GRAFOS NO DIRIGIDOS

UNIVERSIDAD COMPLUTENSE MADRID

ALBERTO VERDEJO

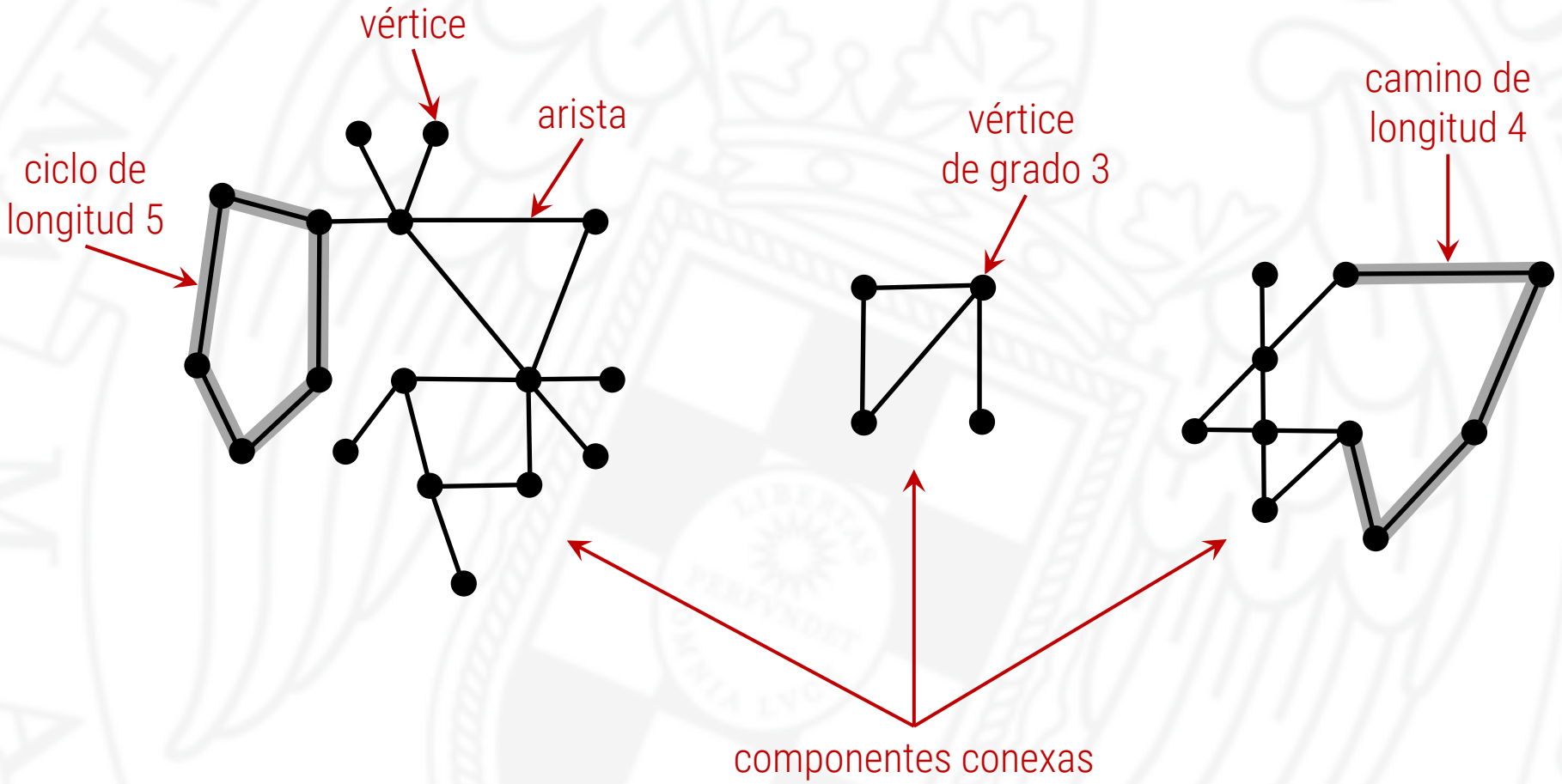
Grafos como modelo abstracto

- 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

Grafos no dirigidos: terminología

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

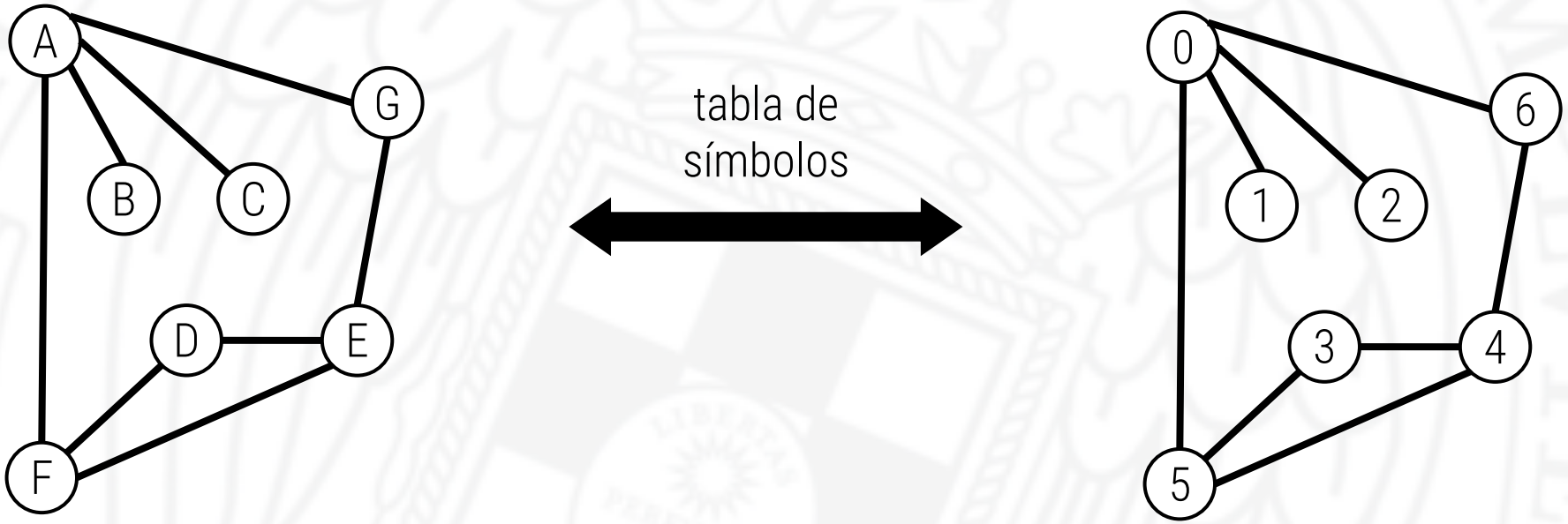


Problemas sobre grafos

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?
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?

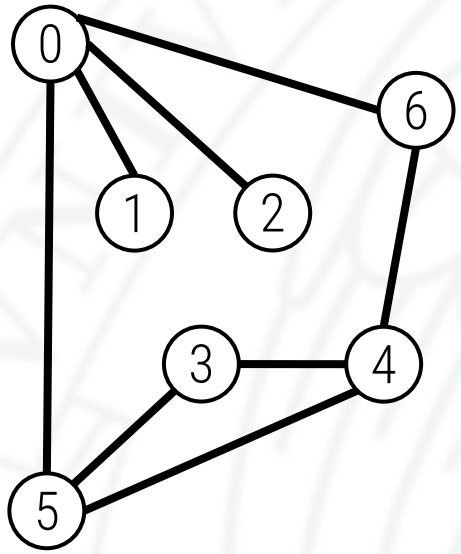
Representación de un grafo

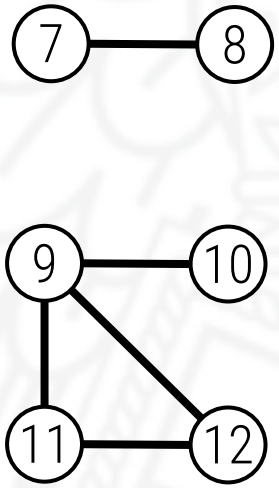
- En general, los nombres de los vértices no son importantes, pero hay que distinguirlos. Los numeramos de 0 a $V - 1$.

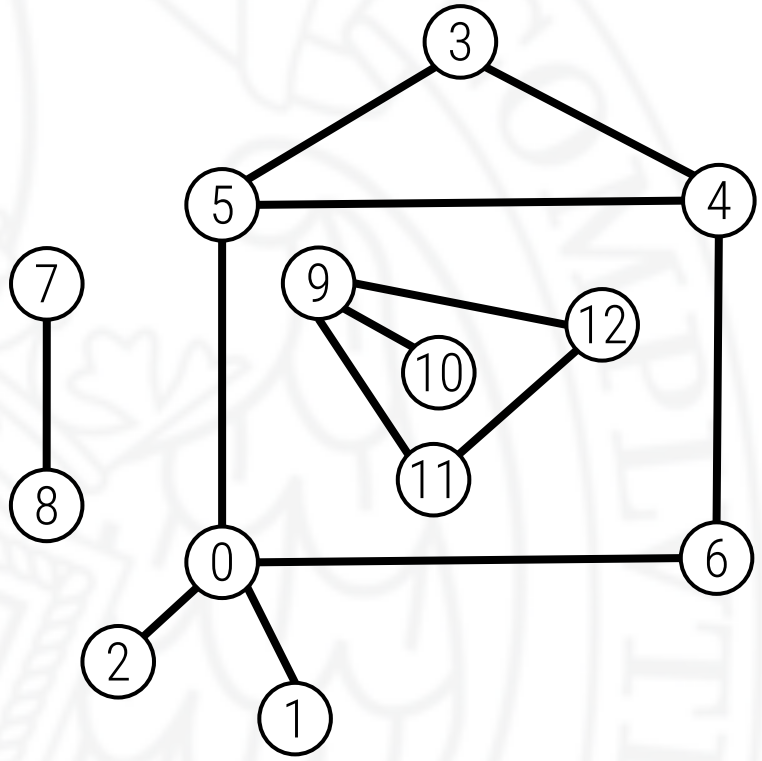


Representación de un grafo

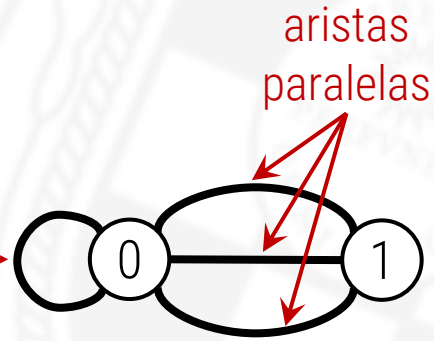
- Un dibujo del grafo nos da intuición sobre su estructura, pero a veces confunde.







- Anomalías: autoarista →



TAD de los grafos

El TAD de los grafos cuenta con las siguientes operaciones:

- crear un grafo vacío, `Grafo(int V)`
- añadir una arista, `void ponArista(int v, int w)`
- consultar los adyacentes a un vértice, `Adys ady(int v) const`
- consultar el número de vértices, `int V() const`
- consultar el número de aristas, `int A() const`

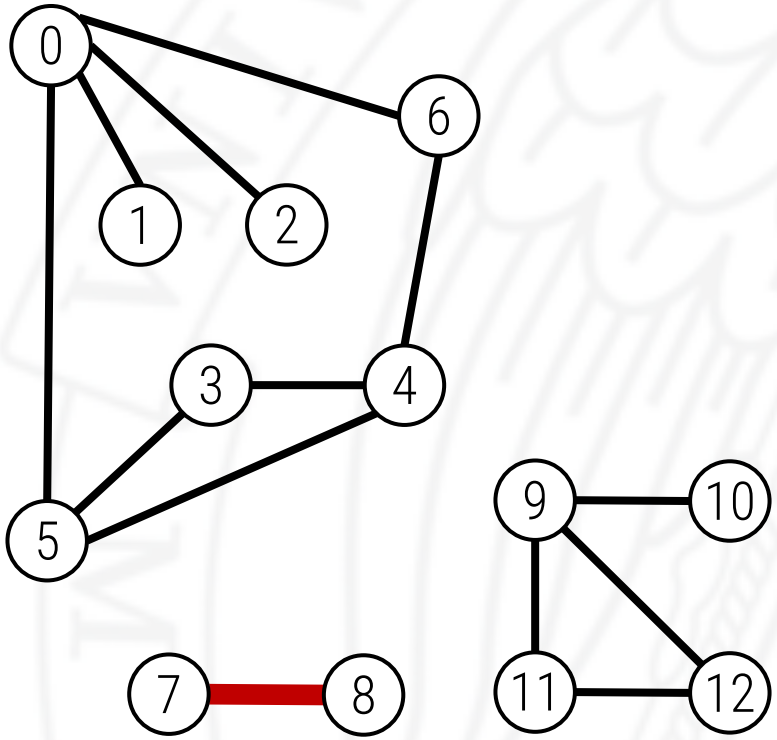
Procesamiento típico de un grafo

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

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

Matriz de adyacencia

- Matriz de booleanos $V \times V$



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

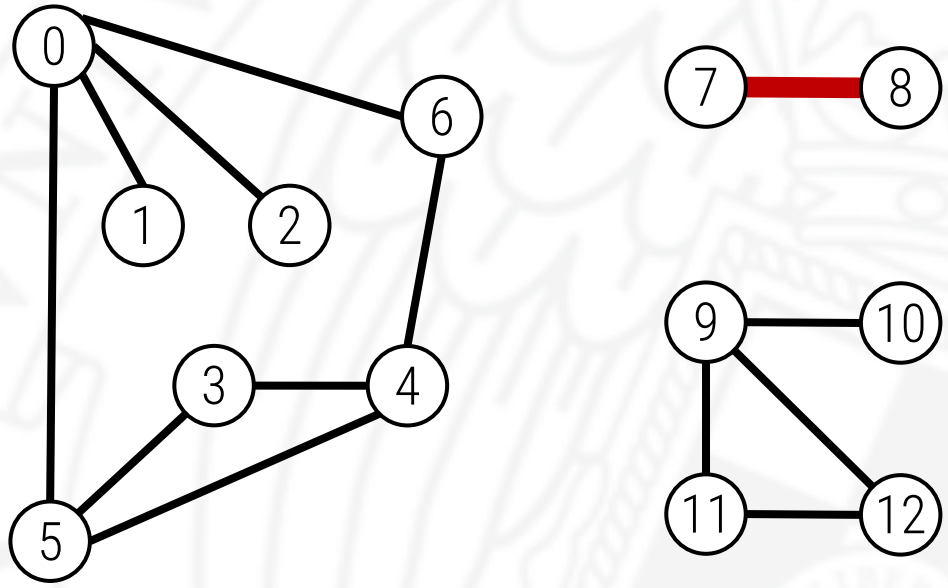
Matriz de adyacencia

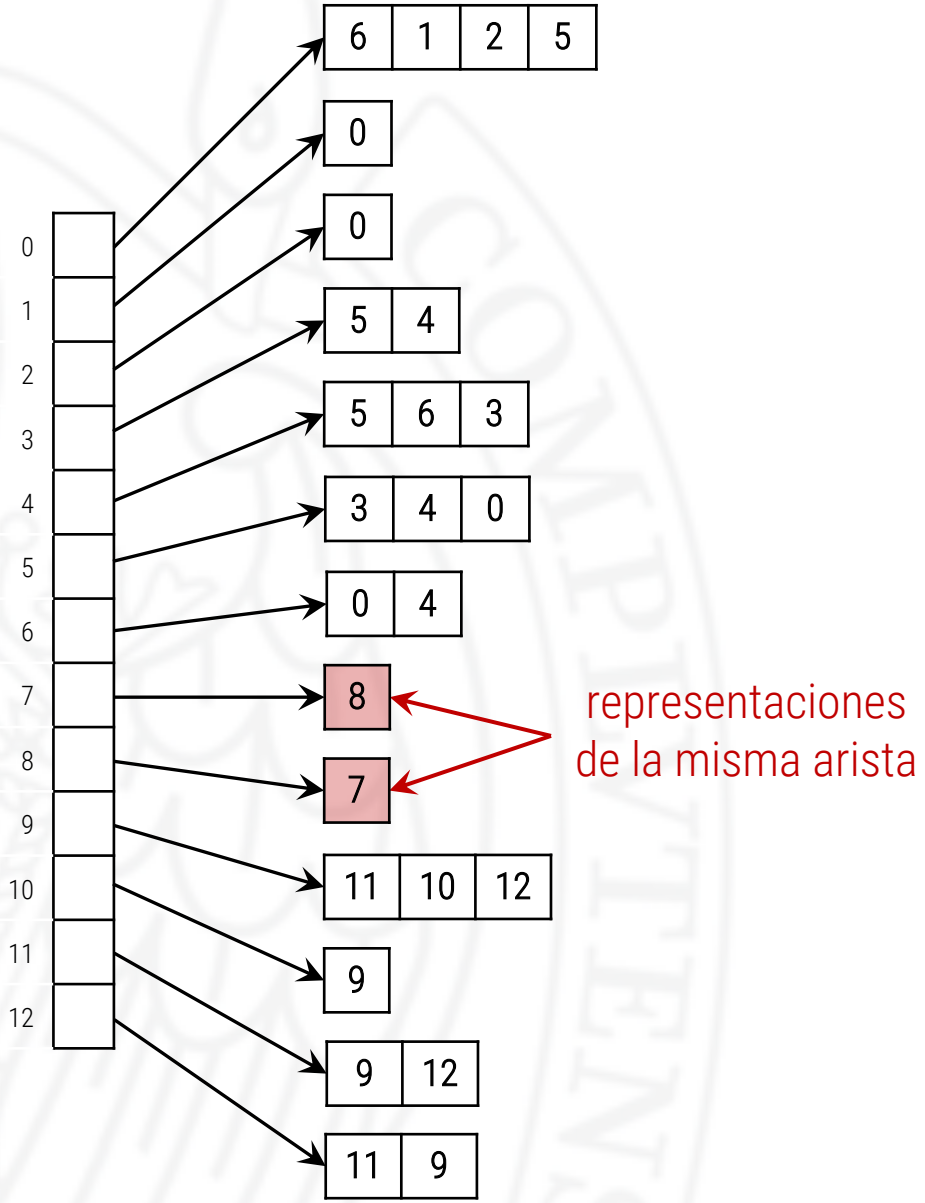
```
int grado(Grafo const& g, int v) {
    int grado = 0;
    for (int w : g.ady(v)) // O(V)
        ++grado;
    return grado;
}

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;
}
```

	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

Listas de adyacentes

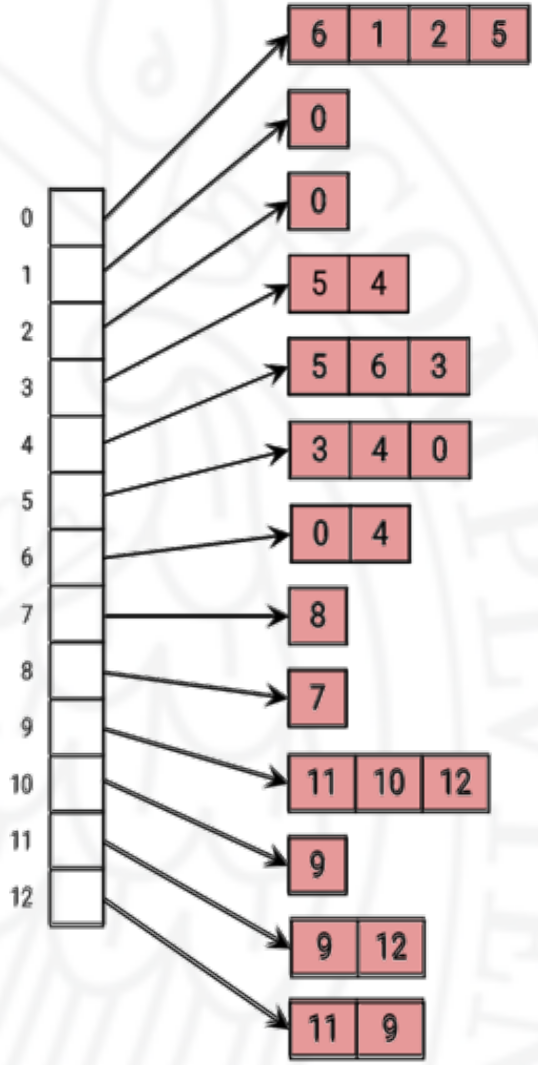




Listas de adyacentes

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

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

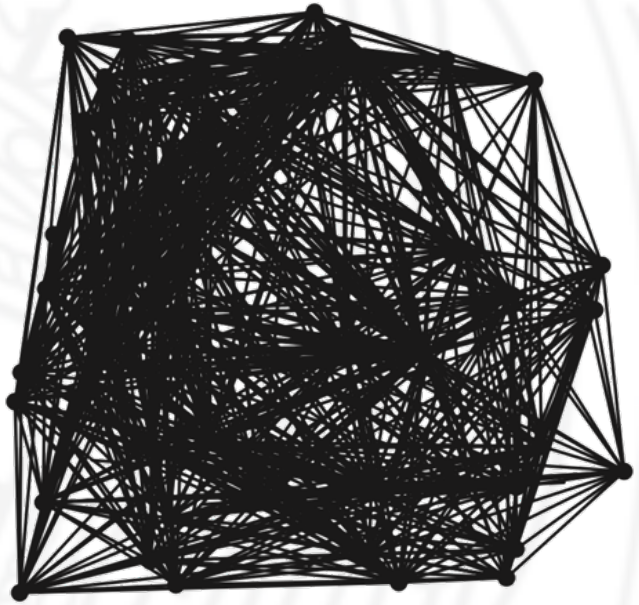


Representación elegida

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

disperso (A = 200)

denso (A = 1000)



dos grafos (V = 50)

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
matriz de adyacencia	V^2	1	1	V
listas de adyacentes	$V + A$	1	grado(v)	grado(v)
conjuntos de adyacentes	$V + A$	log V	log V	grado(v)
lista de aristas	A	1	A	A

Implementación

Grafo.h

```
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; }
```

Implementación

Grafo.h

```
    void ponArista(int v, int w) {
        if (v < 0 || v >= _V || w < 0 || w >= _V)
            throw std::domain_error("Vertice inexistente");
        ++_A;
        _ady[v].push_back(w);
        _ady[w].push_back(v);
    }

    Adys const& ady(int v) const {
        if (v < 0 || v >= _V)
            throw std::domain_error("Vertice inexistente");
        return _ady[v];
    }
};
```

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*.

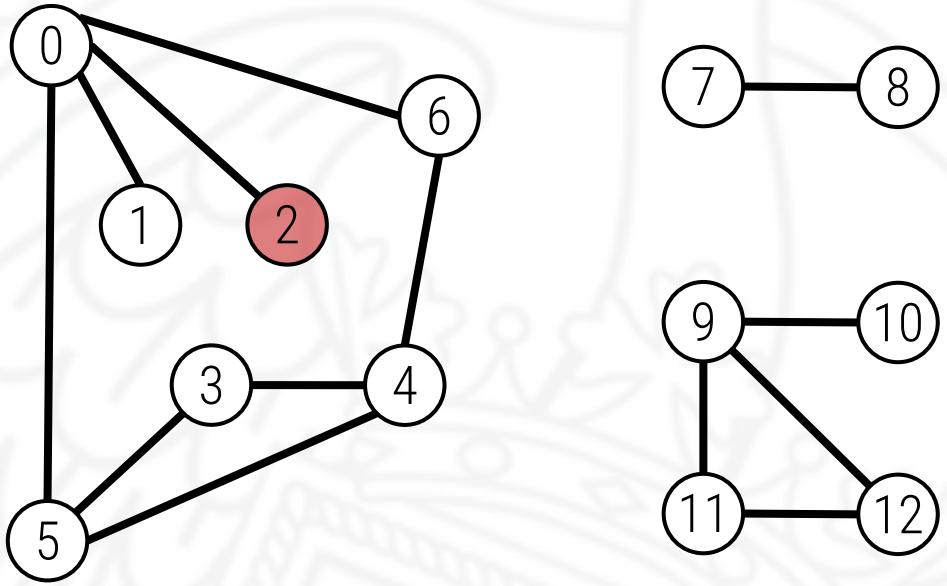
```
class Conexion {
public:
    Conexion(Grafo const& g, int s); // busca vértices conectados a s
    bool conectado(int v) const;     // ¿está v conectado a s?
    int cuantos() const;     // ¿cuántos vértices están conectados a s?
};
```

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
```