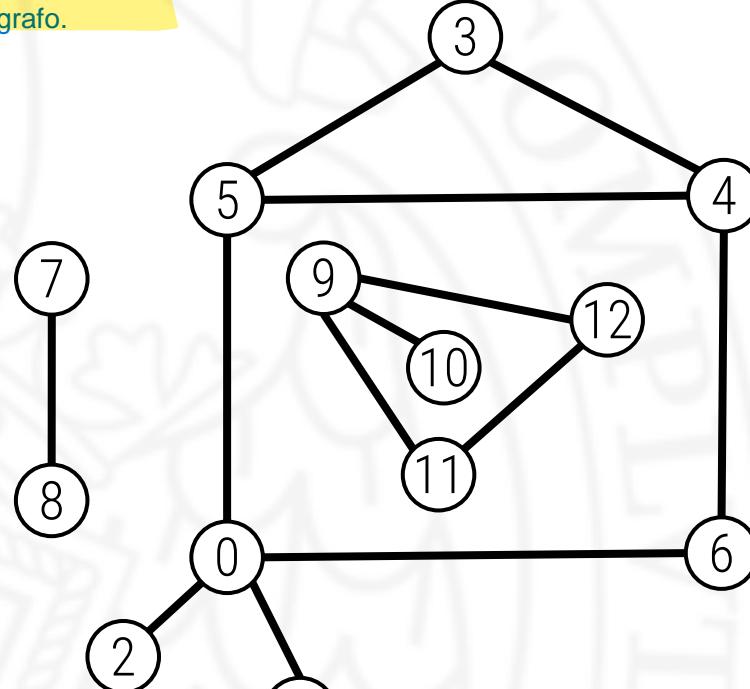
Si el nombre de los vértices es importante, podemos transformar los nombres en números y viceversa.

# Representación de un grafo

- ▶ Un dibujo del grafo nos da intuición sobre su estructura, pero a veces confunde.



Dos dibujos del mismo grafo.

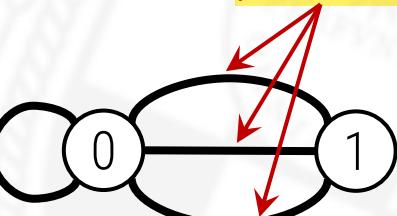


aristas  
paralelas

Si un grafo permite aristas paralelas se llama multigrafos.

- ▶ Anomalías:

autoarista →



Un vértice a si mismo

Estas cosas NO las vamos a permitir.

Aunque la mayoría de algoritmos funcionarían en presencia de estas anomalías, nuestros algoritmos funcionan sin estas anomalías.

# TAD de los grafos

El TAD de los grafos cuenta con las siguientes operaciones:

- ▶ crear un grafo vacío, `Grafo(int v)`  
Tiene un número fijo de vértices que recibe en la constructora.
- ▶ añadir una arista, `void ponArista(int v, int w)`  
Une dos vértices distintos v y w.
- ▶ consultar los adyacentes a un vértice, `Adys ady(int v) const`
- ▶ consultar el número de vértices, `int V() const`  
Devuelve los vértices adyacentes al vértice v
- ▶ consultar el número de aristas, `int A() const`  
Es como una operación size() pero de vértices .  
Es la operación size() pero para el número de aristas.

# Procesamiento típico de un grafo

```
int grado(Grafo const& g, int v) {  
    int grado = 0;  
    for (int w : g.ady(v))  
        ++grado;  
    return grado;  
}
```

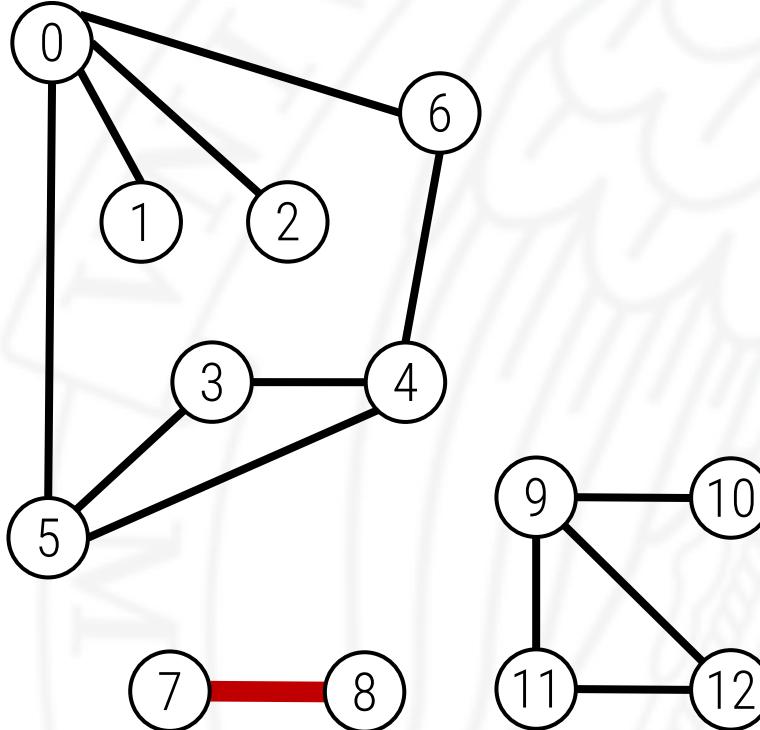
Cuenta el número de aristas de un grafo.

```
int aristas(Grafo const& g) {  
    int aristas = 0;  
    for (int v = 0; v < g.V(); ++v)  
        aristas += grado(g, v);  
    return aristas / 2;  
}
```

Para cada vértice calcula el número de aristas que lo unen.

# Matriz de adyacencia

- Matriz de booleanos  $V \times V$



Representamos el grafo para ahora poder ir definiendo los costes de las dos funciones de arriba.

En la matriz aparecen como un 0 los vértices que no se unen entre sí y con un 1 los que si se unen.

dos entradas  
por arista

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0
4	0	0	0	1	0	1	1	0	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	0	0	0	0
6	1	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	0	1	0	0	0
11	0	0	0	0	0	0	0	0	0	1	0	0	1
12	0	0	0	0	0	0	0	0	0	1	0	1	0

Matriz de booleanos. Decimos si va de i hasta j, y por tanto de j hasta i marcándolo como un 1 en la matriz,

Esta representación NO es buena para conocer los adyacentes de un vértice.

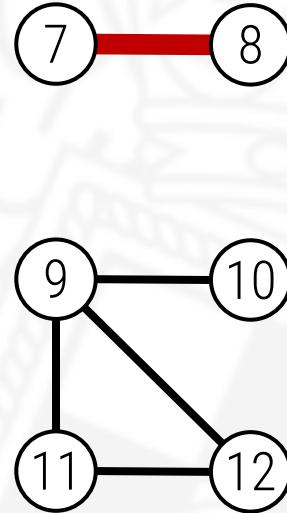
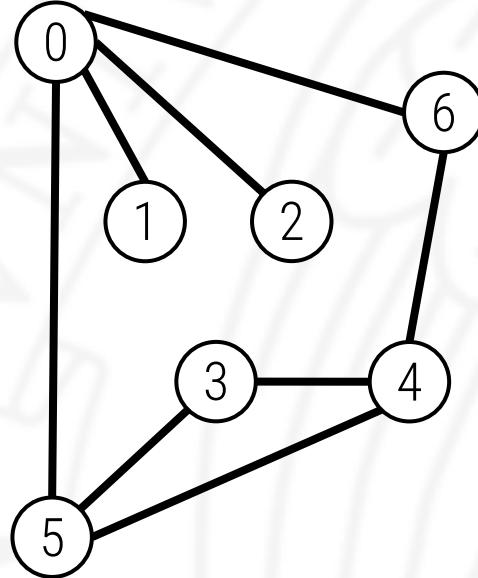
# Matriz de adyacencia

```
int grado(Grafo const& g, int v) {  
    int grado = 0;  
    for (int w : g.ady(v)) // O(V)  
        ++grado;  
    return grado;  
}  
  
int aristas(Grafo const& g) { // O(V2)  
    int aristas = 0;  
    for (int v = 0; v < g.V(); ++v)  
        aristas += grado(g, v);  
    return aristas / 2;  
}
```

Funciones demasiado costosas.

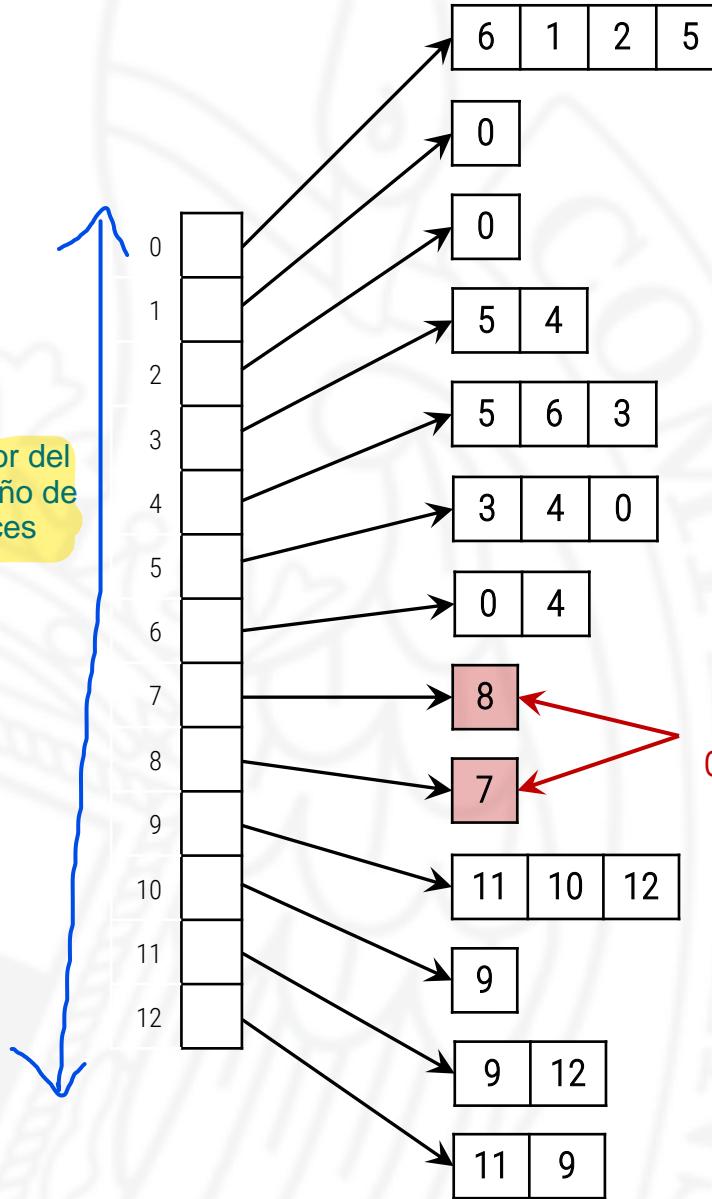
	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0
4	0	0	0	1	0	1	1	0	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	0	0	0	0
6	1	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	0	1	0	0	0
11	0	0	0	0	0	0	0	0	1	0	0	0	1
12	0	0	0	0	0	0	0	0	0	1	0	1	0

# Listas de adyacentes



Vector del tamaño de vértices

En cada posición tenemos el conjunto de vértices adyacentes del vértice índice.



Sobre estas necesitaremos bien recorrer la lista o bien añadir un elemento al final de la lista.

representaciones de la misma arista

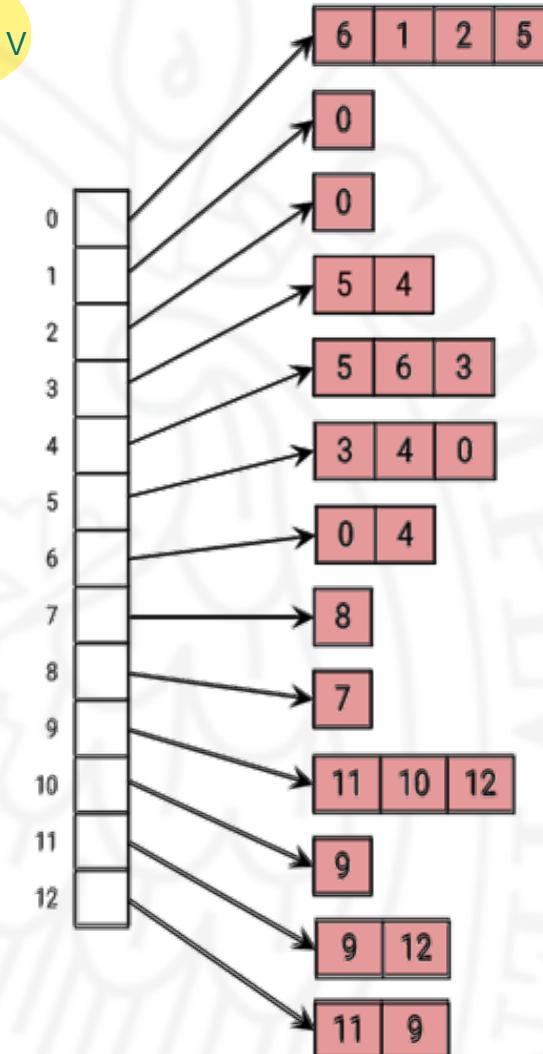
# Listas de adyacentes

```
int grado(Grafo const& g, int v) {  
    int grado = 0;  
    for (int w : g.ady(v)) // 0(grado(v))  
        ++grado;  
    return grado;  
}
```

```
int aristas(Grafo const& g) { // 0(V + A)  
    int aristas = 0;  
    for (int v = 0; v < g.V(); ++v)  
        aristas += grado(g, v);  
    return aristas / 2;  
}
```

En el caso peor es V

Tantas vueltas como el grado del vértice.



# Representación elegida

- En la práctica: listas de adyacentes (algoritmos basados en recorrer los adyacentes a un vértice, los grafos suelen ser dispersos).

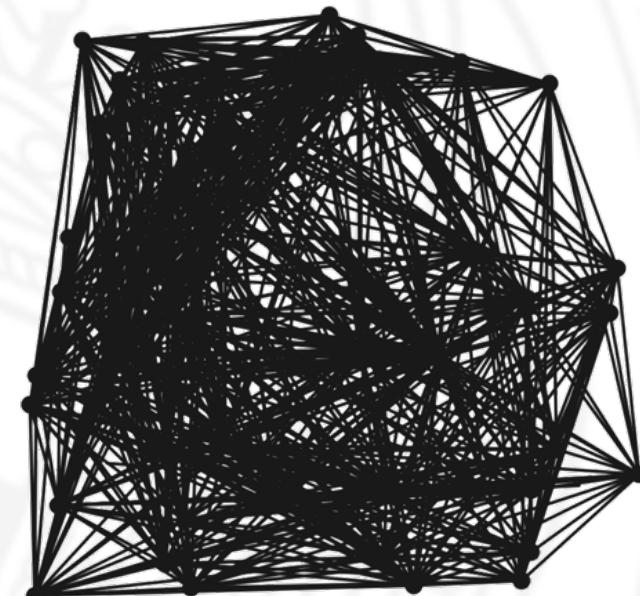
disperso ( $A = 200$ )



Un grafo disperso es un grafo que tiene pocas aristas comparadas con las que podría tener.

dos grafos ( $V = 50$ )

denso ( $A = 1000$ )



Tiene 1000 aristas, pero podría tener  $V^*(V-1)/2$  aristas. Esto sería que todos los vértices se relacionan con todos menos con ellos mismos entre 2 porque se repite.

# Representación elegida

- En la práctica: listas de adyacentes (algoritmos basados en recorrer los adyacentes a un vértice, los grafos suelen ser dispersos).

representación	espacio	añadir arista $v - w$	comprobar si $v$ y $w$ son adyacentes	recorrer los vértices adyacentes a $v$	calcular el grado
matriz de adyacencia	$V^2$	1	1	$V$	
listas de adyacentes	$V + A$	1	<u>grado(<math>v</math>)</u>	<u>grado(<math>v</math>)</u>	$O(V)$
conjuntos de adyacentes	$V + A$	$\log V$	$\log V$	<u>grado(<math>v</math>)</u>	
lista de aristas	<u><math>A</math></u>	1	<u><math>A</math></u>	<u><math>A</math></u>	

Nuestra representación es esta. Esta es la que vamos a utilizar

# Implementación

```
using Adys = std::vector<int>; // lista de adyacentes a un vértice
```

```
class Grafo {  
private:  
    int _V; // número de vértices  
    int _A; // número de aristas  
    std::vector<Adys> _ady; // vector de listas de adyacentes
```

```
public:
```

```
Grafo(int V) : _V(V), _A(0), _ady(_V) {}
```

```
int V() const { return _V; }
```

```
int A() const { return _A; }
```

Es como si fuera un vector de vectores, Empieza siendo vacía porque al principio NO hay adyacencias en el grafo.

# Implementación



```
void ponArista(int v, int w) {  
    if (v < 0 || v >= _V || w < 0 || w >= _V) Comprobamos si los vértices son correctos  
        throw std::domain_error("Vertice inexistente");  
    ++_A; aumentamos el número de aristas  
    _ady[v].push_back(w); ponemos que v tenga como adyacente a w.  
    _ady[w].push_back(v); ponemos que w tenga como adyacente a v.  
}  
}
```

```
Adys const& ady(int v) const { Para consultar los adyacentes devolvemos el vector adyacente.  
    if (v < 0 || v >= _V)  
        throw std::domain_error("Vertice inexistente");  
    return _ady[v]; //O(1)  
}  
};
```

# Patrón de diseño para el procesamiento de grafos

- ▶ Objetivo: separar la resolución de un problema de la representación del grafo.
- ▶ Para cada problema sobre grafos que resolvamos crearemos una clase específica, **Problema**.
- ▶ Generalmente, el constructor realizará cierto trabajo sobre el grafo y creará estructuras para contestar eficientemente a las preguntas del problema.
- ▶ El usuario creará un grafo, después creará un objeto de la clase **Problema** pasándole el grafo como argumento a la constructora, y por último utilizará los métodos de consulta de esta clase para averiguar propiedades del grafo.

# Ejemplo

- ▶ Dado un grafo y un vértice *origen s*, determinar con qué otros vértices está conectado *s*.

Hacer lo mismo para otros problemas.

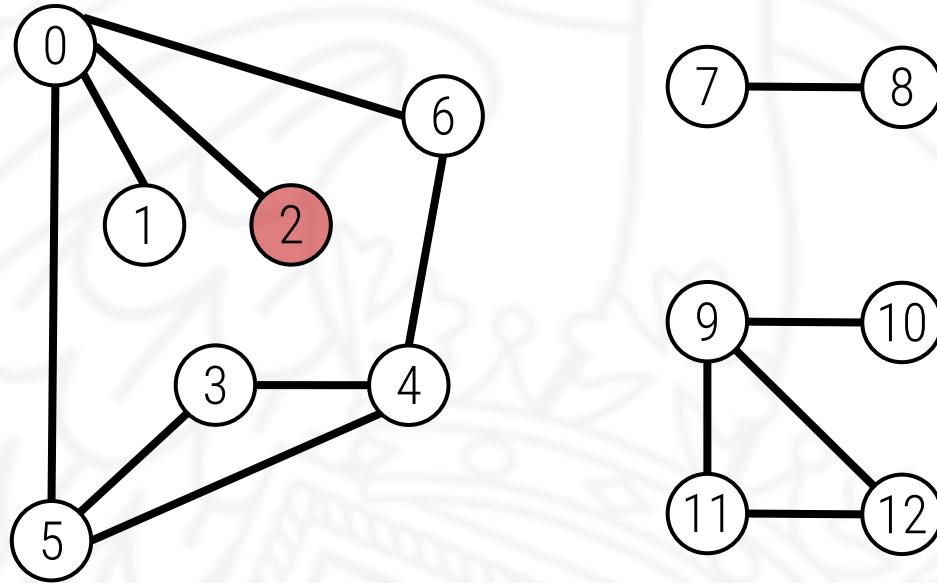
```
class Conexion {  
public:  
    Conexion(Grafo const& g, int s); // busca vértices conectados a s  
    bool conectado(int v) const; // ¿está v conectado a s?  
    int cuantos() const; // ¿cuántos vértices están conectados a s?  
};
```

Contractora recibe el grafo y el vértice

# Ejemplo

```
void resuelve(Grafo const& g, int s) {  
    Conexion conex(g,s);  
    cout << "Vértices conectados a " << s << ":";  
    for (int v = 0; v < g.V(); ++v) {  
        if (v != s && conex.conectado(v))  
            cout << ' ' << v;  
    }  
    cout << '\n';  
  
    if (conex.cuantos() != g.V()) cout << "no ";  
    cout << "es conexo\n";  
}
```

# Ejemplo



Vértices conectados a 2: 0 1 3 4 5 6  
no es conexo