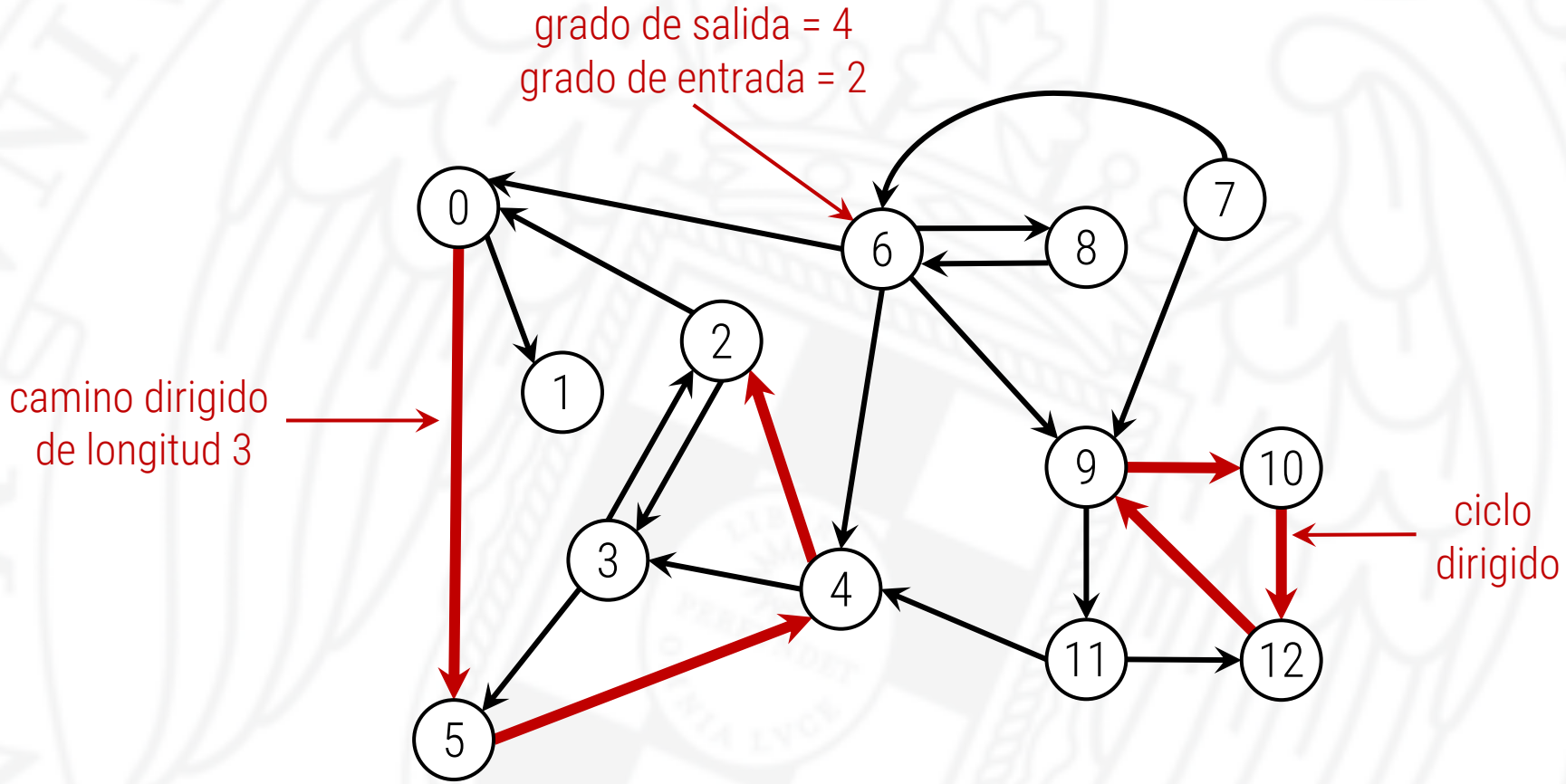
GRAFOS DIRIGIDOS

UNIVERSIDAD
COMPLUTENSE
MADRID

ALBERTO VERDEJO

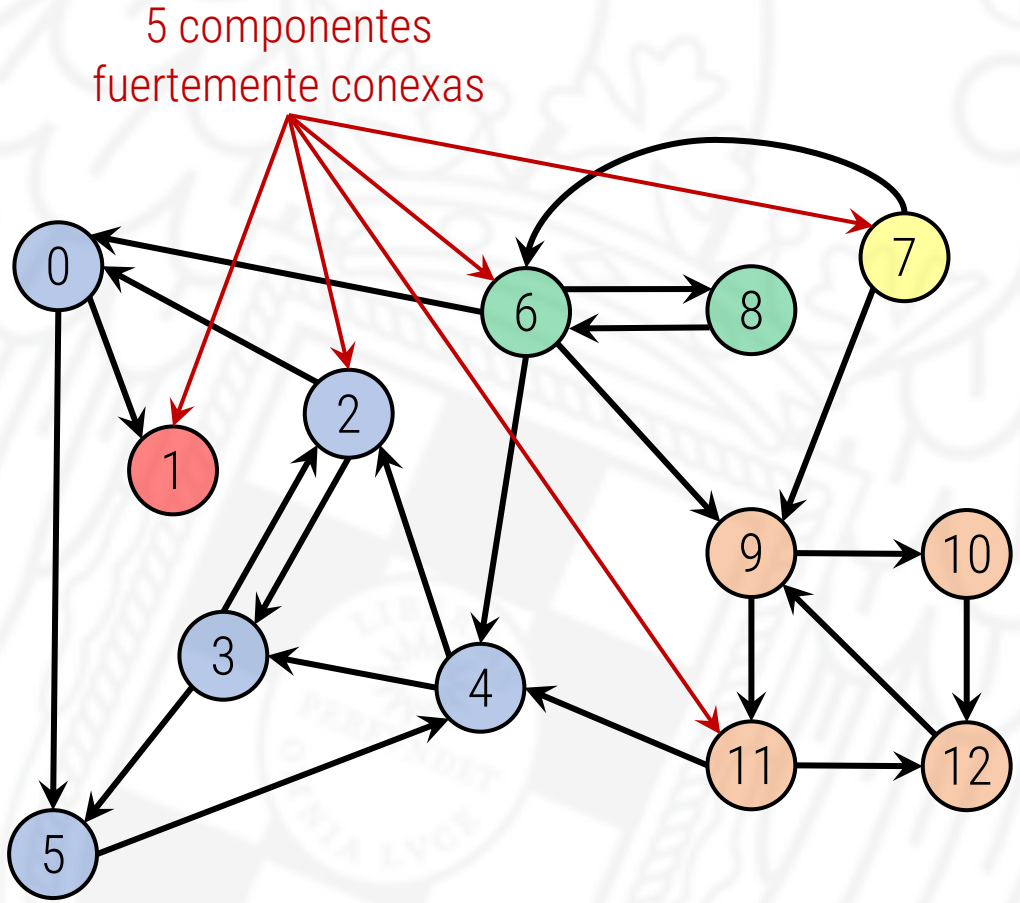
Grafos dirigidos

- También digrafo. Un conjunto de vértices y un conjunto de aristas dirigidas (pares ordenados de vértices).



Grafos dirigidos

- También digrafo. Un conjunto de vértices y un conjunto de aristas dirigidas (pares ordenados de vértices).



Aplicaciones de grafos dirigidos

aplicación	elemento	conexión
mapa	intersección	calle de sentido único
planificación	tarea	precedencia
web	página	enlace
referencias	artículo	cita
juego	estado del tablero	movimiento legal
memoria dinámica	objeto	puntero
orientación a objetos	clase	herencia

Problemas sobre grafos dirigidos

problema	descripción
camino $s \longrightarrow t$	¿Existe un camino dirigido desde s hasta t?
camino más corto $s \longrightarrow t$	¿Cuál es el camino dirigido más corto (menos aristas) desde s hasta t?
ciclo dirigido	¿Existe un ciclo dirigido en el grafo?
grafo fuertemente conexo	¿Existe un camino dirigido entre todo par de vértices?
ordenación topológica	¿Pueden los vértices ordenarse de forma que todas las aristas apunten en el mismo sentido?
cierre transitivo	¿Para qué pares de vértices v y w existe un camino $v \longrightarrow w$?
PageRank	¿Cuál es la importancia de una página web?

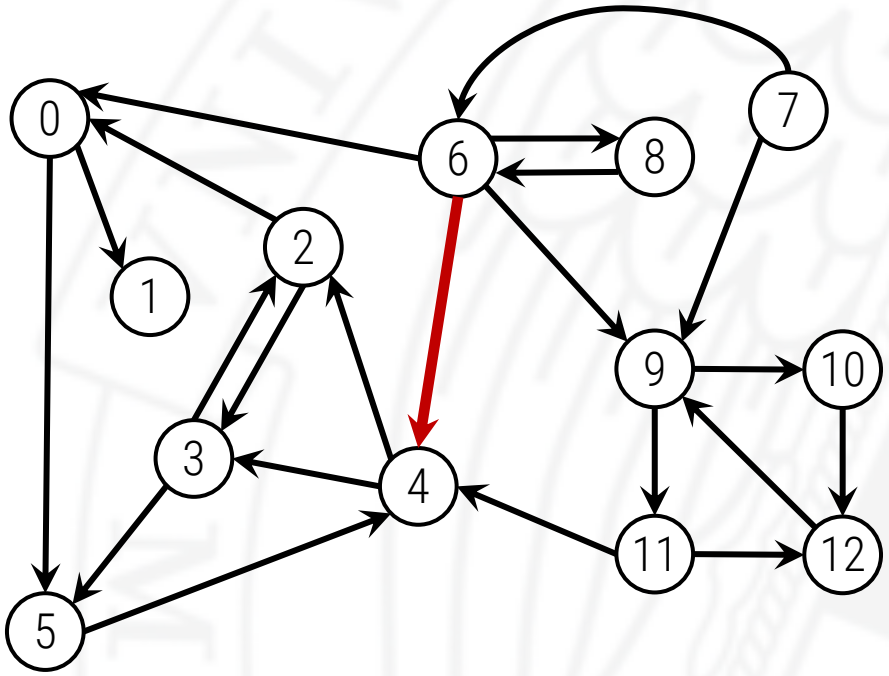
TAD de los grafos dirigidos

El TAD de los grafos dirigidos cuenta con las siguientes operaciones:

- crear un grafo vacío, `Digrafo(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`
- calcular el grafo inverso, `Digrafo inverso() const`

Matriz de adyacencia

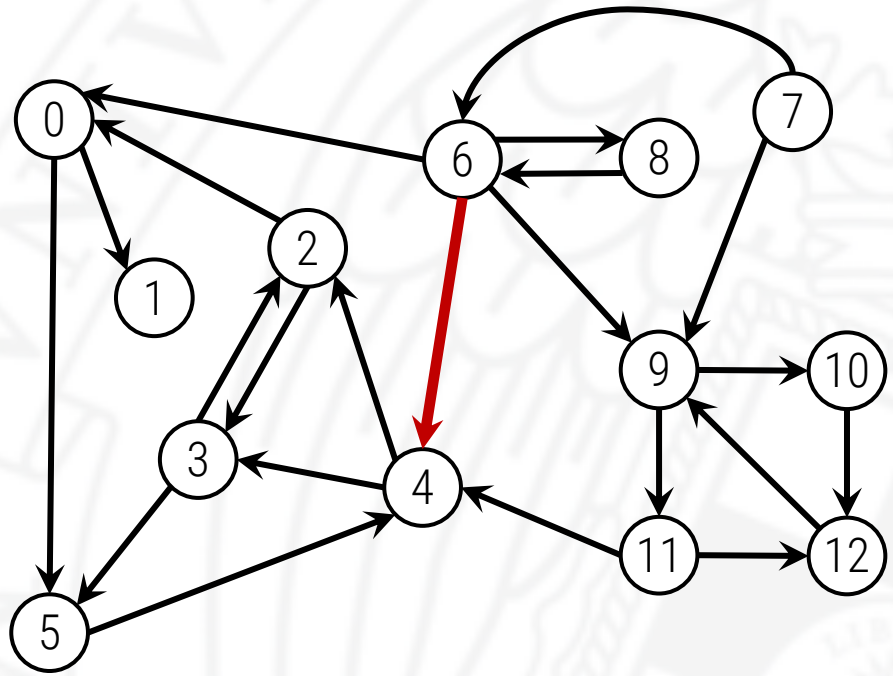
- Matriz de booleanos $V \times V$



una entrada por arista

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

Listas de adyacentes



Representación elegida

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

representación	espacio	añadir arista $v \rightarrow w$	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-sal(v)	grado-sal(v)
lista de aristas	A	1	A	A

Implementación

Digrafo.h

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

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

public:

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

   int V() const { return _V; }

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

Implementación

Digrafo.h

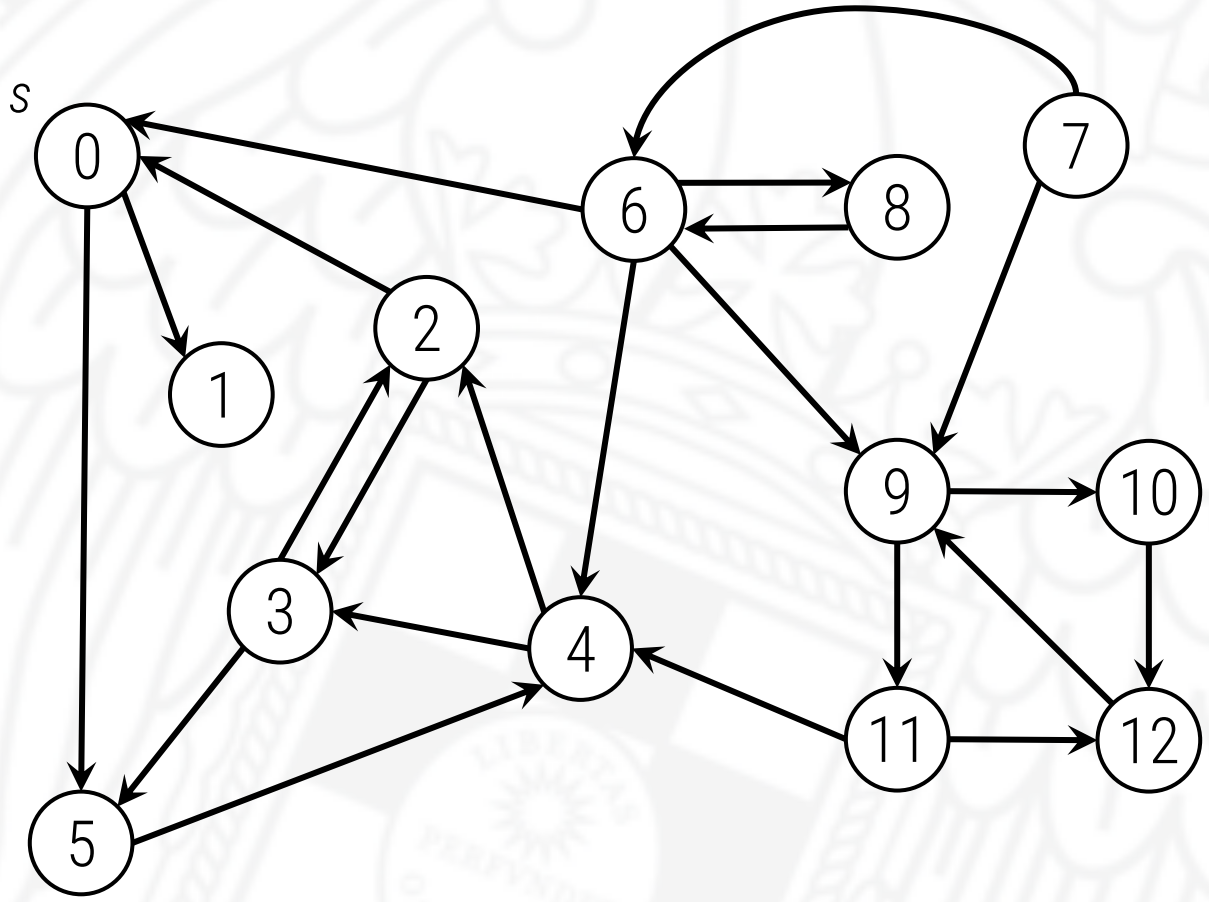
```
void ponArista(int v, int w) {
    if (v < 0 || v >= _V || w < 0 || w >= _V)
        throw std::domain_error("Vertice inexistente");
    ++_A;
    _ady[v].push_back(w);

}

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

Implementación

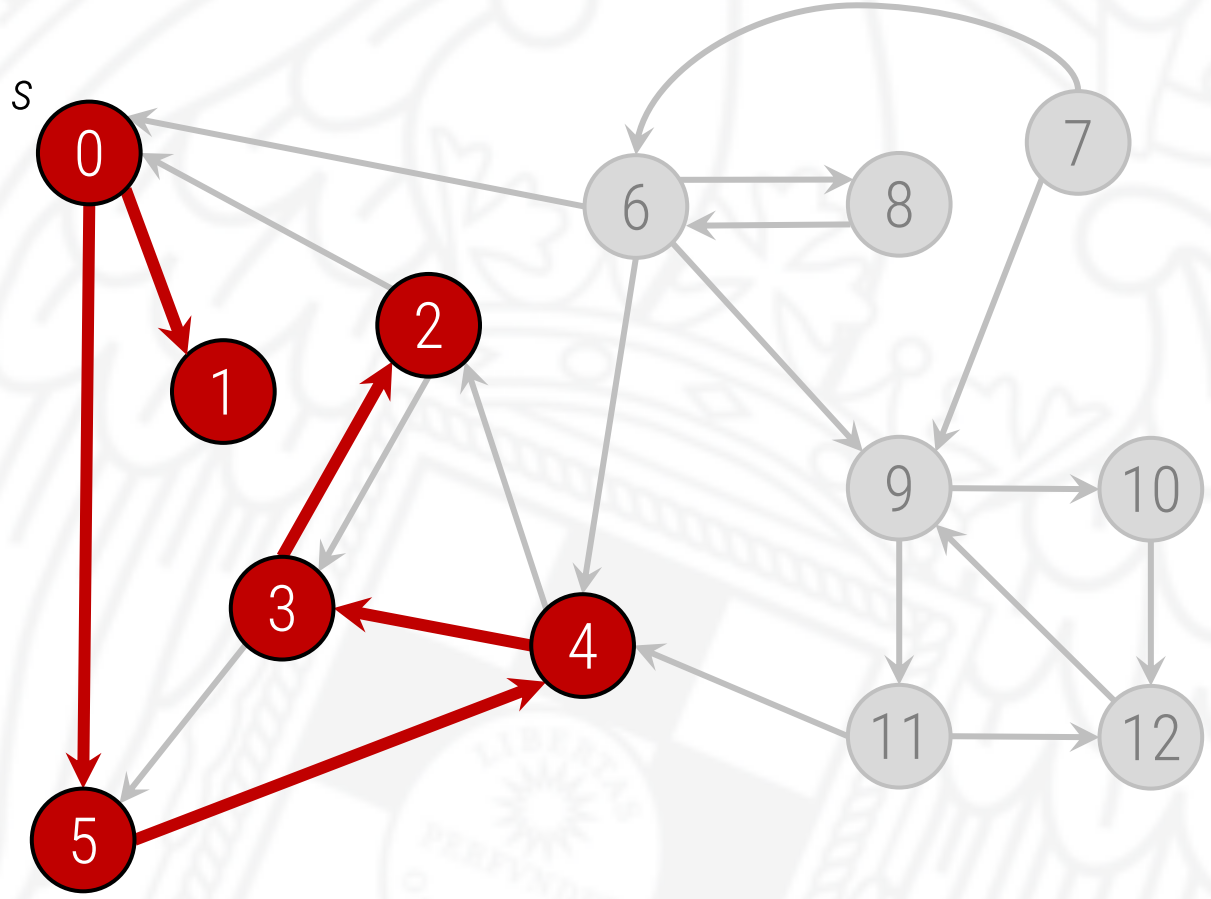
Digrafo.h

```
    Digrafo inverso() const {
        Digrafo inv(_V);
        for (int v = 0; v < _V; ++v) {
            for (int w : _ady[v]) {
                inv.ponArista(w, v);
            }
        }
        return inv;
    }
};
```

Recorrido en profundidad



Recorrido en profundidad



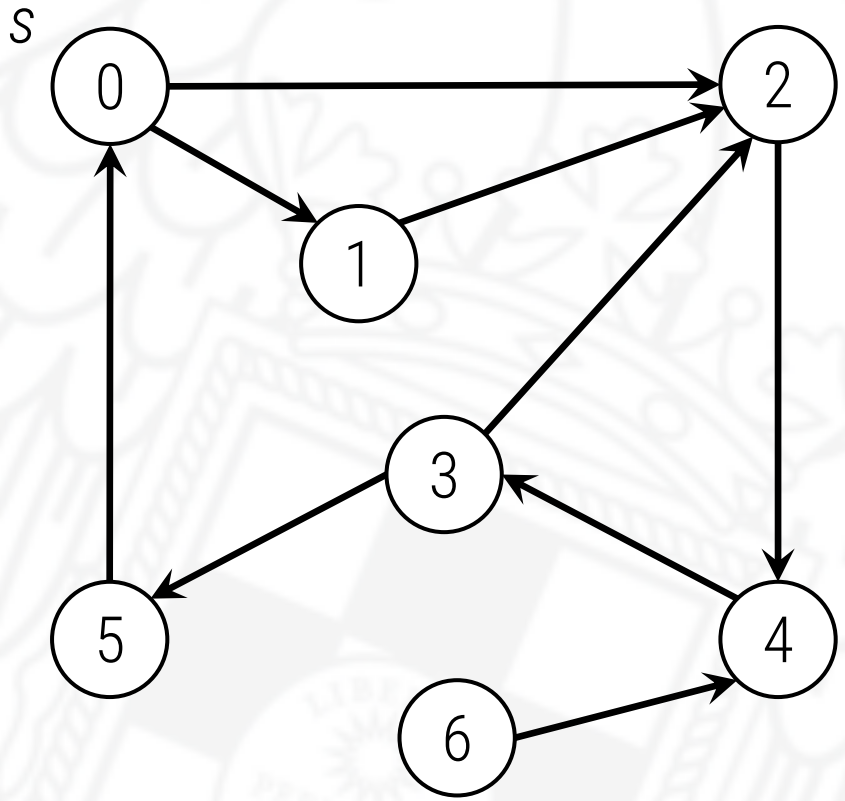
Recorrido en profundidad

```
class DFSDirigido {
public:
   DFSDirigido(Digrafo const& g, int s) : visit(g.V(), false) {
      dfs(g, s);
   }
   bool alcanzable(int v) const {
      return visit[v];
   }
private:
   std::vector<bool> visit; // visit[v] = ¿hay camino dirigido de s a v?
   void dfs(Digrafo const& g, int v) {
      visit[v] = true;
      for (int w : g.ady(v))
         if (!visit[w]) dfs(g, w);
   }
};
```

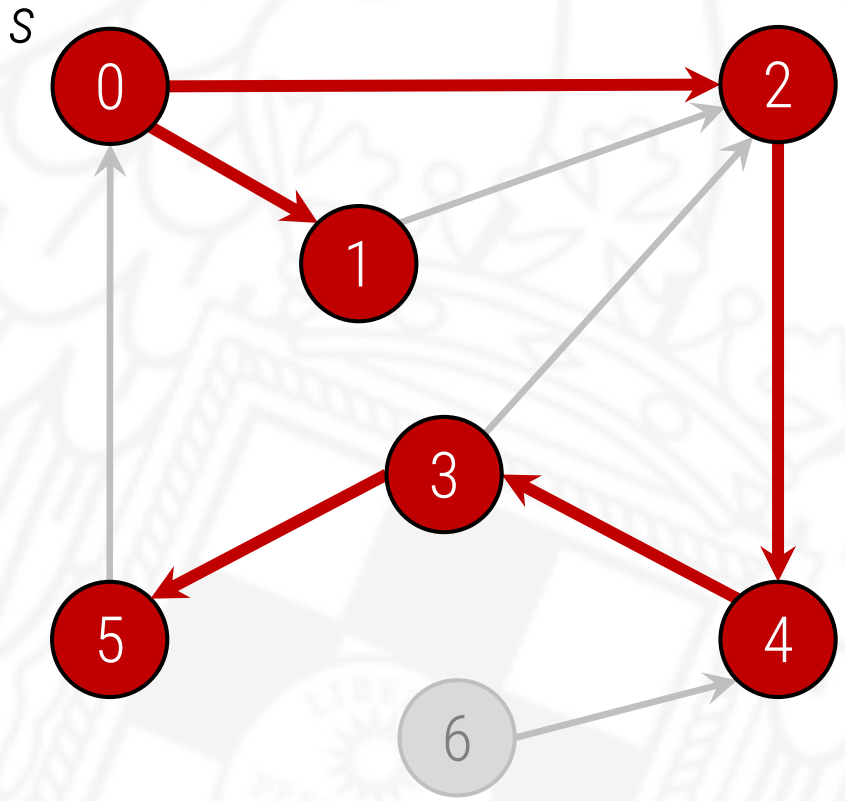
Alcanzabilidad

- Análisis del flujo de control de un programa
  - detectar código muerto
  - detectar bucles infinitos
- Recolector de basura
- Web crawler

Recorrido en anchura



Recorrido en anchura



Recorrido en anchura

```
class BFSDirigido {
public:
   BFSDirigido(Digrafo const& g, int s) : visit(g.V(), false),
                                          ant(g.V()), dist(g.V()), s(s) {
      bfs(g);
   }

   bool hayCamino(int v) const {
      return visit[v];
   }

   int distancia(int v) const {
      return dist[v];
   }
```

Recorrido en anchura

```
    Camino camino(int v) const {
        if (!hayCamino(v)) throw std::domain_error("No existe camino");
        Camino cam;
        for (int x = v; x != s; x = ant[x])
            cam.push_front(x);
        cam.push_front(s);
        return cam;
    }

private:
    std::vector<bool> visit; // visit[v] = ¿hay camino de s a v?
    std::vector<int> ant;  // ant[v]  = último vértice antes de llegar a v
    std::vector<int> dist; // dist[v] = aristas en el camino s->v más corto
    int s;
```

Recorrido en anchura

```
    void bfs(Digrafo const& g) {
        std::queue<int> q;
        dist[s] = 0; visit[s] = true;
        q.push(s);
        while (!q.empty()) {
            int v = q.front(); q.pop();
            for (int w : g.ady(v)) {
                if (!visit[w]) {
                    ant[w] = v; dist[w] = dist[v] + 1; visit[w] = true;
                    q.push(w);
                }
            }
        }
    }
};
```