

# CURSO DE PROGRAMACIÓN COMPETITIVA URJC - 2021

## Sesión 4 (6ª Semana)

- David Morán (david.moran@urjc.es)
- Sergio Pérez (sergio.perez.pelo@urjc.es)
- Jesús Sánchez-Oro (jesus.sanchezoro@urjc.es)
- Isaac Lozano (isaac.lozano@urjc.es)
- Raúl Martín (raul.martin@urjc.es)
- Jakub Jan (jakubjanluczyn@gmail.com)
- Antonio Gonzalez (antonio.gpardo@urjc.es)
- Iván Martín (ivan.martin@urjc.es)
- Leonardo Antonio Santella (leocaracas2010@gmail.com)

# Contenidos

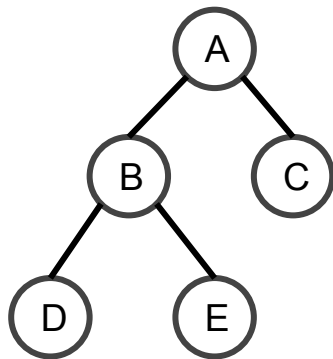
- Grafos
  - Introducción
  - Representación
  - Recorrido en Anchura y Profundidad (BFS, DFS)

# Contenidos

- Grafos
  - Bipartitos
  - Componentes Conexas
  - Ordenamiento Topológico
  - Puntos de articulación

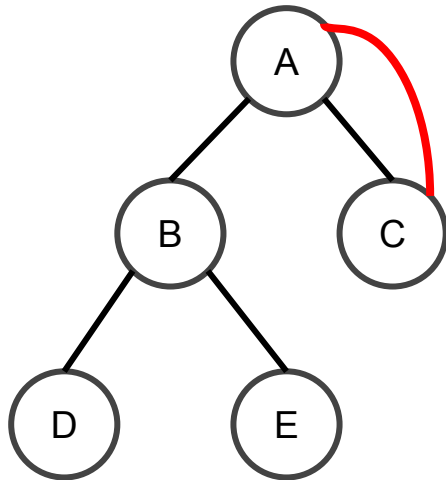
# Grafos

- Árboles: representan relaciones jerárquicas
  - Tienen un padre (excepto la raíz)
  - Pueden tener hijos



# Grafos

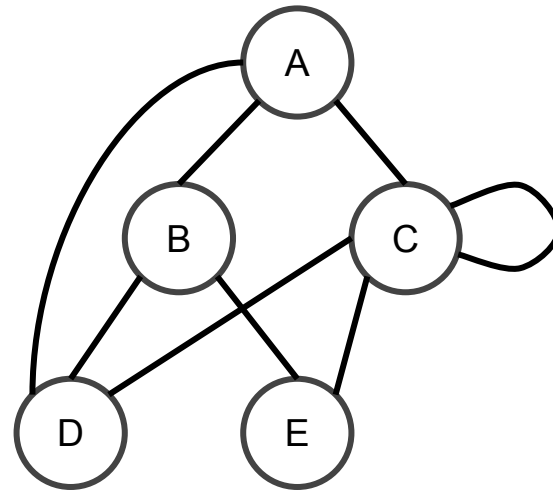
- Restricciones jerárquicas:
  - No admite ciclos



- C no puede ser padre de su padre (A)

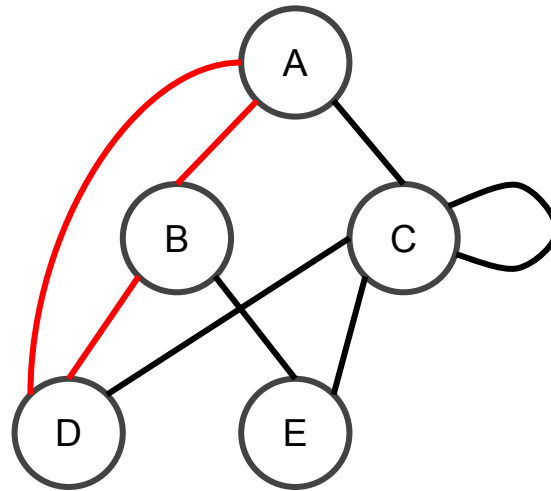
# Grafos

- Grafos: mayor libertad para representar un sistema y sus relaciones/interacciones



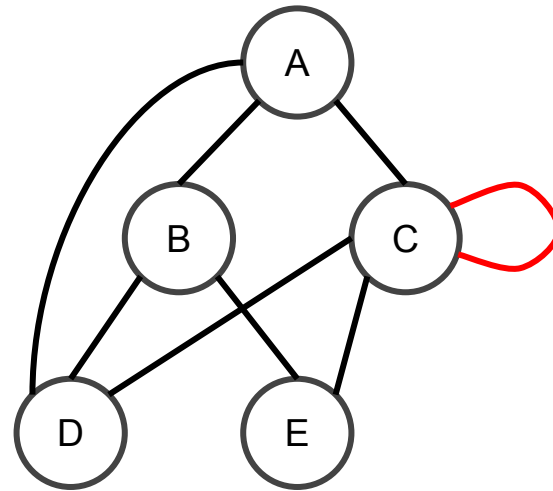
# Grafos

- podemos tener ciclos



# Grafos

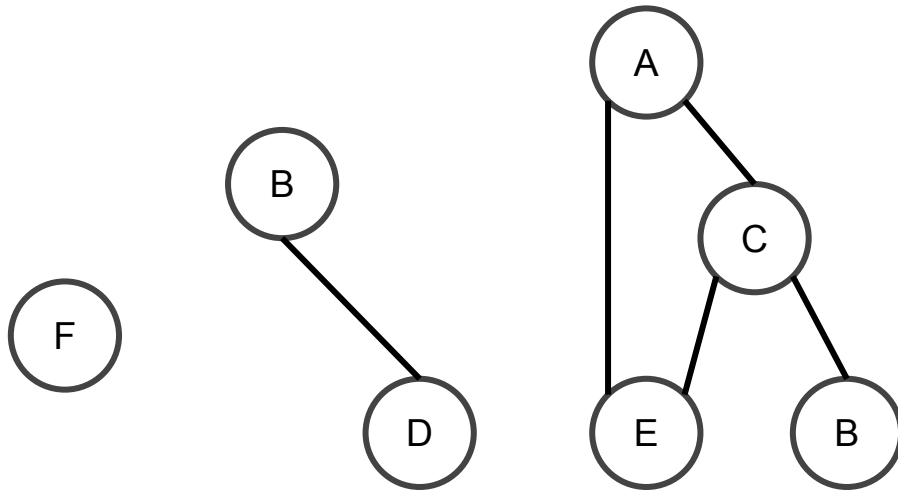
- podemos tener bucles sobre el mismo elemento





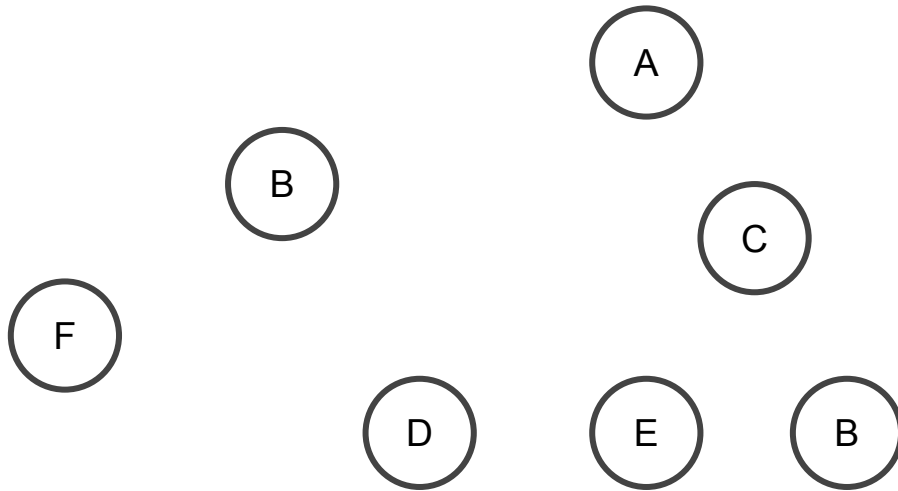
# Grafos

- podemos tener grupos aislados en un mismo grafo



# Grafos

- o elementos totalmente aislados entre sí



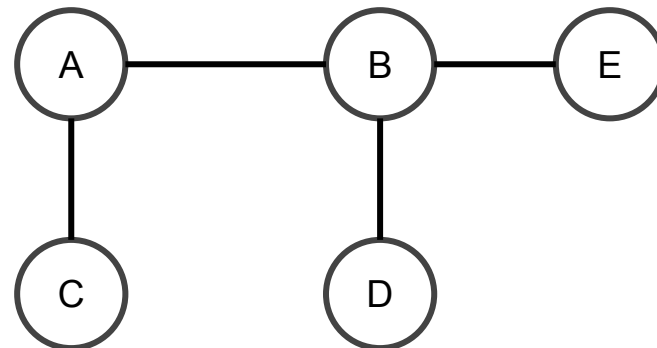
# Grafos

- Definición:  $G = (V, E)$ 
  - Conjunto de vértices:

$V = \{A, B, C, D, E\}$

- Conjunto de aristas:

$E = \{(A, B), (A, C), (B, D), (B, E)\}$

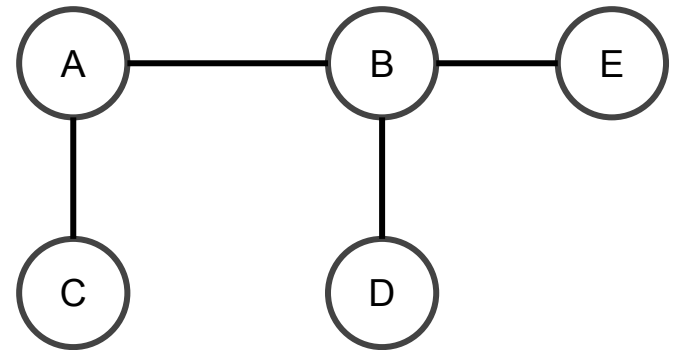


# Grafos

- Grafo no dirigido  $G = (V, E)$ 
  - las aristas no tienen dirección

$$E = \{(A, B), (A, C), (B, D), (B, E)\}$$

$$(A, B) \Leftrightarrow (B, A)$$

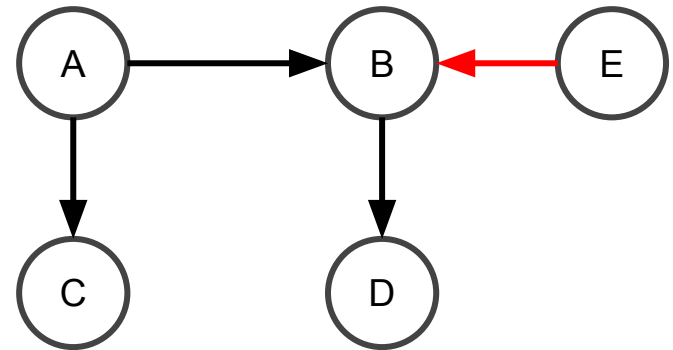


# Grafos

- Grafo dirigido  $G = (V, E)$ 
  - aristas tienen dirección (orden)

$E = \{(A, B), (A, C), (B, D), (E, B)\}$

$(E, B) \neq (B, E)$



# Grafos

- Grafo ponderado  $G = (V, E)$ 
  - las aristas tienen pesos/valores

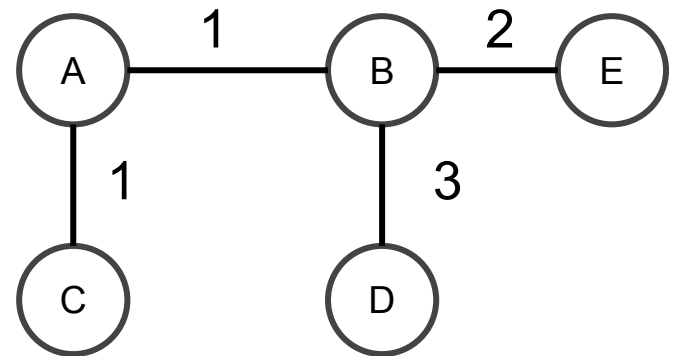
$E = \{(A, B), (A, C), (B, D), (B, E,)\}$

- función de pesos ( $f$ )

$$f((A, B)) = 1$$

$$f((B, D)) = 3$$

...

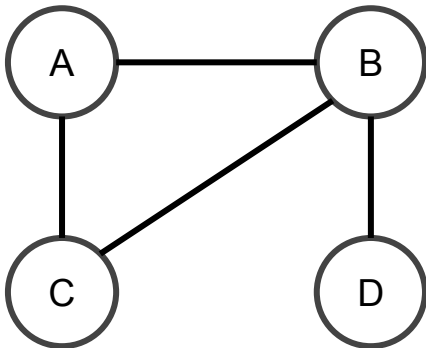


# Grafos - Implementación

- Implementaciones
  - Matriz de adyacencia
  - Lista de adyacencia
  - Lista de aristas

# Grafos - Matriz

- Matriz de adyacencia
  - Array de dos dimensiones

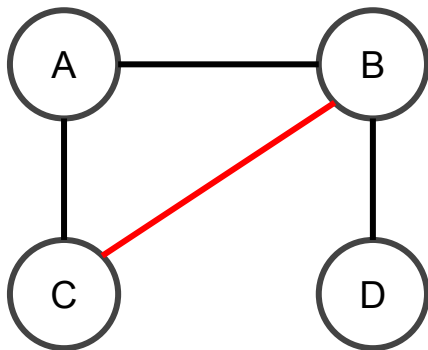


	A	B	C	D
A	0	1	1	0
B	1	0	1	1
C	1	1	0	0
D	0	1	0	0



# Grafos - Matriz

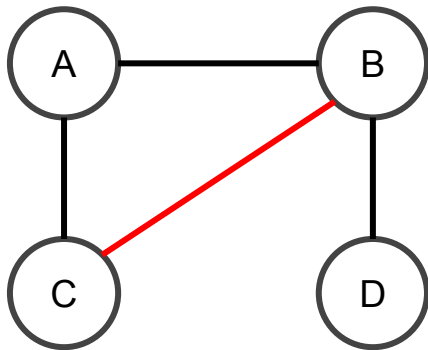
- Arista (B,C)  $\Rightarrow m[1][2] = 1$



	A	B	C	D
A	0	1	1	0
B	1	0	1	1
C	1	1	0	0
D	0	1	0	0

# Grafos - Matriz

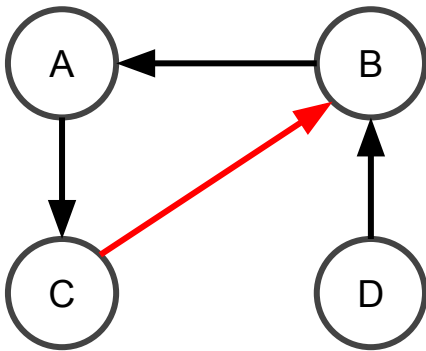
- Matriz simétrica en grafos no dirigidos
- $m[\text{fila}][\text{col}] == m[\text{col}][\text{fila}]$



	A	B	C	D
A	0	1	1	0
B	1	0	1	1
C	1	1	0	0
D	0	1	0	0

# Grafos - Matriz

- Si el grafo es dirigido...
- $m[2][1]=1$  y  $m[1][2]=0$



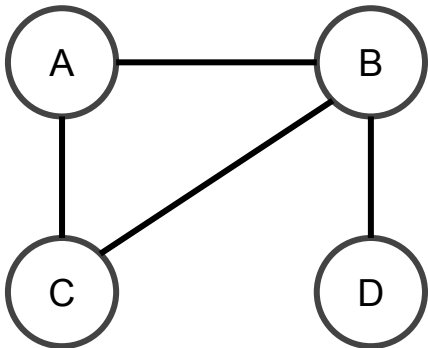
	A	B	C	D
A	0	0	1	0
B	1	0	0	0
C	0	1	0	0
D	0	1	0	0

# Grafos - Matriz

- Matriz de adyacencia
  - Memoria:  $O(|V|^2)$
  - Acceso:  $O(1)$
  - Aristas de un vértice:  $O(|V|)$ 
    - hay que recorrer toda la fila (incluso si solo tiene una o ninguna)
- Caso de uso: grafos densos ( $N \sim 5000$ )

# Grafos - Lista

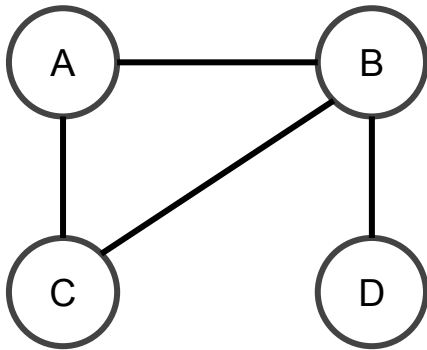
- Lista de adyacencia
  - enumerar las aristas por vértice



A	⇒	{B,C}
B	⇒	{A,C,D}
C	⇒	{A,B}
D	⇒	{B}

# Grafos - Lista

- Lista de adyacencia
  - guardar cada lista en un array



A	{B,C}
B	{A,C,D}
C	{A,B}
D	{B}

# Grafos - Lista

- Lista de adyacencia
  - Memoria:  $O(|V|+|E|)$
  - Acceso:  $O(|V|)$ 
    - recorrer todas las aristas de la lista
  - Aristas de un vértice:  $O(1)$ 
    - en el peor caso tiene aristas a todos los vértices
- útil en grafos dispersos

# Grafos - Lista

- Consultar si existe una arista (pseudocódigo)

```
grafo = vector<int> adyacentes[]
```

```
//inicializar el vector / añadir aristas
```

```
A = 0
```

```
C = 2
```

```
adyacentesA = grafo[A]
```

```
existeAC = grafo[A].contiene(C)
```



# Grafos - Mapas

- Permite utilizar cualquier tipo de etiquetas
  - no solo indices sino strings u otros

```
traduccion = mapa<string, int>
```

```
grafo = vector<int> adyacentes
```

```
adyacentes = grafo[traduccion["madrid"]]
```

```
existeArista = adyacentes.contiene("murcia")
```

# Grafos - Lista Aristas

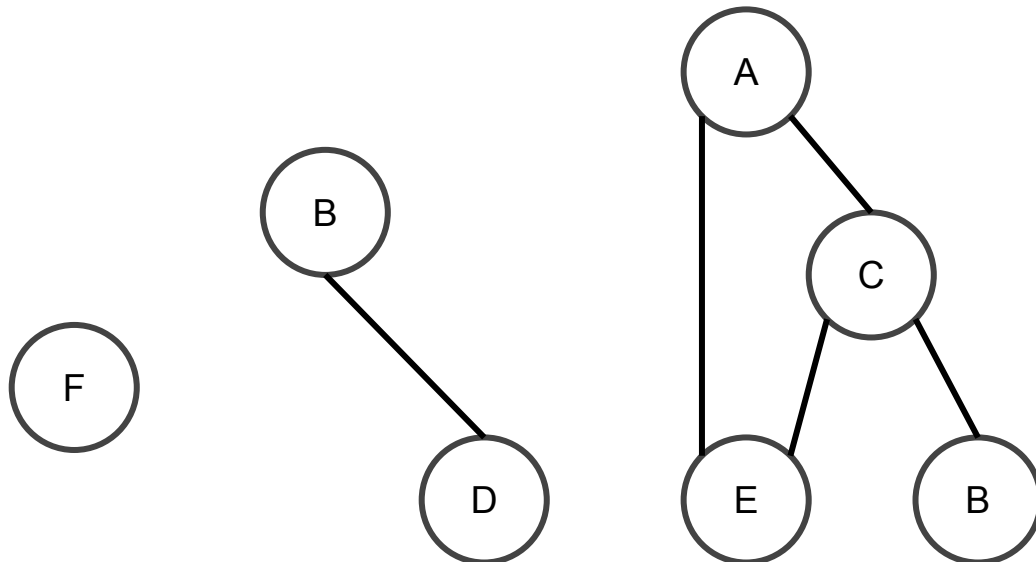
- Existe otra implementación para representar un grafo
- Más utilizada para algoritmos específicos
- Guardar en una lista las conexiones de A a B
- `vector<arista> aristas`
- `aristas.insertar(arista(a, b))`

# Grafos - Recorridos

- Recorrer los los vértices de un grafo
  - Recorrido en anchura (BFS - Breadth First Search)
  - Recorrido en profundidad (DFS - Depth First Search)

# Grafos - Recorridos

- Recorrer los vértices de un grafo
  - Seleccionamos un vértice al azar
  - Recorrido BFS o DFS



# Grafos - Recorridos

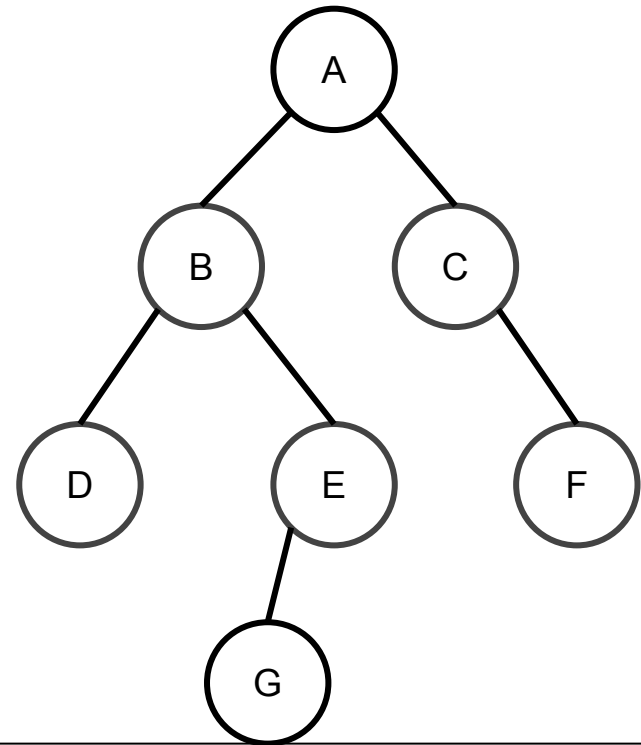
- Recorrer los vértices de un grafo
  - Necesitamos llevar cuenta de en qué nodos hemos estado ya
  - Ventajas: Si llegamos a un nodo destino, está garantizado que habremos llegado en la menor cantidad de pasos (solo BFS)
  - Cada arista recorrida cuenta como 1 paso

# Grafos - Recorridos

- Recorrer los vértices de un grafo
  - No siempre se puede recorrer todo desde un vértice inicial
- Soluciones:
  - Elegir un vértice nuevo (no visitado)
  - Ignorar vértices desconectados
  - etc (depende del problema)

# Grafos - BFS

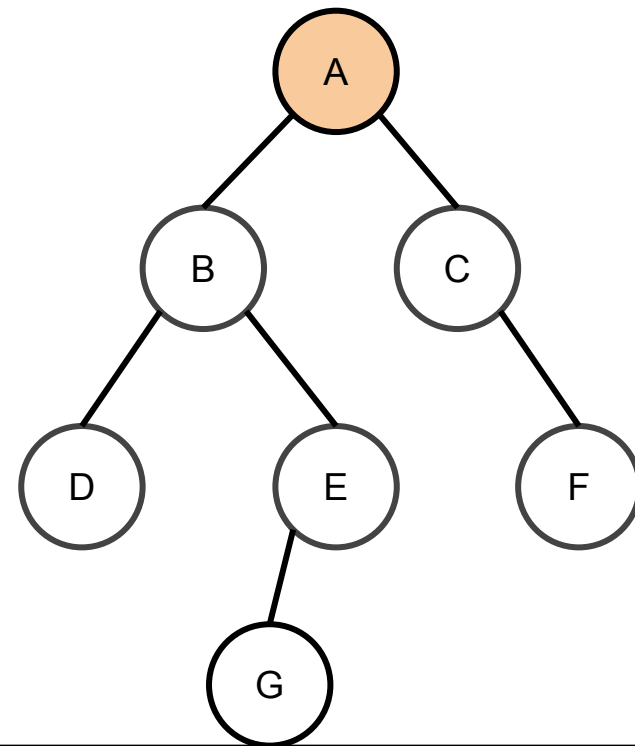
- Recorrido en anchura



# Grafos - BFS

- Si empezamos desde A
- Recorrido por niveles

A

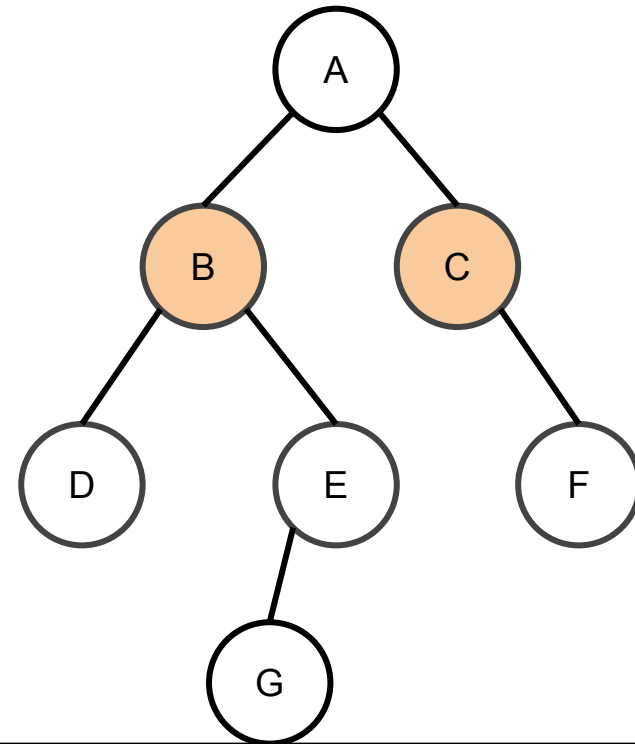




# Grafos - BFS

- Si empezamos desde A
- Recorrido por niveles

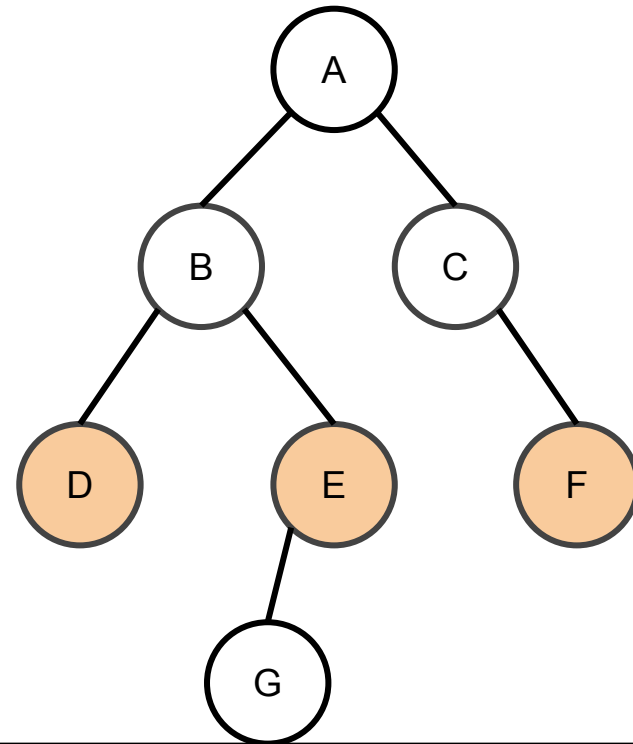
A ⇒ B ⇒ C



# Grafos - BFS

- Si empezamos desde A
- Recorrido por niveles

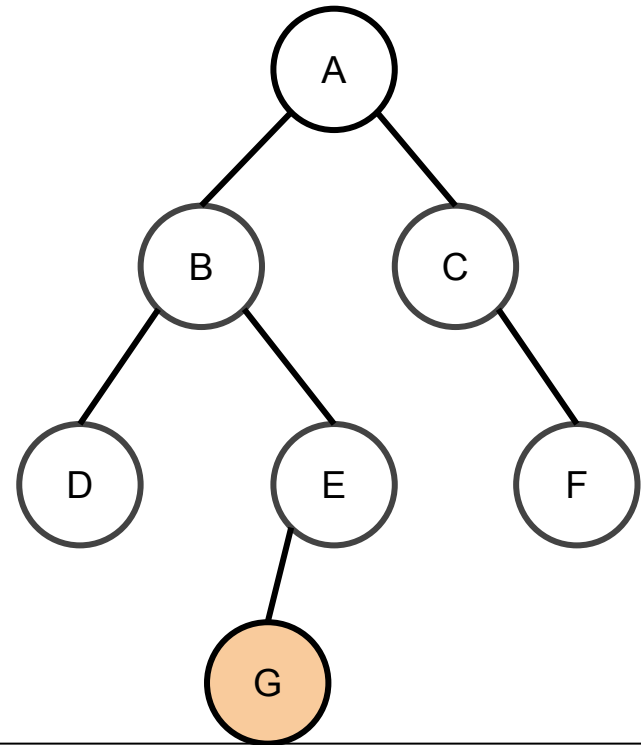
A ⇒ B ⇒ C ⇒ D ⇒ E  
⇒ F ⇒ G



# Grafos - BFS

- Si empezamos desde A
- Recorrido por niveles

A  $\Rightarrow$  B  $\Rightarrow$  C  $\Rightarrow$  D  $\Rightarrow$  E  
 $\Rightarrow$  F  $\Rightarrow$  G



# Grafos - BFS

- Implementación
  - Array de valores booleanos (visitados)
  - Cola de vértices a explorar
    - Se procesan en orden de llegada

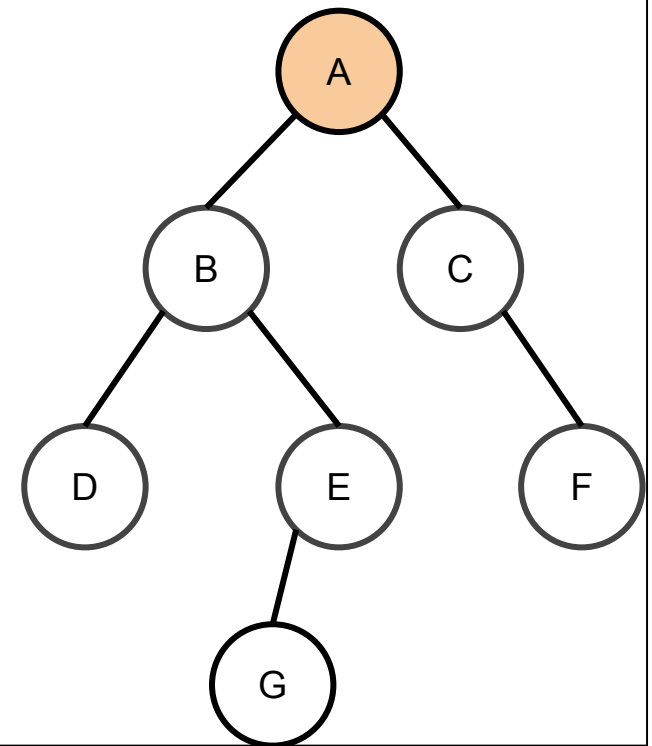
# Grafos - BFS

- Inicialización: Elegimos un vértice inicial

`inicial = 0`

`visitado[inicial]=true`

`cola.add(inicial)`



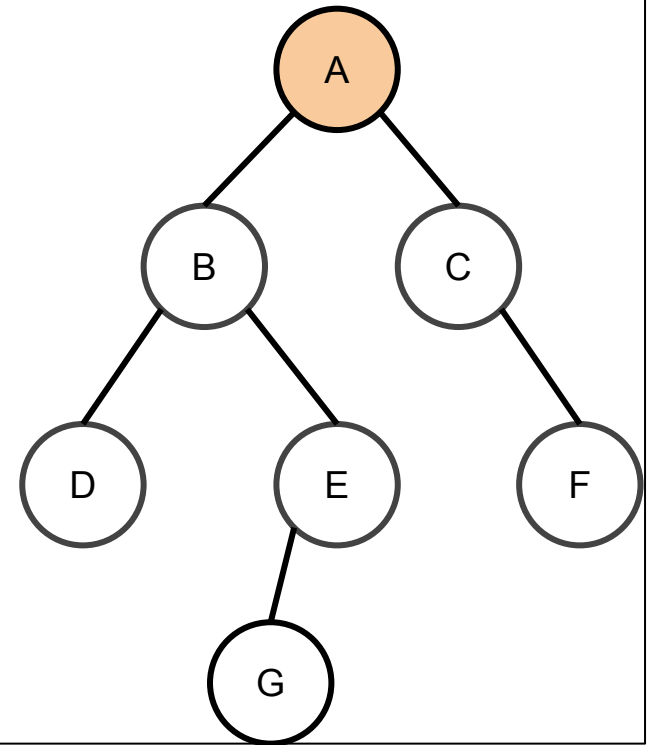
# Grafos - BFS

- Cola = {A}

inicial = A

visitado[inicial]=true

cola.add(inicial)



# Grafos - BFS

- Recorremos los vértices

```
mientras(colea.size() > 0)
```

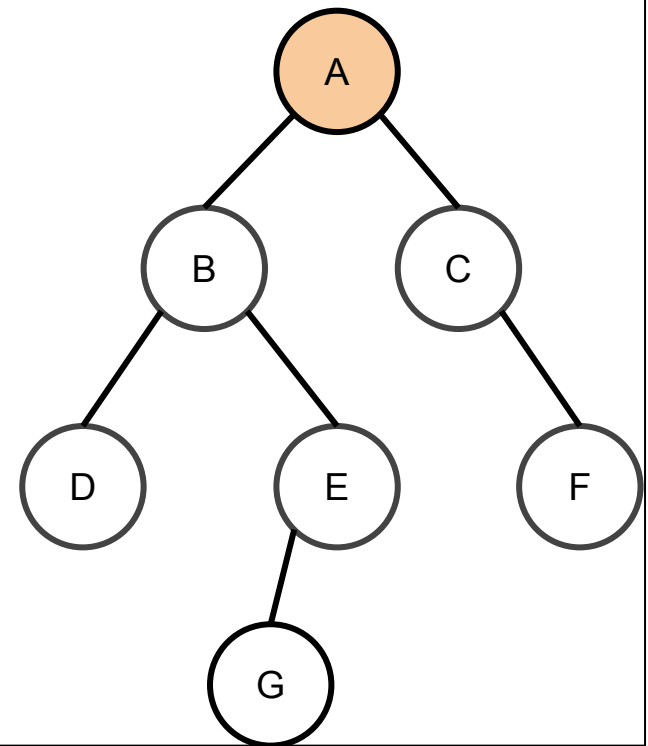
```
  v = colea.extraer()
```

```
  para cada ady de v:
```

```
    if(no visitado[ady])
```

```
      visitado[ady]=true
```

```
      colea.add(ady)
```



# Grafos - BFS

- Cola = {}
- Sacamos A

mientras(`cola.size() > 0`)

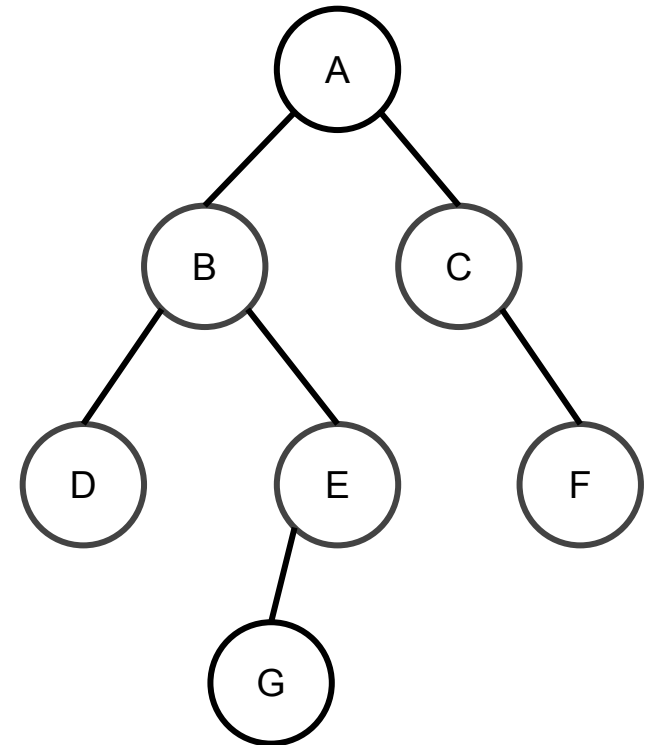
`v = cola.extraer()`

para cada `ady` de `v`:

if(`no visitado[ady]`)

`visitado[ady]=true`

`cola.add(ady)`





# Grafos - BFS

- Cola = {B,C}

mientras(`cola.size() > 0`)

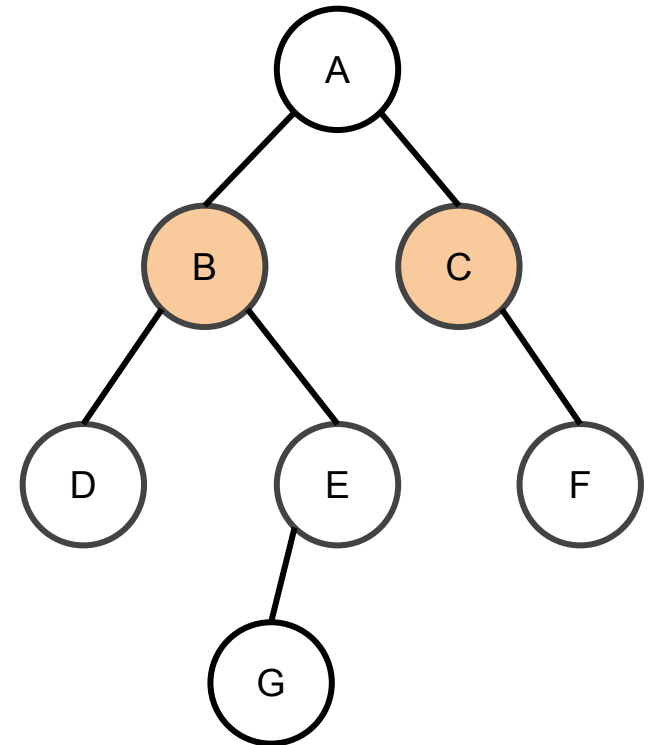
`v = cola.extraer()`

para cada **ady** de v:

if(`no visitado[ady]`)

`visitado[ady]=true`

**`cola.add(ady)`**



# Grafos - BFS

- Cola = {C}
- Sacamos B

mientras(`cola.size() > 0`)

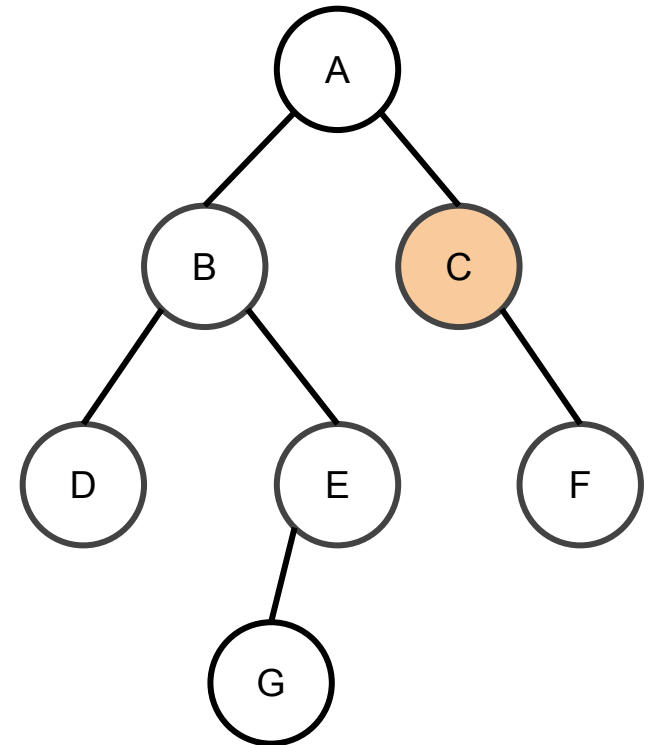
`v = cola.extraer()`

para cada `ady` de `v`:

`if(no visitado[ady])`

`visitado[ady]=true`

`cola.add(ady)`



# Grafos - BFS

- Cola = {C, D, E}



mientras(`cola.size() > 0`)

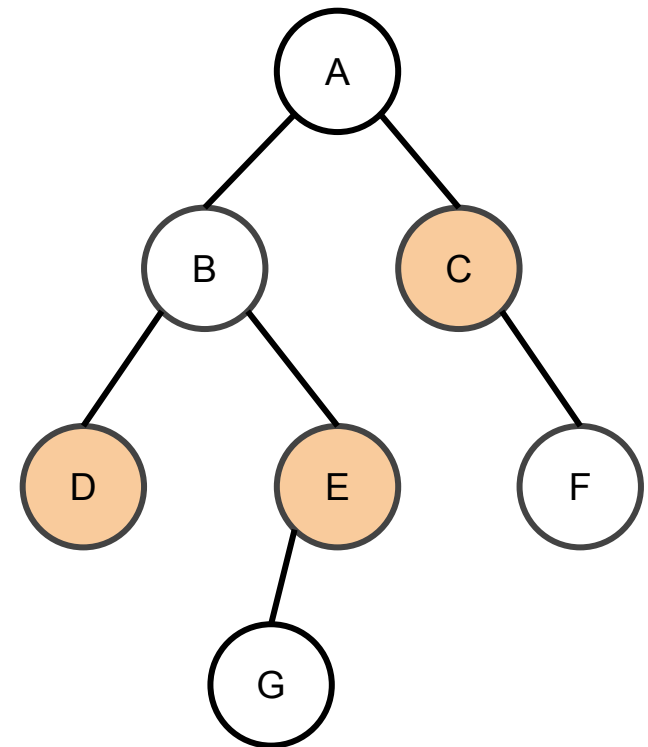
`v = cola.extraer()`

    para cada `ady` de `v`:

        if(`no visitado[ady]`)

`visitado[ady]=true`

`cola.add(ady)`



# Grafos - BFS

- Cola = {D, E, F}
- Sacamos C, Metemos F mientras(`cola.size()` > 0)

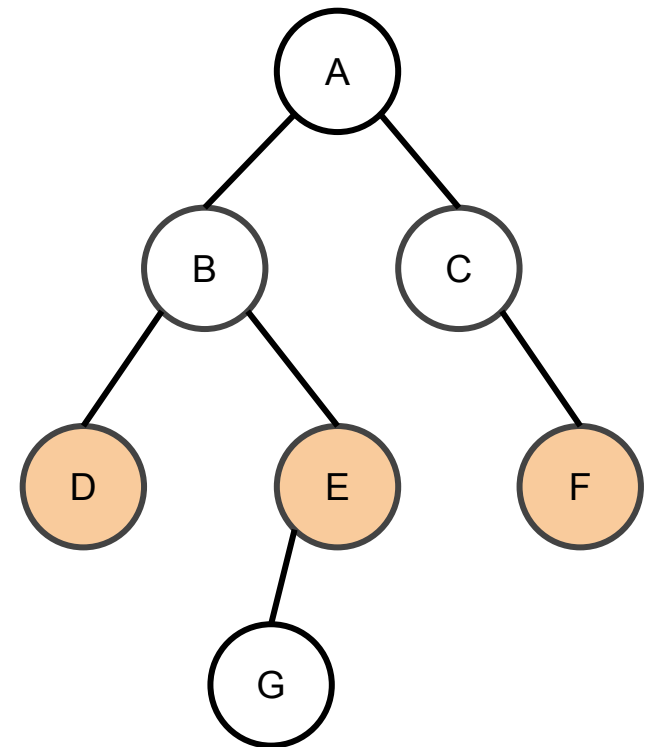
`v = cola.extraer()`

para cada ady de v:

if(`no visitado[ady]`)

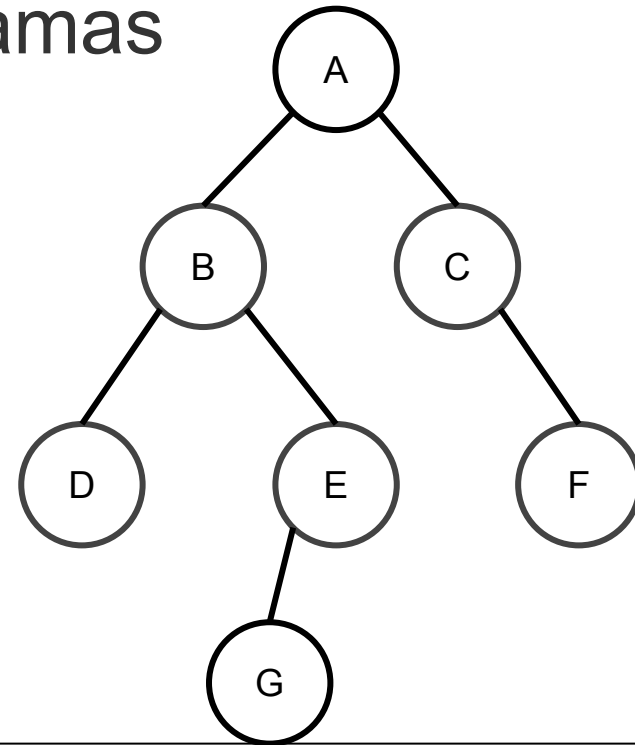
`visitado[ady]=true`

`cola.add(ady)`



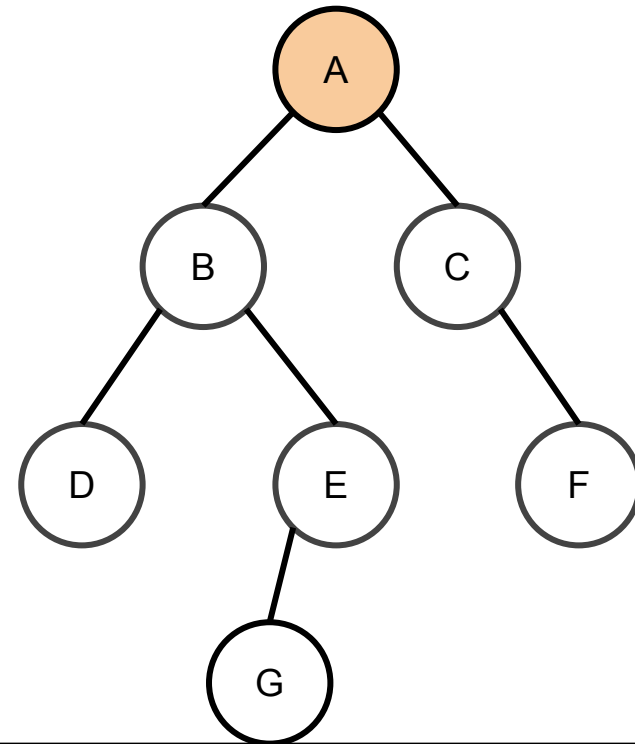
# Grafos - DFS

- Recorrido en profundidad
- Equivalente a recorrer ramas



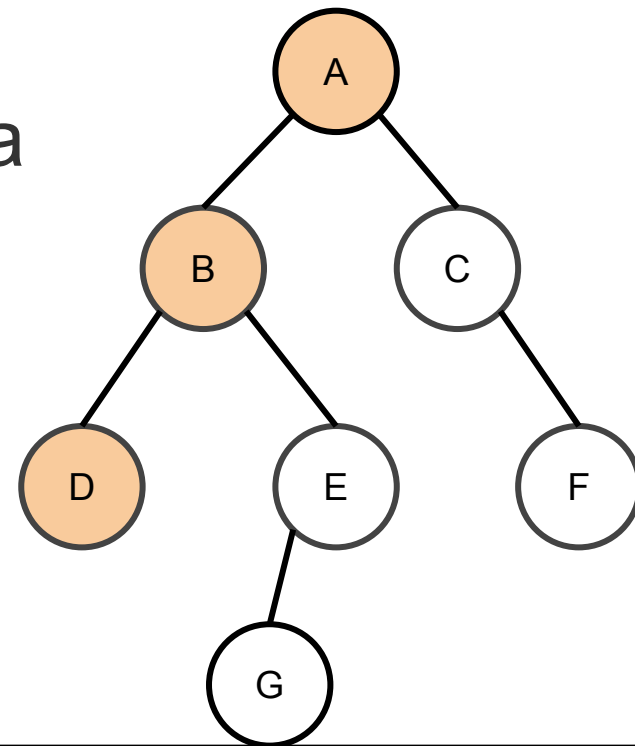
# Grafos - DFS

- Recorrido en profundidad
- Seleccionamos A



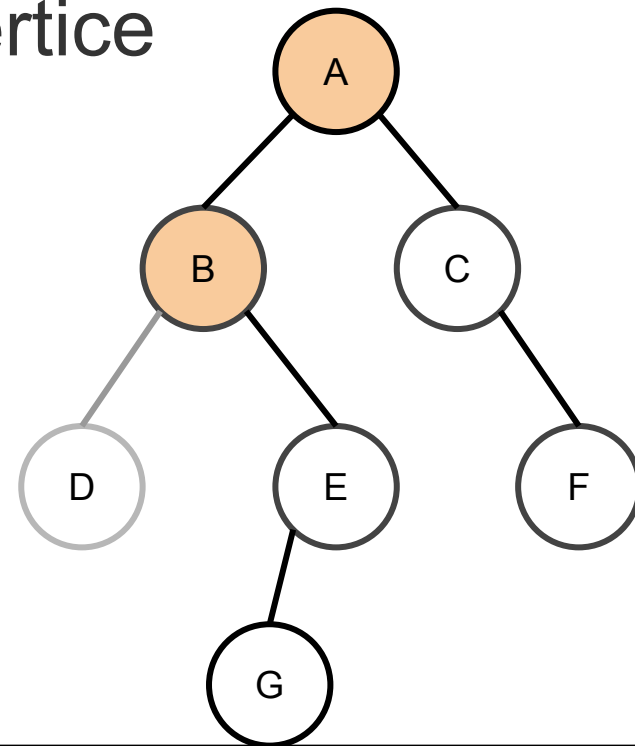
# Grafos - DFS

- Recorrido en profundidad
- Seleccionamos A
  - Recorremos una rama hasta el final



# Grafos - DFS

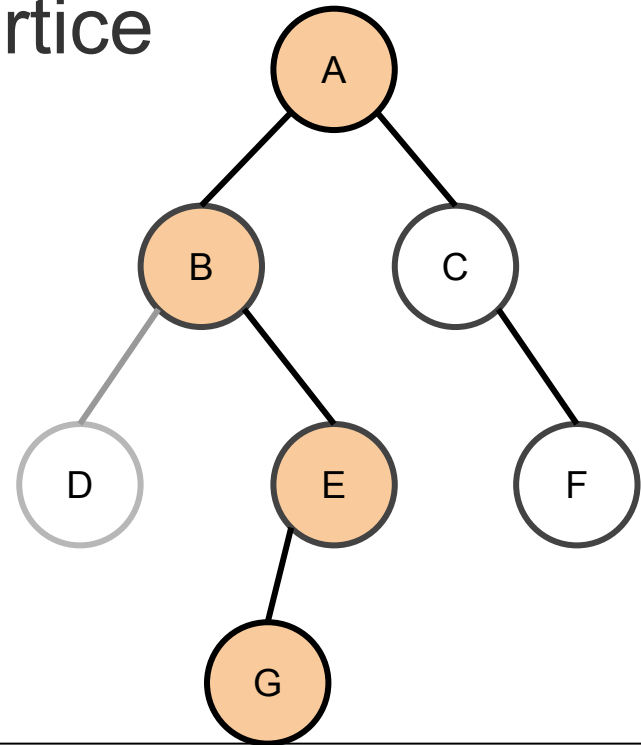
- Recorrido en profundidad
- volvemos al siguiente vértice con adyacentes (B)





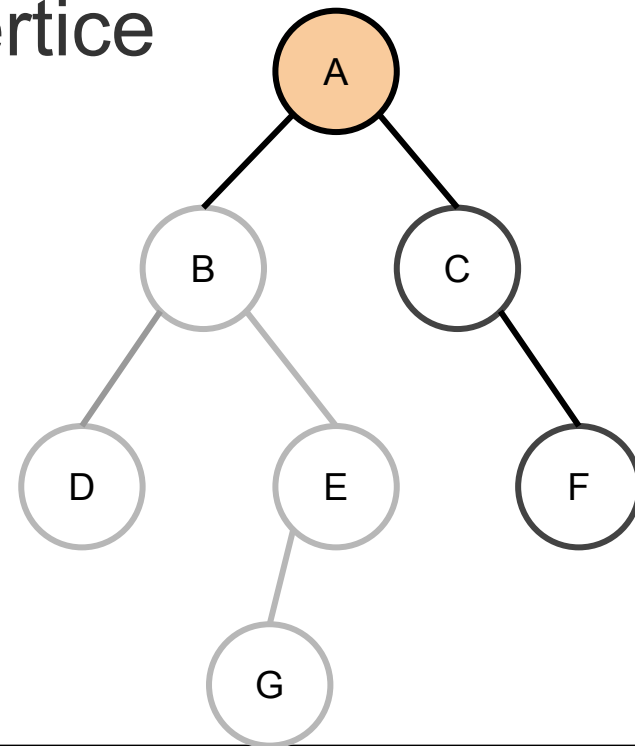
# Grafos - DFS

- Recorrido en profundidad
- volvemos al siguiente vértice con adyacentes (B)
  - recorreremos la nueva rama hasta el final



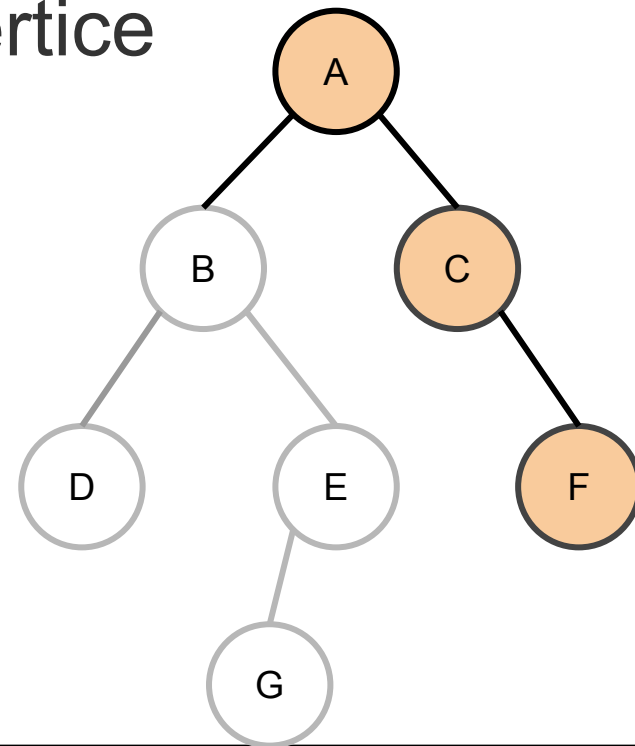
# Grafos - DFS

- Recorrido en profundidad
- volvemos al siguiente vértice con adyacentes (A)



# Grafos - DFS

- Recorrido en profundidad
- volvemos al siguiente vértice con adyacentes (A)



# Grafos - DFS

- Implementación
  - Array de valores booleanos (visitados)
  - Pila de vértices a explorar
    - Se procesa el más reciente primero
    - Acumulamos los más antiguos para el final

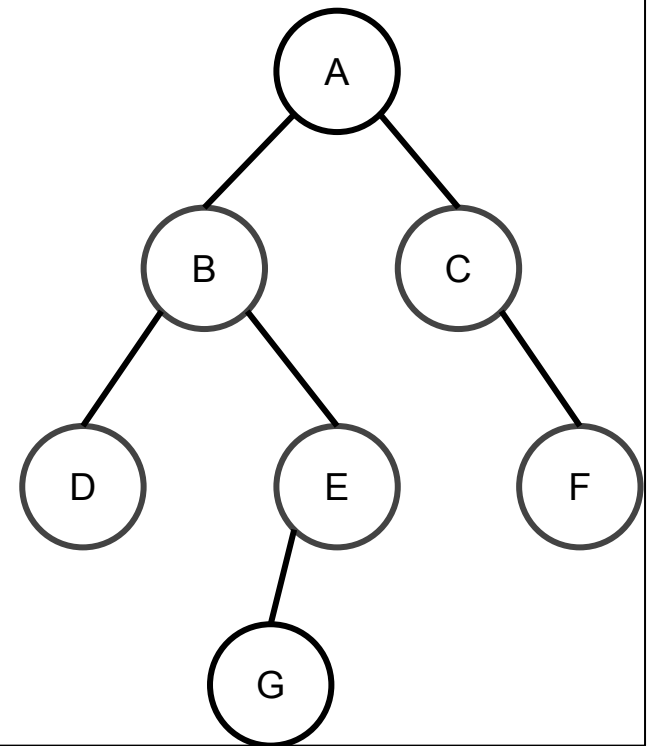
# Grafos - DFS

- Inicialización: Elegimos un vértice inicial

`inicial = A`

`visitado[inicial]=true`

`pila.add(inicial)`



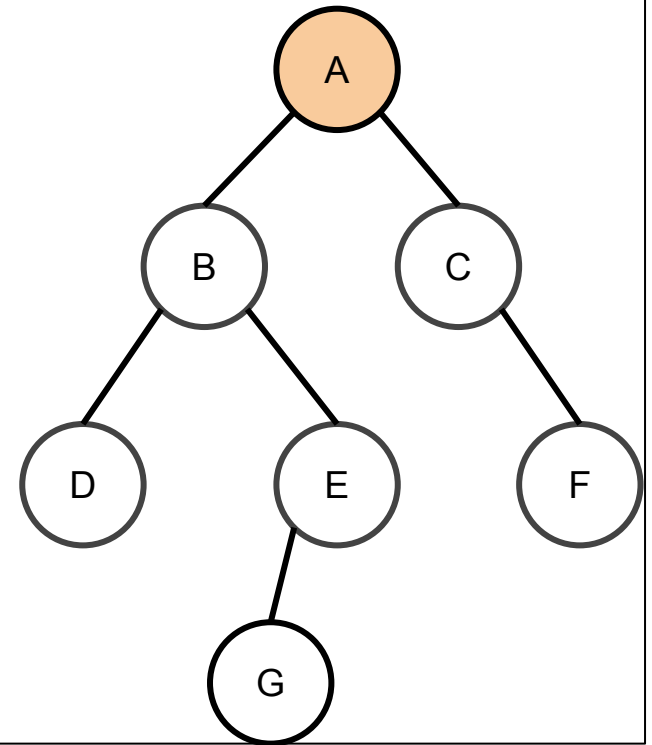
# Grafos - DFS

- Pila = {A}  $\leftarrow$  cima de la pila por la derecha

inicial = A

visitado[inicial]=true

pila.add(inicial)



# Grafos - DFS

- Recorremos los vértices

```
mientras(pila.size() > 0)
```

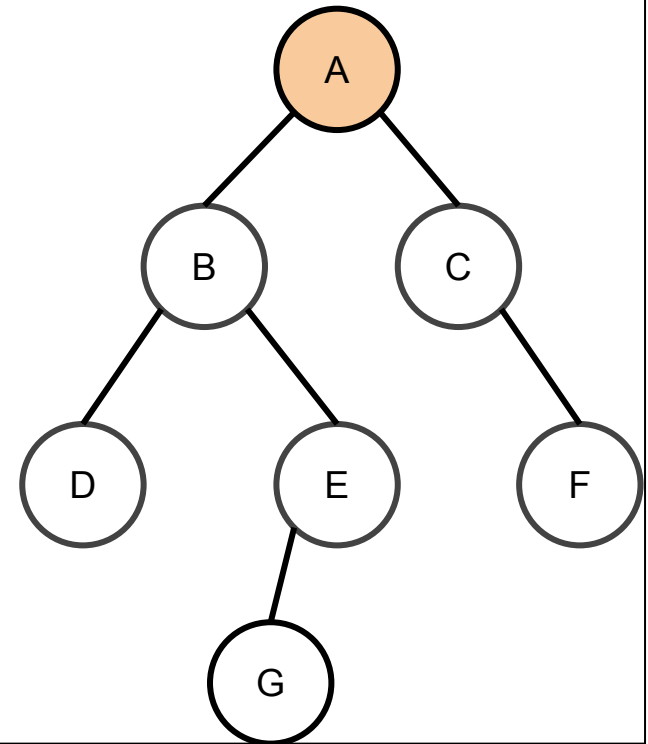
```
  v = pila.extraer()
```

```
  para cada ady de v:
```

```
    if(no visitado[ady])
```

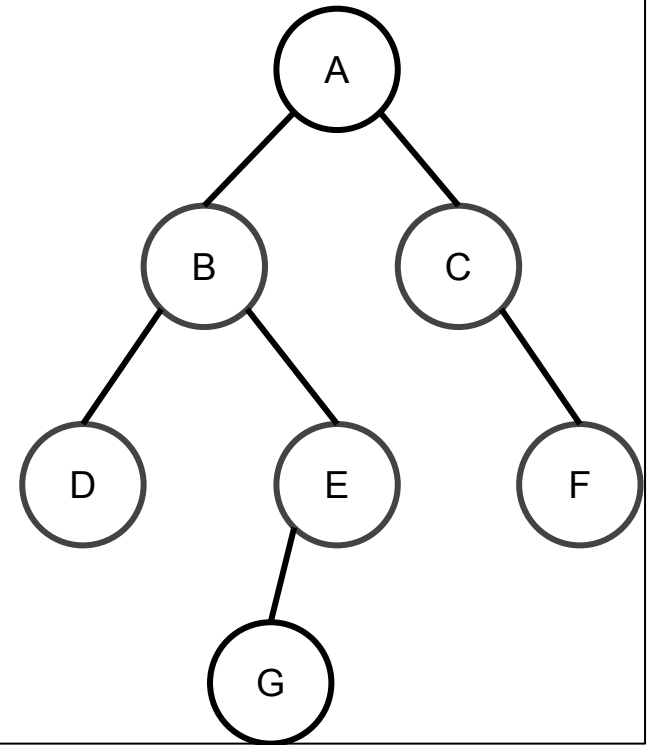
```
      visitado[ady]=true
```

```
      pila.add(ady)
```



# Grafos - DFS

- Pila = {}
  - Sacamos A
- mientras(pila.size() > 0)
- v = pila.extraer()**
- para cada ady de v:
- if(no visitado[ady])
- visitado[ady]=true
- pila.add(ady)





# Grafos - DFS

- Pila = {B,C}

mientras(pila.size() > 0)

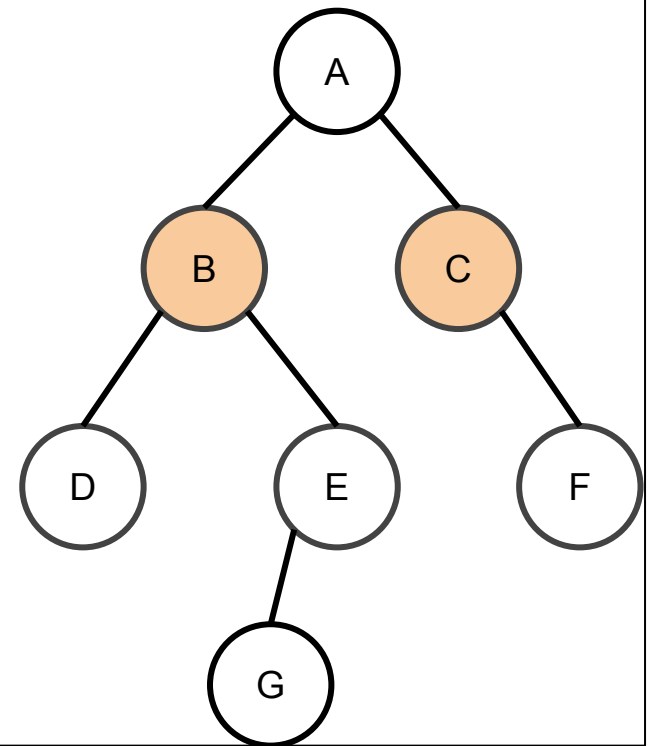
  v = pila.extraer()

  para cada **ady** de v:

    if(no visitado[ady])

      visitado[ady]=true

**pila.add(ady)**



# Grafos - DFS

- Pila = {B}
- Sacar C

mientras(pila.size() > 0)

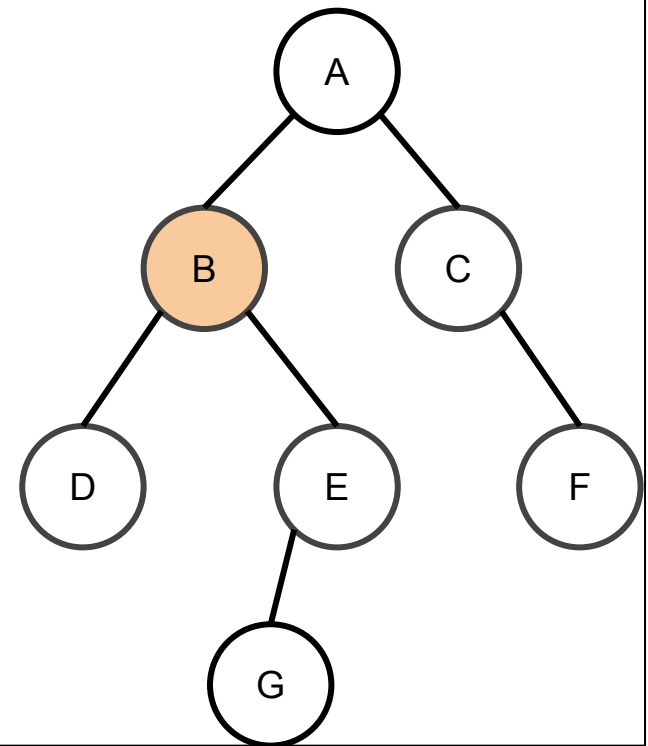
**v = pila.extraer()**

para cada ady de v:

if(no visitado[ady])

visitado[ady]=true

pila.add(ady)



# Grafos - DFS

- Pila = {B,F}

mientras(pila.size() > 0)

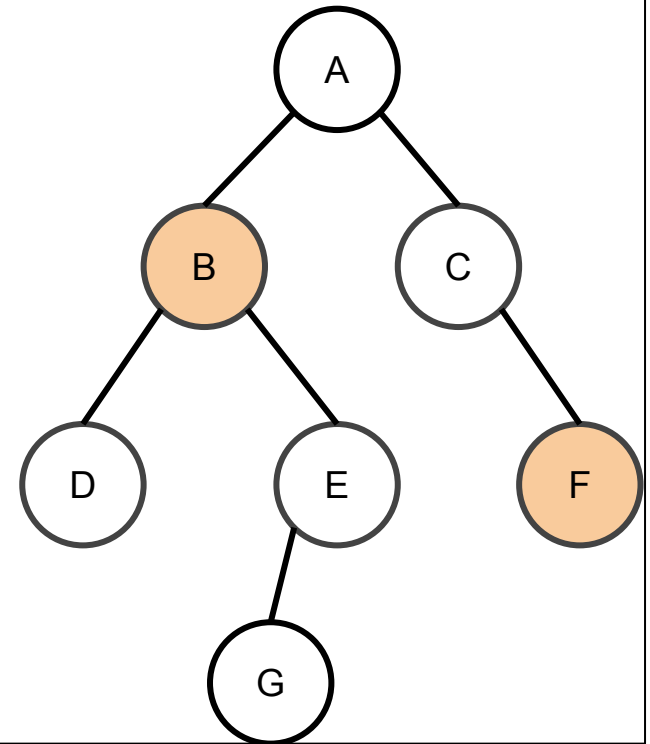
  v = pila.extraer()

  para cada **ady** de v:

    if(no visitado[ady])

      visitado[ady]=true

**pila.add(ady)**



# Grafos - DFS

- Pila = {B}
- Sacar F

mientras(pila.size() > 0)

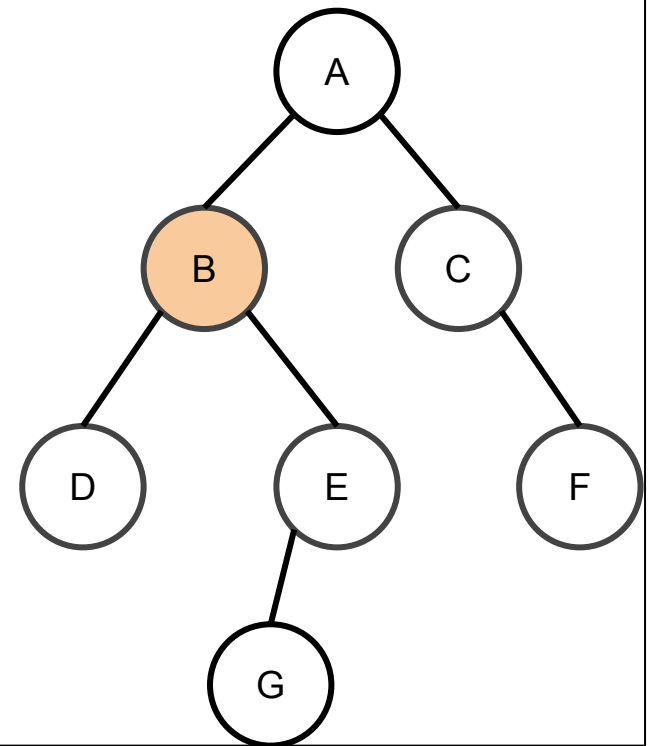
**v = pila.extraer()**

para cada ady de v:

if(no visitado[ady])

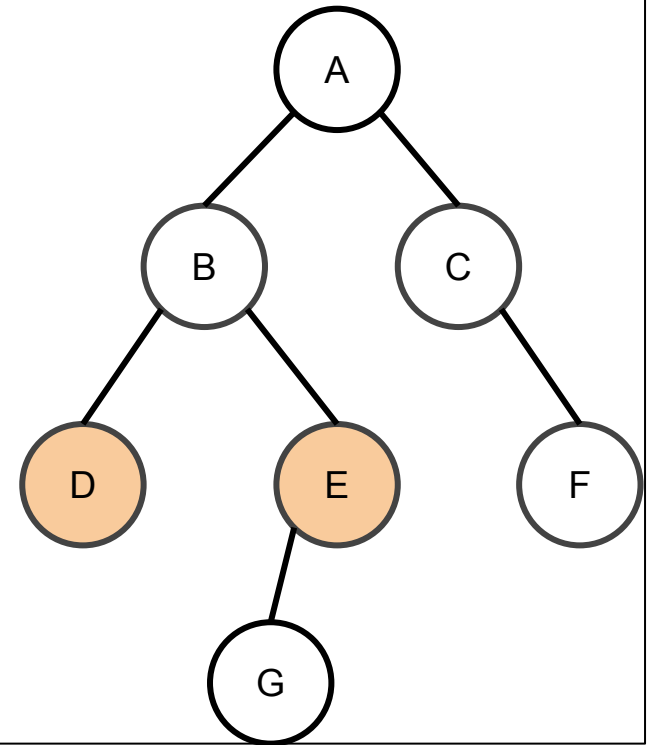
visitado[ady]=true

pila.add(ady)



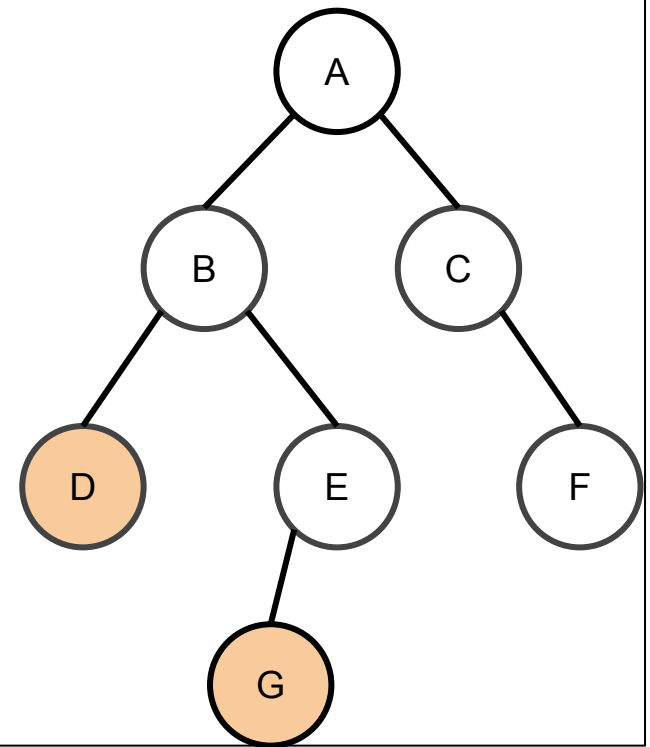
# Grafos - DFS

- Pila = {D,E}
  - Sacar B, Meter D,E
- mientras(pila.size() > 0)
- v = pila.extraer()**
- para cada ady de v:
- if(no visitado[ady])
- visitado[ady]=true
- pila.add(ady)**



# Grafos - DFS

- Pila = {D, G}
  - Sacar E, Meter G
- mientras(pila.size() > 0)
- v = pila.extraer()**
- para cada ady de v:
- if(no visitado[ady])
- visitado[ady]=true
- pila.add(ady)**



# Grafos - DFS

- También se puede utilizar la pila de sistema (recursión) para recorrer el grafo

DFS( $v$ )

visitado[ $v$ ] = true

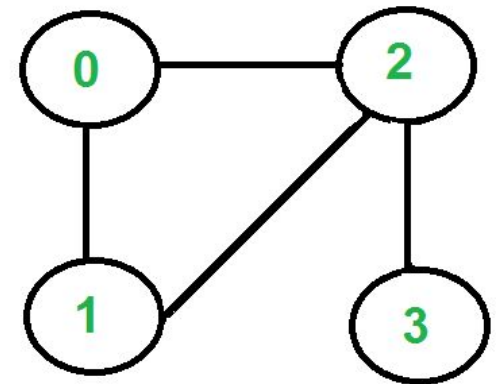
para cada ady de  $v$ :

si (no visitado[ady])

DFS(ady)

# Grafos - Eulerianos

- Camino euleriano
  - ¿Se puede recorrer un grafo de manera que solo se recorra cada una de sus aristas solo **una** vez?

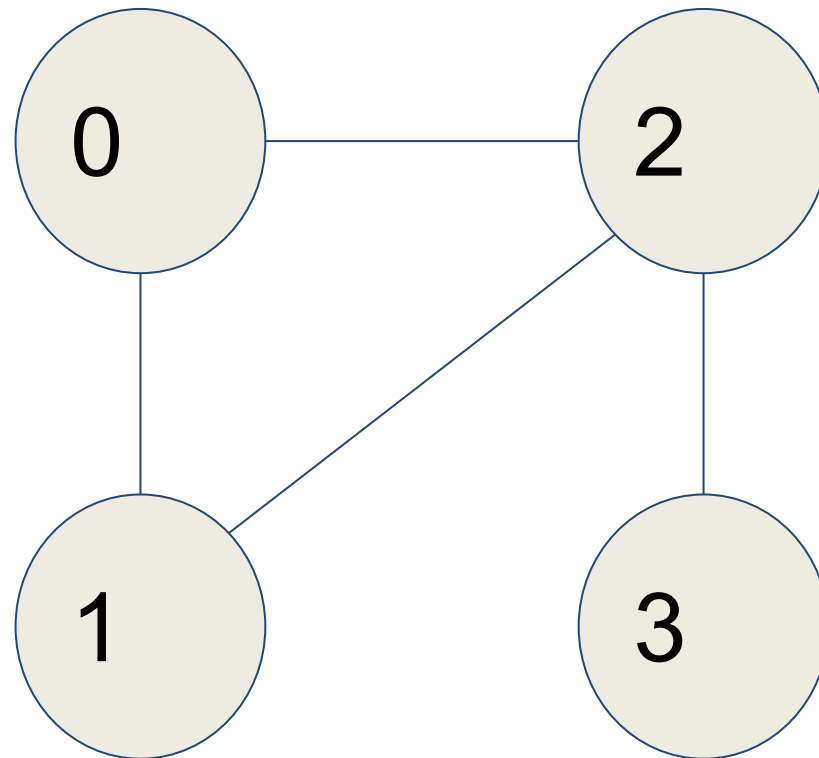




# Grafos - Eulerianos

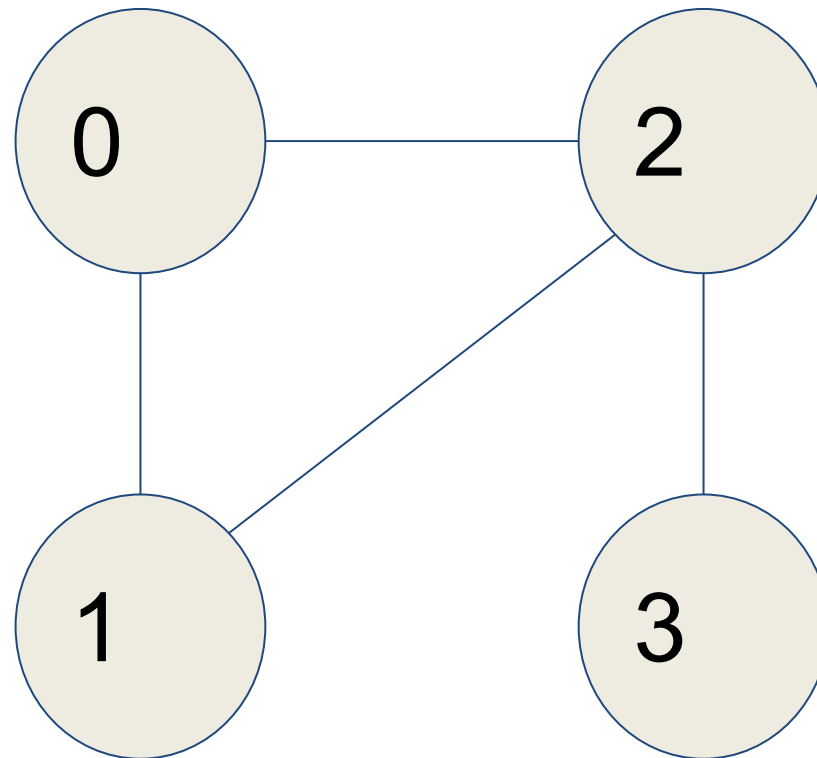
- Camino euleriano
  - Sea  $C(v)$  el número de aristas que tiene el vértice  $v$
  - Es camino euleriano si todos los vértices cumplen que  $C(v)$  es par
  - Ó si  $C(v)$  es impar en solo 2 vértices

# Grafos - Eulerianos



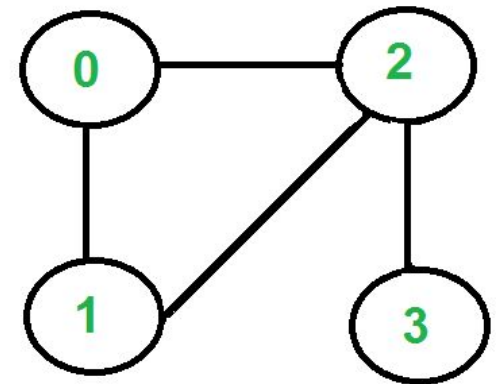
# Grafos - Eulerianos

3 -> 2  
2 -> 1  
1 -> 0  
0 -> 2



# Grafos - Eulerianos

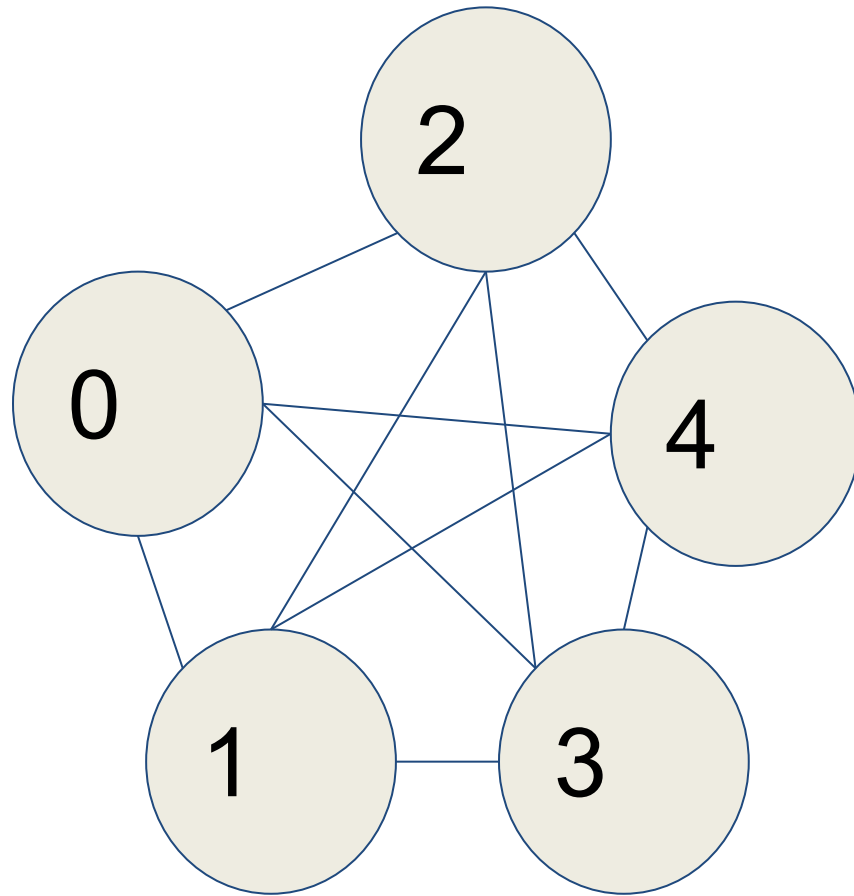
- Ciclo euleriano
  - ¿Se puede recorrer un grafo de manera que solo se recorra cada una de sus aristas solo **una** vez y llegar al mismo punto de inicio?



# Grafos - Eulerianos

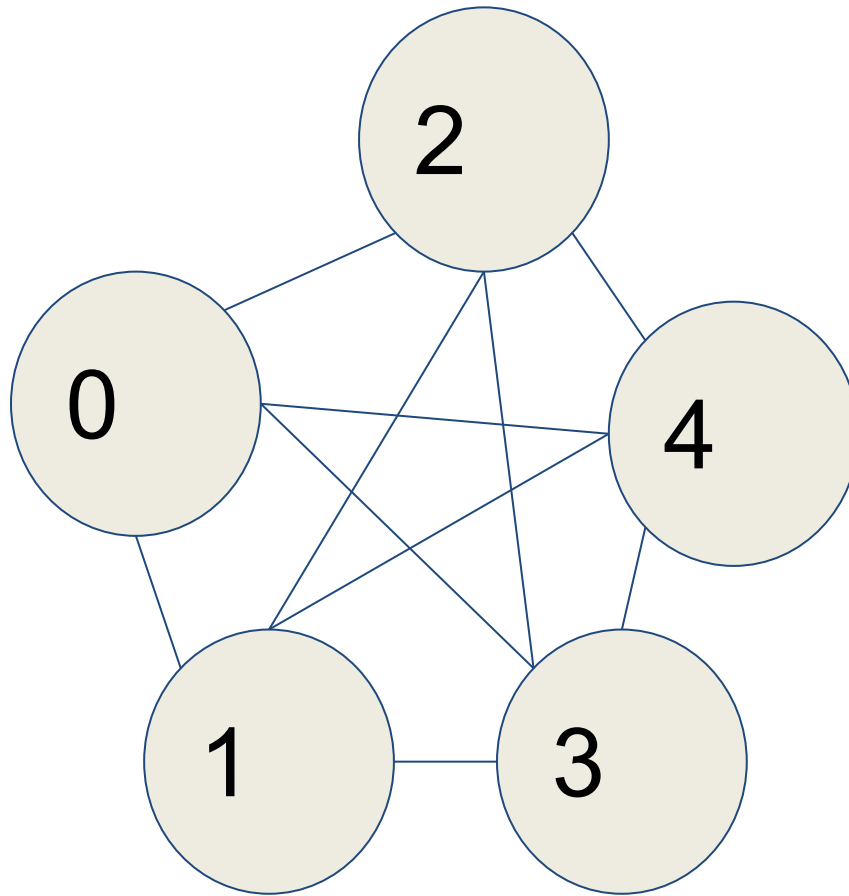
- Camino euleriano
  - Sea  $C(v)$  el número de aristas que tiene el vértice  $v$
  - Es camino euleriano si todos los vértices cumplen que  $C(v)$  es par

# Grafos - Eulerianos



# Grafos - Eulerianos

0 -> 1  
1 -> 2  
2 -> 3  
3 -> 4  
4 -> 1  
1 -> 3  
3 -> 0  
0 -> 2  
2 -> 4  
4 -> 0



# Grafos | Camino y ciclo Hamiltoniano

- Recorriendo solo una vez todos los nodos del grafo
  - (Camino) Recorrer todos los nodos del grafo
  - (Ciclo) Recorrer todos los nodos del grafo y llegar al mismo punto de inicio



# Grafos | Camino y ciclo Hamiltoniano

- Camino/Ciclo euleriano es trivial
- Hamiltoniano es un algoritmo NP-Completo
- Noción de máscara de bits
  - Un entero puede ser representado por hasta 32 bits
  - Utilizar el mismo principio para saber qué nodo hemos visitado

# Grafos | Camino y ciclo Hamiltoniano

- Si tuviésemos 3 nodos y estuviésemos interesados en chequear un ciclo hamiltoniano tendríamos que usar el entero 7 ( $2^3-1$ ), cuya representación de bits es 111
- Por cada nodo que visitemos lo marcamos a 0
  - `mask = 7 // 111`
  - `mask ^ (1 << i)` donde  $i$  es el nodo

# Grafos | Camino y ciclo Hamiltoniano

- Recordemos la tabla de verdad de XOR
  - $1 \text{ XOR } 1 = 0$
  - $0 \text{ XOR } 1 = 1$
  - $1 \text{ XOR } 0 = 1$
  - $0 \text{ XOR } 0 = 0$

# Grafos | Camino y ciclo Hamiltoniano

- Si queremos saber si el  $i$ -ésimo bit está encendido:
  - $\text{mask} \& (1 \ll i)$ 
    - i. Desplazamos 1  $i$  veces a la izquierda
    - ii. Para  $i=2$ ;  $100$  (4)
    - iii.  $7 \& 4 = 111 \& 100 \Rightarrow (4)$
    - iv. Mientras que  $\text{mask} \& (1 \ll i)$  no tome un valor de 0, el  $i$ -ésimo bit está encendido

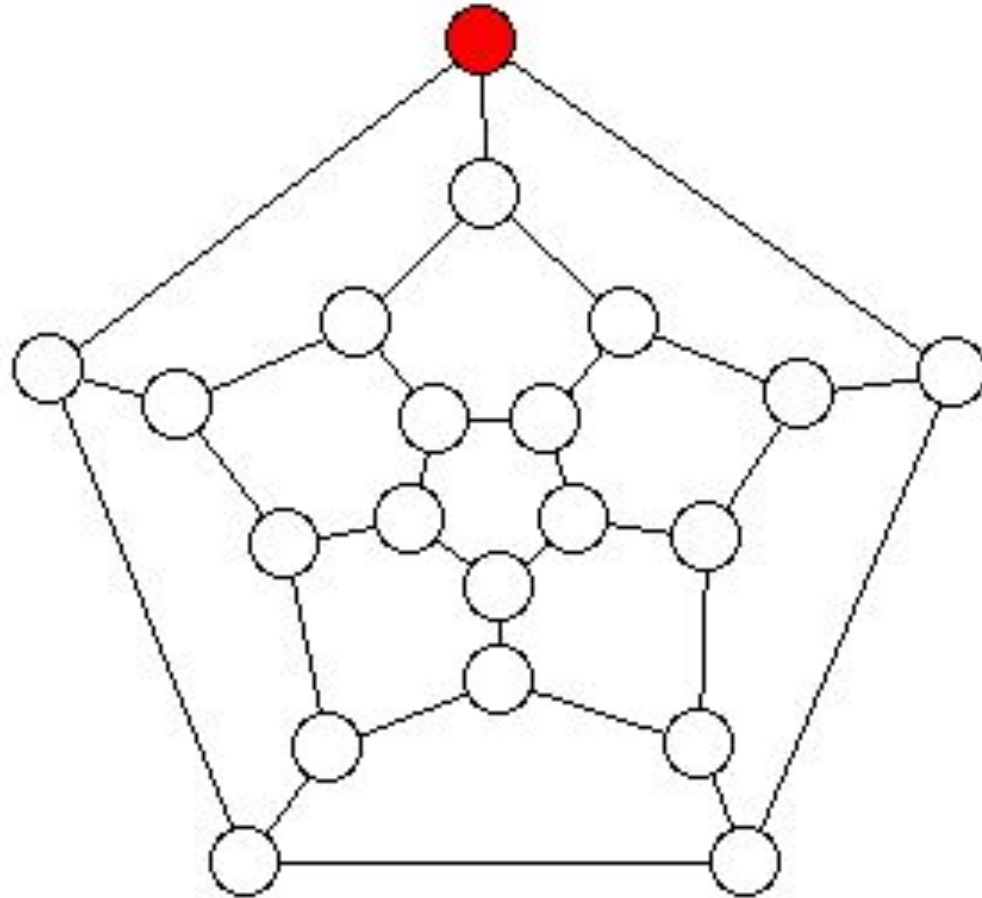
# Grafos | Camino y ciclo Hamiltoniano

- Si queremos convertir el  $i$ -ésimo bit a 0 (sabiendo que está encendido de antes):
  - $\text{mask} \wedge (1 \ll i)$ 
    - i. Desplazamos 1  $i$  veces a la izquierda
    - ii. Para  $i=2$ ;  $100$  (4)
    - iii.  $7 \wedge 4 = 111 \wedge 100 \Rightarrow (011)$

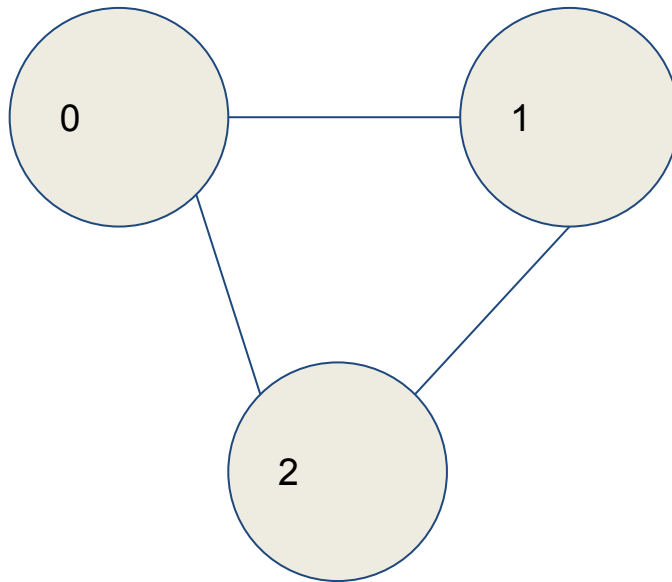
# Grafos | Camino y ciclo Hamiltoniano

- En caso del camino hamiltoniano si  $mask=0$  hemos terminado
- En caso de ciclo si  $mask=0$  tenemos que comprobar que el nodo actual es igual al nodo de inicio

# Grafos | Camino y ciclo Hamiltoniano



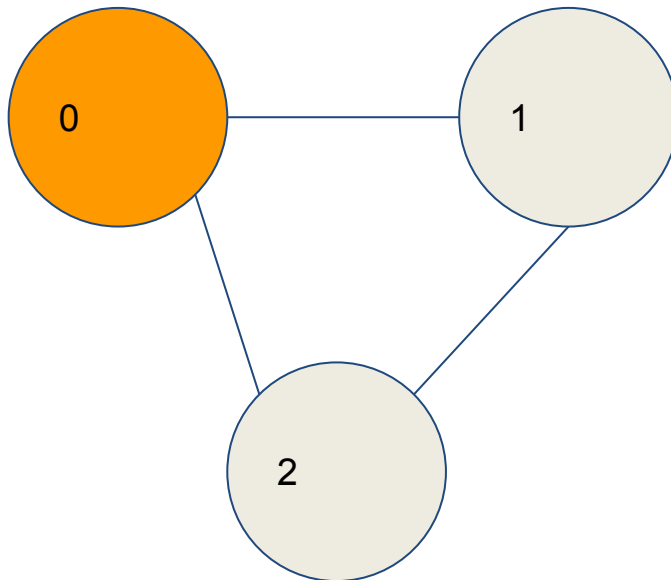
# Grafos | Camino y ciclo Hamiltoniano



**NODO=0,**  
**MASK=111<sub>2</sub>**

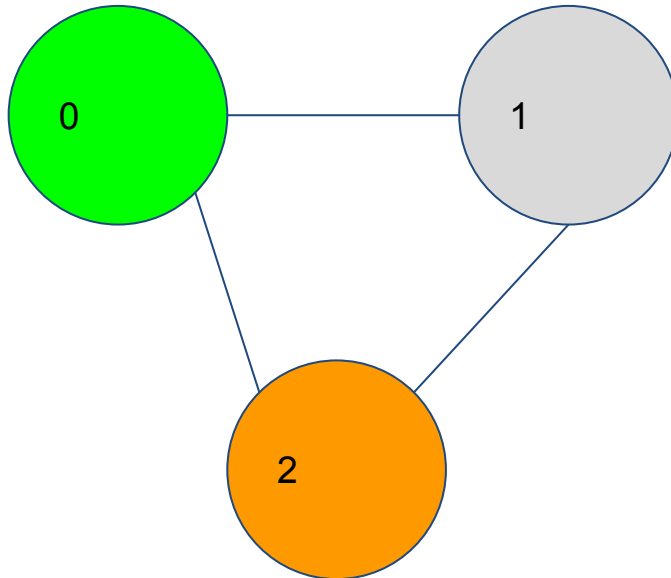


# Grafos | Camino y ciclo Hamiltoniano



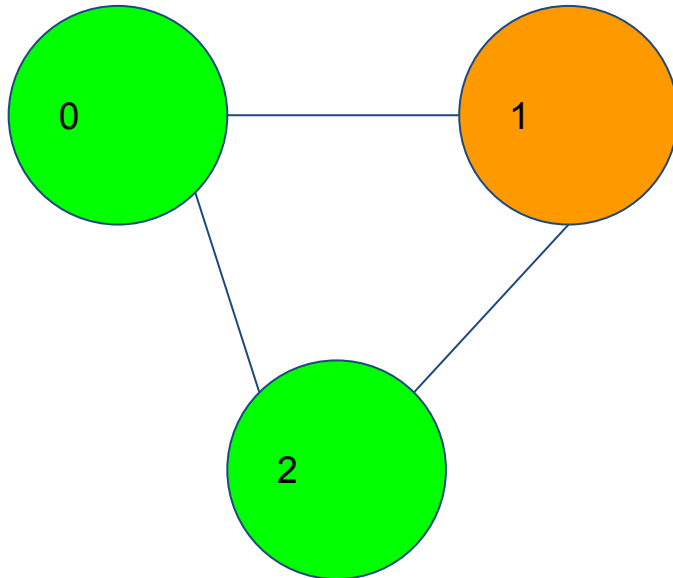
**NODO=0,**  
**MASK=111<sub>2</sub>**  
**2<sup>0</sup> = 1**  
**111 & 001 > 0?**

# Grafos | Camino y ciclo Hamiltoniano



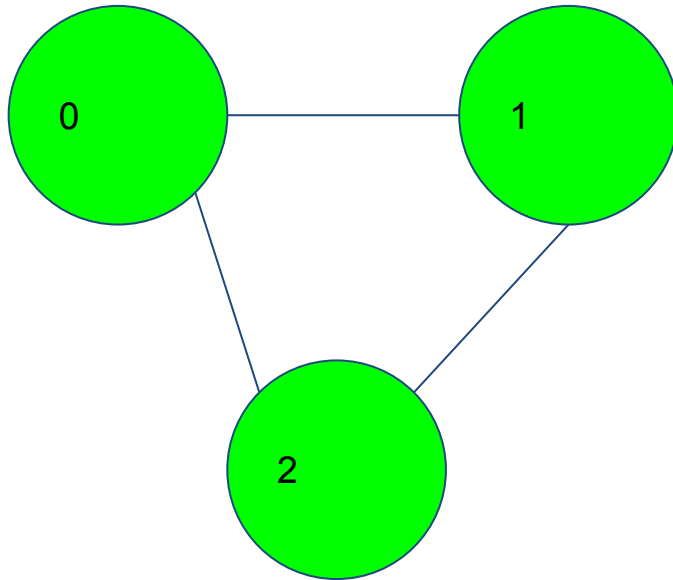
**NODO=2,**  
**MASK=110<sub>2</sub>**  
**2<sup>2</sup> = 4 (100)**  
**110 & 100 > 0?**

# Grafos | Camino y ciclo Hamiltoniano



**NODO=1,**  
**MASK=010<sub>2</sub>**  
**2<sup>1</sup> = 2 (010)**  
**010 & 010 > 0?**

# Grafos | Camino y ciclo Hamiltoniano



**NODO=X,**  
**MASK=000<sub>2</sub>**  
**Es camino**  
**Hamiltoniano**

¿Es también ciclo?

# Grafos | Bipartito

- Es un grafo  $G$  que se puede separar en dos conjuntos  $U$  y  $V$  tal que la unión de  $U$  y  $V$  son todos los nodos y la intersección de los mismos es vacío
  - ¿Un árbol es bipartito?

# Grafos | Bipartito

- Para probar que un grafo es bipartito solo falta hacer un recorrido dentro del grafo y “pintarlo” con uno de los dos colores, llevando siempre en cuenta que debes cambiar el color por nodo vecino
- Si encuentras un nodo vecino que tenga el mismo color que el actual, ¡no puede ser un grafo bipartito!

# Grafos | Bipartito

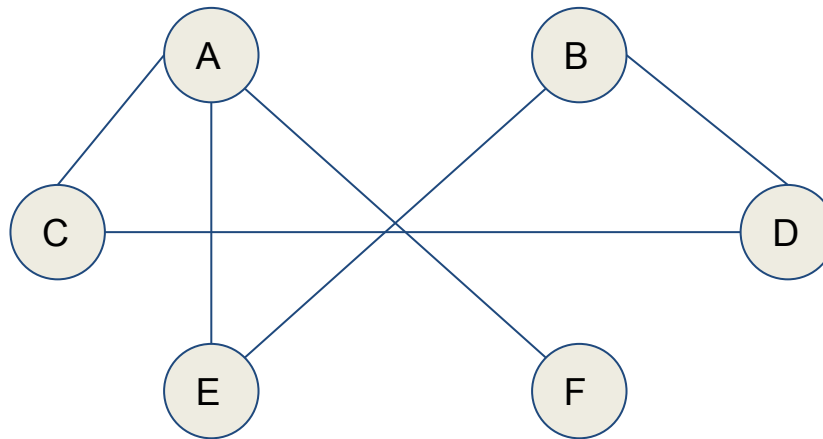
Con BFS

iniciando en A

$V = [0, 0, 0, 0, 0, 0]$

$C = [0, 0, 0, 0, 0, 0]$

$Q = [A]$



# Grafos | Bipartito

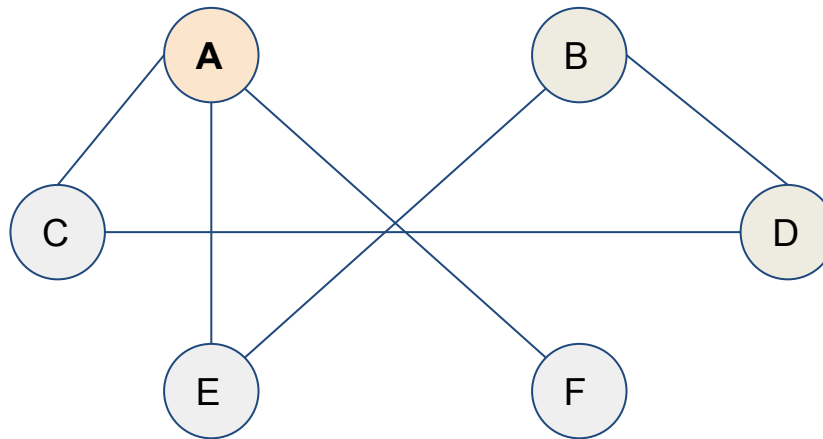
Con BFS

iniciando en A

$V = [1, 0, 0, 0, 0, 0]$

$C = [1, 0, 0, 0, 0, 0]$

$Q = [A]$





# Grafos | Bipartito

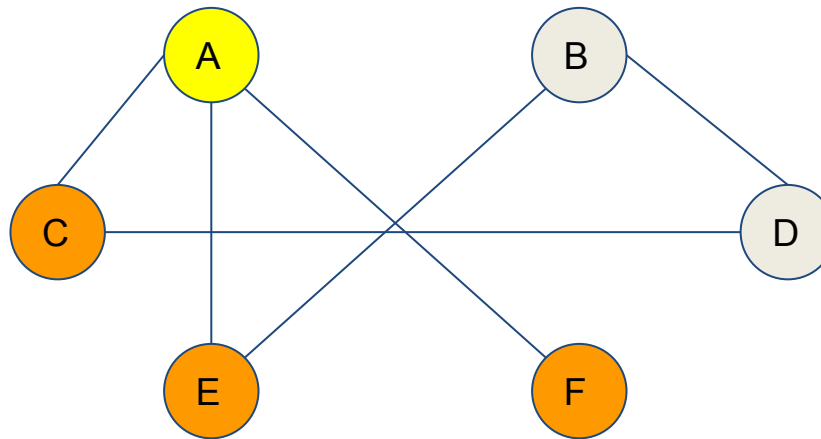
Con BFS

iniciando en A

$V = [1, 0, 0, 0, 0, 0]$

$C = [1, 0, 2, 0, 2, 2]$

$Q = [C, E, F]$



# Grafos | Bipartito

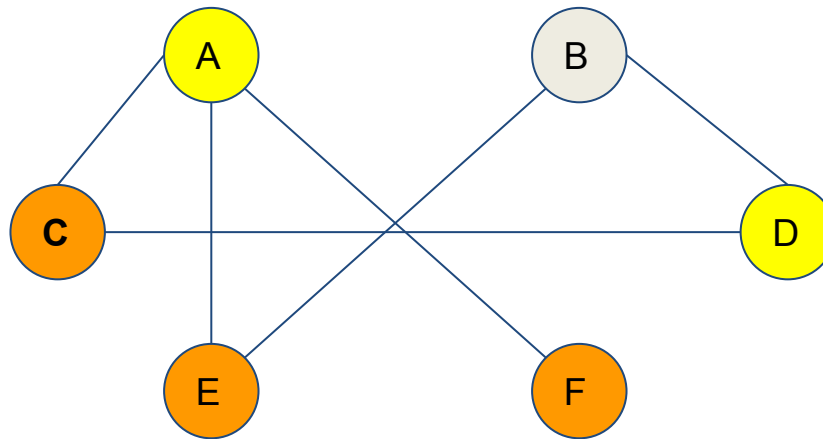
Con BFS

iniciando en A

$V = [1, 0, 1, 0, 0, 0]$

$C = [1, 0, 2, 1, 2, 2]$

$Q = [E, F, D]$



# Grafos | Bipartito

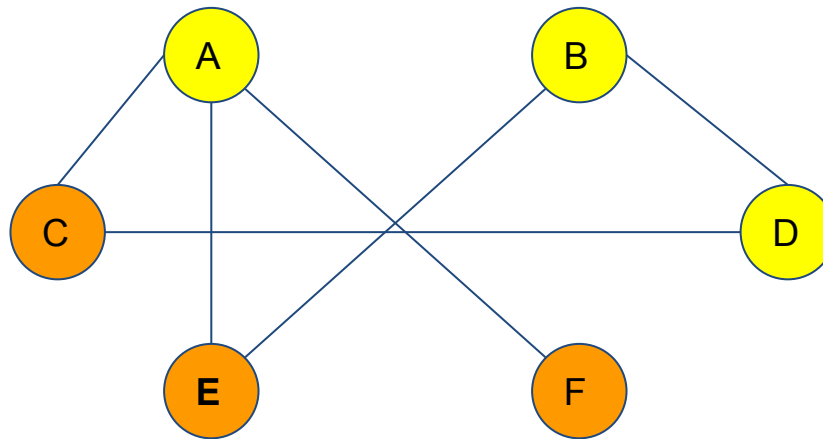
Con BFS

iniciando en A

$V = [1, 0, 1, 0, 1, 0]$

$C = [1, 1, 2, 1, 2, 2]$

$Q = [F, D, B]$



# Grafos | Bipartito

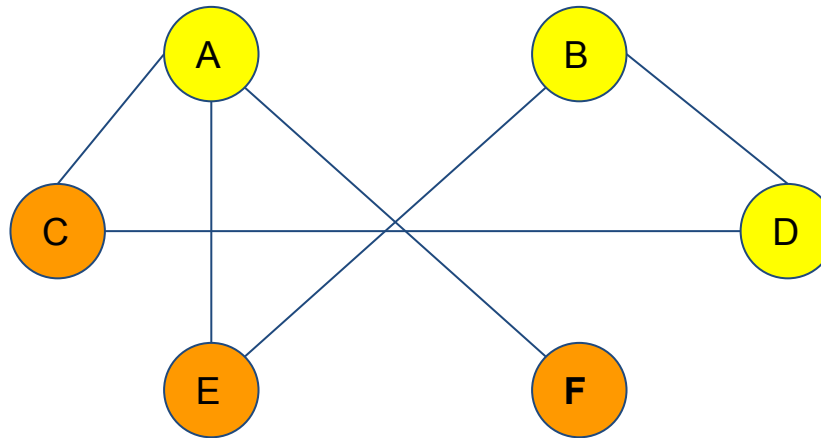
Con BFS

iniciando en A

$V = [1, 0, 1, 0, 1, 1]$

$C = [1, 1, 2, 1, 2, 2]$

$Q = [D, B]$



# Grafos | Bipartito

Con BFS

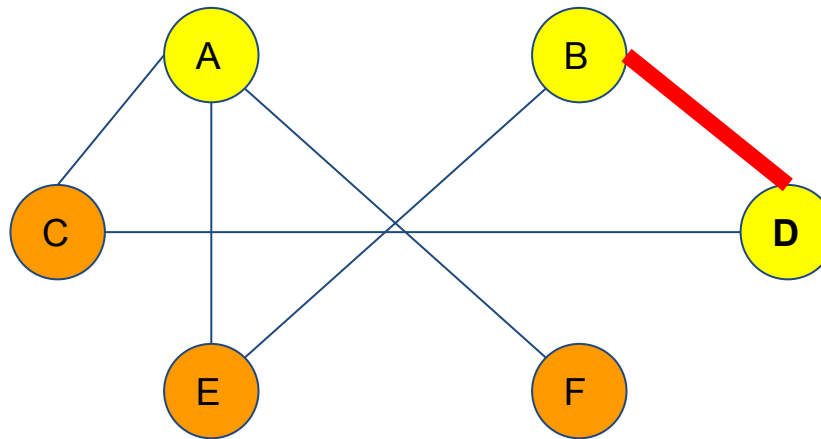
iniciando en A

$V = [1, 0, 1, 1, 1, 1]$

$C = [1, 1, 2, 1, 2, 2]$

$Q = [B]$

No es bipartito!



# Grafos | Bipartito

- Pseudocódigo

```
isBipartite(init):
    q = [(init, 1)]
    v[init] = t, c[init] = 1
    while !q.empty():
        current = q.top(), q.pop()
        for edge in edges(current):
            neighbor_color = current.c == 1 ? 2 : 1
            if c[edge.dest] == current.c:
                return f
            if !v[edge.dest]:
                v[edge.dest] = t
                c[edge.dest] = neighbor_color
                q.push((edge.dest, neighbor_color))
    return t
```

# Grafos | Componentes Conexas

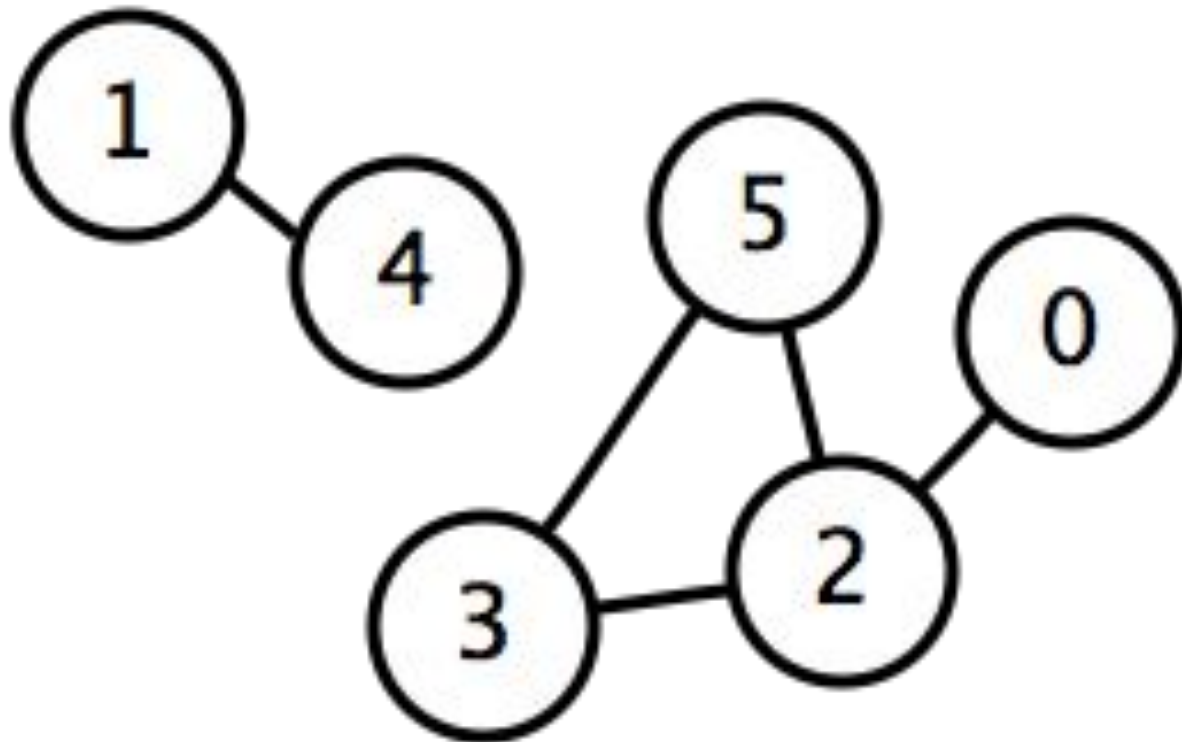
- Es un subgrafo donde dos vértices cualesquiera están conectados a través de uno o más caminos
- Se parte de un grafo no dirigido

# Grafos | Componentes Conexas

- La idea básica es hacer un BFS/DFS por cada vértice  $i$  desde  $0..N$  siempre y cuando  $i$  no haya sido recorrido por un algoritmo anterior
- Se cuenta 1 y se recorre  $i$

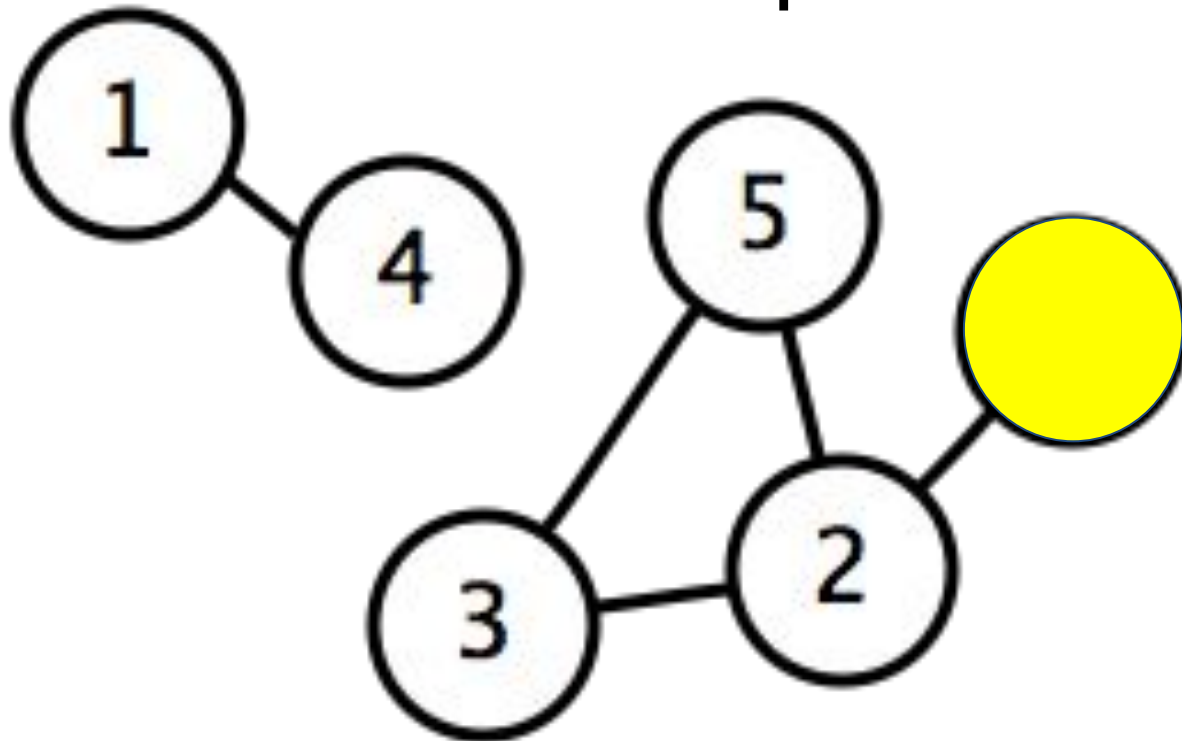


# Grafos | Componentes Conexas

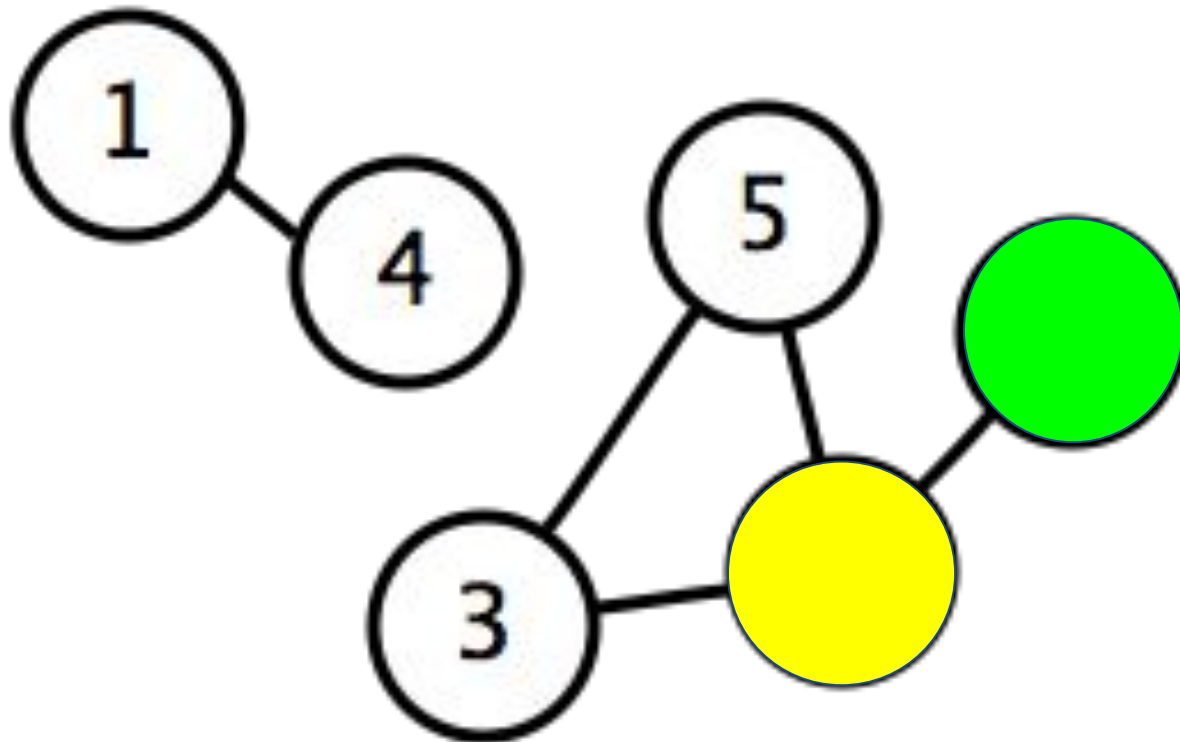


# Grafos | Componentes Conexas

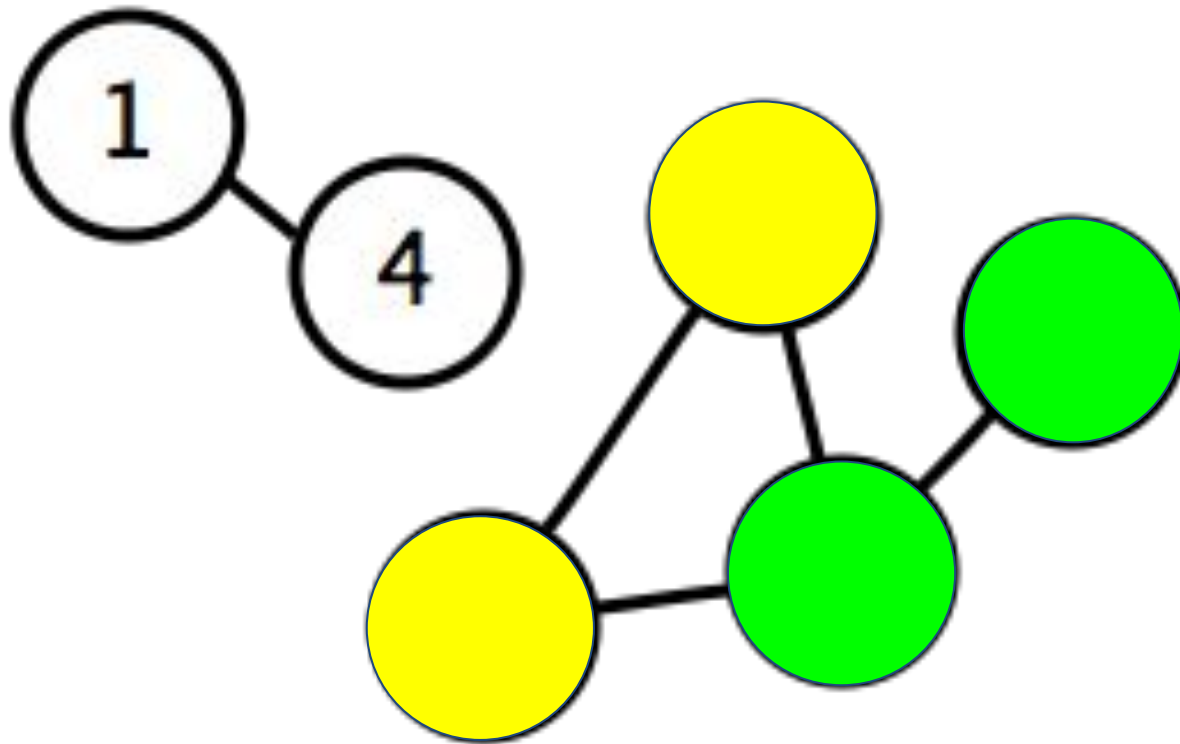
**+1 Componente Conexa (1)**



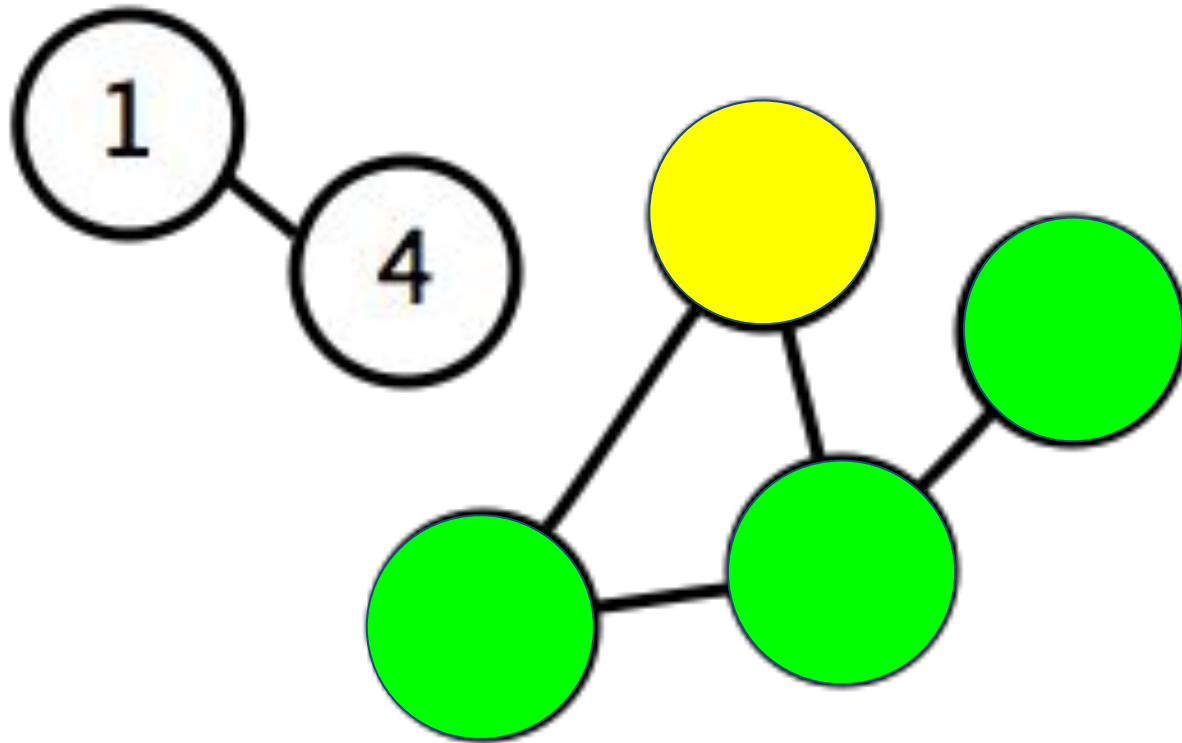
# Grafos | Componentes Conexas



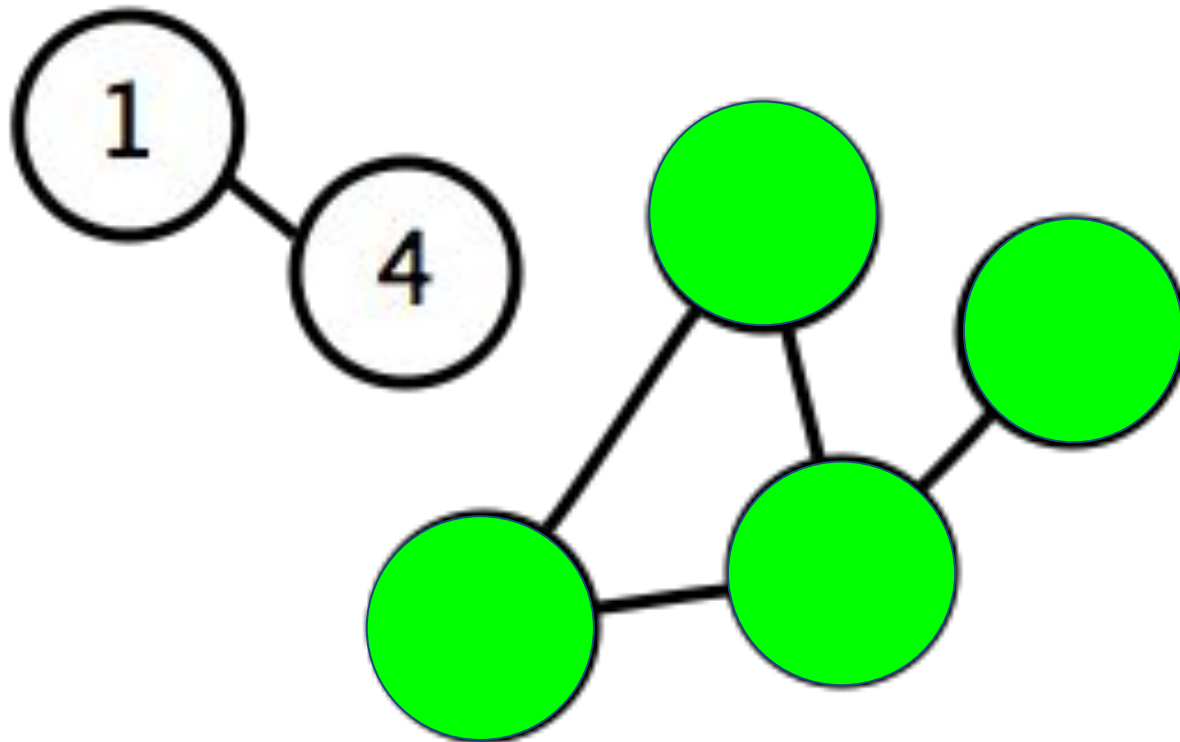
# Grafos | Componentes Conexas



# Grafos | Componentes Conexas

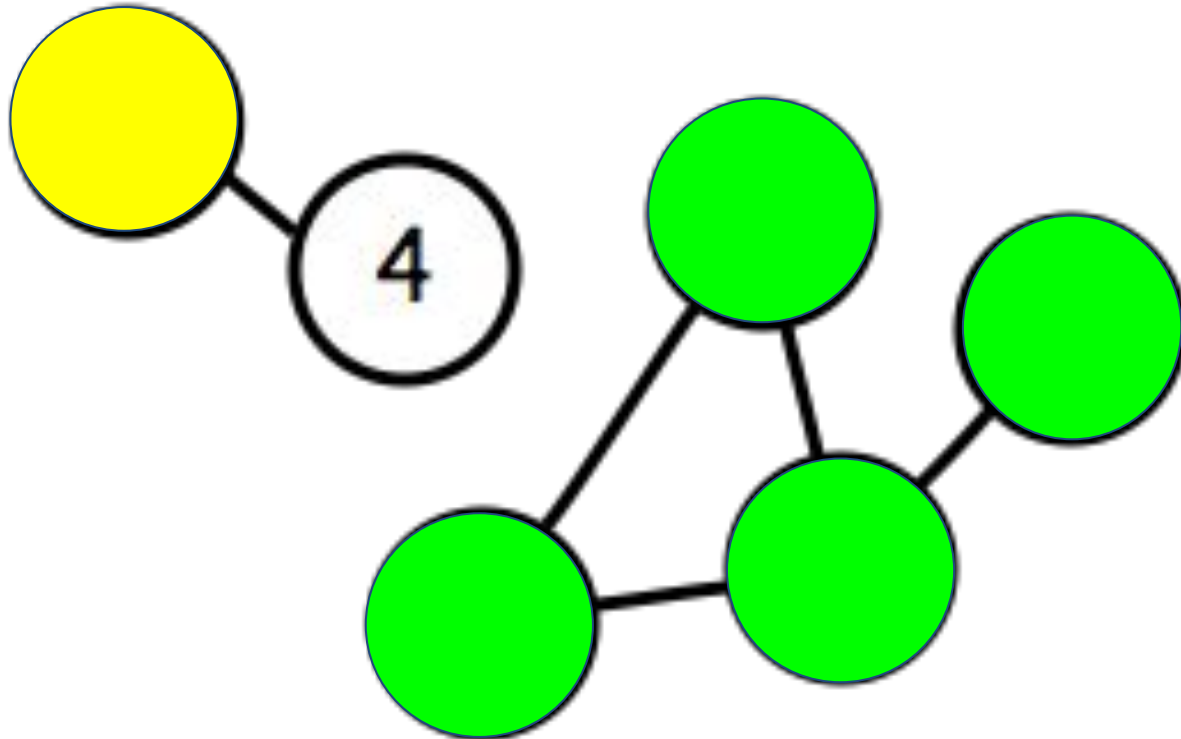


# Grafos | Componentes Conexas

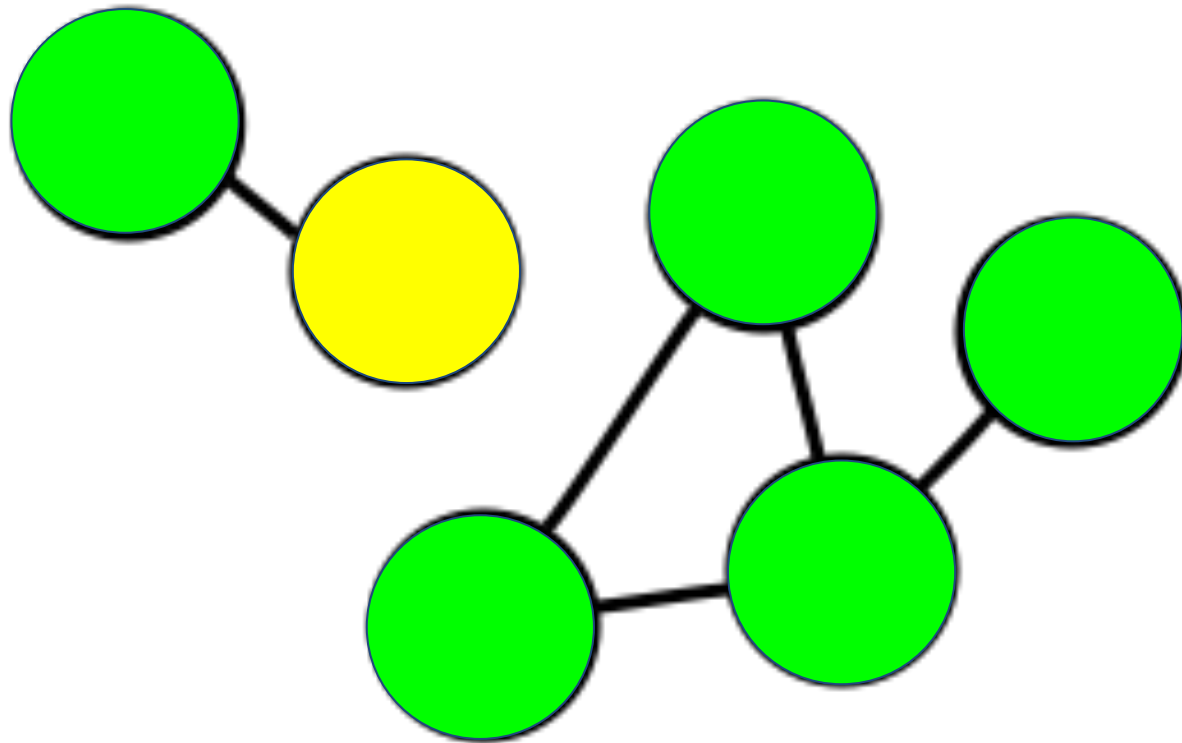


# Grafos | Componentes Conexas

**+1 Componente Conexo (2)**

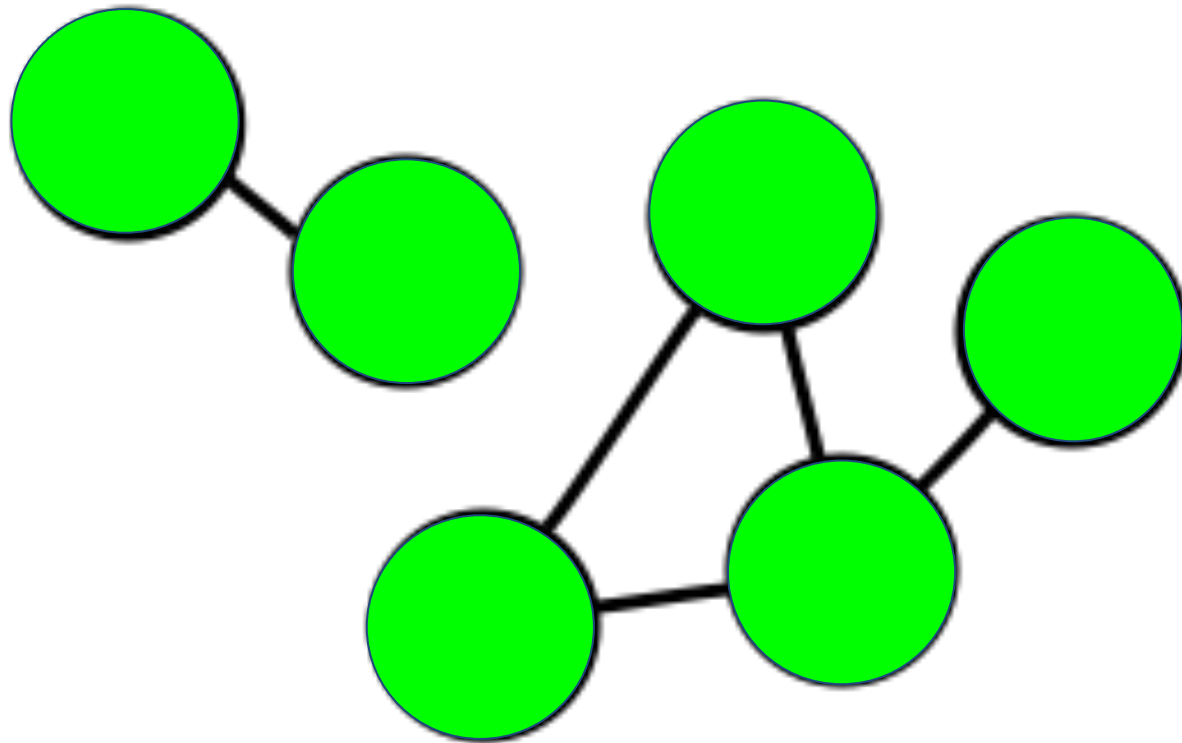


# Grafos | Componentes Conexas





# Grafos | Componentes Conexas



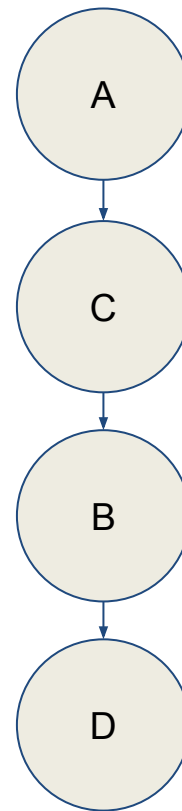
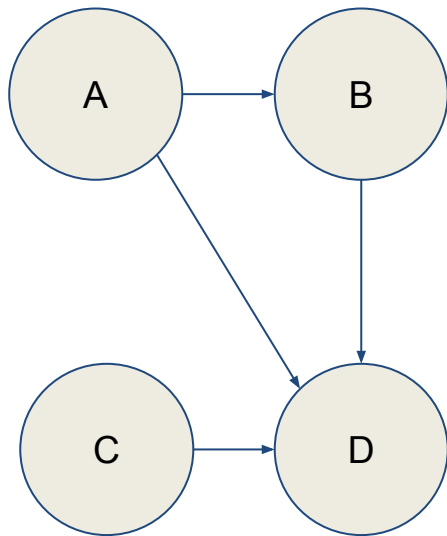
# Grafos | Componentes Conexas

- Se empieza desde 0 y se recorre:
  - 0-2-5-3
- Al estar 1 no visitado se recorre los adyacentes de 1
  - 1-4
- Al estar 2,3,4,5 no visitado se ignoran
- Hay dos componentes conexas

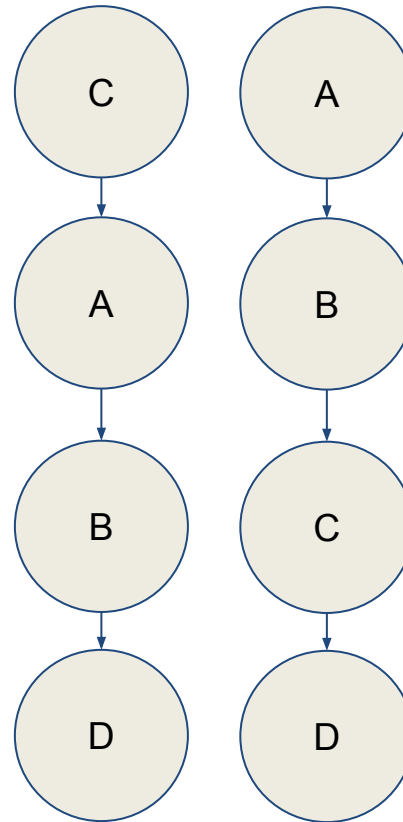
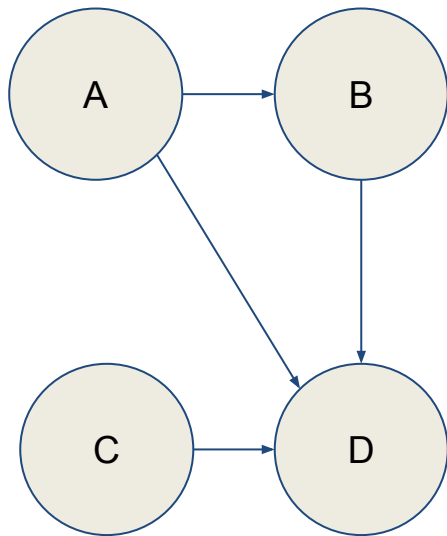
# Grafos | Ordenamiento Topológico

- Sobre un grafo acíclico dirigido (DAG)
- Ordenar nodos tal que para cualquier nodo  $u, v$ ; al momento de eliminar  $u$ ,  $v$  no contenga aristas hacia ella
- Utilizar el algoritmo de DFS

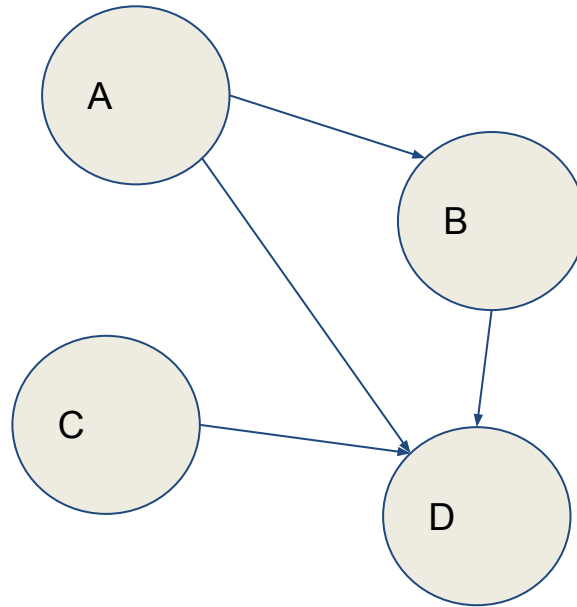
# Grafos | Ordenamiento Topológico



# Grafos | Ordenamiento Topológico

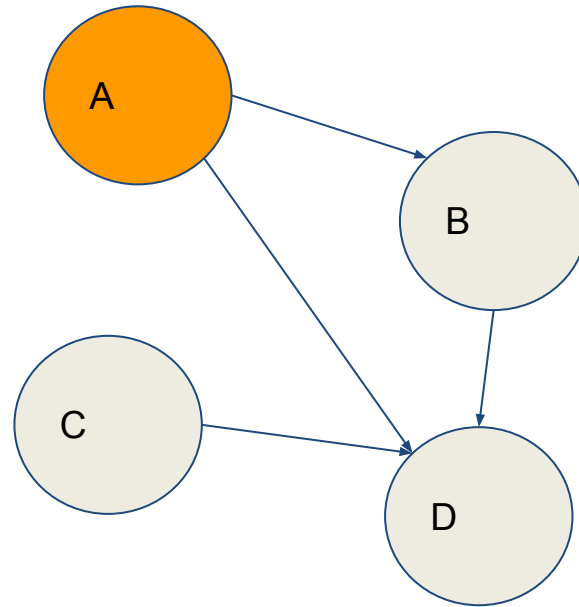


# Grafos | Ordenamiento Topológico



**A y C no les incide ningún otro nodo**

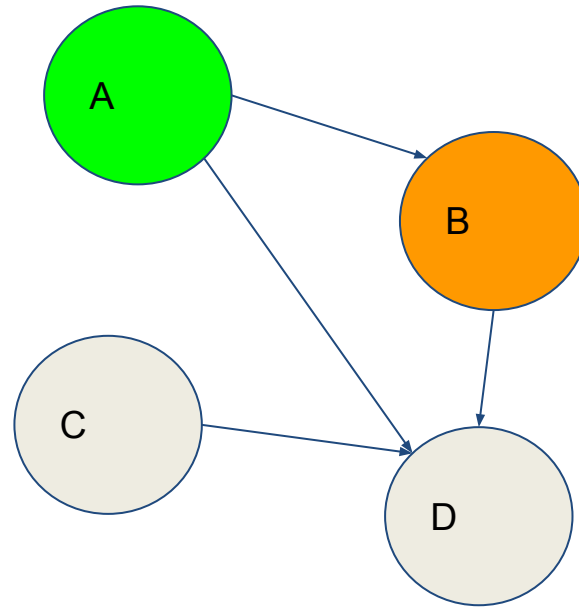
# Grafos | Ordenamiento Topológico



**TS = { }**

**DFS(A) = DFS(B), DFS(D)**

# Grafos | Ordenamiento Topológico

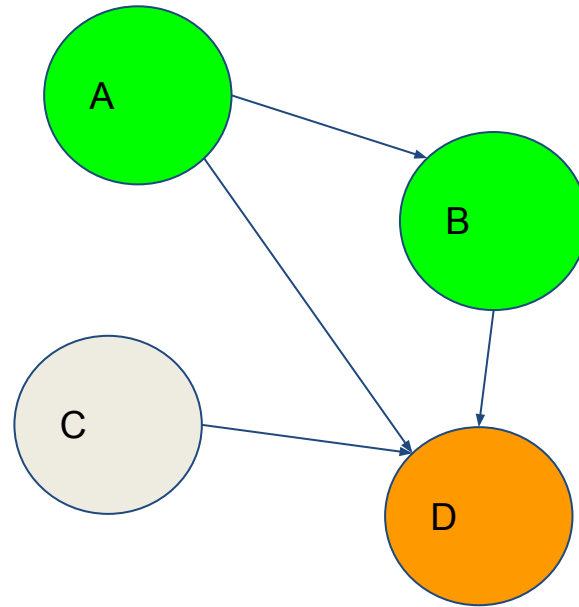


**TS = { }**

**DFS(B) = DFS(D)**



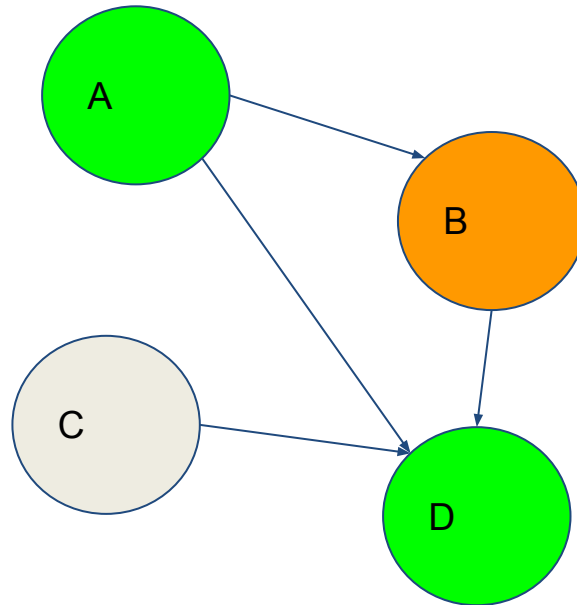
# Grafos | Ordenamiento Topológico



**TS = {D}**

**DFS(D) =  $\emptyset$ , TS.push(D)**

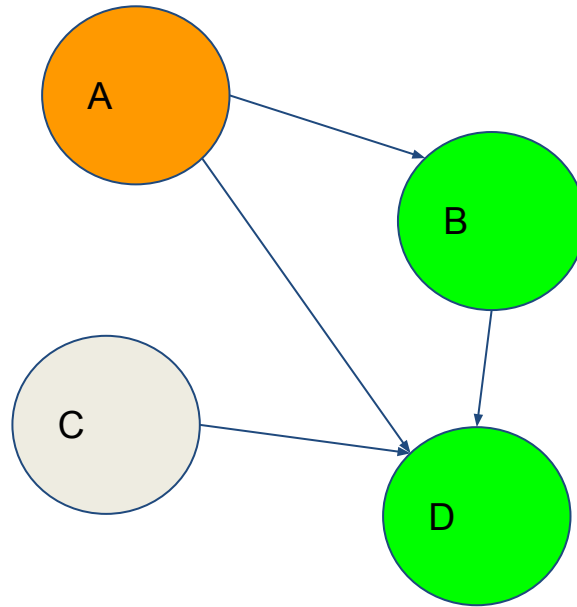
# Grafos | Ordenamiento Topológico



**TS = { B, D }**

**DFS(B) = DFS(D), TS.push(B)**

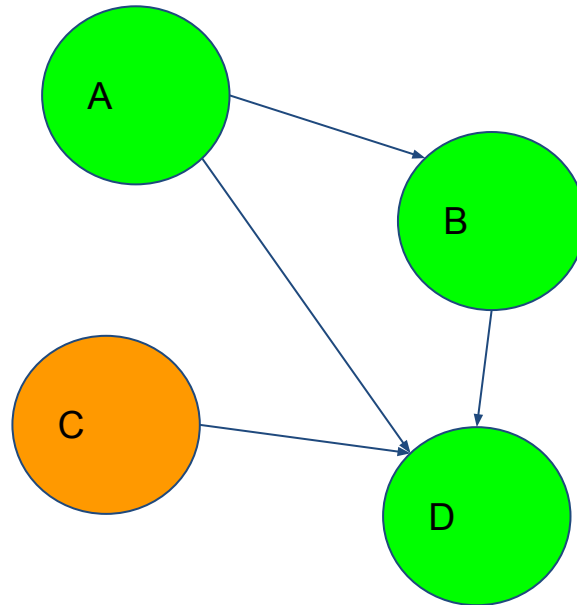
# Grafos | Ordenamiento Topológico



**TS = { A, B, D }**

**DFS(A) = DFS(B), DFS(D), TS.push(A)**

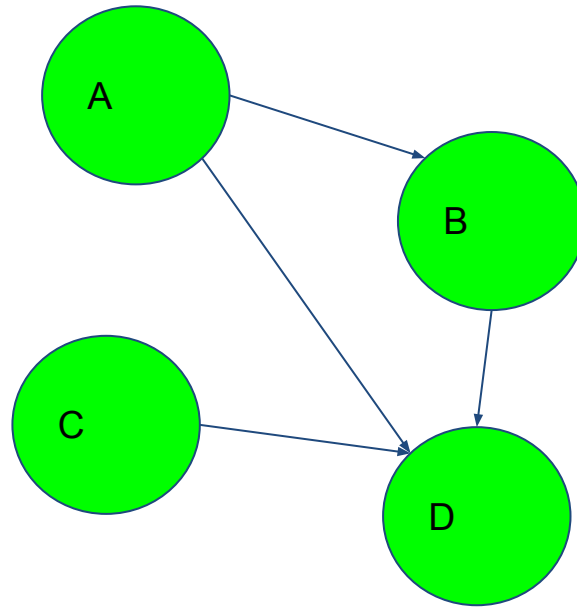
# Grafos | Ordenamiento Topológico



**TS = { C, A, B, D }**

**DFS(C) = DFS(D), TS.push(C)**

# Grafos | Ordenamiento Topológico



**TS = { C, A, B, D }**

**A y C pueden ir en cualquier orden, por lo que los toposort son  $\Rightarrow$  {C,A,B,D} ó {A,C,B,D}**

# Grafos | Ordenamiento Topológico

- Pseudocódigo

```
TopoSort (n) :
```

```
  si v(n) ret  $\emptyset$ 
```

```
  v(n) = 1
```

```
  for edge in edges (n) :
```

```
    TopoSort (edge)
```

```
  TS.push (n)
```

```
  ret  $\emptyset$ 
```

# Grafos | Ordenamiento Topológico

- Pseudocódigo
- Siendo TS una pila
- Siendo  $\text{edges}(N)$  una lista de vértices a quien incide N ( $N \rightarrow X$ )
- Siendo  $V(N)$  un array de booleanos donde se entiende que si  $V(N) = 1$  ya ha pasado un recorrido en profundidad por el nodo N

# Grafos | Puntos de Articulación

- Dado un grafo, nos interesa saber que nodos, de ser removidos, hacen que el número de componentes conexas del grafo se incremente
- Idea ingenua: Contar componentes conexas “haciendo como si no existe” cualquier vértice  $u$  del grafo  $G=\{V,E\}$



# Grafos | Puntos de Articulación

- Esta idea es  $O(N^2)$  con  $N =$  nodos del grafo

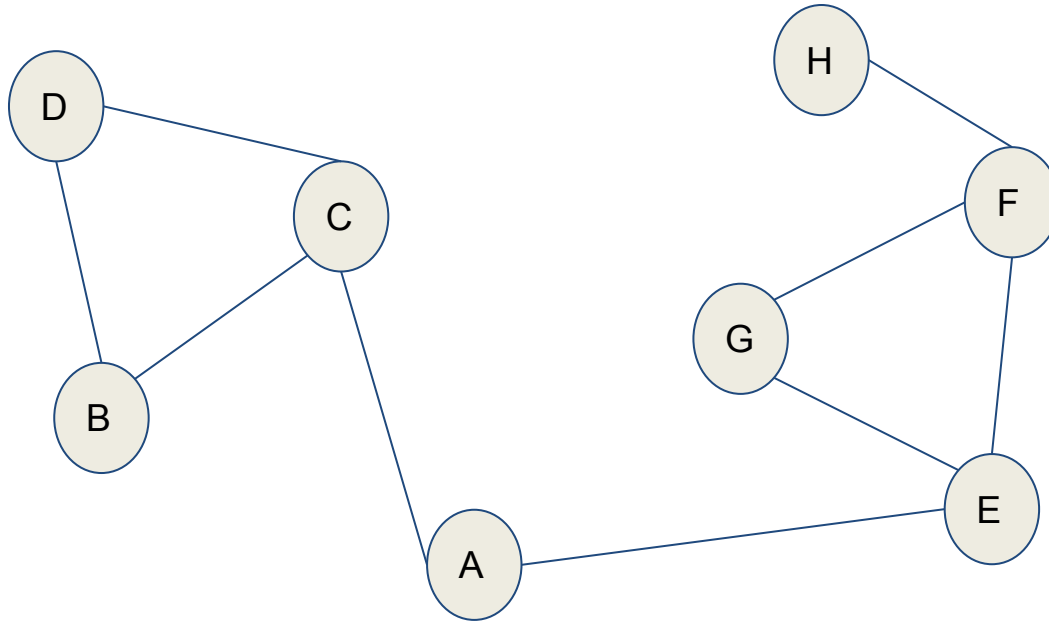
# Grafos | Puntos de Articulación

- Algoritmo de Tarjan
  - Tomamos un nodo cualquiera y recorremos a través de DFS ese nodo
  - Llevamos cuenta de la unidad de tiempo en el que llegamos a cualquier nodo “v” (visitar 1 nodo suma 1 al tiempo). Llamamos a esto, “discovery”
  - Con esta misma idea podemos observar si se puede llegar a cualquier nodo por otro “camino”. Llamamos a esto, “lowest”

# Grafos | Puntos de Articulación

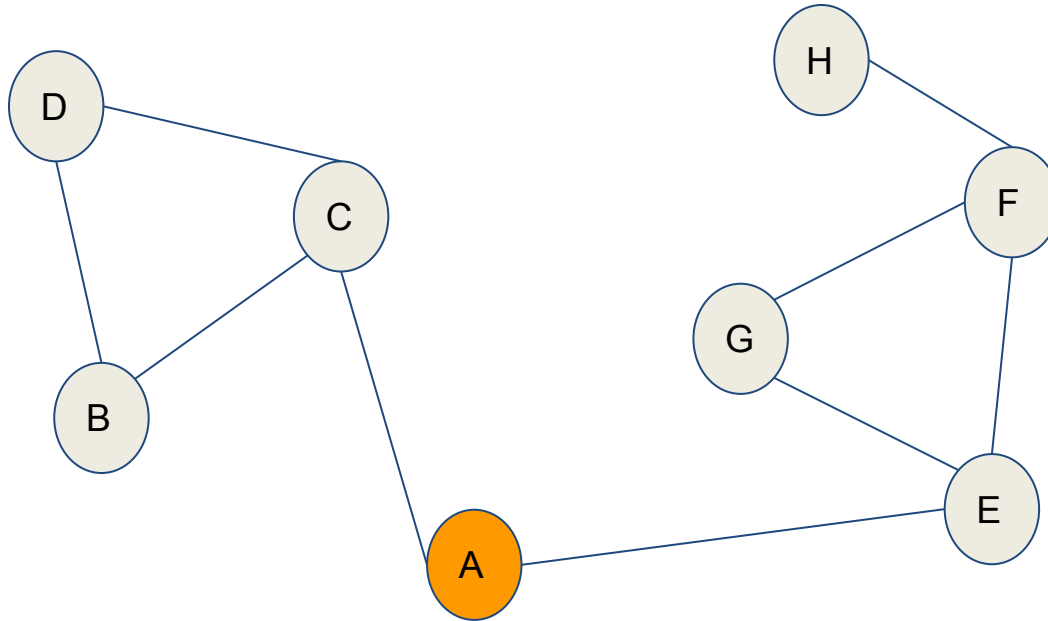
- Algoritmo de Tarjan
  - Condiciones para punto de articulación en  $u$ :
    - i. El nodo raíz  $u$  (inicio) hizo más de un DFS hacia sus vecinos
    - ii. Cualquier nodo no-raíz  $v$  conectado a  $u$  tiene un mínimo de tiempo (lowest) mayor o igual al tiempo que se descubrió (discover) un nodo  $u$

# Grafos | Puntos de Articulación



DFS

# Grafos | Puntos de Articulación



DFS\_STACK = (A, 1)

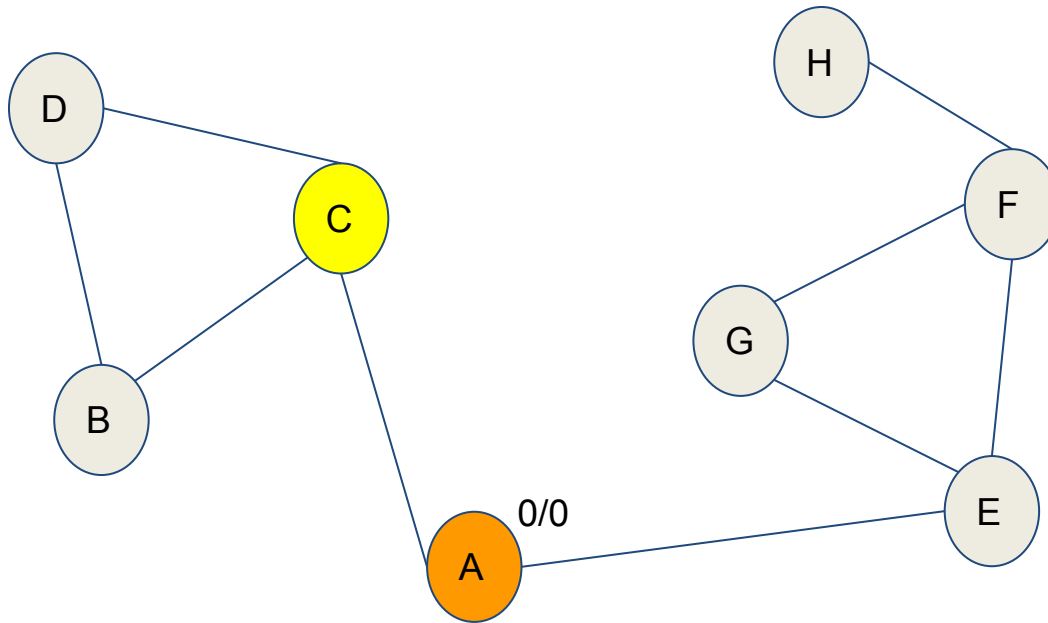
LOWEST = { 0, -1, -1, -1, -1, -1, -1, -1 }

DISCOVERY = { 0, -1, -1, -1, -1, -1, -1, -1 }

VISITED = { 1, 0, 0, 0, 0, 0, 0, 0 }

PARENTS = { -1, -1, -1, -1, -1, -1, -1, -1 }

# Grafos | Puntos de Articulación



DFS\_STACK = (C, 2)

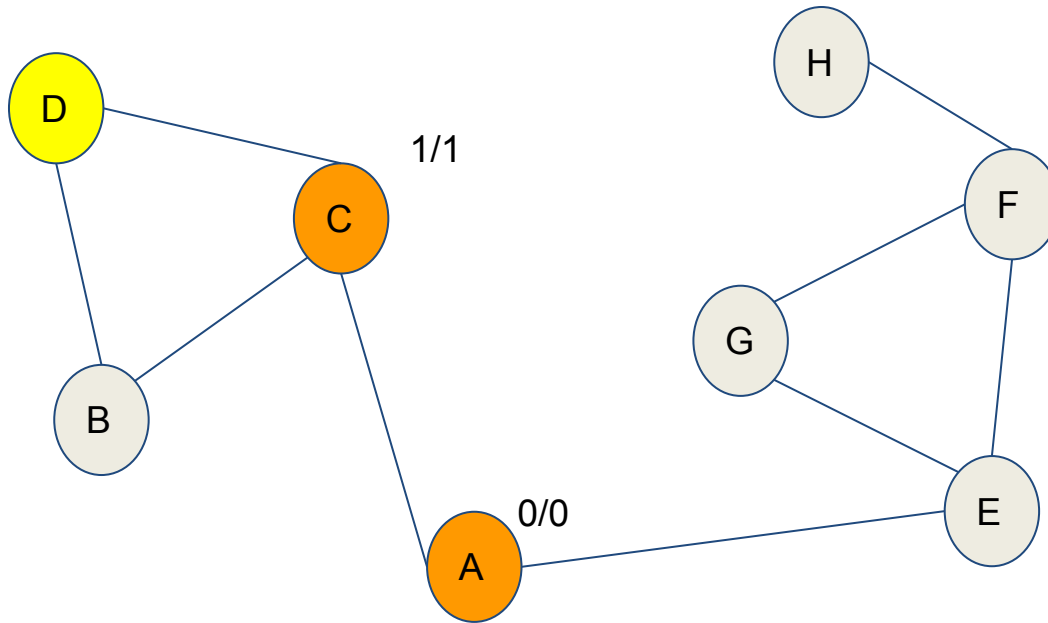
LOWEST = { 0, -1, 1, -1, -1, -1, -1, -1 }

DISCOVERY = { 0, 1, 1, -1, -1, -1, -1, -1 }

VISITED = { 1, 0, 1, 0, 0, 0, 0, 0 }

PARENTS = { -1, -1, 0, -1, -1, -1, -1, -1 }

# Grafos | Puntos de Articulación



DFS\_STACK = (D, 3)

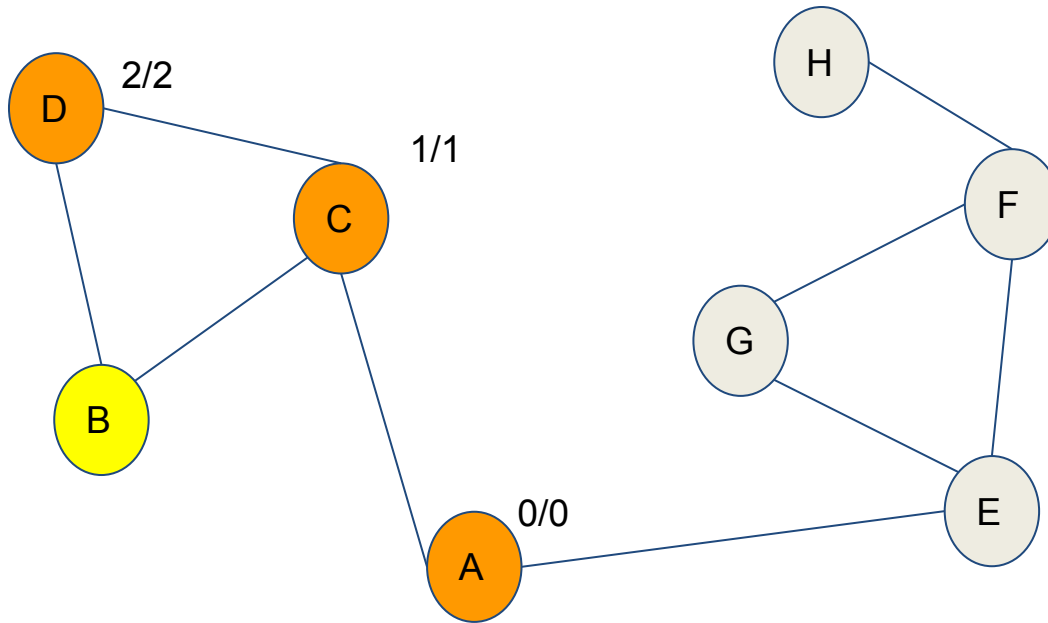
LOWEST = { 0, -1, 1, 2, -1, -1, -1, -1 }

DISCOVERY = { 0, 1, 1, 2, -1, -1, -1, -1 }

VISITED = { 1, 0, 1, 1, 0, 0, 0, 0 }

PARENTS = { -1, -1, 0, 2, -1, -1, -1, -1 }

# Grafos | Puntos de Articulación



DFS\_STACK = (B, 4)

LOWEST = { 0, 3, 1, 2, -1, -1, -1, -1 }

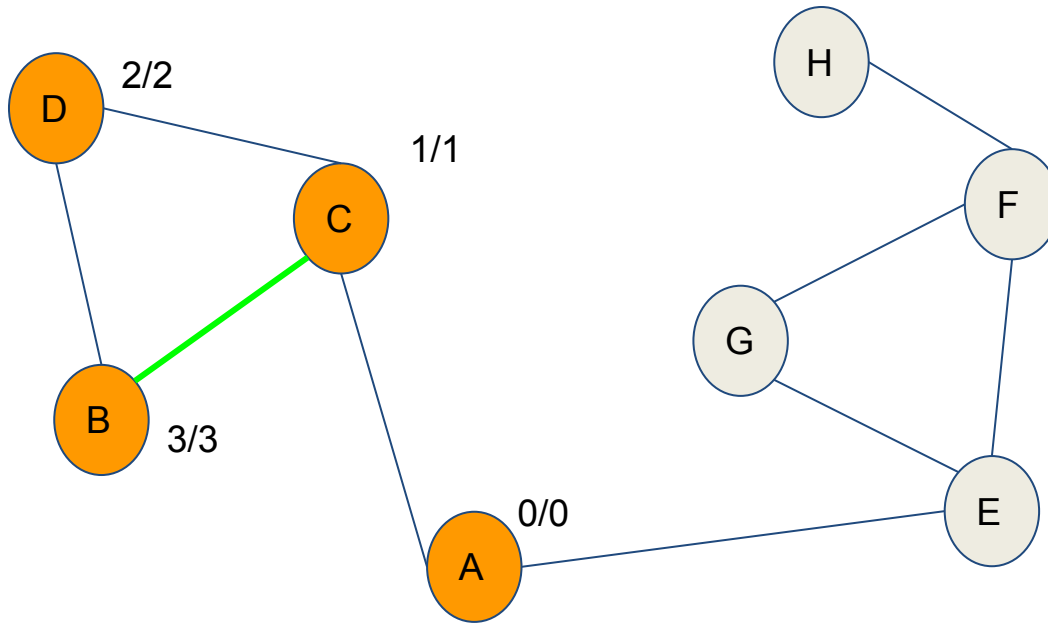
DISCOVERY = { 0, 3, 1, 2, -1, -1, -1, -1 }

VISITED = { 1, 1, 1, 1, 0, 0, 0, 0 }

PARENTS = { -1, 3, 0, 2, -1, -1, -1, -1 }



# Grafos | Puntos de Articulación



DFS\_STACK = (B, 4)

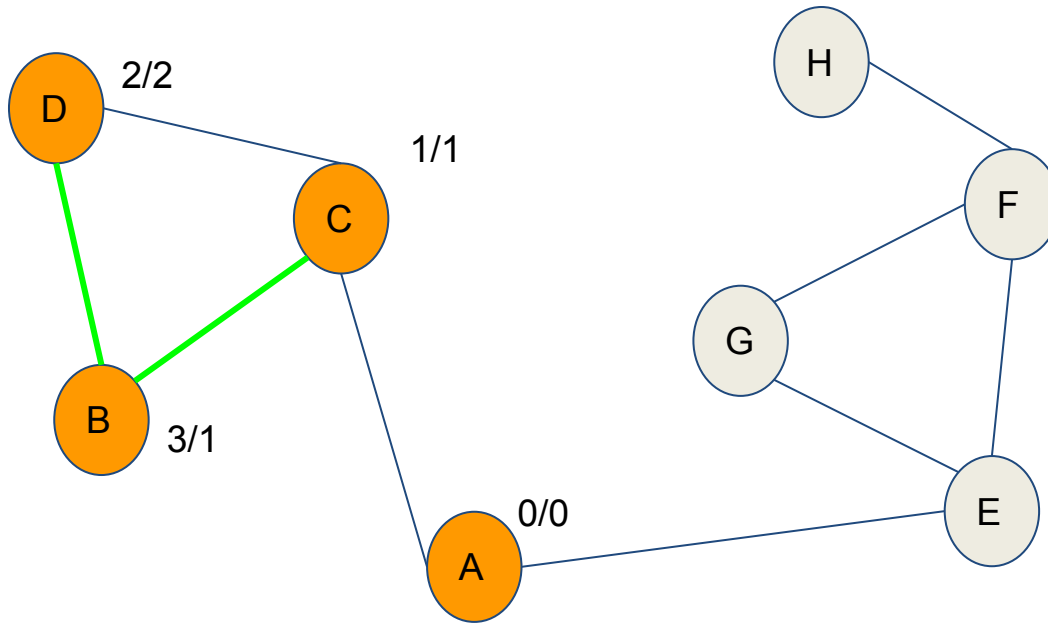
LOWEST = { 0, 3, 1, 2, -1, -1, -1, -1 }

DISCOVERY = { 0, 3, 1, 2, -1, -1, -1, -1 }

VISITED = { 1, 1, 1, 1, 0, 0, 0, 0 }

PARENTS = { -1, 3, 0, 2, -1, -1, -1, -1 }

# Grafos | Puntos de Articulación



DFS\_STACK = (B, 4)

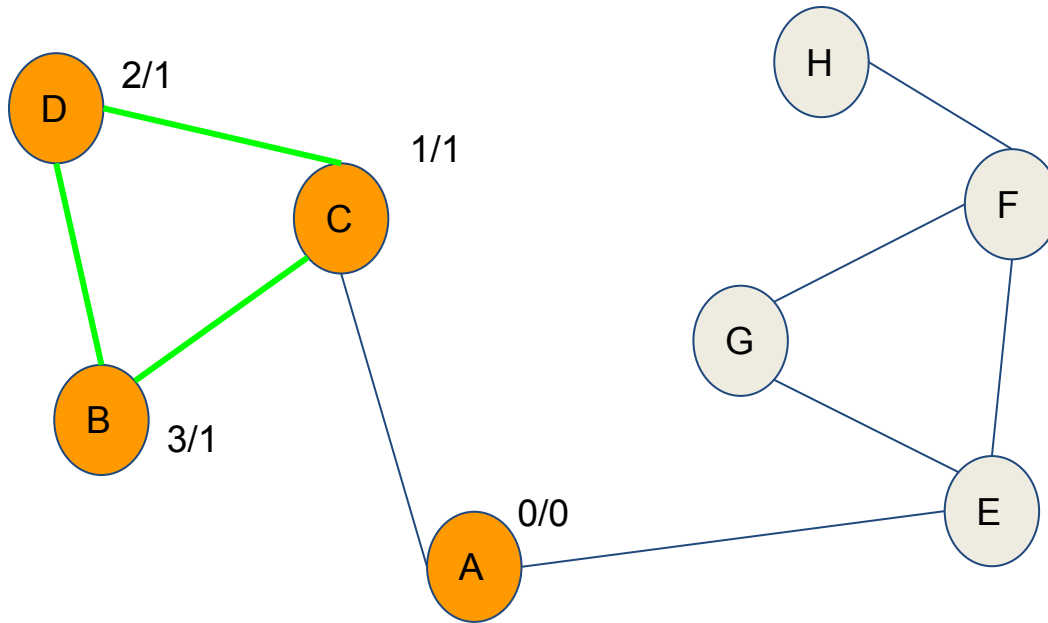
LOWEST = { 0, 1, 1, 2, -1, -1, -1, -1 }

DISCOVERY = { 0, 3, 1, 2, -1, -1, -1, -1 }

VISITED = { 1, 1, 1, 1, 0, 0, 0, 0 }

PARENTS = { -1, 3, 0, 2, -1, -1, -1, -1 }

# Grafos | Puntos de Articulación



DFS\_STACK = (B, 4)

LOWEST = { 0, 1, 1, 1, -1, -1, -1, -1 }

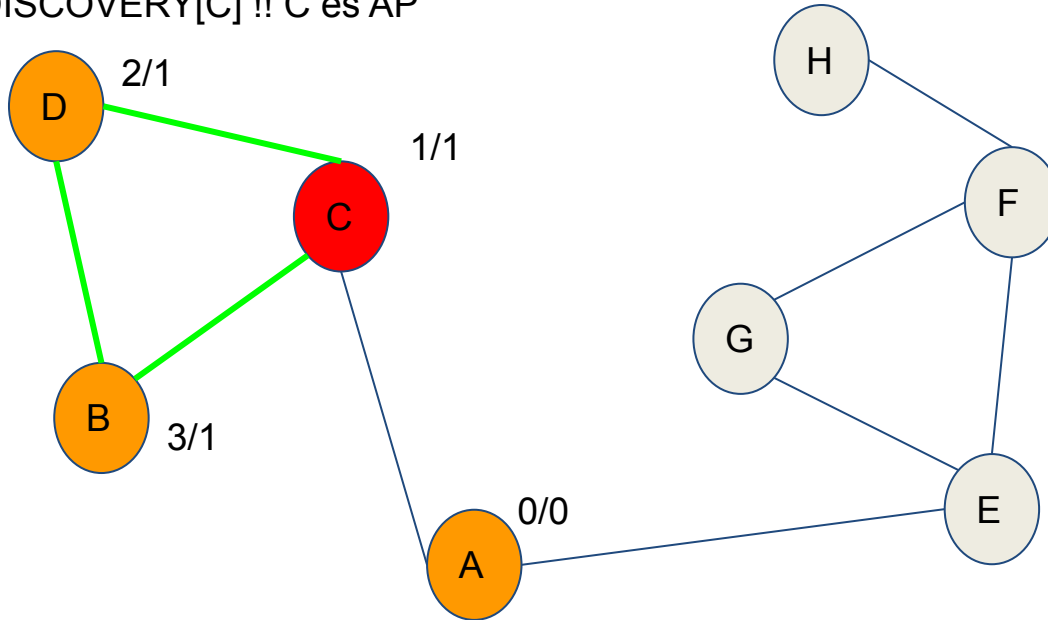
DISCOVERY = { 0, 3, 1, 2, -1, -1, -1, -1 }

VISITED = { 1, 1, 1, 1, 0, 0, 0, 0 }

PARENTS = { -1, 3, 0, 2, -1, -1, -1, -1 }

# Grafos | Puntos de Articulación

LOWEST[D]  $\geq$  DISCOVERY[C] !! C es AP



DFS\_STACK = (B, 4)

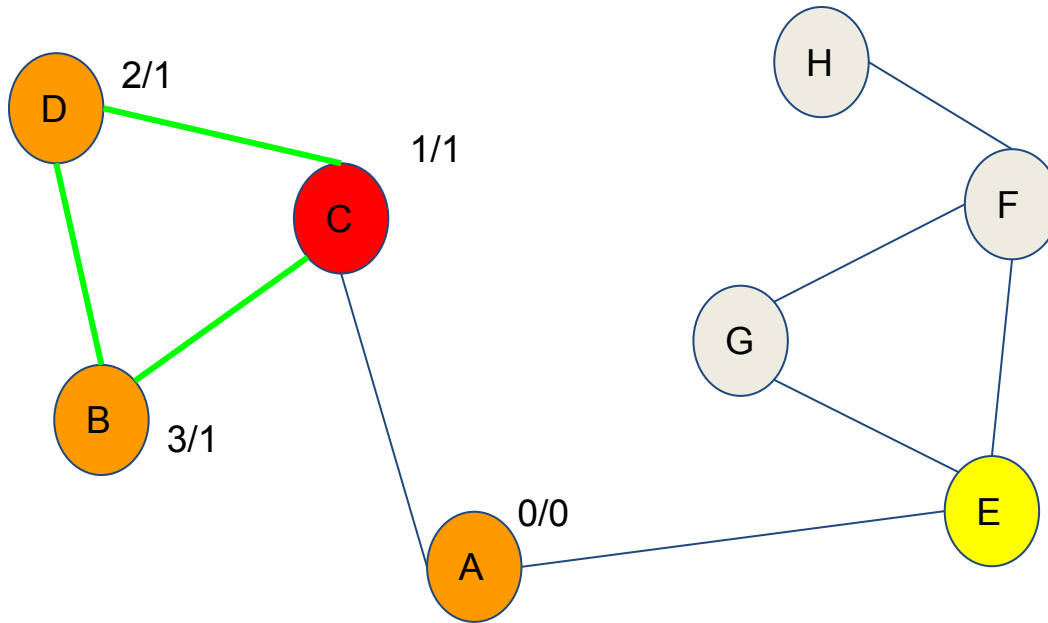
LOWEST = { 0, 1, 1, 1, -1, -1, -1, -1 }

DISCOVERY = { 0, 3, 1, 2, -1, -1, -1, -1 }

VISITED = { 1, 1, 1, 1, 0, 0, 0, 0 }

PARENTS = { -1, 3, 0, 2, -1, -1, -1, -1 }

# Grafos | Puntos de Articulación



DFS\_STACK = (E, 5)

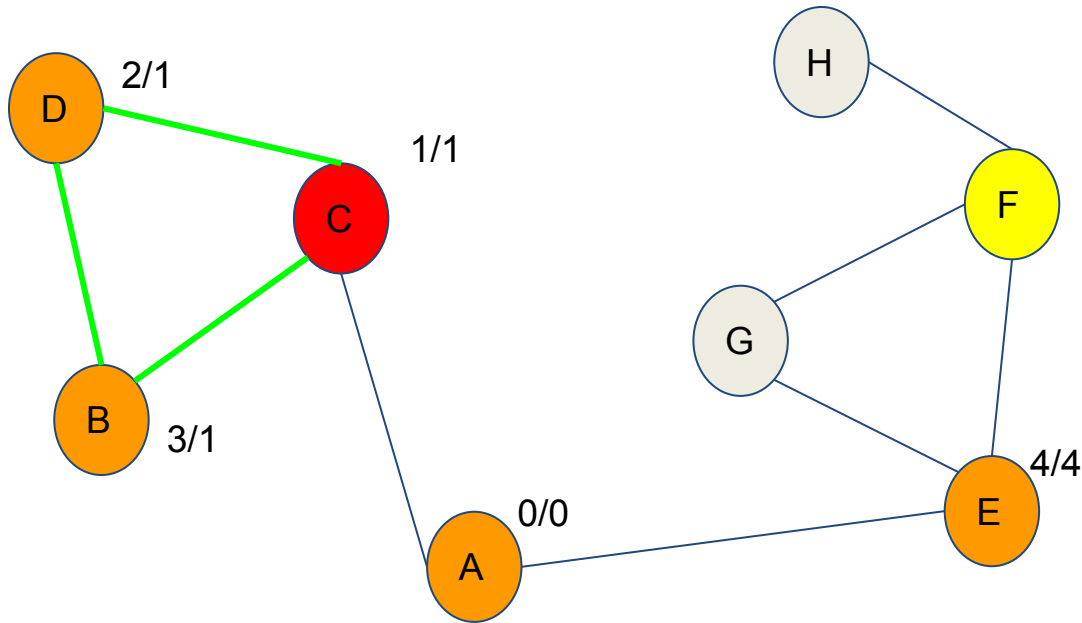
LOWEST = { 0, 1, 1, 1, 4, -1, -1, -1 }

DISCOVERY = { 0, 3, 1, 2, 4, -1, -1, -1 }

VISITED = { 1, 1, 1, 1, 1, 0, 0, 0 }

PARENTS = { -1, 3, 0, 2, 0, -1, -1, -1 }

# Grafos | Puntos de Articulación



DFS\_STACK = (F, 6)

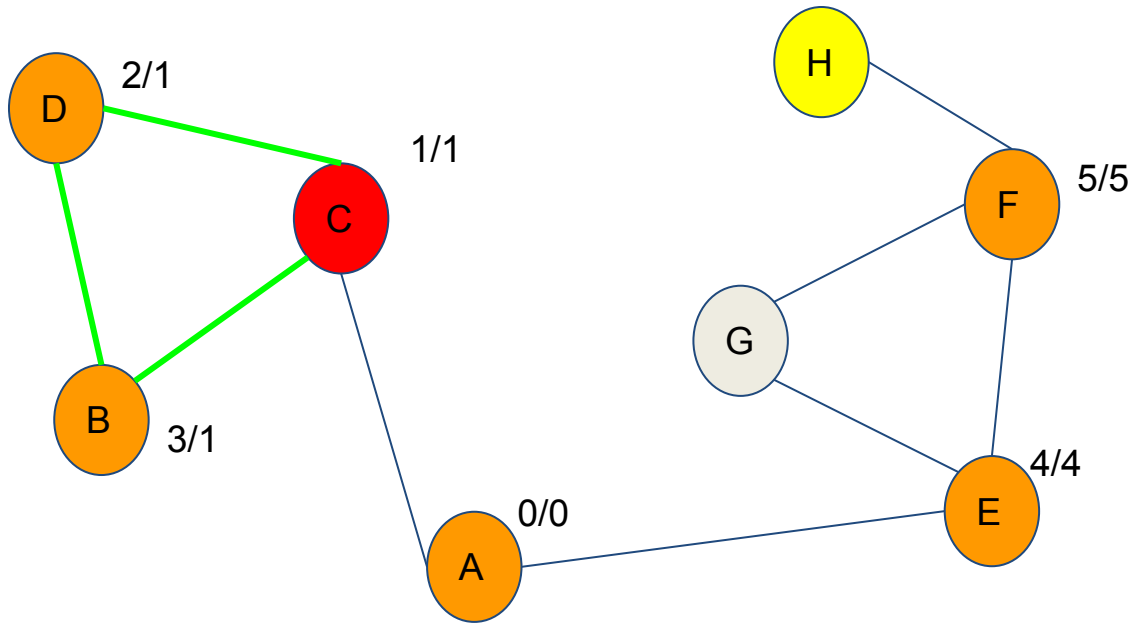
LOWEST = { 0, 1, 1, 1, 4, 5, -1, -1 }

DISCOVERY = { 0, 3, 1, 2, 4, 5, -1, -1 }

VISITED = { 1, 1, 1, 1, 1, 1, 0, 0 }

PARENTS = { -1, 3, 0, 2, 0, 4, -1, -1 }

# Grafos | Puntos de Articulación



DFS\_STACK = (H, 7)

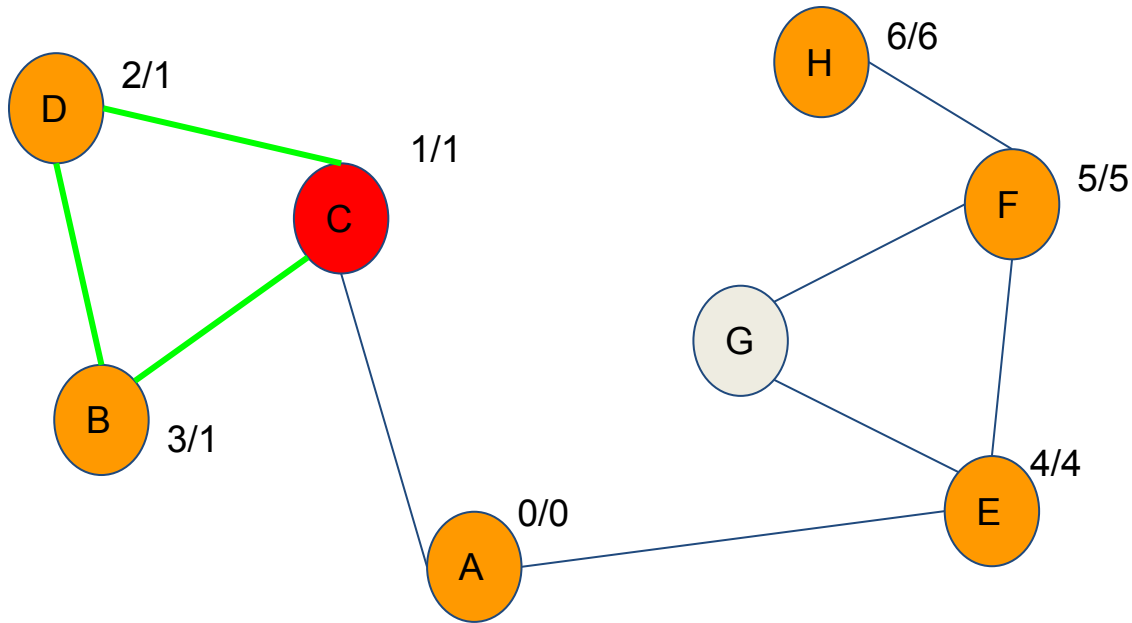
LOWEST = { 0, 1, 1, 1, 4, 5, -1, 6 }

DISCOVERY = { 0, 3, 1, 2, 4, 5, -1, 6 }

VISITED = { 1, 1, 1, 1, 1, 1, 0, 1 }

PARENTS = { -1, 3, 0, 2, 0, 4, -1, 5 }

# Grafos | Puntos de Articulación



DFS\_STACK = (H, 7)

LOWEST = { 0, 1, 1, 1, 4, 5, -1, 6 }

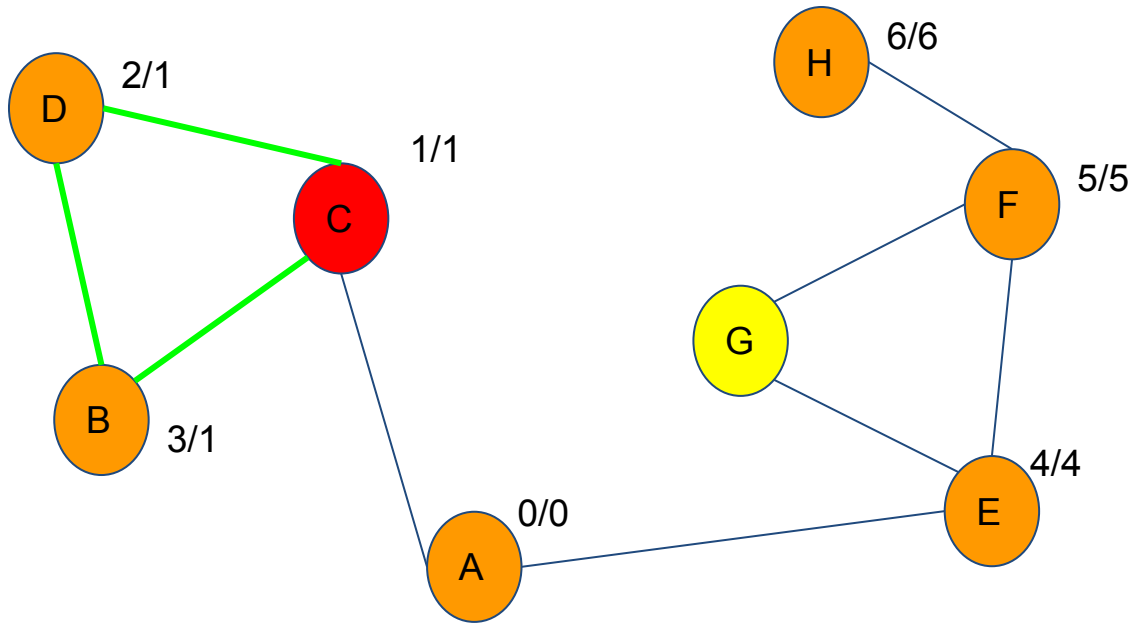
DISCOVERY = { 0, 3, 1, 2, 4, 5, -1, 6 }

VISITED = { 1, 1, 1, 1, 1, 1, 0, 1 }

PARENTS = { -1, 3, 0, 2, 0, 4, -1, 5 }

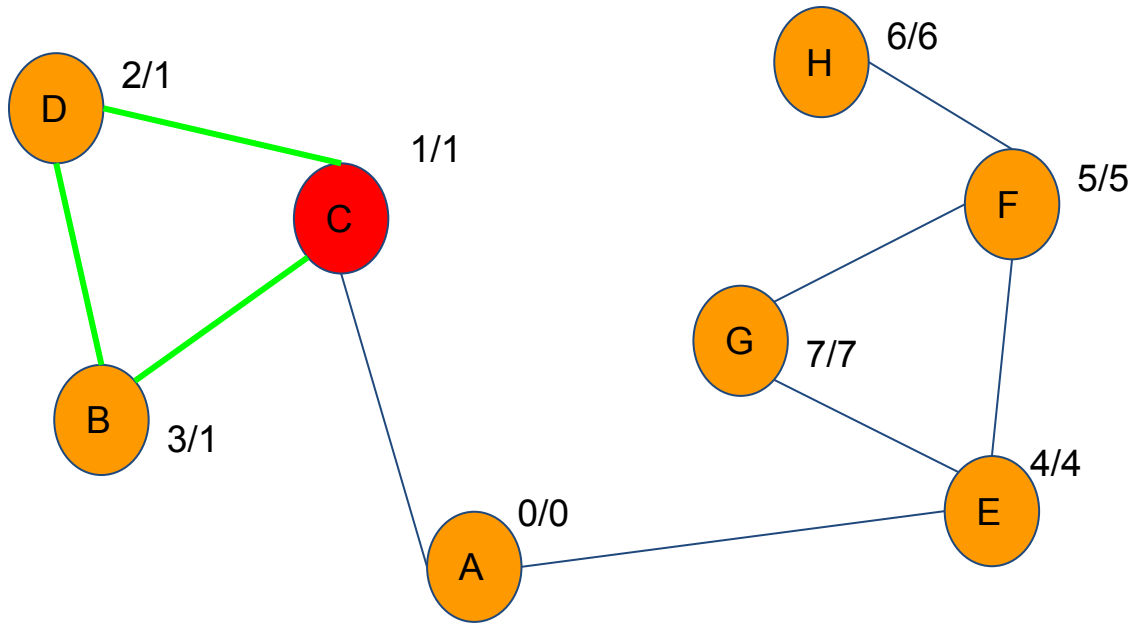


# Grafos | Puntos de Articulación



DFS\_STACK = (G, 8)  
LOWEST = { 0, 1, 1, 1, 4, 5, 7, 6 }  
DISCOVERY = { 0, 3, 1, 2, 4, 5, 7, 6 }  
VISITED = { 1, 1, 1, 1, 1, 1, 1, 1 }  
PARENTS = { -1, 3, 0, 2, 0, 4, 5, 5 }

# Grafos | Puntos de Articulación



DFS\_STACK = (G, 8)

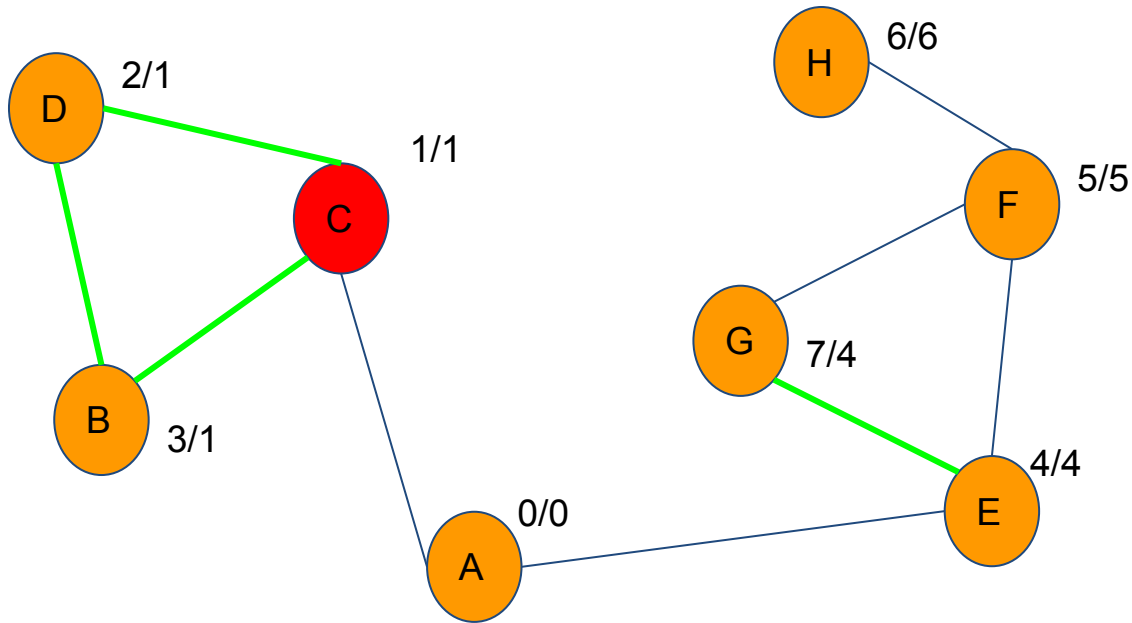
LOWEST = { 0, 1, 1, 1, 4, 5, 7, 6 }

DISCOVERY = { 0, 3, 1, 2, 4, 5, 7, 6 }

VISITED = { 1, 1, 1, 1, 1, 1, 1, 1 }

PARENTS = { -1, 3, 0, 2, 0, 4, 5, 5 }

# Grafos | Puntos de Articulación



DFS\_STACK = (G, 8)

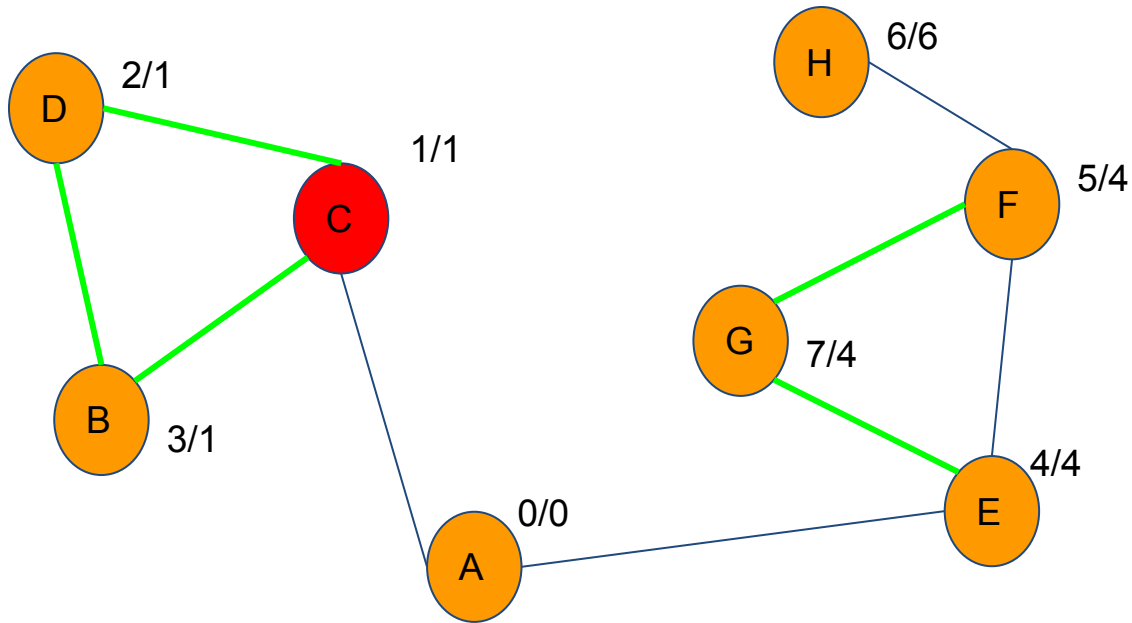
LOWEST = { 0, 1, 1, 1, 4, 5, 4, 6 }

DISCOVERY = { 0, 3, 1, 2, 4, 5, 7, 6 }

VISITED = { 1, 1, 1, 1, 1, 1, 1, 1 }

PARENTS = { -1, 3, 0, 2, 0, 4, 5, 5 }

# Grafos | Puntos de Articulación



DFS\_STACK = (G, 8)

LOWEST = { 0, 1, 1, 1, 4, 4, 4, 6 }

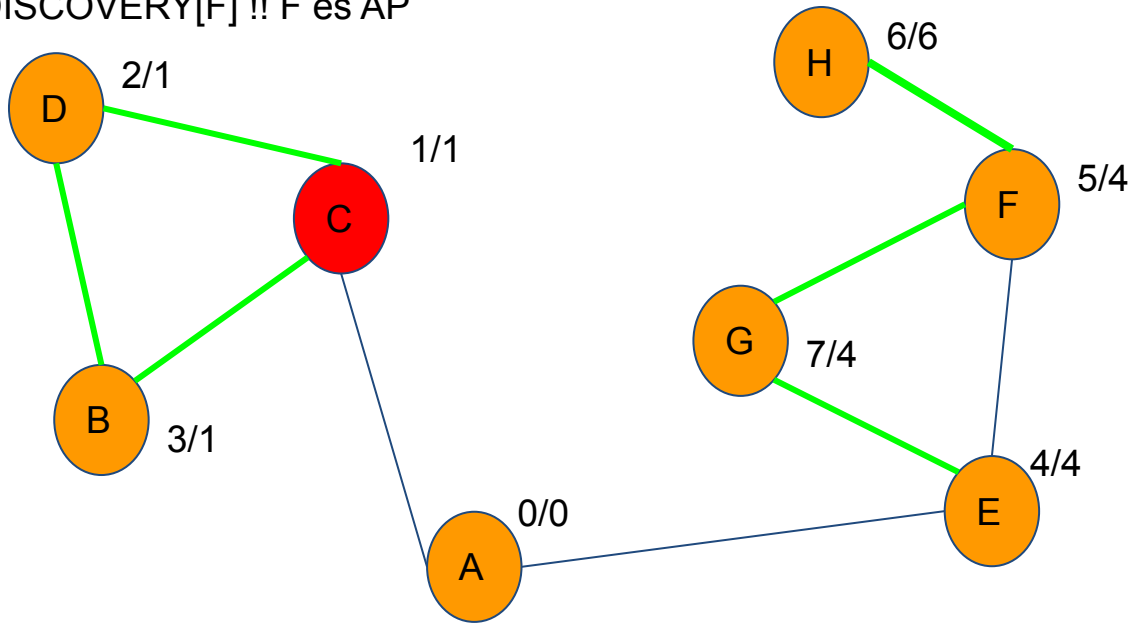
DISCOVERY = { 0, 3, 1, 2, 4, 5, 7, 6 }

VISITED = { 1, 1, 1, 1, 1, 1, 1, 1 }

PARENTS = { -1, 3, 0, 2, 0, 4, 5, 5 }

# Grafos | Puntos de Articulación

LOWEST[H] >= DISCOVERY[F] !! F es AP



DFS\_STACK = (G, 8)

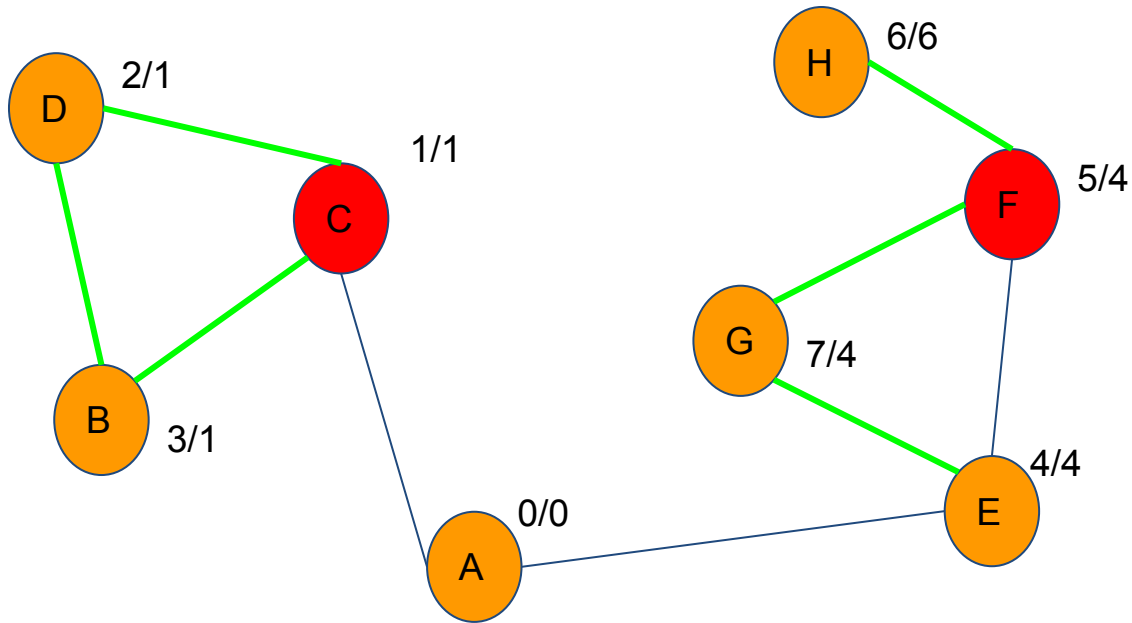
LOWEST = { 0, 1, 1, 1, 4, 4, 4, 6 }

DISCOVERY = { 0, 3, 1, 2, 4, 5, 7, 6 }

VISITED = { 1, 1, 1, 1, 1, 1, 1, 1 }

PARENTS = { -1, 3, 0, 2, 0, 4, 5, 5 }

# Grafos | Puntos de Articulación



DFS\_STACK = (G, 8)

LOWEST = { 0, 1, 1, 1, 4, 4, 4, 6 }

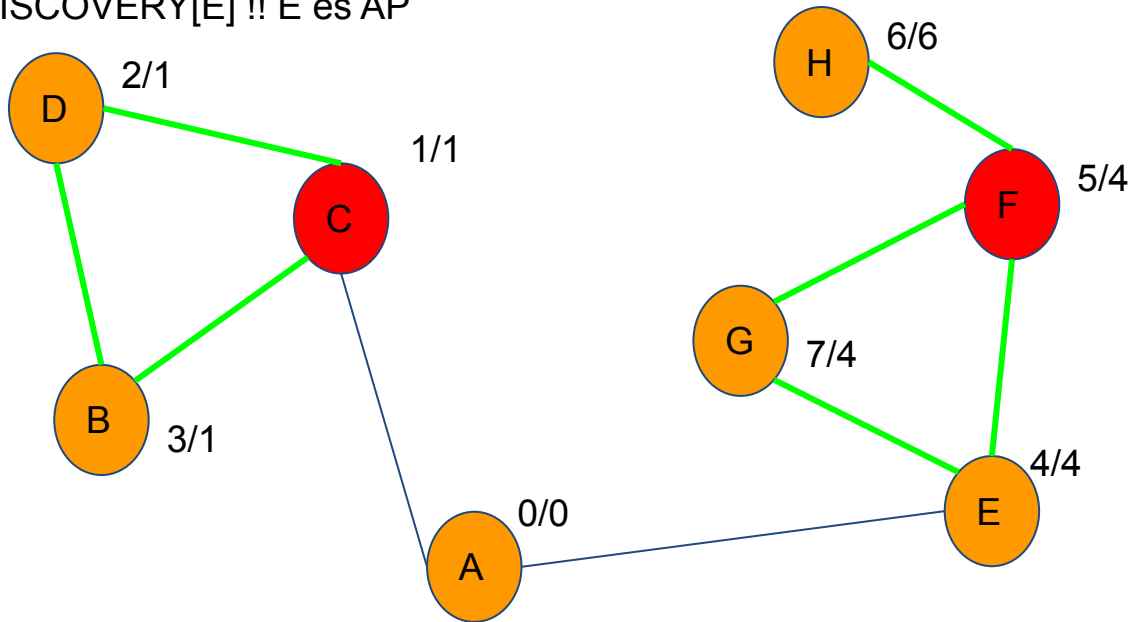
DISCOVERY = { 0, 3, 1, 2, 4, 5, 7, 6 }

VISITED = { 1, 1, 1, 1, 1, 1, 1, 1 }

PARENTS = { -1, 3, 0, 2, 0, 4, 5, 5 }

# Grafos | Puntos de Articulación

LOWEST[F] >= DISCOVERY[E] !! E es AP



DFS\_STACK = (G, 8)

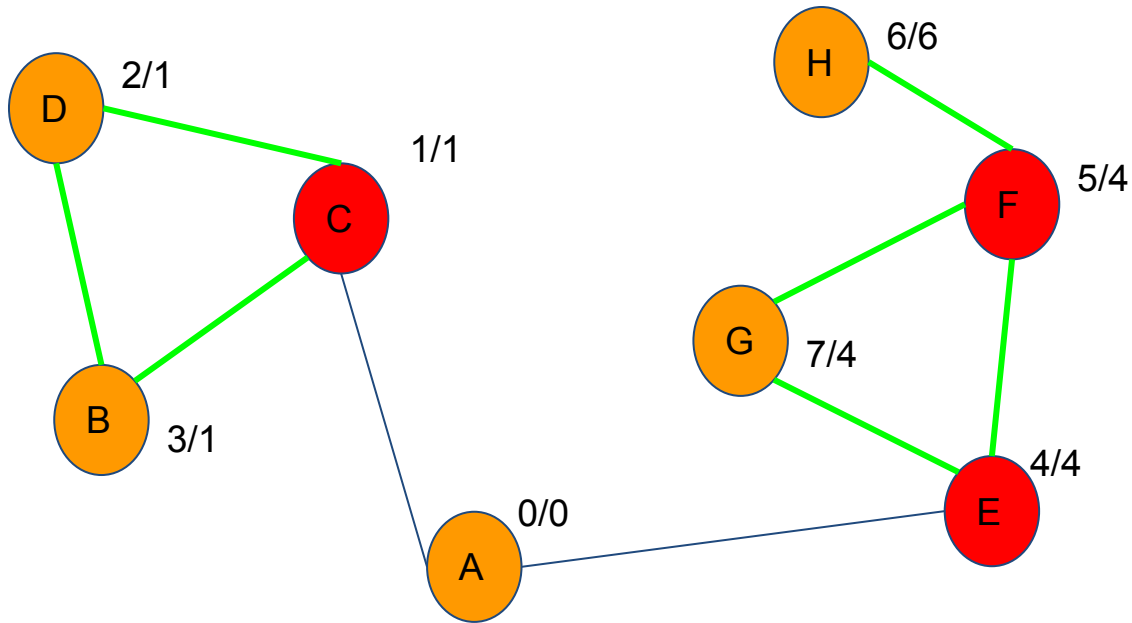
LOWEST = { 0, 1, 1, 1, 4, 4, 4, 6 }

DISCOVERY = { 0, 3, 1, 2, 4, 5, 7, 6 }

VISITED = { 1, 1, 1, 1, 1, 1, 1, 1 }

PARENTS = { -1, 3, 0, 2, 0, 4, 5, 5 }

# Grafos | Puntos de Articulación



DFS\_STACK = (G, 8)

LOWEST = { 0, 1, 1, 1, 4, 4, 4, 6 }

DISCOVERY = { 0, 3, 1, 2, 4, 5, 7, 6 }

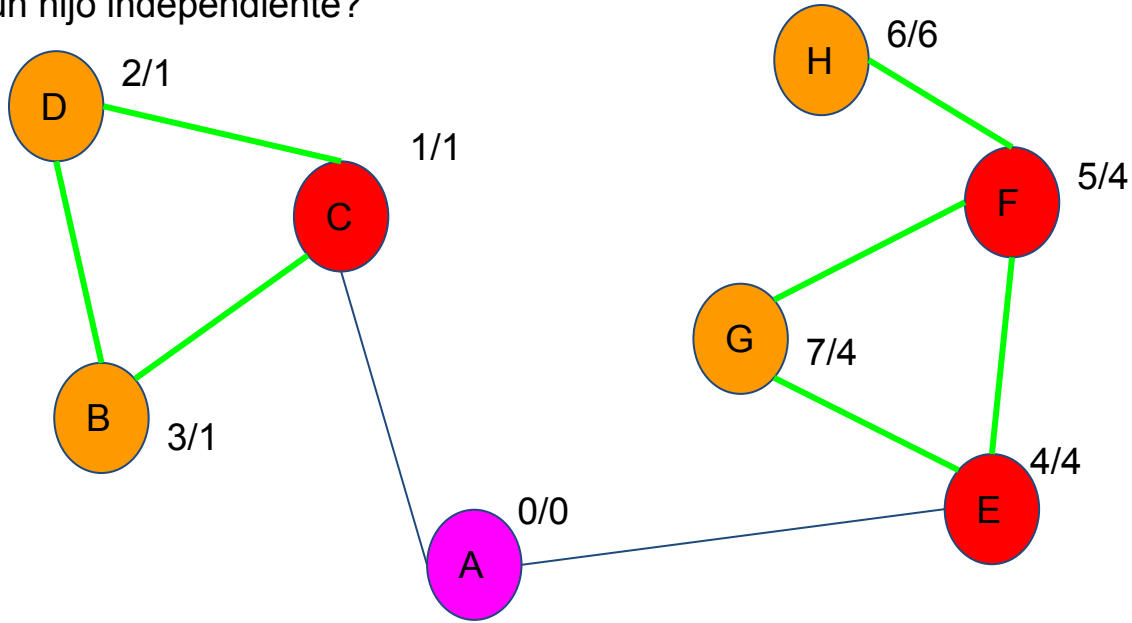
VISITED = { 1, 1, 1, 1, 1, 1, 1, 1 }

PARENTS = { -1, 3, 0, 2, 0, 4, 5, 5 }



# Grafos | Puntos de Articulación

¿A tiene mas de un hijo independiente?



DFS\_STACK = (G, 8)

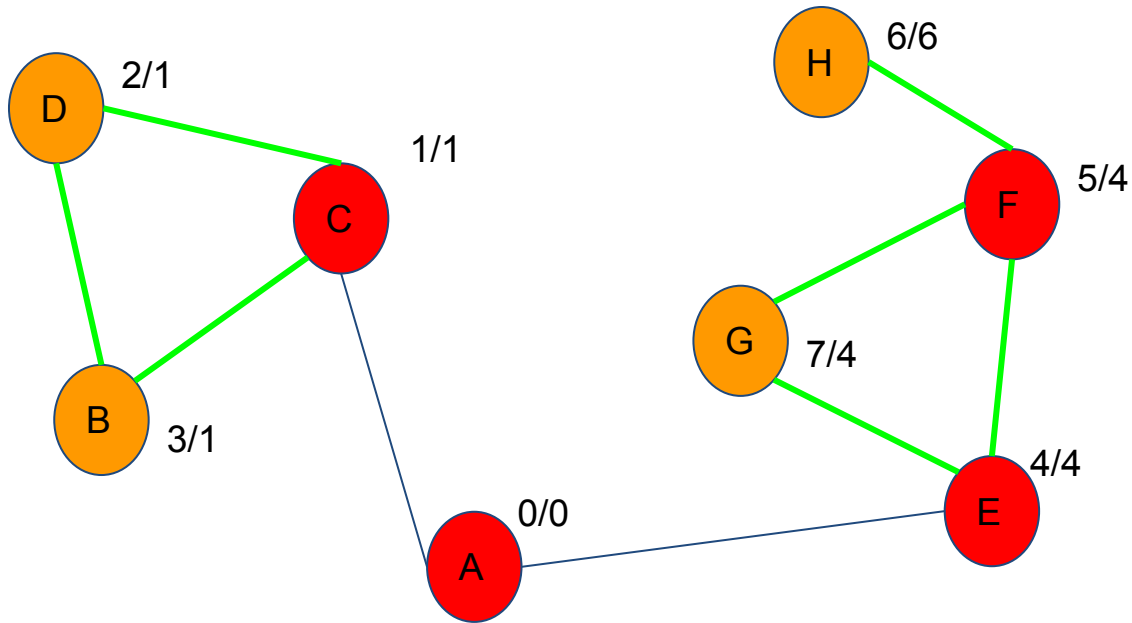
LOWEST = { 0, 1, 1, 1, 4, 4, 4, 6 }

DISCOVERY = { 0, 3, 1, 2, 4, 5, 7, 6 }

VISITED = { 1, 1, 1, 1, 1, 1, 1, 1 }

PARENTS = { -1, 3, 0, 2, 0, 4, 5, 5 }

# Grafos | Puntos de Articulación



DFS\_STACK = (G, 8)

LOWEST = { 0, 1, 1, 1, 4, 4, 4, 6 }

DISCOVERY = { 0, 3, 1, 2, 4, 5, 7, 6 }

VISITED = { 1, 1, 1, 1, 1, 1, 1, 1 }

PARENTS = { -1, 3, 0, 2, 0, 4, 5, 5 }

# Grafos | Puntos de Articulación

- Pseudocódigo

AP(u, t):

V(u) = true

d[u] = low[u] = t++

children = 0

for v in edges[u]:

    if(!V(v)):

        children++, p[v] = u

        AP(v, t)

        low[u] = min(low[u], low[v])

        checkAP(u, v, children)

    else if(v != p[u])

        low[u] = min(low[u], d[v])

# Grafos | Puntos de Articulación

- Pseudocódigo

```
checkAP(u, v, children):  
    if p[u] == -1 && children > 1:  
        ap[u] = true  
    if p[u] != -1 && low[v] >= disc[u]:  
        ap[u] = true
```

# Semana que viene...

- Grafos (parte II)
  - Ponderamiento en grafos
  - Colas de prioridad
  - Algoritmos de distancia mínima (floyd warshall, dijkstra)
  - Estructura Union-Find
  - Árboles de recubrimiento (Prim, Kruskal)

# ¡Hasta la próxima semana!

Ante cualquier duda sobre el curso o sobre los problemas podéis escribirnos (preferiblemente con copia a algunos / todos los docentes)

- David Morán ([david.moran@urjc.es](mailto:david.moran@urjc.es))
- Sergio Pérez ([sergio.perez.pelo@urjc.es](mailto:sergio.perez.pelo@urjc.es))
- Jesús Sánchez-Oro ([jesus.sanchezoro@urjc.es](mailto:jesus.sanchezoro@urjc.es))
- Isaac Lozano ([isaac.lozano@urjc.es](mailto:isaac.lozano@urjc.es))
- Raúl Martín ([raul.martin@urjc.es](mailto:raul.martin@urjc.es))
- Jakub Jan ([jakubjanluczyn@gmail.com](mailto:jakubjanluczyn@gmail.com))
- Antonio Gonzalez ([antonio.gpardo@urjc.es](mailto:antonio.gpardo@urjc.es))
- Iván Martín ([ivan.martin@urjc.es](mailto:ivan.martin@urjc.es))
- Leonardo Antonio Santella ([leocaracas2010@gmail.com](mailto:leocaracas2010@gmail.com))