

# DETECCIÓN DE CICLOS

Cómo detectar ciclos en un grafo dirigido. De esta forma un conjunto de tareas NO sería posible.

Utilizamos el recorrido en profundidad o DFS.

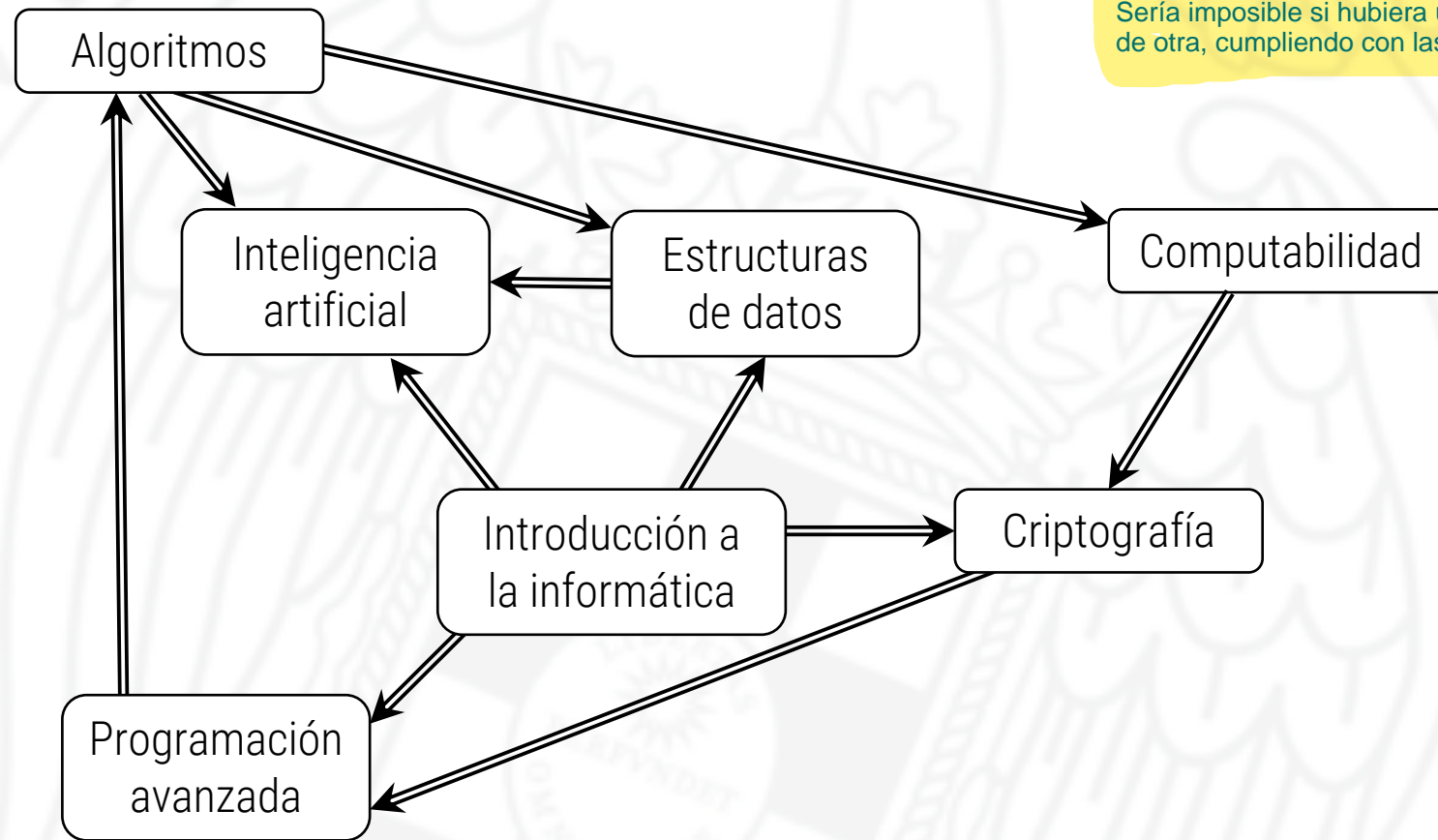


U N I V E R S I D A D  
COMPLUTENSE  
M A D R I D

**ALBERTO VERDEJO**

# Ciclos dirigidos

- ▶ En ocasiones representan problemas.



Sería imposible si hubiera un ciclo cumplir con las tareas, una detrás de otra, cumpliendo con las restricciones.

# Ciclos dirigidos

- En ocasiones representan problemas.

```
public class A extends B
{
    ...
}
```

```
public class B extends C
{
    ...
}
```

```
public class C extends A
{
    ...
}
```

Ciclos en la representación de herencia de un lenguaje OO

No podrían crearse objetos de ninguna de estas clases.

```
% javac A.java
A.java:1: cyclic inheritance
involving A
public class A extends B { }
                ^
1 error
```

# Ciclos dirigidos

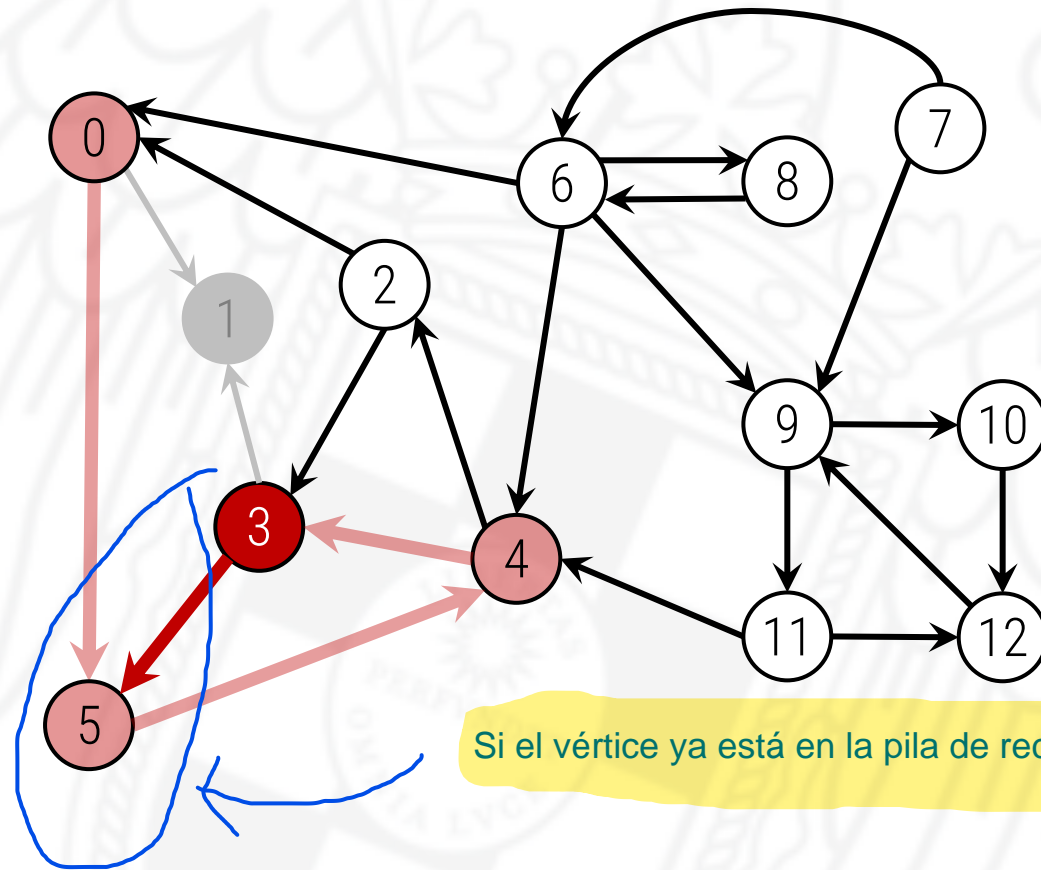
- ▶ En ocasiones representan problemas.

The screenshot shows a Google Sheets interface with a spreadsheet titled "ciclo en fórmulas". The spreadsheet has columns A, B, and C. In row 1, the formulas are: A1 contains  $=B1+1$ , B1 contains  $=C1+1$ , and C1 contains  $=A1+1$ . Blue arrows are drawn between the cells to illustrate the circular dependency: from A1 to B1, from B1 to C1, and from C1 back to A1. A red error message box is displayed on the right side of the spreadsheet, stating: "Error. Se ha detectado una dependencia circular. Para resolverla mediante cálculos iterativos, ve a Archivo > Configuración de la hoja de cálculo." The error message is in Spanish and suggests using iterative calculations to resolve the circular dependency.

Hay ciclos entre las fórmulas y detecta un error.

# Detección de ciclos dirigidos

- Utilizamos un recorrido en profundidad. La pila de la recursión contiene el camino *actual*.



Si el vértice ya está en la pila de recursión implica que hay un ciclo.

# Implementación

```
class CicloDirigido {
public:
    CicloDirigido(Digrafo const& g) : visit(g.V(), false), ant(g.V()),
                                     apilado(g.V(), false), hayciclo(false) {
        for (int v = 0; v < g.V(); ++v)
            if (!visit[v])
                dfs(g, v);
    }

    bool hayCiclo() const { return hayciclo; }

    Camino const& ciclo() const { return _ciclo; }
}
```


Vector que marca si está apilado o no un vértice.



# Implementación

private:

```
std::vector<bool> visit;    // visit[v] = ¿se ha alcanzado a v en el dfs?  
std::vector<int> ant;      // ant[v] = vértice anterior en el camino a v  
std::vector<bool> apilado; // apilado[v] = ¿está el vértice v en la pila?  
Camino _ciclo;             // ciclo dirigido (vacío si no existe)  
bool hayciclo;
```



# Implementación

```
void dfs(Digrafo const& g, int v) {  
    apilado[v] = true;  
    visit[v] = true;  
    for (int w : g.ady(v)) { recorremos los adyacentes.  
        if (hayciclo) // si hemos encontrado un ciclo terminamos  
            return; Lo ponemos dentro porque el bucle con operador no permite condiciones booleanas-  
        if (!visit[w]) { // encontrado un nuevo vértice, seguimos  
            ant[w] = v; dfs(g, w);  
        } else if (apilado[w]) { // hemos detectado un ciclo  
            // se recupera retrocediendo  
            hayciclo = true;  
            for (int x = v; x != w; x = ant[x])  
                _ciclo.push_front(x);  
            _ciclo.push_front(w); _ciclo.push_front(v);  
        }  
    }  
    apilado[v] = false;  
}
```

$O(V + A)$

