

Sesión 4 (7ª Semana)

David Morán (ddavidmorang@gmail.com)

Juan Quintana (juandavid.quintana@urjc.es)

Sergio Pérez (sergio.perez.pelo@urjc.es)

Jesús Sánchez-Oro (jesus.sanchezoro@urjc.es)

Antes de empezar

- Ada-Byron el 8 y 9 de marzo
 - No habrá sesión práctica en el sitio
 - Será una competición larga desde el sábado a las 09:00 hasta el viernes siguiente a las 23:55
 - Serán más problemas de lo habitual (¡hay que entrenar mucho!)

Contenidos

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

Contenidos

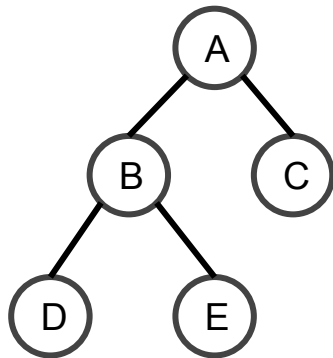
- Grafos
 - Componentes Conexas
 - Ordenamiento Topológico
 - Componentes Fuertemente Conexas

Grafos

- ¿Qué es?
- Representación
 - Con índices
 - i. $A \rightarrow B = \text{mat}[a][b] = \text{true}$
 - Con strings (utilizar map)
 - i. $A \rightarrow B = \text{mat}[\text{map}[A]][\text{map}[B]] = \text{true}$

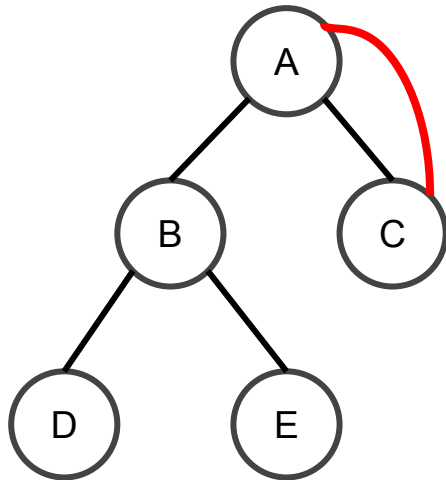
Grafos

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



Grafos

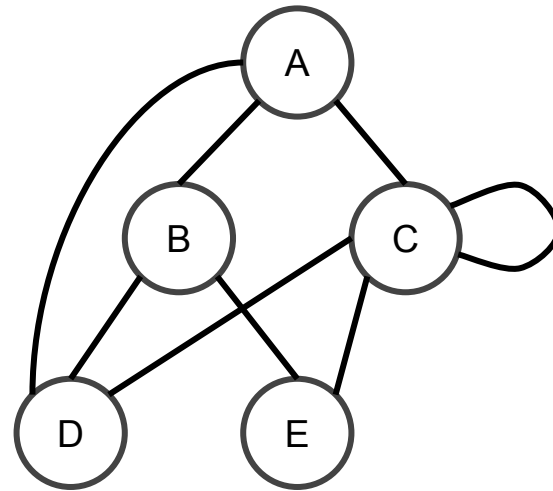
- Restricciones jerárquicas:
 - No admite ciclos



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

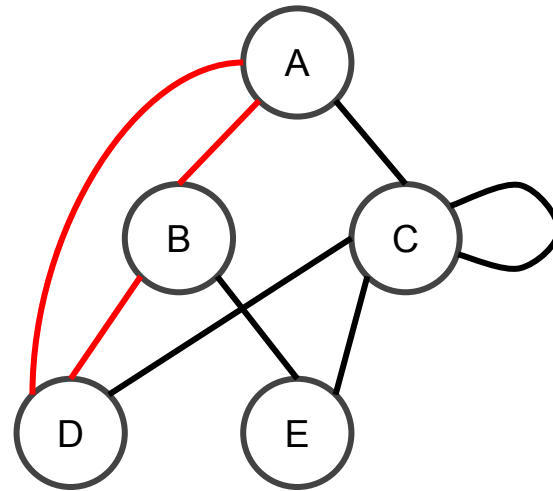
Grafos

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



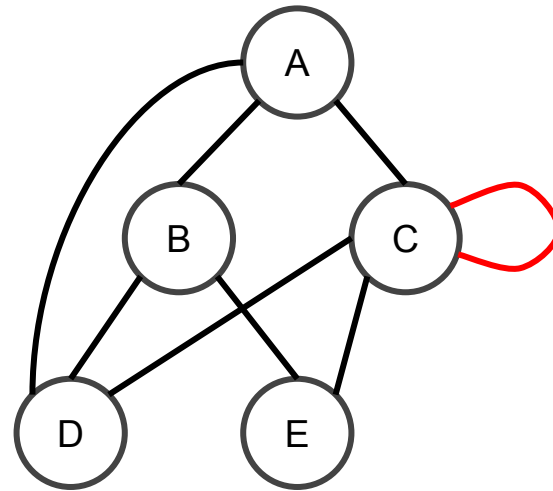
Grafos

- podemos tener ciclos



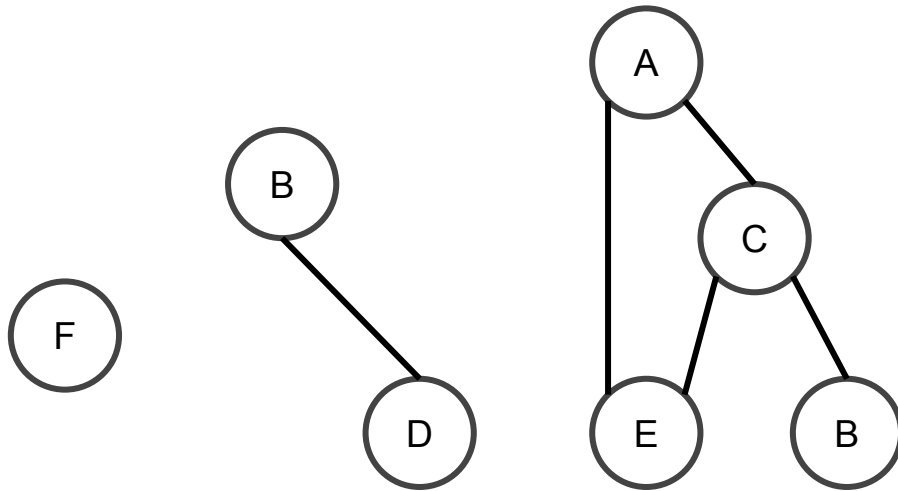
Grafos

- podemos tener bucles sobre el mismo elemento



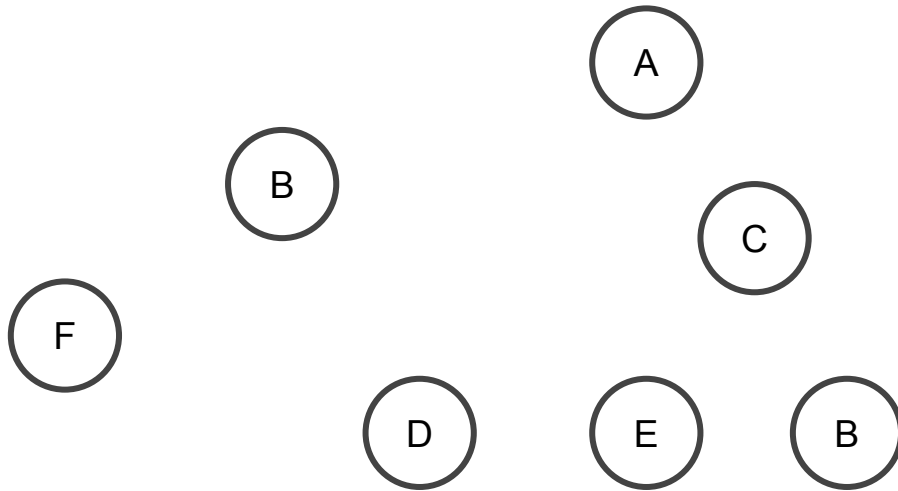
Grafos

- podemos tener grupos aislados en un mismo grafo



Grafos

- o elementos totalmente aislados entre sí



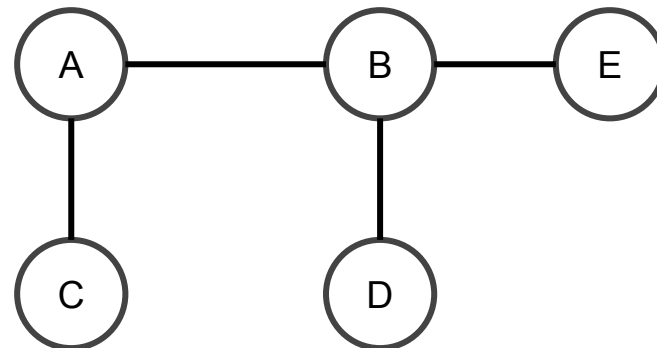
Grafos

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

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

- Conjunto de aristas:

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

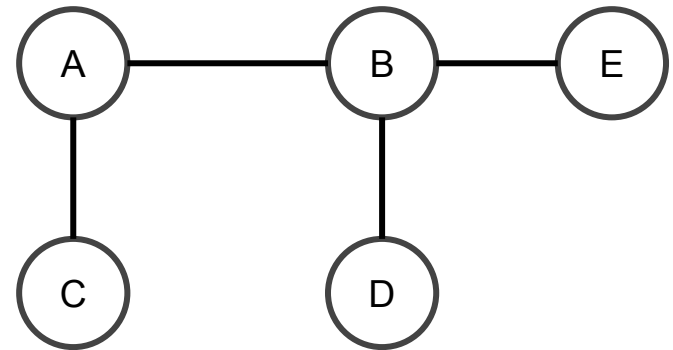


Grafos

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

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

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

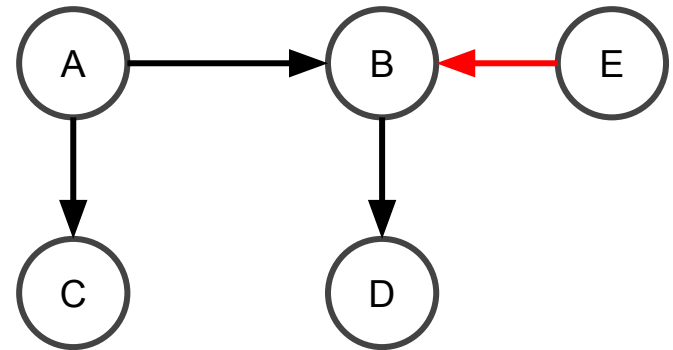


Grafos

- Grafo no 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

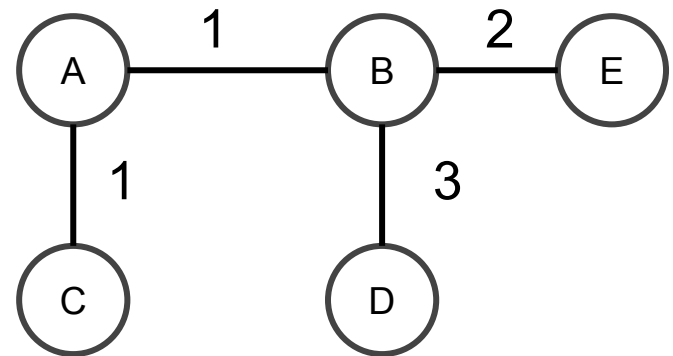
$E = \{(A, B), (A, C), (B, D), (B, E,)\}$

- función de pesos (f)

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

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

...

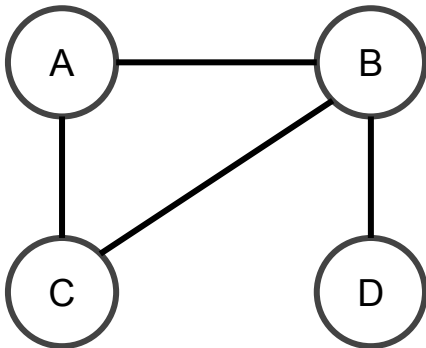


Grafos - Implementación

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

Grafos - Matriz

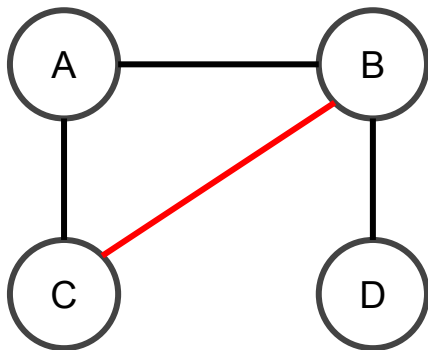
- Matriz de adyacencia
 - Array de dos dimensiones



	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

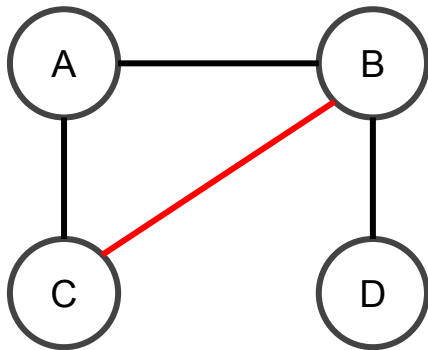
- Arista (B,C) $\Rightarrow m[1][2] = 1$



	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

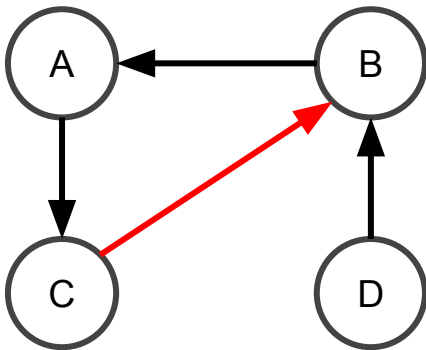
- Matriz simétrica en grafos no dirigidos
- $m[\text{fila}][\text{col}] == m[\text{col}][\text{fila}]$



	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 $m[1][2]=0$



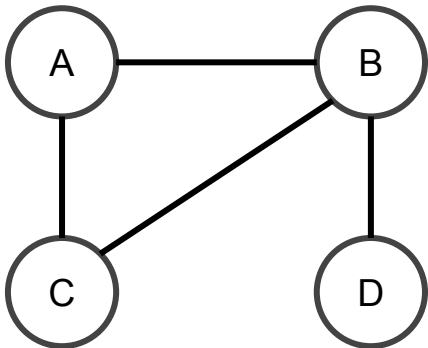
	A	B	C	D
A	0	0	1	0
B	1	0	0	0
C	0	1	0	0
D	0	1	0	0

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

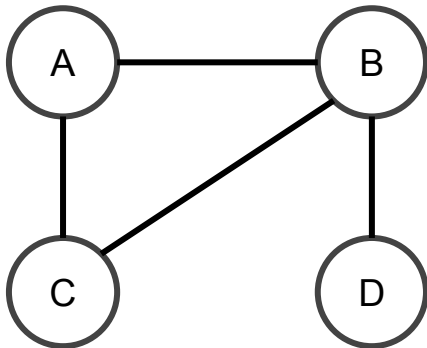
- Lista de adyacencia
 - enumerar las aristas por vértice



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: $O(|V|)$
 - 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

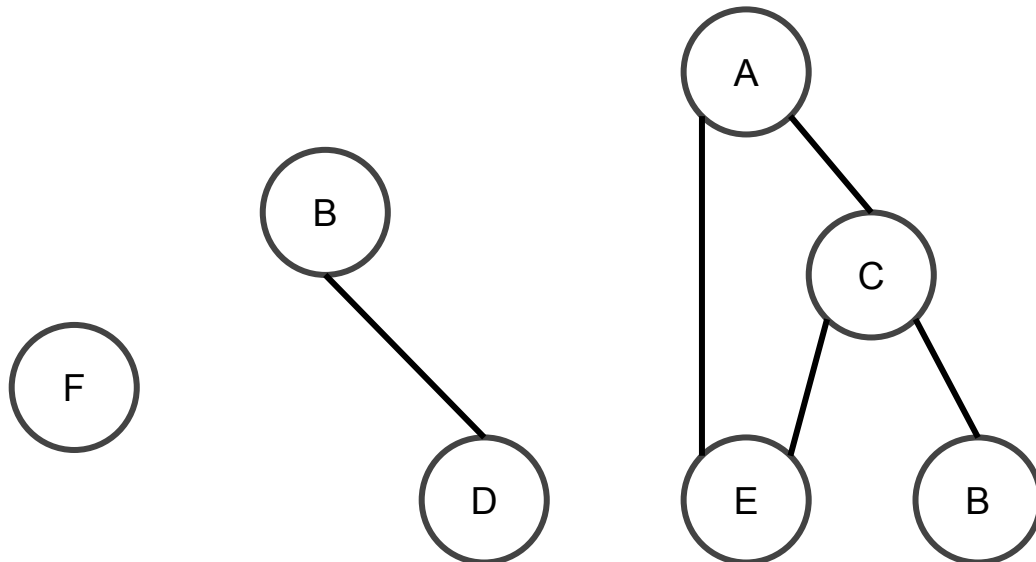
- Existe otra implementación para representar un grafo
- Más utilizada para algoritmos específicos
- Guardar en una lista las conexiones de A a B
- `vector<arista> aristas`
- `aristas.insertar(arista(a, b))`

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

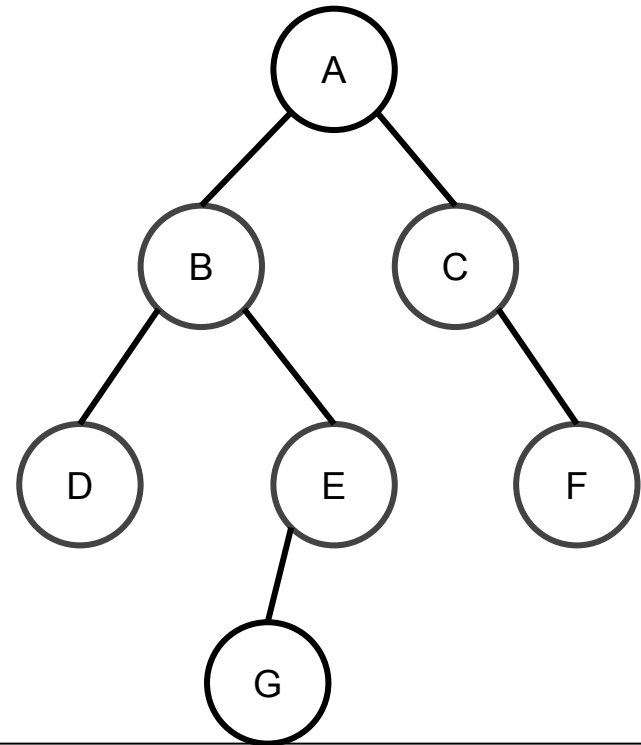
- Recorrer los vértices de un grafo
 - Necesitamos llevar cuenta de en qué nodos hemos estado ya
 - Ventajas: Si llegamos a un nodo destino, está garantizado que habremos llegado en la menor cantidad de pasos
 - Cada arista recorrida cuenta como 1 paso

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

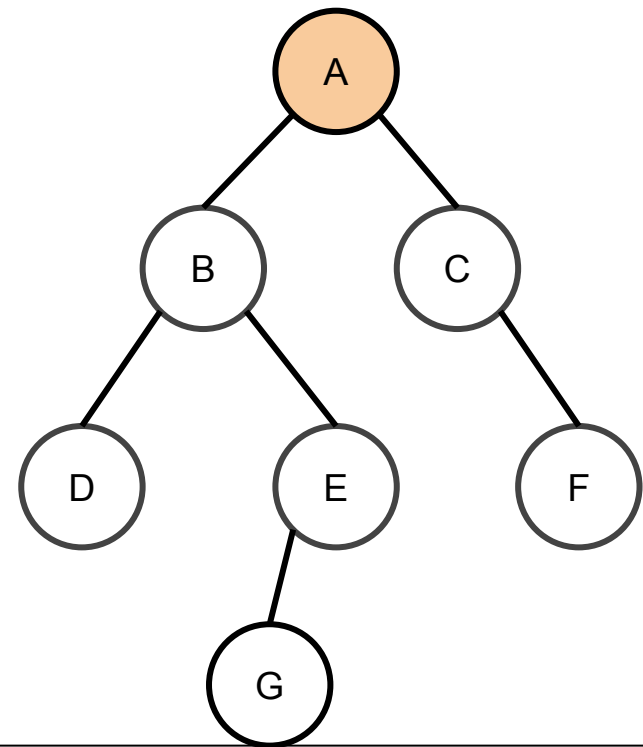
- Recorrido en anchura



Grafos - BFS

- Si empezamos desde A
- Recorrido por niveles

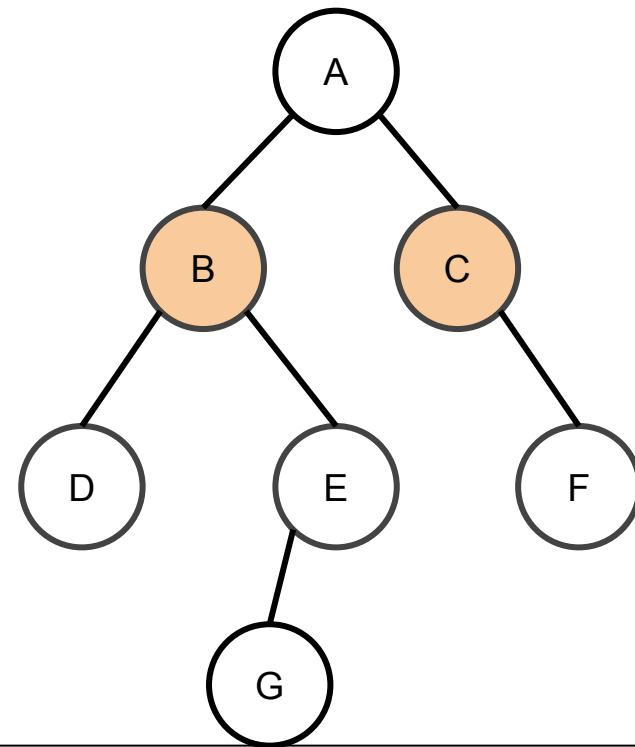
A



Grafos - BFS

- Si empezamos desde A
- Recorrido por niveles

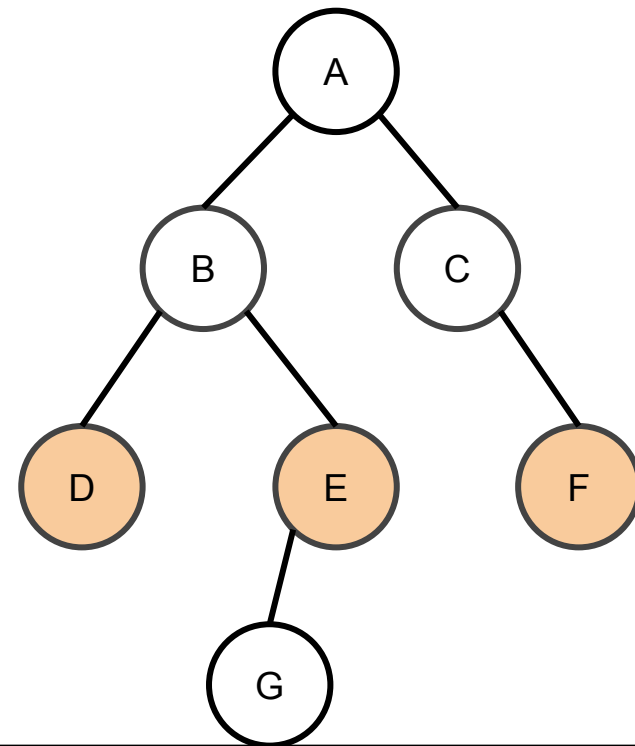
A ⇒ B ⇒ C



Grafos - BFS

- Si empezamos desde A
- Recorrido por niveles

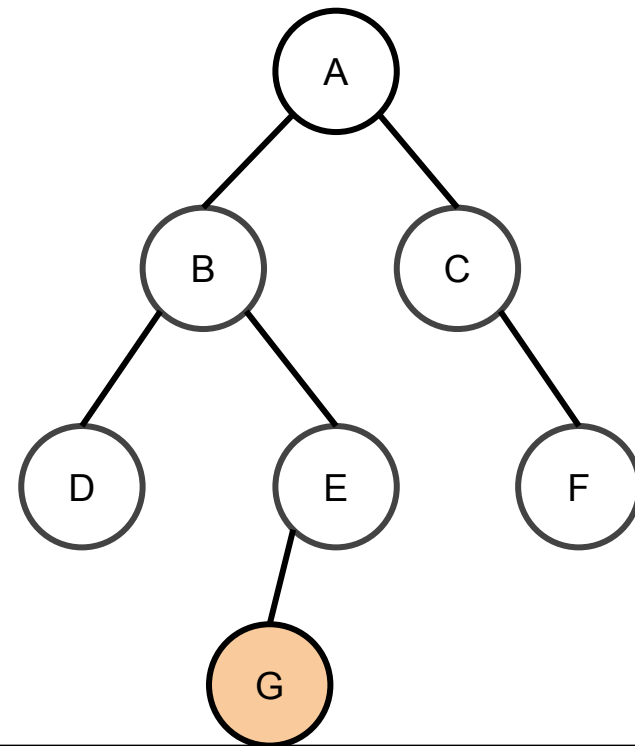
A ⇒ B ⇒ C ⇒ D ⇒ E
⇒ F ⇒ G



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

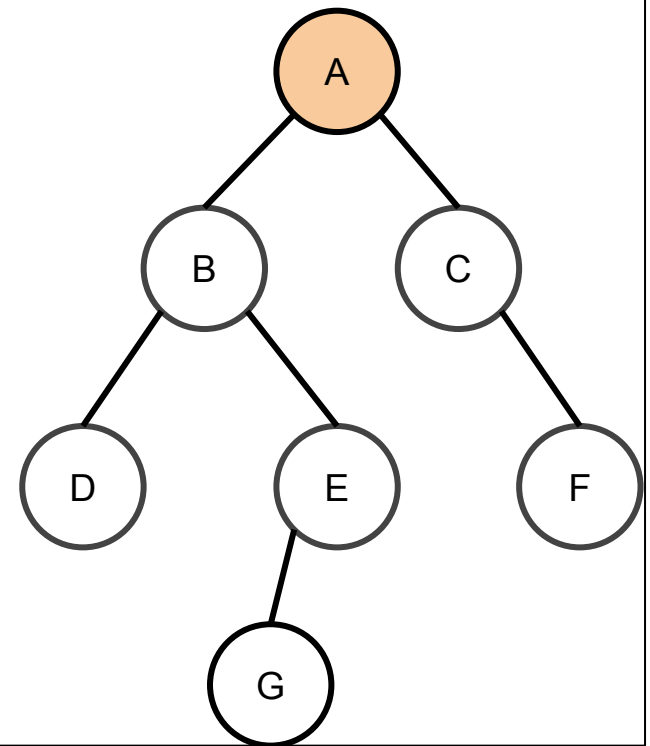
Grafos - BFS

- Inicialización: Elegimos un vértice inicial

`inicial = 0`

`visitado[inicial]=true`

`cola.add(inicial)`



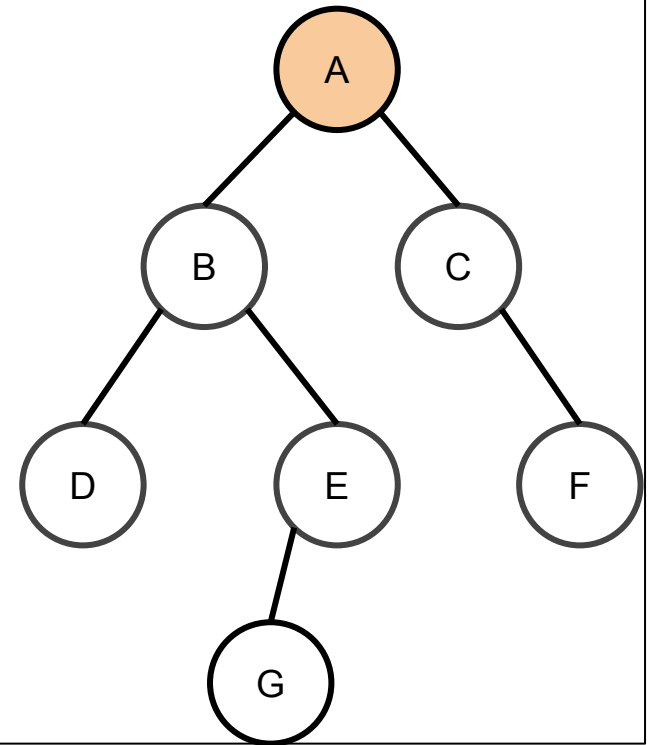
Grafos - BFS

- Cola = {A}

inicial = A

visitado[inicial]=true

cola.add(inicial)



Grafos - BFS

- Recorremos los vértices

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

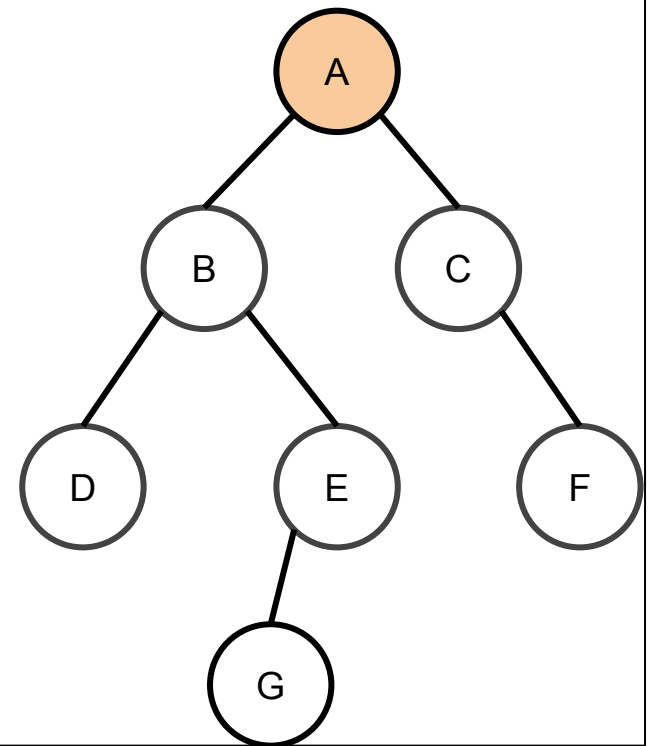
```
  v = cola.extraer()
```

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

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

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

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



Grafos - BFS

- Cola = {}
- Sacamos A

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

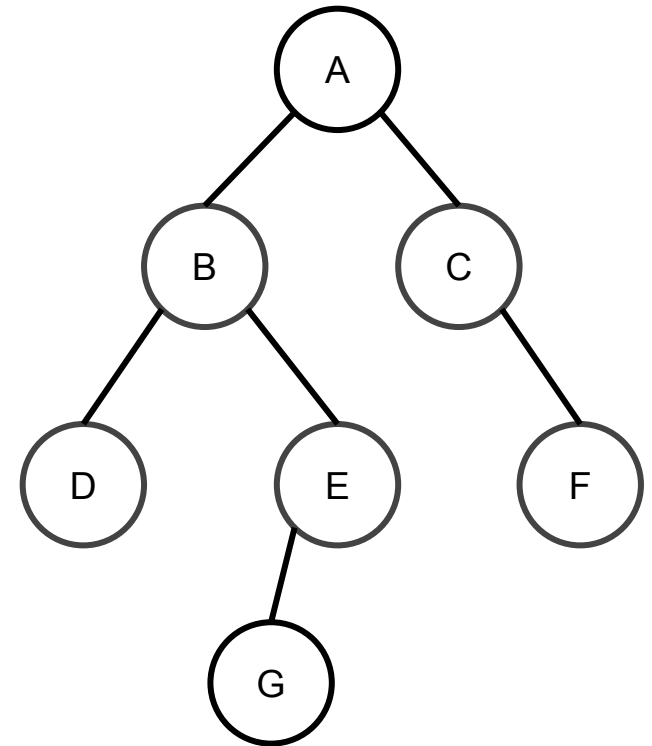
`v = cola.extraer()`

para cada `ady` de `v`:

if(`no visitado[ady]`)

`visitado[ady]=true`

`cola.add(ady)`



Grafos - BFS

- Cola = {B,C}

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

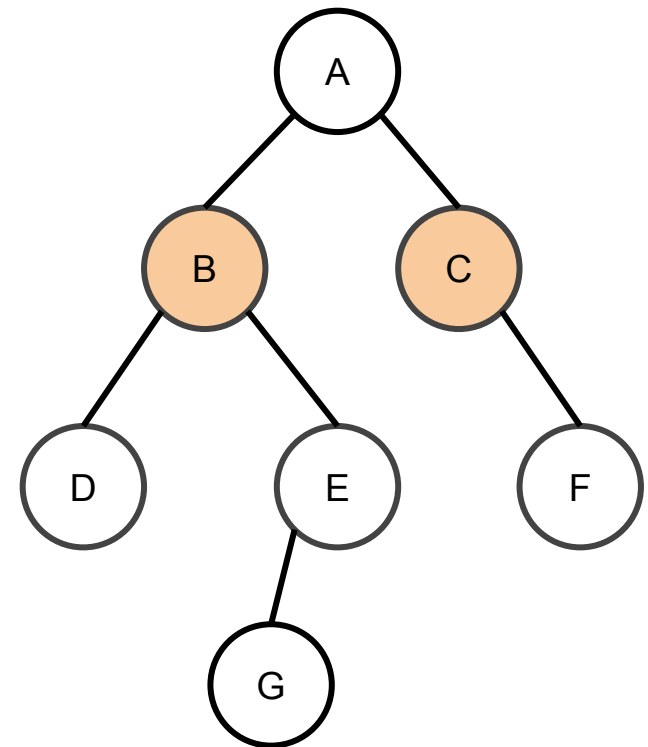
`v = cola.extraer()`

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`)

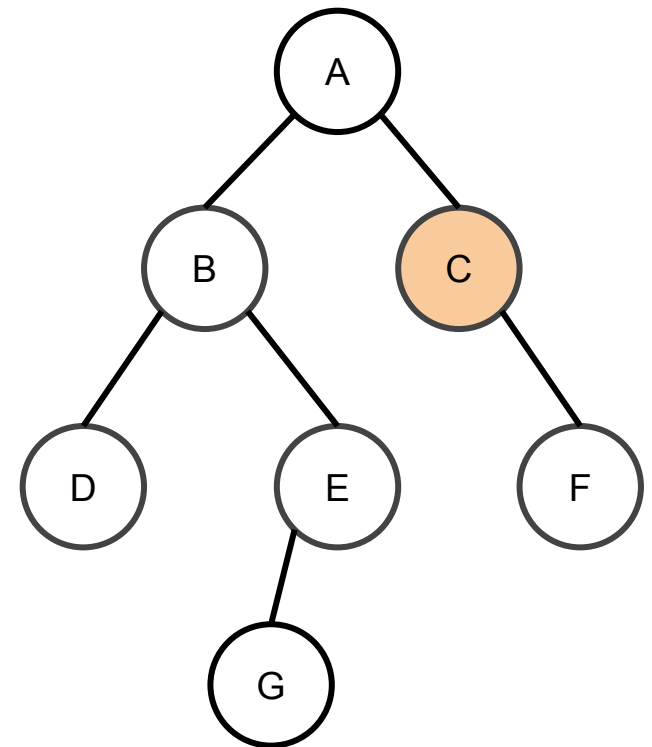
`v = cola.extraer()`

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`)

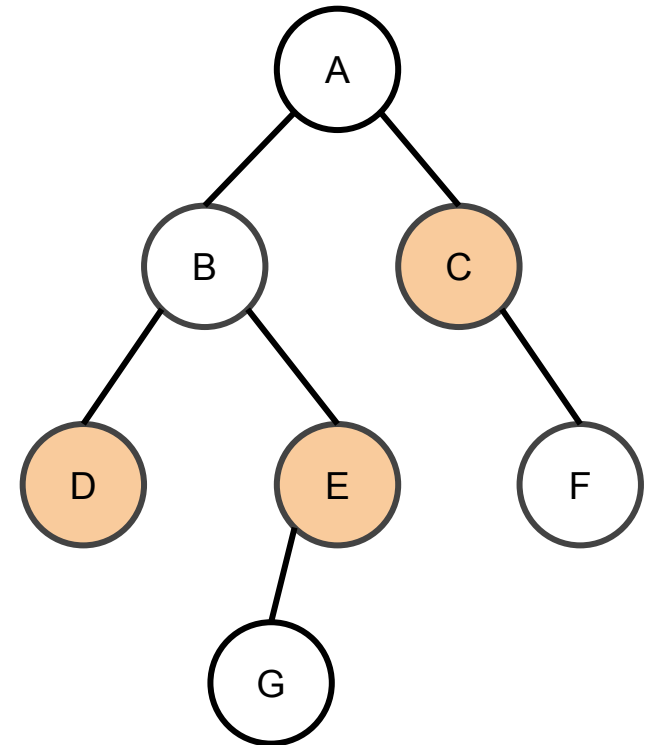
`v = cola.extraer()`

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`)

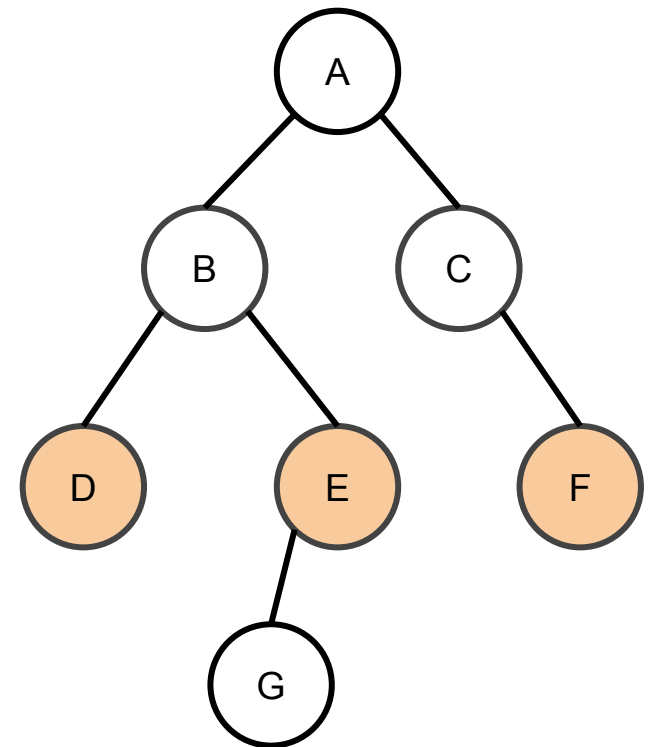
`v = cola.extraer()`

para cada ady de v:

if(`no visitado[ady]`)

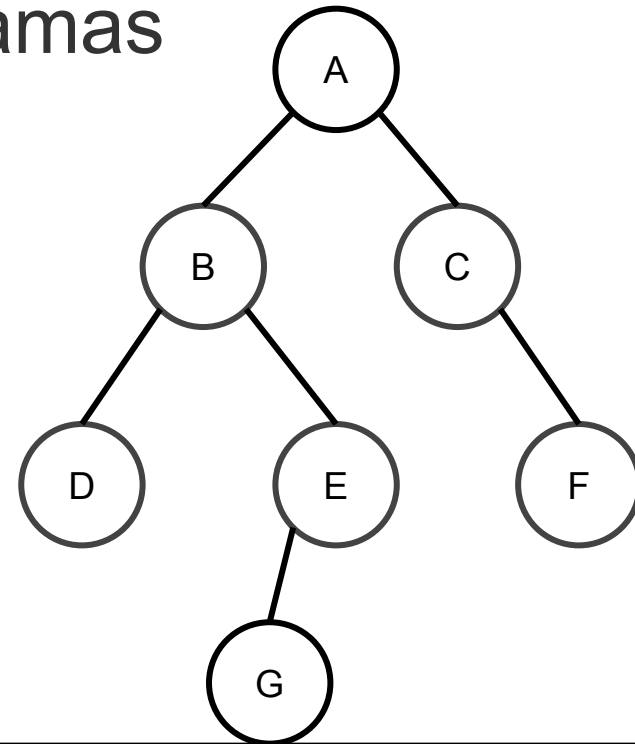
`visitado[ady]=true`

`cola.add(ady)`



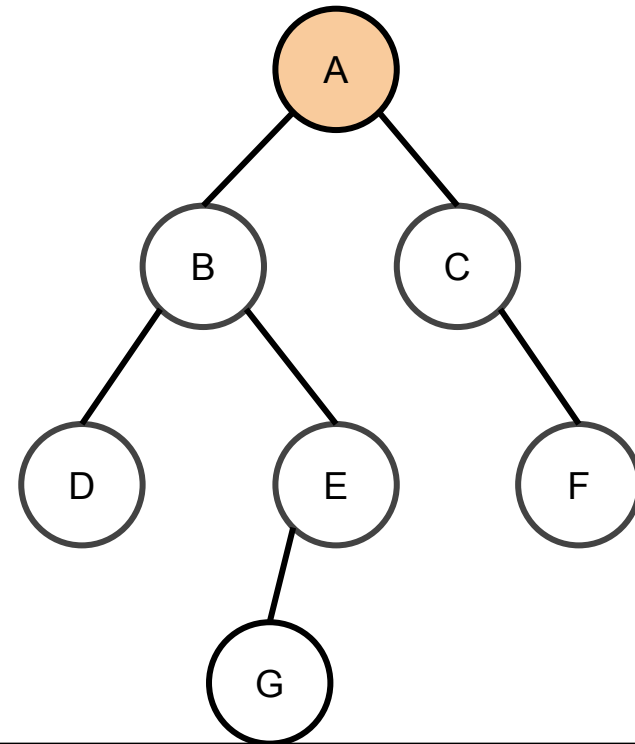
Grafos - DFS

- Recorrido en profundidad
- Equivalente a recorrer ramas



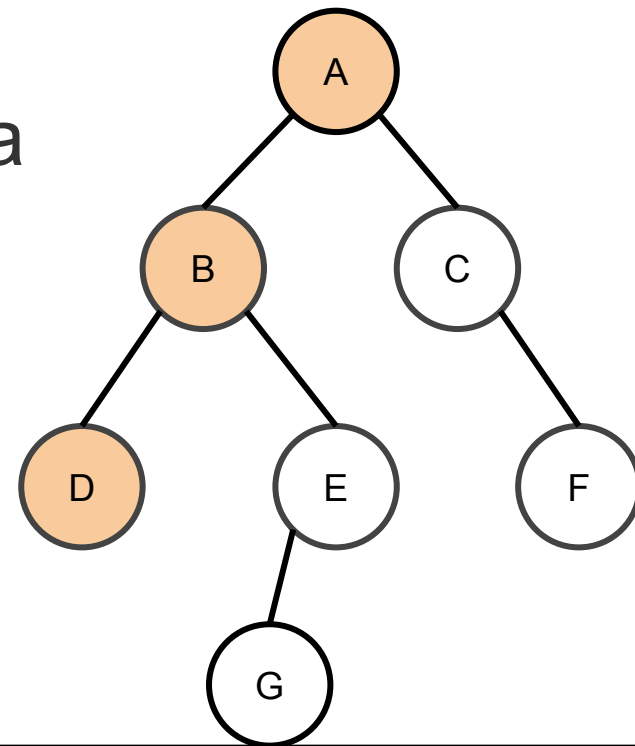
Grafos - DFS

- Recorrido en profundidad
- Seleccionamos A



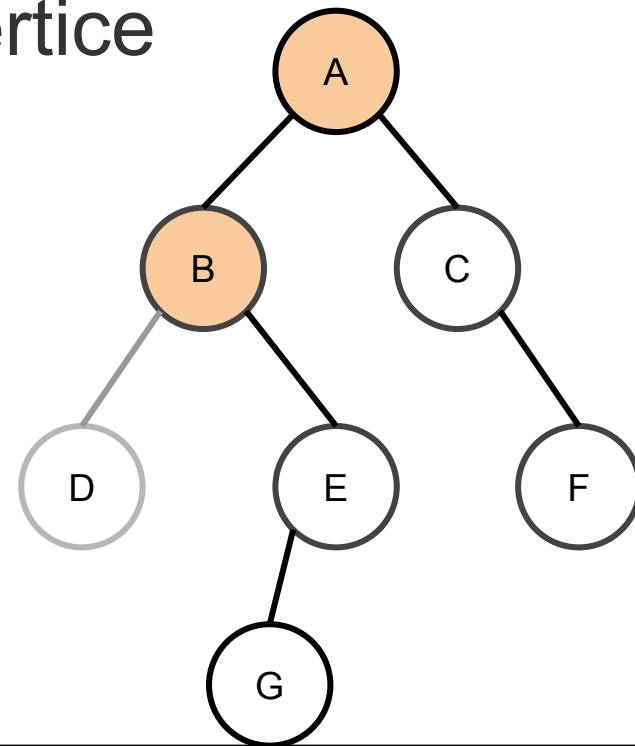
Grafos - DFS

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



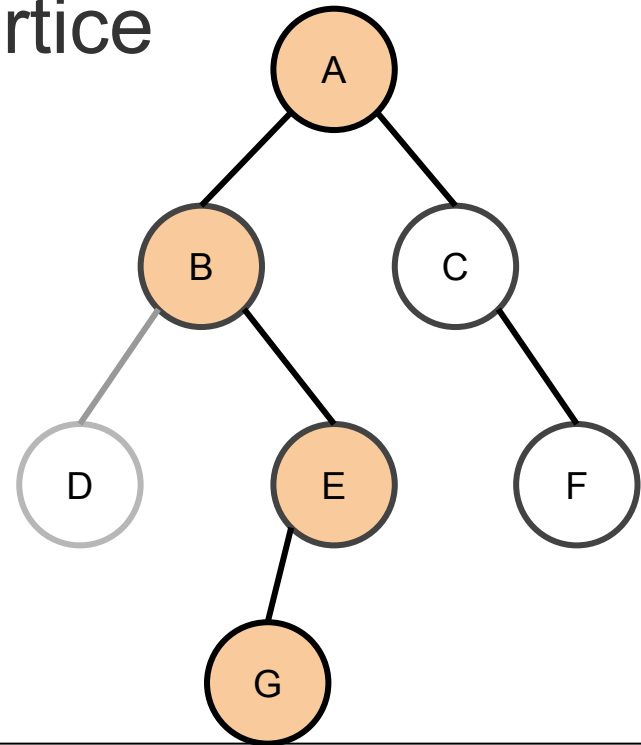
Grafos - DFS

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



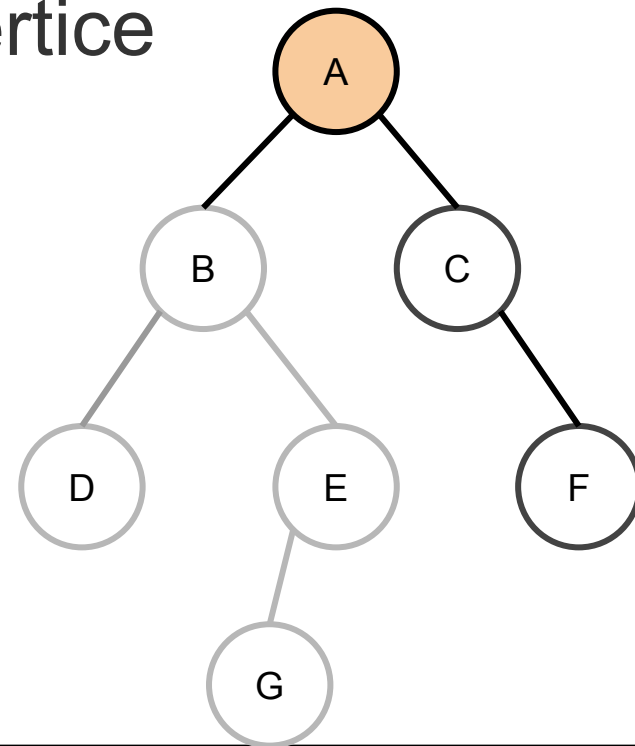
Grafos - DFS

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



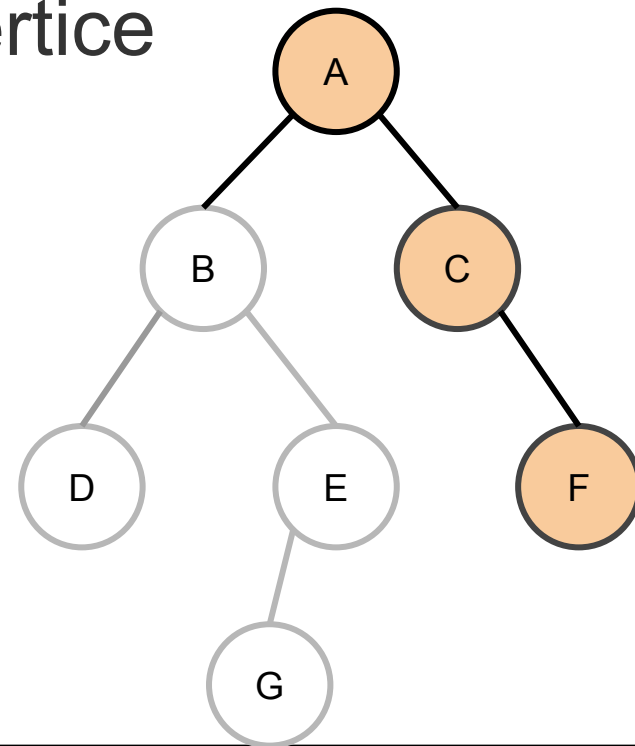
Grafos - DFS

- 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

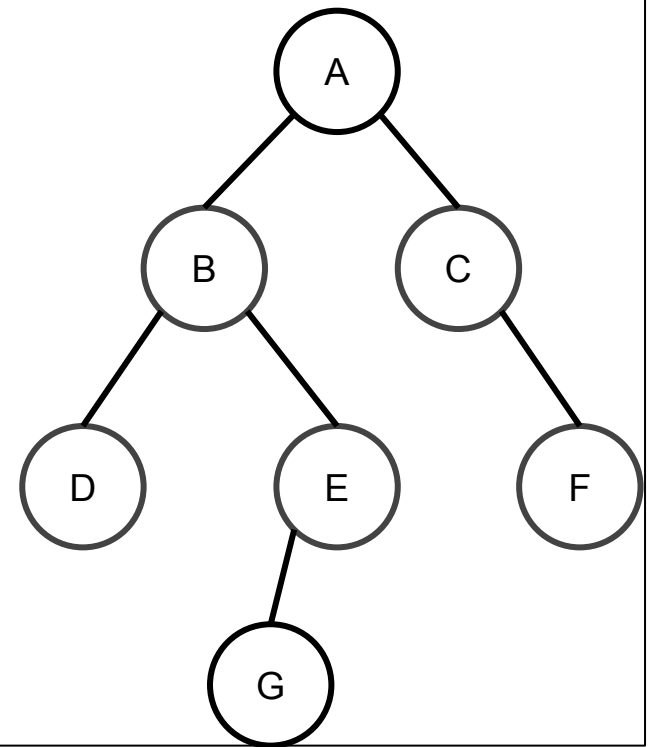
Grafos - DFS

- Inicialización: Elegimos un vértice inicial

`inicial = A`

`visitado[inicial]=true`

`pila.add(inicial)`



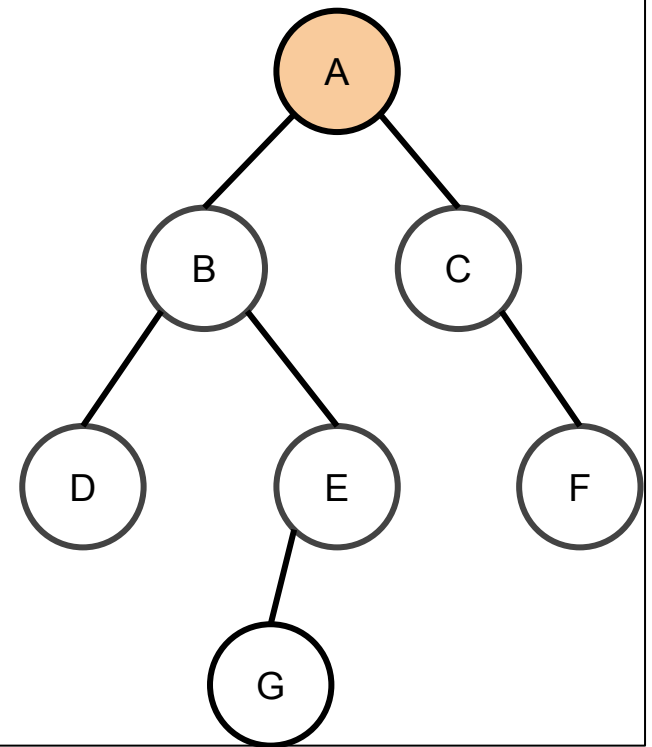
Grafos - DFS

- 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)
```

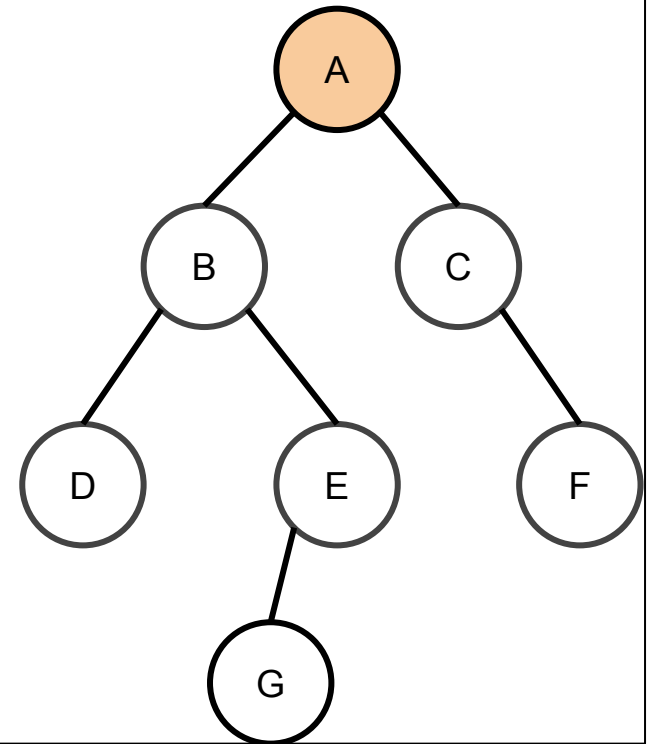
```
  v = pila.extraer()
```

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

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

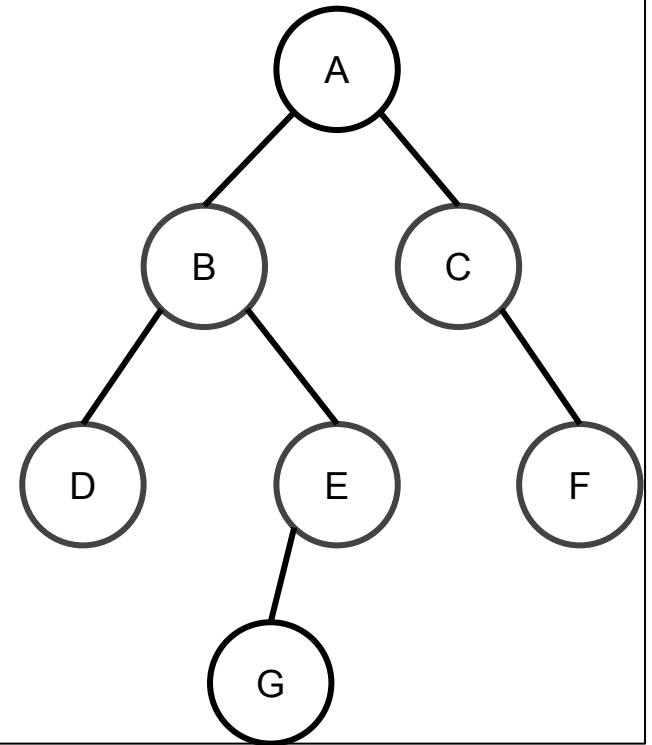
```
      visitado[ady]=true
```

```
      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)

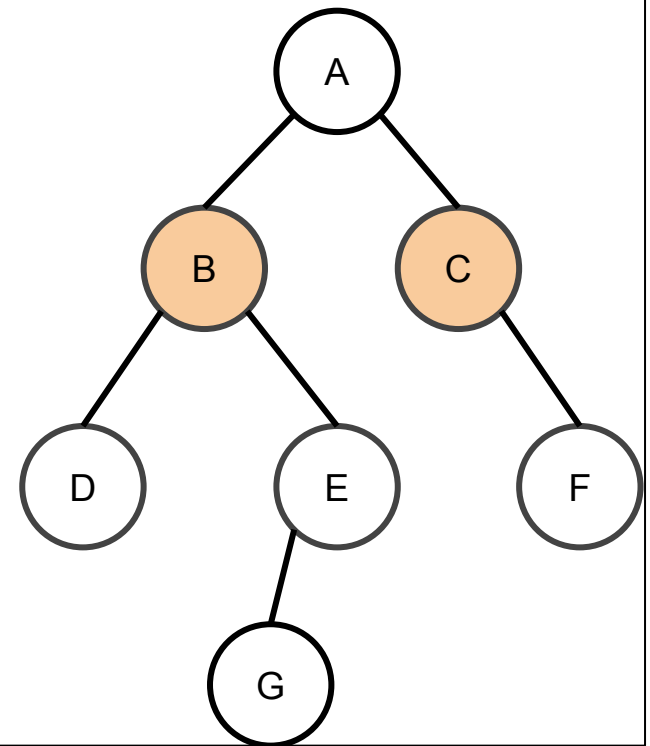
 v = pila.extraer()

 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)

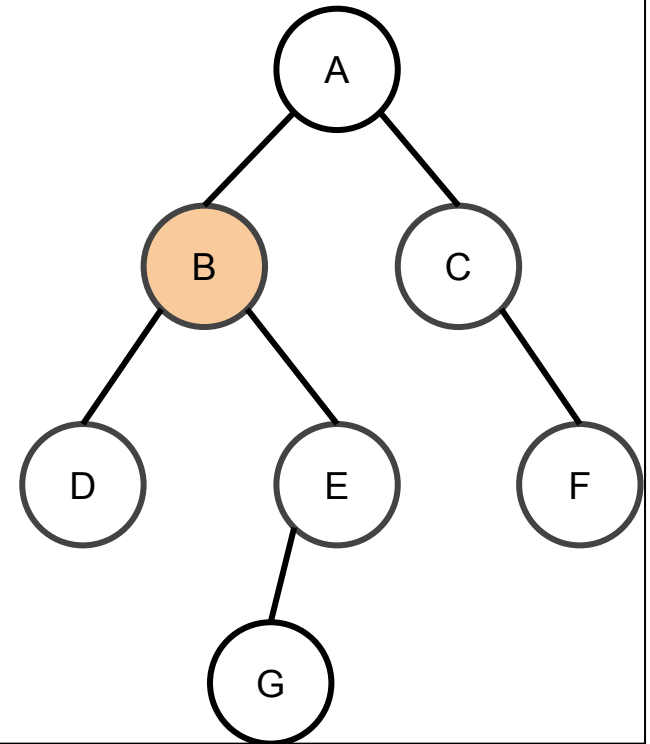
v = pila.extraer()

para cada ady de v:

if(no visitado[ady])

visitado[ady]=true

pila.add(ady)



Grafos - DFS

- Pila = {B,F}

mientras(pila.size() > 0)

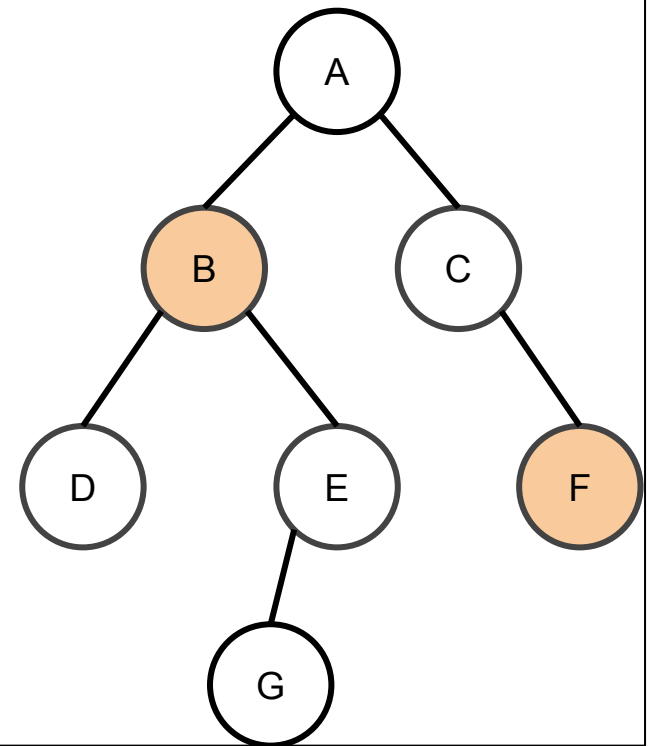
 v = pila.extraer()

 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)

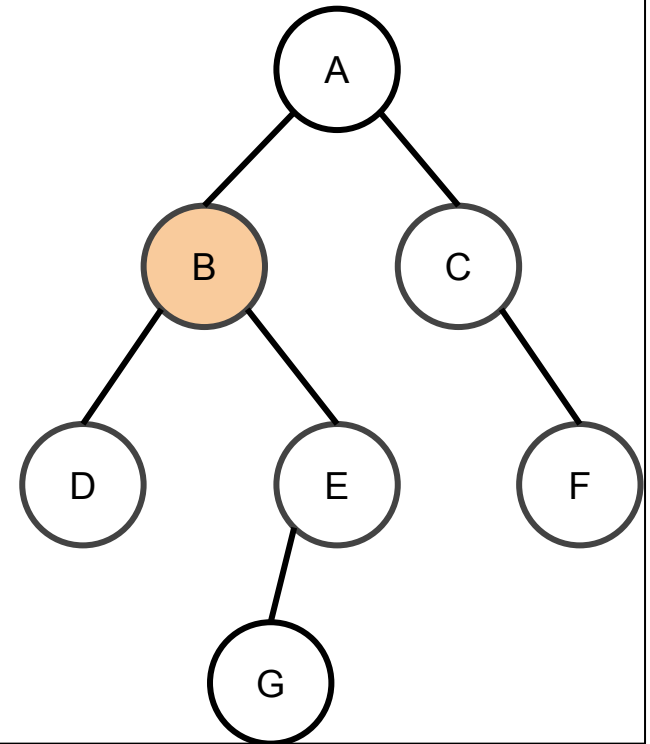
v = pila.extraer()

para cada ady de v:

if(no visitado[ady])

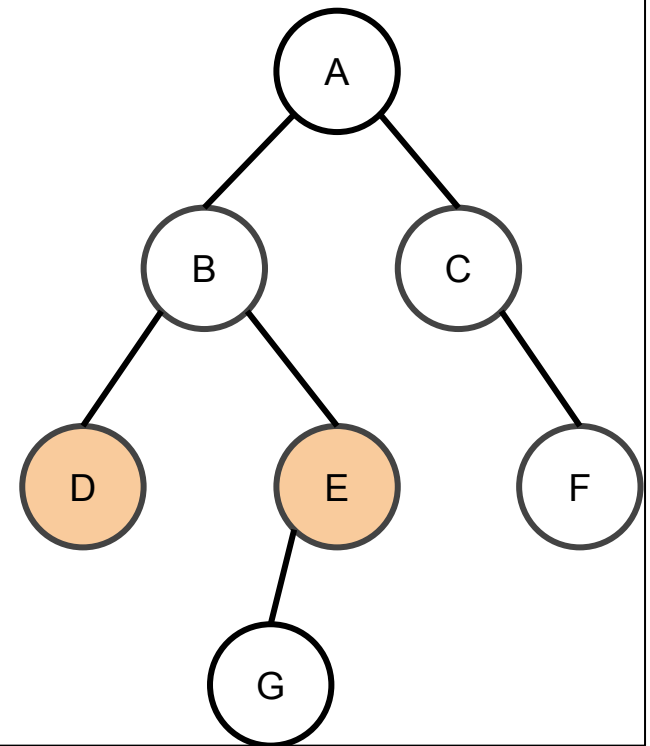
visitado[ady]=true

pila.add(ady)



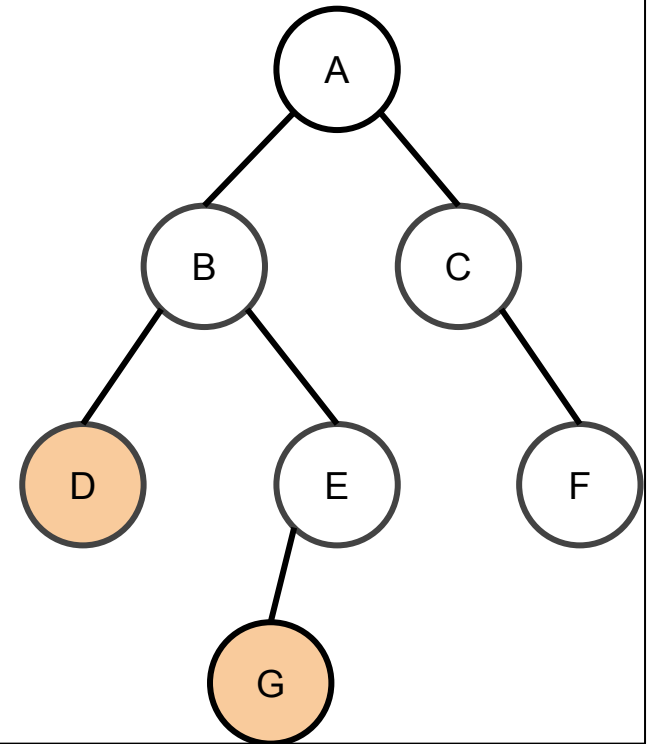
Grafos - DFS

- 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

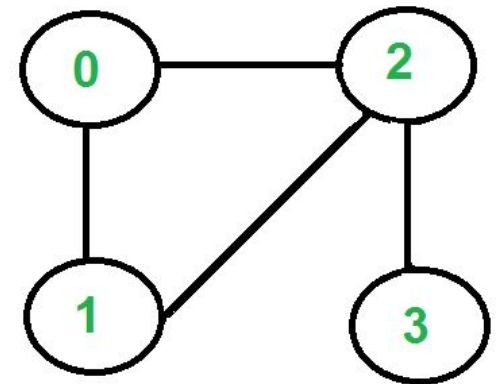
para cada ady de v:

si (no visitado[ady])

DFS(ady)

Grafos - Eulerianos

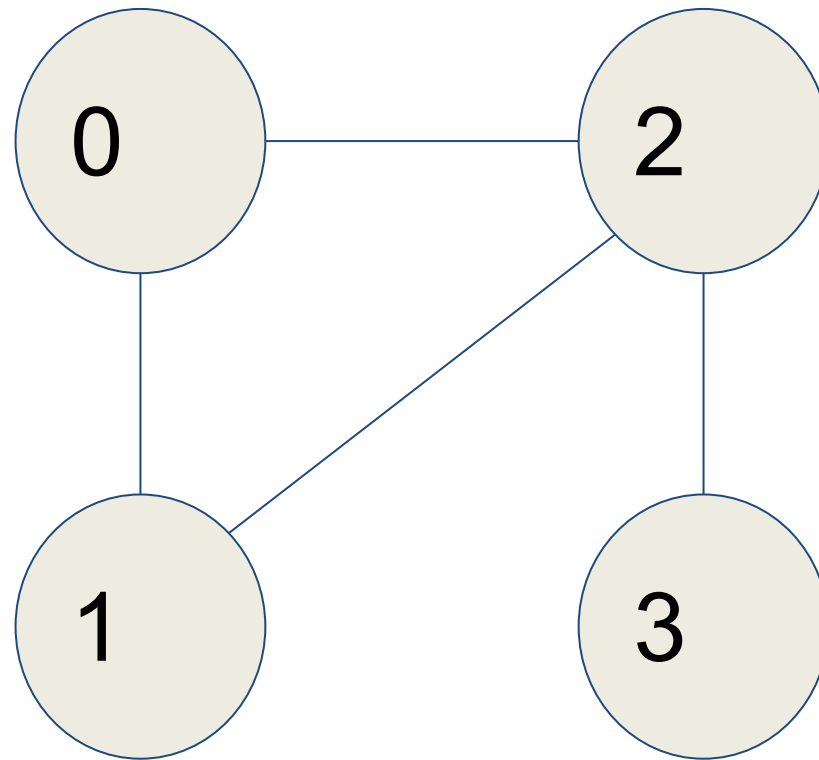
- Camino euleriano
 - ¿Se puede recorrer un grafo de manera que solo se recorra cada una de sus aristas solo **una** vez?



Grafos - Eulerianos

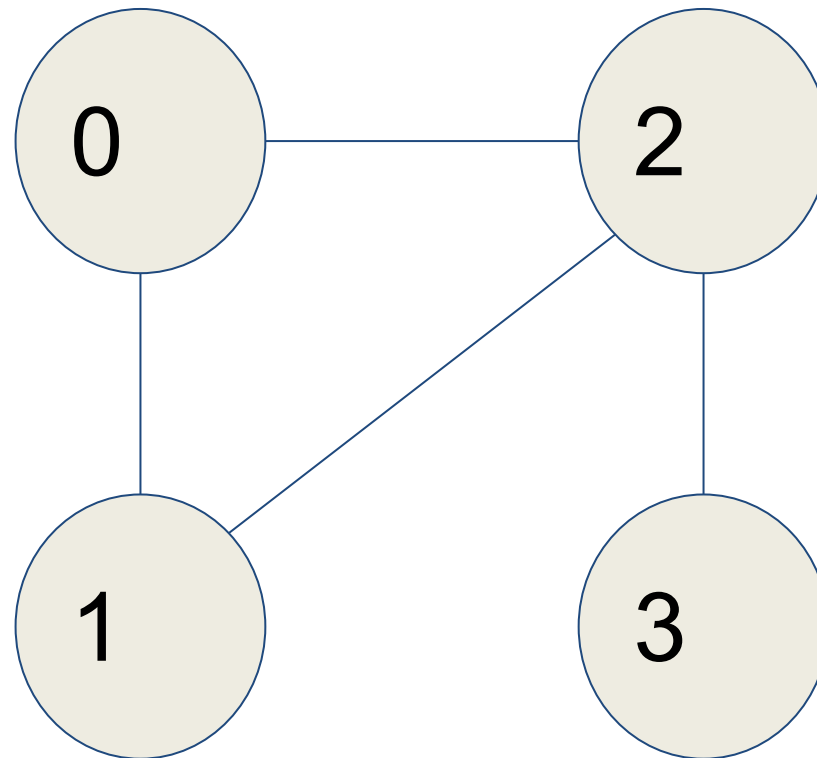
- Camino euleriano
 - Sea $C(v)$ el número de aristas que tiene el vértice v
 - Es camino euleriano si todos los vértices cumplen que $C(v)$ es par
 - Ó si $C(v)$ es impar en solo 2 vértices

Grafos - Eulerianos



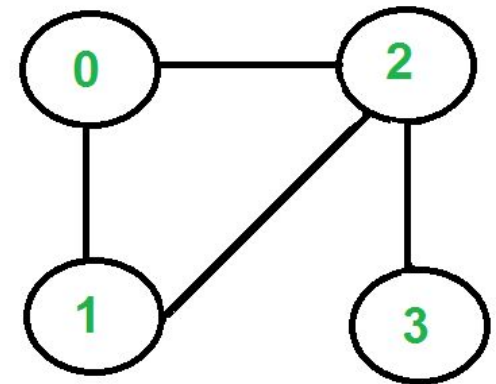
Grafos - Eulerianos

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



Grafos - Eulerianos

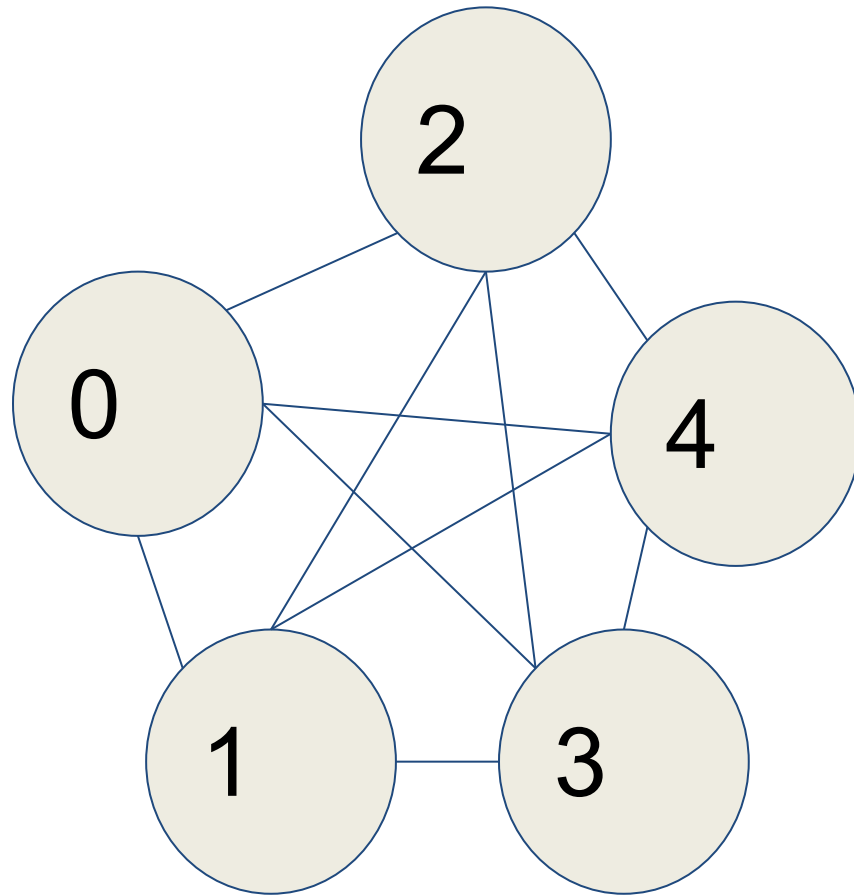
- Ciclo euleriano
 - ¿Se puede recorrer un grafo de manera que solo se recorra cada una de sus aristas solo **una** vez y llegar al mismo punto de inicio?



Grafos - Eulerianos

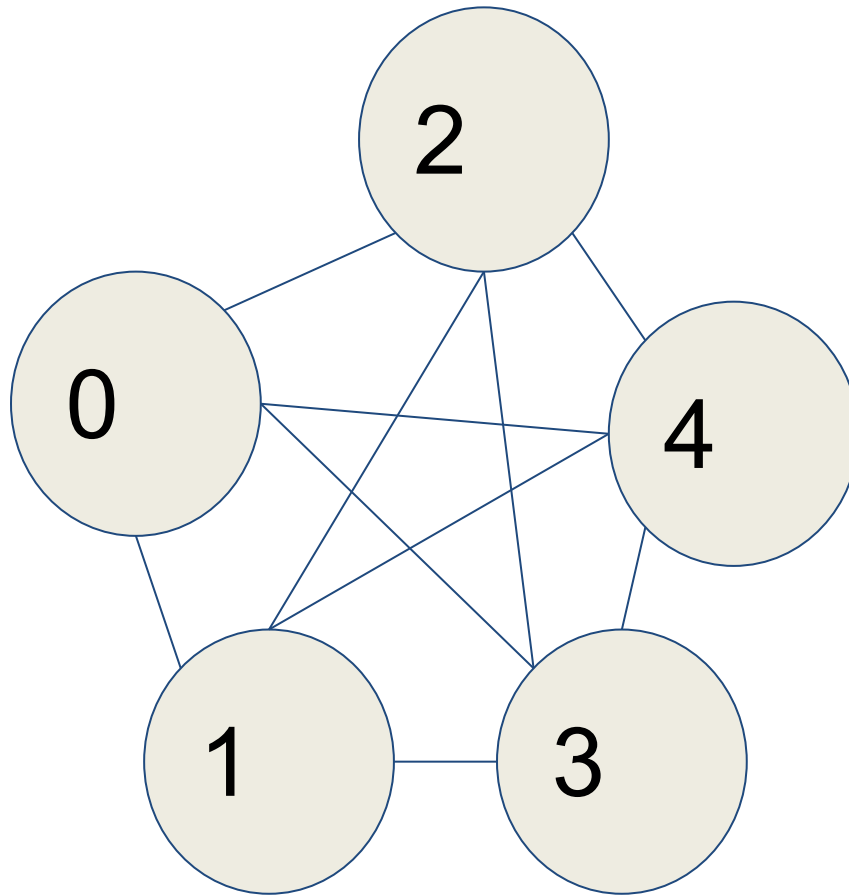
- Camino euleriano
 - Sea $C(v)$ el número de aristas que tiene el vértice v
 - Es camino euleriano si todos los vértices cumplen que $C(v)$ es par

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

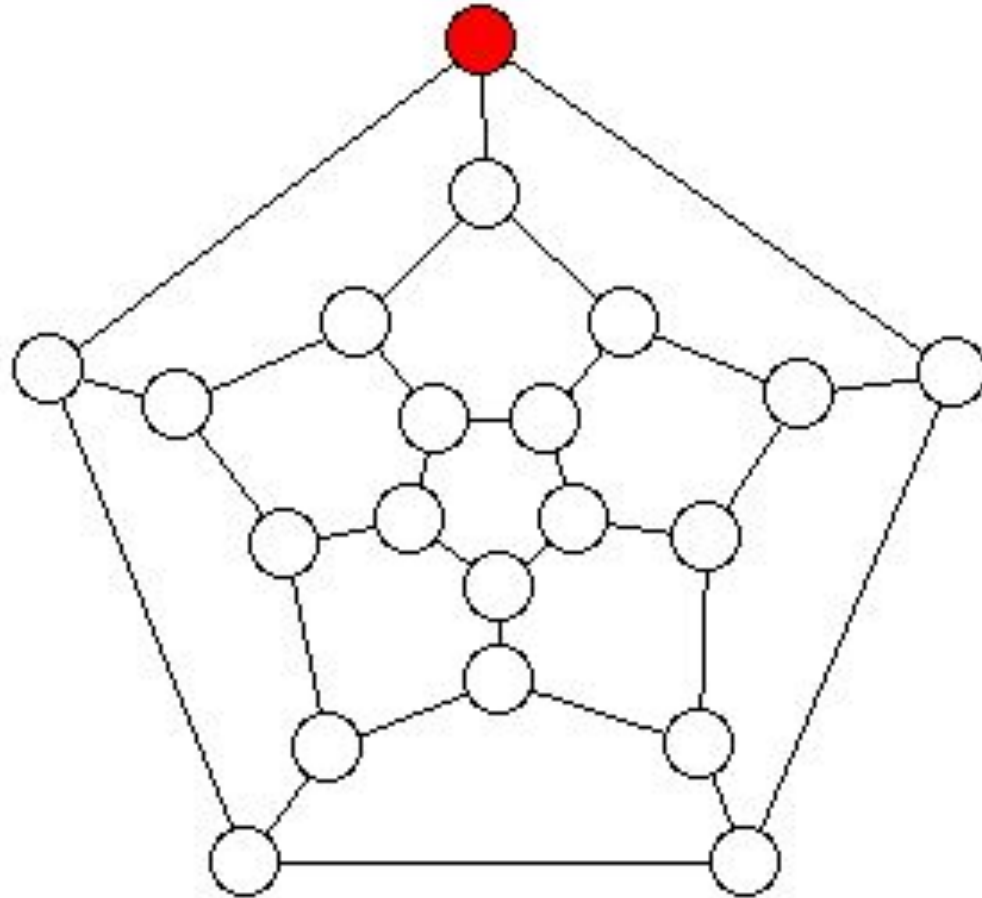
Grafos | Camino y ciclo Hamiltoniano

- 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. $7 \wedge 4 = 111 \wedge 100 \Rightarrow (011)$

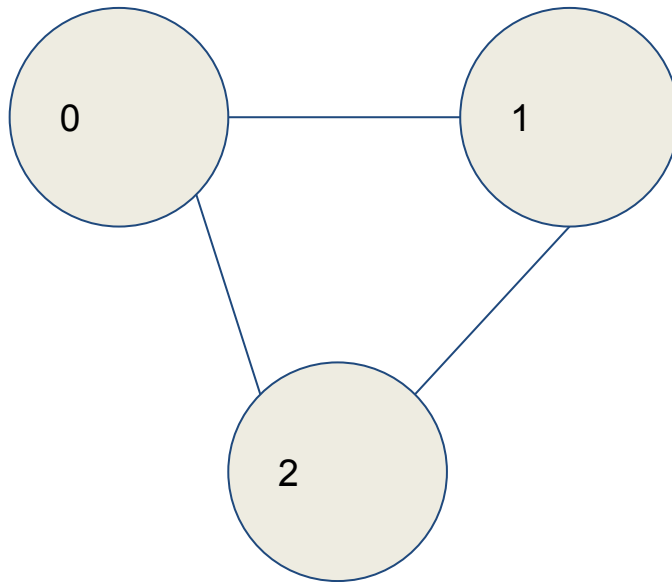
Grafos | Camino y ciclo Hamiltoniano

- 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

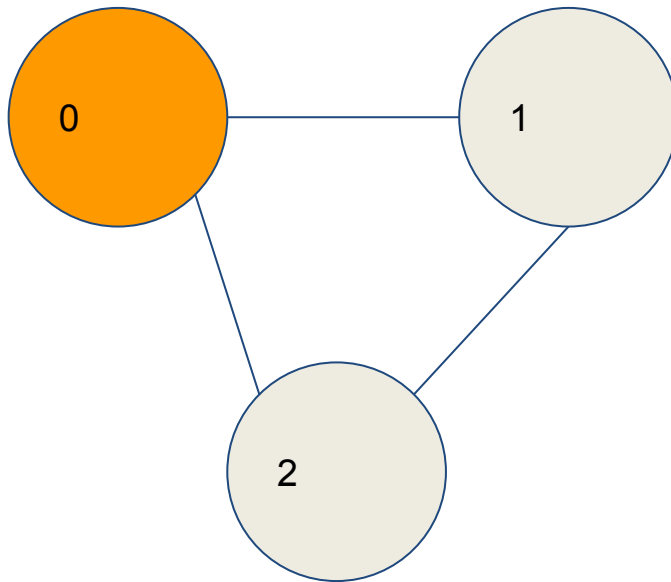


Grafos | Camino y ciclo Hamiltoniano



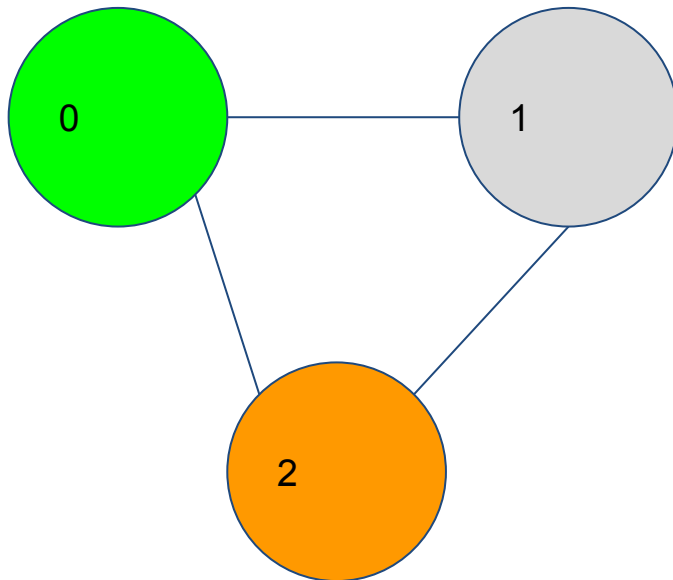
NODO=0,
MASK=111₂

Grafos | Camino y ciclo Hamiltoniano



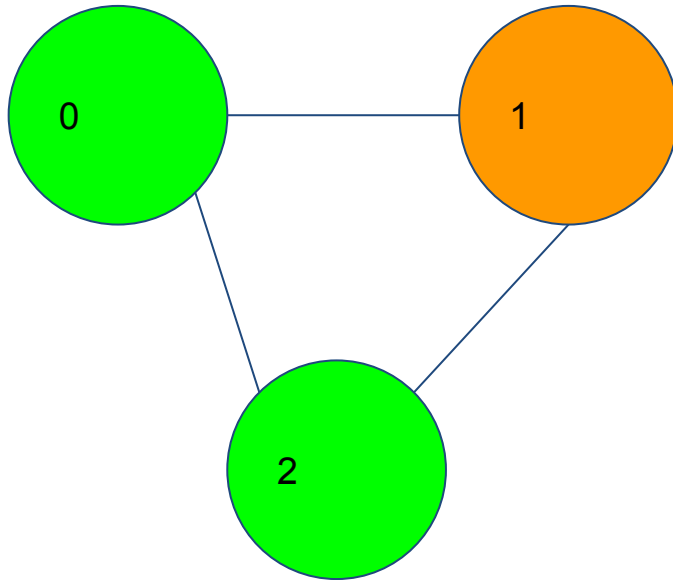
NODO=0,
MASK=111₂
2⁰ = 1
111 & 001 > 0?

Grafos | Camino y ciclo Hamiltoniano



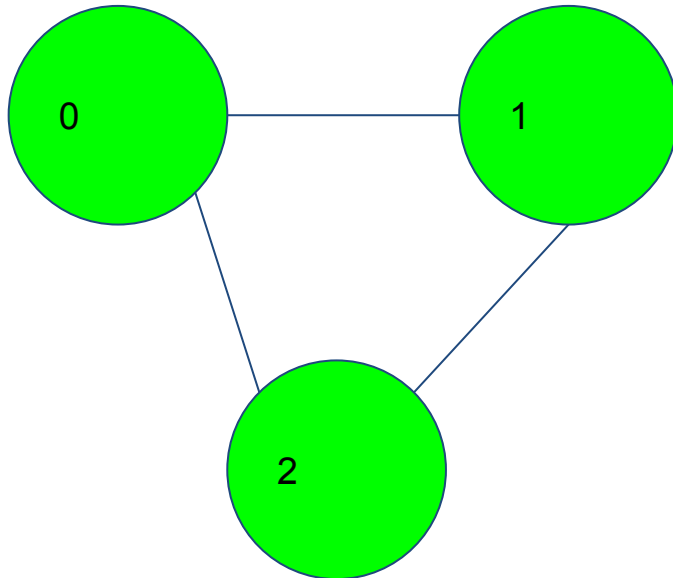
NODO=2,
MASK=110₂
2² = 4 (100)
110 & 100 > 0?

Grafos | Camino y ciclo Hamiltoniano



NODO=1,
MASK=010₂
2¹ = 2 (010)
010 & 010 > 0?

Grafos | Camino y ciclo Hamiltoniano



NODO=X,
MASK=000₂
Es camino
Hamiltoniano

¿Es también ciclo?

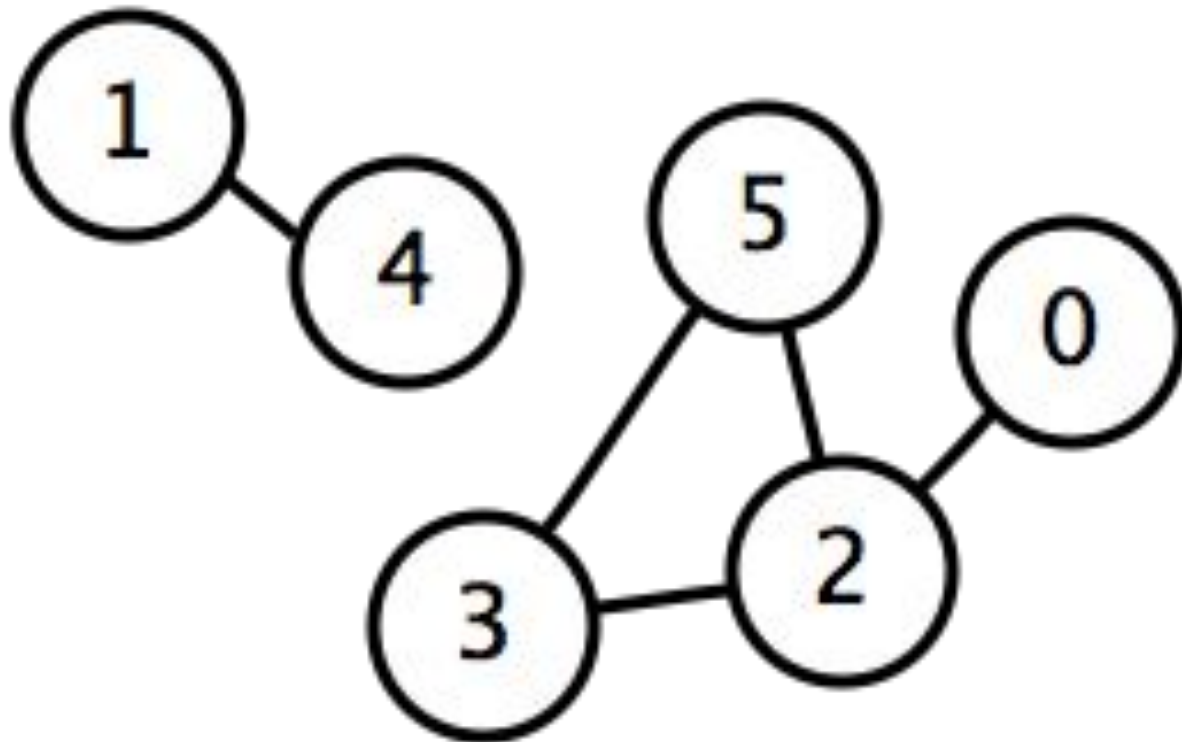
Grafos | Componentes Conexas

- 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

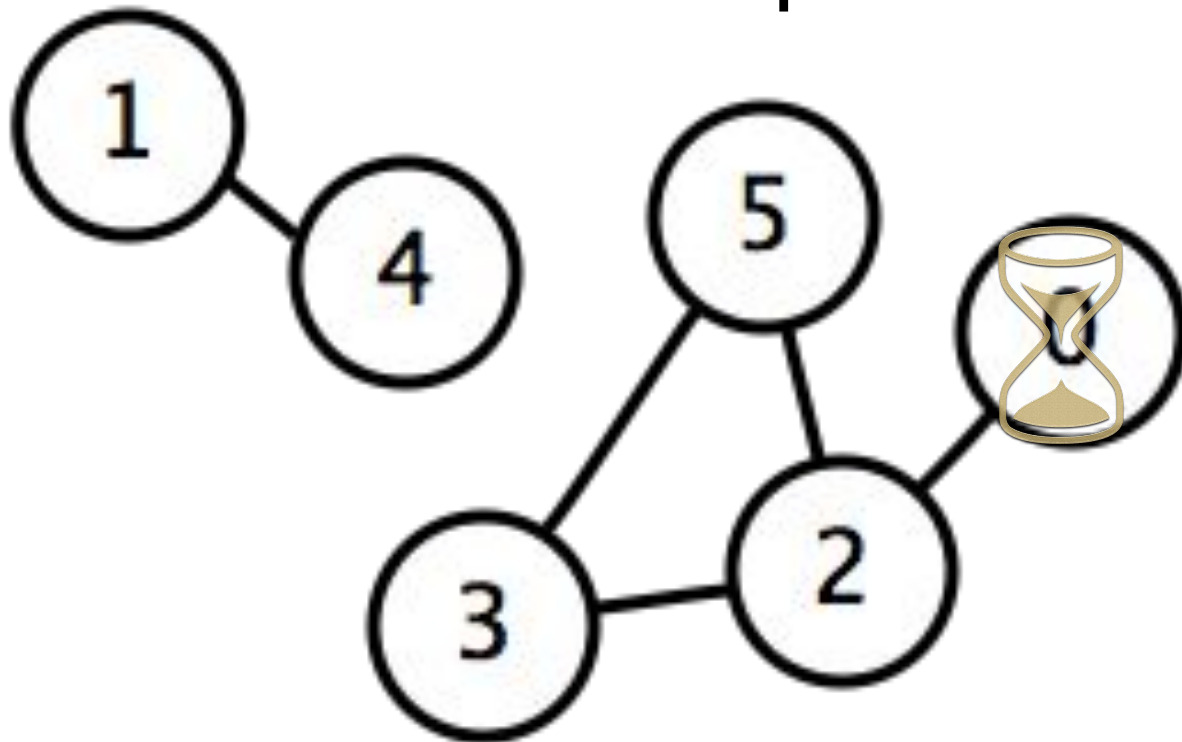
- La idea básica es hacer un BFS/DFS por cada vértice i desde $0..N$ siempre y cuando i no haya sido recorrido por un algoritmo anterior
- Se cuenta 1 y se recorre i

Grafos | Componentes Conexas

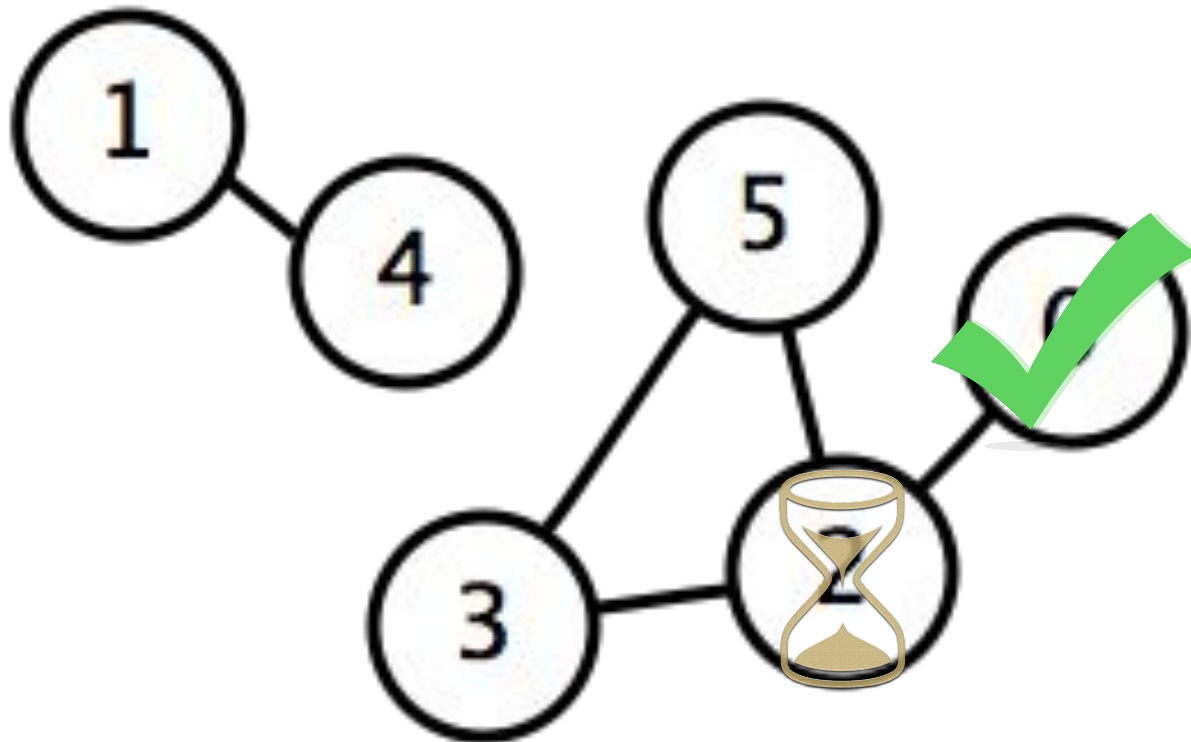


Grafos | Componentes Conexas

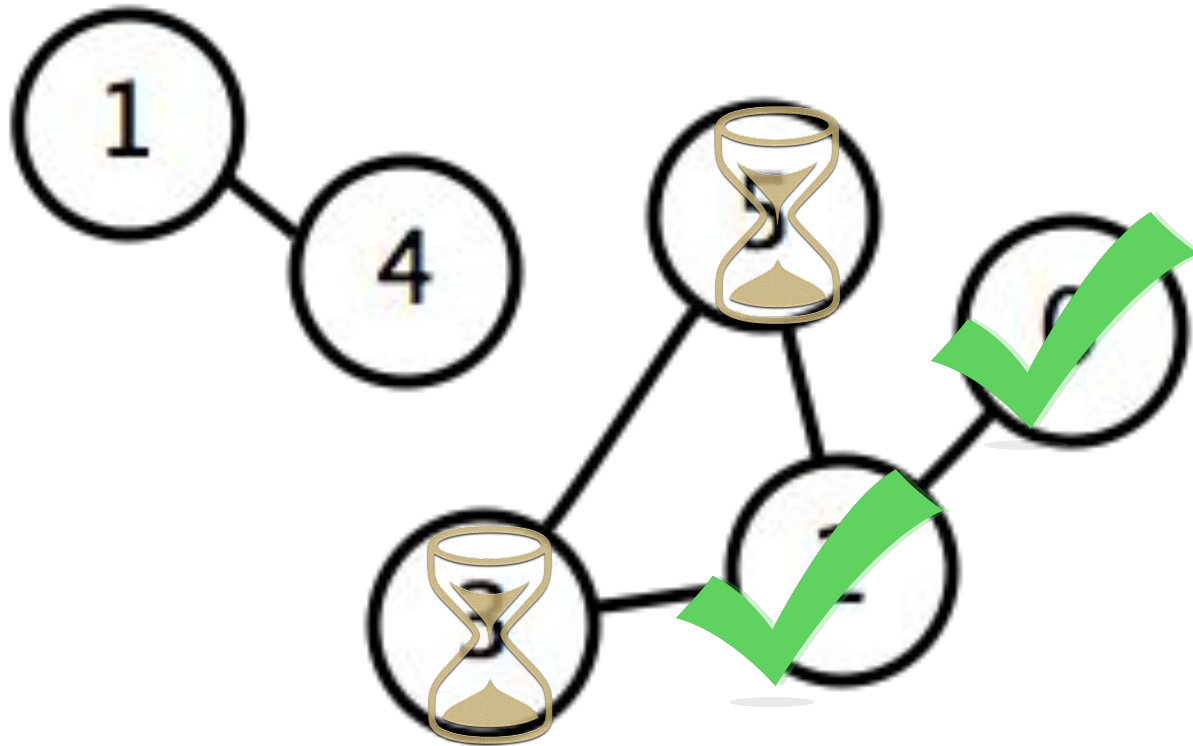
+1 Componente Conexa (1)



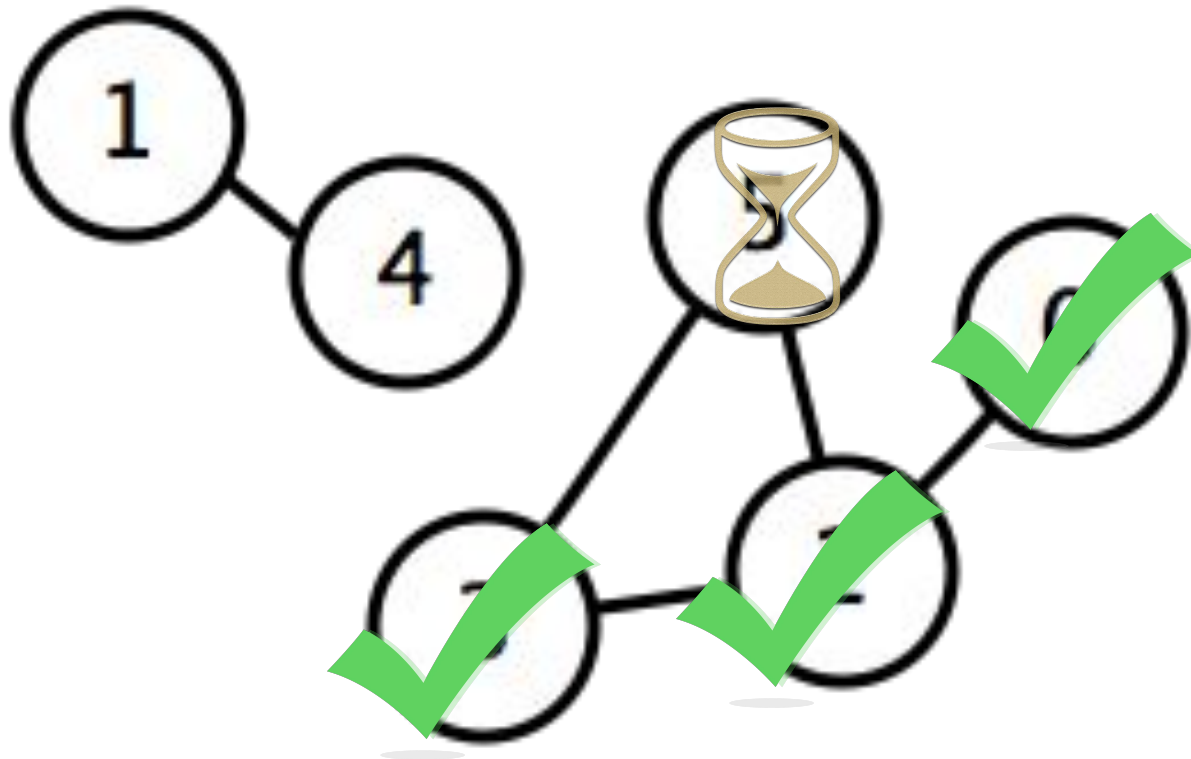
Grafos | Componentes Conexas



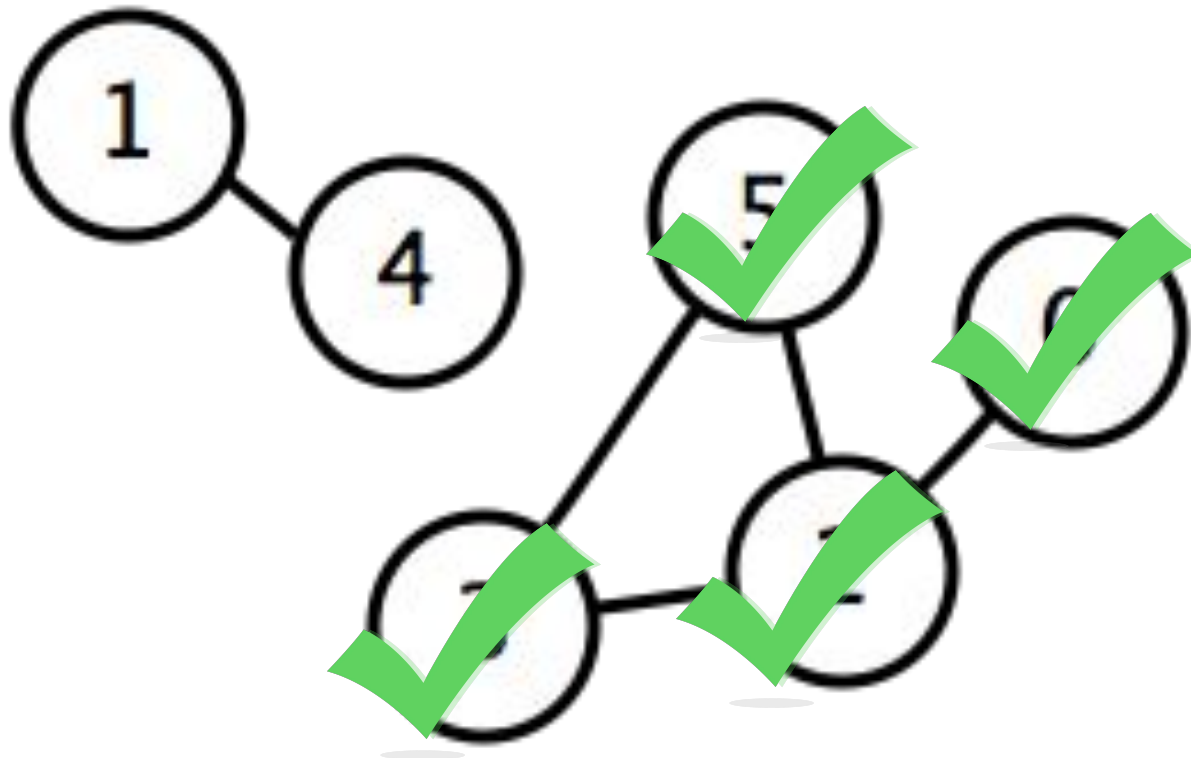
Grafos | Componentes Conexas



Grafos | Componentes Conexas

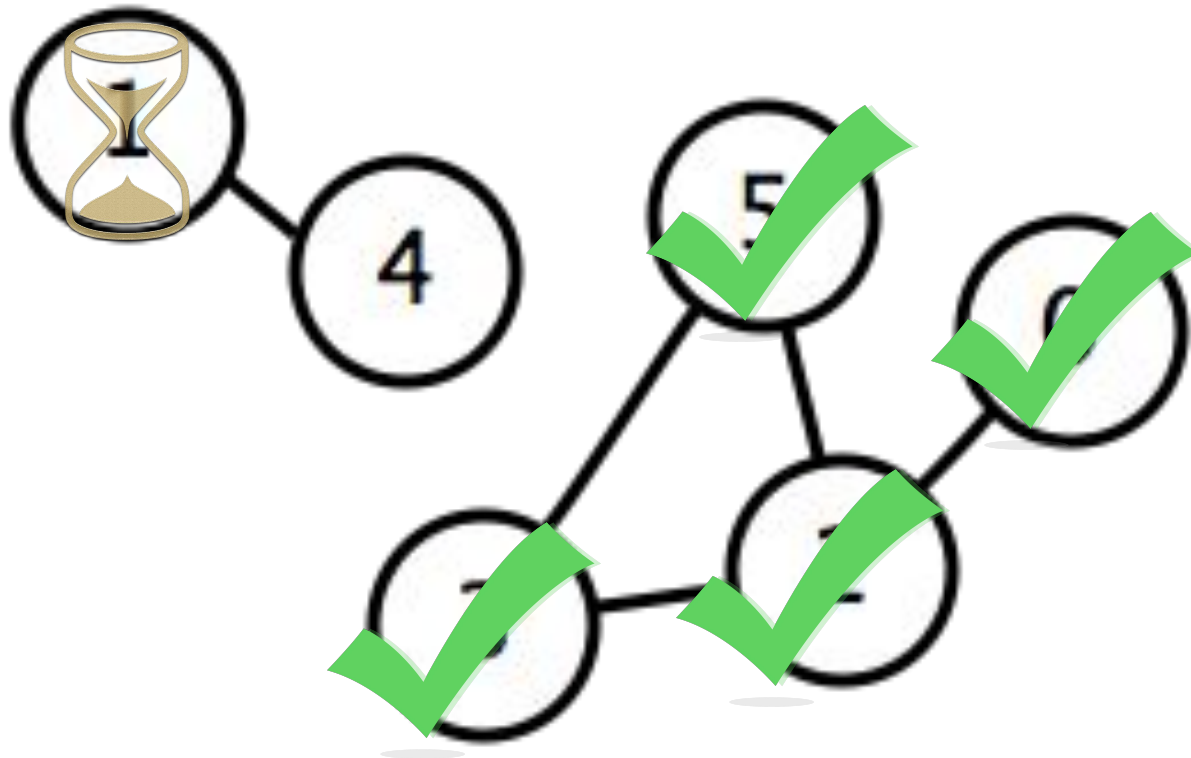


Grafos | Componentes Conexas

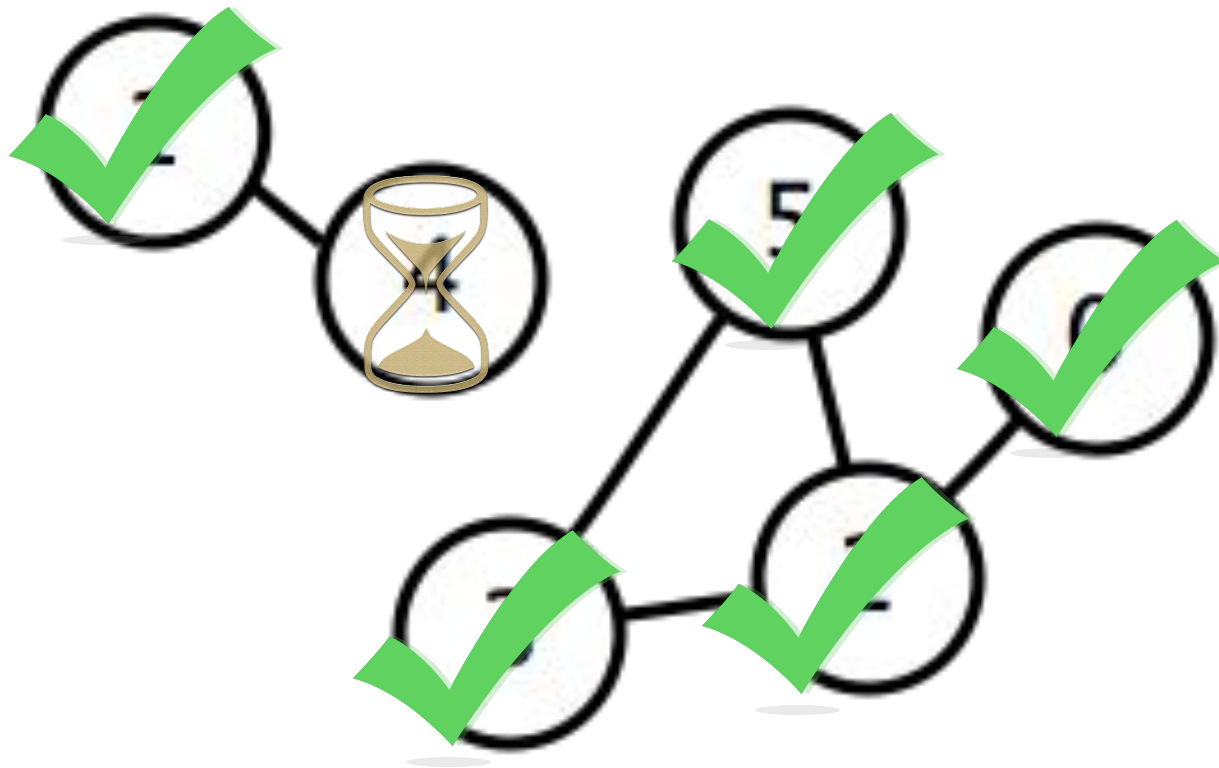


Grafos | Componentes Conexas

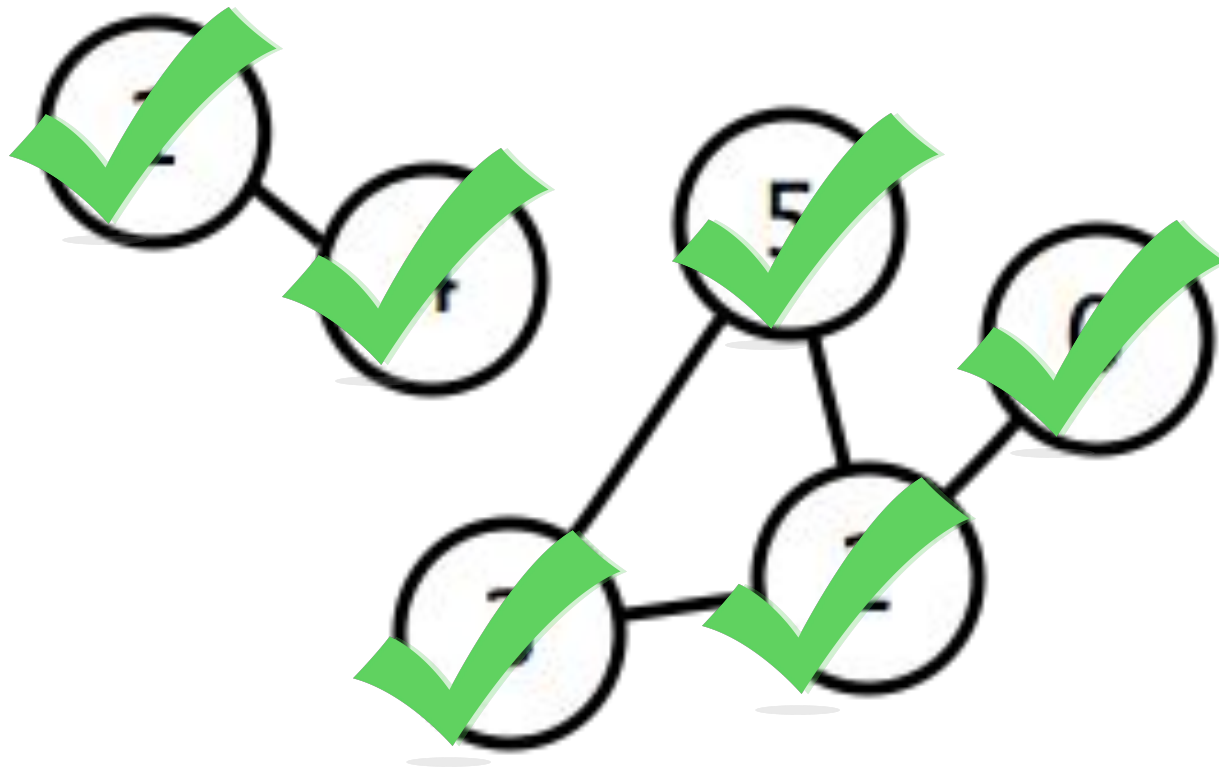
+1 Componente Conexo (2)



Grafos | Componentes Conexas



Grafos | Componentes Conexas



Grafos | Componentes Conexas

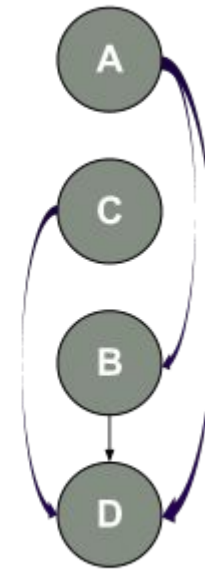
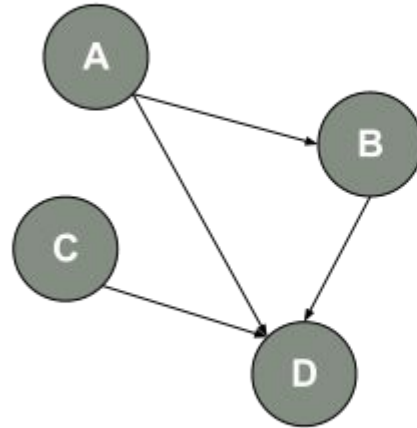
- Se empieza desde 0 y se recorre:
 - 0-2-5-3
- Al estar 1 no visitado se recorre los adyacentes de 1
 - 1-4
- Al estar 2,3,4,5 no visitado se ignoran
- Hay dos componentes conexas

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

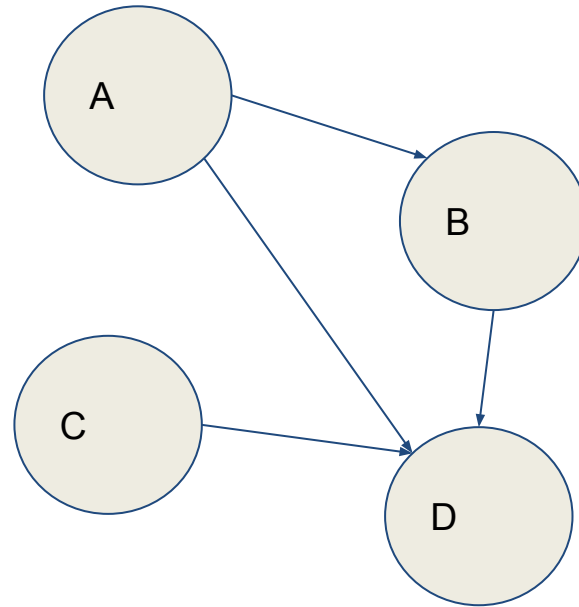
Grafos | Ordenamiento Topológico

DAG AND ITS TOPOLOGICAL SORT



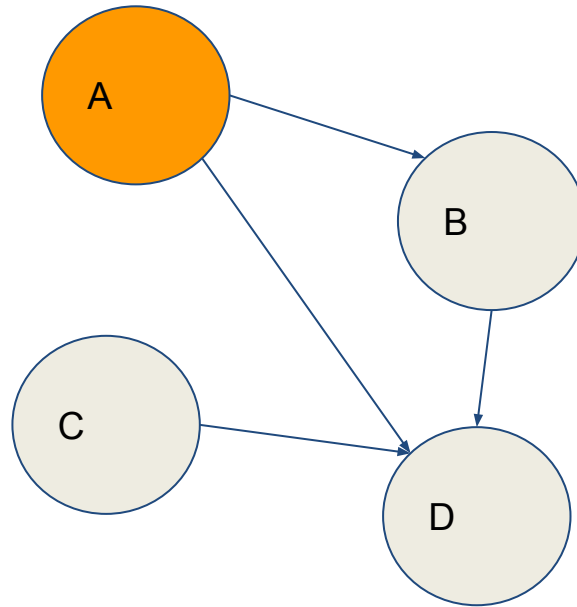
The above graph can be "sorted" as follows: [A, C, B, D]. That means that if the edge (A, B) exists, A must precede B in the final order!

Grafos | Ordenamiento Topológico



A y C no les incide ningún otro nodo

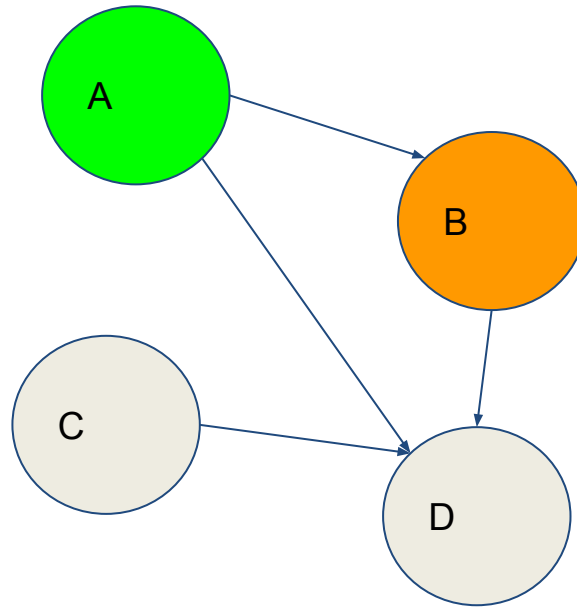
Grafos | Ordenamiento Topológico



TS = { }

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

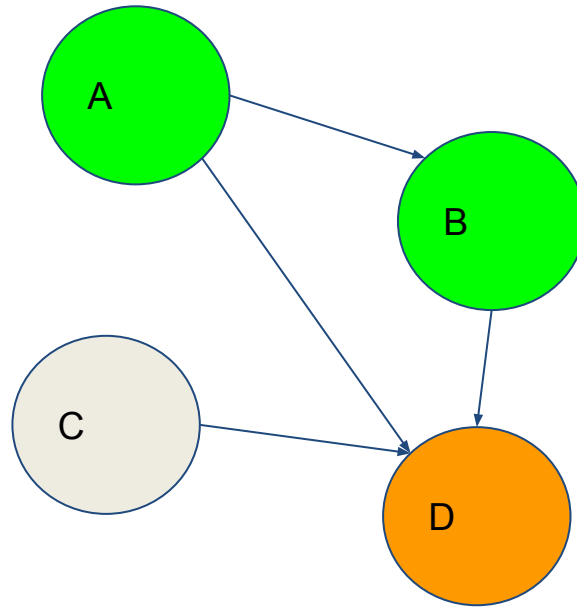
Grafos | Ordenamiento Topológico



TS = { }

DFS(B) = DFS(D)

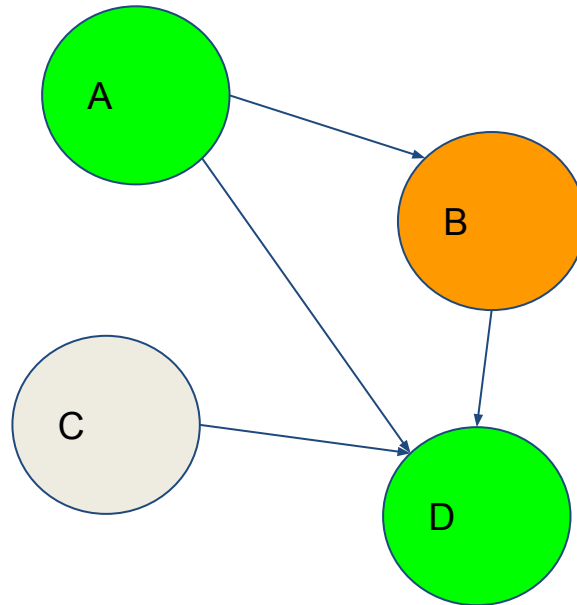
Grafos | Ordenamiento Topológico



TS = {D}

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

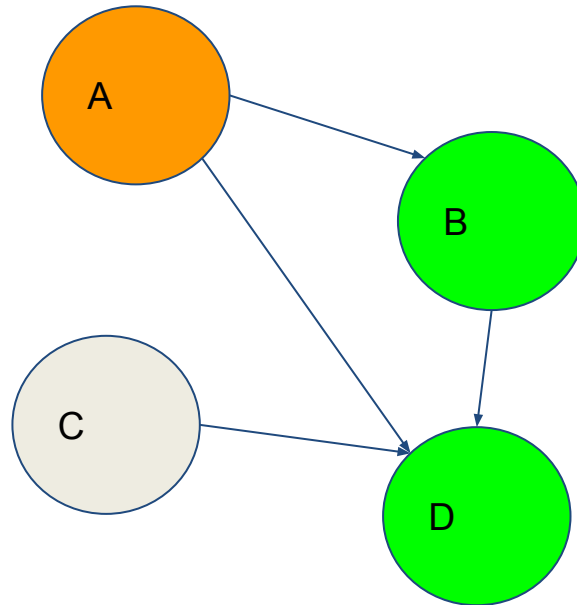
Grafos | Ordenamiento Topológico



TS = { B, D }

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

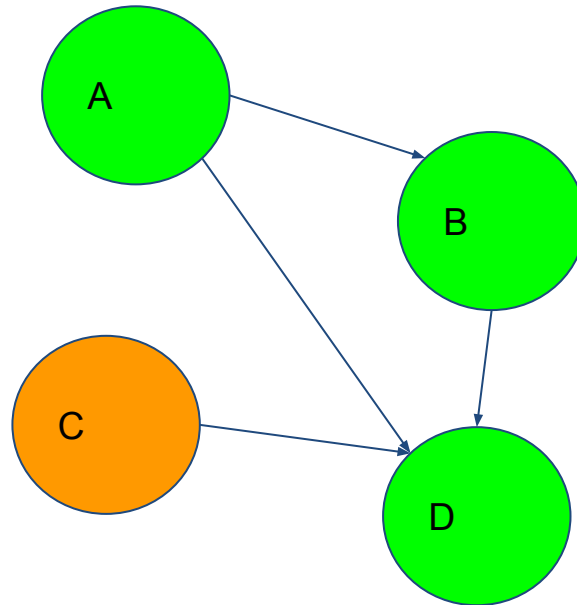
Grafos | Ordenamiento Topológico



TS = { A, B, D }

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

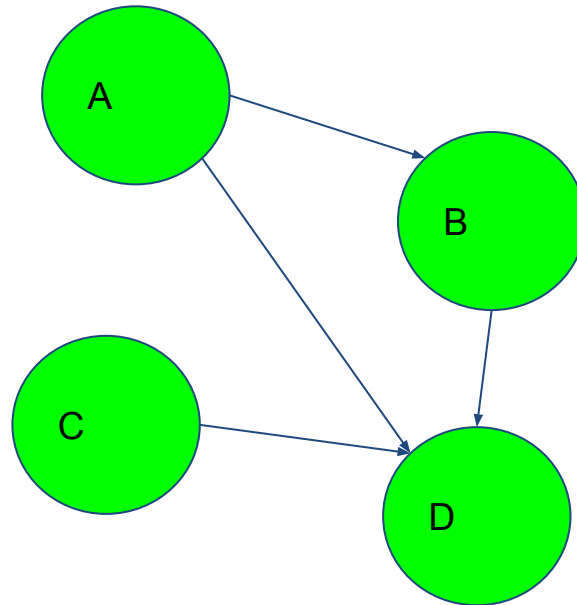
Grafos | Ordenamiento Topológico



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 que inciden en el nodo N
- 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 | Componentes Fuertemente Conexas

- Para cada par de nodos u, v en un grafo dirigido
- Se puede llegar desde u , v a través de 0 o más nodos intermedios
- Algoritmo de Kosaraju

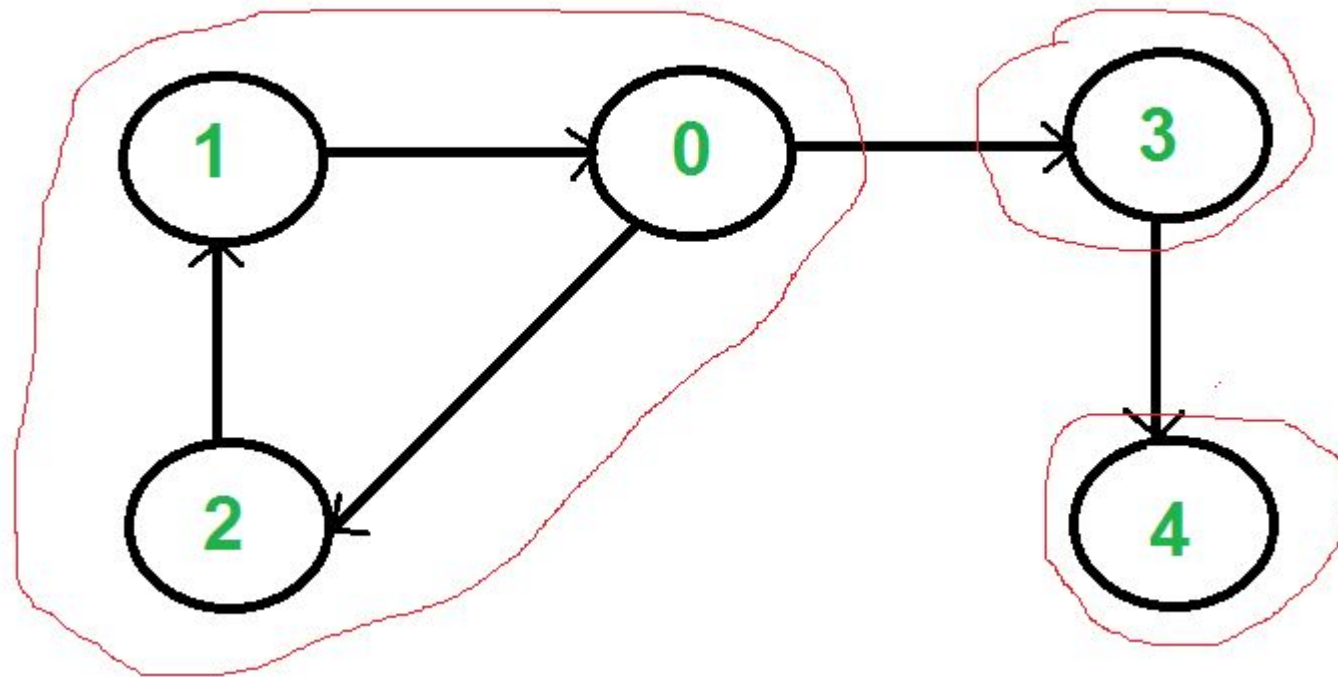
Grafos | Componentes Fuertemente Conexas

- Algoritmo de Kosaraju
 - Llevar una transposición del grafo (grafo invertido)
 - Realizar un ordenamiento topológico del grafo dirigido en cuestión
 - Desapilar el i -ésimo nodo de la pila y realizar un DFS desde ese nodo hacia todos los demás en el grafo invertido

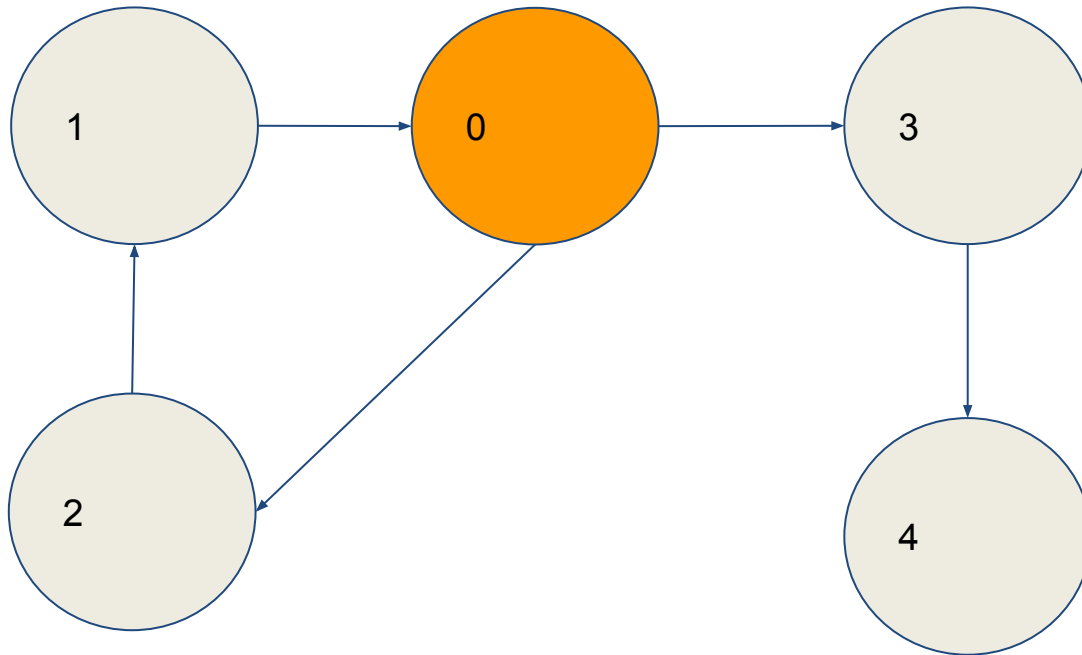
Grafos | Componentes Fuertemente Conexas

- Algoritmo de Kosaraju
 - Si ya ha sido visitado el j -ésimo nodo del grafo revertido, no tomarlo en cuenta
 - Finalmente, contar 1 por cada vez que se realice un DFS

Grafos | Componentes Fuertemente Conexas



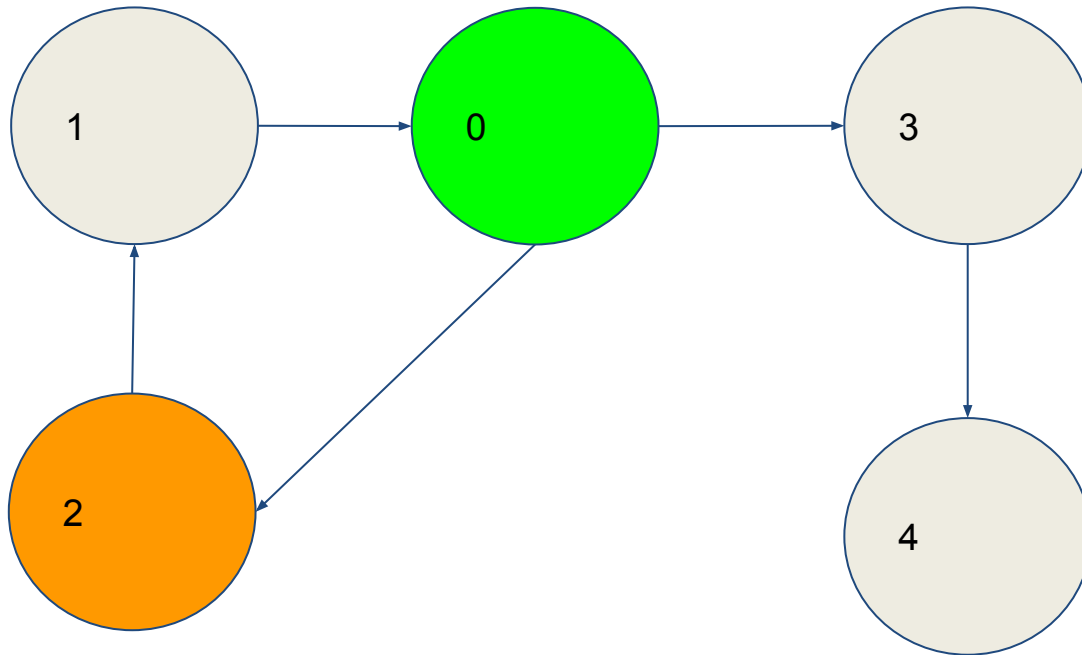
Grafos | Componentes Fuertemente Conexos



TS = { }

DFS(0) => DFS(2), DFS(3)

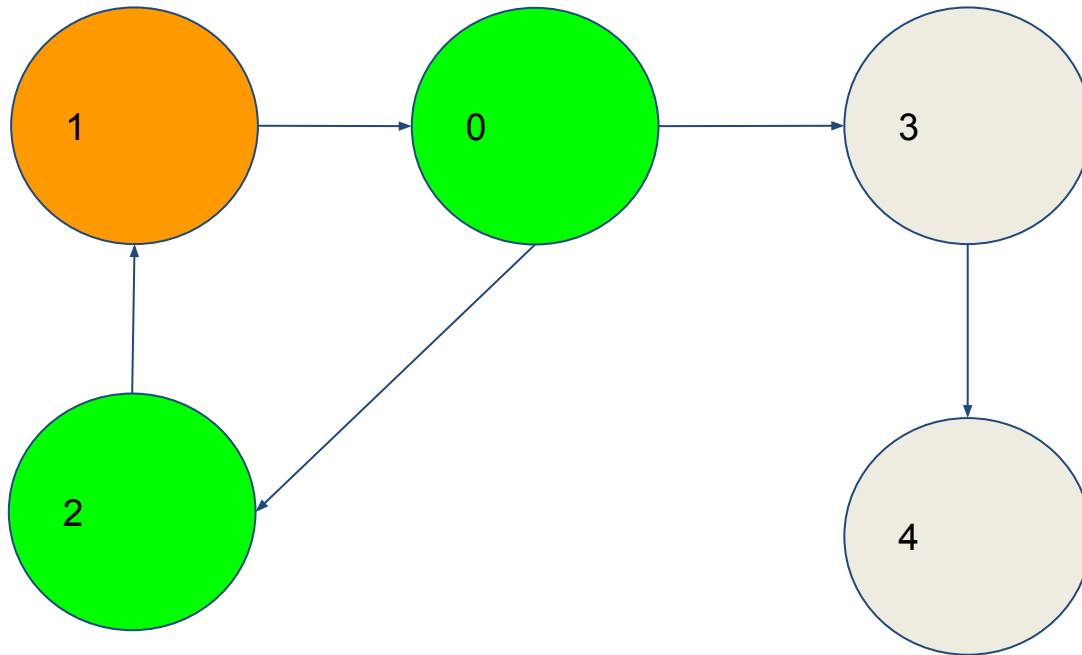
Grafos | Componentes Fuertemente Conexas



TS = { }

DFS(2) => DFS(1)

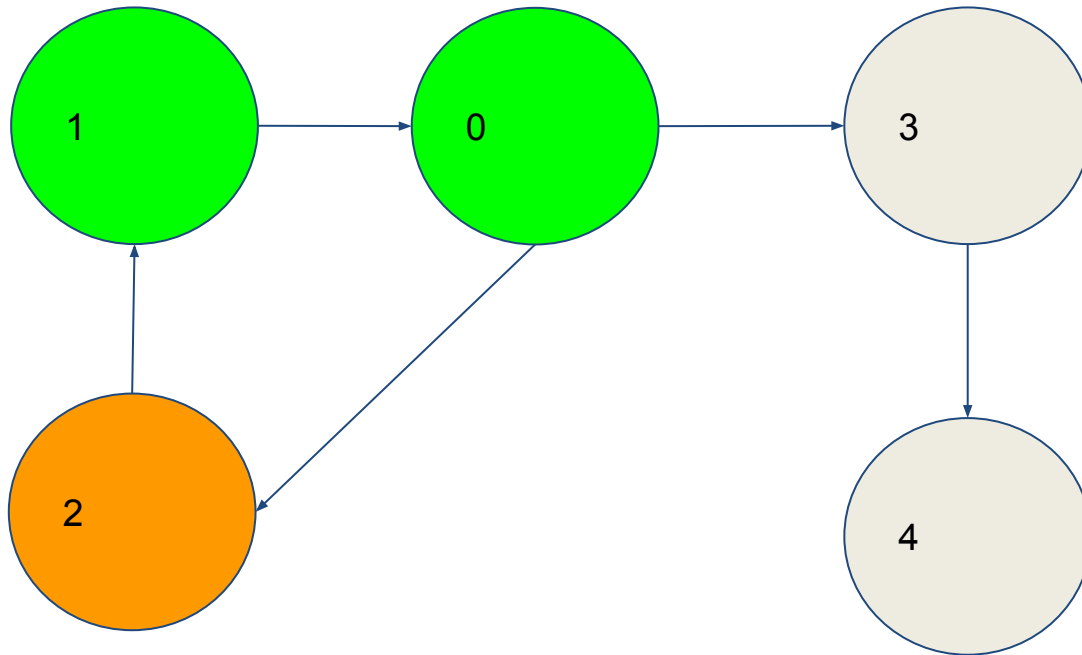
Grafos | Componentes Fuertemente Conexas



TS = { 1 }

DFS(1) => DFS(0), TS.push(1)

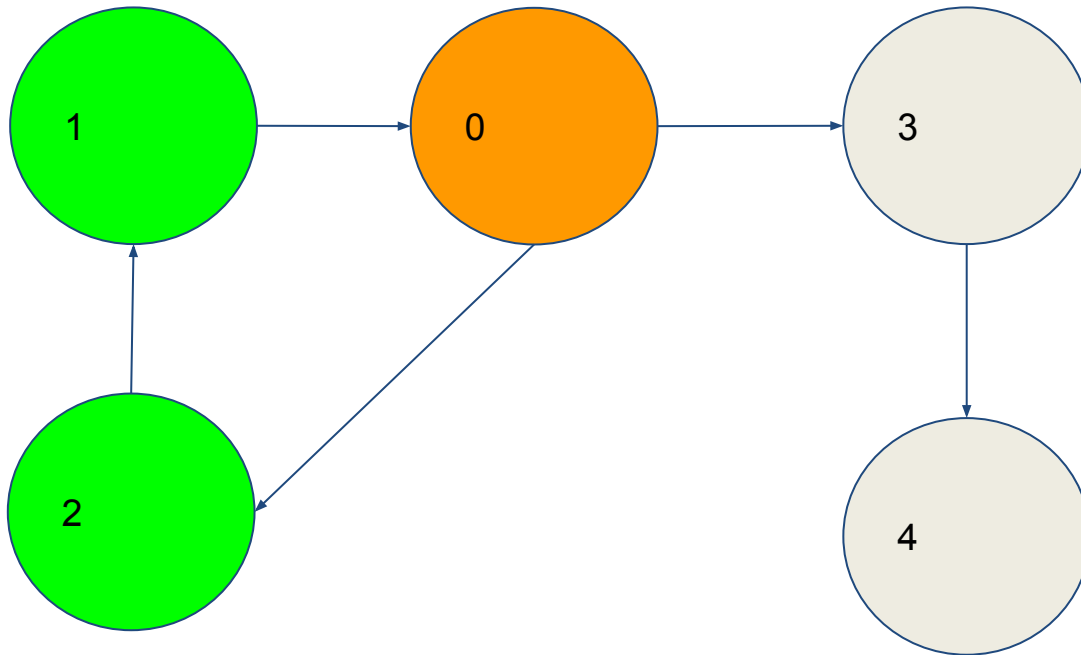
Grafos | Componentes Fuertemente Conexas



TS = {
2, 1
}

DFS(2) => DFS(1), TS.push(2)

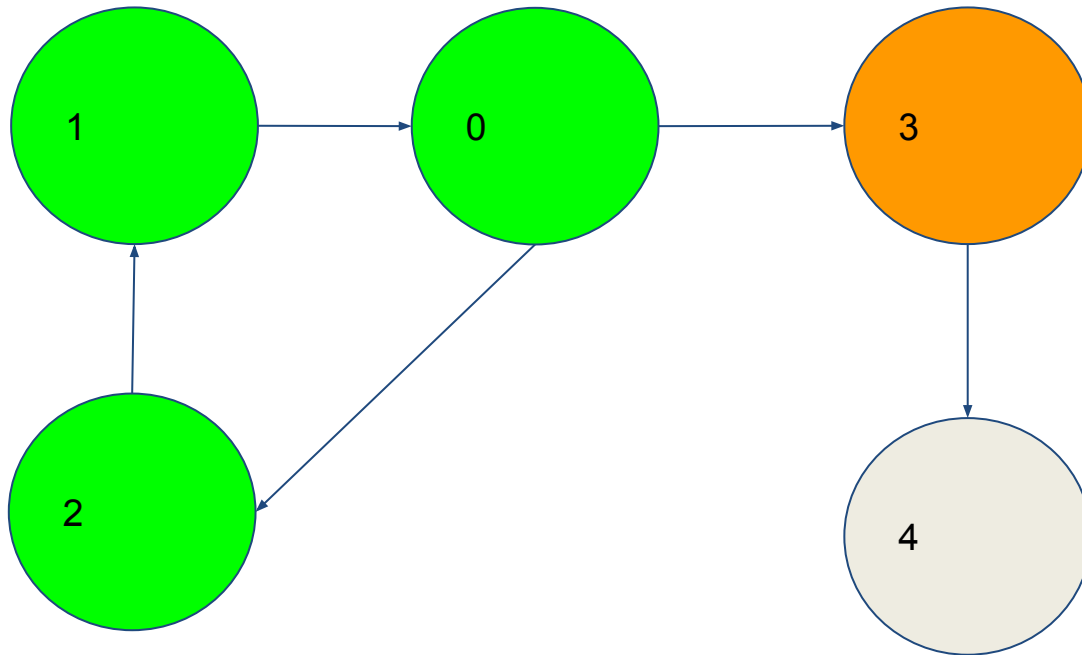
Grafos | Componentes Fuertemente Conexas



TS = {
2, 1
}

DFS(0) => DFS(2), DFS(3)

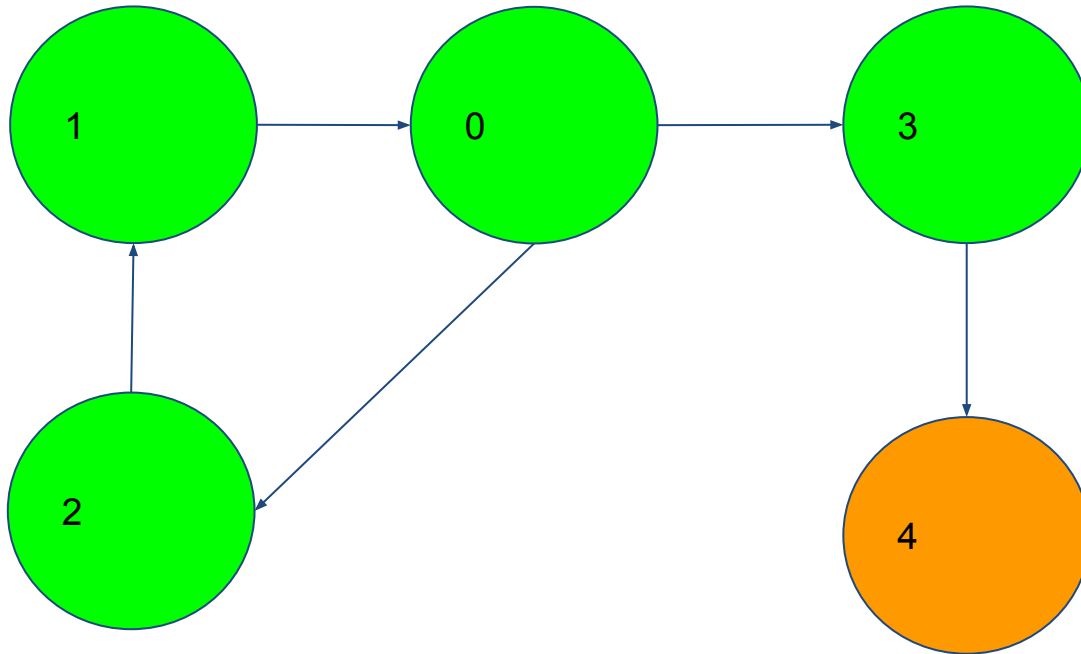
Grafos | Componentes Fuertemente Conexos



TS = {
2, 1
}

DFS(3) => DFS(4)

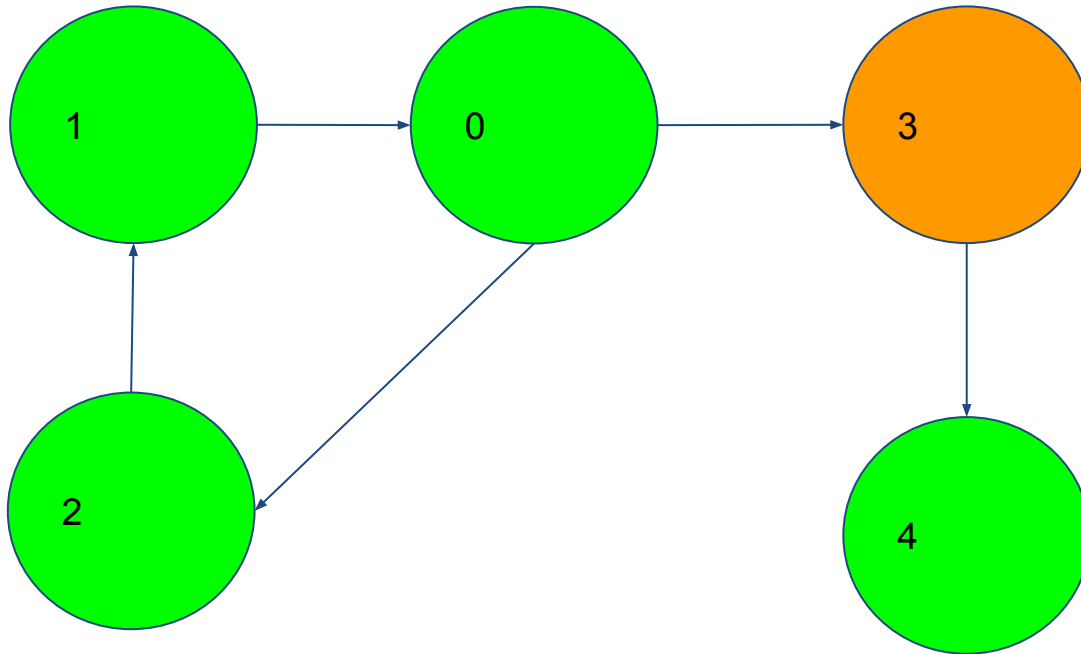
Grafos | Componentes Fuertemente Conexas



TS = {
4, 2, 1
}

DFS(4) => \emptyset , TS.push(4)

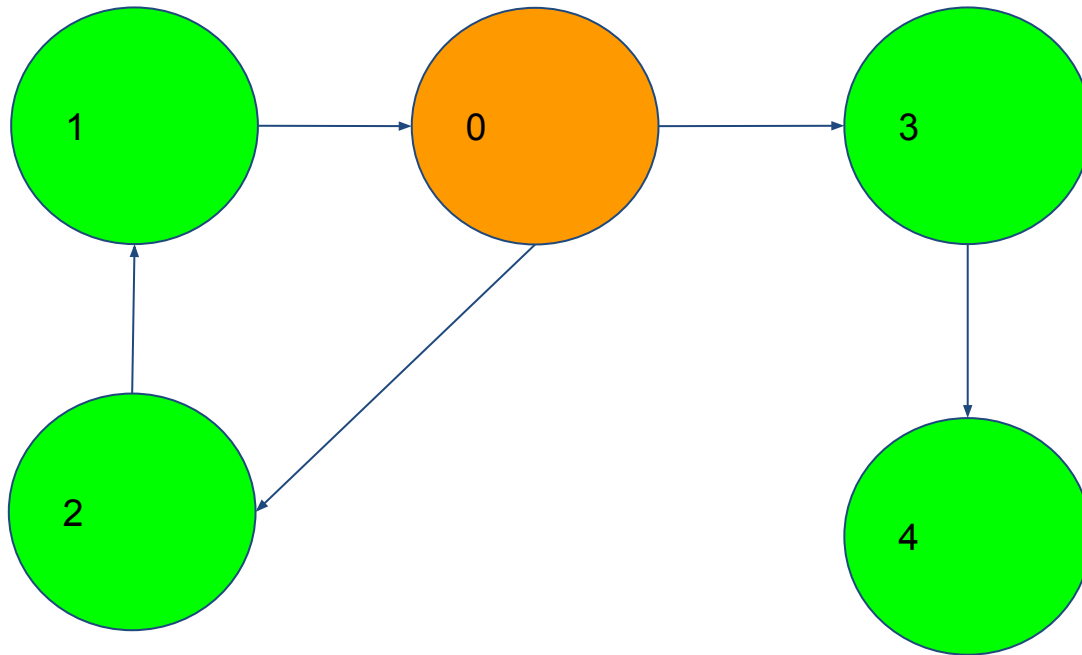
Grafos | Componentes Fuertemente Conexas



TS = {
3, 4, 2, 1
}

DFS(3) => DFS(4), TS.push(3)

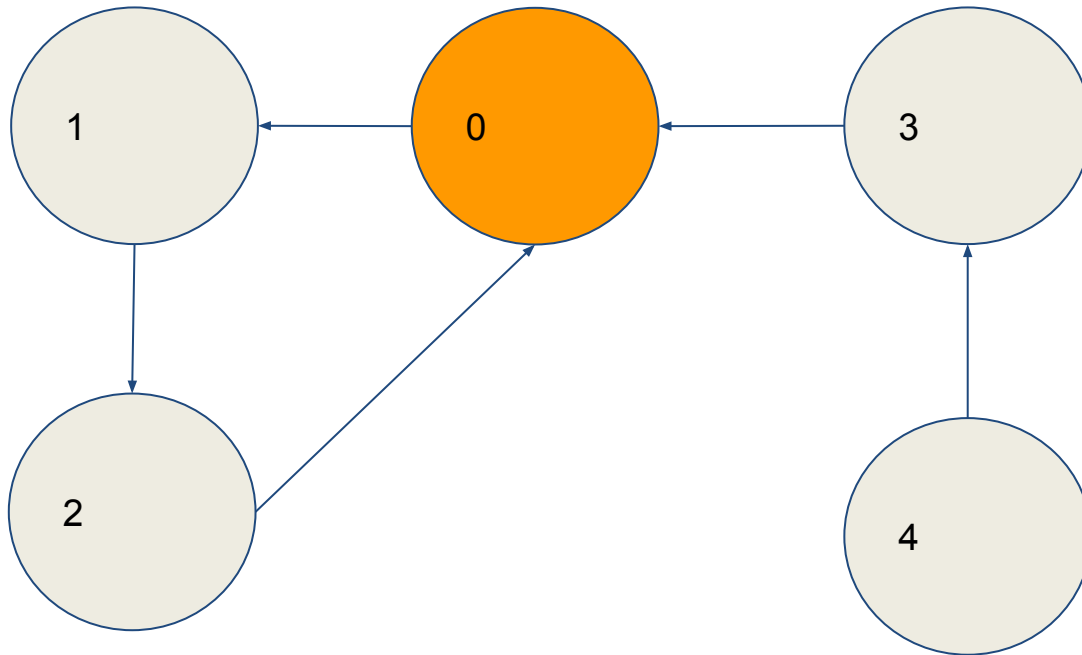
Grafos | Componentes Fuertemente Conexas



TS = {
0, 3, 4, 2, 1
}

DFS(0) => DFS(2), DFS(3), TS.push(0)

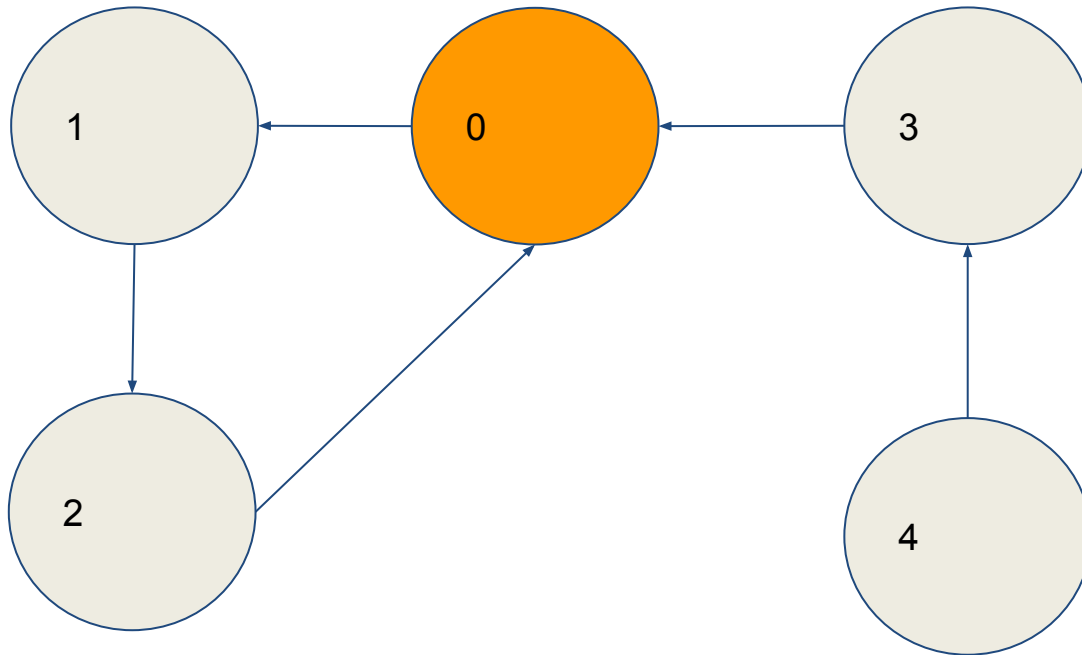
Grafos | Componentes Fuertemente Conexos



TS = {
0,3,4,2,1
}

Invertimos el grafo y seguimos un DFS en el orden del TS

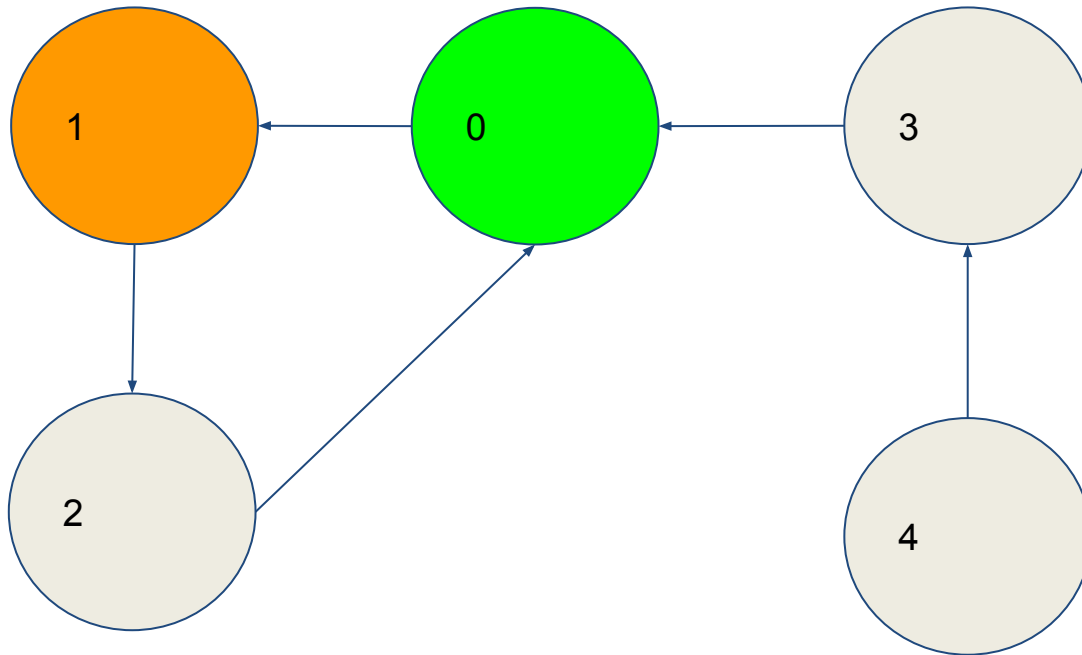
Grafos | Componentes Fuertemente Conexas



TS = {
0,3,4,2,1
}
SCC=1

DFS(0), SCC+1

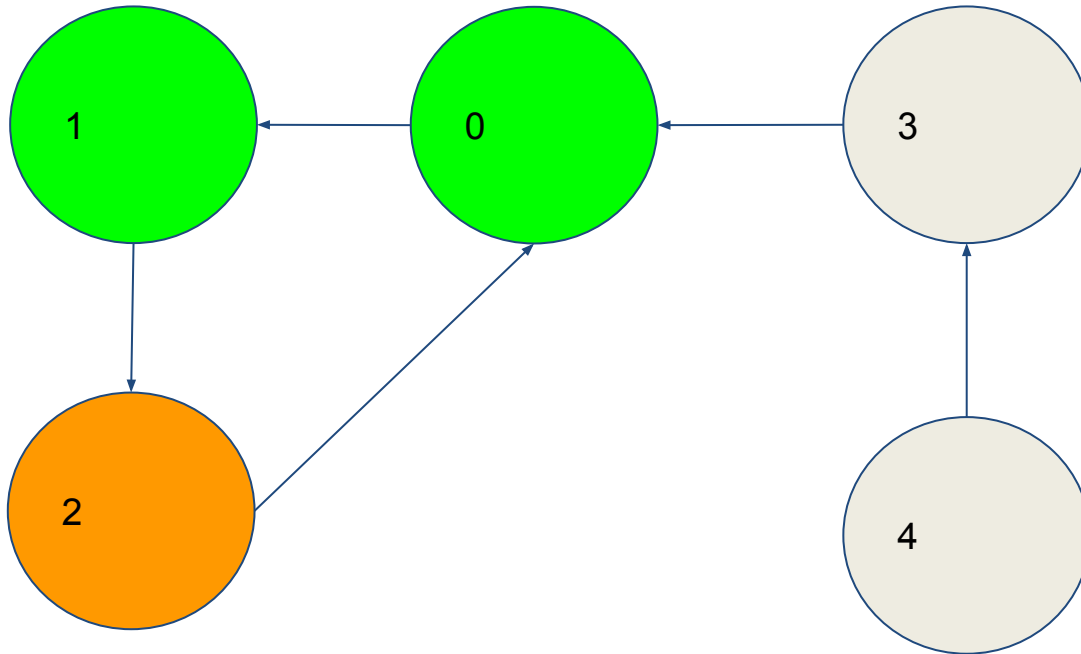
Grafos | Componentes Fuertemente Conexas



TS = {
0,3,4,2,1
}
SCC=1

DFS(1)

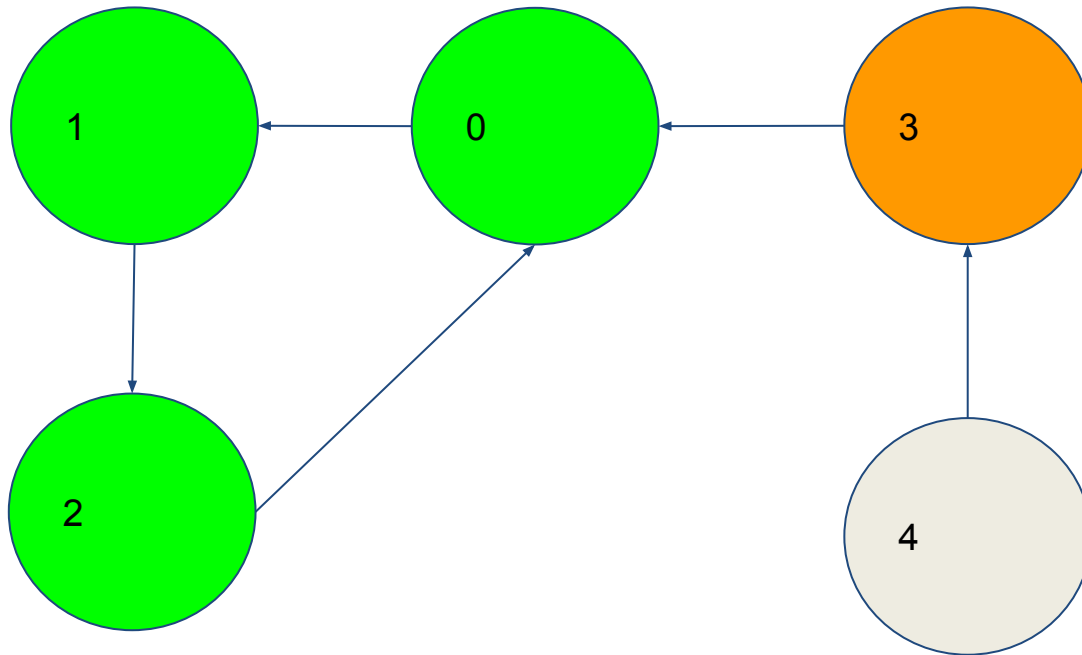
Grafos | Componentes Fuertemente Conexas



TS = {
0,3,4,2,1
}
SCC=1

DFS(2)

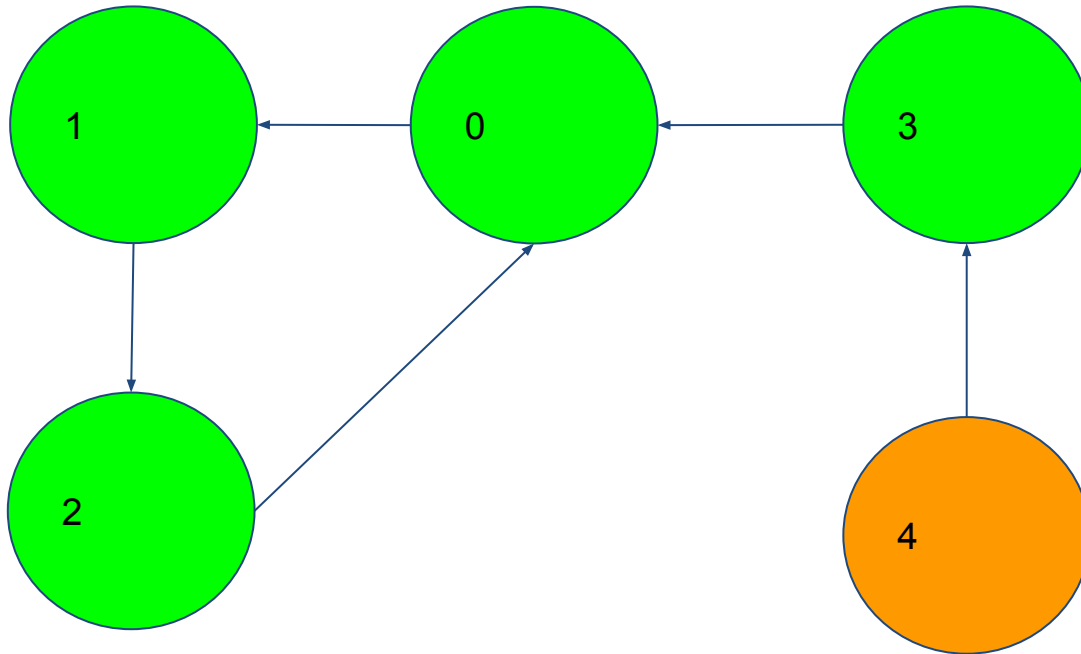
Grafos | Componentes Fuertemente Conexas



TS = {
0,3,4,2,1
}
SCC=2

3 no está verde, por tanto, DFS(3), SCC+1

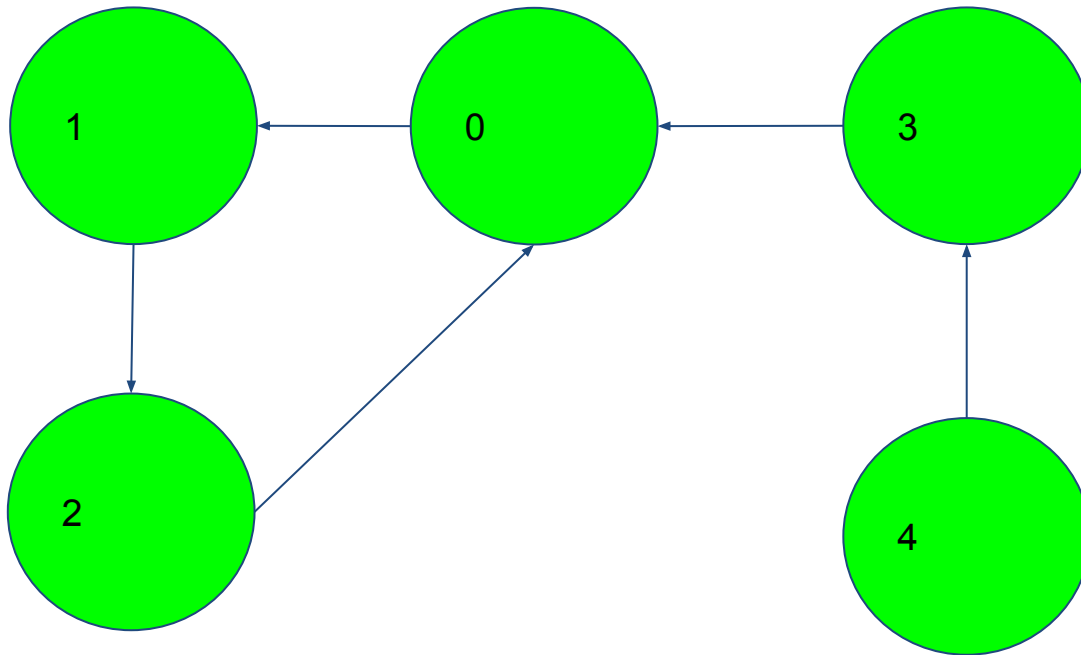
Grafos | Componentes Fuertemente Conexas



TS = {
0,3,4,2,1
}
SCC=3

4 no está verde, por tanto, DFS(4), SCC+1

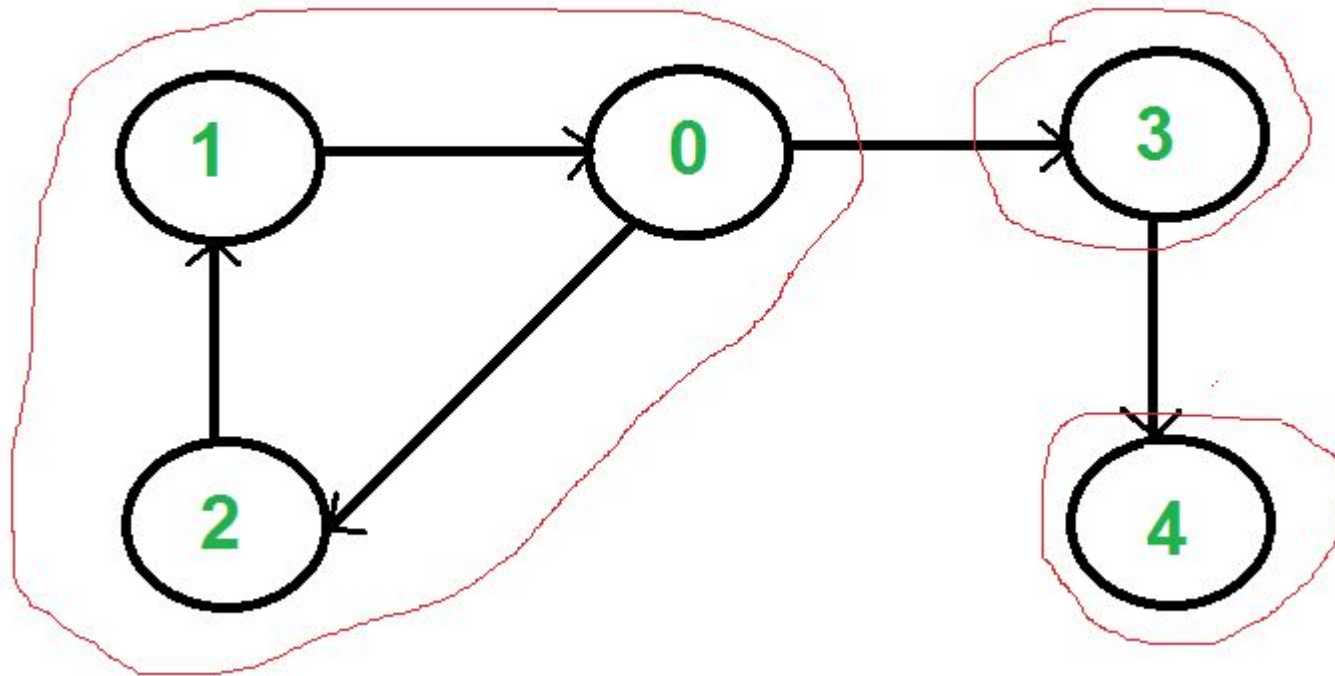
Grafos | Componentes Fuertemente Conexas



TS = {
0,3,4,2,1
}
SCC=3

1 y 2 están verdes, ya fueron tomados en cuenta, no hace falta visitar, SCC=3

Grafos | Componentes Fuertemente Conexas



Grafos | Componentes Fuertemente Conexas

- Para el ejemplo
 - Desde 0 \rightarrow Toposort = [1,2,4,3,0]
 - Gr[1] \Rightarrow [1,2,0] (+1)
 - Gr[2] \Rightarrow [] (+0)
 - Gr[3] \Rightarrow [0(x)] (+1)
 - Gr[4] \Rightarrow [3(x)] (+1)
- 3 componentes fuertemente conexas

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 copia a los tres)

David Morán (ddavidmorang@gmail.com)

Juan Quintana (juandavid.quintana@urjc.es)

Sergio Pérez (sergio.perez.pelo@urjc.es)

Jesús Sánchez-Oro (jesus.sanchezoro@urjc.es)